“黑色经典”系列之《嵌入式 Linux 应用程序开发详解》

FARSIGHT

第 12 章　Qt 图形编程

本章目标

本书从第 6 章到第 10 章详细讲解了嵌入式 Linux 应用程序的开发，这些都是属于用户空间的内容。本章将进入到 Linux 的内核空间，初步介绍嵌入式 Linux 设备驱动的开发。驱动的开发流程相对于应用程序的开发是全新的，希望读者能尽可能地抛弃以前的编程习惯来进行本章的学习。通过本章的学习，读者将会掌握以下内容。

- Linux 设备驱动的基本概念
- Linux 设备驱动程序的基本功能
- Linux 设备驱动的运作过程
- 常见设备驱动接口函数
- 掌握 LCD 设备驱动程序编写步骤
- 掌握键盘设备驱动程序编写步骤

Linux公社（LinuxIDC.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

LinuxIDC.com提供包括Ubuntu，Fedora，SUSE技术，以及最新IT资讯等Linux专业类网站。

并被收录到Google 网页目录-计算机 > 软件 > 操作系统 > Linux 目录下。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

包括:

Ubuntu专题

Fedora专题

RedHat专题

SUSE专题

红旗Linux专题

Android专题

12.1 嵌入式 GUI 简介

目前的桌面机操作系统大多有着美观、操作方便、功能齐全的 GUI（图形用户界面），例如 KDE 或者 GNOME。GUI（图形用户界面）是指计算机与其使用者之间的对话接口，可以说，GUI 是当今计算机技术的重大成就。它的存在为使用者提供了友好便利的界面，并大大地方便了非专业用户的使用，使得人们从繁琐的命令中解脱出来，可以通过窗口、菜单方便地进行操作。

而在嵌入式系统中，GUI 的地位也越来越重要，但是不同于桌面机系统，嵌入式 GUI 要求简单、直观、可靠、占用资源小且反应快速，以适应系统硬件资源有限的条件。另外，由于嵌入式系统硬件本身的特殊性，嵌入式 GUI 应具备高度可移植性与可裁减性，以适应不同的硬件条件和使用需求。总体来讲，嵌入式 GUI 具备以下特点：

- 体积小；
- 运行时耗用系统资源小；
- 上层接口与硬件无关，高度可移植；
- 高可靠性；
- 在某些应用场合应具备实时性。

UNIX 环境下的图形视窗标准为 X Window System，Linux 是类 UNIX 系统，所以顶层运行的 GUI 系统是兼容 X 标准的 XFree86 系统。X 标准大致可以划分 X Server、Graphic Library（底层绘图函数库）、Toolkits、Window Manager 等几大部分。其好处是具有可扩展性、可移植性等优点，但对于嵌入式系统而言无疑太过庞大、累赘、低效。目前流行的嵌入式 GUI 与 X 思路不同，这些 GUI 一般不局限于 X 标准，更强调系统的空间和效率。

12.1.1 Qt/Embedded

表 12.1 归纳了 Qt/Embedded 的一些优缺点

表 12.1　　Qt/Embedded 分析

Qt/Embedded 分析		
优点	以开发包形式提供	包括了图形设计器，Makefile 制作工具，字体国际化工具，Qt 的 C++类库等
	跨平台	支持 Microsoft Windows 95/98/2000、Microsoft Windows NT、MacOS X、Linux、Solaris、HP-UX、Tru64 (Digital UNIX)、Irix、FreeBSD、BSD/OS、SCO、AIX 等众多平台
	类库支持跨平台	Qt 类库封装了适应不同操作系统的访问细节，这正是 Qt 的魅力所在
	模块化	可以任意裁减
缺点	结构也过于复杂臃肿，很难进行底层的扩充、定制和移植	例如： • 尽管 Qt/Embedded 声称它最小可以裁剪到 630KB，但这时的 Qt/Embedded 库已经基本失去了使用价值 • 它提供的控件集沿用了 PC 风格，并不太适合许多手持设备的操作要求 • Qt/Embedded 的底层图形引擎只能采用 framebuffer，只能针对高端嵌入式图形领域的应用而设计的 • 由于该库的代码追求面面俱到，以增加它对多种硬件设备的支持，造成了其底层代码比较凌乱，各种补丁较多的问题

12.1.2 MiniGUI

提起国内的开源软件，就肯定会提到 MiniGUI，它由魏永明先生和众多志愿者开发，是一个基于 Linux 的实时嵌入式系统的轻量级图形用户界面支持系统。

MiniGUI 分为最底层的 GAL 层和 IAL 层，向上为基于标准 POSIX 接口中 pthread 库的 Mini-thread 架构和基于 Server/Client 的 Mini-Lite 架构。其中前者受限于 thread 模式对于整个系统的可靠性影响——进程中某个 thread 的意外错误可能导致整个进程的崩溃，该架构应用于系统功能较为单一的场合。Mini-Lite 应用于多进程的应用场合，采用多进程运行方式设计的 Server/Client 架构能够较好地解决各个进程之间的窗口管理、Z 序剪切等问题。MiniGUI 还有一种从 Mini-Lite 衍生出的 standalone 运行模式。与 Lite 架构不同的是，standalone 模式一次只能以窗口最大化的方式显示一个窗口。这在显示屏尺寸较小的应用场合具有一定的应用意义。

MiniGUI 的 IAL 层技术 SVGA lib、LibGGI、基于 framebuffer 的 native 图形引擎以及哑图形引擎等，对于 Trolltech 公司的 QVFB 在 X Window 下也有较好的支持。IAL 层则支持 Linux 标准控制台下的 GPM 鼠标服务、触摸屏、标准键盘等。

MiniGUI 下丰富的控件资源也是 MiniGUI 的特点之一。当前 MiniGUI 的最新版本是 1.3.3。在该版本的控件中已经添加了窗口皮肤、工具条等桌面 GUI 中的高级控件支持。对比其他系统，“Mini”是 MiniGUI 的特色，轻量、高性能和高效率的 MiniGUI 已经应用在电视机顶盒、实时控制系统、掌上电脑等诸多场合。

12.1.3 Microwindows、Tiny X 等

Microwindows Open Source Project 成立的宗旨在于针对体积小的装置，建立一套先进的视窗环境，在 Linux 桌面上通过交叉编译可以很容易地制作出 Microwindows 的程序。Microwindows 能够在没有任何操作系统或其他图形系统的支持下运行，它能对裸显示设备进行直接操作。这样 Microwindows 就显得十分小巧，便于移植到各种硬件和软件系统上。

然而 Microwindows 的免费版本进展一直很慢，几乎处于停顿状态，而且至今为止，国内没有任何一家对 Microwindows 提供全面技术支持、服务和担保的专业公司。

Tiny X Server 是 XFree86 Project 的一部分，由 Keith Pachard 发展起来的，而他本身就是 XFree86 专案的核心成员之一。一般的 X Server 都太过于庞大，因此 Keith Packard 就以 XFree86 为基础，精简而成 Tiny X Server，它的体积可以小到几百 KB 而已，非常适合应用于嵌入式环境。

就纯 X Window System 搭配 Tiny X Server 架构来说，其最大的优点就是具有很好的弹性开发机制，并能大大提高开发速度。因为与桌面的 X 架构相同，因此相对于很多以 Qt、GTK+、FLTK 等为基础开发的软件可以很容易地移植上来。

虽然移植方便，但是却有体积大的缺点，由于很多软件本来是针对桌面环境开发的，因此无形之中具备了桌面环境中很多复杂的功能。因此“调校”变成采用此架构最大的课题，有时候重新改写都可能比调校所需的时间还短。

表 12.2 所示总结了常见 GUI 的参数比较。

表 12.2　　常见 GUI 参数比较

名称 / 参数	MiniGUI	OpenGUI	Qt/Embedded
API（完备性）	Win32（很完备）	私有（很完备）	Qt（C++）（很完备）
函数库典型大小	300KB	300KB	600KB
移植性	很好	只支持 x86 平台	较好
授权条款	LGPL	LGPL	QPL/GPL
系统消耗	小	最小	最大
操作系统支持	Linux	Linux，DOS，QNX	Linux

12.2　Qt/Embedded 开发入门

12.2.1　Qt/Embedded 介绍

1．架构

Qt/Embedded 以原始 Qt 为基础，并做了许多出色的调整以适用于嵌入式环境。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 很省内存，因为它不需要一个 X 服务器或是 Xlib 库，它在底层撅弃了 X lib，采用 framebuffer（帧缓冲）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/Embedde 的应用程序可以直接写内核缓冲帧，这避免开发者使用繁琐的 Xlib/Server 系统。图 12.1 所示比较了 Qt/Embedded 与 Qt/X11 的架构区别。

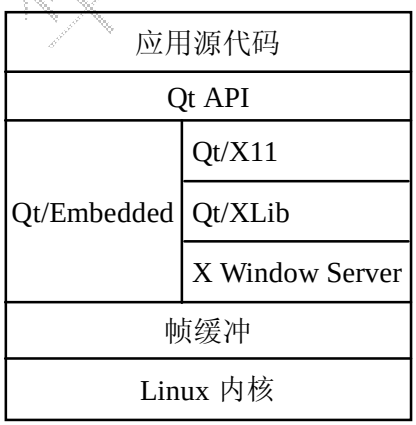

图 12.1　Qt/Embedded 与 Qt/X11 的 Linux 版本的比较

使用单一的 API 进行跨平台的编程可以有很多好处。提供嵌入式设备和桌面计算机环境下应用的公司可以培训开发人员使用同一套工具开发包，这有利于开发人员之间共享开发经验与知识，也使得管理人员在分配开发人员到项目中的时候增加灵活性。更进一步来说，针对某个平台而开发的应用和组件也可以销售到 Qt 支持的其他平台上，从而以低廉的成本扩大

产品的市场。

（1）窗口系统

一个 Qt/Embedded 窗口系统包含了一个或多个进程，其中的一个进程可作为服务器。该服务进程会分配客户显示区域，以及产生鼠标和键盘事件。该服务进程还能够提供输入方法和一个用户接口给运行起来的客户应用程序。该服务进程其实就是一个有某些额外权限的客户进程。任何程序都可以在命令行上加上“-qws”的选项来把它作为一个服务器运行。

客户与服务器之间的通信使用共享内存的方法实现，通信量应该保持最小，例如客户进程直接访问帧缓冲来完成全部的绘制操作，而不会通过服务器，客户程序需要负责绘制它们自己的标题栏和其他式样。这就是 Qt/Embedded 库内部层次分明的处理过程。客户可以使用 QCOP 通道交换消息。服务进程简单的广播 QCOP 消息给所有监听指定通道的应用进程，接着应用进程可以把一个插槽连接到一个负责接收的信号上，从而对消息做出响应。消息的传递通常伴随着二进制数据的传输，这是通过一个 QDataStream 类的序列化过程来实现的，有关这个类的的描述，请读者参考相关资料。

QProcess 类提供了另外一种异步的进程间通信机制。它用于启动一个外部的程序并且通过写一个标准的输入和读取外部程序的标准输出和错误码来和它们通信。

（2）字体

Qt/Embedded 支持四种不同的字体格式：True Type 字体（TTF），Postscript Type1 字体，位图发布字体（BDF）和 Qt 的预呈现（Pre-rendered）字体（QPF）。Qt 还可以通过增加 QfontFactory 的子类来支持其他字体，也可以支持以插件方式出现的反别名字体。

每个 TTF 或者 TYPE1 类型的字体首次在图形或者文本方式的环境下被使用时，这些字体的字形都会以指定的大小被预先呈现出来，呈现的结果会被缓冲。根据给定的字体尺寸（例如 10 或 12 点阵）预先呈现 TTF 或者 TYPE1 类型的字体文件并把结果以 QPF 的格式保存起来，这样可以节省内存和 CPU 的处理时间。QPF 文件包含了一些必要的字体，这些字体可以通过 makeqpf 工具取得，或者通过运行程序时加上“-savefonts”选项获取。如果应用程序中使用到的字体都是 QPF 格式，那么 Qt/Embedded 将被重新配置，并排除对 TTF 和 TYPE1 类型的字体的编译，这样就可以减少 Qt/Embedded 的库的大小和存储字体的空间。例如一个 10 点阵大小的包含所有 ASII 字符的 QPF 字体文件的大小为 1300Byte，这个文件可以直接从物理存储格式映射成为内存存储格式。

Qt/Embedded 的字体通常包括 Unicode 字体的一部分子集，ASII 和 Latin-1。一个完整的 16 点阵的 Unicode 字体的存储空间通常超过 1MB，我们应尽可能存储一个字体的子集，而不是存储所有的字，例如在一个应用中，仅仅需要以 Cappuccino 字体、粗体的方式显示产品的名称，但是却有一个包含了全部字形的字体文件。

（3）输入设备及输入法

Qt/Embedded 3.0 支持几种鼠标协议：BusMouse、IntelliMouse,Microsoft 和 MouseMan.Qt/Embedded 还支持 NECVr41XX 和 iPAQ 的触摸屏。通过从 QWSMouseHandler 或者 QcalibratedMouseHandler 派生子类，开发人员可以让 Qt/Embedded 支持更多的客户指示设备。

Qt/Embedded 支持标准的 101 键盘和 Vr41XX 按键，通过子类化 QWSKeyboardHandler 可以让 Qt/Embedded 支持更多的客户键盘和其他的非指示设备。

对于非拉丁语系字符（例如阿拉伯，中文，希伯来和日语）的输入法，需要把它写成过

滤器的方式，并改变键盘的输入。输入法的作者应该对全部的 Qt API 的使用有完全的认识。在一个无键盘的设备上，输入法成了惟一的输入字符的手段。Qtpia 提供了 4 种输入方法：笔迹识别器、图形化的标准键盘、Unicode 键盘和居于字典方式提取的键盘。

（4）屏幕加速

通过子类化 QScreen 和 QgfxRaster 可以实现硬件加速，从而为屏幕操作带来好处。Trolltech 提供了 Mach64 和 Voodoo3 视频卡的硬件加速的驱动例子，同时可以按照协议编写其他的驱动程序。

2．Qt 的开发环境

Qt/Embedded 的开发环境可以取代那些我们熟知的 UNIX 和 Windows 开发工具。它提供了几个跨平台的工具使得开发变得迅速和方便，尤其是它的图形设计器。UNIX 下的开发者可以在 PC 机或者工作站使用虚拟缓冲帧，从而可以模仿一个和嵌入式设备的显示终端大小，像素相同的显示环境。

嵌入式设备的应用可以在安装了一个跨平台开发工具链的不同的平台上编译。最通常的做法是在一个 UNIX 系统上安装跨平台的带有 libc 库的 GNU C++编译器和二进制工具。在开发的许多阶段，一个可替代的做法是使用 Qt 的桌面版本，例如通过 Qt/X11 或是 Qt/Windows 来进行开发。这样开发人员就可以使用他们熟悉的开发环境，例如微软公司的 Visual C++或者 Borland C++。在 UNIX 操作系统下，许多环境也是可用的，例如 Kdevelop，它也支持交互式开发。

如果 Qt/Embedded 的应用是在 UNIX 平台下开发的话，那么它就可以在开发的机器上以一个独立的控制台或者虚拟缓冲帧的方式来运行，对于后者来说，其实是有一个 X11 的应用程序虚拟了一个缓冲帧。通过指定显示设备的宽度，高度和颜色深度，虚拟出来的缓冲帧将和物理的显示设备在每个像素上保持一致。这样每次调试应用时开发人员就不用总是刷新嵌入式设备的 FLASH 存储空间，从而加速了应用的编译、链接和运行周期。运行 Qt 的虚拟缓冲帧工具的方法是在 Linux 的图形模式下运行以下命令：

```
qvfb (回车)
```

当 Qt 嵌入式的应用程序要把显示结果输出到虚拟缓冲帧时，我们在命令行运行这个程序时，在程序名后加上-qws 的选项。例如：$> hello－qws。

3．Qt 的支撑工具

Qt 包含了许多支持嵌入式系统开发的工具，有两个最实用的工具是 qmake 和 Qt designer（图形设计器）。

- qmake 是一个为编译 Qt/Embedded 库和应用而提供的 Makefile 生成器。它能够根据一个工程文件（.pro）产生不同平台下的 Makefile 文件。qmake 支持跨平台开发和影子生成（影子生成是指当工程的源代码共享给网络上的多台机器时，每台机器编译链接这个工程的代码将在不同的子路径下完成，这样就不会覆盖别人的编译链接生成的文件。qmake 还易于在不同的配置之间切换。）
- Qt 图形设计器可以使开发者可视化地设计对话框而不需编写代码。使用 Qt 图形设计器的布局管理可以生成能平滑改变尺寸的对话框。

qmake 和 Qt 图形设计器是完全集成在一起的。

12.2.2 Qt/Embedded 信号和插槽机制

1. 机制概述

信号和插槽机制是 Qt 的核心机制，要精通 Qt 编程就必须对信号和插槽有所了解。信号和插槽是一种高级接口，应用于对象之间的通信，它是 Qt 的核心特性，也是 Qt 区别于其他工具包的重要地方。信号和插槽是 Qt 自行定义的一种通信机制，它独立于标准的 C/C++语言，因此要正确的处理信号和插槽，必须借助一个称为 moc（Meta Object Compiler）的 Qt 工具，该工具是一个 C++预处理程序，它为高层次的事件处理自动生成所需要的附加代码。

所谓图形用户接口的应用就是要对用户的动作做出响应。例如，当用户单击了一个菜单项或是工具栏的按钮时，应用程序会执行某些代码。大部分情况下，是希望不同类型的对象之间能够进行通信。程序员必须把事件和相关代码联系起来，这样才能对事件做出响应。以前的工具开发包使用的事件响应机制是易崩溃的，不够健壮的，同时也不是面向对象的。

以前，当使用回调函数机制把某段响应代码和一个按钮的动作相关联时，通常把那段响应代码写成一个函数，然后把这个函数的地址指针传给按钮，当那个按钮被单击时，这个函数就会被执行。对于这种方式，以前的开发包不能够确保回调函数被执行时所传递进来的函数参数就是正确的类型，因此容易造成进程崩溃。另外一个问题是，回调这种方式紧紧地绑定了图形用户接口的功能元素，因而很难开发进行独立的分类。

信号与插槽机制是不同的。它是一种强有力的对象间通信机制，完全可以取代原始的回调和消息映射机制。在 Qt 中信号和插槽取代了那些上述这些凌乱的函数指针，使得用户编写这些通信程序更为简洁明了。信号和插槽能携带任意数量和任意类型的参数，他们是类型完全安全的，因此不会像回调函数那样产生 core dumps。

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）出去了，这就是对象所要做的全部工作，它不知道另一端是谁在接收这个信号。这就是真正的信息封装，它确保对象被当作一个真正的软件组件来使用。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信号与自己相连接。而且，对象并不了解具体的通信机制。

用户可以将很多信号与单个插槽进行连接，也可以将单个信号与很多插槽进行连接，甚至将一个信号与另外一个信号相连接也是可能的，这时无论第一个信号什么时候发射，系统都将立刻发射第二个信号。总之，信号与插槽构造了一个强大的部件编程机制。

图 12.2 所示表示了对象间信号与插槽之间的关系。

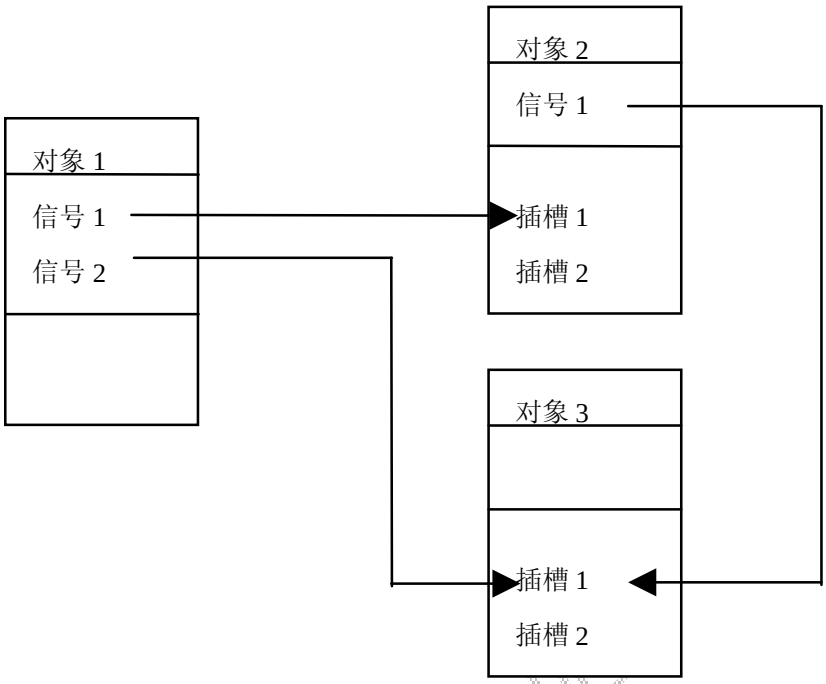

图 12.2　对象间信号与插槽的关系

2．信号与插槽实现实例

（1）信号

当某个信号对其客户或所有者内部状态发生改变时，信号就被一个对象发射。只有定义了这个信号的类及其派生类才能够发射这个信号。当一个信号被发射时，与其相关联的插槽将被立刻执行，就像一个正常的函数调用一样。信号－插槽机制完全独立于任何 GUI 事件循环。只有当所有的槽返回以后发射函数（emit）才返回。如果存在多个槽与某个信号相关联，那么，当这个信号被发射时，这些槽将会一个接一个地执行，但是它们执行的顺序将会是随机的、不确定的，用户不能人为地指定哪个先执行、哪个后执行。

Qt 的 signals 关键字指出进入了信号声明区，随后即可声明自己的信号。例如，下面定义了 3 个信号：

```
signals:
void mySignal();
void mySignal(int x);
void mySignalParam(int x,int y);
```

在上面的定义中，signals 是 Qt 的关键字，而非 C/C++的。接下来的一行 void mySignal()定义了信号 mySignal，这个信号没有携带参数；接下来的一行 void mySignal(int x)定义了重名信号 mySignal，但是它携带一个整形参数，这有点类似于 C++中的虚函数。从形式上讲信号的声明与普通的 C++函数是一样的，但是信号却没有函数体定义。另外，信号的返回类型都是 void。信号由 moc 自动产生，它们不应该在.cpp 文件中实现。

（2）插槽

插槽是普通的 C++成员函数，可以被正常调用，它们惟一的特殊性就是很多信号可以与其相关联。当与其关联的信号被发射时，这个插槽就会被调用。插槽可以有参数，但插槽的参数不能有缺省值。

既然插槽是普通的成员函数，因此与其他的函数一样，它们也有存取权限。插槽的存取权限决定了谁能够与其相关联。同普通的 C++成员函数一样，插槽函数也分为 3 种类型，即 public slots、private slots 和 protected slots。

- public slots：在这个区内声明的槽意味着任何对象都可将信号与之相连接。这对于组件编程非常有用，用户可以创建彼此互不了解的对象，将它们的信号与槽进行连接以便信息能够正确地传递。
- protected slots：在这个区内声明的槽意味着当前类及其子类可以将信号与之相连接。这适用于那些槽，它们是类实现的一部分，但是其界面接口却面向外部。
- private slots：在这个区内声明的槽意味着只有类自己可以将信号与之相连接。这适用于联系非常紧密的类。

插槽也能够被声明为虚函数，这也是非常有用的。插槽的声明也是在头文件中进行的。例如，下面声明了 3 个插槽：

```
    public slots:
    void mySlot();
    void mySlot(int x);
    void mySignalParam(int x,int y);
```

（3）信号与插槽关联

通过调用 QObject 对象的 connect 函数可以将某个对象的信号与另外一个对象的插槽函数相或信号相关联，当发射者发射信号时，接收者的槽函数或信号将被调用。

该函数的定义如下所示：

```
    bool QObject::connect ( const QObject * sender, const char * signal,const
QObject * receiver, const char * member ) [static]
```

这个函数的作用就是将发射者 sender 对象中的信号 signal 与接收者 receiver 中的 member 插槽函数联系起来。当指定信号 signal 时必须使用 Qt 的宏 SIGNAL()，当指定插槽函数时必须使用宏 SLOT()。如果发射者与接收者属于同一个对象的话，那么在 connect 调用中接收者参数可以省略。

- 信号与插槽相关联

下例定义了两个对象：标签对象 label 和滚动条对象 scroll，并将 valueChanged()信号与标签对象的 setNum()插槽函数相关联，另外信号还携带了一个整型参数，这样标签总是显示滚动条所处位置的值。

```
    QLabel *label = new QLabel;
    QScrollBar *scroll = new QScrollBar;
    QObject::connect( scroll, SIGNAL(valueChanged(int)),label, SLOT(setNum(int)) );
```

- 信号与信号相关联

在下面的构造函数中，MyWidget 创建了一个私有的按钮 aButton，按钮的单击事件产生的信号 clicked()与另外一个信号 aSignal()进行了关联。这样，当信号 clicked()被发射时，信号 aSignal()也接着被发射。如下所示：

```
class MyWidget : public QWidget
{
public:
MyWidget();
...
signals:
void aSignal();
...
private:
...
QPushButton *aButton;
};

MyWidget::MyWidget()
{
aButton = new QPushButton( this );
connect( aButton, SIGNAL(clicked()), SIGNAL(aSignal()) );
}
```

（4）解除信号与插槽关联

当信号与槽没有必要继续保持关联时，用户可以使用 disconnect 函数来断开连接。其定义如下所示：

```
bool QObject::disconnect ( const QObject * sender, const char * signal,const
Object * receiver, const char * member ) [static]
```

这个函数断开发射者中的信号与接收者中的槽函数之间的关联。

有 3 种情况必须使用 disconnect()函数。

- 断开与某个对象相关联的任何对象

当用户在某个对象中定义了一个或者多个信号，这些信号与另外若干个对象中的槽相关联，如果想要切断这些关联的话，就可以利用这个方法，非常之简洁。如下所示：

```
disconnect( myObject, 0, 0, 0 )
或者
myObject->disconnect()
```

- 断开与某个特定信号的任何关联

这种情况是非常常见的，其典型用法如下所示：

```
disconnect( myObject, SIGNAL(mySignal()), 0, 0 )
或者
myObject->disconnect( SIGNAL(mySignal()) )
```

- 断开两个对象之间的关联

这也是非常常用的情况，如下所示：

```
disconnect( myObject, 0, myReceiver, 0 )
或者
myObject->disconnect( myReceiver )
```

注意 在 disconnect 函数中 0 可以用作一个通配符，分别表示任何信号、任何接收对象、接收对象中的任何槽函数。但是发射者 sender 不能为 0，其他 3 个参数的值可以等于 0。

12.2.3 搭建 Qt/Embedded 开发环境

一般来说，用 Qt/Embedded 开发的应用程序最终会发布到安装有嵌入式 Linux 操作系统的小型设备上，所以使用装有 Linux 操作系统的 PC 机或者工作站来完成 Qt/Embedded 开发当然是最理想的环境，此外 Qt/Embedded 也可以安装在 UNIX 或 Windows 系统上。这里就以安装到 Linux 操作系统为例进行介绍。

这里需要有 3 个软件安装包：tmake 工具安装包，Qt/Embedded 安装包，Qt 的 X11 版的安装包。

- tmake1.11 或更高版本：生成 Qt/Embedded 应用工程的 Makefile 文件。
- Qt/Embedded：Qt/Embedded 安装包。
- Qt 2.3.2 for X11：Qt 的 X11 版的安装包，产生 x11 开发环境所需要的两个工具。

注意 这些软件安装包都有许多不同的版本，由于版本的不同会导致这些软件在使用时可能引起的冲突，为此必须依照一定的安装原则，Qt/Embedded 安装包的版本必须比 Qt for X11 的安装包的版本新，这是因为 Qt for X11 的安装包中的两个工具 uic 和 designer 产生的源文件会和 Qt/Embedded 的库一起被编译链接，因此要本着“向前兼容”的原则，Qt for X11 的版本应比 Qt/Embedded 的版本旧。

1．安装 tmake

用户可使用普通的解压缩即可，注意要将路径添加到全局变量中去，如下所示：

```
tar zxvf tmake-1.11.tar.gz
export TMAKEDIR=$PWD/tmake-1.11
export TMAKEPATH=$TMAKEDIR/lib/qws/linux-x86-g++
export PATH=$TMAKEDIR/bin:$PATH
```

2．安装 Qt/Embedded 2.3.7

这里使用常见的解压命令及安装命令即可，要注意这里的路径与不同的系统有关，读者

要根据实际情况进行修改。另外，这里的 configure 命令带有参数“-qconfig –qvfb –depths 4816，32”分别为指定 Qt 嵌入式开发包生成虚拟缓冲帧工具 qvfb，并支持 4，8，16，32 位的显示颜色深度。另外读者也可以在 configure 的参数中添加“-system”、“-jpeg”或“gif”命令，使 Qt/Embedded 平台能支持 jpeg、gif 格式的图形。

Qt/Embedded 开发包有 5 种编译范围的选项，使用这些选项可控制 Qt 生成的库文件的大小。如命令 make sub-src 指定按精简方式编译开发包，也就是说有些 Qt 类未被编译。其他编译选项的具体用法可参见“./configure–help”命令查看。精简方式的安装步骤若下所示：

```
tar zxvf qt-embedded-2.3.7.tar.gz
cd qt-2.3.7
export QTDIR=$PWD
export QTEDIR=$QTDIR
export PATH=$QTDIR/bin:$PATH
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
./configure -qconfig local-qvfb -depths 4,8,16,32
make sub-src
```

3．安装 Qt/X11 2.3.2

与上一步类似，用户也可以在 configure 后添加一定的参数，如“-no-opengl”或“-no-xfs”，可以键入命令“./configure –help”来获得一些帮助信息。

```
tar xfz qt-x11-2.3.2.tar.gz
cd qt-2.3.2
export QTDIR=$PWD
export PATH=$QTDIR/bin:$PATH
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
./configure -no-opengl
make
make -C tools/qvfb
mv tools/qvfb/qvfb bin
cp bin/uic $QTEDIR/bin
```

12.2.4 Qt/Embedded 窗口部件

QT 提供了一整套的窗口部件。它们组合起来可用于创建用户界面的可视元素。按钮、菜单、滚动条、消息框和应用程序窗口都是窗口部件的实例。因为所有的窗口部件既是控件又是容器，因此 QT 的窗口部件不能任意地分为控件和容器。通过子类化已存在的 QT 部件或少数时候必要的全新创建，自定义的窗口部件能很容易地创建出来。

窗口部件是 QWidget 或其子类的实例，用户自定义的窗口通过子类化得到。如下图 12.3

所示：

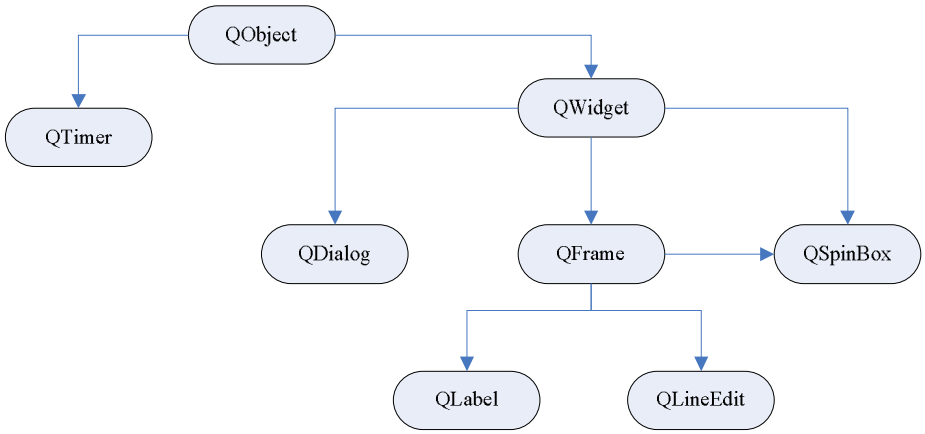

图 12.3　源自 QWidget 的类层次结构

一个窗口部件可包含任意数量的子部件。子部件在父部件的区域内显示。没有父部件的部件是顶级部件（比如一个窗口），通常在桌面的任务栏上有它们的入口。QT 不在窗口部件上施加任何限制。任何部件都可以是顶级部件，任何部件都可以是其他部件的子部件。通过自动或手动（如果你喜欢）则使用布局管理器可以设定子部件在父部件区域中的位置。如果父部件被停用，隐藏或删除，则同样的动作会地应用于它的所有子部件。

1．Hello 窗口实例

下面是一个显示“Hello Qt/Embedded!”的程序的完整的源代码：

```
#include <qapplication.h>
#include <qlabel.h>
int main( int argc, char **argv )
{
QApplication app( argc, argv );
QLabel *hello=new QLabel ("<font color=blue>Hello""<i>Qt Embedded!</i></font>",0);
app.setMainWidget( hello );
hello->show();
return app.exec();
}
```

下图 12.4 是该 Hello 窗口的运行效果图：

2．常见通用窗口组合

Qt 中还有一些常见的通用窗口，它们使用了 Windows 风格显示。如下图 12.5、12.6、12.7、12.8 分别描述了常见的一些通用窗口的组合使用。

图 12.4　Hello 窗口运行效果图

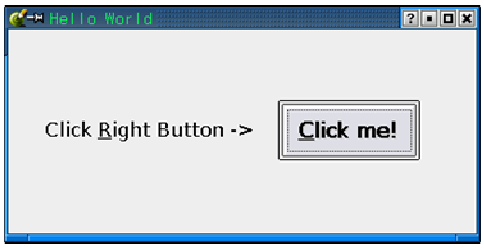

图 12.5　使用了 QHBox 进行排列一个标签和一个按钮

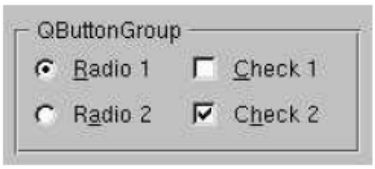

图 12.6　使用了 QbuttonGroup 的两个单选框和两个复选框

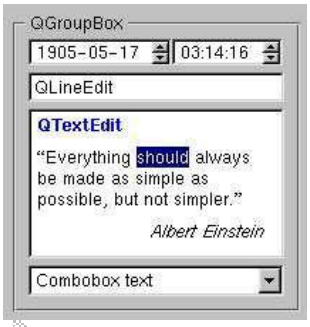

图 12.7　QgroupBox 组合图示

图 12.8 使用了 QgroupBox 进行排列的的日期类 QdateTimeEdit、一个行编辑框类 Qline-Edit、一个文本编辑类 QTextEdit 和一个组合框类 QcomboBox。

图 12.9 是以 QGrid 排列的一个 Qdial、一个 QprogressBar、一个 QspinBox、一个 QscrollBar、一个 QLCDNumber 和一个 Qslider。

图 12.10 是以 QGrid 排列的一个 QiconView、一个 QlistView、一个 QListBox 和一个 Qtable。

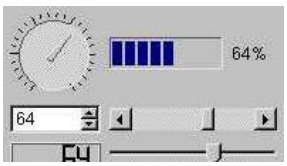

图 12.8　Qgrid 组合图示 1

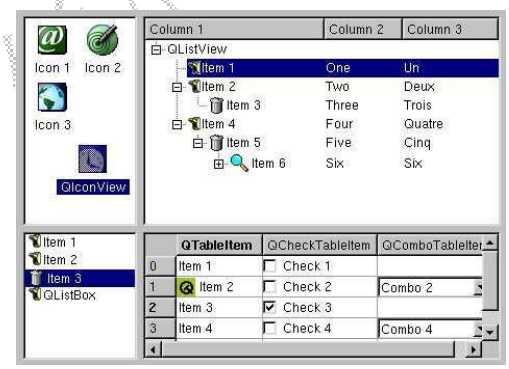

图 12.9　Qgrid 组合图示 2

图 12.10　钟表部件图示

3．自定义窗口

开发者可以通过子类化 QWidget 或它的一个子类创建他们自己的部件或对话框。为了举例说明子类化，下面提供了数字钟部件的完整代码。

钟表部件是一个能显示当前时间并自动更新的 LCD。一个冒号分隔符随秒数的流逝而闪烁，如图 12.10 所示。

Clock 从 QLCDNumber 部件继承了 LCD 功能。它有一个典型部件类所拥有的典型构造

函数，带有可选 parent 和 name 参数的（如果设置了 name 参数测试和调试会更容易）。系统有规律地调用从 QObject 继承的 timerEvent()函数。

它在 clock.h 中定义如下所示：

```
#include <qlcdnumber.h>
class Clock:public QLCDNumber
{
public:
Clock(QWidget *parent=0,const char *name=0);
protected:
void timerEvent(QTimerEvent *event);
private:
void showTime();
bool showingColon;
};
```

构造函数调 showTime()是用当前时间初始化钟表，并且告诉系统每 1000 毫秒调用一次 timerEvent()来刷新 LCD 的显示。在 showTime()中，通过调用 QLCDNumber::display()来显示当前时间。每次调用 showTime()来让冒号闪烁时，冒号就被空白代替。

clock.cpp 的源码如下所示：

```
#include <qdatetime.h>
#include "clock.h"
Clock::Clock(QWidget *parent,const char *name)
:QLCDNumber(parent,name),showingColon(true)
{
showTime();
startTimer(1000);
}
void Clock::timerEvent(QTimerEvent *)
{
showTime();
}
void Clock::showTime()
{
QString timer=QTime::currentTime().toString().left(5);
if(!showingColon)
time[2]=' ';
display(time);
showingColon=!showingColon;
}
```

文件 clock.h 和 clock.cpp 完整地声明并实现了 Clock 部件。

```
#include <qapplication.h>
#include "clock.h"
int main(int argc,char **argv)
{
QApplication app(argc,argv);
Clock *clock=new Clock;
app.setMainWidget(clock);
clock->show();
return app.exec();
}
```

12.2.5 Qt/Embedded 图形界面编程

Qt 提供了所有可能的类和函数来创建 GUI 程序。Qt 既可用来创建“主窗口”式的程序，即一个有菜单栏，工具栏和状态栏作为环绕的中心区域；也可以用来创建“对话框”式的程序，使用按钮和必要的选项卡来呈现选项与信息。Qt 支持 SDI（单文档界面）和 MDI（多文档界面）。Qt 还支持拖动，放下和剪贴板。工具栏可以在工具栏区域内移动，拖拽到其他区域或者作为工具托盘浮动起来。这个功能是内建的，不需要额外的代码，但程序员在需要时可以将约束工具栏的行为。

使用 Qt 可以大大简化编程程序。例如，如果一个菜单项，一个工具栏按钮和一个快捷键都完成同样的动作，那么这个动作只需要一份代码。

Qt 还提供消息框和一系列标准对话框，使得程序向用户提问和让用户选择文件、文件夹、字体以及颜色变得更加简单。为了呈现一个消息框或一个标准对话框，只需要用一个使用一个方便的 Qt 静态函数的一行的语句。

1．主窗口类

QMainWindow 类提供了一个典型应用程序的主窗口框架。

一个主窗口包含了一组标准窗体的集合。主窗口的顶部包含一个菜单栏，它的下方放置着一个工具栏，工具栏可以移动到其他的停靠区域。主窗口允许停靠的位置有顶部、左边、右边和底部。工具栏可以被拖放到一个停靠的位置，从而形成一个浮动的工具面板。主窗口的下方，也就是在底部的停靠位置下方有一个状态栏。主窗口的中间区域可以包含其他的窗体。提示工具和“这是什么”帮助按钮以旁述的方式阐述了用户接口的使用方法。

对于小屏幕的设备，使用 Qt 图形设计器定义的标准的 Qwidget 模板比使用主窗口类更好一些。典型的模板包含有菜单栏、工具栏，可能没有状态栏（在必要的情况下，可以用任务栏，标题栏来显示状态）。

例如，一个文本编辑器可以把 QTextEdit 作为中心部件：

```
QTextEdit *editor=new QTextEdit(mainWindow);
mainWindow->setCentralWidget(editor);
```

2．菜单类

弹出式菜单 QpopupMenu 类以垂直列表的方式显示菜单项，它可以是单个的（例如上下文相关菜单），可以以菜单栏的方式出现，或者是别的弹出式菜单的子菜单出现。

每个菜单项可以有一个图标，一个复选框和一个加速器（快捷键），菜单项通常对应一个动作（例如存盘），分隔器通常显示成一条竖线，它用于把一组相关联的动作菜单分立成组。

下面是一个建立包含有 New，Open 和 Exit 菜单项的文件菜单的例子。

```
QPopupMenu *fileMenu = new QPopupMenu( this );
fileMenu->insertItem( "&New", this, SLOT(newFile()), CTRL+Key_N );
fileMenu->insertItem( "&Open...", this, SLOT(open()), CTRL+Key_O );
fileMenu->insertSeparator();
fileMenu->insertItem( "E&xit", qApp, SLOT(quit()), CTRL+Key_Q );
```

当一个菜单项被选中，和它相关的插槽将被执行。加速器（快捷键）很少在一个没有键盘输入的设备上使用，Qt/Embedded 的典型配置并未包含对加速器的支持。上面出现的代码“&New”意思是在桌面机器上以“New”的方式显示出来，但是在嵌入式设备中，它只会显示为“New”。

QmenuBar 类实现了一个菜单栏，它会自动地设置几何尺寸并在它的父窗体的顶部显示出来，如果父窗体的宽度不够宽以至不能显示一个完整的菜单栏，那么菜单栏将会分为多行显示出来。Qt 内置的布局管理能够自动调整菜单栏。

Qt 的菜单系统是非常灵活的，菜单项可以被动态使能，失效，添加或者删除。通过子类化 QcustomMenuItem，用户可以建立客户化外观和功能的菜单项。

3．工具栏

工具栏可以被移动到中心区域的顶部、底部、左边或右边。任何工具栏都可以拖拽到工具栏区域的外边，作为独立的浮动工具托盘。

QtoolButton 类实现了具有一个图标，一个 3D 框架和一个可选标签的工具栏。切换型工具栏按钮具有可以打开或关闭某些特征。其他的则会执行一个命令。可以为活动、关闭、开启等模式，打开或关闭等状态提供不同的图标。如果只提供一个图标，Qt 能根据可视化线索自动地辨别状态，例如将禁用的按钮变灰，工具栏按钮也能触发弹出式菜单。

QToolButton 通常在 QToolBar 内并排出现。一个程序可含有任意数量的工具栏并且用户可以自由地移动它们。工具栏可以包括几乎所有部件，例如 QComboBox 和 QSpinBox。

4．旁述

现在的应用主要使用旁述的方式去解释用户接口的用法。Qt 提供了两种旁述的方式，即“提示栏”和“这是什么”帮助按钮。

- “提示栏”是小的，通常是黄色的矩形，当鼠标在窗体的某些位置游动时，它就会自动地出现。它主要用于解释工具栏按钮，特别是那些缺少文字标签说明的工具栏按钮的用途。下面就是如何设置一个“存盘”按钮的提示代码。

```
QToolTip::add(saveButton,"Save");
```

当提示字符出现之后，还可以在状态栏显示更详细的文字说明。

对于一些没有鼠标的设备（例如那些使用触点输入的设备），就不会出现鼠标的光标在窗体上进行游动，这样就不能激活提示栏。对于这些设备也许就需要使用“这是什么”帮助按钮，或者使用一种姿态来表示输入设备正在进行游动，例如用按下或者握住的姿态来表示现在正在进行游动。

- “这是什么”帮助按钮和提示栏有些相似，只不过前者是要用户单击它才会显示旁述。

在小屏幕设备上，要想单击“这是什么”帮助按钮，具体的方法是，在靠近应用的 X 窗口的关闭按钮“x”附近你会看到一个“？”符号的小按钮，这个按钮就是“这是什么”的帮助按钮。一般来说，“这是什么”帮助按钮按下后要显示的提示信息应该比提示栏要多一些。下面是设置一个存盘按钮的“这是什么”文本提示信息的方法：

```
QWhatsThis::add( saveButton, "Saves the current file." );
```

QToolTip 和 QWhatsThis 类提供了可以通过重新实现来获取更多特殊化行为的虚函数，比如根据鼠标在部件的位置来显示不同的文本。

5．动作

应用程序通常提供几种不同的方式来执行特定的动作。比如，许多应用程序通过菜单（Flie->Save），工具栏（像一个软盘的按钮）和快捷键（Ctrl+S）来提供“Save”动作。QAction 类封装了“动作”这个概念。它允许程序员在某个地方定义一个动作。

下面的代码实现了一个“Save”菜单项，一个“Save”工具栏按钮和一个“Save”快捷键，并且均有旁述帮助：

```
QAction *saveAct=new QAction("Save",saveIcon,"&Save",CTRL+Key_S,this);
connect(saveAct,SIGNAL(activated()),this,SLOT(save()));
saveAct->setWhatsThis("Saves the current file.");
saveAct->addTo(fileMenu);
saveAct->addTo(toolbar);
```

为了避免重复，使用 QAction 可保证菜单项的状态与工具栏保持同步，而工具提示能在需要的时候显示。禁用一个动作会禁用相应的菜单项和工具栏按钮。类似地，当用户单击切换型按钮时，相应的菜单项会因此被选中或不选。

12.2.6　Qt/Embedded 对话框设计

Qt/Embedded 对话框的设计比较复杂，要使用布局管理自动地设置窗体与别的窗体之间相对的尺寸和位置，这样可以确保对话框能够最好地利用屏幕上的可用空间，接着还要使用 Qt 图形设计器地可视化设计工具建立对话框。下面就详细讲解具体的步骤。

1．布局

Qt 的布局管理用于组织管理一个父窗体区域内的子窗体。它的特点是可以自动设置子窗

体的位置和大小，并可确定出一个顶级窗体的最小和缺省的尺寸，当窗体的字体或内容变化后，它可以重置一个窗体的布局。

使用布局管理，开发者可以编写独立于屏幕大小和方向之外的程序，从而不需要浪费代码空间和重复编写代码。对于一些国际化的应用程序，使用布局管理，可以确保按钮和标签在不同的语言环境下有足够的空间显示文本，不会造成部分文字被剪掉。

布局管理提供部分用户接口组件，例如输入法和任务栏变得更容易。我们可以通过一个例子说明这一点，当 Qtopia 的用户输入文字时，输入法会占用一定的文字空间，应用程序这时也会根据可用屏幕尺寸的变化调整自己。

Qtopia 的布局管理示例图如图 12.11 所示。

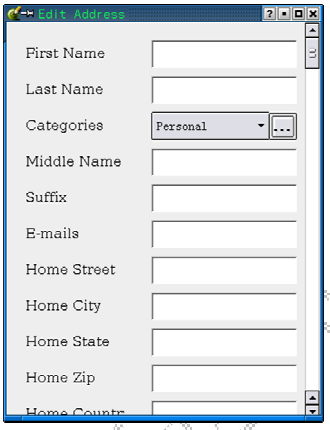

图 12.11　Qtopia 的布局管理

（1）内建布局管理器

Qt 提供了 3 种用于布局管理的类：QHBoxLayout，QVBox-Layout 和 QgridLayout。

- QHBoxLayout 布局管理把窗体按照水平方向从左至右排成一行。
- QVBoxLayout 布局管理把窗体按照垂直方向从上至下排成一列。
- QGridLayout 布局管理以网格的方式来排列窗体，一个窗体可以占据多个网格。

它们的示意图如图 12.12 所示。

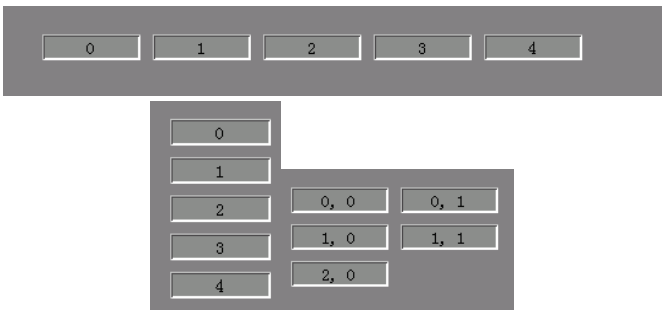

图 12.12　3 种布局管理类示意图

在多数情况下，Qt 的布局管理器为其管理的部件挑选一个最适合的尺寸以便窗口能够平滑地缩放。如果其缺省值不合适，开发者可以使用以下机制微调布局：

- 设置一个最小尺寸，一个最大尺寸，或者为一些子部件设置固定的大小。
- 设置一些延伸项目或间隔项目，延伸或间隔项目会填充空余的布局空间。
- 改变子部件的尺寸策略。通过调用 QWidget::setSizePolicy()，程序员可以仔细调整子部件的缩放行为。子部件可以设置为扩展、收缩、保持原大小等状态。
- 改变子部件的建议大小。QWidget::sizeHint()和 QWidget::minimumSizeHint()会根据内容返回部件的首选尺寸和最小首选尺寸。内建部件提供了合适的重新实现。
- 设置延伸因子。延伸因子规定了子部件的相应增量，比如，2/3 的可用空间分配给部件 A 而三分之一分配给 B。

（2）布局嵌套

布局可以嵌套任意层。图 12.13 显示了一个对话框的两种大小。

图 12.13　一个对话框的两种大小

这个对话框使用了 3 种布局：一个 QVBoxLayout 组合了按钮，一个 QHBoxLayout 组合了国家列表和那组按钮，一个 QVBoxLayout 组合了“Select a country”标签和剩下的部件。一个延伸项目用来维护 Cancel 和 Help 按钮间的距离。

下面的代码创建了对话框部件和布局：

```
QVBoxLayout *buttonBox=new QVBoxLayout(6);
buttonBox->addWidget(new QPushButton("OK",this));
buttonBox->addWidget(new QPushButton("Cancel",this));
buttonBox->addStretch(1);
buttonBox->addWidget(new QPushButton("Help",this));
QListBox *countryList=new QListBox(this);
countryList->insertItem("Canada");
/*...*/
countryList->insertItem("United States of America");
QHBoxLayout *middleBox=new QHBoxLayout(11);
middleBox->addWidget(countyList);
middleBox->addLayout(buttonBox);
QVBoxLayout *topLevelBox=new QVBoxLayout(this,6,11);
topLevelBox->addWidget(new QLabel("Select a country",this));
topLevelBox->addLayout(middleBox);
```

可以看到，Qt 让布局变得非常容易。

（3）自定义布局

通过子类化 QLayout 开发者可以定义自己的布局管理器。和 Qt 一起提供的 customlayout 样例展示了三个自定义布局管理器：BorderLayout、CardLayout 和 SimpleFlow，程序员可以使用并修改它们。

Qt 还包括 QSplitter，是一个最终用户可以操纵的分离器。某些情况下，QSplitter 可能比布局管理器更为可取。

为了完全控制，重新实现每个子部件的 QWidget::resizeEvent()并调用 QWidget::setGeometry()，就可以在一个部件中手动地实现布局。

2．Qt/Embedded 图形设计器

Qt 图形设计器是一个具有可视化用户接口的设计工具。Qt 的应用程序可以完全用源代码来编写，或者使用 Qt 图形设计器来加速开发工作。启动 Qt 图形设计器的方法是：

```
cd qt-2.3.2/bin
./designer
```

这样就可以启动一个图形化的设计界面，如图 12.14 所示。

开发者单击工具栏上的代表不同功能的子窗体/组件的按钮，然后把它放到一个表单上，这样就可以把一个子窗体/组件放到表单上了。开发者可以使用属性对话框来设置子窗体的属性，精确地设置子窗体的位置和尺寸大小是没必要的。开发者可以选择一组窗体，然后对他们进行排列。例如，我们选定了一些按钮窗体，然后使用“水平排列（lay out horizontally）”选项对它们进行一个接一个的水平排列。这样做不仅使得设计工作变得更快，而且完成后的窗体将能够按照属性设置的比例填充窗口的可用尺寸范围。

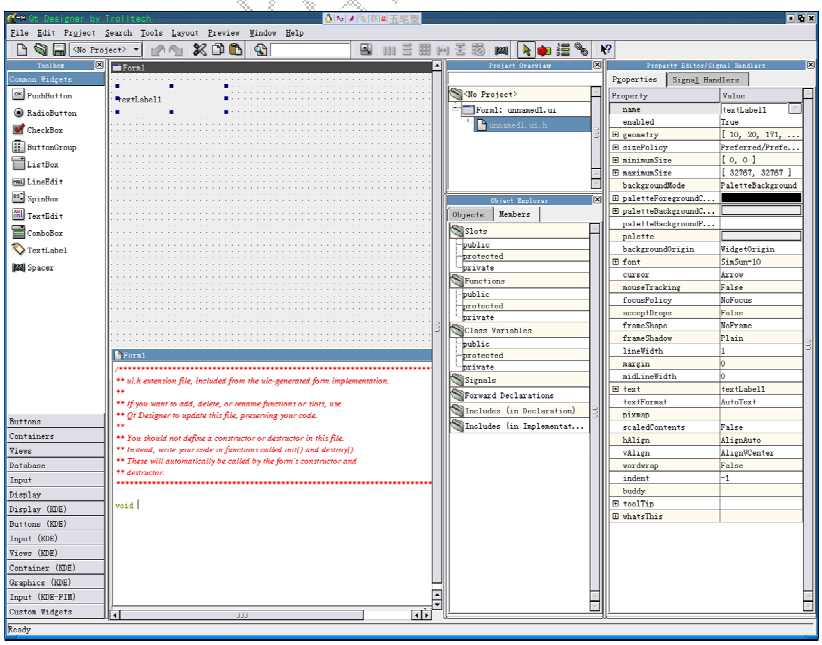

图 12.14　Qt 图形设计器界面

使用 Qt 图形设计器进行图形用户接口的设计可以消除应用的编译、链接和运行时间，同时使修改图形用户接口的设计变得更容易。Qt 图形设计器的预览功能可以使开发者能够在开发阶段看到各种样式的图形用户界面，也包括客户样式的用户界面。通过 Qt 集成功能强大的数据库类，Qt 图形设计器还可提供生动的数据库数据浏览和编辑操作。

开发者可以建立同时包含有对话框和主窗口的应用，其中主窗口可以放置菜单，工具栏，旁述帮助等的子窗口部件。Qt 图形设计器提供了几种表单模板，如果窗体会被多个不同的应用反复使用，那么开发者也可建立自己的表单模板以确保窗体的一致性。

Qt 图形设计器使用向导来帮助人们更快更方便地建立包含有工具栏、菜单和数据库等方面的应用。程序员可以建立自己的客户窗体，并把它集成到 Qt 图形设计器中。

Qt 图形设计器设计的图形界面以扩展名为“ui”的文件进行保存，这个文件有良好的可读性，这个文件可被 uic（Qt 提供的用户接口编译工具）编译成为 C++的头文件和源文件。Qmake 工具在它为工程生成的 Makefile 文件中自动包含了 uic 生成头文件和源文件的规则。

另一种可选的做法是在应用程序运行期间载入 ui 文件，然后把它转变为具备原先全部功能的表单。这样开发者就可以在程序运行期间动态地修改应用的界面，而不需重新编译应用，另一方面，也使得应用的文件尺寸减小了。

3．建立对话框

Qt 为许多通用的任务提供了现成的包含了实用的静态函数的对话框类，主要有以下几种。

- QmessageBox 类：是一个用于向用户提供信息或是给用户进行一些简单选择（例如“yes”或“no”）的对话框类。如图 12.15 所示。
- progressDialog 类：包含了一个进度栏和一个“Cancel”按钮。如图 12.16 所示。
- Qwizard 类：提供了一个向导对话框的框架。如图 12.17 所示。

图 12.15　QmessageBox 类对话框

图 12.16　progressDialog 类对话框

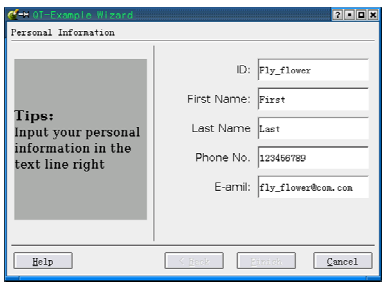

图 12.17　Qwizard 类对话框

另外，Qt 提供的对话框还包括 QcolorDialog、QfileDialog、QFontDialog 和 QprintDialog。这些类通常适用于桌面应用，一般不会在 Qt/Embedded 中编译使用它们。

12.3 实验内容—使用 Qt 编写“Hello，World”程序

1．实验目的

通过编写一个跳动的“Hello,World”字符串，进一步熟悉嵌入式 Qt 的开发过程。

2．实验步骤

（1）生成一个工程文件（.pro 文件）

使用命令 progen 产生一个工程文件（progen 程序可在 tmake 的安装路径下找到）。

如下所示：

```
progen -t app.t -o hello.pro
```

那样产生的 hello.pro 工程文件并不完整，开发者还需添加工程所包含的头文件，源文件等信息。

（2）新建一个窗体

启动 Qt 图形编辑器，使用如下命令：

```
./designer（该程序在 qt-2.3.x for x11 的安装路径的 bin 目录下）
```

接着单击编辑器的“new”菜单，弹出了一个“new Form”对话框，在这个对话框里选择“Widget”，然后单击“OK”按钮，这样就新建了一个窗体。

接下来再对这个窗体的属性进行设置，注意把窗体的“name”属性设为“Hello”；窗体的各种尺寸设为宽“240”，高“320”，目的是使窗体大小和 FS2410 带的显示屏的大小一致；窗体背景颜色设置为白色。具体设置如图 12.18 所示。

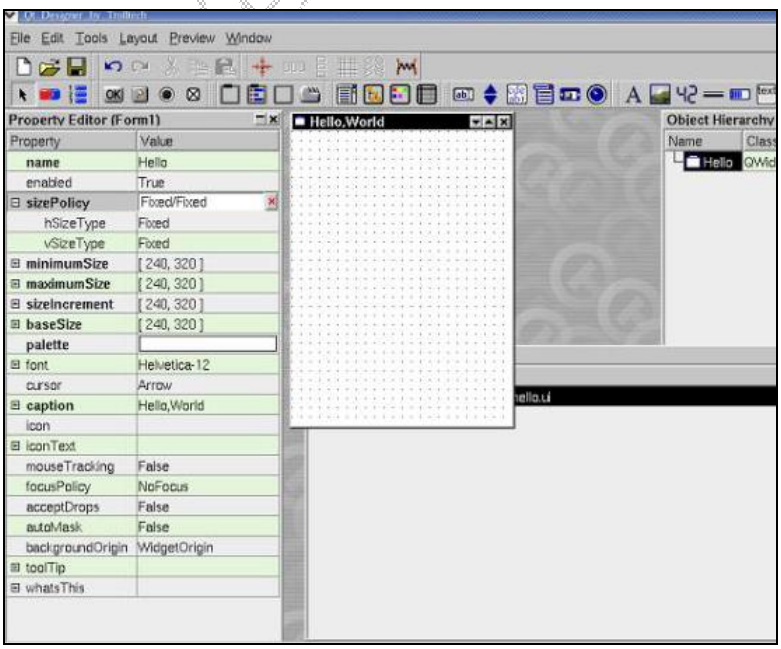

图 12.18　Hello 窗体的属性设置

设置完成后，将其保存为 hello.ui 文件，这个文件就是 Hello 窗体的界面存储文件。

（3）生成 Hello 窗体类的头文件和实现文件

下面根据上述的界面文件 hello.ui 使用 uic 工具产生 Hello 窗体类的头文件和实现文件，具体方法是：

```
cd qt-2.3.7/bin
uic –o hello.h hello.ui
uic –o hello.cpp –impl hello.h hello.ui
```

这样就得到了 Hello 窗体类的头文件 hello.h 和实现文件 hello.cpp。下面就可以根据需要实现的具体功能，在 hello.cpp 文件里添加相应的代码。

比如要在 Hello 的窗体上显示一个动态的字符串“Hello，World”，那么使用需要重新实现 paintEvent（QPaintEvent *）方法，同时还需要添加一个定时器 Qtimer 实例，以周期性刷新屏幕，从而得到动画得效果。下面是修改后的 hello.h 和 hello.cpp 文件。

```
/****************************************************************************
** 以下是 hello.h 的代码
****************************************************************************/
#ifndef HELLO_H
#define HELLO_H
#include <qvariant.h>
#include <qwidget.h>
class QVBoxLayout;
class QHBoxLayout;
class QGridLayout;
class Hello : public QWidget
{
Q_OBJECT
public:
Hello( QWidget* parent = 0, const char* name = 0, WFlags fl = 0 );
~Hello();
//以下是手动添加的代码
signals:
void clicked();
protected:
void mouseReleaseEvent( QMouseEvent * );
void paintEvent( QPaintEvent * );
private slots:
void animate();
private:
QString t;
```

```
int b;
};
#endif // HELLO_H
/*****************************************************************************
** 以下是 hello.cpp 源代码
*****************************************************************************/
#include "hello.h"
#include <qlayout.h>
#include <qvariant.h>
#include <qtooltip.h>
#include <qwhatsthis.h>
#include <qpushbutton.h>
#include <qtimer.h>
#include <qpainter.h>
#include <qpixmap.h>
/*
* Constructs a Hello which is a child of 'parent', with the
* name 'name' and widget flags set to 'f'
*/
Hello::Hello( QWidget* parent, const char* name, WFlags fl )
: QWidget( parent, name, fl )
{
if ( !name )
setName( "Hello" );
resize( 240, 320 );
setMinimumSize( QSize( 240, 320 ) );
setMaximumSize( QSize( 240, 320 ) );
setSizeIncrement( QSize( 240, 320 ) );
setBaseSize( QSize( 240, 320 ) );
QPalette pal;
QColorGroup cg;
cg.setColor( QColorGroup::Foreground, black );
cg.setColor( QColorGroup::Button, QColor( 192, 192, 192) );
cg.setColor( QColorGroup::Light, white );
cg.setColor( QColorGroup::Midlight, QColor( 223, 223, 223) );
cg.setColor( QColorGroup::Dark, QColor( 96, 96, 96) );
cg.setColor( QColorGroup::Mid, QColor( 128, 128, 128) );
cg.setColor( QColorGroup::Text, black );
cg.setColor( QColorGroup::BrightText, white );
```

```
    cg.setColor( QColorGroup::ButtonText, black );
    cg.setColor( QColorGroup::Base, white );
    cg.setColor( QColorGroup::Background, white );
    cg.setColor( QColorGroup::Shadow, black );
    cg.setColor( QColorGroup::Highlight, black );
    cg.setColor( QColorGroup::HighlightedText, white );
    pal.setActive( cg );
    cg.setColor( QColorGroup::Foreground, black );
    cg.setColor( QColorGroup::Button, QColor( 192, 192, 192) );
    cg.setColor( QColorGroup::Light, white );
    cg.setColor( QColorGroup::Midlight, QColor( 220, 220, 220) );
    cg.setColor( QColorGroup::Dark, QColor( 96, 96, 96) );
    cg.setColor( QColorGroup::Mid, QColor( 128, 128, 128) );
    cg.setColor( QColorGroup::Text, black );
    cg.setColor( QColorGroup::BrightText, white );
    cg.setColor( QColorGroup::ButtonText, black );
    cg.setColor( QColorGroup::Base, white );
    cg.setColor( QColorGroup::Background, white );
    cg.setColor( QColorGroup::Shadow, black );
    cg.setColor( QColorGroup::Highlight, black );
    cg.setColor( QColorGroup::HighlightedText, white );
    pal.setInactive( cg );
    cg.setColor( QColorGroup::Foreground, QColor( 128, 128, 128) );
    cg.setColor( QColorGroup::Button, QColor( 192, 192, 192) );
    cg.setColor( QColorGroup::Light, white );
    cg.setColor( QColorGroup::Midlight, QColor( 220, 220, 220) );
    cg.setColor( QColorGroup::Dark, QColor( 96, 96, 96) );
    cg.setColor( QColorGroup::Mid, QColor( 128, 128, 128) );
    cg.setColor( QColorGroup::Text, black );
    cg.setColor( QColorGroup::BrightText, white );
    cg.setColor( QColorGroup::ButtonText, QColor( 128, 128, 128) );
    cg.setColor( QColorGroup::Base, white );
    cg.setColor( QColorGroup::Background, white );
    cg.setColor( QColorGroup::Shadow, black );
    cg.setColor( QColorGroup::Highlight, black );
    cg.setColor( QColorGroup::HighlightedText, white );
    pal.setDisabled( cg );
    setPalette( pal );
    QFont f( font() );
```

```
f.setFamily( "adobe-helvetica" );
f.setPointSize( 29 );
f.setBold( TRUE );
setFont( f );
setCaption( tr( "" ) );
//以下是手动添加的代码
t = "Hello,World";
b = 0;
QTimer *timer = new QTimer(this);
connect( timer, SIGNAL(timeout()), SLOT(animate()) );
timer->start( 40 );
}
/*
* Destroys the object and frees any allocated resources
*/
Hello::~Hello()
{
}
//以下至结尾是手动添加的代码
void Hello::animate()
{
b = (b + 1) & 15;
repaint( FALSE );
}
/*
Handles mouse button release events for the Hello widget.
We emit the clicked() signal when the mouse is released inside
the widget.
*/
void Hello::mouseReleaseEvent( QMouseEvent *e )
{
if ( rect().contains( e->pos() ) )
emit clicked();
}
/* Handles paint events for the Hello widget.
Flicker-free update. The text is first drawn in the pixmap and the
pixmap is then blt'ed to the screen.
*/
void Hello::paintEvent( QPaintEvent * )
```

```
{
static int sin_tbl[16] = {
0, 38, 71, 92, 100, 92, 71, 38, 0, -38, -71, -92, -100, -92, -71, -38};
if ( t.isEmpty() )
return;
// 1: Compute some sizes, positions etc.
QFontMetrics fm = fontMetrics();
int w = fm.width(t) + 20;
int h = fm.height() * 2;
int pmx = width()/2 - w/2;
int pmy = height()/2 - h/2;
// 2: Create the pixmap and fill it with the widget's background
QPixmap pm( w, h );
pm.fill( this, pmx, pmy );
// 3: Paint the pixmap. Cool wave effect
QPainter p;
int x = 10;
int y = h/2 + fm.descent();
int i = 0;
p.begin( &pm );
p.setFont( font() );
while ( !t[i].isNull() ) {
int i16 = (b+i) & 15;
p.setPen( QColor((15-i16)*16,255,255,QColor::Hsv) );
p.drawText( x, y-sin_tbl[i16]*h/800, t.mid(i,1), 1 );
x += fm.width( t[i] );
i++;
}
p.end();
// 4: Copy the pixmap to the Hello widget
bitBlt( this, pmx, pmy, &pm );
}
```

（4）编写主函数 main()

一个 Qt/Embeded 应用程序应该包含一个主函数，主函数所在的文件名是 main.cpp。主函数是应用程序执行的入口点。以下是“Hello,World”例子的主函数文件 main.cpp 的实现代码：

```
/*****************************************************************************
** 以下是 main.cpp 源代码
```

```
****************************************************************************/
#include "hello.h"
#include <qapplication.h>
/*
The program starts here. It parses the command line and builds a message
string to be displayed by the Hello widget.
*/
#define QT_NO_WIZARD
int main( int argc, char **argv )
{
QApplication a(argc,argv);
Hello dlg;
QObject::connect( &dlg, SIGNAL(clicked()), &a, SLOT(quit()) );
a.setMainWidget( &dlg );
dlg.show();
return a.exec();
}
```

（5）编辑工程文件 hello.pro 文件

到目前为止，为 Hello,World 例子编写了一个头文件和两个源文件，这 3 个文件应该被包括在工程文件中，因此还需要编辑 hello.pro 文件，加入这 hello.h,hello.cpp,main.cpp 这三个文件名。具体定义如下：

```
/****************************************************************************
** 以下是 hello.pro 文件的内容
****************************************************************************/
TEMPLATE = app
CONFIG = qt warn_on release
HEADERS = hello.h
SOURCES = hello.cpp \
main.cpp
INTERFACES =
```

（6）生成 Makefile 文件

编译器是根据 Makefile 文件内容来进行编译的，所以需要生成 Makefile 文件。Qt 提供的 tmake 工具可以帮助我们从一个工程文件（.pro 文件）中产生 Makefile 文件。结合当前例子，要从 hello.pro 生成一个 Makefile 文件的做法是首先查看环境变量$TMAKEPATH 是否指向 arm 编译器的配置目录，在命令行下输入以下命令：

echo $TMAKEPATH

如果返回的结果末尾不是…/qws/linux-arm-g++的字符串，那您需要把环境变量 $TMAKEPATH 所指的目录设置为指向 arm 编译器的配置目录，过程如下，

export TMAKEPATH = /tmake 安装路径/qws/linux-arm-g++

同时，应确保当前的 QTDIR 环境变量指向 Qt/Embedded 的安装路径,如果不是，则需要执行以下过程。

export QTDIR = ……/qt-2.3.7

上述步骤完成后，就可以使用 tmake 生成 Makefile 文件，具体做法是在命令行输入以下命令：

tmake –o Makefile hello.pro

这样就可以看到当前目录下新生成了一个名为 Makefile 的文件。下一步，需要打开这个文件，做一些小的修改。

1）将 LINK = arm-linux-gcc 这句话改为：

LINK = arm-linux-g++

这样做是因为要是用 arm-linux-g++进行链接。

2）将 LIBS = $(SUBLIBS) -L$(QTDIR)/lib -lm – lqte 这句话改为：

LIBS = $(SUBLIBS) -L/usr/local/arm/2.95.3/lib -L$(QTDIR)/lib -lm –lqte

这是因为链接时要用到交叉编译工具 toolchain 的库。

（7）编译链接整个工程

最后就可以在命令行下输入 make 命令对整个工程进行编译链接了。

make

make 生成的二进制文件 hello 就是可以在 FS2410 上运行的可执行文件。

本 章 小 结

本章主要讲解了嵌入式 Linux 的图形编程。首先介绍了几种常见的嵌入式图形界面编程机制，并给出了它们之间的相互关系。

接下来，本章介绍了 Qt/Embedded 了开发入门，包括环境的搭建、信号与插槽的概念与应用以及图形设计器的应用。

本章的实验介绍了如何使用 Qt 编写“Hello，World”小程序，从中可以了解到 Qt 编程过程的全过程。