

# 第六讲 嵌入式Linux图形界面编程

华中科技大学电信学院

鄢舒

E-mail: [yan0shu@gmail.com](mailto:yan0shu@gmail.com)



# 内容提纲(1/6)

- 1. 嵌入式Linux图形界面编程基础
- 2. Linux的帧缓冲设备编程基础
- 3. Qt/Embedded开发基础
- 4. Qt/Embedded开发实例
- 5. 用Qt Designer设计图形界面
- 6. 实验内容与要求



# 1.1 嵌入式Linux图形界面编程概述

一般嵌入式Linux图形界面采用的编程方式：

- 直接操作Linux的帧缓冲设备；

- 系统开销小，但开发周期长。

- 采用成熟的图形库。

- 有成熟的组件，开发周期短，系统效率稍差，占用存储空间大。



## 1.2 常见的图形开发库

- Qt/Embedded
  - Microwindows
  - MiniGUI
- 还有其他。。。



## 1.3 Qt/Embedded简介

- Qt/Embedded(简称QtE)是一个专门为嵌入式系统设计图形用户界面的工具包。Qt是挪威Trolltech软件公司的产品，它为各种系统提供图形用户界面的工具包，QtE就是Qt的嵌入式版本。



## 1.4 QtE的授权

- QtE虽然公开代码和技术文档，但是它不是免费的，当开发者的商业化产品需要用到他的运行库时，必须向Trolltech公司支持license费用(每套3美金)，如果开发的东西不用于商业用途则不需要付费。
- QtE由于平台无关性和提供了很好的GUI编程接口，在许多嵌入式系统中得到了广泛的应用，是一个成功的嵌入式GUI产品。

## 1.5 Microwindows简介

- Microwindows是嵌入式系统中广为使用的一种图形用户接口，其官方网站是：<http://www.microwindows.org>。这个项目的早期目标是在嵌入式Linux平台上提供和普通个人电脑上类似的图形用户界面。
- 作为PC上XWindow的替代品，Microwindows提供了和XWindow类似的功能，但是占用的内存要少得多，根据用户得配置，Microwindows占用得内存资源在100KB-60KB。

## 1.6 Microwindows的特点

- Microwindows的核心基于显示设备接口，因此可移植性很好，Microwindows有自己的framebuffer，因此它并不局限于Linux开发平台，在eCos、FreeBSD、RTEMS等操作系统上都能很好地运行。
- 此外，Microwindows能在宿主机上仿真目标机。这意味着基于Linux的Microwindows应用程序的开发和调试可以在普通的个人电脑上进行，而不需要使用普通嵌入式软件的“宿主机—目标机”调试模式，从而大大加快了开发速度。
- Microwindows是完全免费的一个用户图形系统。





## 1.7 MiniGUI简介

- MiniGUI是由北京飞漫软件技术有限公司主持的一个自由软件项目(遵循GPL条款)，其目标是为基于Linux的实时嵌入式系统提供一个轻量级的图形用户界面支持系统。
- MiniGUI为应用程序定义了一组轻量级的窗口和图形设备接口。利用这些接口，每个应用程序可以建立多个窗口，而且可以在这些窗口中绘制图形。用户也可以利用MiniGUI建立菜单、按钮、列表框等常见的GUI元素。

## 1.8 MiniGUI的配置模式

- 用户可以将MiniGUI配置成“MiniGUI-Threads”或者“MiniGUI-Lite”。
- 运行在MiniGUI-Threads上的程序可以在不同的线程中建立多个窗口，但所有的窗口在一个进程中运行。
- 相反，运行在MiniGUI-Lite上的每个程序是单独的进程，每个进程也可以建立多个窗口。
- MiniGUI-Threads适合于具有单一功能的实时系统，而MiniGUI-Lite 则适合于类似于PDA和瘦客户机等嵌入式系统。



# 内容提纲(2/6)

- 1. 嵌入式Linux图形界面编程基础
- 2. Linux的帧缓冲设备编程基础
- 3. Qt/Embedded开发基础
- 4. Qt/Embedded开发实例
- 5. 用Qt Designer设计图形界面
- 6. 实验内容与要求

## 2.1 帧缓冲设备(Framebuffer)简介

- 帧缓冲（Framebuffer）是Linux为显示设备提供的一个接口，把显存抽象后的一种设备，他允许上层应用程序在图形模式下直接对显示缓冲区进行读写操作。这种操作是抽象的，统一的。用户不必关心物理显存的位置、换页机制等等具体细节。这些都是由Framebuffer设备驱动来完成的。
- 帧缓冲驱动的应用广泛，在Linux的桌面系统中，Xwindow服务器就是利用帧缓冲进行窗口的绘制，尤其是通过帧缓冲还可显示汉字点阵。

## 2.2 Framebuffer编程模型

- Linux Framebuffer 本质上只是提供了对图形设备的硬件抽象，在开发者看来，Framebuffer 是一块显示缓存，往显示缓存中写入特定格式的数据就意味着向屏幕输出内容。所以说Framebuffer就是一块白板。例如对于初始化为16 位色的Framebuffer 来说，Framebuffer中的两个字节代表屏幕上一个点，从上到下，从左至右，屏幕位置与内存地址是顺序的线性关系。
- 帧缓存可以在系统存储器(内存)的任意位置，视频控制器通过访问帧缓存来刷新屏幕。帧缓存也叫刷新缓存Framebuffer或refresh buffer, 这里的帧(frame)是指整个屏幕范围。
- 帧缓存有个地址，是在内存里。我们通过不停的向Framebuffer中写入数据，显示控制器就自动的从Framebuffer中取数据并显示出来。全部的图形都共享内存中同一个帧缓存。

## 2.3 Linux的Framebuffer设备

- 帧缓冲设备对应的设备文件为/dev/fb\*, 如果系统有多个显示卡, Linux下还可支持多个帧缓冲设备, 最多可达32个。帧缓冲设备为标准字符设备, 主设备号为29, 次设备号则从0到31。分别对应从/dev/fb0到/dev/fb31。
- /dev/fb为当前缺省的帧缓冲设备, 通常指向/dev/fb0。当然在嵌入式系统中支持一个显示设备就够了。
- 用#cat /dev/fb0 > screenshot将屏幕内存导入一个文件, 恢复刚才的屏幕截图则可使用: #cat screenshot > /dev/fb0。

## 2.4 对Framebuffer设备的操作

■ 通过/dev/fb，应用程序的操作主要有这几种：

1. 读/写（read/write）/dev/fb：相当于读/写屏幕缓冲区。例如用 `cp /dev/fb0 tmp` 命令可将当前屏幕的内容拷贝到一个文件中，而命令 `cp tmp > /dev/fb0` 则将图形文件tmp显示在屏幕上。
2. 映射（map）操作：由于Linux工作在保护模式，每个应用程序都有自己的虚拟地址空间，在应用程序中是不能直接访问物理缓冲区地址的。为此，Linux在文件操作file\_operations结构中提供了mmap函数，可将文件的内容映射到用户空间。对于帧缓冲设备，则可通过映射操作，可将屏幕缓冲区的物理地址映射到用户空间的一段虚拟地址中，之后用户就可以通过读写这段虚拟地址访问屏幕缓冲区，在屏幕上绘图了。实际上，使用帧缓冲设备的应用程序都是通过映射操作来显示图形的。由于映射操作都是由内核来完成，下面我们将看到，帧缓冲驱动留给开发人员的工作并不多。
3. I/O控制：对于帧缓冲设备，对设备文件的ioctl操作可读取/设置显示设备及屏幕的参数，如分辨率，显示颜色数，屏幕大小等等。ioctl的操作是由底层的驱动程序来完成的。





## 2.5 操作Framebuffer的步骤

1. 打开/dev/fb设备文件。
2. 用ioctl操作取得/改变当前显示屏幕的参数，如屏幕分辨率，每个像素点的比特数。根据屏幕参数可计算屏幕缓冲区的大小。
3. 将屏幕缓冲区映射到用户空间。
4. 映射后就可以直接读写屏幕缓冲区，进行绘图和图片显示了。



## 2.6 Framebuffer设备的信息结构(1/5)

- 从Framebuffer设备取回的信息，有两个结构包含着我们需要的信息。第一个包含固定的屏幕信息，这部分是由硬件和驱动的能力决定的；第二个包含着可变的屏幕信息，这部分是由硬件的当前状态决定的，可以由用户空间的程序调用*ioctl()*来改变。
- 分别是struct fb\_fix\_screeninfo和struct fb\_var\_screeninfo。

## 2.6 Framebuffer设备的信息结构(2/5)

```
struct fb_fix_screeninfo {  
    char id[16]; /* identification string eg "TT Builtin" */  
    unsigned long smem_start; /* Start of frame buffer mem(physical address) */  
    __u32 smem_len; /* Length of frame buffer mem */  
    __u32 type; /* see FB_TYPE_* */  
    __u32 type_aux; /* Interleave for interleaved Planes */  
    __u32 visual; /* see FB_VISUAL_* */  
    __u16 xpanstep; /* zero if no hardware panning */  
    __u16 ypanstep; /* zero if no hardware panning */  
    __u16 ywrapstep; /* zero if no hardware ywrap */  
    __u32 line_length; /* length of a line in bytes */  
    unsigned long mmio_start; /* Start of Memory Mapped I/O(physical address) */  
    __u32 mmio_len; /* Length of Memory Mapped I/O */  
    __u32 accel; /* Type of acceleration available */  
    __u16 reserved[3]; /* Reserved for future compatibility */  
};
```

- 在这里非常重要的域是smem\_len和line-length。smem-len告诉我们framebuffer设备的大小，第二个域告诉我们指针应该前进多少字节去得到下一行的数据。

## 2.6 Framebuffer设备的信息结构(3/5)

```
struct fb_var_screeninfo {
    __u32 xres; /* visible resolution */
    __u32 yres;
    __u32 xres_virtual; /* virtual resolution */
    __u32 yres_virtual;
    __u32 xoffset; /* offset from virtual to visible resolution */
    __u32 yoffset;
    __u32 bits_per_pixel; /* guess what */
    __u32 grayscale; /* != 0 Graylevels instead of colors */
    struct fb_bitfield red; /* bitfield in fb mem if true color, */
    struct fb_bitfield green; /* else only length is significant */
    struct fb_bitfield blue;
    struct fb_bitfield transp; /* transparency */
    __u32 nonstd; /* != 0 Non standard pixel format */
    __u32 activate; /* see FB_ACTIVATE_* */
    __u32 height; /* height of picture in mm */
    __u32 width; /* width of picture in mm */
    __u32 accel_flags; /* acceleration flags (hints) */
    /* Timing: All values in pixclocks, except pixclock (of course) */
    __u32 pixclock; /* pixel clock in ps (pico seconds) */
    __u32 left_margin; /* time from sync to picture */
    __u32 right_margin; /* time from picture to sync */
    __u32 upper_margin; /* time from sync to picture */
    __u32 lower_margin;
    __u32 hsync_len; /* length of horizontal sync */
    __u32 vsync_len; /* length of vertical sync */
    __u32 sync; /* see FB_SYNC_* */
    __u32 vmode; /* see FB_VMODE_* */
    __u32 reserved[6]; /* Reserved for future compatibility */
};

struct fb_bitfield {
    __u32 offset; /* beginning of bitfield */
    __u32 length; /* length of bitfield */
    __u32 msb_right; /* != 0 Most significant bit is right */
};
```

## 2.6 Framebuffer设备的信息结构(4/5)

- 第二个结构的前几个成员决定了分辨率。xres和yres是在屏幕上可见的实际分辨率，在通常的vga模式将为640和480。\*res-virtual决定了构建屏幕时视频卡读取屏幕内存的方式。例如当实际的垂直分辨率为400，虚拟分辨率可以是800。这意味着800行的数据被保存在了屏幕内存区中。因为只有400行可以被显示，决定从那一行开始显示就是你的事了。这个可以通过设置\*offset来实现。给yoffset赋0将显示前400行，赋35将显示第36行到第435行，如此重复。这个功能在许多情形下非常方便实用。
- 它可以用来做双缓冲。双缓冲就是你的程序分配了可以填充两个屏幕的内存。将offset设为0，将显示前400行，同时可以秘密的在400行到799行构建另一个屏幕，当构建结束时，将yoffset设为400，新的屏幕将立刻显示出来。现在将开始在第一块内存区中构建下一个屏幕的数据，如此继续。这在动画中十分有用。

## 2.6 Framebuffer设备的信息结构(5/5)

- 另外一个应用就是用来平滑的滚动整个屏幕。就像在前面屏幕中一样，在内存分配800行的空间。每隔10毫秒设定一个定时器(timer)，将offset设为1或是更多，你将看到一个平滑滚动的屏幕。
- 将bits\_per\_pixel 设为1, 2, 4, 8, 16, 24或32来改变颜色深度（color depth）。不是所有的视频卡和驱动都支持全部颜色深度。当颜色深度改变，驱动将自动改变fb-bitfields。它将决定在一个特定的颜色基准上，多少和哪些比特被哪种颜色使用。如果bits-per-pixel小于8，则fb-bitfields将无定义而且颜色映射将启用。
- 在fb-var-screeninfo结构结尾的定时的设置是当你选择一个新的分辨率的时候用来设定视频定时的。

## 2.7 操作Framebuffer的常用函数

```
int framebuffer_handler;  
struct fb_fix_screeninfo fixed_info;  
struct fb_var_screeninfo variable_info;  
open ("/dev/fb0", O_RDWR); /*in real life, check every ioctl if it returns -1 */  
ioctl (framebuffer_handler, FBIOGET_VSCREENINFO, &variable_info);  
variable_info.bits_per_pixel = 32;  
Ioctl (framebuffer_handler, FBIOPUT_VSCREENINFO, &variable_info);  
ioctl (framebuffer_handler, FBIOGET_FSCREENINFO, &fixed_info);  
variable_info.yoffset = 513;  
ioctl (framebuffer_handler, FBIOPAN_DISPLAY, &variable_info);
```

- 这些 FBIOGET\_\*的ioctl命令将请求的信息写入最后一个变量所指向的结构体中。FBIOPUT\_VSCREENINFO将所有提供的信息复制回内核。如果内核不能激活新的设置，将返回-1。而FBIOPAN\_DISPLAY也从用户复制信息，但并不重新初始化视频模式。最好在只有xoffset或yoffset改变时使用。

## 2.8 操作Framebuffer的关键代码(1/2)

```
#include <linux/fb.h>
int main()
{
    int fbfd = 0;
    struct fb_var_screeninfo vinfo;
    struct fb_fix_screeninfo finfo;
    long int screensize = 0;
    /*打开设备文件*/
    fbfd = open("/dev/fb0", O_RDWR);
    /*取得屏幕相关参数*/
    ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo);
    ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo);
    /*计算屏幕缓冲区大小*/
    screensize = vinfo.xres * vinfo.yres * vinfo.bits_per_pixel / 8;
    /*映射屏幕缓冲区到用户地址空间*/
    fbp=(char*)mmap(0,screensize,PROT_READ|PROT_WRITE,MAP_SHARED, fbfd, 0);
    /*下面可通过fbp指针读写缓冲区*/
    . . .
```



## 2.8 操作Framebuffer的关键代码(2/2)

- 在这一部分新用到的是mmap函数。第一个变量在这种情形下可以忽略，第二个是映射的内存大小，第三个变量声明我们将共享内存进行读和写。第四个变量表示这段内存将和其他进程共享。在Framebuffer上面建一个MAP\_PRIVATE是不可能的。通常这意味着你需要中断控制台的切换去备份和恢复屏幕内容，而且不在自己没有权利的时候向屏幕内存写东西。



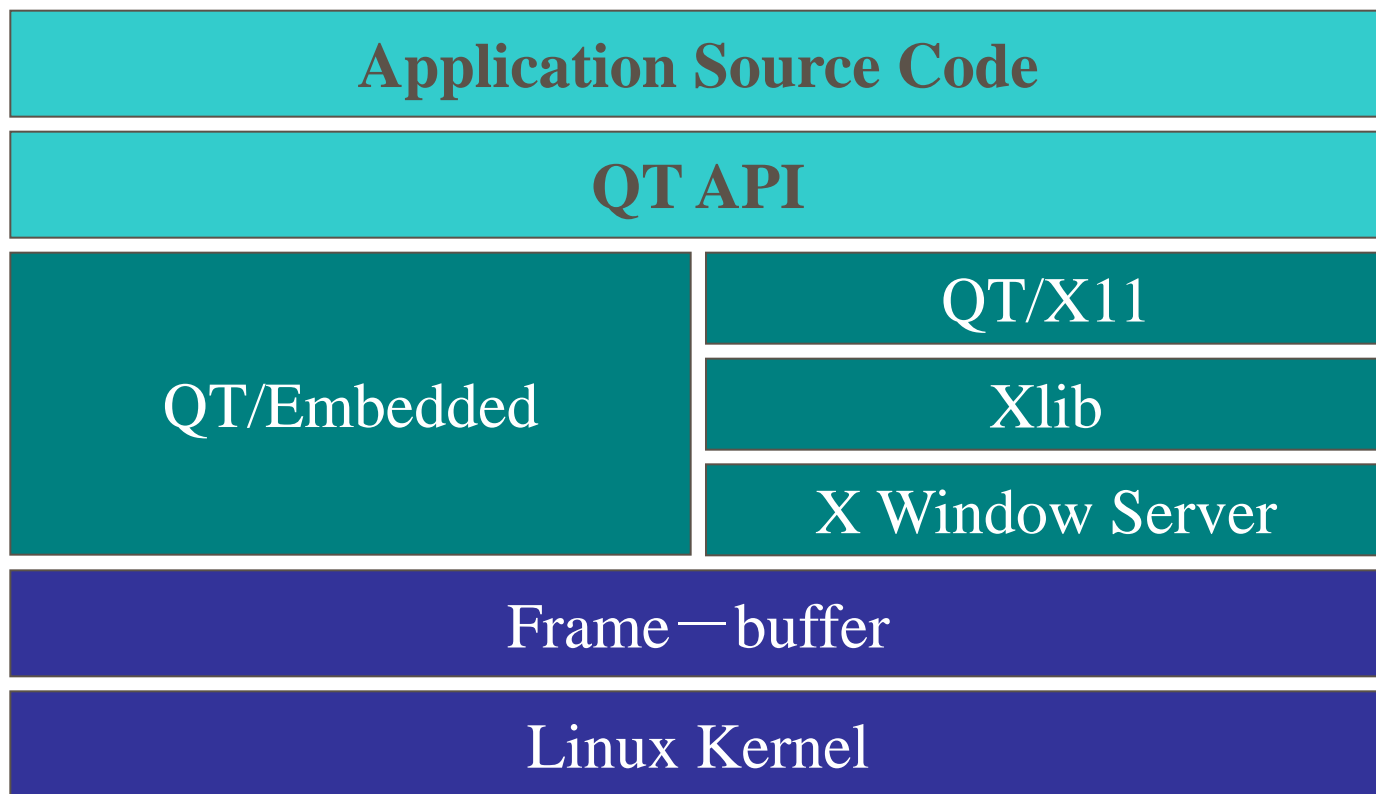


# 内容提纲(3/6)

- 1. 嵌入式Linux图形界面编程基础
- 2. Linux的帧缓冲设备编程基础
- 3. Qt/Embedded开发基础
- 4. Qt/Embedded开发实例
- 5. 用Qt Designer设计图形界面
- 6. 实验内容与要求



## 3.1 QtE与Qt/X11的关系



## 3.2 QtE与Qt/X11的关系

- Qt/Embedded通过Qt API 与Linux I/O设施直接交互，成为嵌入式Linux端口。同Qt/X11相比， Qt/Embedded很节省内存，其不需要一个X服务器或是Xlib库，它在底层摒弃了Xlib，采用framebuffer（帧缓存）作为底层图形接口。
- 同时，将外部输入设备抽象为keyboard和mouse输入事件。 Qt/Embedded 的应用程序可以直接写内核缓冲帧，这可避免开发者使用繁琐的Xlib/Server系统。



### 3.3 Qtopia与QtE的关系

- Trolltech公司在QtE的基础上开发了一个应用的环境—Qtopia，这个应用环境为移动和手持设备开发。其特点就是拥有完全的、美观的GUI，同时它也提供可上百个应用程序用于管理用户信息、办公、娱乐、Internet交流等。
- 已经有很多公司采用了Qtopia来开发他们主流的PDA。



## 3.4 Qtopia应用平台

- Qtopia是一种全方位的应用开发平台，它可用于基于嵌入式Linux的PDA，移动电话，web pads，以及其他移动计算设备。
- Qtopia构建于Qt/Embedded之上，是专为基于Linux的消费电子产品提供和创建图形用户界面而设计的。常见的有两种版本：
  - Qtopia Phone版：专为基于Linux的智能电话和多功能电话设计。
  - Qtopia PDA版：专为基于Linux的PDA设计。



## 3.5 Qtopia的特色

Qtopia包含以下组件：

- 视窗操作系统；同步窗口；开发环境；本地化支持；游戏和多媒体；PIM应用程序；输入法；个性化选项；Internet应用程序；Java集成；无线支持。

## 3.6 Qt/Embedded开发模型

- 嵌入式软件开发通常都采用交叉编译的方式进行，基于Qt/Embedded和Qttopia的GUI应用开发也采用这样的模式。先在宿主机上调试应用程序，调试通过后，经过交叉编译移植到目标板上。
- Qt/Embedded直接写入帧缓存，在宿主机上则是通过qvfb（virtual framebuffer）来模拟帧缓存。qvfb是X窗口用来运行和测试Qttopia应用程序的系统程序。qvfb使用了共享内存存储区域（虚拟的帧缓存）来模拟帧缓存并且在一个窗口中模拟一个应用程序来显示帧缓存，显示的区域被周期性的改变和更新。



## 3.7 Qt/Embedded开发工具

宿主机移植所需工具及环境变量声明：

工具软件	描述	变量声明
Tmake—1.11	生成Makefile文件	TMAKEDIR/TMAKEPATH /PATH
Qt-x11-2.3.2	Qvfb—虚拟帧缓存工具 Uic—用户界面编辑器 Designer Qt 图形设计器	LD_LIBRARY_PATH/PATH
Qt-embedded-2.3.7	Qt库支持 libqte.so	QTEDIR/LD_LIBRARY_PATH/ PATH
Qtopia-free-1.7.0	应用程序开发包桌面环境	QPEDIR/LD_LIBRARY_PATH/ PATH



## 3.8 Qt/Embedded的信号与插槽

- 信号与插槽是Qt自定义的一种通信机制，它独立于标准的C/C++语言。他的实现必须借助于moc（Meta Object Compiler）的Qt工具，他是一个C++预处理程序，为高层次的事件处理自动生成所需要的附件代码。
- 所谓图形用户接口的应用就是对用户的动作作出响应。程序员则必须把事件和相关代码联系起来，这样才能对事件作出正确的响应。



## 3.9 信号和插槽的作用

- 所有从QObject或其子类（例如QWidget）派生的类都能够包含信号和插槽。
- 当对象改变状态时，信号就由该对象发射（emit）出来。
- 插槽用于接收信号，但它们是普通的对象成员函数。
- 一个插槽并不知道是否有任何消息与自己相连。用户可以将很多信号与一个插槽相连，也可将单个消息与多个插槽进行链接。

## 3.10 信号与插槽的实现

信号与插槽的实现方式如下：

■ 信号：

```
void mysignal(int x);
```

■ 插槽：

```
void myslot(int x);
```

■ 信号与插槽关联：

```
connect(abutton, SIGNAL(clicked),SLOT(quit));
```

## 3.11 建立Qt/Embedded开发环境

### 1. 安装x86-qtopia源代码

- `# tar xvzf x86-qtopia.tgz -C /smdk2410`
- `# cd /smdk2410/x86-qtopia`
- `#./build` (shell 程序)

### 2. 安装SMDK-2410X版本的arm-qtopia源代码

- `#tar xvzf arm-qtopia.tgz -C /smdk2410`
- `# cd /smdk2410/arm-qtopia`
- `#./build` (shell 程序)

## 3.12 设置链接库

- 当在PC上模拟Qtopia的运行时，需要用到对应Qt版本的库文件，因此需要修改/etc/ld.so.conf 文件以适应刚刚安装的Qt(Linux安装时带有Qt库，但不适合我们最新安装的版本)，修改后的ld.so.conf文件内容如下：
  - /smdk2410/x86-qtopia/qt/lib
  - /smdk2410/x86-qtopia/qtopia/lib
  - /usr/kerberos/lib
  - /usr/X11R6/lib
  - /usr/lib/sane
- 修改完此文档后，为了让刚刚安装的库生效，必须运行ldconfig。至此Qt的开发环境已经建立。



## 3.13 运行Qtopia

### (1) 设置环境变量

在/smdk2410/x86-qtopia/目录下输入“source set-env”或“`. set-env`”。

### (2) 启动虚拟帧缓存

“`$ qvfb &`”

或 “`$ qvfb -width 640 -height 480 &`”

### (3) 运行qtopia

`qpe &`



# 内容提纲(4/6)

- 1. 嵌入式Linux图形界面编程基础
- 2. Linux的帧缓冲设备编程基础
- 3. Qt/Embedded开发基础
- 4. Qt/Embedded开发实例
- 5. 用Qt Designer设计图形界面
- 6. 实验内容与要求

## 4.1 Qt编程初始化

- 在Qt应用程序中，首先要创建一个QApplication对象，QApplication类负责图形用户界面应用程序的控制流和主设置，在main.cpp中定义如下：

```
int main(int argc, char **argv) {  
    QApplication a(argc, argv);  
    .....  
}
```



## 4.2 QApplication类

- QApplication包含在main()函数的事件循环体中，对所有来自Window系统和其它源文件的事件进行处理和调度，还包括处理应用程序的初始化和结束，并且提供会话管理。
- 在Qt应用程序中，不管有多少个窗口，QApplication对象只能有一个，而且必须在其他对象之前创建。

QApplication类中封装了很多函数，其中包括：

- 系统设置：setFont() 用来设置字体
- 事件处理：sendEvent() 用来发送事件
- GUI风格：setStyles() 设置图形用户界面的风格
- 颜色使用：colorSpec() 用来返回颜色文件
- 文本处理：translate() 用来处理文本信息
- 创建组件：setMainWidget() 用来设置窗口的主组件

.....

## 4.3 窗口的创建

- 在Qt程序中，创建窗口比较简单，只要在main.cpp文件中为ApplicationWindow建立一个指针：  
`ApplicationWindow *mw = new ApplicationWindow();`
- ApplicationWindow是在Application.h中定义类，它是一个QmainWindow的继承类。

## 4.4 组件的创建(1/2)

- 组件的创建需要调用相应组件的类，并在头文件中包含此类的头文件或者创建自定义类，继承以后组件类的功能，如：

```
#include "qpushbutton.h"
```

```
class hello::public QWidget
```

```
{
```

```
.....
```

```
}
```

## 4.4 组件的创建(2/2)

- hello类继承了QWidget类的特征，并加入了自定义的特征功能，同样需要在头文件中包含此类的头文件
- 在main.cpp的函数中需要创建hello类的实例，或创建QPushButton类的实例，才可以使用

```
hello h(string);
```

```
QPushButton hello("Hello,world!",0);
```

- 如果组件本身可以作为主窗口，则无需设置主窗口。在上例中，下压按钮创建时其构造函数中的第二个参数为0，表示按钮所在窗口为主窗口，不需要设置主窗口。否则需要调用QWidget成员函数setMainWidget()来进行设置

```
h.setMainWidget(&h);
```

- 组件创建时一般是不可见的，这样的好处在于避免大量组件创建时造成的屏幕闪烁现象，要使组件可见需要调用QWidget类的成员函数show()来显示组件

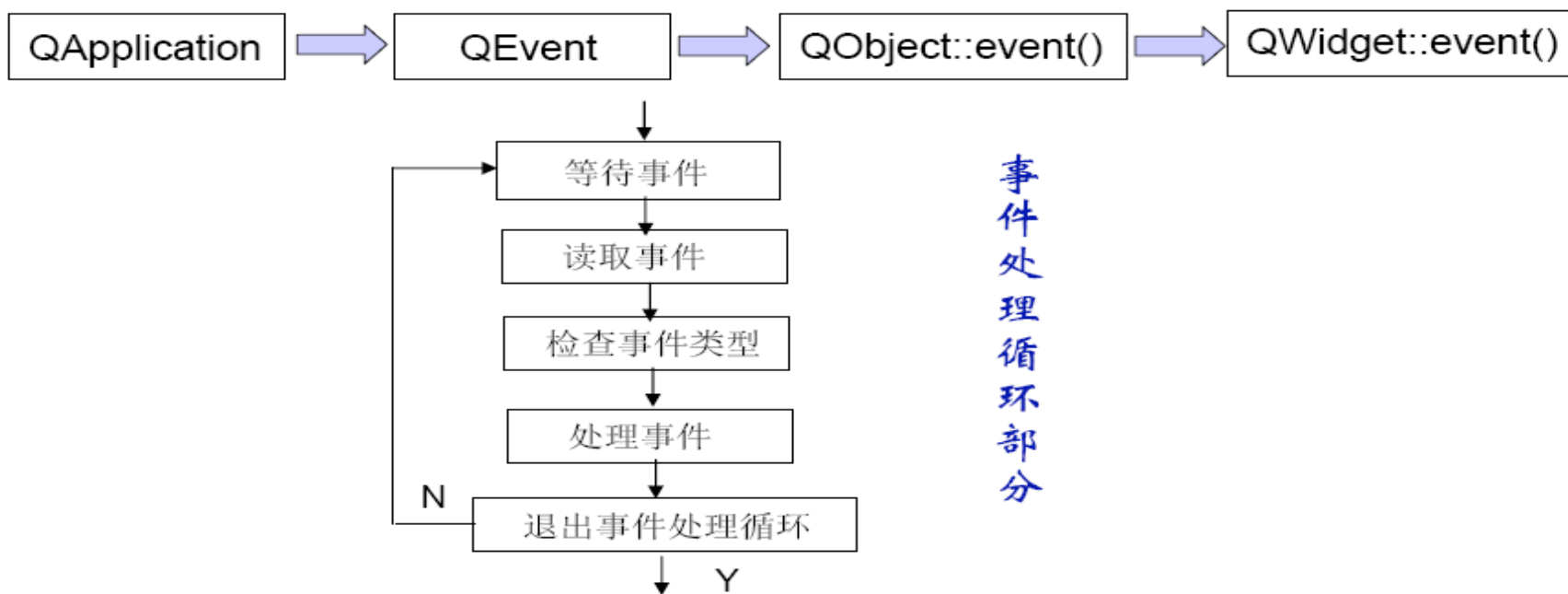
```
h.show();
```

## 4.5 事件(1/2)

- 在X程序中，敲击键盘，鼠标指针在窗口中的移动或鼠标按键动作等，都是事件。
- 在Qt中提供了一种叫做回调的事件处理方式。它通过翻译表，将事件映射为相应的动作，当组件得到事件通知，就去表中找出相应的动作例程进行处理。这种机制需要应用程序注册有关组件的回调函数或普通的事件处理函数，以分发循环Qt的事件。
- Qt事件的处理过程：QApplication的事件循环体从事件队列中拾取本地窗口系统事件或其他事件，译成QEvent()，并送给QObject::event()，最后送给QWidget::event()分别对事件处理。
- 其实在Qt程序中，事件处理的方式也是回调，但与以往所不同的是，事件的发出和接收采用了信号（signal）和插槽（slot）机制，无须调用翻译表。利用信号和插槽进行对象间的通信是Qt的最主要特征之一。

## 4.5 事件(2/2)

- 当对象状态发生改变的时候，发出signal通知所有的slot接收signal，尽管它并不知道哪些函数定义了slot，而slot也同样不知道要接收怎样的signal。
- signal和slot机制真正实现了封装的概念，slot除了接收signal之外和其它的成员函数没有什么不同，而且signal和slot之间也不是一一对应。



## 4.6 signal和slot的声明(1/2)

- 在Qt程序设计中，凡是包含signal和slot的类中都要加上Q\_OBJECT的定义，下面的例子给出了如何在一个类中定义signal和slot:

```
class Student : public QObject
{
    Q_OBJECT
public:
    Student() { myMark = 0; }
    int mark() const { return myMark; }
    public slots:
    void setMark(int newMark);
    signals:
    void markChanged(int newMark);
private:
    int myMark;
};
```

## 4.6 signal和slot的声明(2/2)

- signal的发出一般在事件的处理函数中，利用emit发出signal，在下面的例子中在在事件处理结束后发出signal:

```
void Student::setMark(int newMark)
{
    if (newMark != myMark) {
        myMark = newMark;
        emit markChanged(myMark);
    }
}
```



## 4.7 signal和slot的连接(1/2)

- 在signal和slot声明以后，需要使用connect()函数将它们连接起来。connect()函数属于QObject类的成员函数，它能够连接signal和slot，也可以用来连接signal和signal。

- 函数原形如下：

```
bool connect(const QObject *sender,const char *signal,const QObject *receiver,const char *member)
```

- 其中第一个和第三个参数分别指出signal和slot是属于那个对象或组件。

## 4.7 signal和slot的连接(2/2)

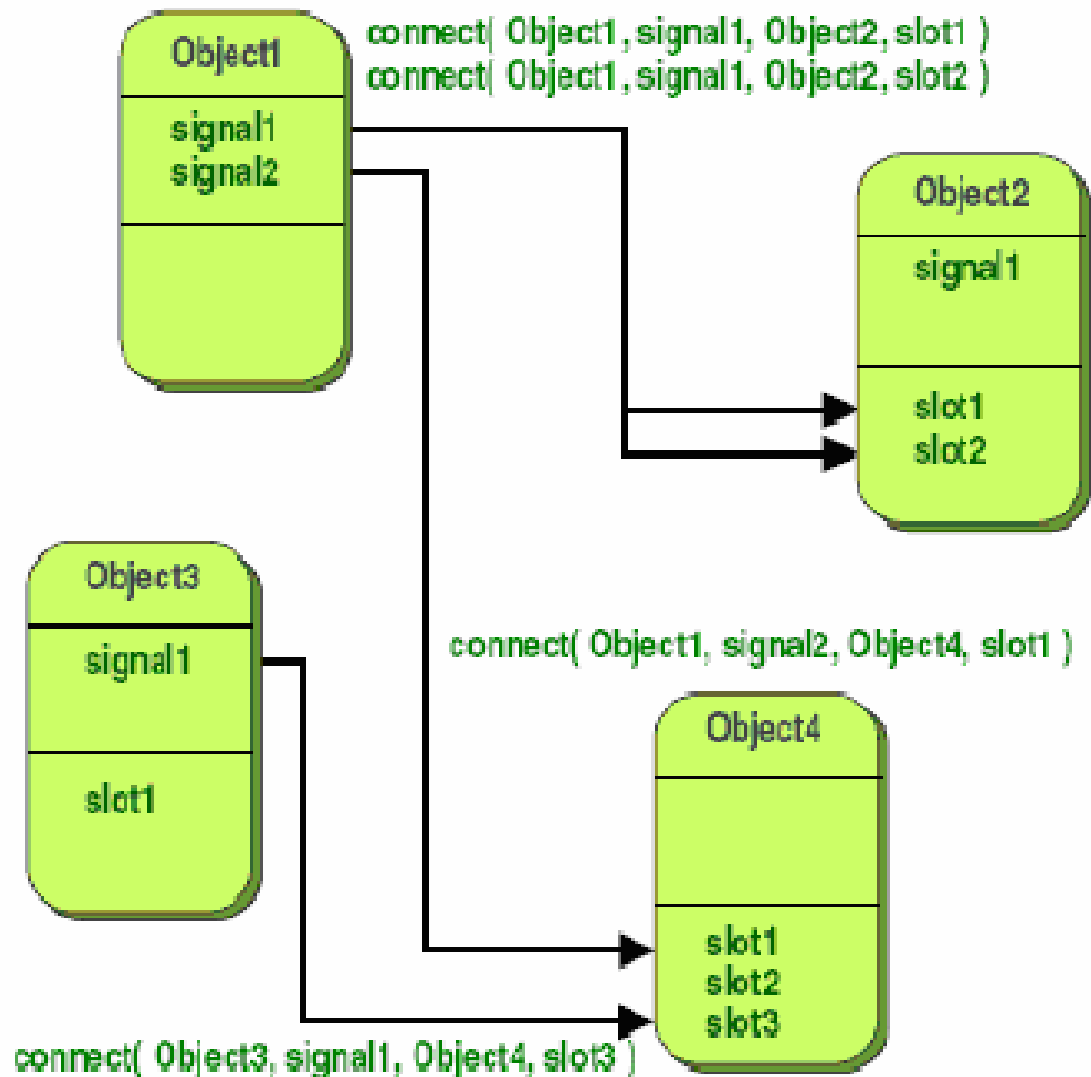
- 在使用connect()函数进行来连接的时候，还需要用到SIGNAL()和SLOT()这两个宏，使用方法如下：

```
QLabel *label = new QLabel;
```

```
QScrollBar *scroll = new QScrollBar;
```

```
QObject::connect (scroll, SIGNAL(valueChanged  
(int)), label, SLOT(setNum(int)) );
```

## 4.8 signal和slot的连接方式(1/3)



## 4.8 signal和slot的连接方式(2/3)

- 同一个信号连接多个插槽

```
connect( slider, SIGNAL(valueChanged(int)), spinBox,  
        SLOT(setValue(int)) );
```

```
connect( slider, SIGNAL(valueChanged(int)), this,  
        SLOT(updateStatusBarIndicator(int)) );
```

- 多个信号连接到同一个插槽

```
connect( lcd, SIGNAL(overflow()), this,  
        SLOT(handleMathError()) );
```

```
connect( calculator, SIGNAL(divisionByZero()), this,  
        SLOT(handleMathError()) );
```

## 4.8 signal和slot的连接方式(3/3)

- 一个信号连接到另一个信号

```
connect( lineEdit, SIGNAL(textChanged(const QString &)), this,  
        SIGNAL(updateRecord(const QString &)) );
```

- 取消一个连接

```
disconnect( lcd, SIGNAL(overflow()), this,  
           SLOT(handleMathError()) );
```

- 取消一个连接不是很常用，因为Qt会在一个对象被删除后自动取消这个对象所包含的所有的连接。

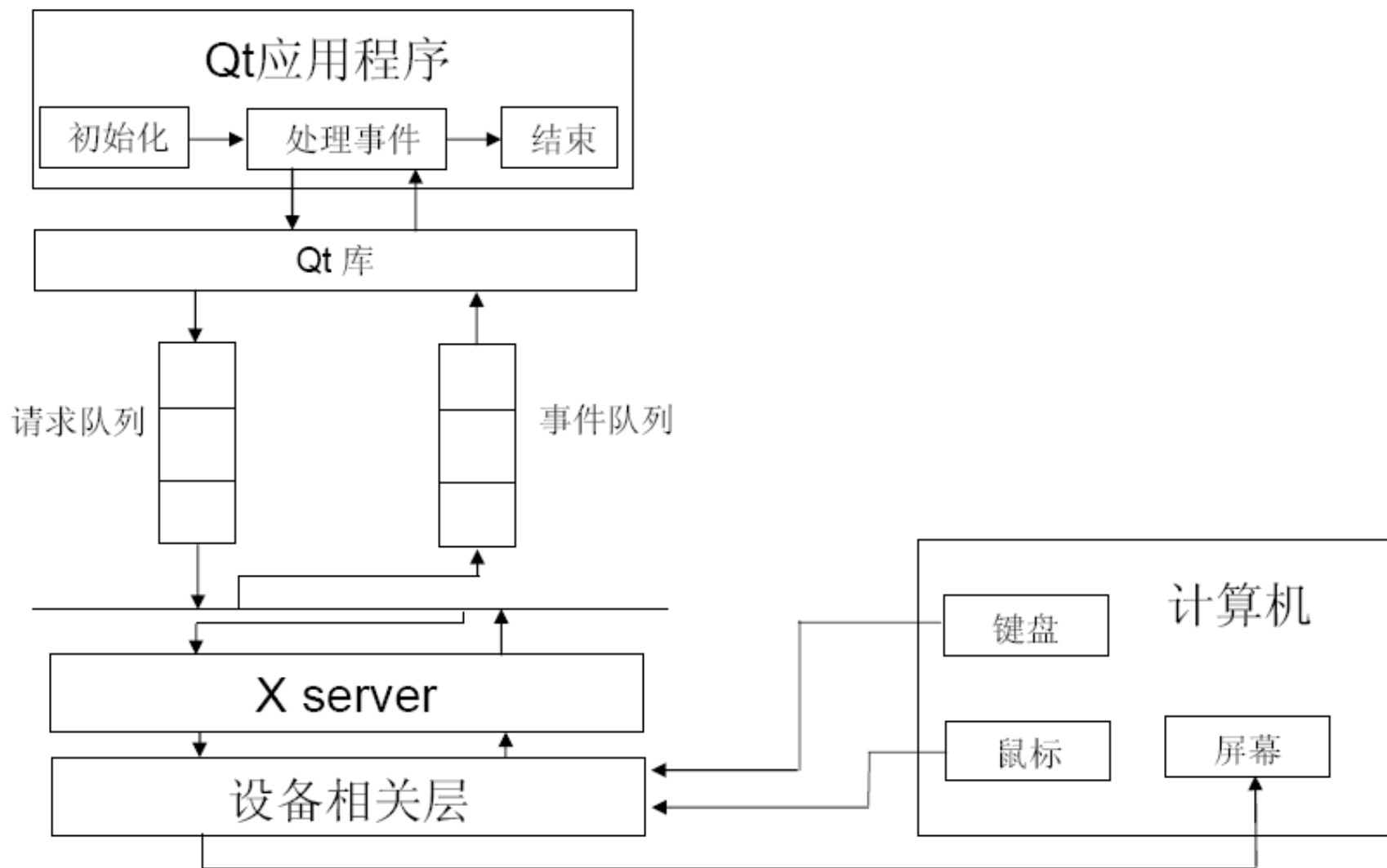
## 4.9 退出事件程序

- 退出事件程序，只需要在程序结束时返回一个exec()，例如：

```
return a.exec();
```

- 其中a为QApplication的实例，当调用exec()将进入主事件的循环中，直到exit()被调用或主窗口部件被销毁。

## 4.10 整个Qt程序的执行过程





## 4.11 一个“Hello Embedded”Qt程序

```
#include <qapplication.h>
#include <qlabel.h>

int main(int argc, char **argv)
{
    QApplication app(argc,argv);
    QLabel *hello = new QLabel("hello Qt/Embedded!",0);
    app.setMainWidget(hello);
    hello->show();
    return app.exec();
}
```



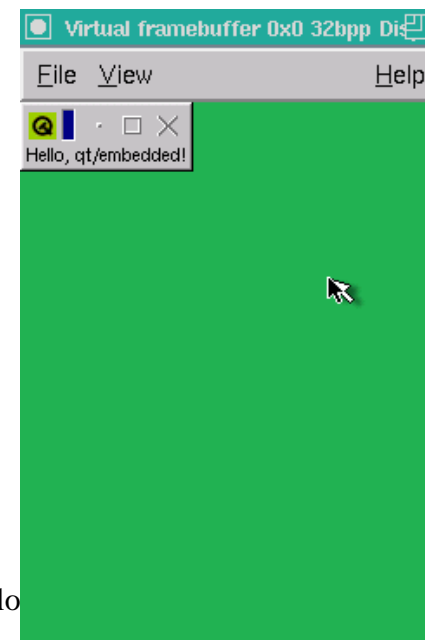


## 4.12 对hello.cpp程序的分析

- 程序第1行和第2行包含了两个头文件，这两个头文件中包含了QApplication 和 QLabel类的定义。
- 第5行创建了一个QApplication 对象，用于管理整个程序的资源，它需要2个参数，因为Qt 本身需要一些命令行的参数。
- 第6行创建了一个用来显示Hello Qt/Embedded!的部件。在Qt 中，部件是一个可视化用户接口，按钮、菜单、滚动条都是部件的实例。部件可以包含其它部件，例如，一个应用程序窗口通常是一个包含QMenuBar、QToolBar、QStatusBar 和其它部件的一个部件。在QLabel 函数中的参数0 表示，这是一个窗口而不是嵌入到其它窗口中的部件。
- 第7行设置hello 部件为程序的主部件，当用户关闭主部件后，应用程序将会被关闭。如果没有主部件的话，即使用户关闭了窗口程序也会在后台继续运行。
- 第8行使hello 部件可视，一般来说部件被创建后都是被隐藏的，因此可以在显示前根据需要来订制部件，这样的好处是可以避免部件创建所造成的闪烁。
- 第9行把程序的控制权交还给Qt，这时候程序进入就绪模式，可是随时被用户行为激活，例如点击鼠标、敲击键盘等。

## 4.13 程序的编译和运行

- 在配置好环境变量后，要在qvfb中运行程序需要如下的步骤
  1. 生成工程文件（.pro）
    - [root@localhost ~]\$ progen -t app.t -o hello.pro
  2. 生成Makefile文件
    - [root@localhost ~]\$ tmake -o Makefile hello.pro
  3. 编译
    - [root@localhost ~]\$ make
  4. 运行(确保qvfb 在后台运行)
    - [root@localhost ~]\$ ./hello -qws





# 内容提纲(5/6)

- 1. 嵌入式Linux图形界面编程基础
- 2. Linux的帧缓冲设备编程基础
- 3. Qt/Embedded开发基础
- 4. Qt/Embedded开发实例
- 5. 用Qt Designer设计图形界面
- 6. 实验内容与要求



## 5.1 使用Qt Designer的步骤

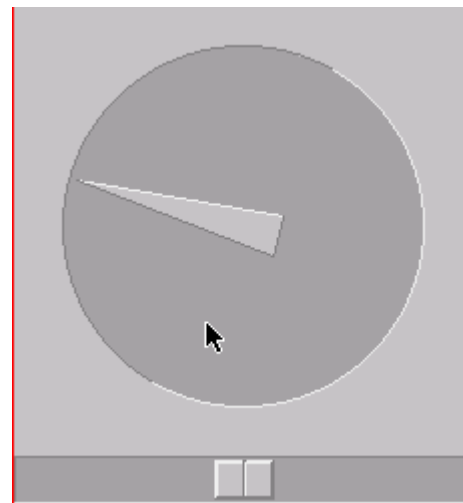
- 使用Qt Designer可以方便地设计出图形界面，一般需要如下的步骤：
  - 创建和初始化子部件
  - 设置子部件的布局
  - 设置Tab键的次序
  - 建立信号与插槽的连接

## 5.2 Qt Designer的使用实例

- 下面通过一个简单的例子来体验一下Qt Designer的使用。

### ■ 涉及的控件:

- 一个dial控件(上面的控件)
- 一个Slider控件(下面的控件)

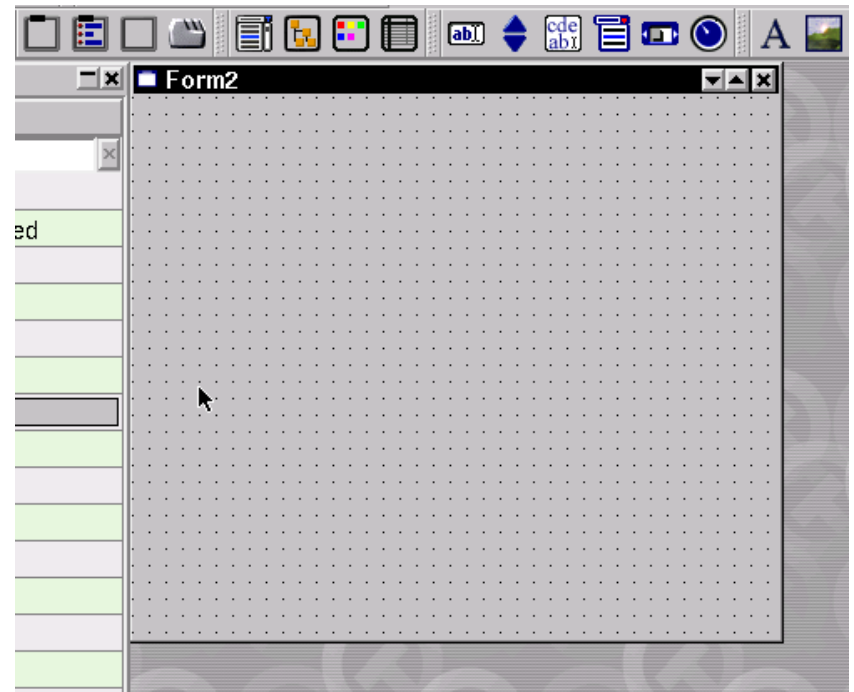
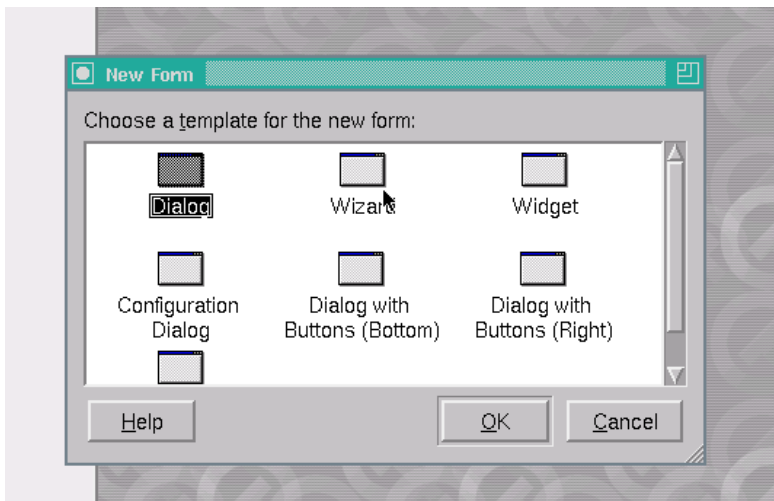


### ■ 实现的功能:

- 拖动slider时，dial中的指针会随着转动
- 用鼠标拖动dial中的指针的时候，slider指示会变化


## 5.3 启动Qt Designer

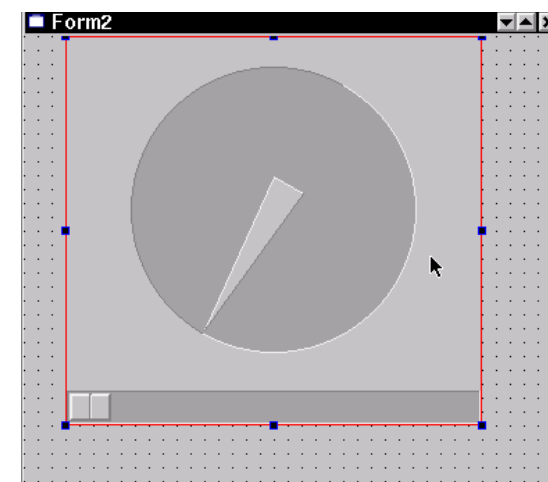
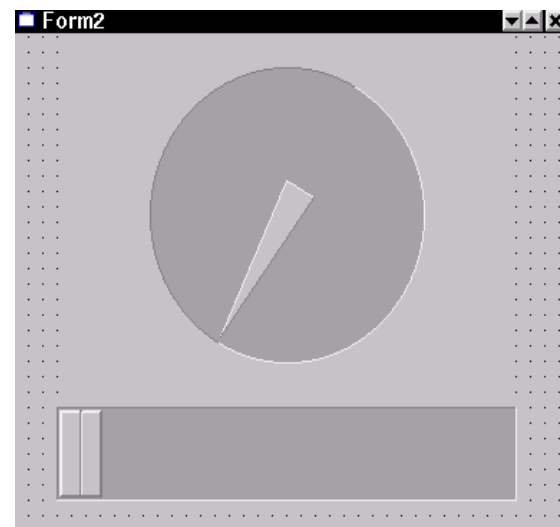
- 启动Qt Designer (在\$QT2DIR/bin里面, QT2DIR即qt-2.3.2/所在的目录)  
[root@localhost test]# \$QT2DIR/bin/designer
- 新建一个dialog




## 5.4 控件的加入与布置

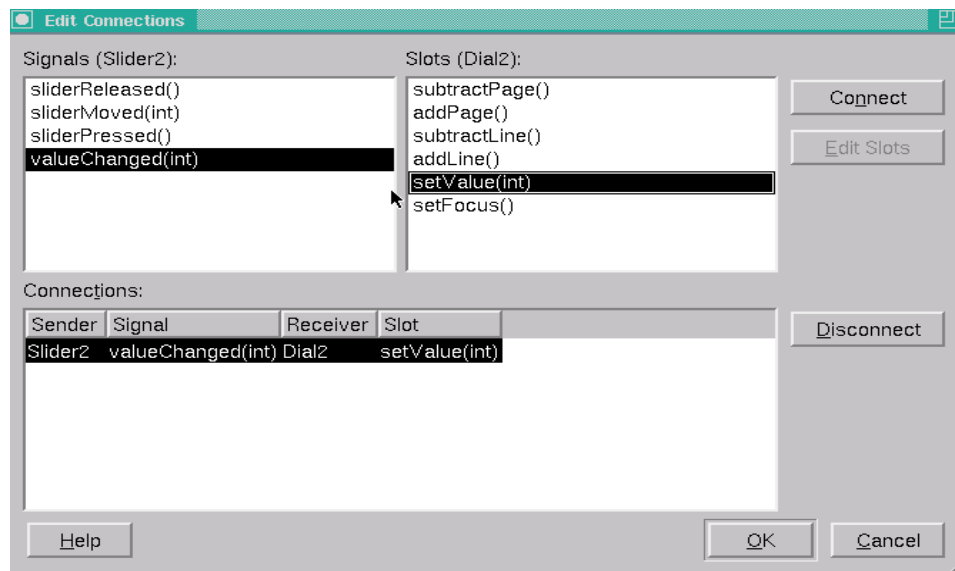
- 将所需的控件加入到dialog中

- 调整布局，使用工具栏上面的布局的控件进行调整，右图使用 



## 5.5 建立信号与插槽的连接

- 建立信号与插槽的连接方法如下：
  - 首先是slider发送signal、dial接收的情况
    - 点击signal/slot的图标 
    - 在鼠标左键被按下的情况下连接slider和dial控件，出现如下对话框
    - 在signal栏中选择valueChanged(int),在slot栏中选择setValue(int)
    - 然后点击connect按钮得到连接
  - 同理，可以得到dial发送信号，而slider接收的情况。





## 5.6 ui文件和cpp文件

### ■ 保存ui文件:

- 将生成的ui文件保存到项目所在目录中(请专门建立一个项目的目录, 里面不要有无关的文件), 比如存为form.ui。

### ■ 根据ui文件编写包含main函数的cpp文件:

- 内容如下

```
#include <qapplication.h>
```

```
#include "form.h" /*头文件名称与ui文件名称相同*/
```

```
int main(int argc, char** argv) {
```

```
    QApplication app(argc, argv);
```

```
    Form1 form; /*默认创建的类是Form1, 可以在Designer的属性窗口中进行修改*/
```

```
    app.setMainWidget(&form);
```

```
    form.show();
```

```
    return app.exec();
```

```
}
```

## 5.7 编译和运行程序

- 首先设置环境变量(qtopia\_env/目录下的qvfb.sh):

```
[root@localhost: test] source ~/qtopia_env/qvfb.sh
```

- 进入项目的目录

```
[root@localhost: test] progen -t app.t -o form.pro
```

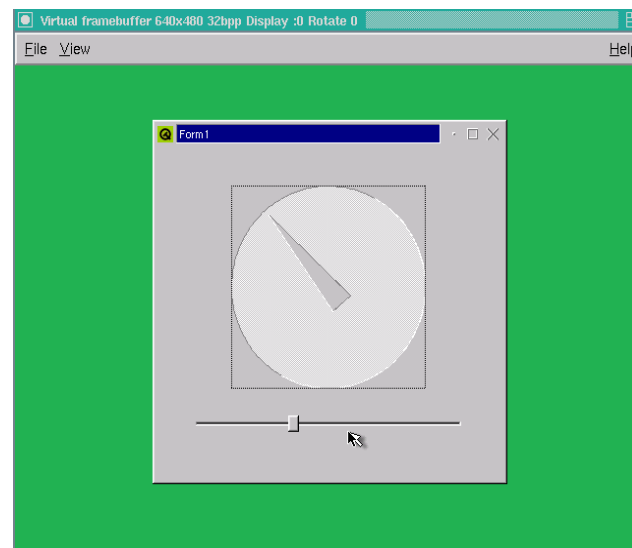
```
[root@localhost: test] tmake -o Makefile form.pro
```

```
[root@localhost: test] make
```

- 得到编译完成的二进制文件

- 运行如右图所示:

```
[root@localhost: test] ./form -qws
```



## 5.8 设置qt embedded运行环境

为了能够在目标板上运行qt embedded程序：

- 首先编译前设置交叉编译环境变量(qtopia\_env/目录下的target.sh):

```
[root@localhost: test] source ~/qtopia_env/target.sh
```

- 然后将qt-embedded-2.3.10/lib/目录下的libqte.so.2.3.10传到目标板的/mnt/yaffs/Qtopia/lib目录下面，并建立如下的连接

```
[/mnt/yaffs/Qtopia/lib] ln -sf libqte.so.2.3.10 libqte.so.2.3
```

```
[/mnt/yaffs/Qtopia/lib] ln -sf libqte.so.2.3.10 libqte.so.2
```

```
[/mnt/yaffs/Qtopia/lib] ln -sf libqte.so.2.3.10 libqte.so
```

- 最后在目标板上设置如下的环境变量

```
export QTDIR=/usr/qpe
```

```
export KDEDIR=/usr/qpe
```

```
export LD_LIBRARY_PATH=/usr/qpe/lib
```

```
export QWS_MOUSE_PROTO="TPanel:/dev/input/event0 USB"
```



# 内容提纲(6/6)

- 1. 嵌入式Linux图形界面编程基础
- 2. Linux的帧缓冲设备编程基础
- 3. Qt/Embedded开发基础
- 4. Qt/Embedded开发实例
- 5. 用Qt Designer设计图形界面
- 6. 实验内容与要求

## 6.1 本次实验要求

- 学习嵌入式Linux图形界面编程的基本方法;
- 在实验平台上实现对Framebuffer设备操作:
  - 在实验平台上通过直接操作Framebuffer设备实现基本绘图功能,如画线、矩形、圆等基本功能以及填充图形、显示图片等高级功能。
- 有能力可选在实验平台移植Qt/E开发一个简单程序;



## 6.2 实验注意事项

- 显示图片需要把图形转换成数组，这与在单片机等设备上显示图形是一样的，有一些现成的转换程序可利用；
- 显示汉字其实和显示图片是一样的，关键在于从汉字库中找到字模。