

LAZARUS

实战宝典



www.fpcen.com

序

随着 IT 的快速发展，编程面向的系统日益多元化，程序员们希望能有一个快速应用程序开发（RAD）工具，能以熟悉的语言、易用的集成开发环境（IDE）编写能运行于多种平台的应用程序。Lazarus 正是这种需求下诞生的。

Lazarus 是以 Free Pascal 为编译器，以“一次编写、到处编译”为目标的集成开发环境。Lazarus 实现了 Free Pascal 所缺少的图形界面库，并且基本上 Free Pascal 支持的 CPU 与 OS，Lazarus 基本都能支持。在 Lazarus 中，你可以像 Delphi 等工具一样事件驱动式地可视化开发，拖几个控件，设置一下属性，然后双击控件编写事件代码就完成了应用编程。

Free Pascal 是一个 Pascal 语言的编译器，支持面向对象开发，高度兼容 DELPHI 语法，并且可以用同样的代码编译出在多种 CPU（如 i386/x64/arm）与多种 OS（如 Win32/Win64/WinCE/Linux/BSD/MAC）上运行的程序。Free Pascal 有丰富的运行库，而且绝大部分代码都是开源并且允许用于商业开发的。全国青少年信息学奥林匹克竞赛使用的 Pascal 编译器就是 Free Pascal。

采用 Lazarus 进行编程，相对其它工具，是比较有优势的：

1、免费、开源、可用于商业开发，这意味着你不需要花几万块买 DELPHI、也不需要几千块买 VS 就可以开发商业应用，如果发现编译器或运行库有错误，你还可以找出源码修正它；

2、基于 Pascal 并且高度兼容 DELPHI，Pascal 的语法是容易理解、语法优美的，很多算法的书都是用 Pascal 写的，如果你有用 DELPHI 写的代码，那么可以稍为修改甚至不用任何修改就可用于 Free Pascal；

3、国内已经有了中文社区网站与多个 QQ 交流群，这有助于入门或互相讨论解决难题；

4、这是第一个支持 WinCE 开发的 Pascal 工具！这是第一个支持 Win64 开发的 Pascal 工具！在很多领域，都远超商业开发软件 DELPHI，甚至 DELPHI 的一些版本，还要使用 Free Pascal 的编译器。在应用开发领域，领先一步的好处不言而喻。

移动应用开发逐渐流行，Lazarus 除了支持 WinCE，也开始加入 IOS 应用开发与安卓应用开发的支持，虽然目前完成度还不是很高，用于商业快速开发还比较困难，但毕竟成为了可能，未来肯定可以像 WinCE 那样比较完美地支持。

我在 2007 年使用 Lazarus 编写了一个公司的应用程序，运行于 Windows Mobile，到 2009 年有 300 用户；后来 Windows Mobile 没落，在 2010 年界面略加修改、编译为车载 WinCE

应用，在 2011 年曾有 1500 用户，直到目前也有 500 用户；在 2012 年 3 月我将该应用核心代码编译为安卓的库，然后仅是通过 JAVA 编写了界面，就重现了大部分功能，目前有近 3000 用户。从我编写这个应用的发展经历来看，使用 Lazarus 的优势可略见一斑。

本书作为 Lazarus 的第一本中文图书，可为广大程序员或爱好者提供入门的捷径，也可以为大家解决一些开发上的疑难，值得认真阅读。愿本书助力开发者抓住多平台的发展机遇，共享成功。

东兰梦舞

2013-12-20

编者语

1、书的起源

笔者在 Delphi 日渐坠落的时候，偶然发现了 Lazarus，通过一段时间的接触，感觉它可以替代 Delphi，于是开始深入学习，加入 Lazarus 中文社区，拜访前辈..... 原本想以札记的形式记录学习 Lazarus 的历程，后来发现有不少网友同样苦于学习 Lazarus 而无中文资料，遂产生了编写点东西给初学者参考的念头。

Lazarus 是新生的工具，在学习和使用的过程中是充满血与泪的。笔者编写的章节，都是经过实战，多次验证的，不是剪刀加浆糊的流水线产品，所以我们的速度可能很慢，但都是汗水的结晶。写《LAZARUS 实战宝典》，其实也是在充当 Lazarus 的开荒牛角色，我们的目的就是给初学者止止血、抹把眼泪。

2、书的构成

《LAZARUS 实战宝典》，并不是一部纯粹的软件手册，它立足于 lazarus 实战的经验，着眼于怎样解决 lazarus 使用中碰到的问题，尽可能地起到画龙点睛的作用；它面向具有 Pascal 语言基础的读者；本书的 V1.0 是以 lazarus 0.9.28 为蓝本进行编写。

本书分为 4 大篇章：基础篇，提高篇，应用篇，实例篇。

基础篇：以 lazarus 官方网站的资源为素材，阐述 Lazarus 的语法基础，IDE 的使用等。

提高篇：这个是针对初学者进阶学习而设计的。覆盖 Lazarus 的控件（包括可视、不可视的），数据库，线程等方面。

应用篇：讲解 Lazarus 在各种环境中的使用方法，如 wince, linux, android 等等。

实例篇：收集 Lazarus 在各行各业中应用的例子，展示 lazarus 的创造力。

本书 V1.0 提供的章节：

基础篇	1、绪论 2、Lazarus 用户指南 3、Lazarus IDE
提高篇	1、控件的制作 2、消息编程
应用篇	1、WINCE 应用 2、使用 Lazarus 进行 Linux 开发
实例篇	1、文字图案生成器

3、未来展望

通过 Lazarus 的官方网站，我们可以看到 lazarus 更新很快，这是新技术出现的迫切需要，也是不断完善自身的体现。

本书未来的 V2.0，将跟随 Lazarus 的变化，更新相应的内容，增大涉及面，采集更多的例子，逐步完善预定的 4 大篇章。同时也为读者提供一些实用的工具。

4、感谢板

《LAZARUS 实战宝典》的诞生，首先应该感谢 Lazarus 中文社区的站长熊文彬（网名：猫工），是他的坚定信念和不懈努力，www.fpccn.com 成为了 Lazarus 的中文官方网站，并且为《LAZARUS 实战宝典》提供了生存的平台和发展的空间；同时创建了多个 QQ 交流群（见中文官方网站）。

参与《LAZARUS 实战宝典》编写的作者均来自 Lazarus 中文社区，他们是：

姜元东：Lazarus 中文社区的版主之一，也是《LAZARUS 实战宝典》的创建者之一，为社区贡献了很多代码。社区网名：猫者，Email: lazarus.com.cn@gmail.com

沈雪华：对 Lazarus 有比较深的了解，尤其熟悉 Lazarus 的 Android 环境，Pascal4andriod 出自其手。社区网名：truetom，Email: 1339838080@qq.com

杨晓峰：一个低调的高手，他写的 Linux 编程章节，是《LAZARUS 实战宝典》中最具份量的章节，重磅级别的作品。社区网名：stlxv，Email: n.akr.akiiya@gmail.com

李建中：《LAZARUS 实战宝典》的发起者，也是创建者之一。社区网名：逍遥派掌门人，Email: cnlazarus@163.com

在此亦感谢 Lazarus 中文社区众多网友的直接或间接的支持，给《LAZARUS 实战宝典》提供帮助或建议。

如果觉得本书对您有帮助，并且愿意赞助 lazarus 中文社区开展此项目的，请点：

[赞助 lazarus 中文社区](#)

一句话

为感谢大家一直对本社区和 Lazarus 的支持，站长猫工特意在 QQ 交流群里开辟了群广场，大家可以在群广场对本书和 Lazarus 进行一句话的总结发言。以下便是从中采集到的：

“Lazarus 我很喜欢，社区和群给了我工作很大帮忙，谢谢！”

“喔，期待已久，pascal，仍是最爱。”

“不用 lazarus 的 delphier 不是好 delphier”

“终于能有一本比较正规的 Lazarus 的中文教程了，谢谢作者们！”

“Lazarus 跨平台，我的最爱。”

“用了 Lazarus 以后，再也不想用 Delphi 了！”

“现今最需要的就是跨平台的快速开发工具了，Lazarus 就是！”

“Lazarus 实战宝典，前人栽树，后人乘凉”

“深入浅出非常屌，承前启后十分高。”

“Lazarus 未来的选择”

目 录

章节内容	页码	作者
基础篇		
第 1 章 绪 论		姜元东
1.1 LAZARUS 的简介	
1.1.1 什么是 Free Pascal	
1.1.2 LAZARUS 的来源	
1.1.3 LAZARUS 名称的来源	
1.1.4 什么是 LAZARUS	
1.2 使用 LAZARUS 可开发的应用	
控制台应用程序	
动态链接库	
GUI 程序	
1.3 LAZARUS 的主要特点	
1.4 LAZARUS 的架构	
第 2 章 用户指南		姜元东
2.1 LAZARUS 相关问题解答	
2.1.1 Lazarus 快速问答 FAQ	
2.2 LAZARUS 的安装	
2.2.1 获得 Lazarus 的相应版本	
2.2.2 安装 Lazarus	
2.3 程序语言介绍	
2.3.1 基本类型	

章节内容	页码	作者
第 3 章 Lazarus IDE （集成开发环境）		沈雪华
3.1 IDE 介绍	
3.1.1 主窗体	
3.1.1.1 主菜单	
3.1.1.2 快速按钮工具栏	
3.1.1.3 组件面板	
3.1.2 对象观察器	
3.1.3 窗体设计器	
3.1.4 源代码编辑器	
3.1.5 消息窗口	
3.2 IDE 技巧	
3.2.1 把开发环境改为简体中文版	
3.2.2 创建新的文件	

提高篇

第 1 章 控件制作		李建中
1.1 控件的认识	
1.2 控件的创建	
1.2.1 需求分析	
1.2.2 创建控件	
1.2.3 创建控件图标	
1.2.4 编写控件代码	
1.2.5 安装控件	
1.3 控件的使用	

章节内容	页码	作者
第 2 章 消息编程		李建中
2.1 消息类	
2.1.1 消息的构成	
2.1.2 消息的句柄	
2.1.3 消息的传送	
2.1.4 消息的取值	
2.1.5 队列消息和非队列消息	
2.2 自定义消息	
2.2.1 自定义消息的创建	
2.2.2 自定义消息的例子	

应用篇

第 1 章 linux 应用		杨晓峰
1.1 在 Linux 下搭建 Lazarus 开发环境	
1.1.1 Free Pascal 编译器的安装	
1.1.2 安装 Lazarus	
1.1.2.1 从软件仓库安装 Lazarus	
1.1.2.2 从源代码编译安装	
1.1.2.2.1 编译前的准备	
1.1.2.2.2 编译	
1.1.2.2.3 启动 Lazarus	
1.2 实战：简单的 Linux 应用程序	
1.2.1 第一个 Linux 桌面程序	
1.2.2 调试	

章节内容	页码	作者
1.2.2.1 在 Lazarus 开发环境里调试	杨晓峰
1.2.2.2 单独使用 gdb 进行调试	
1.2.3 脱离 Lazarus 来编译你的应用程序	
1.2.3.1 使用 Lazbuild 进行编译	
1.2.3.2 使用 fpc 进行编译	
1.3 LCL 的 GUI 后端	
1.3.1 切换到 Qt4 GUI 后端	
1.3.1.1 重新编译前的准备	
1.3.1.2 重新编译 Lazarus	
1.3.1.2 重新编译 project1	
1.3.2 切换到 gtk2 后端	
1.3.3 直接调用 GUI 后端	
1.3.3.1 实例:一个简单的 QT4 应用程序	
1.3.3.2 实例:一个简单的 GTK2 应用程序	
1.4 使用 Linux 系统调用	
1.5 编写和使用库	
1.5.1 实例:静态库 libmydemo.a	
1.5.2 实例:使用 Free Pascal 调用 libmydemo.a	
1.5.3 实例:编写 C 版本的 libmydemo.a	
1.5.4 实例:使用 C 语言调用 libmydemo.a	
1.5.5 实例:将 libmydemo.a 转换成动态库 libmydemo.so	
1.5.6 实例:使用 Free Pascal 调用 libmydemo.so	
1.5.7 实例:使用 C 语言调用 libmydemo.so	
1.5.8 一些补充	
1.6 发布你的程序	
1.6.1 使用源代码进行发布	
1.6.2 发布编译好的应用程序	

章节内容	页码	作者
第 2 章 WINCE 应用		李建中
2.1 WINCE 运行环境的建立	
2.1.1 建立 WINCE 编译环境	
2.1.2 建立 WINCE 运行环境	
2.2 创建第一个例子	
2.2.1 建立 LAZARUS 的 WINCE 工程	
2.2.2 编译 LAZARUS 的 WINCE 工程	
2.2.3 运行效果	
2.3 KOL-CE 组件在 WINCE 中的应用	
2.3.1 KOL-CE 组件的安装	
2.3.2 使用 KOL-CE 组件创建程序	
2.4 WINCE 程序的调试技术	
2.4.1 使用 WINCE 模拟器的调试技术	
2.4.2 使用真实机器的调试技术	
2.5 WINCE 编程点滴	
实例篇		
第 1 章 文字图案生成器	李建中
1.1 原理	
1.2 关键代码	
1.3 界面效果	

第 1 章

绪 论

1.1 LAZARUS 的简介

1.1.1 什么是 Free Pascal

Free Pascal (FPC) 是一个开源的 Pascal 编译器。它有着以下两个显著特点：高度的 Delphi 兼容性，和在多种操作系统——包括 Windows, Mac OS X 和 Linux——上的可用性。

Free Pascal 与 Delphi 的兼容不仅是因为它与 Delphi 同样使用 Object Pascal 语言,而且还因为它提供了许多与 Delphi 相同的、功能强大的例程 (routines) 和 类 (classes)。这包括了许多我们熟悉的单元,例如 System、SysUtils、StrUtils、DateUtils、Classes、Variants、Math、IniFiles 和 Registry。并且,而且这些单元在所有支持的平台上都能使用。

当然,Free Pascal 也提供了像 Windows、ShellAPI、BaseUnix、Unix 和 DynLibs 这样的单元,用于使用特定操作系统的功能。所有这些单元组成了通常被称作“Free Pascal 运行时库”(run-time library, RTL) 的核心。

1.1.2 LAZARUS 的来源

曾经的 delphi 作为一个时代的产品,其辉煌已经过去,作为 pascal 的忠实 fans 陷入迷茫。随着 LAZARUS 的闪亮登场,一个崭新的局面到来了。

Lazarus 是从 1999 年 2 月开始的,成立时的主要成员是这三个人: Cliff Baeseman, Shane Miller, Michael A. Hess, 当时,他们三个曾经为之努力的 megido 计划(megido 计划致力于打造一个开源、跨平台、可视化的 Object Pascal 快速应用开发环境)由于种种原因被解散。在挫折面前他们并不气馁,决定发起 Lazarus 计划。在随后的几年中,这个计划得到了稳步发

展，引起很多人的关注并拥有了一群稳定的支持者和开发者。遗憾的是，上述三个创始人中，只有 Michael A. Hess 仍在参与这项计划。开发组中另一个元老是 Marc Weustink，他在 1999 年 8 月就参与这个项目。在他之后的是 2000 年 9 月加入的 Mattias Gaertner，他们两人一直是核心代码的主要编写者，是他们的共同努力让 Lazarus 变得成熟。

1.1.3 LAZARUS 名称的来源

Lazarus 一词是 Eleazar 的拉丁文写法，本意是“神是我的帮助”，来源于圣经人物，是耶稣的朋友。Lazarus 在死后第三天被耶稣从坟墓中唤醒复活(《圣经·约翰福音》第 14 章 44 节)。

原来的项目叫 Megido (尝试建立跨平台的 Delphi 克隆)，但是这个努力失败了，众所周知，Lazarus 曾经拯救过基督，所以，项目取名 Lazarus，因为她的出现拯救了 Megido。

1.1.4 什么是 LAZARUS

Lazarus 的设计目标是应用 Free Pascal，所以所有凡是 Free Pascal 能运行的平台，Lazarus 都可以运行。最新版本能运行于 Windows, Mac OS X, and Linux 系统中。整个界面的外观和操作和 Delphi IDE 一样，因此，如果你会使用 Delphi 的话，用起 Lazarus IDE 来就一定能得心应手了。

Lazarus 是一个用于 Free Pascal 的快速应用开发 (RAD) 的面向对象的 Pascal 集成开发环境 (IDE)。Lazarus 对于窗口管理来说是中性的。可以工作在 KDE 下，也可以工作在 GNOME 或其他窗口管理器 (MVM、WindowMaker)。目前，已提供 32 位和 64 位版本支持。Lazarus 的工作界面、外观和操作和 Borland 的 Delphi IDE 非常相似，所不同的是 Lazarus 是完全的自由软件。Lazarus 可以直接移植 Delphi 的代码。Lazarus 的编程语言是以 Pascal 为基础的。

Pascal 语言具有可读性好、编写容易的特点，这使得它很适合作为基础的开发语言。同

时，使用编译器创建的应用程序只生成单个可执行文件(.EXE，但生成的可执行文件体积相对 Delphi 的来说有点大，只包含一个空窗体的工程生成的可执行文件就达到了 10 多 M。这里，可以通过编译选项来减小可执行文件的大小，可以减为 1M 多点，然后通过 UPX 压缩，可以减为 600 多 K。)。正是这种结合，使得 Pascal 成为 Lazarus 这种先进开发环境的编程语言。

由于 Lazarus 为开放的 IDE，且在 linux 下表现良好，目前被中国计算机学会指定为 NOI 系列竞赛的 Pascal 语言推荐 IDE。

1.2 使用 LAZARUS 可开发的应用

控制台应用程序

控制台（console）程序不提供 GUI，而是在控制台中启动，并在其中进行输入 / 输出（input/output, I/O)的。在 Windows 中，控制台通常被称作“命令提示符窗口”(command prompt window)，而在 Mac OS X 和 Linux 中则被称作“终端窗口”（terminal window）。一些小工具（utilities），如 Windows **FC**（file compare，文件比较）程序、UNIX 上的 **cd** 和 **cp** 命令等，都是控制台程序。当然，控制台程序也可能是功能强大的数值计算、建模或数据处理程序——它们不需要引人注目的 GUI，因为他们可由其他程序启动，或者可由批处理（batch）文件（或 UNIX / Linux 上的外壳脚本（shell scripts））调用。

Free Pascal 编译器及其包括的工具程序都是控制台程序。这意味着它们可以在控制台中运行，由批处理文件调用，或者在 Lazarus IDE 中启动。要创建命令行程序，您事实上只用一个文本编辑器和 Free Pascal 编译器就能做到，而不必使用 Lazarus。当然，如果您愿意，在 Lazarus 中也可以创建、编辑、编译和调试命令行程序。

动态链接库

动态链接库（dynamically loaded library, DLL）通常是一组已编译函数的集合，这些函数可被其他程序调用。顾名思义，使用 DLL 的程序并非在编译时将其链接到其中，而是在运行时动态地加载。这类文件在 Windows 中通常拥有 .dll 后缀，在 Mac OS X 上为 .dylib（表示 dynamic shared library，动态共享库），而在 Linux 上则是 .so（表示 shared object library，共享目标程序库）。

动态链接库一般被用于开发程序的插件（add-ons）、开发可被用其他语言（如 C 和 C++）编写的程序调用的函数库，或者用于将大型的项目“化整为零”，使不同开发者能独立开发项目

的某一部分。Windows 本身就是由上百个 DLL 组成的。其他一些大型应用程序，如 OpenOffice.org，亦是如此。

像控制台程序一样，您同样只需要使用文本编辑器和 Free Pascal 编译器就可以创建 DLL；同样，您也可以使用 Lazarus 来创建、编译和调试您的 DLL。

值得注意的是，在 Windows 上，DLL 有时被误认为过于复杂，而使系统变得不稳定。通常可能是因为安装方式不当，而不是 DLL 本身的问题。事实上，程序与 DLL 之间的数据交换通常是基于标准的简单数据类型（而不是某些语言专有的对象或结构）的，而这就迫使程序员更多地注意他们在做什么。如果做得对，就会得到更好、更稳定的程序。

GUI 程序

我们每天使用的多数程序都是 GUI 程序，包括字处理程序（word processors）、Web 浏览器、电子表格（spreadsheet）程序，甚至许多开发工具。比如，Lazarus 和 Delphi 都是功能强大的 GUI 程序的典范。

在使用 Lazarus 开发 GUI 程序时，您不只编写 Pascal 单元中的代码，而且还需设计窗体（forms）。窗体上可以放置像按钮、列表框之类的可视控件，也可以放置一些非可视控件（non-visual controls）。像 Delphi 一样，在 Lazarus 中的窗体设计过程也是可视化的，控件属性的设置可以在 IDE 中完成，也可用代码实现。

因为 LCL 中的控件在所有支持的平台上都可使用，在某个平台（如 Windows）上开发的 GUI 程序，不加改动就可在其他平台（如 Mac OS X 或 Linux）上使用。但需要在相应的平台上重新编译。

1.3 LAZARUS 的主要特点

- 开源、免费

由于 Free Pascal 及 Lazarus 是基于 GPL/LGPL 许可协议的。这就意味着此两者是开源的、免费的。是可以用来开发、发行商业产品的。甚至两者的源代码也是可修改的，你一定要发布这些改变，当有人想使用你的改动时，你有义务提供那些改变后的源代码。

- 可移植性

Java 号称是编译一次到处运行，而 lazarus 则可号称编写一次到处编译。

使用 Free Pascal 及 lazarus 的标准组件开发的程序，可以在不修改的情况下，在多个操作系统下编译运行。

- 类 delphi, 有丰富的资源可供参考

在 Lazarus 中有向导可以自动把 Delphi 的工程转换为 Lazarus 的工程。因此已经在 Delphi 中开发的程序可以不经修改或简单修改即可移植使用。

- 可开发手机移动应用程序

1.4 LAZARUS 的架构

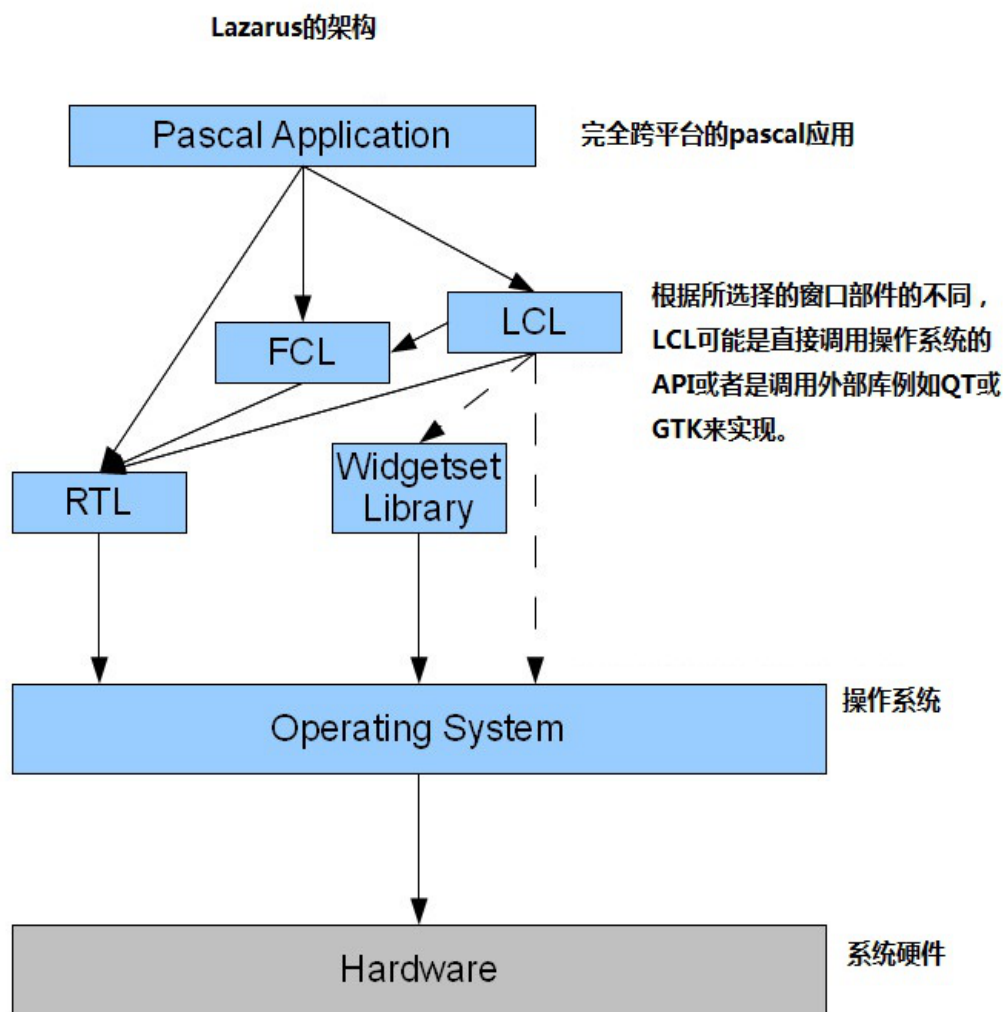


图 1.4.1

Lazarus LCL 组件库的架构图

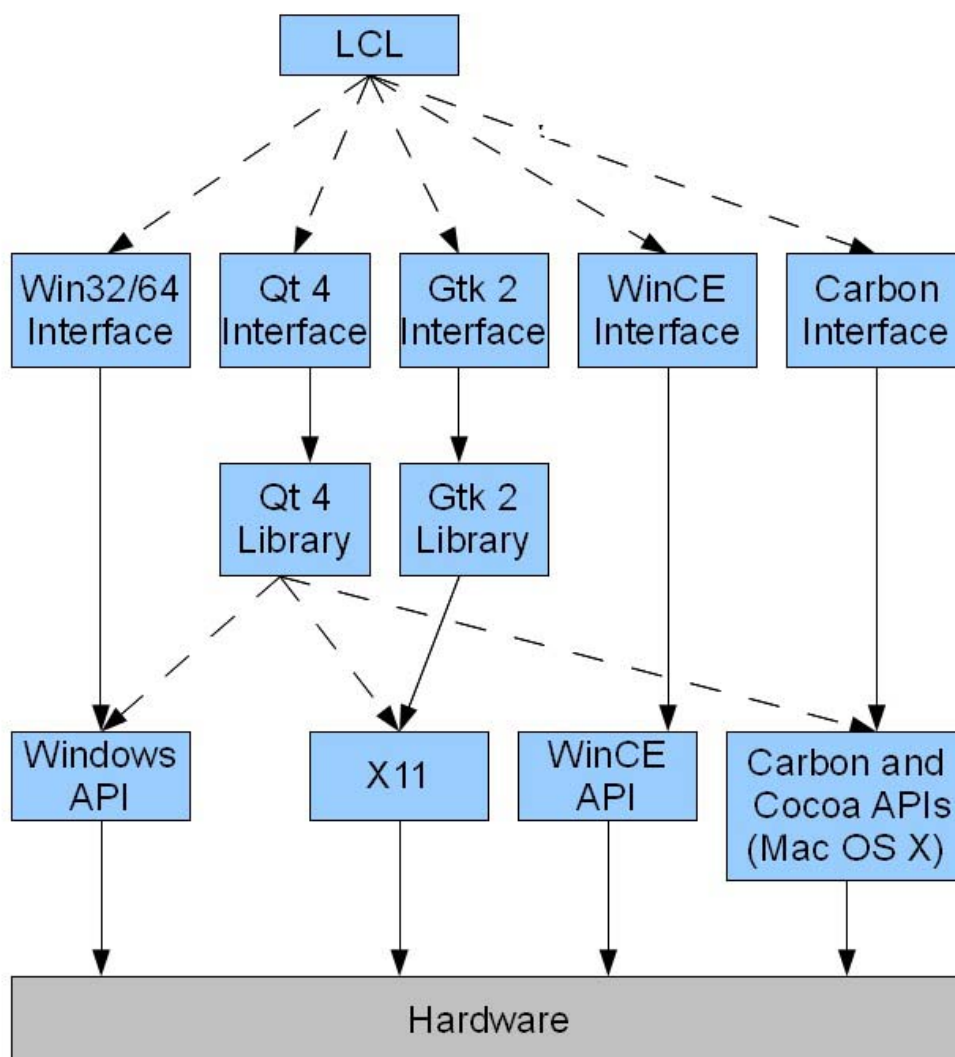


图 1.4.2

第 2 章

用户指南

2.1 LAZARUS 相关问题解答

2.1.1 Lazarus 快速问答 FAQ

2.1.1.1 一般问题

2.1.1.1.1 为什么生成的二进制程序非常大？

生成二进制程序文件大的原因是在编译过程中包含了很多 GNU Denugger 所需要的调试信息，可以通过工程选项设置。

Project|Compiler Options|Code|Smart Linkable (-CX) -> Checked 选择

Project|Compiler Options|Linking|Debugging| Uncheck all except 全不选所有异常

Project|Project options|Compiler Options|Linking|Display Line Numbers in Run-time Error Backtraces(-gl) 不选

Project|Compiler Options|Linking|Strip Symbols From Executable (-Xs) -> Checked 选择

Project|Compiler Options|Linking|Link Style|Link Smart (-XX) -> Checked 选择

另外也可使用 UPX 来缩小二进制文件的尺寸。

虽然生成的二进制文件尺寸比其他编译器的大，但以后程序尺寸的增长速度是很缓慢的，因为在开始二进制程序中几乎包含了 Lazarus 中所需要的支持内容。Lazarus 和 C++编译器生成文件尺寸随工程复杂度变化对比图如下所示：

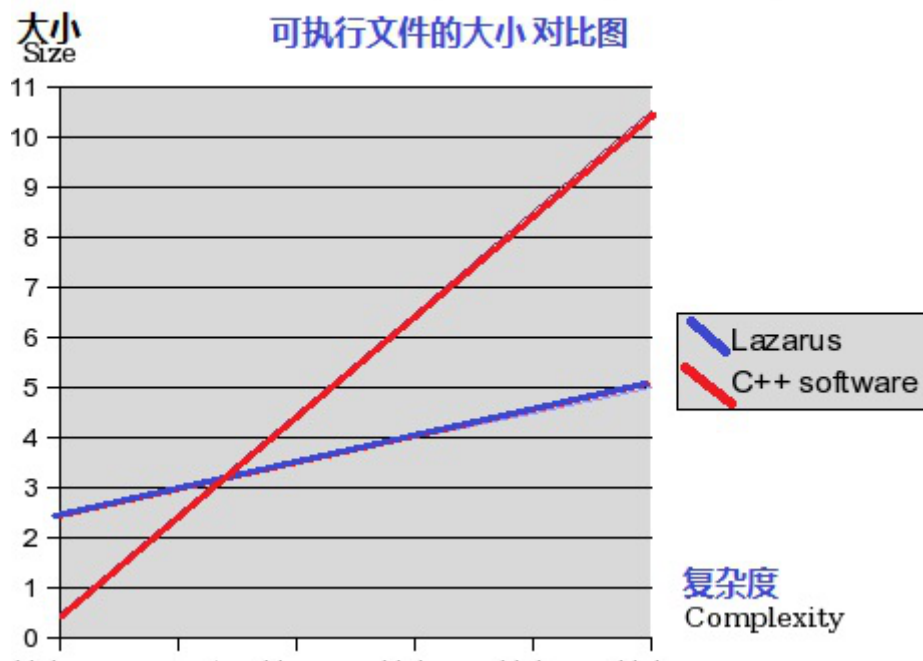


图 2.1-01 可执行文件大小对比图

2.1.1.1.2 如何将小文件嵌入到可执行程序中，从而不必附带单独的文件？

例如要将两个声音文件 sound1.wav, sound2.wav 嵌入到可执行程序中，进行如下操作：
首先将声音文件加入到资源文件 sound.lrs 中。

```
c:\lazarus\tools\lazres.exe sound.lrs sound1.wav, sound2.wav
```

然后将 sound.lrs 包含在 form 的资源包含之后，如下所示：

...

initialization

{ \$i unit1.lrs } // 这里是窗体的主资源文件

{ \$i sound.lrs } // 这里是用户自己生成定义的资源文件

end.

在程序中按如下代码使用：

```
Sound1AsString:=LazarusResources.Find('sound1').Value;
```

2.1.1.1.3 Lazarus 中使用很多不同扩展名的文件，这些文件各有什么用途？

***.lpi**

Lazarus 的工程信息文件（以 XML 格式存储； 包含了工程的描述设置信息）。

***.lpr**

Lazarus 的程序文件；源文件是 Pascal 格式的主程序。

***.lfm**

Lazarus 的窗体文件；包含了一个窗体中所有对象的配置信息。（以 Lazarus 的描述格式存储； 对象对应的动作描述存储在相应的 Pascal 源码文件中。*.pas ）

***.pas or *.pp**

Pascal 代码的单元文件（一般与窗体相应的*.lfm 文件相对应）

***.lrs**

Lazarus 的资源文件（这是生成的文件；不要与 Windows 的资源文件相混淆）。
此资源文件可以由 Lazarus 的工具(在 Lazarus/Tools 目录中) 在命令行中生成：
命令行的格式如下： lzares myfile.lrs myfile.lfm

***.ppu**

编译后的单元文件。

***.lpk**

Lazarus 的包信息文件。（以 XMI 格式存储；包含了包的描述配置信息）

Can I make commercial applications with Lazarus ?

2.1.1.1.4 我可以使用 Lazarus 来开发商业应用吗？

可以，LCL 是在 LGPL 许可协议下的，因此允许你进行静态链接程序而不必发布你应用程序的源代码。但是修改或者改进 LCL 的时候，必须要公开所有源代码。Lazarus,IDE 是在 GPL 许可协议下的。LCL 只是由在 lcl 目录下的程式所组成。其他代码可能不被此种协议覆盖。

2.1.1.1.5 为什么某些组件被限制应用于商业系统中？

Lazarus 的某些附加组件是由第三方开发的。这些组件可能使用了不同的许可协议。如果你要使用这些组件，必须查看相应组件包中关于许可协议的说明。大多数的第三方组件都保存在目录“components”中。

2.1.1.1.6 如何判断一个组件是否属于 LCL 范畴？

所有的 LCL 单元文件都存储在目录“lcl”中。所有的 LCL 列表可浏览此网络连接查看：<http://lazarus-ccr.sourceforge.net/docs/lcl/>。如果你所使用的单元不在这个列表中，那么你所使用的组件就不属于 LCL 范围的。

2.1.1.1.7 我可以为 Lazarus 做商业插件吗？

可以，IDEIntf IDE 的部分是在 LGPL 许可下的，这部分通过共享数据结构不会强迫您的插件遵循 GPL 许可证。你可以自由选择任何许可证插件；我们不想限制你的选择。因此，非 GPL 兼容的插件是允许的。请注意，不允许分发这些非 GPL 兼容的插件，包括静态的一个预编译的 Lazarus。

2.2 LAZARUS 的安装

2.2.1 获得 Lazarus 的相应版本

2.2.1.1 获得并安装 Lazarus 的发行版本

2.2.1.1.1 从 SourceForge

通过 [Lazarus Sourceforge 的下载区域](#)可下载 Lazarus 二进制的发行版本。
网址是：http://sourceforge.net/project/showfiles.php?group_id=89339

2.2.1.1.2 从 SourceForge 下载特定平台的 Lazarus 版本

- Windows: 从以上的链接下载即可。

在以下平台的安装可通过如下网址了解。

- Ubuntu: http://wiki.lazarus.freepascal.org/Lazarus_release_version_for_Ubuntu
- Fedora :
http://wiki.lazarus.freepascal.org/Lazarus_release_version_for_Fedora
- Suse: http://wiki.lazarus.freepascal.org/Lazarus_release_version_for_Suse
- Mandriva :
http://wiki.lazarus.freepascal.org/index.php?title=Lazarus_release_version_for_Mandriva&action=edit
- Mac :
http://wiki.lazarus.freepascal.org/index.php?title=Lazarus_release_version_for_Mac&action=edit
- Solaris: http://wiki.lazarus.freepascal.org/Lazarus_on_Solaris

2.2.2 安装 Lazarus

2.2.2.1 概述

2.2.2.1.2 Lazarus 的系统要求

- A. Free Pascal 的编译器，包，和源代码。（重要：需要是相同的版本及日期）
- B. 支持小部件的工具包

WIN32:

可使用 Windows 原生的 API 或者是 QT 的视窗原件；

Linux/xxxBSD:

GTK+ 2.x or Qt：大多数的 Linux 的发行版本及*BSDs 已经安装了 GTK+ 2.x 的库。你也可以在此网站 <http://www.gtk.org> 找到相关的信息。
Qt 也在所有的发行包中支持了。（如果你指定了 KDE 也会自动安装）。

Mac OS X:

你需要有苹果开发者工具（the Apple developer tools.）。可看下面的关于在 Mac OS X 的安装说明。Qt 也会被使用到。

Qt 窗口部件集在 Linux 32/64, Win32/64, Mac OS X, Haiku and 嵌入 linux (qtopia) 平台都能得到支持。

2.3 程序语言介绍

2.3.1 基本类型

所有的变量都必须有一个类型，Free pascal 支持像 Turbo pascal 一样的基本类型，还有一些是来自 Delphi 的特别类型。程序员也可以声明自己的类型。

有 7 种主要的类型：

简单类型

字符串类型
结构体类型
指针类型
过程类型
一般类型
专有类型
类型标识

类型标识一般用于给一个现有的类型以另外一个名字。

2.3.1.1 有序类型

除了 `int64`，`qword` 和 `real` 类型之外，所有的基本类型都是有序类型。

有序类型具有如下特性：

1. 有序类型是可数的并且是可排序的。例如，可以按一定的顺序一个、一个地数他们。这个属性允许对这些所定义的有序类型进行加、减、排序的操作。
2. 有序变量都有一个最小可能的值，可以对有序变量应用 `Pred` 函数来检测变量的取值范围。
3. 有序变量也会有一个最大值，可使用 `Succ` 函数来检测最大值的范围。

2.3.1.1.1 整型类型 Integers

整型列表如下所示：

类型	范围	占用字节数
Byte	0 .. 255	1
ShortInt	-128 .. 127	1
SmallInt	-32768 .. 32767	2
Word	0 .. 65535	2
Integer	与 SmallInt 或 longint 相同	2 或者 4
Cardinal	与 longword 相同	4
LongInt	-2147483648 .. 214783647	4
LongWord	0 .. 4294967295	4
Int64	-9223372036854775808 .. 9223372036854775807	8
Qword	0 .. 18446744073709551615	8

在 Free Pascal 模式中 Integer 总被映射为 SmallInt，在 Delphi 或 ObjFPC 中也会被映射为 LongInt. Cardinal 类型当前总是被映射为 LongWord. 类型。

2.3.1.1.2 Boolean 布尔类型

FreePascal 支持布尔类型。有两个预定义的值 True 及 False. 对于布尔类型也仅有这两个值。当然任何表达式都可以解析为布尔类型值。Free Pascal 也支持 ByteBool、WordBool、LongBool.

名称	字节数	True 值
Boolean	1	1
ByteBool	1	任何非 0 值
WordBool	2	任何非 0 值
LongBool	4	任何非 0 值

类型 Byte、Word、LongInt 的值都与 boolean 兼容。当做类型转换时值 False 相当于 0，而其他非 0 值则相当于 True。当把 Boolean 类型转换为 LongBool 时，True 被转换为-1.

如果字母 B 是 Boolean 类型，则以下的赋值是有效的：

B := True;

B := False;

B := 1<>2; 【结果相当于 B := True】

布尔类型的表达式也可以被应用在条件中。

2.3.1.1.3 枚举类型

在 Free pascal 中也支持枚举类型。例如：

Type

Direction = (North, East, South, West);

也支持 C 格式的枚举类型，例如：

Type

EnumType = (one, two, three, forty := 40, fortyone);

最终，forty 的值是 40 而不是 3. Fortyone 的值是 41.

必须要注意的是，枚举类型的值必须按照升序排列，否则就会产生错误。例如：

Type

```
EnumType = (one, two, three, forty := 40, thirty := 30);
```

因此有必须调换 forty 和 thirty 的顺序。

在使用枚举类型时,，必须重点记住以下几点：

1. Pred 和 Succ 函数不能用于枚举类型的变量，如果这样做了会产生编译错误；
2. 枚举类型会存储默认的、独立的实际值，编译器不会进行变量占用空间的优化。此特性可使用 {\$PACKENUM n}的编译指令来改变，使用这样的指令可告诉编译器枚举类型占用的最小的字节数，例如：

```
Type
{$PACKENUM 4}
LargeEnum = ( BigOne, BigTwo, BigThree );
{$PACKENUM 1}
SmallEnum = ( one, two, three );
Var S : SmallEnum;
    L : LargeEnum;
begin
WriteLn ('Small enum 占用的字节数:',SizeOf(S));
WriteLn ('Large enum 占用的字节数: ',SizeOf(L));
End.
```

最后程序运行后，会打印出如下信息：

```
Small enum 占用的字节数: 1
Large enum 占用的字节数: 4
```

2.3.1.1.4 子界类型

子界类型是主类型一定范围内的定义。要定义子界类型必须要指定子界类型的最大值及最小值。一些预先定义的 Integers 整型类型就是通过子界类型定义的。例如：

```
Type
Longint = $80000000..$7fffffff;
Integer = -32768..32767;
shortint = -128..127;
byte = 0..255;
Word = 0..65535;
```

也可以定义枚举性质的子界类型，例如：

```
Type
Days = (monday,tuesday,wednesday,thursday,friday,saturday,sunday);
WorkDays = monday .. friday;
WeekEnd = Saturday .. Sunday;
```

2.3.1.1.5 实数类型

Free Pascal 使用数学协处理器（或仿真）来进行浮点数的计算。不管是单精度还是双精度实数的本地类型是依赖于处理器的。只有美国电气和电子工程师协会(IEEE) 的浮点数类型被支持并且取决于目标处理器或仿真选项。真正与 Turbo pascal 兼容的类型列举如下：

类型	范围	有效数字	占用字节数
Real	依赖于平台	? ? ?	4 或 8
Single	1.5E-45 .. 3.4E38	7-8	4
Double	5.0E-324 .. 1.7E308	15-16	8
Extended	1.9E-4932 .. 1.1E4932	19-20	10
Comp	-2E64+1 .. 2E63-1	19-20	8
Currency	-922337203685477.5808	922337203685477.5807	8

2.3.1.1.6 字符类型

2.3.1.1.6.1 Char 字符

Free Pascal 支持 char 类型，此类型有一个字节的大小，可存储一个 Accii 码。
一个字符的常量可用单引号括起来。例如： 'a' 或者 'A'.
字符类型也可通过字符对应的 ascii 码赋值，例如：值#65 与 'A' 是等价的。
符号 ^ 也可以同字母组合来表示 ascii 吗小于 27 的字符。例如 ^G 等价于 #7.

2.3.1.1.6.2 String 字符串

Free Pascal 可支持类似 Turbo pascal 中的 String 类型：可设定长度的字符串，也支持像 Delphi 中的不限定长度的字符串。

2.3.1.1.6.3 Short String 短字符串

声明为短字符串有以下两种情况：

1. 如果开关 {\$H-} 是关闭的，则字符串的定义都是短字符串；
2. 如果开关 {\$H+} 是打开的，并且指定了字符串的最大长度，则字符串就被声明为短字符串。

Short String 被预先定义为长度为 255 的字符串，如下所示：

```
ShortString = String[255];
```

如果没有指定字符串的长度，则默认的长度为 255。在进程运行时可得到字符串的长度。例如：

```
{$H-}  
Type  
NameString = String[10];  
StreetString = String;
```

NameString 最多可包含 10 个字符。StreetString 可包含 255 个字符。

备注：ShortString 的最大长度只能是 255 个字符，如果所指定的长度超过 255，则编译器会报错。提示：字符串的长度只能定义为 0~255 之间。

对于短字符串来说，字符串的长度被存储在索引为 0 的位置。为了便于移植，建议使用 SetLength 来设置字符串的长度，使用 Length 来得到字符串的长度。

2.3.1.1.6.4 Ansistrings

AnsiString 是没有长度限制的字符串，使用计数器及 Null 来表示字符串结束。

通常情况下，AnsiString 被当做指针来对待：字符串的实际内容存储在堆中。为了存储字符串内容需要申请相应的内存。

备注：

Null 空终止并不表示可以使用空字符（char(0) 或者 #0）来截止字符串。其内部并不使用空来截止。但是为了与外部进程（一些 C 的进程）通信可能要用到空截止。

如果{\$H}的开关是开的，则使用 String 关键字定义的字符串是不指定长度的，也会被当做 AnsiString 来处理。如果指定长度了，则会当做 Short String 来处理。而不管是否定义了{\$H} 开关。

如果字符串是空的(""),则字符串的内部指针为 Nil。如果字符串不为空，则其指针指向堆中分配的内存地址。

将一个字符串赋值给另外一个字符串，并不会引起字符的移动。如下语句：

```
S2 := S1;
```

结果是 S2 的参考数量减 1，S1 的参考数量加 1 了。最终 S1 的指针被指向了 S2，这样可以提高代码的速度。

如果字符串的参考数量降到 0 时，则此字符串占用的内存会被自动释放。指针被设置为 Nil。所以就不会有内存泄露产生。

当一个字符串被声明时，Free pascal 的编译器只分配了其指针所用的内存，不会占用更多的内存。并且这个指针保证是空的即 Nil。表示此字符串是空的。不管字符串是局部的、全局的、在数组中、在结构体中都是如此。

下面介绍一下超过范围的情况，例如声明如下：

```
Var
```

```
  A : Array[1..100000] of string;
```

将会复制 10 万次 Nil 到 A。如果 A 超出了范围，则数组中的每个字符串的参考数都会减一。所有这些动作对程序员来说是不可见的。但如果出于对性能的考虑，这就非常重要了。因为会有 10 万次对每个字符串减 1 的动作。

2.3.1.1.6.5 WideString 宽字符串

WideString 用于保存 Unicode 字符串。类似于 AnsiString 字符串，有参考计数及以 null 结尾的数组。由 wideChars 构成。并不是通常的 chars。一个 WideChar 有两个 char 构成。编译器本身不支持 WideString 及 ansiString 的相互转换。

2.3.1.1.6.6 Constant strings 常量字符串

常量字符串类似于 char 类型，字符串值以单引号定义，只不过是超过一个字符了。下面是一些常量字符串的定义：

```
S := 'This is a string.';
```

```
S := 'One'+', Two'+', Three';
```

```
S := 'This isn"t difficult !';
```

```
S := 'This is a weird character : '#145' !';
```

常量字符串以 String 存储或以 Short String 存储取决于开关 {\$H}。

2.3.1.1.6.7 PChar 以 null 结尾的字符串

Free Pascal 支持 Delphi 中实现的 PChar 类型。PChar 类型被定义为指向字符串的指针。允

许进行额外的操作。PChar 类型可以理解为 C 语言中以 null 结尾的字符串。例如，一个 PChar 类型的变量被指向一个字符串数组，以空的字符（#0）结尾。在 fpc 中可以对 Pchar 类型的变量进行常量初始化或直接进行赋值初始化。例如：

```
//程序一 ,直接赋值初始化;  
var P : PChar;  
begin  
  P := 'This is a null-terminated string.';  
  WriteLn (P);  
end.
```

与下面程序的效果相同

```
//程序二，使用常量字符串初始化;
```

```
const P : PChar = 'This is a null-terminated string.';  
begin  
  WriteLn (P);  
end.
```

2.3.2 结构类型

结构类型的一个变量可以保存多个类型的变量，并且可以无限嵌套定义。

-- structured type array type

- record type
- object type
- class type
- class reference type
- interface type
- set type
- file type

与 delphi 不同，Free pascal 不支持在结构里使用 Packed 关键字。

2.3.2.1 数组

像在 Turbo pascal 中一样 Free Pascal 支持数组，多维数组，位对齐的数组也支持(bit packed) 还有动态数组。

2.3.2.1.1 静态数组

当数组的维数是在数组定义里包含的则就是静态数组。可通过数字的元素下标访问数组的值。如果超出范围则会报一个运行时错误。

下面是单维静态数组的定义：

Type

```
RealArray = Array [1..100] of Real;
```

有效的数组范围从 1 到 100. 边界 1 和 100 都可使用。

下面是多维数组的定义

Type

```
APoints = array[1..100] of Array[1..3] of Real;
```

与以下定义是等效的：

Type

```
APoints = array[1..100,1..3] of Real;
```

函数 High 及 Low 返回数组的最大下标及最小下标，分别是 100 及 1.

第 3 章

Lazarus IDE(集成开发环境)

3.1 IDE 介绍

以下所有内容以官方 Lazarus 0.9.28.3 beta 版为准。

Lazarus 的 IDE (集成开发环境) 分为主窗体 (Main Editor Windows)、对象观察器 (Object Inspector)、窗体设计器 (Form Designer)、源代码编辑器 (Source Editor) 以及消息窗口 (Messages Windows)。当你首次运行 Lazarus 时, 你会发现有这样 5 个独立浮动的窗体出现在你的桌面上。首先, 在桌面的顶部, 有一个标题为 **Lazarus IDE V0.9.28.3 beta - project1** (此文件名是你打开工程文件的名字) 的窗口, 这就是主窗体。在主窗体的下面, 左边是对象观察器, 右边是窗体设计器和源代码编辑器, 通常, 你会看到一个小窗体浮动在源代码编辑器的上面, 这个就是窗体设计器。如果它没有出现, 你可以按 F12 键把它显示出来。你可以通过按 F12 在源代码编辑器和窗体设计器之间来回切换显示。在源代码编辑器的下面就是消息窗口。下面, 我们将对这些内容分别介绍。

3.1.1 主窗体(Main Editor Windows)

主窗体如图 3.1 所示, 它是整个 Lazarus 的控制中心, 它与通常的 Windows 程序主窗口相似, 拥有所有标准功能。主窗体主要分为主菜单 (Main Menu)、快速按钮工具栏 (Speed Button ToolBar)、组件面板 (Component) 等三部分。

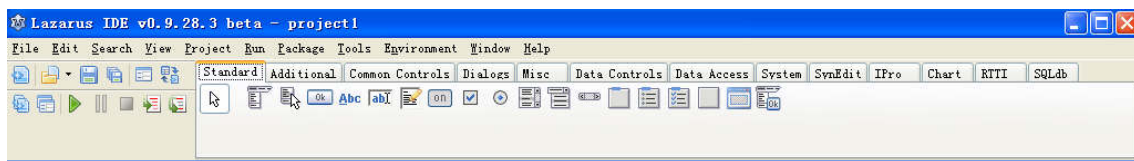


图 3.1 主窗体

3.1.1.1 主菜单 (Main Menu)

在标题的下面就是 Lazarus 主菜单, 主菜单提供了 IDE 的所有功能, 通常情况下, 你可以直

接单击菜单的标题来打开相应的菜单或者也可以按 ALT+菜单标题文本中带下划线的字符（例如 ALT+F 打开 File（文件）菜单）来打开菜单。下面简单介绍一下 Lazarus 0.9.28.3 beta 版本中的各个菜单，其它版本有可能有一些差别。

1. File（文件）菜单

文件 (File) 菜单如图 3.2 所示，它提供 IDE 中各种文件操作功能，各菜单的功能说明如下：

- New Unit (新建单元): 创建一个新的单元文件 (Pascal 源代码)。
- New Form (新建窗体): 创建一个新的窗体，提供可视化窗口及 Pascal 源代码。
- New ... (新建 ...): 弹出一个对话框，可以选择不同类型的文档进行创建。
- Open ... (打开 ...): 弹出打开对话框，可以浏览并选择打开工程、包或其它文件。快捷键：Ctrl+O。
- Revert (重新打开): 放弃当前打开文件所进行的编辑，恢复到此文件的初始状态。
- Open Recent ... (打开最近的文件): 列出最近编辑的文件，选择打开其中的某一个。
- Save (保存): 保存当前文件，使用其当前文件名，如果该文件未命名，系统将提示你取一个 (类似于另存为)。快捷键：Ctrl+S。
- Save As ... (另存为 ...): 允许您选择一个目录和其它文件名称保存当前文件。
- Save All (全部保存): 官方文件译为“全选”，有误。将当前所有文件都保存。快捷键：Shift + Ctrl + S。
- Close (关闭): 关闭当前文件，如果编辑的文件未保存，将提示是否保存。快捷键：Ctrl +F4。
- Close all editor files (关闭所有编辑器中的文件): 关闭当前文件编辑器中所有的文件，如果编辑的文件未保存，将提示是否保存。
- Clean directory ... (清除目录): 弹出一个对话框，可以通过设置过滤条件来删除当前工程目录中的一些文件。例如 .bak 文件。
- Print ... (打印): 弹出打印对话框，选择系统中的打印机来打印源代码编辑器中的文件。

此菜单在非 Windows 系统中没有默认安装，你可以通过打开

`$Lazdir/components/printers/design/printers4lazide.pas` 并且重构 Lazarus IDE 来安装它。快捷键：Ctrl + P。

- Restart(重启):退出并重新启动 Lazarus。
- Quit(退出):退出 Lazarus, 会提示保存所有修改过的文件。

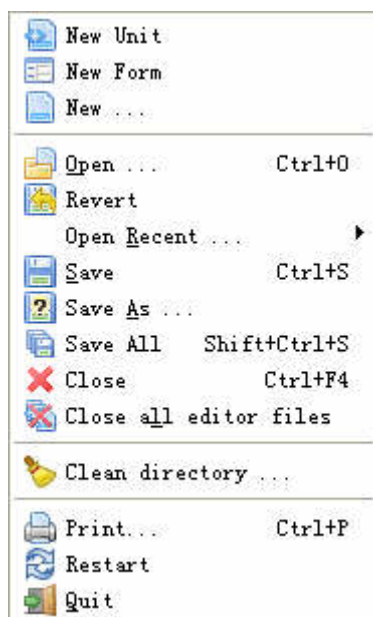


图 3.2 File 菜单

2.Edit (编辑) 菜单

编辑菜单如图 3.3 所示, 它提供了 IDE 中的各种编辑功能, 各菜单的功能说明如下:

- Undo(撤消):撤消上一次的操作。快捷键: Ctrl + Z。
- Redo(重做):重新执行上一次的操作。快捷键: Shift + Ctrl + Z。
- Cut(剪切):剪切。把当前所选的文本或其它项目移至剪贴板中。快捷键: Ctrl+X。
- Copy(复制):复制。复制所选的文本, 在剪贴板中放置此文本的副本。快捷键: Ctrl+C。
- Paste(粘贴):粘贴。将剪贴板中文本复制到光标位置, 如果光标位置有选定文本, 剪贴板中的内容将取代所选文本。快捷键: Ctrl+V。
- Indent selection(缩进所选):将所选文本内容按照 Lazarus 设置的缩进方式进行缩进。此功能可以将你的 Pascal 代码格式化整齐。默认缩进方式为右缩进 2 格。具体设置可以查看: Environment -> Options -> Editor -> General。快捷键: Ctrl + I。
- Unindent selection(取消所选缩进):删除所选文本一个级别的缩进, 使之向左移动 2 格。快捷键: Ctrl + U。

- `Enclose selection...` (封装所选 ...): 弹出一个窗口, 可以选择一个语句结构来对所选文本内容进行封闭。所选文本将包含在此语句结构中。如: `Begin.. End` , `Try.. Except`。
- `Comment selection` (注释所选): 对所选文本进行注释。默认在所选文本前加注释符 “//”。
- `Uncomment selection` (取消注释所选): 对所选文本取消注释。去掉注释符 “//”。
- `Toggle comment` (切换 注释): 对光标所在行或所选文本在注释和不注释之间切换。官方未翻译此菜单。
- `Insert $IFDEF` (插入 \$IFDEF): 弹出一个窗口, 可以选择条件对所选文本代码进行新的预定义。快捷键: `Shift + Ctrl + D`。
- `Sort selection` (排序所选): 弹出一个窗口, 将行 (或单词或段落) 按字母排序, 可以选择升序或降序、是否区分大小写、是否忽略空格。
- `Uppercase selection` (转换所选为大写): 转换所选的文本为大写字母。
- `Lowercase selection` (转换选区为小写): 转换所选的文本为小写字母。
- `Tabs to spaces in selection` (转换选区制表符转换为空格): 将所选文本中的制表符转换为一定数量的空格。此数量默认每个制表符为 8 个空格, 具体设置可以查看: `Environment -> Options -> Editor -> General -> Tabs to spaces`。
- `Select` (选择): 可以让你选择一个区段的文本。选项有选择所有、选择到括号、选择代码块、选择行、选择一段等等。
- `Insert from Character Map` (从字符表插入): 可以让你插入一个在键盘上没有的特符字符或者外文字符, 从弹出的字符表窗口中选择。
- `Insert text` (插入文本): 子菜单中可以让你插入一些标准的文本格式。如 CVS 关键字, GPL/LGPL 公约, 当前用户名, 当前日期时间, ToDo 标签等等。
- `Complete Code` (自动完成代码): 自动完成光标所在处的代码。会根据前后代码的意思以节省你的时间。快捷键: `Shift + Ctrl + C`。
- `Extract procedure` (解析过程): 利用所选文件代码来建立一个过程。

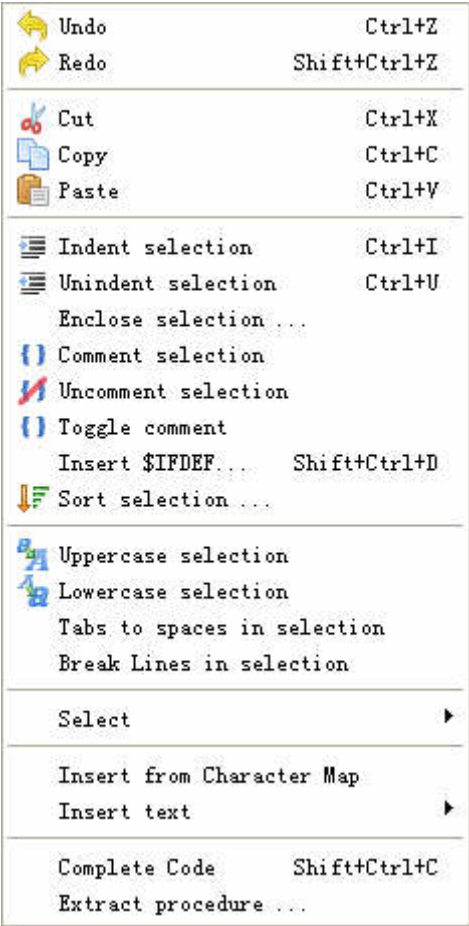


图 3.3 Edit 菜单

3. Search（查找）菜单

查找菜单如图 3.4 所示，它提供了 IDE 中的各种文本查找功能，各菜单的功能说明如下：

- Find ... (查找 ...):弹出查找对话框。查找指定的文本，查找的方式选项可以指定。
- Find Next (查找下一个):利用上次查找的文本，从当前位置向下继续查找。快捷键：F3。
- Find Previous (查找上一个):利用上次查找的文本，从当前位置向上继续查找。快捷键：Shift + F3。
- Find in files ... (在文件中查找):弹出对话框，按选定的条件在文件中查找指定的文本。可以在目录中查找。快捷键：Shift + Ctrl + F。
- Replace ... (替换):与查找 Find 菜单相似。弹出对话框，选择查找的文本和替换的文本，替换的方式可以指定。快捷键：Ctrl + R。

- Incremental Find(增量搜索):采用渐进式查找方式查找项目中各文件的内容。快捷键: Ctrl + E。
- Goto line ... (转到行):直接跳至程序代码中的指定行。快捷键: Ctrl + G。
- Jump back(跳转到后一个):跳到上一个位置。每一次跳到错误或是声明处 IDE 会保存目前的原始位置,此功能可以让你再依跳转历史跳回来。快捷键: Ctrl + H。
- Jump forward(跳转到前一个):跳到下一个位置。快捷键: Shift + Ctrl + H。
- Add jump point to history(添加跳转点到历史):加入目前的位置到跳转点历史中。
- Jump to next error(跳到下一个错误):跳到下一个源代码中报错误的位置。快捷键: Ctrl+F8。
- Jump to previous error(跳到前一个错误):跳到上一个源代码中报错误的位置。快捷键: Shift + Ctrl + F8。
- Set a free bookmark(设置一个自由书签):标记当前光标所在代码行成为可以使用(自由使用)的书签并加上编号,然后加入到书签清单里。注意在源代码编辑器中右击弹出的菜单,其中的书签范围很大,可以指定编号找到书签,也可以指定编号跳到书签处,而不单只是上一个或下一个。
- Jump to next bookmark(跳到下一个书签):依顺序跳到下一个书签。
- Jump to previous bookmark(跳到前一个书签):依顺序跳到前一上书签。
- Find other end of code block(找代码区段的另一端):如果目前处在 Begin 处,那么找到相对应的 End 处,反之亦然。
- Find code block start(查找代码区段开始):光标移动到目前光标所在的函数或过程的 Begin 处。
- Find Declaration at cursor(在光标处查找声明):为当前选取的标识符找到它的声明处。位置有可能在同一个文件中,也可能是在另一个编辑器打开的文件中;若是未打开的文件,程序将会自动打开它。快捷键: Alt + 向上键。
- Open filename at cursor(在光标处打开文件名):打开光标处选取为文件名称的文件。对于 Include 文件的查找或是取得工程中包含的单元特别有用。快捷键: Ctrl+回车。

- Goto include directive(前往包含指示处):如果光标选取指定的文件名称是被包含在其它的文件中,那么移到包含所指文件名称的文件中。
- Find Identifier References ... (查找标识符参考):建立一份在目前文件里或者所有依赖于目前文件的所有标识符的清单。
- Rename Identifier ... (重命名标识符):可以让开发者为标识符重命名。快捷键: Shift + Ctrl + E。
- Procedure List ... (过程列表):建立一份目前文件中所有过程和函数的清单,并加上它们在声明处的行号。快捷键: Alt + G。

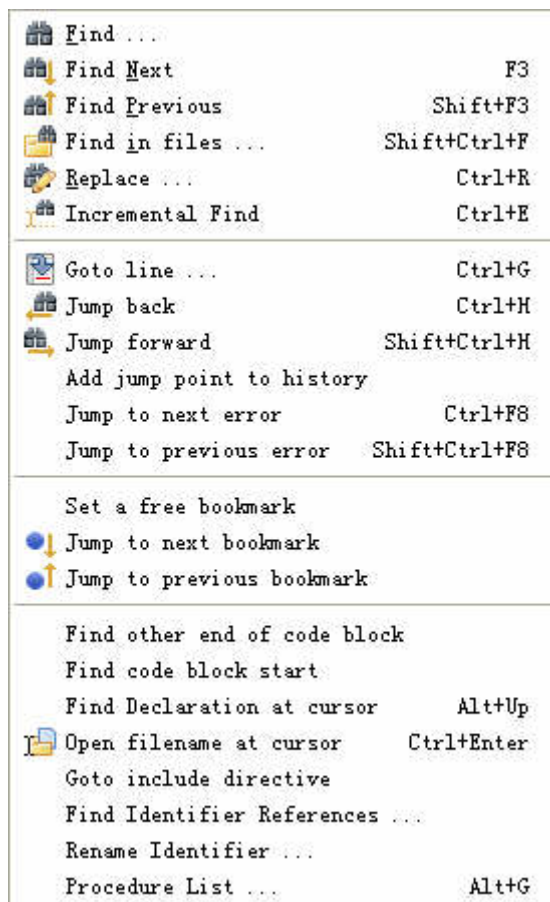


图 3.4 Search 菜单

4.View (查看) 菜单

查看菜单如图 3.5 所示,控制屏幕中各个窗口和面板显示,各菜单的功能说明如下:

- Object Inspector(对象观察器):显示关闭的对象观察器,通常在你桌面的左边。我们将

在 3.1.2 对象观察器一节来具体介绍它。快捷键：F11。

- Source Editor (源代码编辑器):显示关闭的源代码编辑器。编写源代码的主窗口。操作方法与其它的文本编辑器相同。我们将在 3.1.4 源代码编辑器一节来具体介绍它。
- Messages (消息):显示关闭的消息窗口。编译过程的进度、成功信息、错误将在此窗口中显示。我们将在 3.1.5 消息窗口一节来具体介绍它。
- Code Explorer (代码浏览器):显示关闭的代码浏览器。以树型结构出现在桌面的右方。其中列出了当前单元的结构。可以看到类型、变量、常数、引用的单元等内容。如果你在源代码编辑器中修改过后，再切换回代码浏览器时，请使用刷新按钮来重新显示。
- FPDoc Editor (FPDoc 编辑器):Free Pascal 代码文档生成器。此编辑器是用于生成当前单元过程和函数具体描述参考文档的工具。生成的文档以 XML 形式保存。
- Code Browser (代码浏览):显示代码浏览窗口。(官方无介绍)
- Restriction Browser (限制浏览):显示限制浏览窗口。此窗口列出了所有目前架构中组件不支持的属性。
- Components (组件):列出了当前所有安装的组件清单和继承关系。
- Jump History ... (查看跳转历史):显示所有跳转历史清单。
- ToDo List (ToDo 清单):列出与当前工程相关联的 ToDo 项目。当前工程和其它任何有使用的 Lazarus 单元所有 ToDo 注释都会列出来 (注释行//ToDo)。
- Units ... (单元):显示单元对话框。其中列出了当前工程所有的单元列表。你可以选择其中的打开它。快捷键：Ctrl+F12。
- Forms ... (窗体):显示窗体对话框。其中列出了当前工程所有的窗体列表。快捷键：Shift + F12。
- Unit Dependencies (查看单元依赖关系):显示当前打开的单元文件依存性树状图。
- Unit Information (查看单元信息):显示当前单元的名称、路径、包含路径及源代码路径的信息。
- Toggle form/unit view (切换 窗体/单元 视图):在源代码编辑器和窗体设计器之间切换显示。快捷键为 F12。

- Search Results(查找结果):查找结果。(官方无介绍) 快捷键: Ctrl+Alt+F。
- Anchor Editor(锚编辑器):编辑窗体或组件的锚点。
- Component Palette(组件面板):显示或隐藏主窗体上的组件面板。默认为打勾显示。
- IDE Speed buttons(IDE 快速按钮):显示或隐藏主窗体上的快速按钮工具栏。默认为打勾显示。
- Debug windows(调试窗口):可以打开以下调试窗口: 断点、局部变量、注册、调用堆栈、汇编、事件日志、调试输出。
- IDE internals(IDE 内部):显示包(package)连接关系、FPC 信息、IDE 信息。



图 3.5 View 菜单

5. Project (工程) 菜单

工程菜单如图 3.6 所示, 各菜单的功能说明如下:

- New Project ... (新建工程):新建一个工程, 可以从对话框中选择新建工程的类型。
- New Project from file ... (从文件新建工程):出现一个对话框, 选择 .pas、.pp、.lpr

类型的源代码文件从而建立一个新的工程。

- Open Project ... (打开工程): 出现打开对话框, 选择打开已存在的 Lazarus 工程文件。
快捷键: Ctrl+F11。
- Open Recent Project ... (打开最近的工程): 列出最近打开工程的列表, 你可以选择一个打开。
- Close Project (关闭工程): 关闭当前打开的 Lazarus 工程。
- Save Project (保存工程): 保存当前打开的工程中所有的文件。
- Save Project As ... (工程另存为): 出现另存为对话框, 可以给工程另取名称保存。名称不能与当前工程名称相同。
- Publish Project ... (发布工程): 为当前工程建立一个副本。如果你只是想要将你的源代码和编译设定发送给其他人, 那么这个功能就是你的好帮手。通常工程的目录里包含了一堆的信息。但是里面有很多在发布时并不需要: .lpi 文件包含会话期的信息(像脱字符^的位置或是未打开单元中的书签), 而工程目录中包含很多.ppu 和.0 的文件与可执行文件。建立 Lpi 文件会生成仅包含基本信息和源代码过程, 然后放在“发布工程”的子目录中。在对话框中, 你可以设定和选择要或不要的过滤器, 也可以使用压缩命令使其发布成一个单一的文件。
- Project Inspector (工程浏览器): 出现一个对话框, 将当前工程以树状图结构列出, 允许你新增、移除或是打开所选择的文件, 或是更改工程的设定。
- Project Options ... (工程选项): 弹出一个窗口, 有一个树状结构, 分为“工程选项”和“编译选项”两类。可以对应用程序设定选项(标题, 工程输出名称, 版本信息等等); 并且可以设定工程编译的选项和参数(路径, 代码, 目标平台, 链接, 冗余等等)。快捷键: Shift + Ctrl + F11。
- Add editor file to Project (添加代码编辑器中的文件到工程): 将当前源代码编辑器中的文件加入到工程中。快捷键: Shift + F11。
- Remove from Project ... (从工程中移除): 弹出一个对话框, 可以让你选择工程中的文件移除(非删除)。

- View Source(查看源文件):不管你现在在编辑什么, 立即将你转到主工程文件 (.lpr), 如果没有.lpr 文件那么转到主要的.pas 文件中。

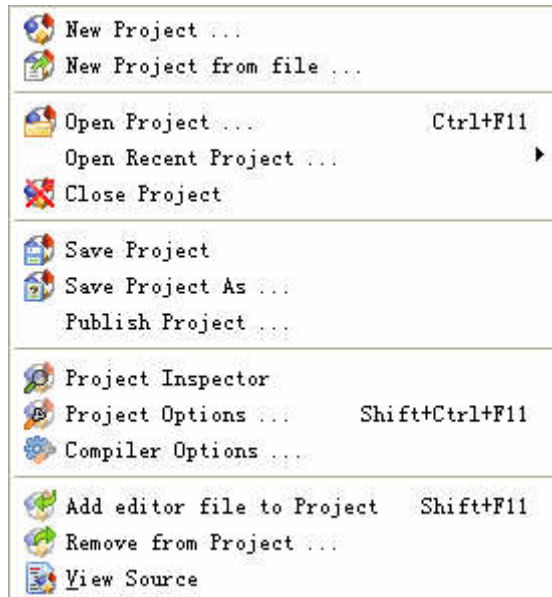


图 3.6 Project 菜单

6. Run (运行) 菜单

运行菜单如图 3.7 所示, 各菜单的功能说明如下:

- Build(构建):进行构建 (或说编译) 在工程中所有从上次构建到这次中有修改过的文件。
(通俗的讲就是编译修改过的文件然后重新连接生成可执行文件) 快捷键: Ctrl + F9。
- Build all(构建所有):工程中所有文件都进行构建, 无论它有没有修改过。
- Quick compile(快速编译):快速的执行编译和构建。
- Abort Build(中断编译):应该为“放弃构建”。停止正在进行的构建。
- Run(运行):这大概是最常用的编译器的方法, 若编译成功完成, 则自动运行该应用程序。
实际上 Lazarus 将你的文件另存为一个副本, 然后关闭编译器和连接器, 然后开始运行最后连接成的二进制程序。快捷键: F9。
- Pause(暂停):将目前运行中的程序暂停。这可以让你侦测程序产生的输出值; 再选一次 Run(运行)可以再继续运行。
- Step into(单步进入):与调试器结合使用, 让编译进行时可以一次一步, 或者按源代码内

的书签点一步步地执行。快捷键：F7。

- Step over(下一步):按定位点先一次执行到此处,然后忽略,然后再继续执行。当想要找出程序中的逻辑错误时很有用。快捷键：F8。
- Run to cursor(运行到光标处):先以正常方式执行编译（不是一次一步），直到执行到光标所在的位置时停止。再按一次 RUN（运行）将继续。快捷键：F4。
- Stop(停止):停止编译程序。无法再从 RUN 命令继续，只能从头编译（用于需要重新编译时）。快捷键：Ctrl + F2。
- Run Parameters ... (带参数运行):弹出一个窗口，用来设定程序运行时要带的命令和参数。
- Reset debugger(重置调试器):重新恢复调试器到初始状态，断点和变量的值都将消失。
- Build File(构建文件):编译（构建）当前源代码编辑器中打开的文件。
- Run File(运行文件):编译，连接和运行当前打开的文件。
- Configure Build+Run File ... (配置 编译+运行 文件):打开一个允许构建文件时有“构建工程”选项选取的多页标签窗口，允许选择工作的目录，使用不同的宏命令等等。然后构建和运行文件。
- Inspect ... (查看):查看。官方无介绍。快捷键：Alt + F5。
- Evaluate/Modify ... (赋值/更改):官方无介绍。快捷键：Ctrl + F7。
- Add watch ... (添加观察点):添加观察点。官方无介绍。快捷键：Ctrl + F5。
- Add breakpoint(添加断点):添加断点。官方无介绍。

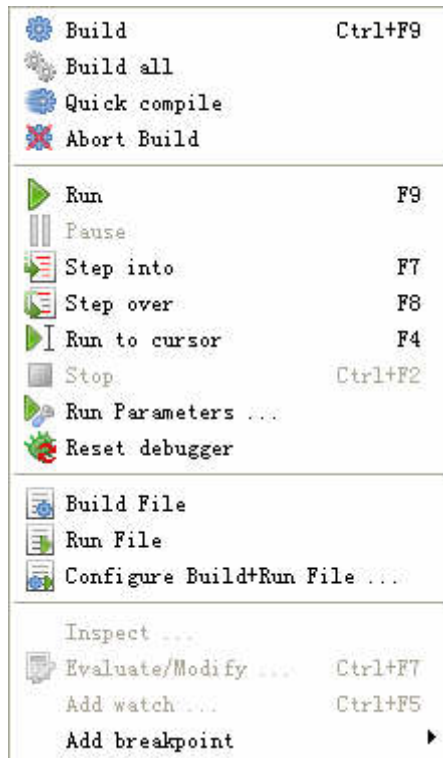


图 3.7 Run 菜单

7.Package（包）菜单

包菜单如图 3.8 所示，各菜单的功能说明如下：

- New Package ... (新建包):弹出一个保存包的对话框和包编辑器。
- Open loaded package ... (打开装载的包):显示一个 Lazarus 已经安装的包的列表，有一个打开包的建议，打开后可选择常规选项或编译选项。
- Open package file(.lpk) ... (打开包文件):打开选择的包文件。
- Open package of current unit(打开当前单元的包):打开当前单元所在的包（如果有的话）。
- Open recent package ... (打开最近的包):选择打开最近打开过的包的列表。
- Add active unit to a package(将活动单元添加到包):添加当前源代码编辑器中的文件到一个包中。
- Package Graph ... (包关系图):显示一个图表，此图表展现了当前使用的包的关系图（如果你没有使用任何包文件，那么将显示 Lazarus 包和 FCL 以及 LCL 库）。

- Configure installed packages ... (配置已安装的包): 安装和卸载包文件。显示一个列表, 左边是已安装的包, 右边是未安装的包。可以对已安装的包进行卸载, 也可以对未安装的包进行安装。如果要安装和卸载生效, 那么需要对 IDE 保存并重新编译(重构)。

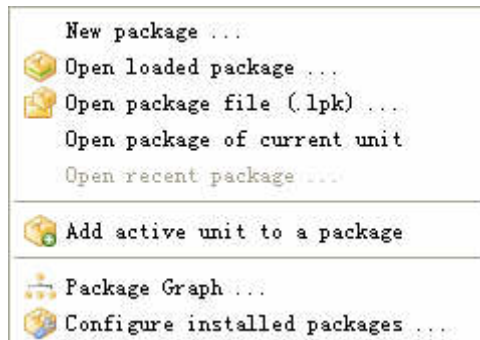


图 3.8 Package 菜单

8. Tools (工具) 菜单

工具菜单如图 3.9 所示, 各菜单的功能说明如下:

- Configure external tools ... (配置外部工具): 允许用户添加各类扩展工具 (通常是宏命令) 到开发包中。
- Quick syntax check (快速语法检查): 对源代码文件执行一次快速语法检查, 并不编译任何东西。事实上这一步是在开发很长或者复杂程序时, 当你不想浪费编译时间于代码出错时。
- Guess unclosed block (猜测未关闭的块): 有用的工具, 如果你有一个复杂嵌套分程序结构并且你在某处漏掉了 “End”。它会帮你自动添上 “End”。
- Guess misplaced IFDEF/ENDIF (猜测错位的 IFDEF/ENDIF): 用于复杂或嵌套宏定义结构并且当你认为可能漏掉了一个 ENDEF 命令时。
- Make Resource String ... (制作资源字符串): 制作选取的字符串作为资源字符串放入资源字符串区。资源字符串的优势是你可以改变它而不需要重新编译你的工程。
- Diff (比较): 允许比较两个文件 (或者, 通常, 同一文件的两个不同版本) 来查找它们的不同。有选项来忽略空白空格在行的开始或结尾处或者行结束处的不同: (CR+LF (回车换行) 对应 LF (换行))。用于检查自从最后一次 CVS 升级后是否有更改等等。

- Project templates options(工程模板选项):选择工程模板保存的文件夹位置。
- JEDI Code Format(JEDI 代码格式化):为当前代码排版成 DELPHI OBJECT PASCAL 标准的代码格式。通过一个 JEDI 代码格式化外部工具来实现。
- Check LFM file in editor(在编辑器中检查 LFM 文件):允许检查 LFM 文件, LFM 文件是一个包含描述当前窗体设定的文件。要检查 LFM 文件, 你必须先打开它。
- Convert Delphi unit to Lazarus unit ... (转换 Delphi 单元为 Lazarus 单元):帮助转换 Delphi 的单元文件为 Lazarus 单元文件; 生成所需的源代码。可以查看 Lazarus 的 Delphi 用户和代码转换指南。
- Convert Delphi project to Lazarus project ... (转换 Delphi 工程为 Lazarus 工程):帮助转换 Delphi 工程文件为 Lazarus 工程。
- Convert Delphi package to Lazarus package ... (转换 Delphi 包为 Lazarus 包):帮助转换 Delphi 包文件为 Lazarus 包文件。
- Convert binary DFM file to text LFM and check syntax ... (转换 DFM 文件为 LFM 文件):转换 Delphi 窗体定义文件 DFM 为 Lazarus 窗体定义文件 LFM。
- Convert encoding of projects/packages ... (转换工程/包的编码):转换当前 Lazarus 环境中的包文件的编码为其它编码方式。
- Build Lazarus(构建 Lazarus):按照下面菜单中的配置资料来重新构建 Lazarus IDE 环境。具体设置请查看下面的菜单。
- Configure "Build Lazarus" ... (配置 "编译 Lazarus"):设置详细的重新构建 Lazarus IDE 环境的信息。允许用户定义 Lazarus 的哪部分应该重新构建或者如何重新构建。例如:你可以只选择重构 LCL 库, 或者除示例程序外所有都重构; 你可以选择用于哪一种 LCL 接口部件类型, 你可以选择目标操作系统类型和指定不同的目标文件夹。

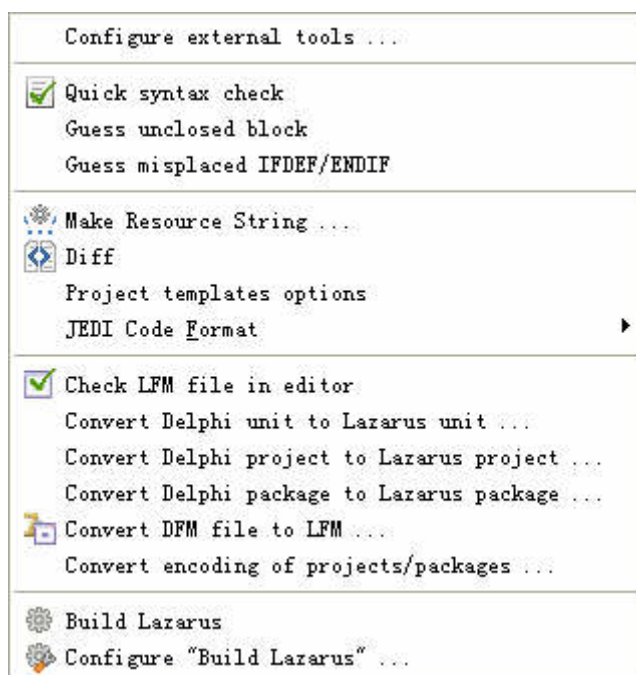


图 3.9 Tools 菜单

9.Environment（环境）菜单

环境菜单如图 3.10 所示，各菜单的功能说明如下：

- Options ... (选项):Lazarus IDE 中所有预设的选项值。可以更改这些选项以符合你个人的习惯。包含：环境选项，编辑器选项，JEDI 代码格式选项，代码工具选项，代码管理器选项，调试器选项，帮助选项。
- Code Templates ... (代码模板):代码模板。代码模板文件。
- CodeTools defines editor ... (代码工具自定义编辑器):在这里你可以看到为分析源码而提供的所有 IDE 内部定义。
- Rescan FPC source directory(重新扫描 FPC 源代码目录):重新检视文件夹。Lazarus 使用 FPC 源码来生成正确的事件处理程序和查找声明。如果有人更改了环境选项中的文件夹位置，那么这个文件夹要重新扫描，用于确定被 Lazarus 使用的 FPC 版本保存在这个文件夹位置中。但是如果更改了此文件夹而不通知 Lazarus，那么当你设计窗体或者查找“声明”时可能得到一些错误。如果你收到这样的错误，那么你可以做以下两件事情：（1）在环境选项中检查 FPC 源代码文件夹的设置。（2）执行“重新扫描 FPC 源代码目录”命令。

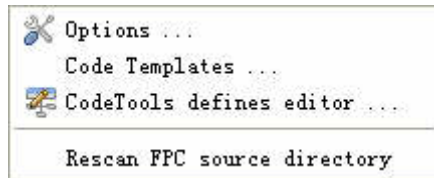


图 3.10 Environment 菜单

10. Window (窗口) 菜单

窗口菜单如图 3.11 所示，包含当前打开的文件和可能的窗口的一个列表，如以下：

- Source Editor(源代码编辑器)
- Object Inspector(对象观察器)
- Messages(消息窗口)
- Form1(窗体设计器:Form1)
- Unit1(单元: Unit1)

点击其中一个窗口的名称，将使其前置显示。



图 3.11 Window 菜单

11. Help (帮助) 菜单

帮助菜单如图 3.12 所示，在当前有三个选项，各菜单的功能说明如下：

- Online Help(在线帮助): 打开一个包含奔跑猎豹的图片和一些连接到 Lazarus, Free Pascal 和 WiKi 页站点的窗口。
- Reporting a bug ... (报告错误): 打一个 WiKi 页面，其中描述了如何报告一个错误的程序。
- About Lazarus(关于 Lazarus): 跳出一个窗口显示 Lazarus 的信息。

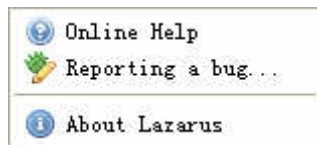


图 3.12 Help 菜单

3.1.1.2 快速按钮工具栏 (Speed Button ToolBar)

在主菜单的下面靠左侧位置有一组小按钮，我们称之为快速按钮工具栏。它包括了一些主菜单中的常用项目，例如新建文件、打开、保存、运行等等。如图 3.13 所示。



图 3.13 快速按钮工具栏

上面一行按钮从左到右依次为：New Unit (新建单元)，Open (打开)，Save (保存)，Save all (保存所有)，New Form (新建窗体)，Toggle Form/Unit (切换 窗体/单元)。

下面一行按钮从左到右依次为：View Units (查看 单元)，View Forms (查看窗体)，Run (运行)，Pause (暂停)，Stop (停止)，Step Into (步进)，Step Over (下一步)。

3.1.1.3 组件面板 (Component)

组件面板位于快速按钮工具栏的右侧，它是一个多标签页工具栏，包含很多不同图标的常用组件用于设计窗体界面。如图 3.14 所示。

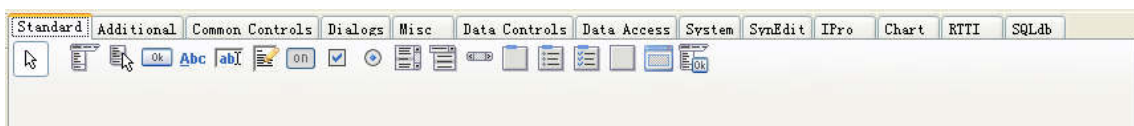


图 3.14 组件面板

每一个标签页显示一组不同的图标，表示一组组件。在每组组件的最左边有一个指向左上侧的箭头，我们称之为选择工具。

3.1.2 对象观察器 (Object Inspector)

3.1.3 窗体设计器 (Form Designer)

3.1.4 源代码编辑器 (Source Editor)

3.1.5 消息窗口 (Messages Windows)

3.2 IDE 技巧

3.2.1 把开发环境改为简体中文版

当安装完 Lazarus 后，默认的语言是英文的，其实，它是有简体中文版本的，可以通过以下操作来设为简体中文版本：Environment->Option->Desktop->Language->Chinese[zh_CN]，然后先关闭 Lazarus，重启 Lazarus，变为简体中文版本了。

3.2.2 创建新的文件

你可以使用以下的方法在源代码编辑器中创建一个新文件，同时，你也可以指定新文件的文件名和文件类型：[文件]->[打开] (或快捷键 CTRL+O) 同时输入一个不存在的文件名称，例如 Unit2.pas，IDE 会提示“文件未找到，是否想创建”，选择“是”，就完成创建了。

第 1 章

控件制作

1.1 控件的认识

自从进入可视化编程以来，控件在人们的日常使用中越来越广泛，轻松的一拖一放，双击一下控件，马上能进入特定的事件进行代码的编写，确实很方便。

下面让我们来认识 LAZARUS 控件的组成：

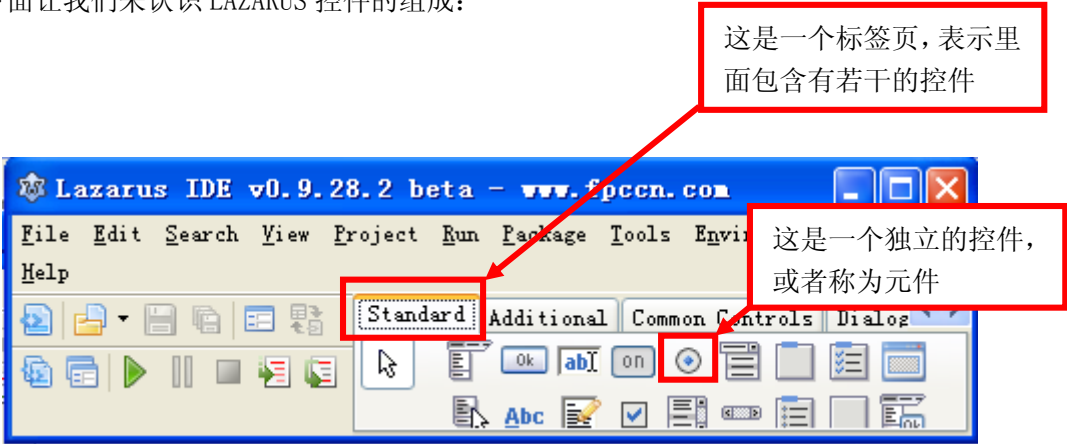


图 1

点击按钮控件，并拖放在窗体上，此时，在 IDE 的对象观察器里就可以看到该控件的所有属性，如图 2 所示。



图 2

再来看看控件的事件，如图 3 所示。

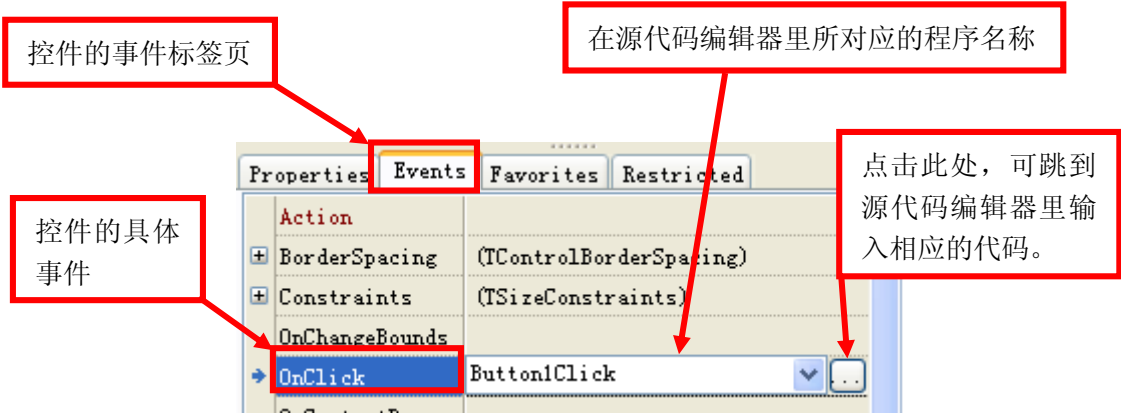


图 3

在源代码编辑器添加一行代码： `showmessage('Hello World');` 最终的显示为：

```
1      procedure TForm1.Button1Click(Sender: TObject);
2
3      begin
4          showmessage('hello world');
5
6      end;
```

1.2 控件的创建

1.2.1 需求分析

在创建控件之前，建议进行需求分析，以便明确控件的作用，在创建的过程保持思路的一致。下面以串口控件为例，演示需求分析。（在后面创建控件的过程也以此为例）

异步串行通讯是一种很常用的，廉价的通讯方式，可以让不同类型的机器之间实现数据的交换，最常用的是 RS232 方式，我们将创建一个串口控件来建立这种通讯方式。

名称	RS232 （此处为一个名称，不等同真实的串口）
作用	将数据写入到本机的串行口，传送到与外部机器
	读取本机串行口接收到的数据（该数据为外部机器发送过来的）
需要的属性	串口号

	波特率
	数据位
	校验
	停止位
需要产生的事件	无

表 1

在创建控件的时候，我们可以将表 1 中的名称作为单元的名称；将作用做成程序或函数；将需要的属性作为控件的属性。

1.2.2 创建控件

打开 LAZARUS 的 IDE，在下拉菜单里，依次点选 “Package” ——》“New package”，如图 4 所示，接着弹出图 5。

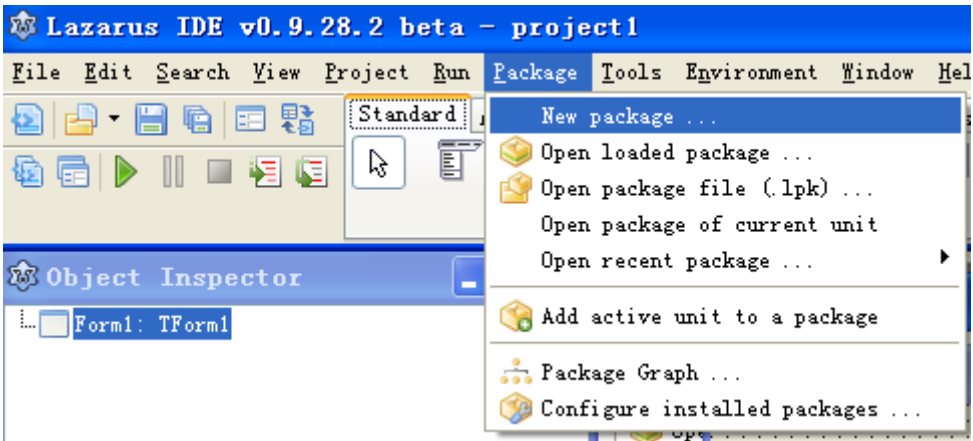


图 4

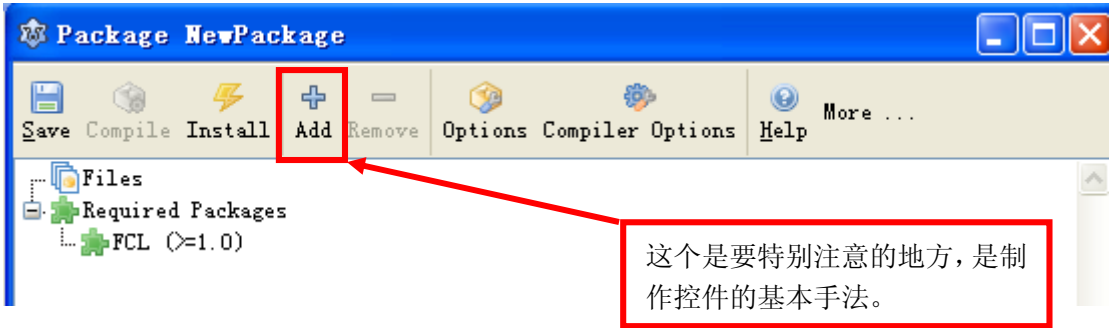


图 5

在点击图 5 的按钮 “Add” 后，在图 6 里要首先选择标签页 “New Component”！

如果所要制作的控件是继承了已有控件的特性的，那么在 Ancestor Type 的下拉框里就选择对应的继承控件。串口控件没有从已有的控件继承任何东西，所以选择了 “Tcomponent”，如图 7 所示。

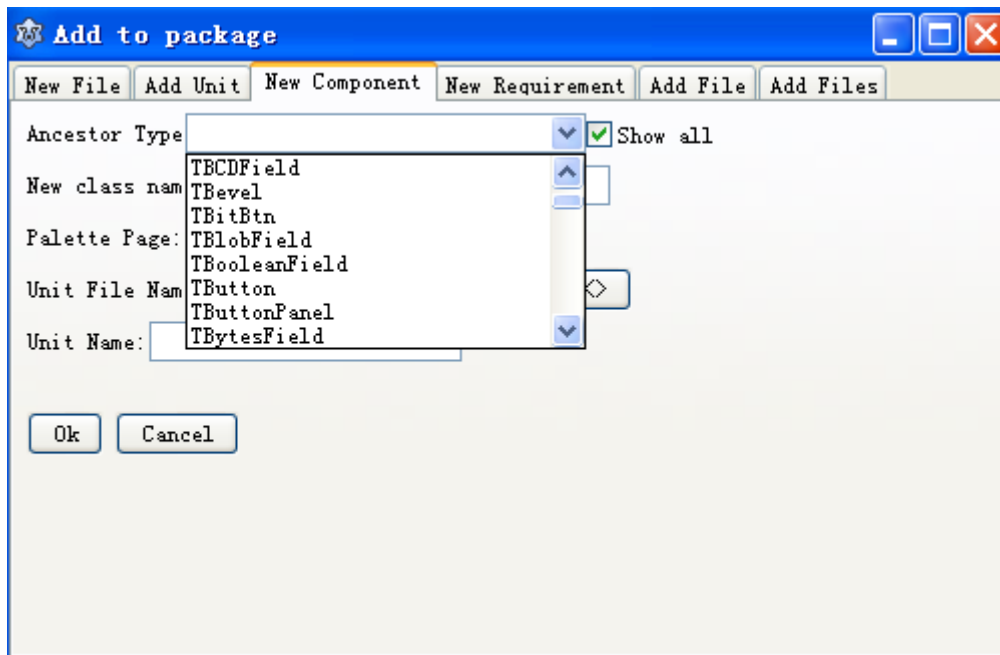


图 6

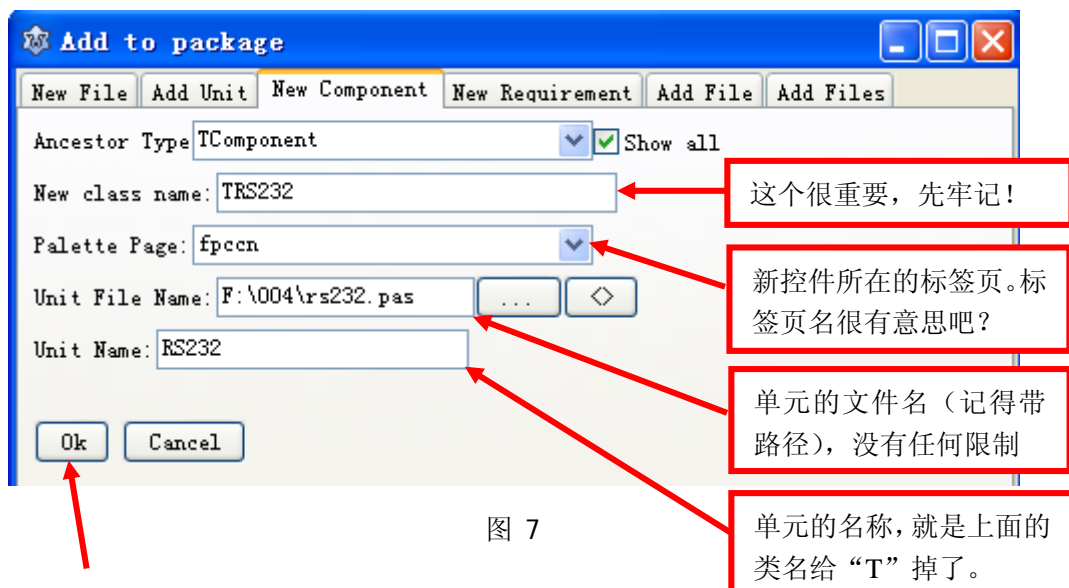


图 7

这个太重要了，不点它，后续的工作就无法做了

完成图 7 的工作后，切换到标签页“Add Unit”添加单元文件，即如图 8 所示。

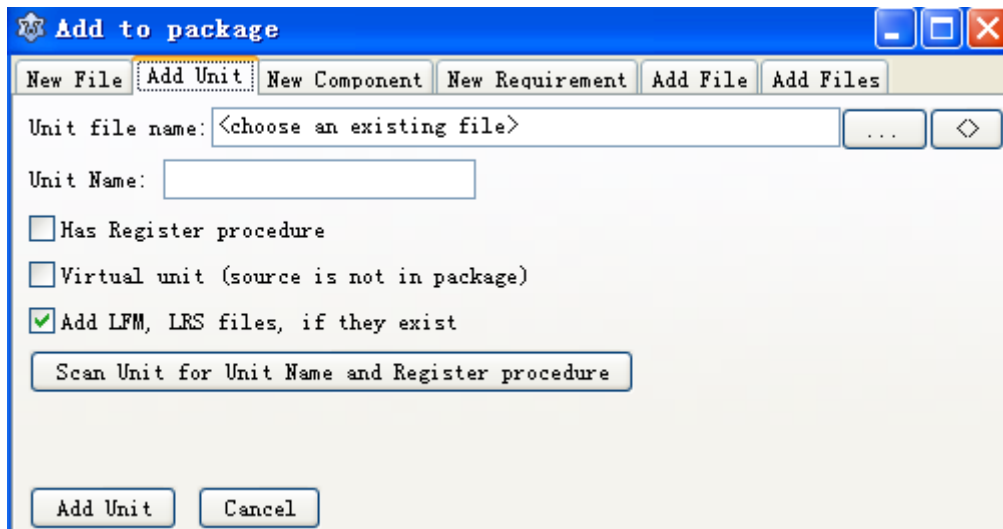


图 8

先从这里查找图 7 保存的文件

这个文件是前面的图 7 中保存的

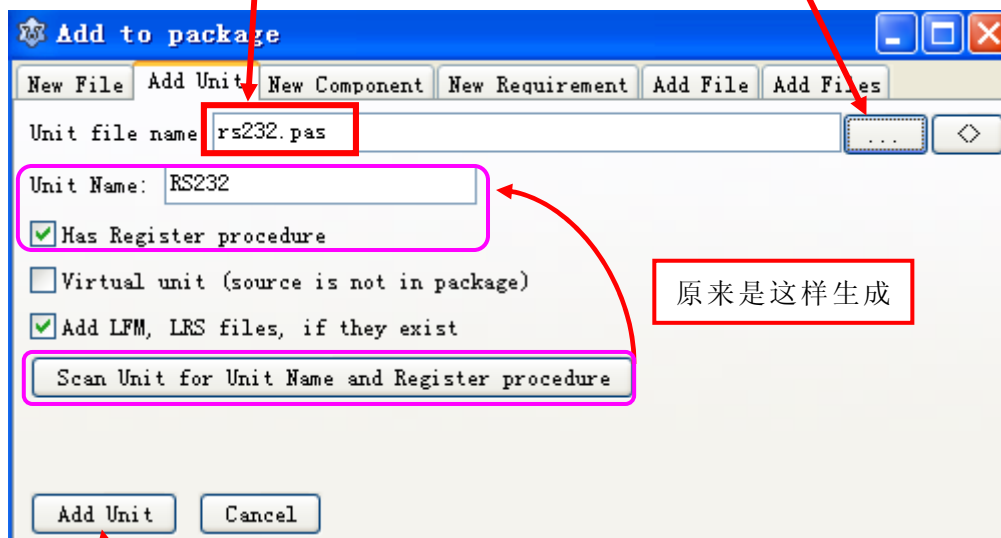


图 9

这里是最不应该忘记的地方，也是最后一步



经过图 9 的一番处理，得到图 10 中的树型结构，这个时候，千万记得保存一下劳动的成果。如图 10，保存成名为：my232 的控件包。

当这个图标实实在在出现的时候，要毫不犹豫地点击它；
当它再次变虚的时候，你的劳动就不会跑掉了。

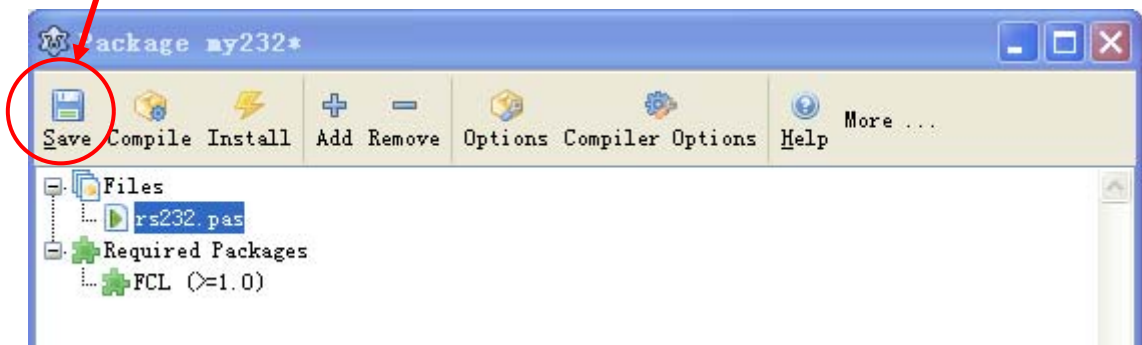


图 10



图 11

图 7 中的设置，在这里可以比较清楚地看到了。控件的基本框架到此已基本创建完毕。

1.2.3 创建控件图标

一个有特色的图标能将你的控件醒目地展现出来，视觉上的良好感受会让你的控件更受欢迎。

首先遵从 LAZARUS 的规定，我们制作 24x24 像素的图形文件。如图 12，使用 WINDOWS 自带的画图软件画出所需要的图，并保存成名为“trs232.png”的图形文件。注意：图形文件名必须与控件的类名一致！

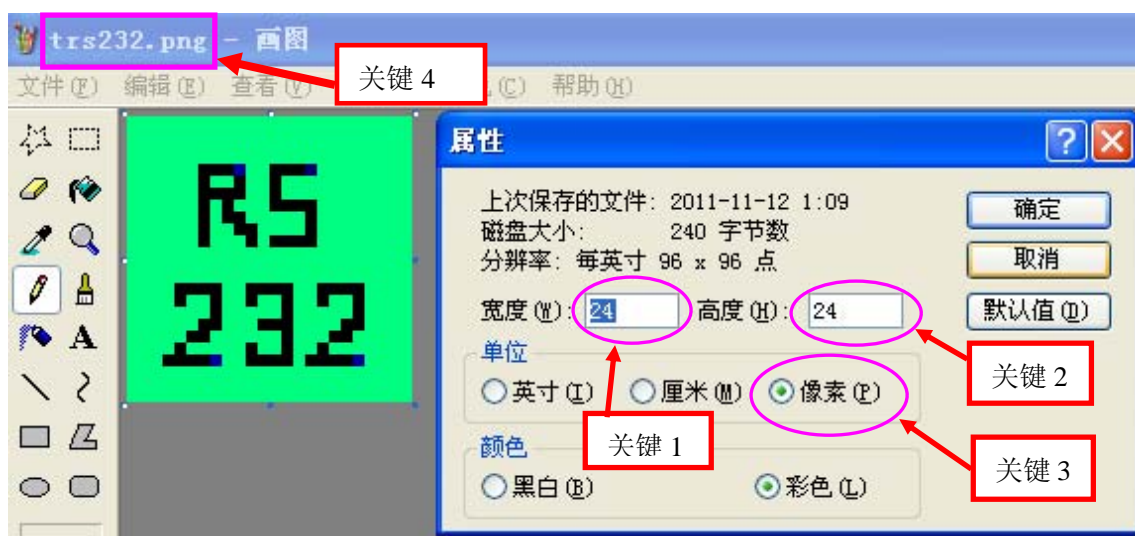


图 12



图 13

在 LAZARUS 的 tools 文件夹里, 有个 lazres.lpi 的工程, 编译后该项目, 得到一个可执行程序: lazres.exe。该程序可将图形文件转化为 LAZARUS 的资源文件, 不过它是行命令的程序, 在使用上就不那么令人愉快了, 最自动化也得自己编个批处理文件, 不给力啊。

笔者修改了 lazres.lpi 的工程, 增加了图形界面, 操作上比较直观了, 如图 13 所示的界面。该软件已经发布在中文社区里, 有需要的读者可以到 www.fpccn.com 下载, 帖子见: <http://www.fpccn.com/read.php?tid=1589>

帖子标题: Lazarus 资源文件的图形化转换器。

这个图形化的软件, 只要简单地使用图 13 中的步骤 1 和步骤 2 就可以生成所需要的资源文件。

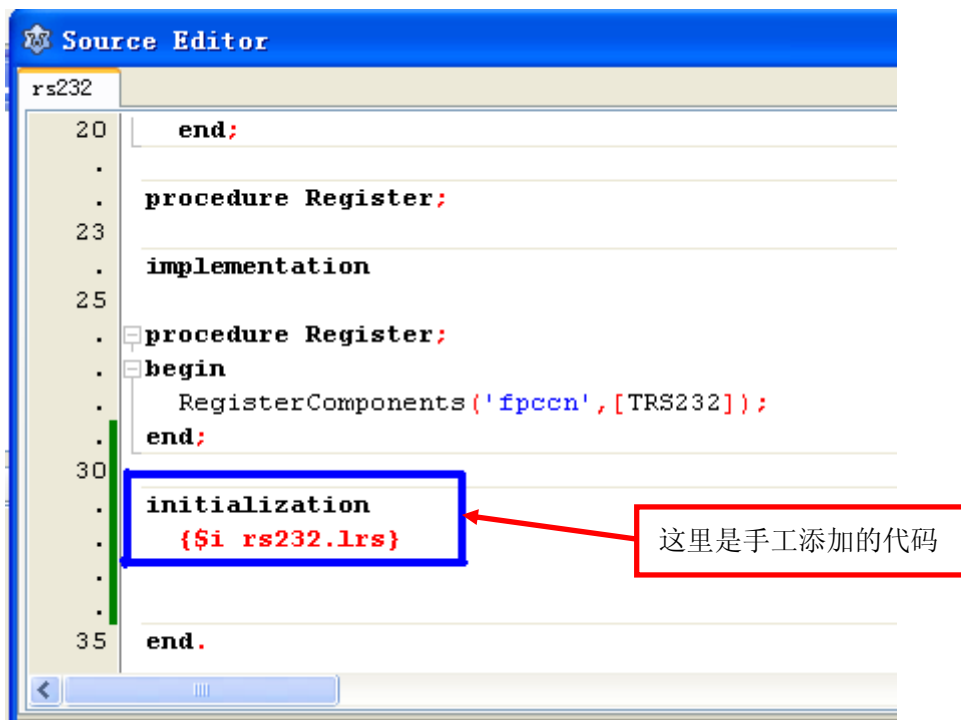


图 14

在图 13 里，我们生成了资源文件 rs232.lrs，为了可以让控件使用该资源文件，还必须在单元文件 rs232.pas 里手工添加相关的引用语句，如图 14 所示。

添加完毕，满怀欣喜地按一下“compile”，这时候令人沮丧的事情发生了：弹出了错误的信息提示！就是图 15 显示的情景了。

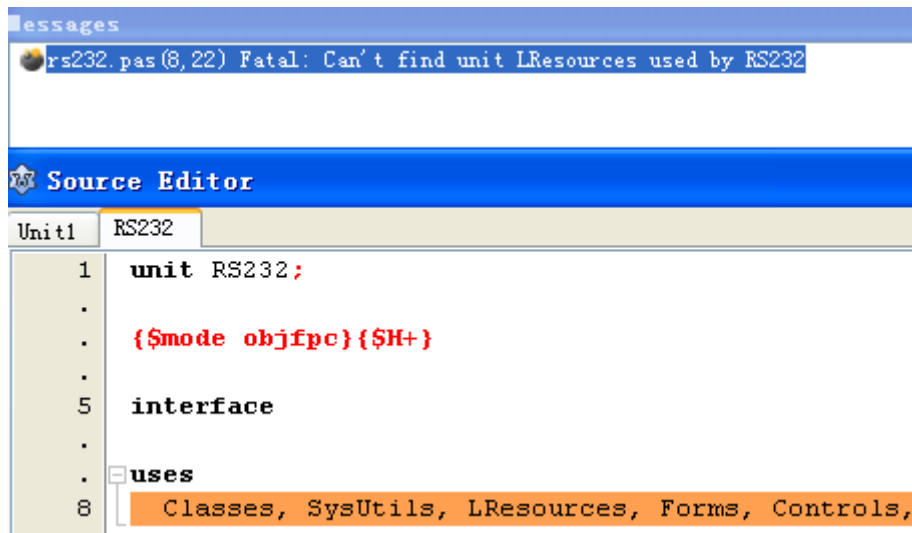


图 15

这里需要增加 LAZARUS 的官方资料里没有提到的一些步骤：

- ① 打开创建的 my232 的控件包，点击 “Add” --> “New Requirement”，弹出图 16 的界面，然后在 “Package Name” 的下拉选择框里选择 “LCL”，接着按 “OK” 按钮，在控件包里看到变化了：LCL (>=1.0) 被添加到控件包里了。图 17 显示了这个变化。

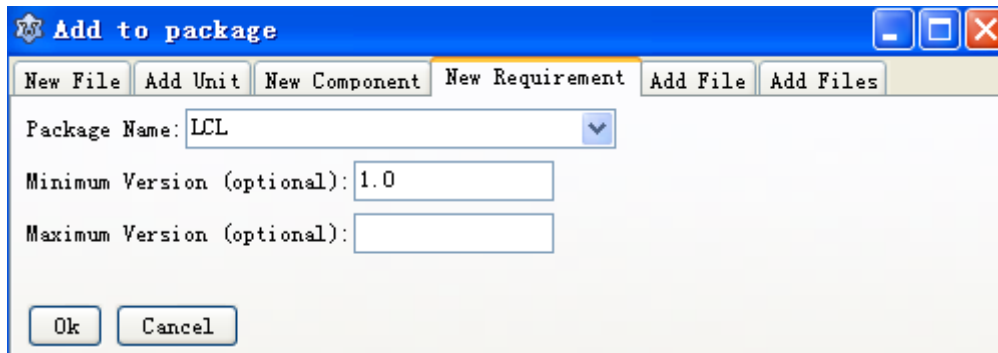


图 16

- ② 点击 “Add” --> “Add File”，在 “File Type” 的选择列表里，选中其中的 “LRS - Lazarus resource”，接着选择需要添加资源文件，请谨记：一定要选择经过图 13 处理得到的文件：rs232.lrs，要保证所使用的资源文件的名称和控件包里的单元的名称的一致，是要高度的统一，不能随意选择别的文件名称，否则，你得到的将是若干个不眠之夜。

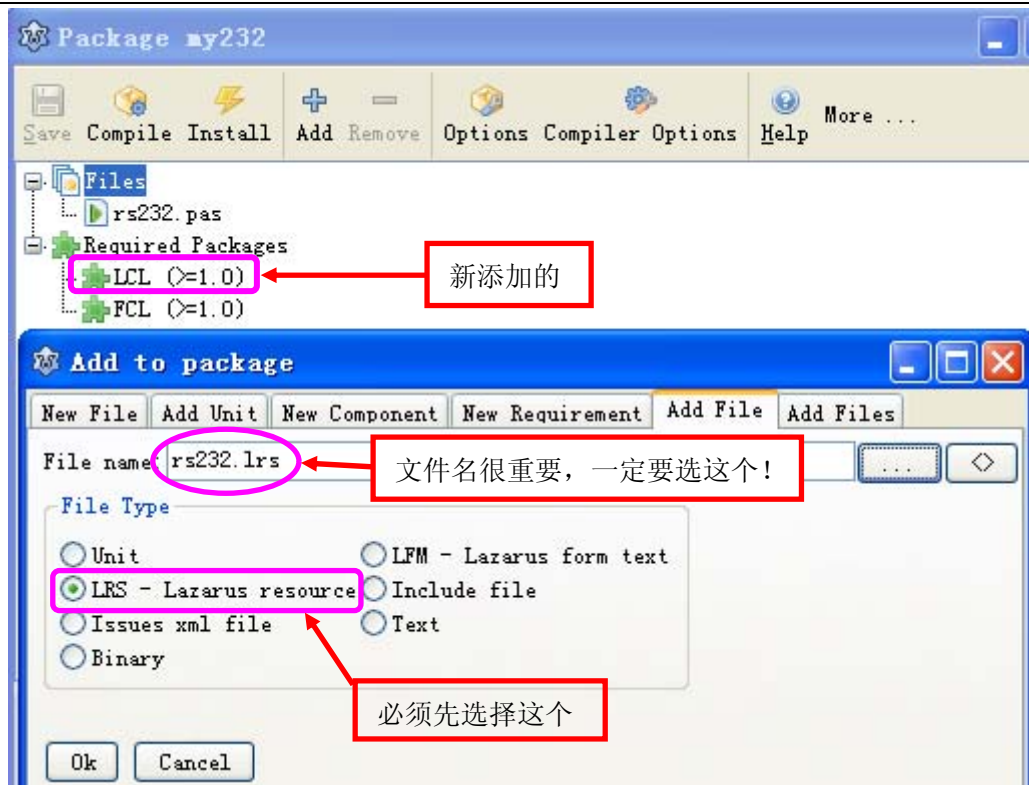


图 17

好了，至此，图标的创建完毕了，图 18 就是最终的效果了。

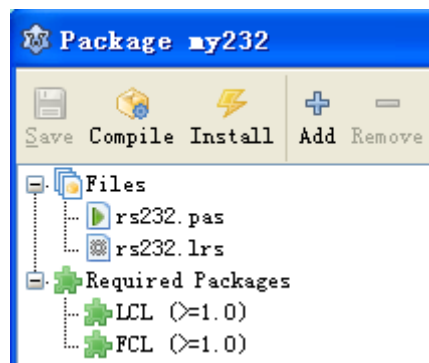


图 18

1.2.4 编写控件代码

经过了上面的操作，控件包的框架基本搭建完毕了，下面就给控件添加血肉，赋予灵魂吧。

根据前面的需求分析，增添数据的枚举类型：

```
1      type
2          TBaudRate=(CBR_4800, CBR_7200, CBR_9600, CBR_14400, CBR_19200);
3          TDataBits=(db8bits, db7bits, db6bits, db5bits);
4          TParity=(pNone, pOdd, pEven, pMark, pSpace);
5          TStopBits=(sbOne, sbTwo);
```

其中：

第 2 行定义了串口的几种常用的波特率；

第 3 行定义了串口的数据位类型；

第 4 行定义了串口的校验类型；

第 5 行定义了串口的停止位类型。

再增添数据常量：

```
1      const
2          dcb_Binary = $00000001;
3          ConstsBaud: array[TBaudRate] of integer =
4              (4800, 7200, 9600, 14400, 19200);
5          ConstsBits: array[TDataBits] of integer = (8, 7, 6, 5);
6          ConstsParity: array[TParity] of integer = (0, 1, 2, 3, 4);
7          ConstsStopBits: array[TStopBits] of integer=(0, 2);
```

这里的第 3 ~ 6 行和前面的枚举类型是对应的。

下面完善串口类的定义：

```
1      type
2          TRS232 = class(TComponent)
3      private
```

```
4          { Private declarations }
5          FCommName :string;
6          FBaudRate : TBaudRate;
7          FDataBits : TDataBits;
8          FParity    : TParity;
9          FStopBits : TStopBits;
10         protected
11         { Protected declarations }
12         hComm: THandle;
13         public
14         { Public declarations }
15         constructor Create(AOwner: TComponent);      override;
16         function  OpenPort:integer;                  virtual;
17         function  _SetCommState:boolean;              virtual;
18         procedure SetCommName(Value:string);          virtual;
19         procedure SetBaudRate(Value: TBaudRate);      virtual;
20         procedure SetDataBits(Value: TDataBits);      virtual;
21         procedure SetParity(Value: TParity);          virtual;
22         procedure SetStopBits(Value: TStopBits);      virtual;
23         procedure Send(str:ansiString);              virtual;
24         function  Receive():ansiString;              virtual;
25         procedure closePort;                          virtual;
26         destructor Destroy;                          override;
27         published
28         { Published declarations }
29         property CommName: string    read FCommName write FCommName;
```

```
30         property BaudRate: TBaudRate read FBaudRate write SetBaudRate;
31         property DataBits: TDataBits read FDataBits write SetDataBits;
32         property Parity: TParity read FParity write SetParity;
33         property StopBits: TStopBits read FStopBits write SetStopBits;
34     end;
```

下面新控件的注册。为控件建立了一个新的选卡，并将控件放在选卡上：

```
1     procedure Register;
2     begin
3         RegisterComponents('fpccn', [TRS232]);
4     end;
```

构造函数，给相关的特性赋予初始值：

```
1     constructor TRS232.Create(AOwner: TComponent);
2     begin
3         FCommName:='';
4         FBaudRate :=CBR_9600;
5         FDataBits :=db8bits;
6         FParity   :=pNone;
7         FStopBits :=sbOne;
8         inherited;
9     end;
```

打开串口，准备干活了：

```
1     function TRS232.OpenPort:integer;
2     var
```



```
3      {$ifdef WINCE}
4          lpFileName : LPCWSTR;
5      {$else}
6          lpFileName : pchar;
7      {$endif}
8      begin
9          if (FCommName='') then
10              begin
11                  result:=1;
12                  exit;
13              end
14          else begin
15              {$ifdef WINCE}
16                  lpFileName:=pwidechar(utf8decode(FCommName)+' :');
17              {$else}
18                  lpFileName:=pchar(FCommName);
19              {$endif}
20              end;
21          try
22              hComm:=CreateFile(lpFileName, GENERIC_READ or GENERIC_WRITE,
23              0, nil, OPEN_EXISTING, 0, 0);
24          except
25              if (hComm=INVALID_HANDLE_VALUE) then
26                  begin
27                      result:=2;
28                      exit;
```

```
28         end;
29     end;
30     if (_SetCommState=false) then
31     begin
32         closehandle(hcomm);
33         result:=3;
34         exit;
35     end;
36     result:=0;
37 end;
```

这里解释一下：上面的第 3，4，7 等程序行的代码是宏定义，跨平台用的。

设置串口的各个参数，这里的细节可以跳过去：

```
1     function TRS232._SetCommState:boolean;
2     var
3         cc:TCOMMCONFIG;
4     begin
5         GetCommState(hComm, cc.dcb);
6         cc.dcb.DCBlength := SizeOf(cc.dcb);
7         cc.dcb.BaudRate := ConstsBaud[FBaudRate];
8         cc.dcb.ByteSize := ConstsBits[FDataBits];
9         cc.dcb.parity := ConstsParity[FParity];
10        cc.dcb.StopBits := ConstsStopBits[FStopBits];
11        cc.dcb.Flags := dcb_Binary;
12        if not SetCommState(hComm, cc.dcb) then
13        begin
14            result:=false;
```

```
15         end
16         else result:=true;
17     end;
```

在每次打开串口，以及在设置串口控件的特性时，必须调用此程序进行处理。

如要更换串口，在这里处理：

```
1     procedure TRS232.SetCommName(Value:string);
2     begin
3         if (FCommName <> Value) then
4         begin
5             FCommName := Value;
6         end;
7     end;
```

如要改变波特率，在这里处理：

```
1     procedure TRS232.SetBaudRate(Value: TBaudRate);
2     begin
3         if (FBaudRate<>Value) then
4         begin
5             FBaudRate:=Value;
6             if (_SetCommState=false) then hcomm:=INVALID_HANDLE_VALUE;
7         end;
8     end;
```

如要改变数据位，在这里处理：：

```
1     procedure TRS232.SetDataBits(Value: TDataBits);
```

```
2      begin
3          if (FDataBits<>Value) then
4              begin
5                  FDataBits:=Value;
6                  if (_SetCommState=false) then hcomm:=INVALID_HANDLE_VALUE;
7              end;
8      end;
```

如要改变检验，在这里处理：

```
1      procedure TRS232.SetParity(Value: TParity);
2      begin
3          if (FParity<>Value) then
4              begin
5                  FParity:=Value;
6                  if (_SetCommState=false) then hcomm:=INVALID_HANDLE_VALUE;
7              end;
8      end;
```

如要改变停止位，在这里处理：

```
1      procedure TRS232.SetStopBits(Value: TStopBits);
2      begin
3          if (FStopBits<>Value) then
4              begin
5                  FStopBits:=Value;
6                  if (_SetCommState=false) then hcomm:=INVALID_HANDLE_VALUE;
7              end;
```

```
8         end;
```

通过串口发送数据：

```
1         procedure TRS232.Send(str:ansistring);
2
3         var
4             lrc:LongWord;
5
6         begin
7             if (hComm=INVALID_HANDLE_VALUE) then exit;
8             lrc:=0;
9             WriteFile(hComm,str[1],Length(str), lrc, nil);
10
11         end;
```

从串口缓冲区取出接受到的数据：

```
1         function TRS232.Receive():ansiString;
2
3         var
4             inbuff: array[0..250] of ansiChar='';
5             nBytesRead, dwError:LongWORD ;
6             cs:TCOMSTAT;
7
8         begin
9             nBytesRead:=0;
10            dwError:=0;
11            ClearCommError(hComm, dwError, @cs);
12            if cs.cbInQue > sizeof(inbuff) then
13            begin
14                PurgeComm(hComm, PURGE_RXCLEAR);
15                exit;
```

```
14         end;
15         ReadFile(hComm, inbuff, cs.cbInQue, nBytesRead, nil);
16         result:=Copy(inbuff, 1, cs.cbInQue);
17     end;
```

关闭打开的串口，释放站用的资源：

```
1     procedure TRS232.closeport;
2     begin
3         if (hComm=INVALID_HANDLE_VALUE) then exit;
4         SetCommMask(hcomm, $0);
5         CloseHandle(hComm);
6     end;
```

析构函数：

```
1     destructor TRS232.Destroy;
2     begin
3         inherited Destroy;
4     end;
```

至此，控件的灵魂已经成型了。

1.2.5 安装控件

安装控件前，先编译一下，是个好习惯，下图显示了步骤：

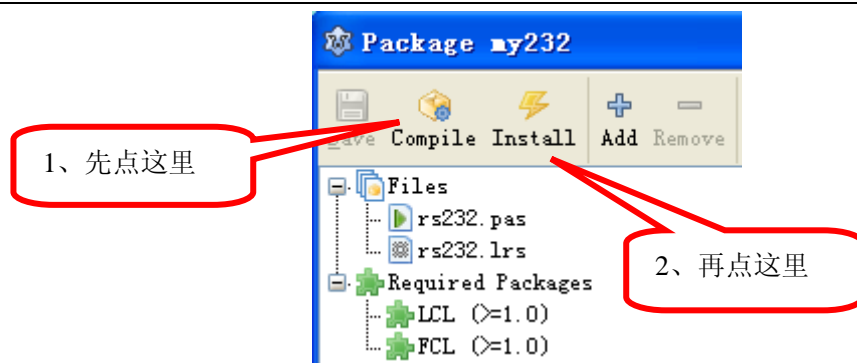


图 19

下图显示成功安装了：

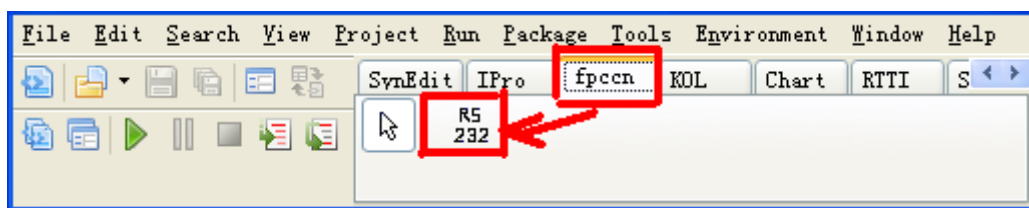


图 20

1.3 控件的使用

在 LAZARUS 里新建一个工程，将上面建立的 RS232 控件拖放到窗体上，接着拖放其它的控件，如图 21 所示。

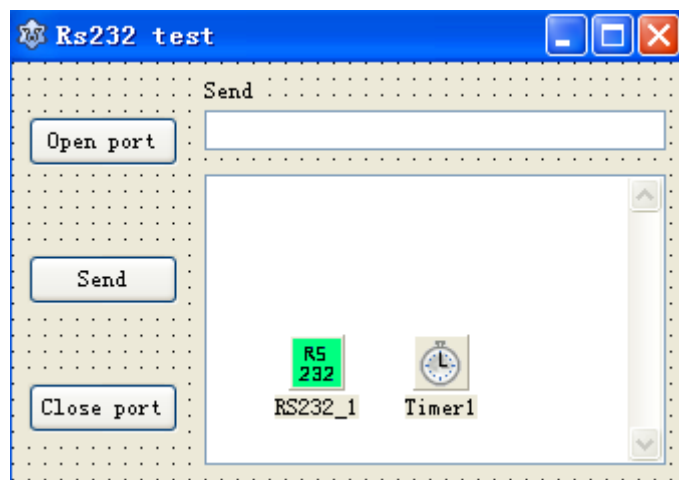


图 21

窗体的具体设置为：

```
1      object Form1: TForm1
2          Left = 518
3          Height = 209
4          Top = 241
5          Width = 335
6          BorderIcons = [biSystemMenu]
7          BorderStyle = bsDialog
8          Caption = 'Rs232 test'
9          ClientHeight = 209
10         ClientWidth = 335
11         Position = poDesktopCenter
12         LCLVersion = '0.9.28.2'
13     object Mem1: TMemo
14         Left = 96
15         Height = 145
16         Top = 56
17         Width = 230
18         ScrollBars = ssAutoBoth
19         TabOrder = 0
20     end
21     object LabeledEdit1: TLabeledEdit
22         Left = 96
23         Height = 20
24         Top = 24
25         Width = 230
26         EditLabel.AnchorSideLeft.Control = LabeledEdit1
```



```
27         EditLabel.AnchorSideBottom.Control = LabeledEdit1
28         EditLabel.Left = 96
29         EditLabel.Height = 13
30         EditLabel.Top = 8
31         EditLabel.Width = 25
32         EditLabel.Caption = 'Send'
33         EditLabel.ParentColor = False
34         TabOrder = 1
35     end
36     object Button1: TButton
37         Left = 8
38         Height = 25
39         Top = 27
40         Width = 75
41         Caption = 'Open port'
42         OnClick = Button1Click
43         TabOrder = 2
44     end
45     object Button2: TButton
46         Left = 8
47         Height = 25
48         Top = 96
49         Width = 75
50         Caption = 'Send'
51         Enabled = False
52         OnClick = Button2Click
```

```
53         TabOrder = 3
54     end
55     object Button3: TButton
56         Left = 8
57         Height = 25
58         Top = 160
59         Width = 75
60         Caption = 'Close port'
61         Enabled = False
62         OnClick = Button3Click
63         TabOrder = 4
64     end
65     object RS232_1: TRS232
66         CommName = 'COM1'
67         BaudRate = CBR_9600
68         DataBits = db8bits
69         Parity = pNone
70         StopBits = sbOne
71         left = 136
72         top = 136
73     end
74     object Timer1: TTimer
75         Enabled = False
76         Interval = 200
77         OnTimer = Timer1Timer
78         left = 200
```

```
79         top = 136
80     end
81 end
```

增添用于打开串口的代码：

```
1     procedure TForm1.Button1Click(Sender: TObject);
2     begin
3         if (RS232_1.OpenPort<>0) then
4         begin
5             showmessage('open port fail !');
6         end
7         else begin
8             Button1.Enabled:=false;
9             Button2.Enabled:=true;
10            Button3.Enabled:=true;
11            timer1.Enabled:=true;
12        end;
13    end;
```

增添从串口发送数据的代码：

```
1     procedure TForm1.Button2Click(Sender: TObject);
2     begin
3         If (LabeledEdit1.Text<>'') then
4             RS232_1.Send(utf8decode(LabeledEdit1.Text))
5         else showmessage('Nothing can be send !');
6     end;
```

增添关闭串口的代码：

```
1      procedure TForm1.Button3Click(Sender: TObject);
2
3      begin
4          RS232_1.closePort;
5          Button2.Enabled:=false;
6          Button3.Enabled:=false;
7          timer1.Enabled:=false;
8      end;
```

增添定时器的代码：

```
1      procedure TForm1.Timer1Timer(Sender: TObject);
2
3      var
4          display:string;
5
6      begin
7          display:=RS232_1.Receive();
8          if (display<>'' ) then  memol.Append(display);
9      end;
```

定时器代码的作用是按照一定的时间间隔将串口的数据取出来。

根据串口处理的数据量，改变定时器的定时时间，就可以做到数据不丢失，效率也不错，足以应付一般的应用了。

为了检验串口是否正常工作，还需要第三方的软件进行合作，从中文社区 www.fpcn.com 下载 commport 软件，版本号为：v0.22 。

图 22 显示了串口数据相互传送的情况，Rs232 test 接收区数据是 commport 发送区的数据：1234567890；Rs232 test 发送区的数据：“abcdefg”与 commport 的接收区的数据吻合。

发送和接收的数据都正确，从而证明前面制作的 RS232 控件是正确的。

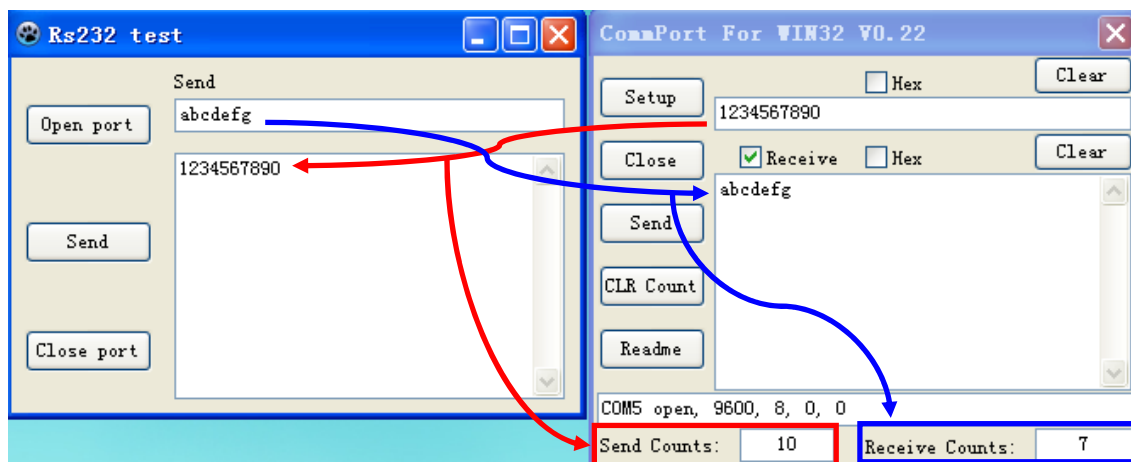


图 22

注意：前面制作的 RS232 控件主要应用于三线连接的串口，即使用： TXD，RXD，GND。

第 2 章

消息编程

2.1 消息类

2.1.1 消息的构成

在 Windows 环境里，消息无所不在。如果不理解 Windows 消息处理机制，将无法深入的理解 Windows 编程。消息，就是指 Windows 发出的一个通知，告诉应用程序某个事情发生了。例如，单击鼠标、改变窗口尺寸、按下键盘上的一个键都会使 Windows 发送一个消息给应用程序。消息本身是作为一个记录传递给应用程序的，这个记录中包含了消息的类型以及其它信息。例如，对于单击鼠标所产生的消息来说，这个记录中包含了单击鼠标时的坐标。这个记录类型叫做 MSG。

它在 LAZARUS 的 Messages.inc 文件中是这样声明的：

```
1      type
2      MSG = record
3          hwnd    : HWND;      // 接受消息的窗口句柄
4          message: UINT;      // 消息常量标识符
5          wParam : WPARAM ;    // 32 位消息的特定附加信息
6          lParam : LPARAM ;    // 32 位消息的特定附加信息
7          time    : DWORD;     // 消息创建时的时间
8          pt      : Point;     // 消息创建时的鼠标位置
9      end;
```

也就是说，对于任何一个消息，都有一个 MSG 变量与之对应，该变量包含了消息的相关信息。而我们在一般情况下，只使用消息的消息标识符，该标识符也

唯一地代表了这个消息。

Windows 消息控制中心一般是三层结构，其顶端就是 Windows 内核。Windows 内核维护着一个消息队列，第二级控制中心从这个消息队列中获取属于自己管辖的消息，后做出处理，有些消息直接处理掉，有些还要发送给下一级窗体(Window)或控件(Control)。第二级控制中心一般是各 Windows 应用程序的 Application 对象。第三级控制中心就是 Windows 窗体对象，每一个窗体都有一个默认的窗体过程，这个过程负责处理各种接收到的消息。

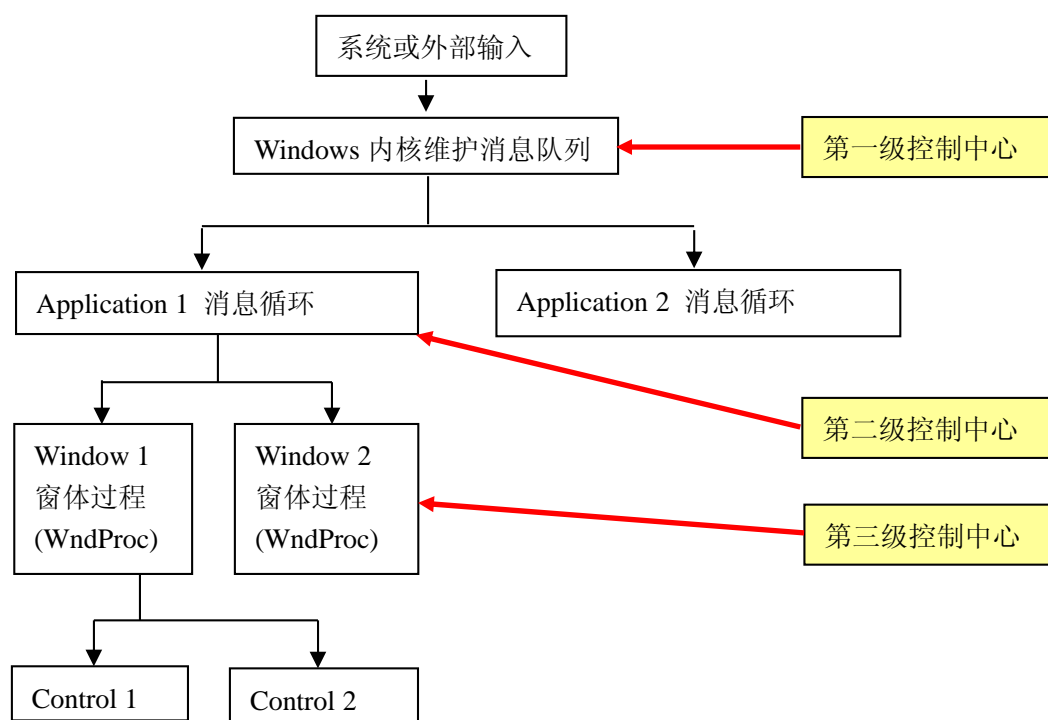


图 2.1-1 Windows 消息多级控制中心

2.1.2 消息的句柄

不是每个控件都能接收消息，转发消息和绘制自身，只有具有句柄（handle）的控件才能做到。有句柄的控件本质上都是一个窗体(window)，它们可以独立存在，可以作为其它控件的容器，而没有句柄的控件，如 Label，是不能独立存在的，只能作为窗口控件的子控件，它不能绘制自身，只能依靠父窗体将它绘制来。

句柄的本质是一个系统自动维护的32位的数值，在整个操作系统的任一时刻，这个数值是唯一的。但该句柄代表的窗体释放后，句柄也会被释放，这个数值又可能被其它窗体使用。也就是说，句柄的数值是动态的，它本身只是一个唯一性标识，操作系统通过句柄来识别和查找它所代表的对象。

然而，并非所有的句柄都是窗体的句柄，Windows 系统中还中很多其它类型的句柄，如画布(hdc)句柄，画笔句柄，画刷句柄，应用程序句柄(hInstance)等。这种句柄是不能接收消息的。但不管是哪种句柄，都是系统中对象的唯一标识。本文只讨论窗体句柄。

那为什么句柄使窗口具有了如此独特的特性呢？实际是都是由于消息的原因。由于有了句柄，窗体能够接收消息，也就知道了该什么时候绘制自己，绘制子控件，知道了鼠标在什么时候点击了窗口的哪个部分，从而作出相应的处理。

2.1.3 消息的传送

1、从消息队列获取消息：

可以通过 PeekMessage 或 GetMessage 函数从 Windows 消息队列中获取消息。Windows 保存的消息队列是以线程(Thread)来分组的，也就是说每个线程都有自己的消息队列。

2、发送消息

发送消息到指定窗体一般通过以下两个函数完成：SendMessage 和 PostMessage。两个函数的区别在于：PostMessage 函数只是向线程消息队列中添加消息，如果添加成功，则返回 True，否则返回 False，消息是否被处理，或处理的结果，就不知道了。而 SendMessage 则有些不同，它并不是把消息加入到队列里，而是直接翻译消息和调用消息处理，直到消息处理完成后才返回。所以，如果我们希望发送的消息立即被执行，就应该调用 SendMessage。

还有一点，就是 SendMessage 发送的消息由于不会被加入到消息队列中，所以通过 PeekMessage 或 GetMessage 是不能获取到由 SendMessage 发送的消息。

另外，有些消息用 PostMessage 不会成功，比如 wm_settext。所以不是所有的消息都能够用 PostMessage 的。

还有一些其它的发送消息 API 函数，如 PostThreadMessage，SendMessageCallback，

SendMessageTimeout, SendNotifyMessage 等。

2.1.4 消息的取值

Windows 操作系统已经给我们定义了大量的消息，这些消息我们称为系统消息。除了系统消息，我们还可以自己定义消息，即自定义消息。

值小于\$0400 的消息都是系统消息，自定义消息一般都大于\$0400。

系统消息取值一般有如下规律，如表 1：

取值范围	含义
\$0001——\$0087	主要是窗口消息
\$00A0——\$00A9	非客户区消息
\$0100——\$0108	键盘消息
\$0111——\$0126	菜单消息
\$0132——\$0138	颜色控制消息
\$0200——\$020A	鼠标消息
\$0211——\$0213	菜单循环消息
\$0220——\$0230	多文档消息
\$03E0——\$03E8	DDE 消息
\$0400	WM_USER
\$0400——\$7FFF	自定义消息

2.1.5 队列消息和非队列消息

Windows 把消息分为两种：一种是需要立即处理的消息，另一种是不需要立即处理的消息。

对于需要立即处理的消息，Windows 直接把它送给窗口的消息处理函数进行处理，这类消息我们叫做非队列消息；

而对于不需要立即处理的消息，Windows 会把它发送给应用程序的消息队列进行排队，由应用程序逐个进行处理，我们把这类消息叫做队列消息。

- 1、Windows 操作系统有一个消息队列，它存放操作系统收到的消息。如：当按键被按下，键盘会发送一个消息到操作系统的消息队列。
- 2、操作系统把系统消息队列中的消息分派到各个应用程序的消息队列。如果它是第1个应用程序

序的消息，操作系统把它发给第1个应用程序，把它放在第1个应用程序的消息队列；如果是第2个应用程序的消息，发送给第2个程序的消息队列。

3、应用程序的消息循环从自己的消息队列中取消息，取出的消息调用窗口过程函数进行处理。

4、PostMessage 是寄送消息，函数执行后立即返回。寄送的消息是队列消息，放在程序的消息队列中排队处理。一般来说，新寄送的消息排在消息队列的末尾，这样可以保证窗口以先进先出的顺序处理消息。

SendMessage 是发送消息，它发出的消息是非队列消息，直接调用窗口过程函数处理。

SendMessage 函数一直等消息处理完成后才返回。

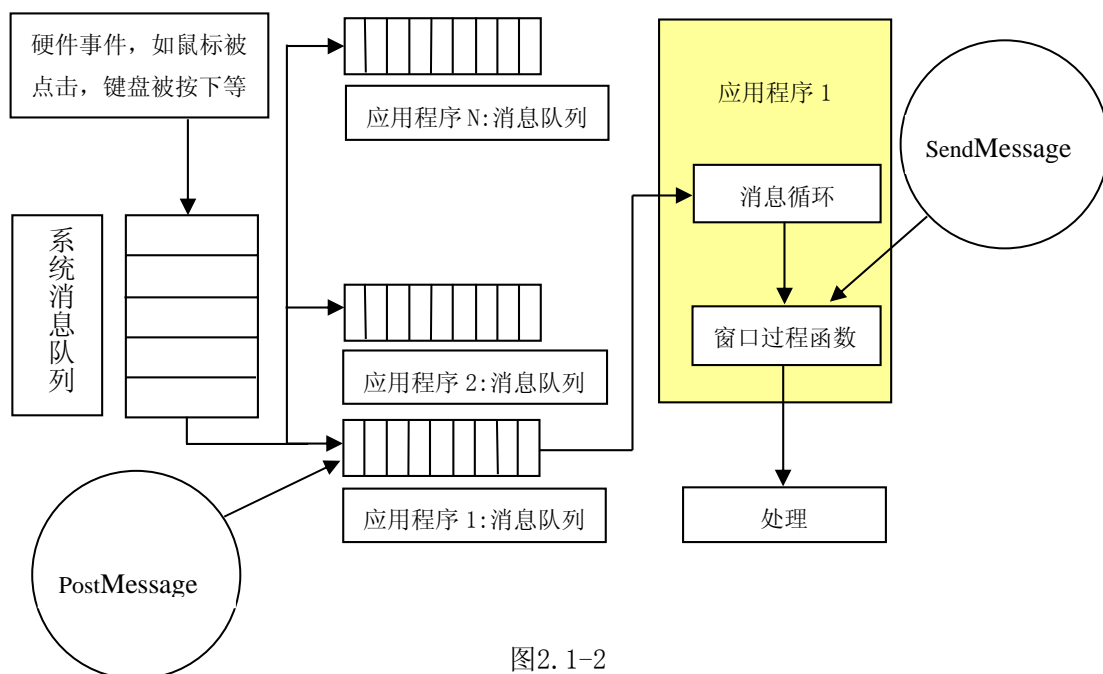


图2.1-2

2.2 自定义消息

2.2.1 自定义消息的创建

Windows 的消息多数在窗体和控件之间传递，而在很多的应用场合，我们需要在后台产生、

处理一些不需要经过控件的消息，在前面的表 1 中，有个“自定义消息”，它正好符合我们的要求，它的取值范围为：\$0400——\$7FFF。

首先，声明使用消息单元；然后定义所用到的自定义消息；接着定义消息处理过程；最后编写自定义消息处理过程。

创建的过程如下：

1、在 `uses` 里添加： `Messages`;

2、定义所用到的自定义消息

`Const`

`my1=WM_USER+1; // 注意：使用了表 1 中的 WM_USER`

3、定义消息处理过程

`procedure show_1(var MSG: tMSG);message my1;`

`// 过程 show_1 是处理消息 my1 的`

4、编写自定义消息处理过程

`procedure TForm1.show_1(var MSG: tMSG);`

`begin`

`edit1.Text:=edit1.Text+'p'; // 实际处理代码`

`end;`

2.2.2 自定义消息的例子

新建一个工程，在窗体放一个编辑框 `edit1` 和一个按钮 `button1`，如图 2.2-1

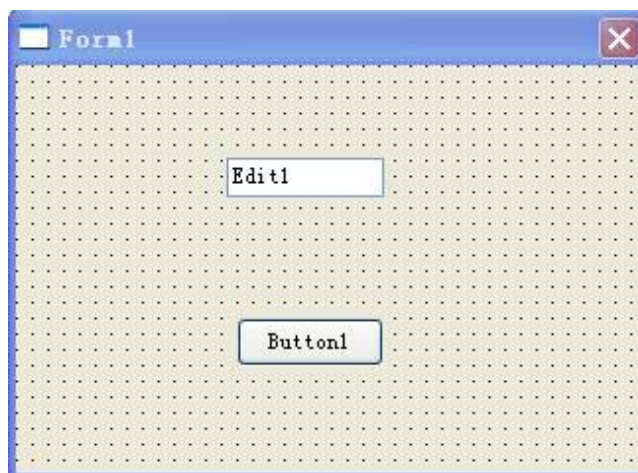


图 2.2-1

按 5.2.1 中的过程创建自定义消息。

双击按钮，添加事件：

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    PostMessage(self.Handle, my1, 1, 1);  
end;
```

在 LAZARUS 的 IDE 里运行该工程，双击 button1，可以看到编辑框 edit1 里不断增加字符“p”。

完整的工程代码可在本书的配书代码中找到，文件名为：self_message.rar。

小结：灵活应用自定义消息，能让编程变得更轻松自如，同时也提升了编程的功力。

第 1 章

使用 Lazarus 进行 Linux 开发

在 Linux 应用程序开发方面，Lazarus 能开发几乎所有类型的应用程序。特别在桌面开发方面，Lazarus 在 Gtk, Qt 等流行的 GUI 库的基础上构建了 LCL 库，不仅隐藏了各个不同界面库之间的差异，而且拥有更加丰富的控件，使得其开发图形界面应用程序的速度远远快于其它流行的 GUI 库，并且开发出来的程序只需重新编译就可以适应各种桌面环境（例如，一个原先基于 Gtk 运行于 GNOME 的 LCL 应用程序只需要重新编译就可以变成基于 Qt 运行在 KDE 上的应用程序，并且可以获得 native 外观）。

本章使用 Ubuntu 10.04 LTS AMD64（使用 GNOME 桌面系统），对 Lazarus 在 Linux 下的使用，做一个简要的介绍。至于 Lazarus 在其它发行版下的使用，也与此类似。

1.1 在 Linux 下搭建 Lazarus 开发环境

由于 Lazarus 是基于 Free Pascal 的 IDE，本身并没有包括编译器。所以首先第一步是要安装好 Free Pascal Compiler。

1.1.1 Free Pascal 编译器的安装

目前大多数流行的 Linux 发行版都带有软件仓库，如 Debian 的 apt, RHEL 的 yum, Gentoo 的 Portage, 等等，使得 Linux 下软件安装变得十分方便。对于大多数流行的 Linux 发行版来说，Free Pascal 编译器均可以从软件仓库进行安装。由于各个发行版的差别比较大，大部分 Linux 发行版都会对各种软件做出相应的适应性修改，因而相对于从官方网站下载 Free Pascal 编译器来说，从发行版自带的软件仓库安装可以获得和相应 Linux 发行版更好的结合性和稳定性。

单击全局菜单“系统”->“系统管理”->“新立得软件包管理器”，启动“新力得软件包

管理器”。对于喜欢用控制台的朋友，在控制台执行命令“gksu synaptic”也同样可以启动“新力得软件包管理器”。启动后界面如图 3.1 所示。



图1.1 新力得软件包管理器

安装的步骤如下：

1. 确保网络能够正常使用。
2. 在右边的列表框中，双击“fpc”一项，此时弹出如图3.2的窗口。

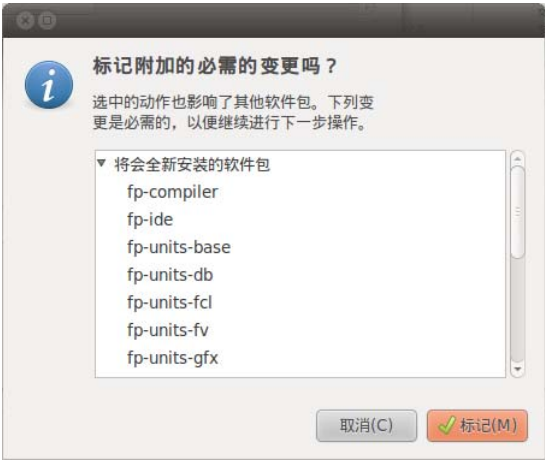


图 1.2 询问是否需要标记 fpc 的依赖项

3. 在弹出的对话框中，单击“标记”。完成后所有的 Free Pascal 编译器程序包括所有官方自带的库都被选中。接下来再将文档和 Free Pascal 编译器的源代码给选中。

- 4. 在“fpc-source”一项双击。
- 5. 在“fp-docs”一项双击。完成后如图3.3所示。

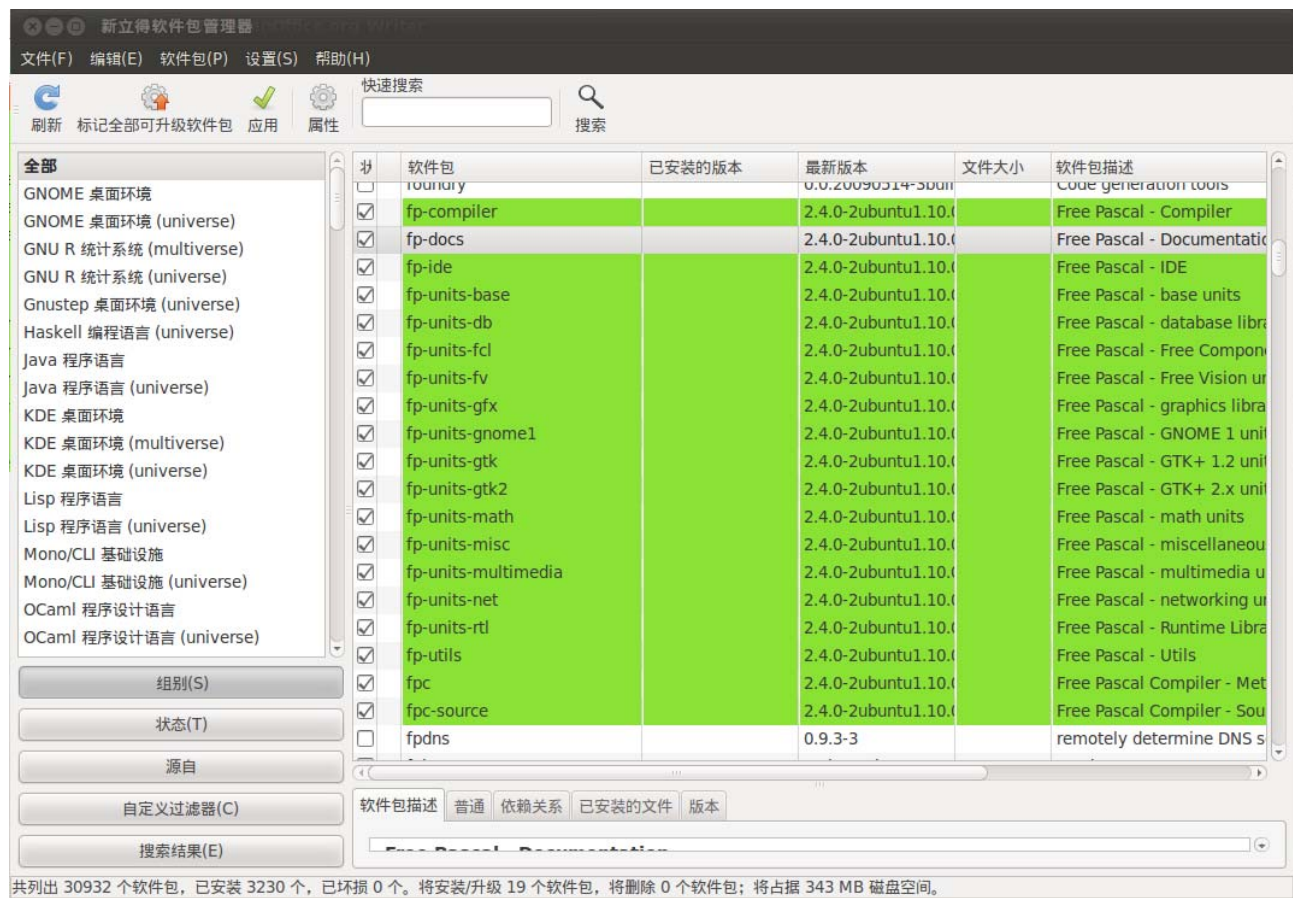


图1.3 已标记整套Free Pascal 编译器、自带库、开发文档、源代码

- 6. 单击“应用”按钮，弹出“摘要”对话框。
- 7. 在弹出的“摘要”对话框中，单击“应用”按钮，弹出“正在下载软件包文件”对话框。
- 8. 稍等一会儿，“新力得软件包管理器”会自动从网络上下载整套 Free Pascal 编译器并将其安装到系统中。完成后将弹出“变更已应用”对话框。如图1.4所示。



图1.4 Free Pascal 编译器已经安装完成。

1.1.2 安装 Lazarus

由于 Lazarus 已经包含于大多数发行版中，Lazarus 的安装同样可以从软件仓库自动下载安装。但是由于 Lazarus 本身的特点，实际应用的开发当中，并不推荐从软件仓库里面安装 Lazarus（包括直接使用包管理器来安装同样是不推荐的）。因为如果从软件仓库安装 Lazarus 的话，若不是以 root 用户的身份进行开发，使用上将会有很多限制；但若是以 root 用户身份进行开发，则容易对系统级的 Lazarus 进行修改，从而对相关的一些其它用户、软件造成影响。解决方法是直接从 Lazarus 的源代码编译安装。

下面首先介绍从软件仓库安装 Lazarus 的方法。

1.1.2.1 从软件仓库安装 Lazarus

从软件仓库安装 Lazarus 的方法与安装 Free Pascal 编译器类似，唯一的不同是在“新力得软件包管理器”里面，要标记名为“lazarus”的软件包。

安装完成后，可以通过全局菜单“应用程序”→“编程”→“Lazarus”来启动“Lazarus”。

1.1.2.2 从源代码编译安装

从源代码编译安装 Lazarus 将获得最大的灵活性，而且编译过程简单、方便。实际上，Lazarus 编译完成后即可使用，根本无需任何安装步骤。

1.1.2.2.1 编译前的准备

除了 Free Pascal 编译器，编译 Lazarus 还需要一些相关的工具。

首先是 GNU Make，Lazarus 的编译脚本需要它来执行，这个软件可以通过在“新力得软件包管理器”里面安装“make”软件包来获得。

其次是 GDB, Lazarus 本身并没有实现调试功能, 它的调试界面是通过调用 gdb 调试器来实现的, 这个软件可以通过在“新力得软件包管理器”里面安装“gdb”软件包来获得。

除了以上讲的两个, 还需要 GTK2 及其开发库, 不过目前绝大多数使用 GNOME 桌面系统的 Linux 发行版中已经自带 GTK2, 其开发库可以通过在“新力得软件包管理器”里面安装

“libgtk2.0-dev”软件包来获得。这里有一点需要说明的是, 目前尽管 Lazarus 已经支持大部分流行的 GUI 库, 在很长一段时间里面, Lazarus 官方最推荐也是最成熟稳定的开发界面也是基于 GTK1 这个十分古老的 GUI 库的 Lazarus 界面, 因而目前在 Lazarus 自带的安装说明文档(docs/INSTALL.txt)里面, GTK1 也是必须的(实际上除非想要使用 GTK1 界面, 否则不是必须的)。

最后, 我们还要需要从 Lazarus 的官方网站下载其源代码。假设下载到的 Lazarus 源代码的文件名是 lazarus-0.9.30-src.tar.bz2, 将该源代码文件解压缩到“主文件夹”(也就是“~”), 会产生一个叫“lazarus”的目录。

1.1.2.2 编译

从全局菜单选择“应用程序”->“附件”->“终端”, 启动“GNOME 终端”。下文的命令部分, 都是在该程序里面输入的。

1. 执行命令“cd ~/lazarus”, 切换到解压后 Lazarus 所在的目录。
2. 执行命令“make clean all”, 编译 Lazarus。
3. 若一切顺利, 稍等片刻 Lazarus 即可编译完成。若编译过程出错, 则可以根据屏幕输出信息来解决。

完成后, Lazarus 的安装路径便是“~/lazarus”, 下文没有做说明均假设 Lazarus 安装在该目录。

1.1.2.3 启动 Lazarus

编译完成后, 在~/lazarus 目录里执行命令“./startlazarus”即可启动 Lazarus。但实际上, 并非每次启动都需要切换到终端来执行此命令, 用“文件浏览器”打开 lazarus 所在文件夹, 然后双击“startlazarus”图标也同样能够启动 Lazarus。

启动后如图 1.5 所示。

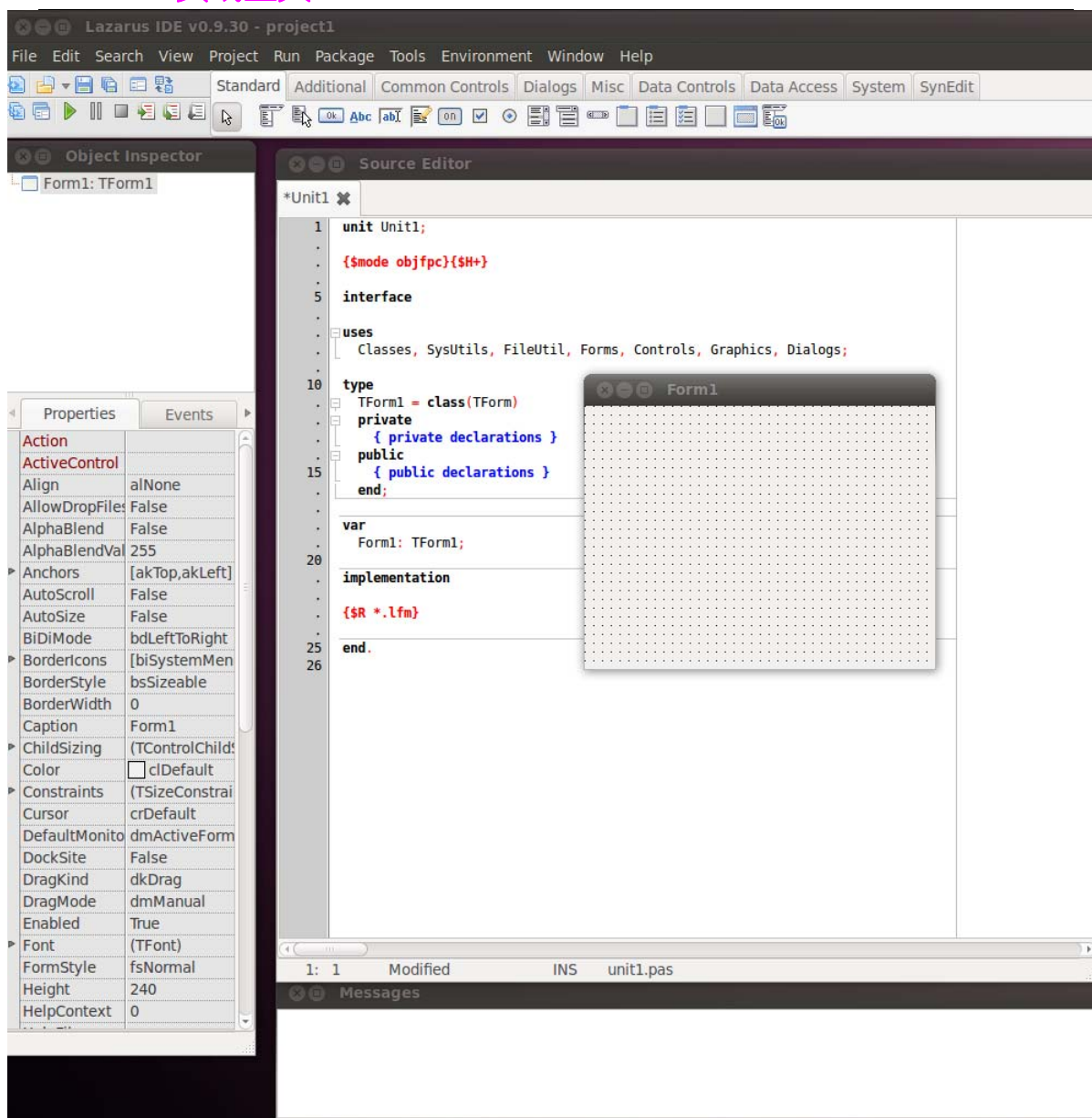



图1.5 启动后的 Lazarus 界面（基于 GTK2库）。

1.2 实战：简单的 Linux 应用程序

Lazarus 提供了一个平台通用的开发方式和界面库。即使是在 Linux 平台下，其操作方式和代码也与其它平台几乎一样。下面请启动 Lazarus，来体验一下 Linux 开发的乐趣。

1.2.1 第一个 Linux 桌面程序

如图 1.5 所示，启动 Lazarus 以后，已经自动生成了一个新的桌面应用程序工程，该工程

包含了一个窗体 Form1 以及相应的代码。此时如果单击 “” 按钮，则会自动编译和运行该工程。

现在让我们给这个窗体增加一些功能。假设现在要做一个这样的程序：提供一个文本框，让用户输入一个正整数 N，单击一个按钮以后，计算 $1+2+3+4+\dots+N$ 的值，然后告诉用户结果。首先给在窗体上放置控件，完成后如图 1.6 所示。



图1.6 放好控件的窗体 Form1。

对应的窗体代码如下：

```
object Form1: TForm1
  Left = 531
  Height = 89
  Top = 355
  Width = 320
  Caption = 'Form1'
  ClientHeight = 89
  ClientWidth = 320
  LCLVersion = '0.9.30'
  object btnCompute: TButton
    Left = 189
    Height = 25
    Top = 56
    Width = 75
    Caption = '计算'
    TabOrder = 0
```

```
end

object Edit1: TEdit

    Left = 136

    Height = 27

    Top = 16

    Width = 168

    TabOrder = 1

end

object Label1: TLabel

    Left = 16

    Height = 18

    Top = 24

    Width = 118

    Caption = ' 请输入一个正整数: '

    ParentColor = False

end

end
```

双击“计算”按钮，将生成 OnClick 事件的代码，添加如下代码到 btnComputeClick。

```
procedure TForm1.btnComputeClick(Sender: TObject);
var
    N: Integer;
    Result : Int64;
begin
    try
        N := StrToInt(Edit1.Text);
```

```
if N < 1 then  
    raise EConvertError.Create('');  
  
Result := 0;  
for N := N downto 1 do  
    Result := Result + N;  
  
ShowMessage(IntToStr(Result));  
except  
on EConvertError do  
    begin  
        ShowMessage('请输入一个正整数!');  
        Edit1.Text := '';  
        Edit1.SetFocus;  
    end;  
end;  
end;
```

保存该工程（这里假设保存后的工程文件名为“project1.lpi”，主窗体单元文件名称为


“unit1.pas”），然后单击“”按钮，编译和运行该工程。运行效果如图3.7所示。



图1.7 运行工程 project1。

此时若用“文件浏览器”打开 project1.lpi 所在的目录，则会发现多了一个文件“project1”，该文件即为编译好的独立可运行的 project1 可执行文件，双击即可运行。由于 Linux 并非靠扩展名（实际上 Linux 根本没有“扩展名”的机制）来判断文件的类型，而且通常可执行文件都没有扩展名，因而生成的可执行文件的文件名是“project1”而不是“project1.exe”。

细心的读者可能会发现，编译好的程序 project1 大小居然达到 15MB（根据实际环境的不同，文件的大小会有所差异）之多！有没有办法变小呢？方法当然是有的。打开终端，运行切换到 project1 所在目录，运行如下命令：

```
strip --strip-all project1
```

于是，project1 一下子从 15MB 缩减到了 6MB（根据实际环境的不同，文件的大小会有所差异）。但是这种方法不能滥用。strip 的作用是将通常可执行程序在实际生产环境中极少用到的内容（如调试信息等）给删除掉，从而减少文件的大小。但是并非所有情况下这些内容都是无用的，因而有的程序经过 strip 处理以后，便无法正常运行了。

但是 6MB 相对于该程序的内容来说仍旧是个很大的数字，有没有办法减小呢？对于 Free Pascal 编译器来说，这确实是个难题。若确实这个大小对实际应用造成影响的话，那么笔者能想到的唯一办法只能是将该程序进行压缩，这可以用一些常见的压缩软件或者一些加壳软件来实现。

1.2.2 调试

承接上一节的例子，本节演示如何在 Linux 下对 Lazarus 应用程序进行调试。在进入接下来的内容之前，请完整重新编译上一节的程序 project1（请勿 strip，否则会丢失调试信息导致无法调试）。

1.2.2.1 在 Lazarus 开发环境里调试

Lazarus IDE 支持对一个 Free Pascal 应用程序进行源码级的调试，支持一切经典的调试方法，如“断点”、“单步执行”，等等。这些调试功能的用法，十分类似于其它流行的 IDE。例如要设置断点，只需要在要设置的源代码行之前单击。但是若是第一次设置断点，则可能会弹出如图 1.8 的对话框。

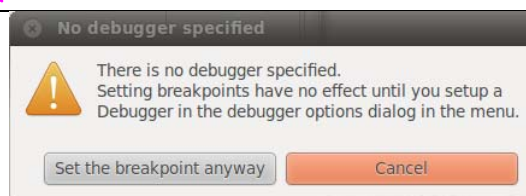


图1.8 由于没有设定调试器而弹出的警告。

的确，由于 Lazarus 的调试工能是通过调用外部调试器实现的，因而在使用前必须设置好外部调试器的信息。前边提到过，如果要使用 Lazarus 的调试功能则必须系统安装有 gdb 调试器，这是 Lazarus 唯一支持的调试器。使用如下步骤来设置调试器信息。

1. 单击菜单“Environment”->“Options”，打开“Options”对话框。
2. 在“Options”对话框中，从左边的列表选择“Debugger”->“General”。
3. 在右边的“Debugger type and path”框中，上面的下拉列表框中选择“GNU debugger (gdb)”，此时下边的下拉文本框会出现 gdb 所在路径：“/usr/bin/gdb”。对于绝大多数流行的 Linux 发行版来说，这个路径是正确的。若不正确，则请将该路径改成 gdb 所在的路径。完成后如图1.9所示。

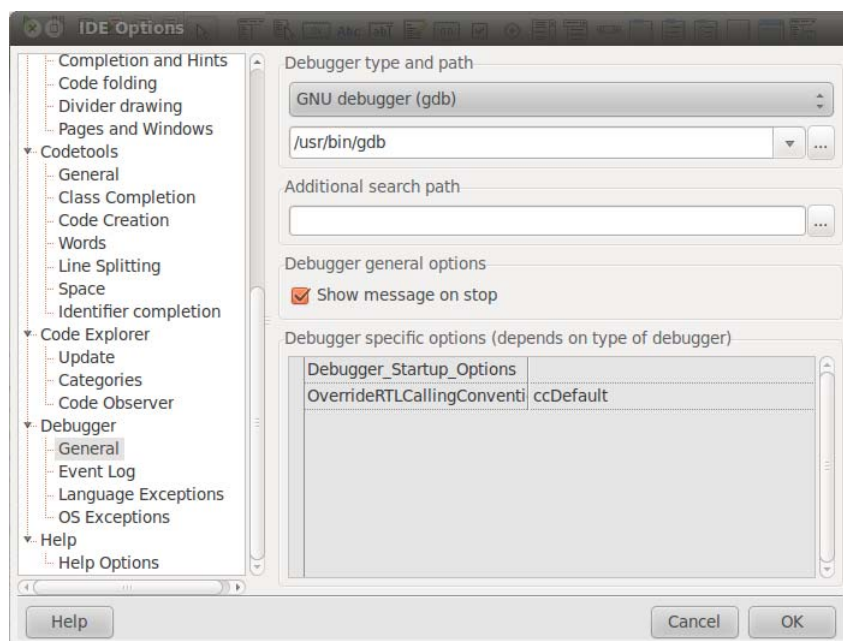


图1.9 设置 gdb 调试器。

4. 单击“OK”按钮使设置生效。

5. 设置好调试器后，再在源代码行前单击，断点就被设置好了。一个设置好的断点如图所示。

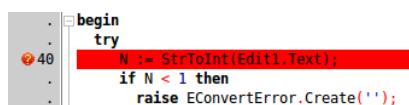


图1.10 设置好断点。

1.2.2.2 单独使用 gdb 进行调试

既然 Lazarus 是调用 gdb 来实现其调试功能的，那么如果不通过 Lazarus IDE，同样也可以使用 gdb 来进行调试。gdb 作为 Linux 下的标准调试器，功能十分丰富，被广泛应用在各个领域，甚至可以实现远程调试、自动调试等高级调试功能。在某些无法使用 Lazarus 来调试的场合，或者需要更高级调试功能的场合，则必须直接使用 gdb 调试器进行调试。

gdb 调试器是一个命令行界面的交互式程序，在大部分流行的 Linux 发行版里面，只要在命令行里面输入“gdb”就可以启动 gdb 调试器。这里以上一节实现的程序为例，简要介绍 gdb 的基本使用方法。完整的使用说明请参阅 gdb 手册。

1. 执行命令“gdb”，启动 gdb 调试器。启动后显示如下信息。

```
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb)
```

2. 执行命令“file project1”，指定要调试的程序文件为 project1。完成后输出信息如下。

```
(gdb) file project1
```



```
Reading symbols from
```

```
/home/lazarus_book/lazarus_examples/first_project/project1...done.
```

3. 此时可以查看当前的源代码。执行命令“1”，显示当前的源代码，如下所示。

```
(gdb) 1
7      cthreads,
8      {$ENDIF} {$ENDIF}
9      Interfaces, // this includes the LCL widgetset
10     Forms, Unit1
11     { you can add units after this };
12
13     {$R *.res}
14
15 begin
16     Application.Initialize;
```

由于程序还没有运行，当前的源代码为程序开头（也就是 program 的 begin 处）处的源代码。

4. 使用命令“break unit1.pas:45”，在 unit1.pas 的第45行处设置一个断点，如下所示。

```
(gdb) break unit1.pas:45
Breakpoint 1 at 0x4cfadc: file unit1.pas, line 45.
```

此处 unit1.pas 第45行也就是 TForm1.btnComputeClick 里面“for N := N downto 1 do”这一句所在的行。

尽管大多数时候如果仅仅使用 gdb 的命令行界面来调试的话，那么往往是使用“break <函数名>”的形式来设置断点的。如果使用这种形式的话，那么就必须知道

TForm1.btnComputeClick 被编译后的真实名称。根据官方文档：

- 所有的符号（函数名、变量名等都是符号）编译后均被转换为全大写的名称。比如，count 编译后就成了 COUNT。

- 方法名称，被翻译成“类名__方法名”。比如，TPoint.Draw 编译后就成了 TPOINT__DRAW。
- 方法如果可以拥有 self 变量，那么它的第一个参数是“this”（全小写），这个参数的值其实就是 self 变量的值。
- 如果是多个同名的 overload 方法，那么他们拥有相同的名称。这种情况下，只能通过指定文件名和行数来设置断点。

那么按照这种约定，方法 TForm1.btnComputeClick 被编译后应该变成 TForm1__BTNCOMPUTECLICK 了。但是实际情况有可能并非如此，比如笔者此处用的 fpc 版本为“2.4.0-2ubuntu1.10.04 [2011/06/17] for x86_64”，编译后函数 TForm1.btnComputeClick 的实际名为“UNIT1_TFORM1_\$__BTNCOMPUTECLICK\$TOBJECT”，因此，这里如果把 break 的参数换成这个实际的函数名成，那么也能起作用，如下。

```
(gdb) break 'UNIT1_TFORM1_$__BTNCOMPUTECLICK$TOBJECT'
Breakpoint 1 at 0x4cf9ea: file unit1.pas, line 38.
```

实际上，由于 Free Pascal 本身机制的问题，如果等到程序运行到一半再用“BTNCOMPUTECLICK”这个函数名称，同样能够起到相同的作用。

5. 执行命令“r”，从头开始执行 project1 程序，如下。

```
(gdb) r
Starting program: /home/lazarus_book/lazarus_examples/first_project/project1
[Thread debugging using libthread_db enabled]
```

此时可以看到程序正在正常运行。

6. 在文本框输入数字“5”以后，单击“计算”按钮。此时 gdb 会提示程序运行到断点处，并且暂停当前程序的执行，并给出如下提示。

```
Breakpoint 1, BTNCOMPUTECLICK (this=0x7ffff136b2b0, SENDER=0x7ffff136c890) at
unit1.pas:45
```

```
45         for N := N downto 1 do  
(gdb)
```

7. 执行命令“1”，可以查看当前正在执行的代码段，如下所示。

```
(gdb) 1  
40         N := StrToInt(Edit1.Text);  
41         if N < 1 then  
42             raise EConvertError.Create('');  
43  
44         Result := 0;  
45         for N := N downto 1 do  
46             Result := Result + N;  
47  
48         ShowMessage(IntToStr(Result));  
49     except
```

8. 要查看某个变量，可以用“p 变量名”来显示当前的值；如果要单步执行，那么可以使用“s”命令。如果不输入命令直接回车，那么会重复执行上一次输入的命令。如下。

```
(gdb) p N  
$1 = 5  
(gdb) p RESULT  
$2 = 0  
(gdb) s  
46         Result := Result + N;  
(gdb)  
45         for N := N downto 1 do  
(gdb)
```

```
46      Result := Result + N;
(gdb)
45      for N := N downto 1 do
(gdb)
46      Result := Result + N;
(gdb)
45      for N := N downto 1 do
(gdb)
46      Result := Result + N;
(gdb)
45      for N := N downto 1 do
(gdb) p N
$3 = 2
(gdb) p RESULT
$4 = 14
```

9. 执行命令“c”，继续当前程序的执行，如下所示。

```
(gdb) c
Continuing.
```

10. 把 project1 退出以后，gdb 会收到程序退出的信息，如下所示。

```
Program exited normally.
(gdb)
```

此时 gdb 并没有退出，可以选择使用命令“r”来重新执行程序，也可以使用命令“q”来退出 gdb。

11. 执行命令 q，退出 gdb。

1.2.3 脱离 Lazarus 来编译你的应用程序

尽管 Lazarus IDE 已经对整个编译过程进行管理，但是对于某些复杂的应用程序，仅仅依靠在 Lazarus IDE 里面点几下鼠标设置几个参数的方法是无法实现的。这个时候就需要脱离 Lazarus 来编译你的应用程序。

这一节仍旧继续 project1 的例子。在该章节的例子里面，假设每次编译前 project1 目录里没有上次编译输出的结果（例如最终的可执行文件，等等）。

1.2.3.1 使用 lazbuild 进行编译

lazbuild 是 Lazarus 自带的命令行界面的编译管理程序，通过它可以简单地编译 Lazarus 应用程序。一般情况下，只需要切换到项目所在的目录，执行命令“lazbuild 项目文件”即可。例如，编译前边的项目 project1。

```
$ ~/lazarus/lazbuild project1.lpi
NOTE: miscellaneous options file not found - using defaults
TFPCTargetConfigCache.NeedsUpdate compiler file changed "/usr/bin/fpc"
FileAge=1308289591 StoredAge=0
TFPCTargetConfigCache.Update /usr/bin/fpc TargetOS=linux TargetCPU=x86_64
CompilerOptions= ExtraOptions=
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
RunTool /usr/bin/fpc -iTOTP -Px86_64 -Tlinux
RunFPCVerbose unable to create test file
TFPCTargetConfigCache.Update WARNING: no unit paths: /usr/bin/fpc -Px86_64 -Tlinux
TFPCTargetConfigCache.Update: has changed
TCompiler.Compile
WorkingDir="/home/lazarus_book/lazarus_examples/first_project/"
CompilerFilename="/usr/bin/fpc" CompilerParams=" -MObjFPC -Scghi -O1 -gl -WG
-vewnhi -l -Filib/x86_64-linux -Fu../../lazarus/lcl/units/x86_64-linux
```

```
-Fu../../lazarus/lcl/units/x86_64-linux/gtk2
-Fu../../lazarus/packager/units/x86_64-linux -Fu. -FUIlib/x86_64-linux/ -oproject1
-dLCL -dLCLgtk2 project1.lpr"
[TCompiler.Compile] CmdLine="/usr/bin/fpc -MObjFPC -Scghi -O1 -gl -WG -vewnhi -l
-Filib/x86_64-linux -Fu../../lazarus/lcl/units/x86_64-linux
-Fu../../lazarus/lcl/units/x86_64-linux/gtk2
-Fu../../lazarus/packager/units/x86_64-linux -Fu. -FUIlib/x86_64-linux/ -oproject1
-dLCL -dLCLgtk2 project1.lpr"
Hint: Start of reading config file /etc/fpc.cfg
Hint: End of reading config file /etc/fpc.cfg
Free Pascal Compiler version 2.4.0-2ubuntu1.10.04 [2011/06/17] for x86_64
Copyright (c) 1993-2009 by Florian Klaempfl
Target OS: Linux for x86-64
Compiling project1.lpr
Compiling unit1.pas
unit1.pas(18,31) Hint: Parameter "Sender" not used
Compiling resource lib/x86_64-linux/project1.or
Linking project1
/usr/bin/ld: warning: link.res contains output sections; did you forget -T?
80 lines compiled, 0.9 sec
3 hint(s) issued
[TCompiler.Compile] end
```

这里有一点值得注意的是，工程文件是“.lpi”文件而不是“.lpr”文件。若要看lazbuild的使用方法，可以直接输入“lazbuild”而不带任何参数。例如：

```
$ ~/lazarus/lazbuild
```

```
lazbuild [options] <project or package-filename>
```

Parameters:

```
--help or -?           this help message
```

```
-B or --build-all      build all files of project/package/IDE
```

```
-r or --recursive       apply build flags (-B) to dependencies too
```

```
-d or --skip-dependencies do not compile dependencies
```

```
--build-ide=<options>   build IDE with packages
```

```
-v or --version         show version and exit
```

```
--primary-config-path=<path>
```

```
or --pcp=<path>
```

primary config directory, where Lazarus stores its
config files. Default is /home/lazarus_book/.lazarus

```
--secondary-config-path=<path>
```

```
or --scp=<path>
```

secondary config directory, where Lazarus searches
for config template files. Default is /etc/lazarus

```
--operating-system=<operating-system>
```

```
or --os=<operating-system>
```

override the project operating system. e.g. win32

```
linux. default: linux

--widgetset=<widgetset>
or --ws=<widgetset>

    override the project widgetset. e.g. gtk gtk2 qt
    win32 carbon. default: gtk2

--cpu=<cpu>

    override the project cpu. e.g. i386 x86_64 powerpc
    powerpc_64 etc. default: x86_64

--build-mode=<project build mode>
or --bm=<project build mode>

    override the project build mode.

--compiler=<ppcXXX>

    override the default compiler. e.g. ppc386 ppcx64
    ppcppc etc. default is stored in
    environmentoptions.xml

--language=

    Override language. For example --language=de. For
    possible values see files in the languages directory.
```

1.2.3.2 使用 fpc 进行编译

由于 Lazarus 程序实际上是 Free Pascal 程序，因此 Lazarus 应用程序也同时可以直接通过调用 fpc 进行编译。fpc 的编译十分智能，当对一个 program 编译的时候会同时将代码里面

指定的所有其它源代码文件一起编译。同时，若无法对整个程序使用一次性编译的方式，那么还可以单独编译每个单元文件，然后再链接成最终的应用程序。这里以前文提到的 project1 作为例子，讲解如何在命令行使用 fpc 进行编译。

对于一个简单的 Lazarus 应用程序，使用 fpc 编译时首先要了解的是，具体的编译参数。对于这一点，可以用下面的方法获得：

1. 单击菜单 “Project” -> “Project Options”，打开 “Options for Project: project1” 对话框。
2. 单击按钮 “Show Options”，打开 “Compiler Options” 对话框，如图1.11所示。

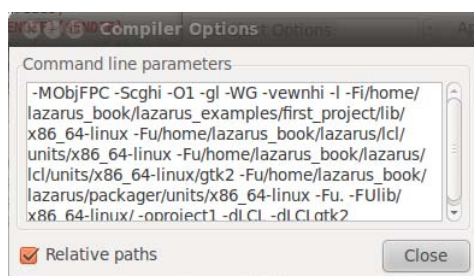


图1.11 fpc 编译的命令行参数（例子 project1）。

3. 该对话框里面就是以 fpc 单独编译的时候所需要带上的命令行参数。可以将这一段命令行参数给完整复制下来。

仔细看该命令行参数，其中 “/home/lazarus_book/lazarus_examples/first_project” 是 “project1” 所在的目录，而其中所提到的目录

“/home/lazarus_book/lazarus_examples/first_project/lib/x86_64-linux” 其实是不存在的，这时会导致编译出错的。其实，Lazarus 为了方便管理，特意在工程源代码目录里面建立单独的目录（如此处的 “lib/x86_64-linux”，这个目录会根据具体的系统平台而变化），然后将编译的中间结果等内容输出到该目录以使得不会和源代码混淆。所以在编译前，首先得确保该目录的存在（或者把参数 “-FUlib/x86_64-linux/” 和

“-Fi/home/lazarus_book/lazarus_examples/first_project/lib/x86_64-linux” 去掉），然后才可以用 “fpc 命令行参数 项目源代码” 的格式来编译。首先切换到 project1 所在的目录，接下来的编译过程如下所示。

```
$ mkdir -p lib/x86_64-linux
$ fpc -MObjFPC -Scghi -O1 -gl -WG -vewnhi -l
-Fi/home/lazarus_book/lazarus_examples/first_project/lib/x86_64-linux
-Fu/home/lazarus_book/lazarus/lcl/units/x86_64-linux
-Fu/home/lazarus_book/lazarus/lcl/units/x86_64-linux/gtk2
-Fu/home/lazarus_book/lazarus/packager/units/x86_64-linux -Fu.
-FUlib/x86_64-linux/ -oproject1 -dLCL -dLCLgtk2 project1.lpr
Hint: Start of reading config file /etc/fpc.cfg
Hint: End of reading config file /etc/fpc.cfg
Free Pascal Compiler version 2.4.0-2ubuntu1.10.04 [2011/06/17] for x86_64
Copyright (c) 1993-2009 by Florian Klaempfl
Target OS: Linux for x86-64
Compiling project1.lpr
Compiling unit1.pas
unit1.pas(18,31) Hint: Parameter "Sender" not used
Compiling resource lib/x86_64-linux/project1.or
Linking project1
/usr/bin/ld: warning: link.res contains output sections; did you forget -T?
80 lines compiled, 0.9 sec
3 hint(s) issued
```

1.3 LCL 的 GUI 后端

与 windows 不同，Linux 内核并不包含一套 GUI 界面库，因而在 Linux 下，没有一套统一的 GUI 库（Windows 则包含在 Windows API 里面了）。所以，在 Linux 下编写的 GUI 应用程序，总是依赖于一套特定的 GUI 库，并且这个库甚至要负责如绘制按钮这样的工作，这使得 Linux 下的 GUI 应用程序界面风格各异，甚至同样的控件，在操作上也有差别。

目前在 Linux 下，最流行的两套 GUI 库当属 GTK 和 Qt 了。GTK 作为 GNOME 桌面系统的基

础，只要安装有 GNOME 桌面系统的，必然安装有 GTK。Qt 作为 KDE 桌面系统的基础，只要安装有 KDE，那么必然安装有 Qt。由于目前流行的 Linux 发行版上都安装有 GNOME 或者 KDE，并且即使默认没有安装其中的一个，那么软件仓库里面必定有另外一个，就算只有其中一个，那么相应的 GTK, Qt 库也是有的。因而在 Linux 下开发 GUI 应用程序，GTK 或者 Qt 是不错的选择。GTK 和 Qt 库编写的程序，都可以同时运行。



图1.12 同时运行的两个计算器程序及其“关于”对话框。左边的是用 GTK 编写的，右边的是用 Qt 编写的。

那么如何使用 Lazarus 来开发 GTK 或者 Qt 应用程序呢？Lazarus 自带的 GUI 库叫 LCL。LCL 本身并不是一套完全独立的 GUI 库，它通过调用其它 GUI 库来实现其自身的功能。LCL 本身可以调用很多不同的 GUI 库，包括 GTK, Qt, Windows API 等等，这些库称为 LCL 的后端。LCL 的目的在于将各种不同的后端隐藏起来，提供一个统一的接口。这样只需要通过一套源代码，经过重新编译，就能切换到不同的后端。

默认情况下, Lazarus 的编译过程将生成基于 gtk2 的 LCL 库, 并且由于 Lazarus 本身也是基于 LCL 库写成的, 因而最终生成的 Lazarus 本身也是基于 gtk2 库的。在这种情况下编写 Lazarus 应用程序, 编译后同样也是一个 gtk2 应用程序。那么, 如何将一个 Lazarus 应用程序切换成其它后端呢? 首先要将 LCL 库重新编译并链接到其它后端, 然后再重新将应用程序重新编译并链接到这个新的 LCL 库。

1.3.1 切换到 Qt4 GUI 后端

这一节以实际例子讲解如何将 Lazarus 重新编译成 Qt4 应用程序, 然后再将 3.2 节的例子 project1 重新编译成一个 Qt4 应用程序。

1.3.1.1 重新编译前的准备

要编译成 Qt4 应用程序, 系统里面就必须安装有 Qt4 的开发包。这可以通过从新立得软件包管理器里面安装 “qt4-” 开头的所有软件包, 以及软件包 “libqt4-dev” 来获得。

仅仅只有这些还不够, 还需要一个叫 libQt4Pas 的库。这个库提供了 Free Pascal 对 Qt4 的支持。这个库可以从其官方网站 <http://users.telenet.be/Jan.Van.hijfte/qtforfpc/fpcqt4.html> 获得。现在从官方网站获取其源代码, 现在假设获得的源代码文件为 qt4pas-V2.5_Qt4.5.3.tar.gz (这个版本支持 Qt 4.5 到 Qt 4.8), 首先将其解压缩, 然后打开命令行, 在命令行里切换到解压后的目录, 通过以下步骤来安装这个库。

1. 执行命令 “qmake”, 生成 Makefile, 如下所示。

```
$ qmake
Project MESSAGE: Note: This binding version was generated for Qt 4.5.3. Current Qt
is 4.6.2
Project MESSAGE: Qt documents binary compatibility in each Version Change Note:
http://qt.nokia.com/developer/changes
Project MESSAGE: Pascal Qt Interface for binding platform: BINUX
```

```
Project MESSAGE: Install location: /usr/lib
```

2. 执行命令“make”，编译 libqt4pas 库。这个操作需要等待较长的一段时间。
3. 执行命令“sudo make install”，安装 libqt4pas 库，如下所示。

```
$ sudo make install  
install -m 755 -p "libQt4Pas.so.5.2.5" "/usr/lib/libQt4Pas.so.5.2.5"  
strip --strip-unneeded "/usr/lib/libQt4Pas.so.5.2.5"  
ln -f -s "libQt4Pas.so.5.2.5" "/usr/lib/libQt4Pas.so"  
ln -f -s "libQt4Pas.so.5.2.5" "/usr/lib/libQt4Pas.so.5"  
ln -f -s "libQt4Pas.so.5.2.5" "/usr/lib/libQt4Pas.so.5.2"
```

1.3.1.2 重新编译 Lazarus

有了 Qt4 的开发包，那么就可以重新将 Lazarus 编译成 Qt4 的应用程序。Lazarus 自带了重新编译自己的功能，启动 Lazarus 以后，通过下面的步骤来将 Lazarus 重新编译成 Qt4 应用程序。

1. 选择菜单“Tools”->“Configure “Build Lazarus”...”，打开“Configure “Build Lazarus”...”对话框。
2. 在左侧的列表里面，将所有项目都改成“Clean+Build”（Examples 一项可以选择“None”）。
3. 将“LCL Widget Type”改成“qt”，完成后如图3.13所示。

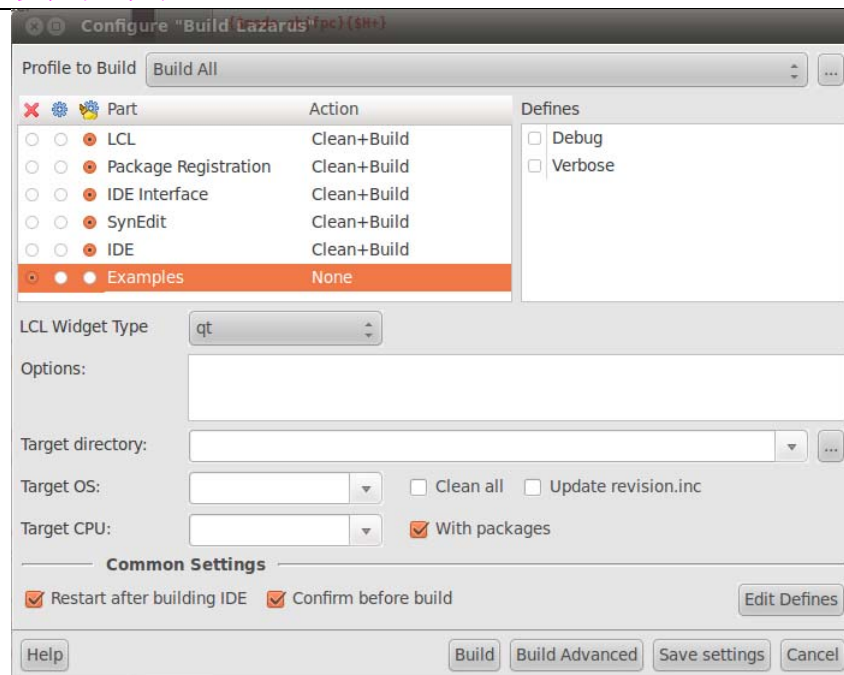


图1.13 选择如何重新编译 Lazarus。

4. 单击“Build”按钮，稍等片刻，Lazarus 在编译完成后会重新启动。

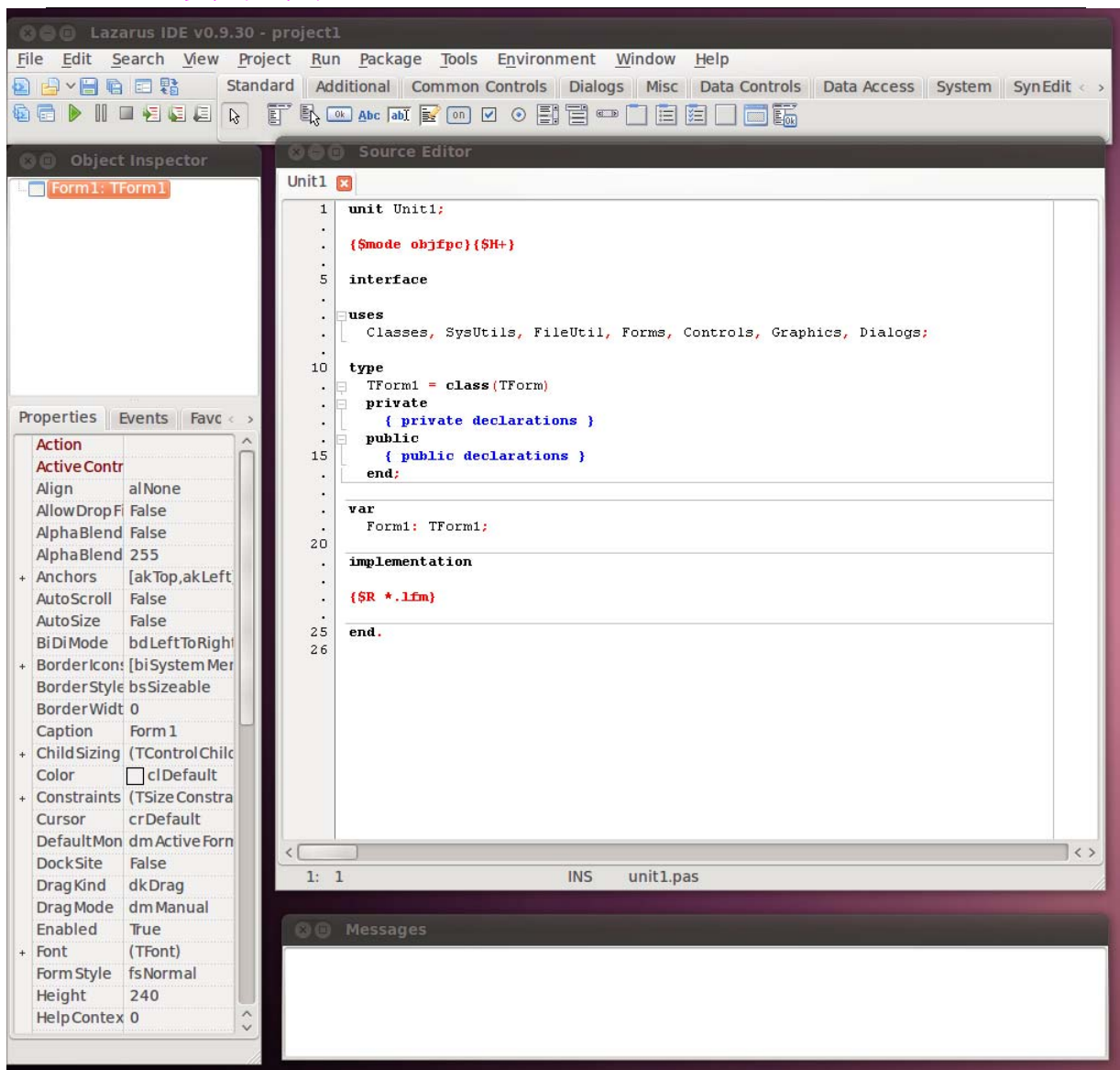


图1.14 作为 Qt4应用程序的 Lazarus。

- 如图1.14所示，此时的 Lazarus 已经是一个 Qt4应用程序了。注意，也许重新启动后的 Lazarus 跟原来的 Lazarus 差别看起来不是很大，甚至和原来几乎是一样的界面，那有可能是 Qt 跟 GNOME 一样主题的原因。要看是否是一个 Qt4应用程序，可以看重新编译后的 lazarus 文件是否已经链接到相应的 Qt 库，如下所示（其中“libQt”开头的库）。

```
$ ldd lazarus
linux-vdso.so.1 => (0x00007fff719ff000)
```

```
libdl.so.2 => /lib/libdl.so.2 (0x00007ffa27e2f000)

libQt4Pas.so.5 => /usr/lib/libQt4Pas.so.5 (0x00007ffa279f2000)

libX11.so.6 => /usr/lib/libX11.so.6 (0x00007ffa276bb000)

libc.so.6 => /lib/libc.so.6 (0x00007ffa27338000)

/lib64/ld-linux-x86-64.so.2 (0x00007ffa28063000)

libQtWebKit.so.4 => /usr/lib/libQtWebKit.so.4 (0x00007ffa25f6a000)

libQtGui.so.4 => /usr/lib/libQtGui.so.4 (0x00007ffa252d2000)

libQtNetwork.so.4 => /usr/lib/libQtNetwork.so.4 (0x00007ffa24fa4000)

libQtCore.so.4 => /usr/lib/libQtCore.so.4 (0x00007ffa24b21000)

libpthread.so.0 => /lib/libpthread.so.0 (0x00007ffa24903000)

libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x00007ffa245ef000)

libm.so.6 => /lib/libm.so.6 (0x00007ffa2436c000)

libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x00007ffa24154000)

libxcb.so.1 => /usr/lib/libxcb.so.1 (0x00007ffa23f38000)

libsqlite3.so.0 => /usr/lib/libsqlite3.so.0 (0x00007ffa23cab000)

libphonon.so.4 => /usr/lib/libphonon.so.4 (0x00007ffa23a51000)

libQtXmlPatterns.so.4 => /usr/lib/libQtXmlPatterns.so.4 (0x00007ffa233de000)

libXrender.so.1 => /usr/lib/libXrender.so.1 (0x00007ffa231d4000)

libfontconfig.so.1 => /usr/lib/libfontconfig.so.1 (0x00007ffa22f9e000)

libaudio.so.2 => /usr/lib/libaudio.so.2 (0x00007ffa22d85000)

libglib-2.0.so.0 => /lib/libglib-2.0.so.0 (0x00007ffa22aa7000)

libpng12.so.0 => /lib/libpng12.so.0 (0x00007ffa2287f000)

libz.so.1 => /lib/libz.so.1 (0x00007ffa22668000)

libfreetype.so.6 => /usr/lib/libfreetype.so.6 (0x00007ffa223e2000)

libgobject-2.0.so.0 => /usr/lib/libgobject-2.0.so.0 (0x00007ffa22199000)

libSM.so.6 => /usr/lib/libSM.so.6 (0x00007ffa21f90000)
```



```
libICE.so.6 => /usr/lib/libICE.so.6 (0x00007ffa21d75000)
libXext.so.6 => /usr/lib/libXext.so.6 (0x00007ffa21b62000)
libgthread-2.0.so.0 => /usr/lib/libgthread-2.0.so.0 (0x00007ffa2195d000)
librt.so.1 => /lib/librt.so.1 (0x00007ffa21755000)
libXau.so.6 => /usr/lib/libXau.so.6 (0x00007ffa21550000)
libXdmcp.so.6 => /usr/lib/libXdmcp.so.6 (0x00007ffa2134a000)
libQtDBus.so.4 => /usr/lib/libQtDBus.so.4 (0x00007ffa210d0000)
libexpat.so.1 => /lib/libexpat.so.1 (0x00007ffa20ea7000)
libXt.so.6 => /usr/lib/libXt.so.6 (0x00007ffa20c42000)
libpcre.so.3 => /lib/libpcre.so.3 (0x00007ffa20a13000)
libuuid.so.1 => /lib/libuuid.so.1 (0x00007ffa2080e000)
libQtXml.so.4 => /usr/lib/libQtXml.so.4 (0x00007ffa205c5000)
```

1.3.1.3 重新编译 project1

有了 Qt4 版本的 Lazarus，同样也就有了 Qt4 版本的 LCL 了。这时候重新打开 1.2 节的项目 project1，对其重新编译，将生成 Qt4 版本的 project1。Project1 运行后如图 1.15 所示。



图1.15 Qt4版本的 project1。

1.3.2 切换到 gtk2 后端

要把一个 Qt4 后端的 Lazarus 切换回 gtk2 后端，只需要采用 3.3.1.2 节所用的方法，只是在选择 “LCL Widget Type” 的时候，要改成 “gtk 2”。

值得一提的是，完全可以让 Lazarus 和应用程序所用的后端是不一样的。例如，可以用 gtk2 的 Lazarus 开发出 qt4 应用程序，等等。当个后端还不成熟的时候，这种做法十分有用。之前很长一段时间里面，Lazarus 官方就是推荐使用 gtk1 后端的 Lazarus 做开发，然后编译出 gtk2 后端的 LCL 应用程序。

那么如何让 Lazarus 编译出不同后端的应用程序呢？现在假设 Lazarus 的后端是 gtk2，要想编译出一个 qt4 的应用程序，那么可以使用如下步骤（以 1.2 节的例子 project1 为例）：

1. 确保已经使用过 3.3.1.2 介绍的方法将 LCL（仅仅是 LCL 即可）编译成 qt 后端的。
2. 选择菜单 “Project” -> “Project Options ...”，打开 “Options for Project: project1” 对话框。
3. 在左侧的树形选择框中选择 “Compiler Options” -> “Paths”。
4. 在右侧的 “LCL Widget Type” 选项中，选择 “qt”。
5. 单击 “OK”，关闭 “Options for Project: project1” 对话框。
6. 完全重新编译 project1。完成后 project1 即为一个 Qt4 应用程序。

1.3.3 直接调用 GUI 后端

既然 LCL 的功能是通过对其后端的调用来实现的，那么也就意味着可以直接调用 LCL 的后端而不需要通过 LCL，特别当实现某些仅仅使用 LCL 无法实现的功能的时候，这也许就是一个有效的解决方案。下面通过两个例子，介绍如何在 Lazarus 里面直接编写 gtk2 和 qt4 应用程序而不通过 LCL。

1.3.3.1 实例：一个简单的 Qt4 应用程序

由于 LCL 是通过调用 libQt4Pas 库来实现对 Qt4 的调用的，所以同样的也可以通过直接调用 libQt4Pas 库的方式，来调用 Qt4。那么，为什么不直接调用 Qt4 呢？因为 Qt 本身是一个纯 C++ 编写的库，所有的接口都是 C++ 的，如果直接调用，那么是十分复杂的一件事情。而 libQt4Pas 的作用是将 Qt4 转换成一个 C 语言的动态库，以方便调用。需要注意的一点是，libQt4Pas 的设计目标是为 LCL 提供 Qt4 支持，并不能完整的利用 Qt4 的所有功能。

在开始下面的例子之前，请首先确保已经将 LCL 编译成 qt 后端，具体方法请参见 1.3.1.2 节。libQt4Pas 的一部分代码已经被放在 LCL 的 qt 后端所在的目录里面，如果将 LCL 编译成 qt 后端，那么 libQt4Pas 的接口单元也将被编译，以方便以后链接。其实如果不这么做的话，直接指定 libQt4Pas 接口单元所在的位置也是可以的。libQt4Pas 的接口单元称为 “qt4”。

下面的实例将创建一个简单的 Qt4 应用程序，这个应用程序将显示一个窗口，窗口上面有一个按钮和一个文本框，按下按钮以后，显示文本框里面的内容。

1. 单击菜单 “Project” -> “New Project...”，打开 “Create a new project” 对话框。
2. 在 “Create a new project” 对话框里面，选择 “Program”，然后单击 “OK”，创建新的工程。
3. 将工程保存，工程名字 “qt4demo”。
4. 单击菜单 “Project” -> “Project Options ...”，打开 “Options for Project: qt4demo” 对话框。
5. 在左侧的树形列表框里面，选中 “Compiler Options” -> “Paths”。
6. 在右侧的 “Other Unit Files (-Fu) (Delimiter is semicolon)” 里面，输入路径 “\$(LazarusDir)/lcl/units/\$(TargetCPU)-\$(TargetOS)/\$(LCLWidgetType);\$(LazarusDir)/lcl/units/\$(TargetCPU)-\$(TargetOS)”。当然，也可以单击该文本框右侧的 “...” 来选中该路径。有一点需要说明的是，这里将第二个路径引入，是为了使用 LCLProc 单元的编码转换功能，这个单元是 LCL 库的组成部分之一。尽管 Free Pascal 的 SysUtils 单元也同样能做到，但是 LCLProc 在这方面实现的更好。
7. 在右侧的 “LCL Widget Type (various)” 下拉列表框里面，选中 “qt”。这一步的作用是，让上一步的 \$(LCLWidgetType) 拥有正确的值。
8. 完成后如图 1.16 所示。

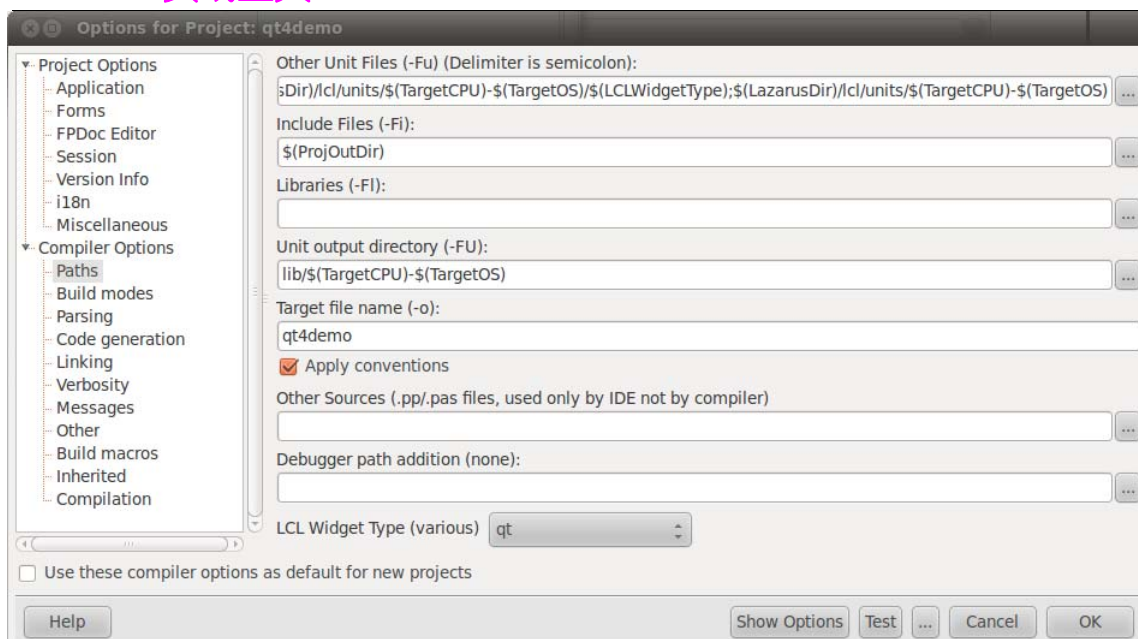


图1.16 设置 Qt4的单元路径。

9. 单击“OK”，关闭“Options for Project: qt4demo”对话框。
10. 以下是整个 qt4demo 的源代码，将其输入到代码编辑器里面。

```
program qt4demo;

{$mode objfpc} {$H+}

uses

  {$IFDEF UNIX} {$IFDEF UseCThreads}
  cthreads,
  {$ENDIF} {$ENDIF}

  Classes,
  qt4,
  LCLProc;
```

```
{ $R *.res }

type
  TMyQtButton = class
  public
    procedure clicked(checke d: Boolean); cdecl;
  end;

var
  appHandle : QApplicationH; // equals to QApplication
  textEditHandle : QTextEditH;
  buttonHandle : QPushButtonH;
  buttonHookHandle : QPushButton_hookH;
  widgetHandle : QWidgetH;
  myQtButtonWrapper : TMyQtButton;
  layoutHandle : QVBoxLayoutH;
  buttonText : WideString;
  contentText : WideString;

procedure TMyQtButton.clicked(checke d: Boolean = False); cdecl;
var
  content: WideString;
  windowTitle: WideString;
begin
  SetLength(content, 65536);
  SetLength(windowTitle, 255);
```

```
QTextEdit_toPlainText(textEditHandle, @content);

QWidget_windowTitle(widgetHandle, @windowTitle);

QMessageBox_information(
    widgetHandle,
    @windowTitle,
    @content
);
end;

begin
    appHandle := nil;
    myQtButtonWrapper := TMyQtButton.Create;

    try
        // create an QApplication object with standard qt arguments
        appHandle := QApplication_create(@argc, argv);
        QApplication_setQuitOnLastWindowClosed(true);

        // create controls
        contentText := UTF8ToUTF16('这是一个用 Free Pascal 编写的 Qt 应用程序!');
        textEditHandle := QTextEdit_create(@contentText);

        buttonText := Utf8ToUtf16('显示上面输入的内容(&S)');
        buttonHandle := QPushButton_create(@buttonText);
```

```
// connect signal and slot ( signal/slot are the mechanism that like the events
in LCL )

buttonHookHandle := QPushButton_hook_create(buttonHandle);

QAbstractButton_hook_hook_clicked(buttonHookHandle,
@(myQtButtonWrapper.clicked));

// create a layout object and add the controls
layoutHandle := QVBoxLayout_create;
QBoxLayout_addWidget(layoutHandle, textEditHandle);
QBoxLayout_addWidget(layoutHandle, buttonHandle);

// create a window
widgetHandle := QWidget_create;
QWidget_setLayout(widgetHandle, layoutHandle);

// show the window
QWidget_show(widgetHandle);

// process messages
QApplication_exec;

finally
    QApplication_destroy(appHandle);
    myQtButtonWrapper.Free;
end;
end.
```

11. 编译运行 qt4demo，效果如图1.17所示。



下面是关于 qt4demo 这个例子的一些说明。

1. 程序的源代码使用 UTF-8 存储。在 Linux 下，UTF-8 是一个实际的标准。
2. 程序里面使用到 LCL 库的仅仅在于有两个地方调用了 Utf8ToUtf16 函数。尽管 WideString 本身就是一个 Utf16 的字符串，但是这里的字符串常量本身就是一个 AnsiString（根据 \$H 编译指令的不同字符串常量还有可能是 ShortString），直接将一个 AnsiString 赋值给 WideString 类型的变量并不能将其转换成 Utf16 字符串，这中间需要一个转换过程，而这个转换过程，Free Pascal 的 SysUtils 单元已经提供相应的功能。然而，LCLProc 对此提供了更好的支持，特别是当遇到 UTF-8 字符串的时候（由于源代码是 UTF-8 的，所以其中的字符串常量同样也是 UTF-8 的）。当然也可以不使用 LCLProc 单元，而直接使用 Qt4 提供的转换函数 `QString::fromUtf8`，如下的函数演示了如何将一个 UTF8 字符串转换成 UTF16。

```
function ConvertString(Origin:string):WideString;  
begin  
    SetLength(Result, 1024);  
    QString_fromUtf8(@Result, PChar(Origin));  
end;
```

那么，为什么一定要使用 UTF16 的 WideString 呢？这是 libQt4Pas 的要求。而 libQt4Pas 之所以这么要求，也是由于 Qt4 本身的字符串处理方式导致的。Qt4 的字符串通过类 Qt 类 `QString` 来实现。`QString` 是一个 Unicode 4.0 的字符串，其内部就是使用 UTF16 进行存储的。

说到这里，也许有的读者会问：UTF16和Unicode又有什么区别呢？Unicode是字符编码，每个Unicode编码是一个数字，对应一个字符；而UTF16是Unicode编码的具体存储形式，由于一个Unicode字符往往需要多个字节才能表示，所以实际使用时，有UTF8、UTF16、UTF32等具体的存储方式，其中的UTF16和UTF32则根据字节序（Byte Order）的不同而又可以细分成多种不同的存储方式，而Unicode的各种存储方式之间并不互相兼容。

3. Qt的事件响应机制采用“信号”（Signal）/“槽”（Slot）的机制。一个信号相当于一个事件，而一个槽相当于一个事件响应函数。在Qt中，一个槽要响应一个信号，必须将其和对应的信号进行连接，这需要通过方法 `QObject::connect()` 来实现。而在qt4单元里面，这种机制则通过一个中间类来实现，这个中间类的名称“Qt类名_hookH”的形式，例如代码里的“`QPushButton_hookH`”就用于响应 `QPushButton` 发出的信号。但是仅仅创建一个中间类还不够，还需要再使用相应的方法将事件处理函数给和信号给连接起来，如例子里的

```
QAbstractButton_hook_hook_clicked(buttonHookHandle, @(myQtButtonWrapper.clicked))
```

就是用户将按钮 `buttonHandle` 的 `clicked()` 信号和事件处理程序

`myQtButtonWrapper.clicked` 连接起来。注意例子中的类 `TMyQtButton`，这个类的存在仅仅只是为了 `QAbstractButton_hook_hook_clicked()` 第二个参数形式上的需要。

4. qt4单元对Qt4提供的接口采用的命名方式大致如下：

- 每个Qt类对应一个名为“类名H”的Free Pascal类。例如，`QApplication` 为“`QApplicationH`”。这个类实际上没有内容，只起到一个标记的作用。其中的“H”代表“Handle”，一个名为“类名H”的类的对象，可以理解为是代表该Qt对象的句柄。
- 对于构造函数，使用“类名_create”的方式。例如，`QApplication::QApplication` 为“`QApplication_create`”。
- 对于一般方法，则使用“类名_方法”的形式。例如，`QWidget::show`，则成为“`QWidget_show`”。若该Qt方法不是静态方法，则第一个参数是该Qt对象的句柄。

5. 这个例子显示了通常一个Qt应用程序的结构。首先是创建一个 `QApplication` 对象（也就

是例子中的 `QApplication_create`)，然后创建和显示各种窗体、控件，最后显示图形界面并进入 Qt 主循环(通过执行 `QApplication::exec()` 来实现，也就是例子中的 `QApplication_exec`。这个主循环相当于 Windows 编程中经典的“消息循环”)。

6. 例子在调用 `QApplication_create` 的时候将程序的命令行作为参数传入，这样可以使得该程序拥有 Qt 本身支持的应用程序命令行选项。例如，在命令行运行 “`./qt4demo -title aaa`”，那么 `qt4demo` 的程序标题将变成 “aaa”，如图1.18所示。



图3.18 在命令行运行 “`./qt4demo -title aaa`”，`qt4demo` 的标题栏被改成了 “aaa”。

7. Qt4支持多种布局管理器，使用布局管理器而不使用绝对位置布局能够在没有窗体编辑器的情况下也能做出理想的界面，并且使用布局管理器以后，每当大小改变时，也能自动重新布局，以保持界面布局不变。例子中使用的 `QVBoxLayout` 就是其中的一种。

8. 当某些东西是在父类实现的时候，`qt4`单元就不会再在子类里面重复了。例如，例子中使用 “`QAbstractButton_hook_hook_clicked`” 而不是 “`QPushButton_hook_hook_clicked`”，因为根本不存在 “`QPushButton_hook_hook_clicked`”。而 “`QBoxLayout_addWidget`” 也是同理。

关于 Qt 更加详细的信息，可以从其官方网站 <http://qt.nokia.com/> 获得。

1.3.3.2 实例：一个简单的 gtk2 应用程序

相比 Qt4需要 `libQt4Pas` 的支持，`gtk2`的接口 Free Pascal 已经自带了。所以可以直接使用 Free Pascal 编写 `gtk2`应用程序，需要的仅仅只是系统安装有 `gtk2`。用 Free Pascal 编写 `gtk2`应用程序的时候，基本的，要引用这三个单元：`glib2`，`gdk2`，`gtk2`。由于 `gtk` 是纯 C 编写的库，仅提供 C 语言接口，所以 Free Pascal 程序可以直接对其进行调用，因此在 Free Pascal 中使用 `gtk` 库跟在 C 语言中使用 `gtk` 库的方法是类似的。

下面的例子 gtk2demo 是一个使用 Lazarus 编写的简单的 gtk2 应用程序。它的功能和上一节的 qt4demo 相同，都是显示一个窗口，窗口上面有一个按钮和一个文本框，按下按钮以后，显示文本框里面的内容。不同的地方在于，qt4demo 的文本框是多行文本框，而 gtk2demo 使用的是单行文本框。

```
program gtk2demo;

{$mode objfpc} {$H+}

uses

  {$IFDEF UNIX} {$IFDEF UseCThreads}
  cthreads,
  {$ENDIF} {$ENDIF}

  Classes,
  glib2,
  gdk2,
  gtk2;

{$R *.res}

var
  window : PGtkWindow;
  button : PGtkButton;
  textEdit : PGtkEntry;
  layout : PGtkVBox;

procedure buttonClicked(widget: PGtkWidget; data:gpointer);cdecl;
```

```
var
    msgbox : PGtkWidget;
begin
    msgbox := gtk_message_dialog_new(
        window,
        GTK_DIALOG_DESTROY_WITH_PARENT,
        GTK_MESSAGE_INFO,
        GTK_BUTTONS_OK,
        gtk_entry_get_text( textEdit )
    );

    gtk_dialog_run( PGtkDialog(msgbox) );
    gtk_widget_destroy( msgbox );
end;

begin

    gtk_init(@argc, @argv);

    // create a single line text editor
    textEdit := PGtkEntry(gtk_entry_new);
    gtk_entry_set_text(textEdit, '这是一个用 Free Pascal 编写的 Gtk 应用程序!');
    gtk_entry_set_width_chars(textEdit, 40);

    // create a button
    button := PGtkButton(gtk_button_new_with_label('显示上面输入的内容(_S)'));
    gtk_button_set_use_underline(button, true);
```

```
// create VBox layout with spacing 5 and put the controls
layout := PGtkVBox(gtk_vbox_new(false, 5));
gtk_container_add( PGtkContainer(layout), PGtkWidget(textEdit) );
gtk_container_add( PGtkContainer(layout), PGtkWidget(button) );

// create a new window and put the controls
window := PGtkWindow(gtk_window_new(GTK_WINDOW_TOPLEVEL));
gtk_container_add( PGtkContainer(window), PGtkWidget(layout) );

// connect the signals and slots
g_signal_connect(window, 'destroy', G_CALLBACK(@gtk_main_quit), nil);
g_signal_connect( button, 'clicked', G_CALLBACK(@buttonClicked), nil );

// make the widgets visible
gtk_widget_show_all(PGtkWidget(window));

// enter the gtk main loop

gtk_main;

end.
```

由于 gtk2 支持是 Free Pascal 自带的，因而编译也变得十分简单，只需要为“fpc”指定该文件即可（当然，直接在 Lazarus 里面编译也是没问题的）。程序运行后效果如图 1.19 所示。



图1.19 例子 gtk2demo 运行结果。

gtk 在很多设计思想上同 Qt 类似，比如：它们都是面向对象的，每个控件都对应着一个类，并且都拥有自己的继承树；它们都使用信号/槽机制，等等。不过这个例子还是有很多值得说明一下的。

1. 由于是 C 语言的库，因而大部分接口的命名仍旧使用 C 语言的名称。不过在命名上，还有一些值得注意的：

- 一些名称会带上 Pascal 风格的命名方法，比如：C 结构体 GtkWidget 被翻译成 Pascal 的记录 TGtkWidget，而 C 指针 GtkWidget * 被翻译成 Pascal 类型 PGtkWidget。
- 一些和 Pascal 关键字一样的名称，会被避开。比如，C 里面 GtkWidget 的 label 字段，被翻译成 Pascal 里 TGtkWidget 的 _label 字段。
- 对于一些 Pascal 里面操作起来比较困难的操作，会提供一些辅助函数，如 width_set，等等。

2. Free Pascal 的 gtk 接口，并没有拥有 C 语言接口的所有功能。例如，gtk 里面的一些宏在翻译成 Pascal 的时候被翻译成了函数，使得使用 C 宏接口所拥有的一些功能在这里便无法以相同的接口进行使用。

3. 同 Qt 类似，程序首先使用 gtk_init 进行初始化，然后创建控件，然后调用 gtk_main 进入 gtk 主循环。

4. gtk 控件的命名以 “Gtk” 开头，如按钮控件 GtkButton，单行文本控件 GtkEntry，等等。

5. gtk 控件的操作函数往往以 “gtk_控件名_方法” 的形式命名，如 gtk2demo 里的 `gtk_widget_show_all`，等等。其中，构造函数以 “gtk_控件名_new” 的形式命名，如 gtk2demo 里 “`gtk_window_new`”，对属性的读写则采用 “gtk_控件名_get_属性” 和 “gtk_控件名_set_属性”，如 gtk2demo 里的 `gtk_entry_get_text` 和 `gtk_entry_set_text`。
6. 如果要使用键盘加速键，那么要写 “_字母” 的形式并且设置相应的属性才行，如 gtk2demo 里面的这两行：

```
button := PGtkButton(gtk_button_new_with_label('显示上面输入的内容(_S)'));  
gtk_button_set_use_underline(button, true);
```

7. 尽管 gtk 使用的是面向对象的设计，但由于语言本身并没有面向对象的支持，所以在使用的过程中往往会遇到大量的类型转换。例如，一个 gtk 对象，要使用父类的方法，则必须强制转换成一个父类对象。
8. gtk2demo 采用了和 qt4demo 类似的布局控件 `GtkVBox`。
9. `g_signal_connect` 用于连接信号和槽。在连接信号和槽的时候，gtk2demo 将 window 的 `destroy` 信号连接到了槽 `gtk_main_quit`。当窗体 window 被关闭的时候，会发射 `destroy` 信号，此时该信号连接的槽 `gtk_main_quit` 将被执行。`gtk_main_quit` 是 gtk 自带的函数，作用是退出整个程序。
10. 同 Qt 一样，`gtk_init()` 在执行的时候会处理 gtk 专用参数，这些参数是传递给 gtk 的。例如，如果执行 “`./gtk2demo --name aaa`”，那么程序的标题就变成了 “aaa”，如图1.20 所示。



图1.20 执行 “`./gtk2demo --name aaa`” 的效果。

更多信息，请参考 GTK 官方网站 <http://www.gtk.org/>。

1.4 使用 Linux 系统调用

相对于 Windows API，Linux 内核提供给应用程序的接口称之为“系统调用”。要在 Free Pascal 里面进行系统调用，可以通过 syscall 单元来实现。不过，syscall 单元提供的接口十分原始。毕竟，无论哪个系统调用，都跟普通的函数都用不一样。要进行 Linux 系统调用，首先要往寄存器里面存放调用的参数等信息，再通过\$80中断进行系统调用，最后再从寄存器里面取回返回值。

下面的例子 linux_syscalls_demo 将打开一个文件 a.out，然后往文件里面打印当前的系统时间，最后将该文件关闭并退出。该程序是一个控制台应用程序。整个过程通过 syscall 单元来实现。

```
program linux_syscalls_demo;

{$mode objfpc} {$H+}

uses
  syscall, BaseUnix, sysutils;

var
  fd : TSysResult;
  text : string;

begin
  // open file with syscal `open'
  fd := Do_SysCall( syscall_nr_open, TSysParam(PChar('a.out')), 0_WRONLY or 0_CREAT
or 0_TRUNC, &644 );
  if fd < 0 then
  begin
```



```
WriteLn(StdErr, 'Cannot open file `a.out`' );  
  
Exit;  
  
end;  
  
try  
  
    // write current time to file  
  
    text := DateTimeToStr(Now) + LineEnding;  
  
    if Do_SysCall( syscall_nr_write, fd, TSysParam(PChar(text)), Length(text) ) <  
0  
    then begin  
  
        WriteLn(StdErr, 'Error when writing to file `a.out`' );  
  
        Exit;  
  
    end;  
  
finally  
  
    // Close file  
  
    if Do_SysCall( syscall_nr_close, fd ) < 0 then  
  
        WriteLn(StdErr, 'Error when closing file `a.out`' );  
  
    end;  
  
end.
```

在 lazarus 输入以上程序，编译后打开终端窗口，运行该程序。如下所示。

```
$ ls a.out  
ls: 无法访问 a.out: 没有那个文件或目录  
$ ./linux_syscalls_demo  
$ cat a.out
```

```
15-1-12 15:56:27
$ ./linux_syscalls_demo
$ cat a.out
15-1-12 15:56:33
```

在如上的运行过程里，一开始 a.out 是不存在的。第一次运行 linux_syscalls_demo 的时候，通过系统调用“open”创建（指定了参数 O_CREAT）文件 a.out，紧接着，通过系统调用“write”将当前时间写入到文件。第二次运行 linux_syscalls_demo 的时候，在使用系统调用 open 打开文件 a.out 的时候，a.out 已经存在，但由于指定了参数 O_TRUNC，a.out 的内容被清空。a.out 内容被清空以后，重新像该文件写入新的系统当前时间。

Do_SysCall() 的第一个参数是系统调用编号，对应前缀为“syscall_nr_”的一些常量。剩余的参数，则为传递给该系统调用的参数。系统调用的返回值，由 Do_SysCall() 以返回值的形式返回给调用者。

对于大部分 Linux 系统调用来说，系统调用执行过程中是否出错，可以通过函数返回值来知晓。例如，linux_syscalls_demo 里面有几处对 Do_SysCall() 返回值的判断。如果出错，那么 linux_syscalls_demo 将会给出错误信息。下面的例子，首先让用户无法对 a.out 写入，然后再执行 linux_syscalls_demo，此时由于权限问题而无法打开文件，因此导致出错。

```
$ ls a.out -lh
-rw-r--r-- 1 root lazarus_book 17 2012-01-15 16:12 a.out
$ ./linux_syscalls_demo
Cannot open file `a.out'
```

如读者所见，直接使用 Do_SysCall() 进行 Linux 系统调用是十分繁琐的事情。因此，Free Pascal 对一些 Linux 系统调用做了高级的封装。要使用这些封装，可以引用如 Linux, BaseUnix, UnixType, syscall, Unix, unixutil 等单元。

1.5 编写和使用库

Linux 下程序库分成两种。一种是动态库，也称为“Shared Object”，扩展名为“.so”，相当于 Windows 的 DLL。另外一种静态库，扩展名为“.a”，相当于 Visual C++的.lib 文件。本章通过一个个的实例，分别介绍这两种库的开发和使用。

1.5.1 实例：静态库 libmydemo.a

Free Pascal 在编译每个文件的时候，首先将单个文件编译成.o 文件。例如，unit1.pas 编译后就成了 unit1.o。最后，将所有的“.o”文件链接起来，生成一个最终的可执行文件。一个静态库实际上就是用 ar 将若干个“.o”文件打包，以供程序链接时用。有时候为了加快压缩包内容的查找速度，会对压缩包里的内容创建索引。

下面的以创建一个简单的静态库 libmydemo.a 为例，介绍静态库的创建过程。

1. 准备单元文件 mydemo_a.pas，内容如下所示。

```
unit mydemo_a;

{$mode objfpc}

interface

function getX:Integer;
function minus(a,b : Integer):Integer;

implementation

uses
mydemo_b;

function getX:Integer;alias:'getX' ;
```

```
begin
    Result := my_number_X;
end; { getX }

function minus(a,b : Integer):Integer;alias:'minus';
begin
    Result := a - b;
end; { plus }

end.
```

2. 准备单元文件 mydemo_b.pas，内容如下所示。

```
unit mydemo_b;

{$mode objfpc}

interface

var my_number_X : Integer;cvar;

implementation

end.
```

3. 确保 mydemo_a.pas 和 mydemo_b.pas 放在同一个目录（假设这个目录叫 libmydemo_a）。
4. 打开终端，切换到目录 libmydemo_a。
5. 分别编译 mydemo_a.pas 和 mydemo_b.pas。由于 fpc 一次只能编译一个文件，因而必须分开编译。如下所示。

```
$ fpc -XS mydemo_b.pas

Free Pascal Compiler version 2.4.0-2ubuntu1.10.04 [2011/06/17] for x86_64
Copyright (c) 1993-2009 by Florian Klaempfl
Target OS: Linux for x86-64
Compiling mydemo_b.pas
9 lines compiled, 0.0 sec

$ fpc -XS mydemo_a.pas

Free Pascal Compiler version 2.4.0-2ubuntu1.10.04 [2011/06/17] for x86_64
Copyright (c) 1993-2009 by Florian Klaempfl
Target OS: Linux for x86-64
Compiling mydemo_a.pas
25 lines compiled, 0.0 sec
```

由于编译的是静态库，因而在这里带上参数“-XS”。

6. 使用 ar 将.o 文件打包。ar 常见的用法为“ar rcs 库文件名 .o 文件...”。尽管 ar 还有其它功能，但是很少被使用。执行如下命令来将.o 文件打包成静态库。

```
$ ar rcs libmydemo.a mydemo_a.o mydemo_b.o
```

7. 最终生成的文件 libmydemo.a 就是静态库了。静态库的标准路径，同动态库一样，32位 Linux 下一般是/usr/lib, 或者/lib; 64位 Linux 下一般是/lib64或者/usr/lib64。不过, Ubuntu 10.04 amd64已经将/lib 和 lib64等同，并且将/usr/lib 和/usr/lib64等同。将 libmydemo.a 放在/usr/lib, 或者/lib, 在下次引用的 libmydemo.a 时候，不需要再指定 libmydemo.a 的路径。

由于静态库本质上只是一堆“.o”文件的压缩包，因而，创建动态库的时候，只需要将每个单元编译后的.o 文件，以及引用到的其它.o 文件，一并用 ar 打包即可。将程序链接到一个静态库的时候，链接器会根据实际情况，将程序和静态库里面的“.o”文件进行链接。这种链接方式，跟直接在命令行指定链接到一堆“.o”文件并无区别。

对于一个“.o”文件来说，只有对外公开的名字（专业点说，那就是“符号”），才能被其它“.o”文件引用。否则，在链接的时候，就会出现找不到引用的错误。为了使一个名字能被其它“.o”文件引用，在代码里面，必须使对应的声明出现在 interface 节中。

在编译的时候，Free Pascal 根据自己的规则，对代码中出现的名字进行重命名。不管是对函数、过程，还是变量，均是如此。例如，libmydemo.a 里面的 mydemo_a.getX，编译后的名字成了“MYDEMO_A_GETX\$\$LONGINT”。如果别的程序要引用 mydemo_a.getX，那么必须指明是“MYDEMO_A_GETX\$\$LONGINT”才行，这显然是大家不愿意看到的。为了使别的程序更容易的引用 libmydemo.a，必须对其中的函数、过程，以及变量，进行重命名。对于函数和过程来说，需要在声明的末尾使用“alias:’别名’;”这样子的语法。例如 mydemo_a.pas 里的：

```
function getX:Integer;alias:’getX’;
```

这样声明以后，别的程序就可以使用“getX”这个名字来引用 mydemo_a.getX 了。那么，对于变量来说，又该怎么实现呢？在一个变量的声明后面加上“cvar;”以后，这个变量在“.o”文件就会以它的原名出现了。例如，mydemo_b.pas 里面的：

```
var my_number_X : Integer;cvar;
```

这样声明以后，“my_number_X”这个名字就会出现在最终的“.o”文件里面了。在使用这种写法的时候，每次只能写一个变量。例如，不可以写“x,y:integer;cvar;”。

在这个例子里面，libmydemo.a 公开了3个符号：“getX”、“minus”，以及“my_number_X”。这三个符号将可以被其它程序所引用。细心的读者可能会发现，为什么代码里的函数都没有指定调用约定？在 x86_64架构里面，目前只有两种调用约定。一种来自微软，为 Windows 专用；其它的，如 Linux 等操作系统，用的是 System V ABI AMD64规范里描述的调用约定。笔者的 Ubuntu 是 x86_64架构的，只有一种调用约定。在该架构下，对于 Free Pascal 来说，不管指定哪种调用约定（如 cdecl, register 等等），最终编译出来的都是同样的代码。当然，如果是32位系统，则默认的调用约定为 register。但是，32位 Linux 下通用的调用约定为 cdecl。如果要为某个函数或过程指定 cdecl 调用约定，只需要在其声明的时候带上“cdecl;”即可。例如，给“getX”指定 cdecl 调用约定：

```
function getX:Integer;cdecl;alias:'getX';
```

1.5.2 实例：使用 Free Pascal 调用 libmydemo.a

这一节讲解如何在一个 Free Pascal 应用程序里引用一个静态库。继续上一节的例子，这一节的例子将创建一个 Free Pascal 应用程序，它引用了 libmydemo.a。打开终端窗口以后，使用以下步骤。

1. 在 libmydemo.a 的目录下创建子目录 “pascal_invoke”。
2. 切换到目录 “pascal_invoke”。
3. 输入源文件 “invoke.pas”，代码如下。

```
program invoke;

{$linklib mydemo}
{$mode objfpc}

var
    my_number_X : Integer;cvar;external;

function getX:Integer;external name 'getX';
function minus(a,b : Integer):Integer;external name 'minus';

begin
    Write('Enter X: ');
    Readln(my_number_X);
    Writeln('X = ', getX);
    Writeln('X - 1 = ', minus(my_number_X, 1));
```

```
end.
```

4. 编译 invoke.pas，如下所示。

```
$ fpc -Xt -Fl.. invoke.pas

Free Pascal Compiler version 2.4.0-2ubuntu1.10.04 [2011/06/17] for x86_64 撸啊撸
Copyright (c) 1993-2009 by Florian Klaempfl
Target OS: Linux for x86-64
Compiling invoke.pas
Linking invoke
/usr/bin/ld: warning: link.res contains output sections; did you forget -T?
16 lines compiled, 0.3 sec
```

由于 libmydemo.a 并不在标准的库目录，所以在编译时必须带上参数“-Fl”来指定 libmydemo.a 的路径。同时还要加上命令行参数“-Xt”，告诉 fpc 在链接的时候要和静态库链接。

5. 运行编译后的 invoke，如下所示。

```
$ ./invoke
Enter X: 2147483647
X = 2147483647
X - 1 = 2147483646
```

由于 libmydemo.a 的代码已经被包含在 invoke 里面，因此在运行的时候，不需要再引用 libmydemo.a 了。

看完例子，该说说源代码了。要链接到一个静态库，首先要在代码里面使用编译指令“{\$linklib 静态库名称}”。注意例子里写的是“{\$linklib mydemo}”，这里为什么不写 libmydemo.a 呢？一般情况下，Linux 下的静态库的命名方式为“lib 库名.a”，所以，这里只需要写“mydemo”就行了。在 fpc 调用链接器的时候，会传递参数“-lmydemo”，这样链接器就会自动寻找“libmydemo.a”。

那么，又如何引用静态库的内容呢？如果要引用一个变量，那么就要写在变量声明后面写上“`cvar;external;`”。例如，`invoke.pas` 里的

```
my_number_X : Integer;cvar;external;
```

这样写了以后，在链接器就会自动的把变量“`my_number_X`”链接到 `libmydemo.a` 里的那个变量“`my_number_X`”了。如果要引用一个函数或者过程，那么要在相应的声明后面加上“`external name '名称';`”。这里的“名称”是指静态库里相应的函数或过程的名称。例如，`invoke.pas` 里的：

```
function getX:Integer;external name 'getX';  
function minus(a,b : Integer):Integer;external name 'minus';
```

值得注意的是，“`name 'XXX'`”这部分其实是可选的。不过，除非编译后的符号名和静态库里的符号名一致，否则还是得用“`name`”来指定静态库里的相应名称。另外，如果是在32位程序，还需要指定相应的函数调用约定，如 `cdecl`、`stdcall` 等。而且 `cdecl` 除了说明调用约定之外，还影响编译后的名称。比如这里在“`getX`”后面写上“`cdecl`”，那么编译后“`getX`”也会有名称“`getX`”。

1.5.3 实例：编写 C 版本的 `libmydemo.a`

接下来的例子，将把 `libmydemo.a` 重新用 C 语言实现。实现后的 C 语言版本的 `libmydemo.a` 和在使用上没有任何区别，同样可以被 `invoke.pas` 链接并以相同的方式调用。

首先打开终端，然后使用下面的步骤来创建 C 版本的 `libmydemo.a`。

1. 准备文件 `mydemo_a.c`，内容如下。

```
int getX()  
{  
    extern int my_number_X;  
    return my_number_X;  
}
```

```
int minus(int a, int b)
{
    return a - b;
}
```

2. 准备文件 mydemo_b.c，内容如下。

```
int my_number_X;
```

3. 分别编译 mydemo_a.c 和 mydemo_b.c。如下所示。

```
$ gcc -c -o mydemo_a.o mydemo_a.c
$ gcc -c -o mydemo_b.o mydemo_b.c
```

4. 创建静态库 libmydemo.a。如下所示。

```
$ ar rcs libmydemo.a mydemo_a.o mydemo_b.o
```

至此，静态库 libmydemo.a 的语言版本已创建完毕。这个 C 语言版本的 libmydemo.a 在使用上就跟 Pascal 版本的 libmydemo.a 一样。

同 pascal 版本一样，如果是32位程序，C 语言版本也要注意调用约定的问题。对于 Linux 下通用的 gcc 编译器来说，默认使用的是 cdecl 调用约定。这和 Linux 下通用的调用约定是一致的。因此，大部分情况下，C 程序不需要再指名相应的调用约定。若需要指名相应的调用约定，那么可以使用“__attribute__”。例如，要指定 getX() 为 stdcall 调用约定，那么要写如下的形式。

```
int getX() __attribute__((stdcall))
```

1.5.4 实例：使用 C 语言调用 libmydemo.a

Free Pascal 编写的静态库同样可以被其它语言所使用。这一节的例子介绍如何用 C 语言

调用 Free Pascal 编写的 libmydemo.a。需要说明的是，这一节的例子，同样也能用于 C 版本的 libmydemo.a。

打开终端窗口，切换到 libmydemo.a 所在目录。以下的步骤将创建一个与 invoke.pas 等价的 C 语言程序，这个程序将连接到 libmydemo.a。

1. 在 libmydemo.a 所在的目录下创建子目录 c_invoke，并切换到该子目录。
2. 创建文件 invoke.c，内容如下。

```
#include <stdio.h>

extern int my_number_X;

int getX();
int minus(int a, int b);

int main()
{
    printf("Enter X: ");
    scanf("%d", &my_number_X);
    printf("X = %d\n", getX());
    printf("X - 1 = %d\n", minus(my_number_X, 1));
    return 0;
}
```

3. 编译 invoke.c，如下所示。

```
$ gcc invoke.c -static -L. -lmydemo -o invoke
```

其中，“-static”是指链接到静态库，“-L.”指定额外的库搜索路径（也就是 libmydemo.a 所在目录），“-lmydemo”说明链接到 libmydemo.a，“-o invoke”说明生成的可执行文件名

为 “invoke” 。

4. 运行编译后的 invoke，如下所示。

```
$ ./invoke
Enter X: 2147483647
X = 2147483647
X - 1 = 2147483646
```

同样的，如果是32位系统，invoke.c 也需要注意函数调用约定。具体方法见1.5.3节。

1.5.5 实例：将 libmydemo.a 转换成动态库 libmydemo.so

这一节的例子建立在 Pascal 版本的 libmydemo.a 的基础上，将 libmydemo.a 转换成动态库 libmydemo.so。首先打开终端窗口，切换到 libmydemo.a 的目录，然后照着下面的步骤，将 libmydemo.a 转换成动态库。

1. 准备文件 mydemo.pas，内容如下。

```
library mydemo;
{$mode objfpc}
uses
mydemo_a in 'mydemo_a.pas',
mydemo_b in 'mydemo_b.pas';

exports
my_number_X,
getX,
minus;

begin
```

```
{ifdef V1}

Writeln(' libmydemo.so.1 loaded');

{$else}

Writeln(' libmydemo.so.2 loaded');

{$endif}

end.
```

在编译成动态库的时候，使用之前的方法来公开一个符号是不行的。必须在” library” 的 “exports” 节里面显式将这些符号导出。“begin” 这一节的内容，则会在库被加载的时候执行。

2. 将 mydemo.pas 分别编译成 libmydemo.so.1 以及 libmydemo.so.2。如下所示。

```
$ fpc -fPIC -dV1 -olibmydemo.so.1 mydemo.pas

Free Pascal Compiler version 2.4.0-2ubuntu1.10.04 [2011/06/17] for x86_64
Copyright (c) 1993-2009 by Florian Klaempfl
Target OS: Linux for x86-64
Compiling mydemo.pas
Compiling mydemo_a.pas
Compiling mydemo_b.pas
Linking libmydemo.so.1
/usr/bin/ld: warning: link.res contains output sections; did you forget -T?
51 lines compiled, 0.0 sec

$ fpc -fPIC -dV2 -olibmydemo.so.2 mydemo.pas

Free Pascal Compiler version 2.4.0-2ubuntu1.10.04 [2011/06/17] for x86_64
Copyright (c) 1993-2009 by Florian Klaempfl
Target OS: Linux for x86-64
Compiling mydemo.pas
```

```
Linking libmydemo.so.2
```

```
/usr/bin/ld: warning: link.res contains output sections; did you forget -T?
```

```
17 lines compiled, 0.0 sec
```

这样子，就生成了两个互不兼容的动态库 libmydemo.so.1 以及 libmydemo.so.2。其中，“-fPIC”编译选项用于指定编译器将生成位置无关代码，这样生成的动态库才能被加载到内存的任意位置。而“-dV1”、“-dV2”这样子的编译参数用于定义一些符号，源代码里面的 {ifdef V1} 就是用于判断是否定义了“V1”这个符号。如果定义了 V1 这个符号，那么就编译源代码里面的这一段：

```
Writeln('libmydemo.so.1 loaded');
```

否则（也就是没有定义“V1”这个符号），那么就编译源代码里面的这一段：

```
Writeln('libmydemo.so.2 loaded');
```

3. 建立符号链接 libmydemo.so 到 libmydemo.so.1，如下所示。

```
$ ln -sfv libmydemo.so.1 libmydemo.so
```

```
"libmydemo.so" -> "libmydemo.so.1"
```

这样，以后每当引用“libmydemo.so”的时候，就会自动引用“libmydemo.so.1”了。

那么，为什么这里要创建两个不同的 libmydemo.so 版本呢？详细信息，得从 soname 这个概念说起。详细信息，请看下一节。

1.5.6 实例：使用 Free Pascal 调用 libmydemo.so

使用 Free Pascal 调用一个动态库，除了可以用传统的 Delphi 方法“external 动态库名 name 符号名”以外，还可以使用前边 invoke.pas 所使用的方法。使用 invoke.pas 所用的方法调用 libmydemo.so 时，只需要修改其编译参数即可（这种方式同样适合于调用其它语言编写的动态库），如下所示。

```
$ fpc -Fl.. invoke.pas
```

```
Free Pascal Compiler version 2.4.0-2ubuntu1.10.04 [2011/06/17] for x86_64
Copyright (c) 1993-2009 by Florian Klaempfl
Target OS: Linux for x86-64
Compiling invoke.pas
Linking invoke
/usr/bin/ld: warning: link.res contains output sections; did you forget -T?
16 lines compiled, 0.1 sec
```

这样编译以后，invoke 就被链接到 libmydemo.so.1（注意，不是“libmydemo.so”，尽管代码里的“{\$linklib mydemo}”会让连接器在链接的过程中寻找 libmydemo.so 而不是 libmydemo.so.1）了。与链接到静态库不同，链接到动态库是默认的动作，只要不使用“-Xt”选项。另外，由于现在 libmydemo.so.1 并不在动态库的标准路径，所以在运行 invoke 的时候，要使用环境变量 LD_LIBRARY_PATH 来指定库的搜索路径。invoke 的运行结果如下所示。

```
$ LD_LIBRARY_PATH=.. ./invoke
libmydemo.so.1 loaded
Enter X: 2147483647
X = 2147483647
X - 1 = 2147483646
```

由于 mydemo.pas 的“begin”一节的内容会在库加载时运行，所以如上所示，invoke 运行结果的第一行就是“libmydemo.so.1 loaded”。

那么，为什么 invoke 链接到的是“libmydemo.so.1”而不是“libmydemo.so”呢？因为在链接的时候，写入到 invoke 里面的，是 libmydemo.so 的 soname（libmydemo.so 实际上是指向 libmydemo.so.1 的一个符号链接），也就是“libmydemo.so.1”。这个 soname 是存在于动态库的文件里面的，和文件名无关。假如只有一个 soname 为“libmydemo.so.1”的动态库文件 libmydemo.so，那么链接以后 invoke 仍将链接到“libmydemo.so.1”而不是“libmydemo.so”。要查看一个动态库的 soname，可以用如下的方法。

```
$ objdump libmydemo.so -p | grep SONAME  
  
SONAME                libmydemo.so.1
```

Free Pascal 在编译一个动态库的时候，会将生成的动态库的 soname 跟生成的动态库的文件名一致。所以，当指定输出的文件名是“libmydemo.so.1”的时候，最后生成的动态库文件的 soname 也会是“libmydemo.so.1”。

那么，soname 又是什么呢？soname 是动态库的一个唯一标识，不同的 soname 用于说明两个接口互不兼容的动态库。Linux 使用 soname 来对库进行管理。在 Linux 下，一般库的文件名采用“lib 库名.so.版本号”的形式，例如“libmydemo.so.1.2.3.4”。而对于每个库文件里面，则都保存着一个 soname，它一般是“lib 库名.so”或者“lib 库名.so.主版本号”的形式。同一个 soname 可以对应很多个不同版本的库文件，但是它们对外的接口都是一样的。每当将一个新的动态库放入标准目录以后，建议运行一下 ldconfig。如果系统里面本身没有任何一个库拥有这个新放入的库的 soname，那么 ldconfig 会产生一个以 soname 命名的链接，指向这个库文件。在这以后，由于引用这个库的应用程序里面记录的是这个库的 soname，并且，在这个应用程序启动的时候，会寻找以 soname 为名称的相应的动态库。如果根据找不到以 soname 为名的动态库文件，那么这个应用程序便无法启动，并且产生一个错误，如下所示。

```
$ ls /lib/libmydemo.so.1  
ls: 无法访问/lib/libmydemo.so.1: 没有那个文件或目录  
$ ./invoke  
./invoke: error while loading shared libraries: libmydemo.so.1: cannot open shared  
object file: No such file or directory
```

现在将 libmydemo.so.1 重命名为 libmydemo.so.1.2.3.4（也就是版本号为 1.2.3.4 且 soname 为“libmydemo.so.1”的动态库），然后将其放入 /lib，再运行 ldconfig，如下所示。

```
$ mv libmydemo.so.1 libmydemo.so.1.2.3.4  
$ sudo cp -fv libmydemo.so.1.2.3.4 /lib 我在左边的的事件页一选择就加几行，还没做什么就插入了 N 多行的代码，我觉得能不能双击再添加
```



```
"libmydemo.so.1.2.3.4" -> "/lib/libmydemo.so.1.2.3.4"

$ pascal_invoke/invoke

pascal_invoke/invoke: error while loading shared libraries: libmydemo.so.1: cannot
open shared object file: No such file or directory

$ sudo ldconfig

$ ls /lib/libmydemo.so* -lh

lrwxrwxrwx 1 root root 20 2012-01-18 13:30 /lib/libmydemo.so.1 ->
libmydemo.so.1.2.3.4

-rwxr-xr-x 1 root root 169K 2012-01-18 13:30 /lib/libmydemo.so.1.2.3.4

$ pascal_invoke/invoke

libmydemo.so.1 loaded

Enter X: 2147483647

X = 2147483647

X - 1 = 2147483646
```

在这个例子里面，ldconfig 根据 libmydemo.so.1.2.3.4 的 soname，创建了一个叫 libmydemo.so.1 的链接，这个链接指向 libmydemo.so.1.2.3.4。并且，在链接 libmydemo.so.1 没有创立之前，是无法运行 invoke 的，创建了以后，那么 invoke 就可以顺利运行。毕竟 invoke 是通过“libmydemo.so.1”来寻找该文件的。

同样的，如果将 libmydemo.so.2 以 libmydemo.so.2.3.4.5 的名字放入到 /lib 里面，然后再运行 ldconfig，则也会创建链接 libmydemo.so.2 到 libmydemo.so.2.3.4.5。如下所示。

```
$ ls /lib/libmydemo.so* -lh

lrwxrwxrwx 1 root root 20 2012-01-18 13:32 /lib/libmydemo.so.1 ->
libmydemo.so.1.2.3.4

-rwxr-xr-x 1 root root 169K 2012-01-18 13:30 /lib/libmydemo.so.1.2.3.4

$ sudo cp -fv libmydemo.so.2 /lib/libmydemo.so.2.3.4.5
```

```
"libmydemo.so.2" -> "/lib/libmydemo.so.2.3.4.5"

$ sudo ldconfig

$ ls /lib/libmydemo.so* -lh

lrwxrwxrwx 1 root root 20 2012-01-18 13:32 /lib/libmydemo.so.1 ->
libmydemo.so.1.2.3.4

-rwxr-xr-x 1 root root 169K 2012-01-18 13:30 /lib/libmydemo.so.1.2.3.4

lrwxrwxrwx 1 root root 20 2012-01-18 13:38 /lib/libmydemo.so.2 ->
libmydemo.so.2.3.4.5

-rwxr-xr-x 1 root root 169K 2012-01-18 13:38 /lib/libmydemo.so.2.3.4.5
```

如果这时候再有个libmydemo.so.1的新版本libmydemo.so.1.2.3.5,将其放入/lib以后,再运行ldconfig,那么它会将libmydemo.so.1的链接指向这个新版本libmydemo.so.1.2.3.5。如下所示。

```
$ ls /lib/libmydemo.so* -lh

lrwxrwxrwx 1 root root 20 2012-01-18 13:32 /lib/libmydemo.so.1 ->
libmydemo.so.1.2.3.4

-rwxr-xr-x 1 root root 169K 2012-01-18 13:30 /lib/libmydemo.so.1.2.3.4

lrwxrwxrwx 1 root root 20 2012-01-18 13:38 /lib/libmydemo.so.2 ->
libmydemo.so.2.3.4.5

-rwxr-xr-x 1 root root 169K 2012-01-18 13:38 /lib/libmydemo.so.2.3.4.5

$ sudo cp -fv /lib/libmydemo.so.1.2.3.4 /lib/libmydemo.so.1.2.3.5

"/lib/libmydemo.so.1.2.3.4" -> "/lib/libmydemo.so.1.2.3.5"

$ sudo ldconfig

$ ls /lib/libmydemo.so* -lh

lrwxrwxrwx 1 root root 20 2012-01-18 13:46 /lib/libmydemo.so.1 ->
libmydemo.so.1.2.3.5
```

```
-rwxr-xr-x 1 root root 169K 2012-01-18 13:30 /lib/libmydemo.so.1.2.3.4
-rwxr-xr-x 1 root root 169K 2012-01-18 13:45 /lib/libmydemo.so.1.2.3.5
lrwxrwxrwx 1 root root 20 2012-01-18 13:38 /lib/libmydemo.so.2 ->
libmydemo.so.2.3.4.5
-rwxr-xr-x 1 root root 169K 2012-01-18 13:38 /lib/libmydemo.so.2.3.4.5
```

后每当引用 libmydemo.so.1 的时候，实际引用的就是这个新版本的 libmydemo.so.1.2.3.5。而由于 libmydemo.so.1.2.3.4 和 libmydemo.so.1.2.3.5 都拥有同样的应用程序接口，因而在使用新版本的库的同时又能保持应用程序的兼容性。

那么，当系统同时拥有 libmydemo.so.1 和 libmydemo.so.2 的时候，如何为 libmydemo.so 选择默认版本呢？只需要建立一个链接 libmydemo.so 指向 libmydemo.so.1 或者 libmydemo.so.2，如下所示。

```
$ sudo ln -sfv /lib/libmydemo.so.1 /lib/libmydemo.so
"/lib/libmydemo.so" -> "/lib/libmydemo.so.1"
```

以后每当链接器寻找 libmydemo.so 的时候，都能使用指定的 libmydemo.so.1 或者 libmydemo.so.2。

1.5.7 实例：使用 C 语言调用 libmydemo.so

使用 C 语言调用 Free Pascal 编写的 libmydemo.so，跟调用 libmydemo.a 从代码上没什么区别。仅仅只是在链接的时候去掉“-static”即可，如下所示。

```
gcc invoke.c -L. -lmydemo -o invoke
```

运行 invoke，则 mydemo.pas 里“begin”一节的内容也同样会被执行，如下所示。

```
$ LD_LIBRARY_PATH=.. ./invoke
libmydemo.so.1 loaded
Enter X: 2147483647
X = 2147483647
```

```
X - 1 = 2147483646
```

1.5.8 一些补充

讲了这么多，以上内容已经涵盖了常用的 Linux 下库编程的大部分方法。也许有的读者会问，前边只是讲到了动态库的初始化代码，那么由如何使得库在卸载的时候做一些收尾工作呢？其实，这些代码可以通过单元的 finalization 节来实现。而初始化代码，并不一定需要在 library 的 “begin” 节来实现，同样也可以通过单元的 initialization 节来实现。例如，在 libmydemo.so 的 mydemo_a.pas 里面增加 initialization 和 finalization 节，如下所示。

```
unit mydemo_a;

{$mode objfpc}

interface

function getX:Integer;
function minus(a,b : Integer):Integer;

implementation

uses
mydemo_b, syscall;

function getX:Integer;alias:'getX';
begin
    Result := my_number_X;
end; { getX }
```

```
function minus(a,b : Integer):Integer;alias:'minus';
begin
    Result := a - b;
end; { plus }

initialization
Writeln(' mydemo_a init');

finalization
Writeln(' mydemo_a finalize');

end.
```

重新编译 libmydemo.so.1 以后，运行 C 版本的 invoke 程序，这时候 initialization 和 finalization 都会被自动执行，如下所示。

```
$ LD_LIBRARY_PATH=.. ./invoke
mydemo_a init
libmydemo.so.1 loaded
Enter X: 3
X = 3
X - 1 = 2
mydemo_a finalize
```

在函数的调用约定方面，编写静态库和编写动态库方面均使用相同的语法。而这在64位 Linux 系统下一般是不需要操心的，但如果是32位 Linux 系统，调用双方都要保证一致的调用约定。在这方面，如果要使一个程序能够32位和64位通用，那么只需要考虑32位的情况。因为，对于64位程序来说，各种说明调用约定的关键字其实都是一样的。

前边讲的调用库的方法，都是静态的将应用程序链接到对应的动态库。如果想要在运行时

动态加载动态库，可以使用 dl 单元。dl 单元实现了对 libdl 库的封装。通常的使用方法是，首先调用 dlopen() 来加载要加载的库，然后调用 dlsym() 来获取每个符号对应的地址，最后使用 dlclose() 来关闭相应的库。若在使用 libdl 的过程中出错，可以调用 dlerror() 来获取错误信息。下面的例子显示了如何调用加了 initialization 和 finalization 的 libmydemo.so.2。

```
program invoke;

{$mode objfpc}

uses
    dl;

type
    TGetX = function:Integer;
    TMinus = function(a,b:Integer):Integer;

var
    my_number_X : PInteger;
    libmydemo_handler: Pointer;
    getX : TGetX;
    minus : TMinus;

begin
    { load the library }
    libmydemo_handler := dlopen('../libmydemo.so.2', RTLD_LAZY);
    if libmydemo_handler = nil then
```

```
begin

    Writeln(StdErr, string(dlerror));

    Exit;

end;

{ retrieve the addresses of the symbols }

my_number_X := PInteger( dlsym( libmydemo_handler, 'my_number_X' ) );

getX := TGetX( dlsym( libmydemo_handler, 'getX' ) );

minus := TMinus( dlsym( libmydemo_handler, 'minus' ) );


Write('Enter X: ');

Readln(my_number_X^);

Writeln('X = ', getX);

Writeln('X - 1 = ', minus(my_number_X^, 1));


dlclose(libmydemo_handler);

end.
```

在 libmydemo. so. 2 的目录下创建一个新目录，输入以上程序，编译并运行，结果如下。

```
$ fpc invoke.pas

Free Pascal Compiler version 2.4.0-2ubuntu1.10.04 [2011/06/17] for x86_64

Copyright (c) 1993-2009 by Florian Klaempfl

Target OS: Linux for x86-64

Compiling invoke.pas

Linking invoke
```

```
/usr/bin/ld: warning: link.res contains output sections; did you forget -T?

39 lines compiled, 0.0 sec

$ ./invoke

mydemo_a init

libmydemo.so.2 loaded

Enter X: 2147483647

X = 2147483647

X - 1 = 2147483646

mydemo_a finalize
```

1.6 发布你的程序

当一个程序完成以后，如何将这个程序发布给用户呢？这是一个值得探讨的问题。对于 Windows 来说，大部分情况下只需要将文件复制给用户即可，进一步的话还可以再做一个安装程序，这些往往都没有很大的困难。然而对于 Linux 应用程序来说，由于不同 Linux 发行版之间的差异较大（这种差异大到甚至连操作系统内核、C 标准库等系统级别的内容往往都会有所不同），而且互不兼容，因此在 Linux 下发布一个应用程序是一个相对复杂的话题。

在 Linux 下发布一个应用程序往往有两种情况，一种是给用户以源代码，用户在自己的电脑上编译并使用，这种情况往往用于开源软件；另外一种则是给用户以编译好的程序，用户不需要再自己编译即可运行该程序。下面分别讨论这两种情况。

1.6.1 使用源代码进行发布

使用源代码的形式发布软件，首先要准备好程序的源代码以及相关的数据文件等内容。对于简单的工程，Lazarus 的菜单项“Project”->“Publish Project”可以将主要的源代码文件复制到一个指定的目录。

有了基本的源代码和相关文件以后，也许简单的程序已经可以拿到别的装有 Lazarus 的机器上顺利地编译和运行了，这时候往往意味着，作为用户，也得启动 Lazarus，然后按照开发者的说明一步一步的加载工程，做一些设置，然后再在 IDE 里面编译。而实际上，大部分用户

并不希望这么做，更加通用的做法是，使用一些简单的命令，输入这些命令之后，可以简单地编译出最终的文件。这种情况下往往需要使用脚本程序来控制编译过程，而这最通用的方法，莫过于使用 Make。

Make 的作用是读取一个 Makefile 文件，然后根据 Makefile 执行一些命令，而往往这些命令的作用就是对程序进行编译。一个 Makefile 文件由一些目标（例如，编译后的文件）组成，每个目标会有一些依赖项（例如，源代码文件）以及根据这些依赖项生成目标所需要执行的命令。当 Make 运行的时候，会根据 Makefile 里面的各个目标，决定是否执行某些命令以及这些命令执行的先后顺序，以确保最终的目标能够正确生成。

Make 软件实际上有很多种，而 Linux 下最常使用的就是 GNU Make。实际上，完全可以将 GNU Make 视为 Linux 世界的标准，因为几乎所有发行版里面的开发工具都包含有 GNU Make，而目前绝大部分 Linux 下的开源软件都使用 GNU Make，这包括 Lazarus 本身，也包括 Free Pascal 本身。而即使不在 Linux 下，大部分开源平台上仍旧默认使用 GNU Make，即使 Windows 下的 mingw/cygwin 等也是如此。因而一般情况下，只需要针对 GNU Make 编写 Makefile 即可到任意平台上编译。

然而，仅仅编写一个通用的 Makefile 仍然不够。在 Linux 发行版之间的巨大差异下，尽管使用源代码发布往往使得程序拥有相对较好的兼容性（只要源代码能够保持良好的跨平台特性，那么到不同机器上编译后的源代码往往能够顺利地在该机器上运行），但实际上，特别是对于大型项目来说，这种源代码的兼容性也并非那么容易实现。因为即使是 C 标准库（对于大部分 Linux 操作系统来说，这对大部分应用程序来说都是十分基本的一个库，几乎所有的应用程序都要使用到它），在不同发行版、编译器上仍旧是不一样的，且不说只是实现上的不一样，甚至连函数原型都有可能不同。对于这种问题，目前通用的解决方案是使用 autotools，几乎所有流行的 Linux 下的开源软件都使用它。

autotools 并不是一款软件，而是指 autoconf 和 automake 两个独立的软件。这两个软件结合使用，最终生成一个叫“configure”的 bash 脚本程序。这个脚本程序往往长达几万行，它的作用就是猜测当前环境的各种信息，除了当前是什么系统，都有哪些编译器之类等信息，

甚至还包括哪个函数是否存在，函数原型是什么等复杂信息。在 configure 执行的过程中，如果系统缺少编译所需要的东西，那么会给用户以相应的提示。configure 执行完以后，会根据当前系统的具体信息生成一个复杂的 Makefile 文件（里面的往往包括了编译、安装、清除等功能），然后用户再通过 make 来进行编译、安装等操作。常见的编译安装操作步骤

“./configure, make, make install”使用的就是 autotools。而在程序的源代码方面，由于有了当前系统的详细信息，因而可以根据这些信息编译（例如，使用条件编译）出针对当前系统的代码。

尽管 autotools 是最通用的选择，然而类似的软件还有很多。例如，Free Pascal 自带了 fpcmake，所需要的只是编写一个 Makefile.fpc 文件，然后 fpcmake 可以生成一个平台通用的 Makefile 文件；另外，由于这几年由于 KDE 项目全面转向使用 CMake，因而 CMake 的使用也变得广泛起来。

1.6.2 发布编译好的应用程序

发布编译好的应用程序往往比发布源代码更难以实现发 Linux 行版之间的兼容。往往在一个 Linux 发行版里面编译好的应用程序，在另外一个 Linux 发行版里面就无法正常运行。就算是同一个发行版的不同版本（例如，Ubuntu 10.04和 Ubuntu 11.10），也有可能无法正常运行。实际上，Linux 发行版向来没有“向下兼容”的说法。一个发行版的不同版本，应该作为两个发行版看待。

那么，如何将编译好的应用程序发布给用户并且让其能正常运行呢？首先要明白的一点是，只要是 native 应用程序，就不可能兼容所有的 Linux 发行版。唯一的做法只能是，限制用户所使用的发行版。如果该程序只能运行于某发行版的某个版本，那么用户的 Linux 发行版也只能是这些特定的发行版的特定版本。这就是为什么闭源的商业软件往往在系统需求上精确到某发行版的某个特定版本的原因，例如，Intel C++ Compiler 11.1 for Linux IA-32只支持这些发行版的这些版本：Asianux 3.0、Debian 4.0、Fedora 10、Red Hat Enterprise Linux 3,4,5、SUSE LINUX Enterprise Server 9,10,11,11 SP1、TurboLinux 11、Ubuntu 9.04。

尽管无法兼容所有的 Linux 发行版，但是兼容某几个特定的 Linux 发行版还是有可能的，

上面提到的 Intel C++ Compiler for Linux IA-32就兼容了12个 Linux 发行版。要做到这一点，有一项技术是值得考虑的，那就是 LSB。

LSB 是 Linux Standard Base 的缩写，它是一个 Linux 发行版的标准，尽管实际上并没有多少 Linux 发行版兼容它，或者就算兼容，也不是默认就兼容，而通常是通过从软件仓库安装一些软件才能对其兼容。不过这几年，不少流行的 Linux 发行版都宣称兼容 LSB 标准或者对其提供支持。

要使一个 Linux 应用程序能够兼容流行的 Linux 发行版，可以从兼容 LSB 入手。LSB 官方网站提供了一整套的开发测试工具和兼容性检查工具。如果一个应用程序是使用 LSB SDK 开发出来的，那么它本身便是兼容 LSB 标准的。由于 LSB SDK 里面所使用的软件（特别是系统级软件）版本都比较低，而且由于是标准的原因从程序接口上考虑了更多的发行版，因而 LSB SDK 开发出来的软件往往有更好的兼容性。一个程序由 LSB SDK 开发出来以后，可以使用 LSB 提供的兼容性测试程序，它可以静态检查编译好的应用程序的二进制兼容性，最后给出一份报告，这份报告会说明这个程序兼容那些流行的 Linux 发行版，哪些地方不兼容，然后再根据实际情况进行修改。在所有的兼容性通过以后，建议再到实际的发行版环境里面进行足够的测试，这样才能保证真正的兼容。

值得一提的是，使用 LSB SDK 开发出来的程序，有可能在默认不支持 LSB 的发行版上根本无法运行。这得从动态链接器 ld-linux 说起。先看下面 ldd 的输出，ldd 命令用于列出某个应用程序或者库都链接到哪些文件。

```
$ ldd /lib/libgcc_s.so.1
    linux-vdso.so.1 => (0x00007fffe1ebf000)
    libc.so.6 => /lib/libc.so.6 (0x00007f989d18f000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f989d759000)
$ ldd /bin/ls
    linux-vdso.so.1 => (0x00007fff1a7fb000)
    librt.so.1 => /lib/librt.so.1 (0x00007faf72b6e000)
```

```
libselinux.so.1 => /lib/libselinux.so.1 (0x00007faf72950000)
libacl.so.1 => /lib/libacl.so.1 (0x00007faf72747000)
libc.so.6 => /lib/libc.so.6 (0x00007faf723c4000)
libpthread.so.0 => /lib/libpthread.so.0 (0x00007faf721a7000)
/lib64/ld-linux-x86-64.so.2 (0x00007faf72da6000)
libdl.so.2 => /lib/libdl.so.2 (0x00007faf71fa2000)
libattr.so.1 => /lib/libattr.so.1 (0x00007faf71d9d000)
```

注意其中均有一行“/lib64/ld-linux-x86-64.so.2”，这个文件在 x86 Linux 发行版里面一般为“/lib/ld-linux.so.2”，这个文件的具体名称主要是由具体的处理器决定的。除了这个文件本身，几乎所有的程序都链接到这个文件。那么这个文件是什么呢？尽管从文件名上看，这个文件很像一个动态库，但实际上，它本身是一个可执行程序。当一个 Linux 应用程序启动时，ld-linux 会被执行，ld-linux 会加载所有未被加载的动态库并且在内存里面进行动态链接以使得程序可以正常运行，完成后再开始执行该应用程序。对于这个应用程序来说，ld-linux 是以绝对路径的形式写死在可执行文件里面的。而对于 LSB 标准来说，这个程序不叫 ld-linux，而叫 ld-lsb（例如，LSB4.1 里面，IA32 架构这个文件为 /lib/ld-lsb.so.3，IA64 架构这个文件为 /lib/ld-lsb-ia64.so.3）。因而在不兼容 LSB 的发行版里面启动 LSB 应用程序的时候，会由于找不到 ld-lsb 而导致程序无法正常启动。那么这个问题如何解决呢？有两个办法，一个是在目标系统里面创建 ld-lsb（只需要创建一个指向 ld-linux 的链接即可）使得程序可以启动，但是这种方法会破坏目标系统；另外一种办法是在开发过程中，使用 ld（这是 Linux 下通用的链接器，被包括 Free Pascal Compiler 在内的大部分流行的 Linux 下的编译器所使用）链接的时候带上 `--dynamic-linker` 参数来指定特定的动态链接器。

其实程序的兼容性问题很大程度上也是程序所依赖库的兼容性问题。尽管 LSB 可以很大程度上解决基础库（也就是 LSB 标准里面列出的库）的依赖问题，然而，像 Lazarus 编译出来的程序，还依赖不少 LSB 所本身并不具备的库，如 GTK, Qt 的。对于程序里面使用的这些非基础库，使用自己提供的版本，可以很大程度上解决这种问题。那么怎样才能使用自己版本的库呢？尽量使用静态链接，将依赖的库全部包括在编译好的程序里面，这样这些库就可以不再依赖系

统原有的库。对于无法使用静态链接的库，那么，可以在提供自己版本的动态库的同时，通过指定 `LD_LIBRARY_PATH` 环境变量，使得程序引用自己提供版本的动态库，而不是用户系统上原有的库。

在兼容性问题上，路径问题也是一个值得考虑的。在应用程序编写的时候，应该尽量使用相对路径而不是绝对路径，这样可以使得程序可以被用户安装在任意位置。然而仅仅这样做还不够，特别是依赖第三方库的程序来说。有不少常见的开源软件，在编译的时候已经将相关的绝对路径写入到可执行文件里面了。不过这种软件往往要么在编译的时候可以通过指定某些特定参数来使用相对路径，要么可以通过环境变量来控制使用其它的目录，不过也有的并没有提供这两种方式。有的开源软件，甚至将引用的动态库的绝对路径写入到编译后的程序里面，不过大部分这种情况是可以使用 `binutils` 等软件进行查看和修改的。而在某些必要的情况下，使用 `chroot` 也是一个不错的选择。

用户权限问题也是一个值得考虑的兼容性问题。这取决于应用程序的具体需求。通常情况下，用户并不希望一个应用程序拥有超出它需要范围的权限。比如，Lazarus 本身并不需要使用 `root` 权限，而如果限制只能在 `root` 权限下运行 Lazarus，那是十分糟糕的情况。在用户权限问题上，一般可以采取这样子的策略：如果程序没有特殊要求，那么应该让用户既可以顺利的安装在自己主文件夹(也就是“`~`”目录)里面，也可以以 `root` 权限安装在系统目录(如 `/opt`)中，但是在使用的时候，所有用户都可以以自己的身份方便地使用该程序(不需要系统管理员身份也能运行)。权限问题同时也限制了配置文件的存放位置，比如，普通用户身份运行的以 `root` 身份安装程序，一般不能直接将配置文件写在程序所在的目录(这通常是 Windows 应用程序的做法)。那么，该如何安排配置文件呢？一般的做法是这样的：对于系统级的配置文件，往往可以放在一个叫“`etc`”的目录，对于商业程序来说，这个目录往往在程序安装目录里面；对于用户级的配置文件，则在“`~`”目录下创建一个专属于本程序的目录，而且这个目录通常以“`.`”开头(以“`.`”开头的文件和目录往往不被显示出来)，然后再在该目录下记录程序的配置信息，而如果用户级别的配置信息并不多，仅仅使用一个以“`.`”开头的配置文件也是可行的。

除了应用程序兼容性问题，还有一个值得考虑的是程序的打包和安装程序制作的问题。由

于编译好的应用程序只能兼容某些特定的发行版,那么一种常用的方法就是分别为这些发行版单独打包制作安装程序,并且使用这些发行版使用的包管理器,制作单独的软件包(比如,单独为 Ubuntu 10.04制作 deb 包,单独为 RHEL 6.0制作 rpm 包,等等)。除此以外,还可以制作通用的安装程序。通用的安装程序除了可以在特定的发行版上顺利安装以外,还可以在其它 Linux 发行版(包括那些兼容性未知的 Linux 发行版)上进行安装。同样的,安装程序也不可能在所有的发行版上顺利运行。实际的安装程序往往以 bash 脚本的形式出现,而且大部分为纯命令行形式的安装程序。使用 bash 脚本的好处在于,一来它是以源代码的形式提供的;二来 bash 几乎是当前 Linux 发行版 Shell 的实际标准;三来它可以在程序的末尾附上二进制数据,这往往被用于制作自解压安装程序。而使用命令行界面来实现安装程序的好处在于,并非所有目标系统都在本机安装有图形界面,所以就算是 GUI 应用程序大部分也使用命令行界面的安装程序(至于安装后的 GUI 应用程序,在本机运行程序但图形界面在其它机器上的可能性也不是没有的)。在打包和安装程序制作方面的工具选择上,软件包的制作可以使用 Linux 发行版官方提供的工具;安装程序的制作则可以考虑一些开源的或者商业的安装程序制作软件(更多的是大型商业软件才用,有不少软件的安装程序都是独立编写的),如 InstallAnywhere, BitRock InstallBuilder, IzPack, 等等。

第 2 章

WINCE 应用

2.1 WINCE 运行环境的建立

2.1.1 建立 WINCE 编译环境

WINCE 是微软的嵌入式操作系统，是一个基础性质的系统。自从 2.2.0 FPC（Free Pascal Compile）开始，就支持 WINCE -ARM 平台。要开发 WINCE 平台的软件，必须安装 WINCE 的交叉编译软件，如 Lazarus-0.9.28.2-fpc-2.2.4-cross-arm-wince-win32.exe，该程序为 LAZARUS 提供 WINCE 交叉编译功能，可在 LAZARUS 官方网站下载。

在前面的章节，我们成功建立了 LAZARUS 的编译环境，WINCE 的交叉编译功能就在此基础上建立：点击运行 Lazarus-0.9.28.2-fpc-2.2.4-cross-arm-wince-win32.exe，选择前面建立的 LAZARUS 路径进行安装，安装完毕后，WINCE 的编译环境建立完成。

2.1.2 建立 WINCE 运行环境

WINCE 操作系统在消费类电子应用较广泛，在工业控制领域也有不少的应用。采用 WINCE 操作系统的产品主要有：PDA（如经典的 COMPAQ 36xx 系列），智能手机（Windows Mobile，WINCE 的衍生版本），GPS 导航仪（安装有凯立德，或灵图天行者导航软件），MP4 等。

WINCE 定位在微处理器（如 ARM）硬件架构上，故此，WINCE 的程序无法运行在 PC 上，必须运行在安装了 WINCE 系统的设备上。常用的方法：

- （1）将 WINCE 程序复制到 SD 卡，CF 卡等移动存储器上，在支持这些移动存储器的 WINCE 设备上找到该程序，并运行之；
- （2）通过微软的 ACTIVESYNC 软件，将 WINCE 程序复制到 WINCE 设备中，并运行之；
- （3）在没有硬件的情况下，我们可以借助微软的 WINCE 模拟器来运行程序，常用的有 WINCE 5.0 的模拟器和 WINCE 6.0 的模拟器。

模拟器虚拟的一个 WINCE 的设备空间，我们将程序装入该空间，再通过模拟器的虚拟界面运行 WINCE 程序。

2.2 创建第一个例子

2.2.1 建立 LAZARUS 的 WINCE 工程

如图 2.2-1 所示，在 LAZARUS 的 IDE 菜单选“File”—》“New”；接着在弹出的界面里选择“Application”，见图 2.2-2；在窗体 FORM1 放一个按钮 Button1，见图 2.2-3。

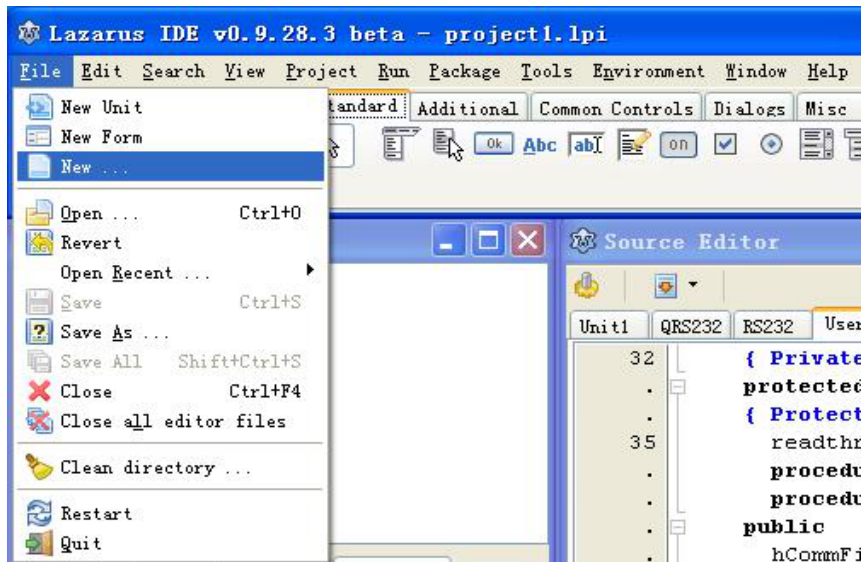


图 2.2-1

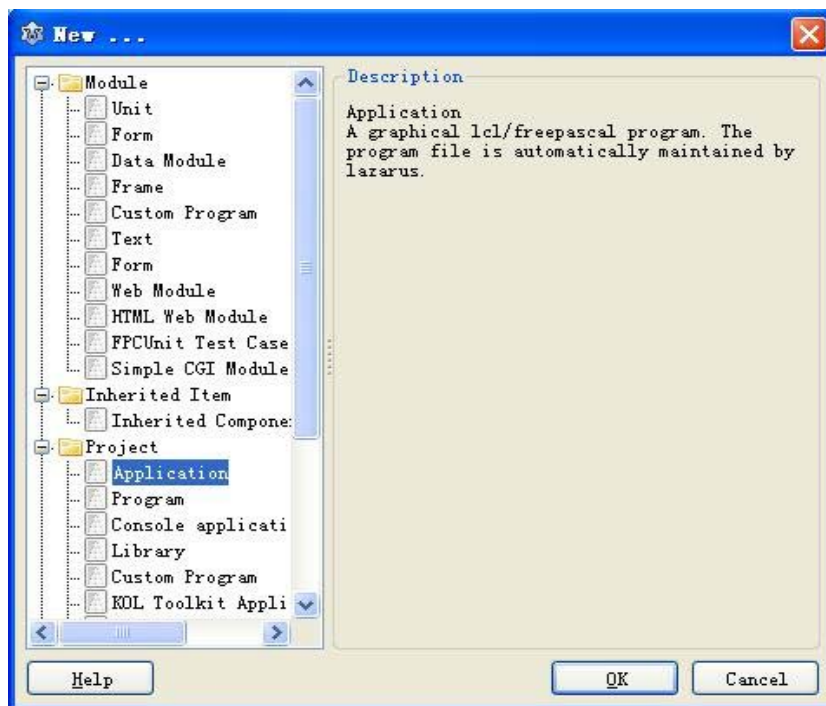


图 2.2-2

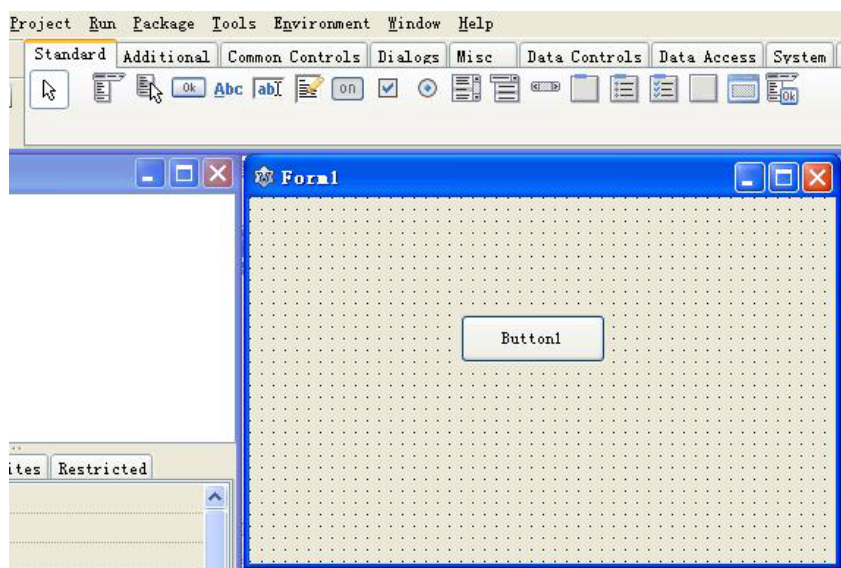


图 2.2-3

双击按钮 Button1，在按钮的双击事件上输入如图 2.2-4 中的代码：

```
procedure TForm1.Button1Click(sender:TObject);  
begin  
    showmessage('hello world');    // 本行是手工输入的代码  
End;
```

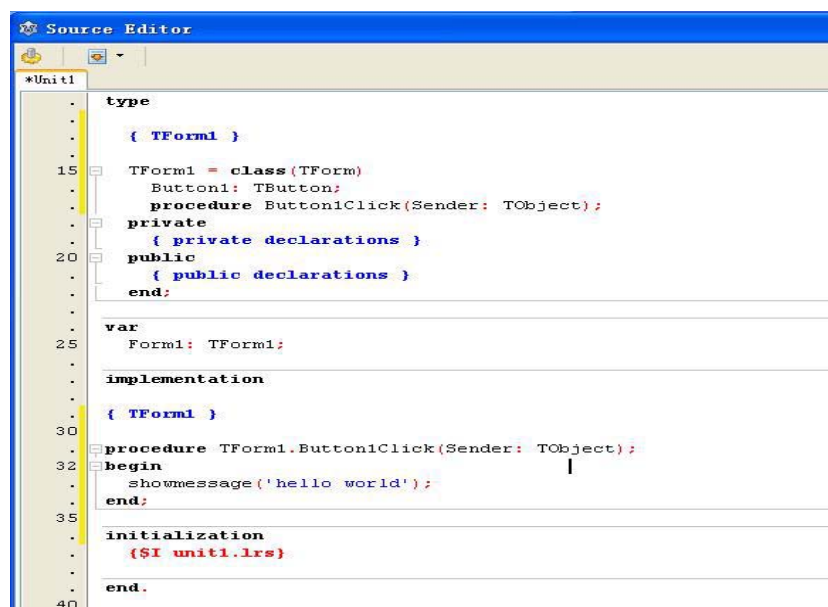


图 2.2-4

接着把工程文件和源文件都保存到硬盘里，WINCE 工程建立完毕。

2.2.2 编译 LAZARUS 的 WINCE 工程

在 IDE 里，按下键盘的“F9”键，运行当前程序。

程序简单，没有任何的错误提示，编译通过了，运行的效果如图 2.2-5 所示。

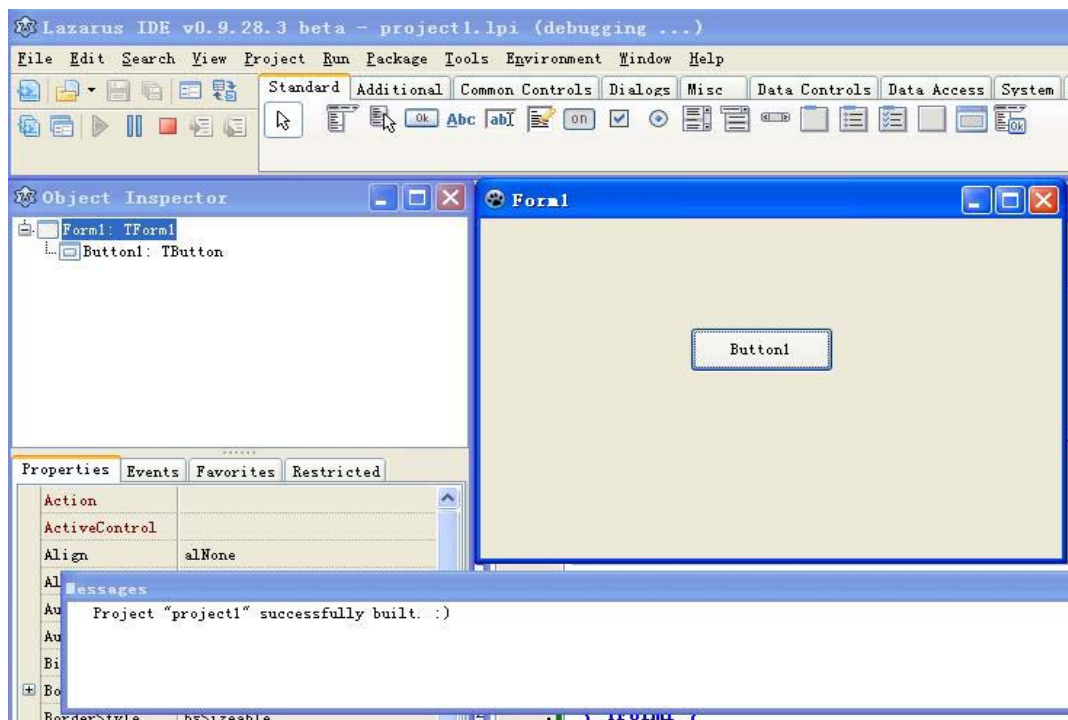


图 2.2-5

点击图 2.2-5 的按钮 Button1，我们看到弹出了一个信息框，里面显示“hello world”，正是我们要求的效果，看看图 2.2-6。

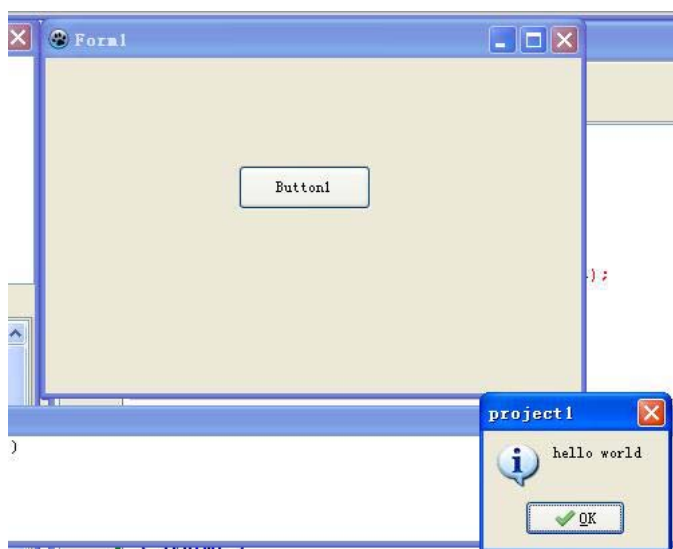


图 2.2-6

效果不错嘛，不过，别高兴太早，这个程序只能在 Windows 里运行，并不是我们所希望的 WINCE 环境里运行，还有工作在等待我们去做……

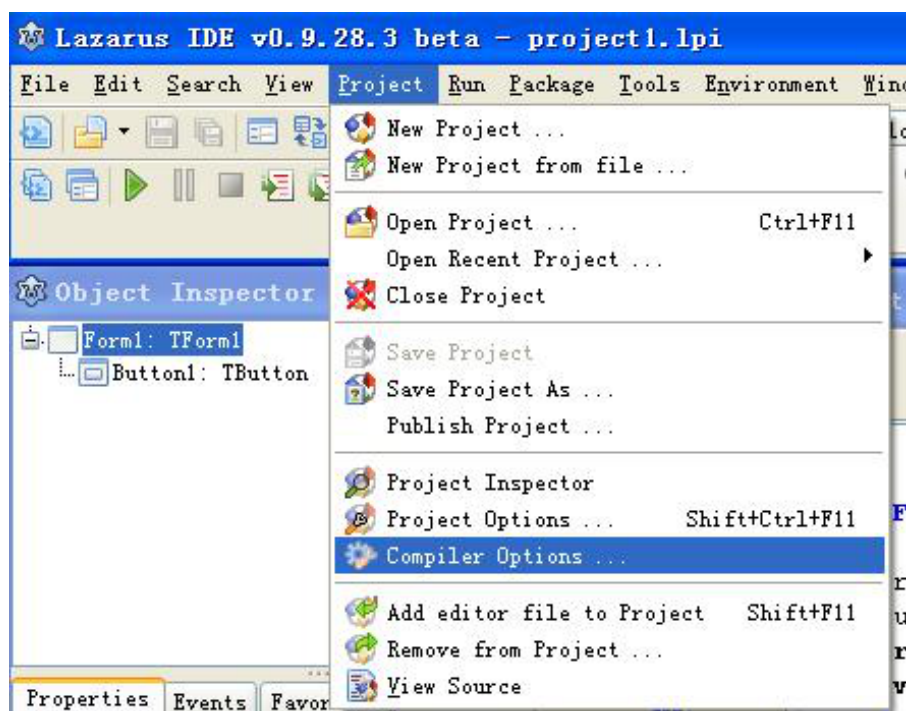


图 2.2-7

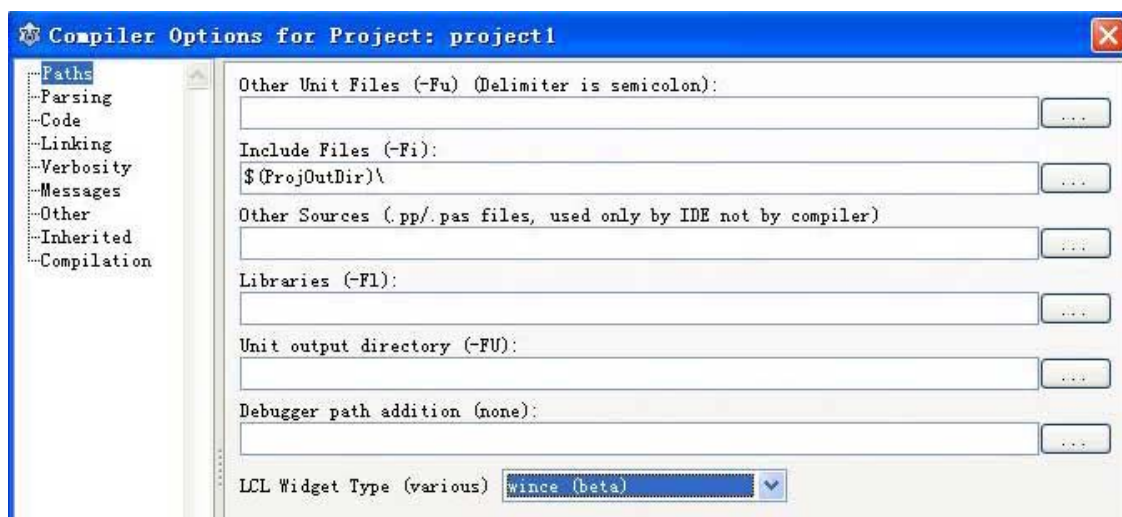


图 2.2-8

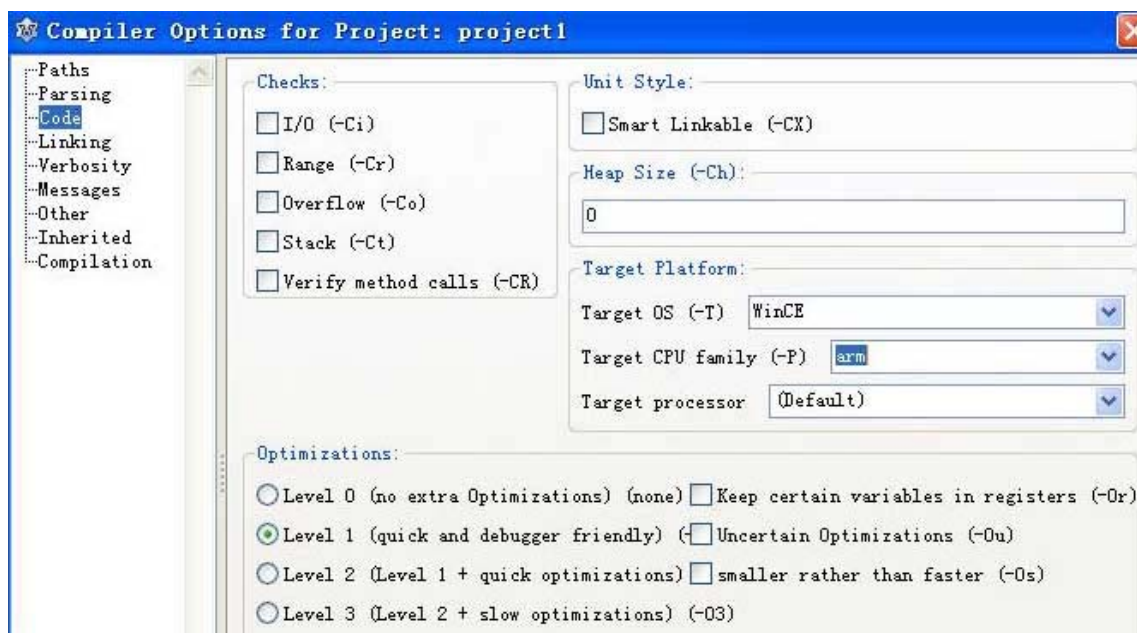


图 2.2-9

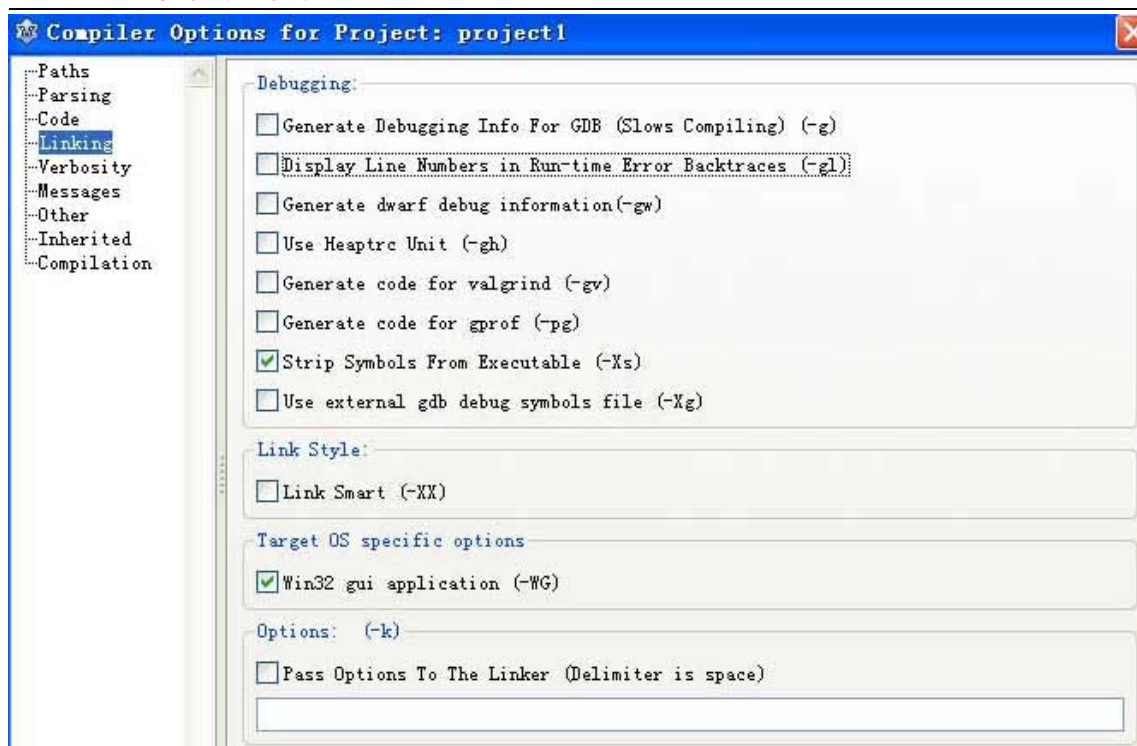


图 2.2-10

图 2.2-7 至图 2.2-10 罗列了编译 WINCE 工程所需要做的工作，下面详细解释：

在图 2.2-7 中，选择的顺序为“Project”-》“Compiler Options”，即为工程的编译参数进行设置；

在图 2.2-8 中，选中左边目录树中的“Paths”，在右下角的“LCL Widget Type”的下拉框中选中“wince beta”，确定程序编译系统使用 WINCE 的库。

在图 2.2-9 中，选中左边目录树中的“Code”，右边的“Target OS”下拉框中选择“WinCE”，“Target CPU family”下拉框中选择“Arm”，确定程序运行的环境为 WINCE 系统。

在图 2.2-10 中，选中左边目录树中的“Linking”，在右边的“Debugging”列表框中只勾选“Strip Symbols From Executable (-Xs)”，这个选项能有效地减少执行文件的体积。例如本程序在在没有勾选这个选项时，执行文件的体积达到 11M 之巨；勾选这个选项后，执行文件的体积缩小到 1.8M，效果相当的明显。

图 2.2-7 至图 2.2-9 是编译 WINCE 程序的关键步骤；图 2.2-10 是编译 WINCE 程序的小窍门。

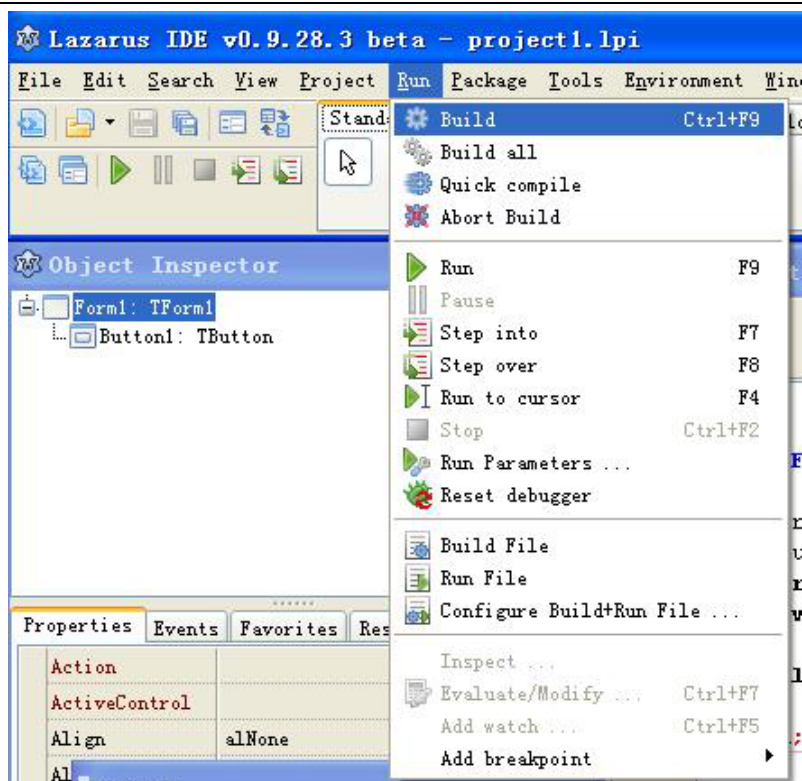


图 2.2-11

经过前面的设置后，接下来的工作就是编译了，如图 2.2-11 所示，在“Run”的下拉菜单里，我们选择“Build”！

接着在 LAZARUS 的 IDE 里弹出图 2.2-12 的画面，这是告诉我们：程序编译通过。于是在程序的保存文件夹里就出现了图 2.2-13 所示的文件，其中红色框框住的就是能在 WINCE 系统里运行的可执行文件。

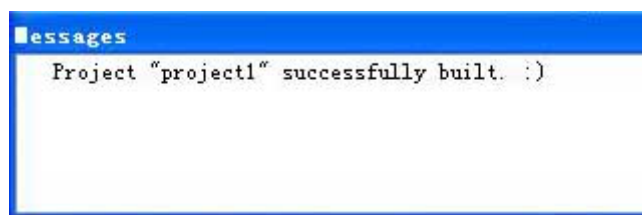


图 2.2-12

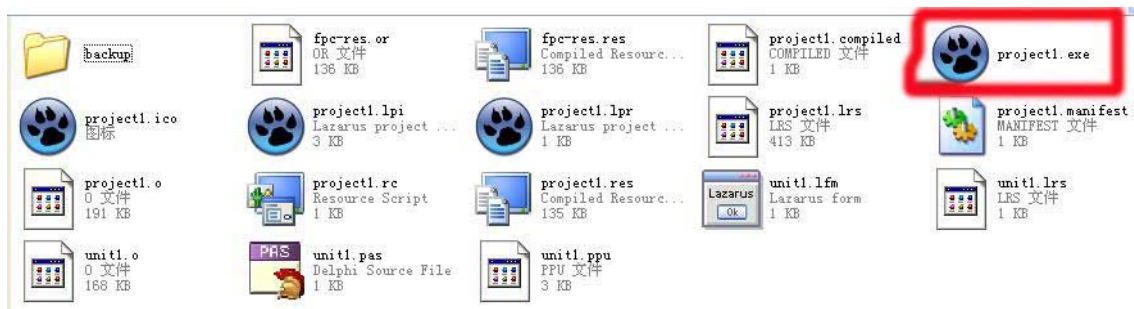


图 2.2-13

在前面的介绍中，特意在 windows 里运行了一次程序代码生成的 windows 程序，这是借助了 LAZARUS 的“一次编写，到处编译”的特性，通过我们熟悉的环境检查程序有没有语法的错误，以及观看程序运行的大概效果。

2.2.3 运行效果

在前面介绍了运行 WINCE 程序的 3 种方法，通过这些方法，可以看到编写的 WINCE 程序的实际效果。

这里采用硬件零成本的方法：采用 WINCE 模拟器。

在互联网上到处可见的 WINCE 模拟器实质是从微软的 VS2005 编程软件那里剥离出来的几个程序文件，其中最关键的是*.bin 的镜像文件，更换不同的镜像文件就能够模拟不同版本的 WINCE 系统。

下图是笔者的 WINCE 模拟器文件夹，批处理文件 Wince50.bat 的内容为：

```
start .\image\DeviceEmulator.exe .\image\Wince5.bin /memsize 128 /sharedfolder  
sd_card /video 480x320x16 /vmname "Wince 模拟器"
```

我们关注的重点是 sharedfolder，它指定存储文件的路径为 sd_card，也就是图 2.2-14 中的 sd_card 文件夹，只要把编译好的程序放到 sd_card 文件夹里，就可以被 WINCE 系统正确识别。图 2.2-15 显示，程序放到了 sd_card 文件夹中。

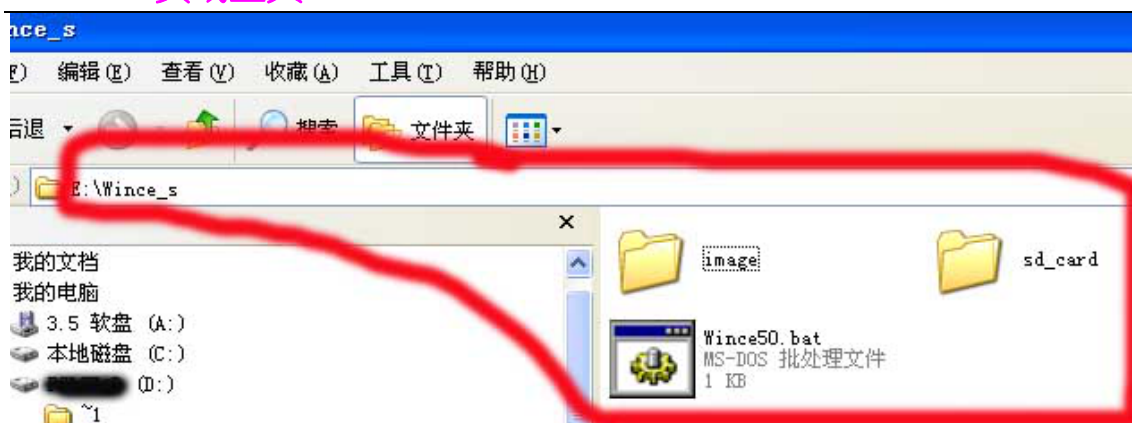


图 2.2-14



图 2.2-15

图 2.2-16 是运行批处理文件 Wince50.bat 后出现的模拟器界面，点击“我的设备”，出现如图 2.2-17 的界面，其中的“SDMMC”的分区比较引人关注，打开该分区，赫然发现我们的程序已经在里面了！图 2.2-18 为证。

图 2.2-19 和图 2.2-20 是程序在 WINCE 模拟器里运行的界面。将图 2.2-20 和图 2.2-6 对比一下，是不是很相似？



图 2.2-16



图 2.2-17

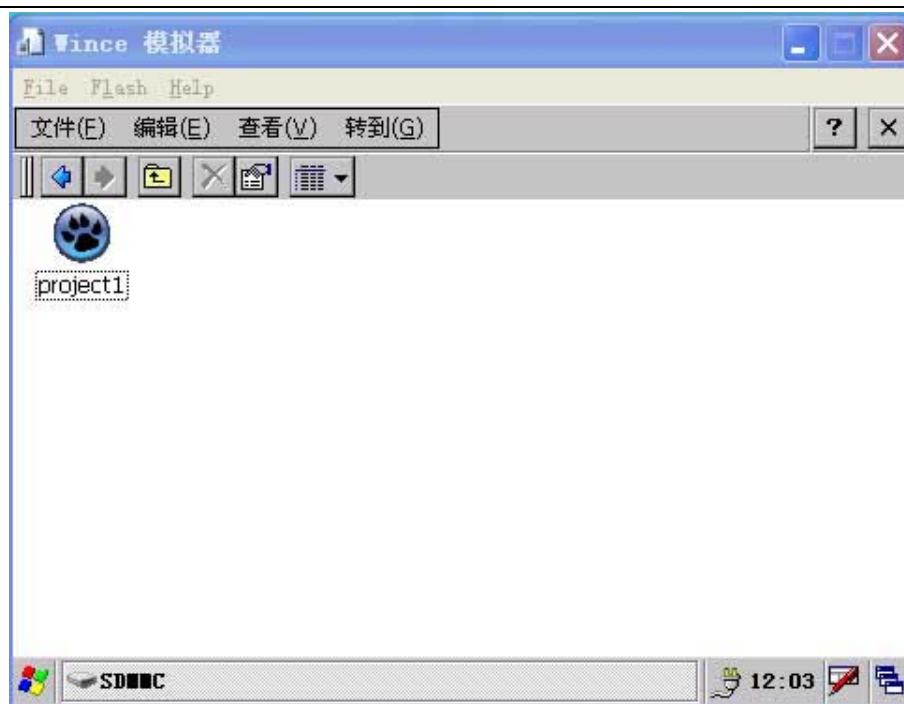


图 2.2-18

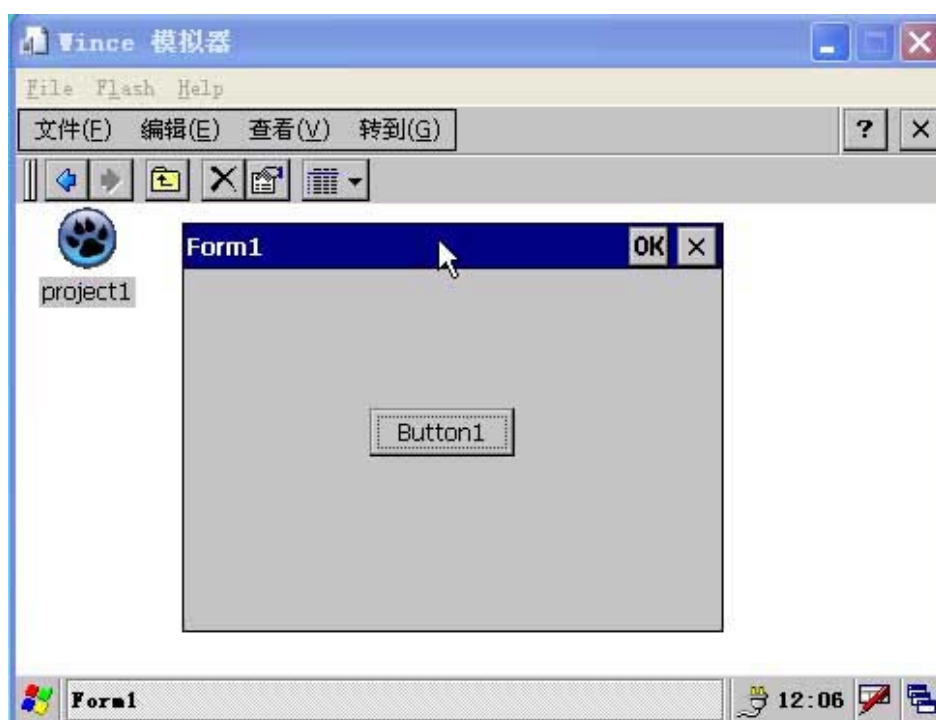


图 2.2-19

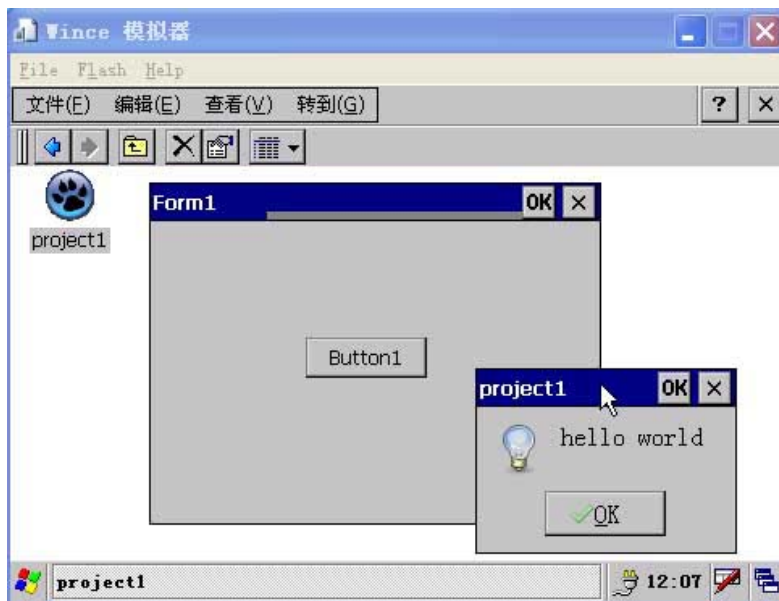


图 2.2-20

2.3 KOL-CE 组件在 WINCE 中的应用

2.3.1 KOL-CE 组件的安装

KOL-CE 组件实际分成 KOL 和 MCK 两组套件。

KOL-CE 可以建立非常精简的 WINCE 图形界面程序(如果项目只包含一个空的视窗的话,编译出来的程序码大约只有40KB 而已)。KOL-CE 安装的步骤如下:

- 1 点选选单项目: Tools -> Configure "Build Lazarus"...
- 2 在 Quick Build Options 分页上,点选 Clean Up + Build all 项目。
- 3 开启 Advanced Build Options 分页,并在 Options 这个对话框里面加入 -dDisableFakeMethods 这个设定字符串。
- 4 点选 Build 按键,重新编译 Lazarus。
- 5 点选选单项目 Components -> Open package file (.lpk),然后寻找到 MCK 套件所在的目录,并选择 MirrorKOLPackage.lpk 这个组件包。
- 6 弹出 MCK 套件的窗口,按下 Install 按键。

- 7 Lazarus 会将 MCK 套件进行编译，然后重新调用 Lazarus IDE。
- 8 Lazarus IDE 重新出现后，KOL 分页就会在元件盘上面出现了。

图 2.2-21 至图 2.2-25 显示了整个安装步骤。

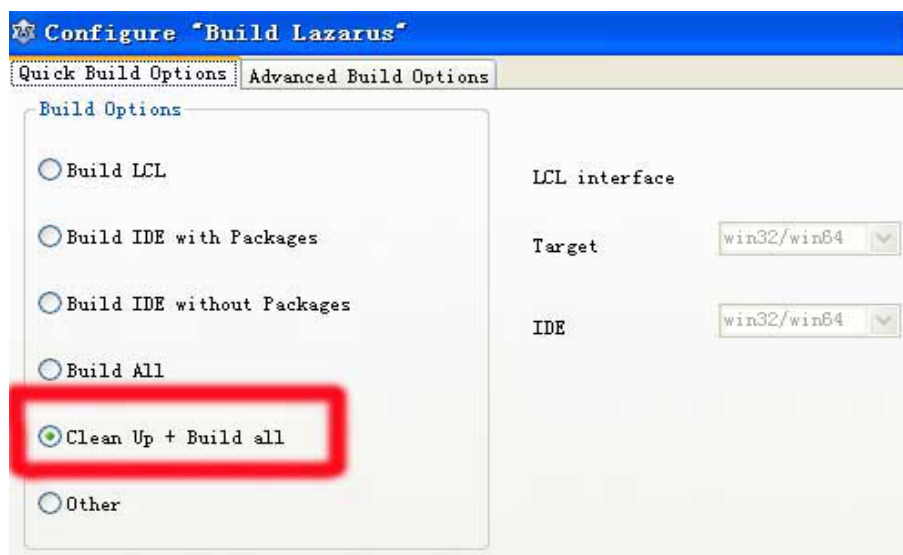


图 2.2-21

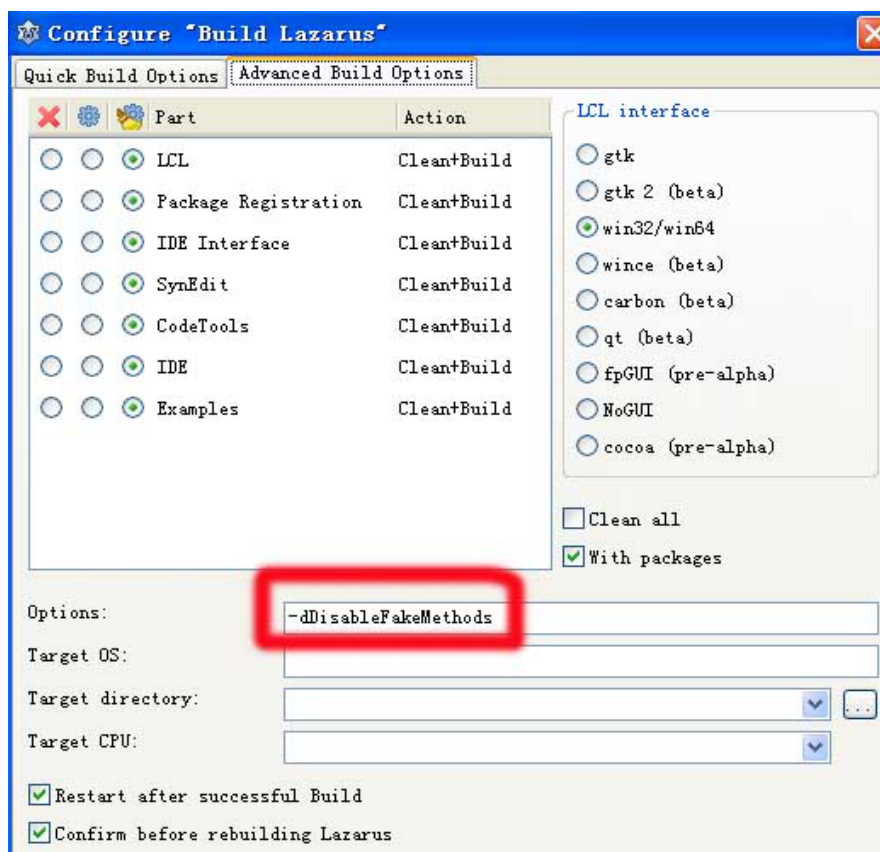


图 2.2-22

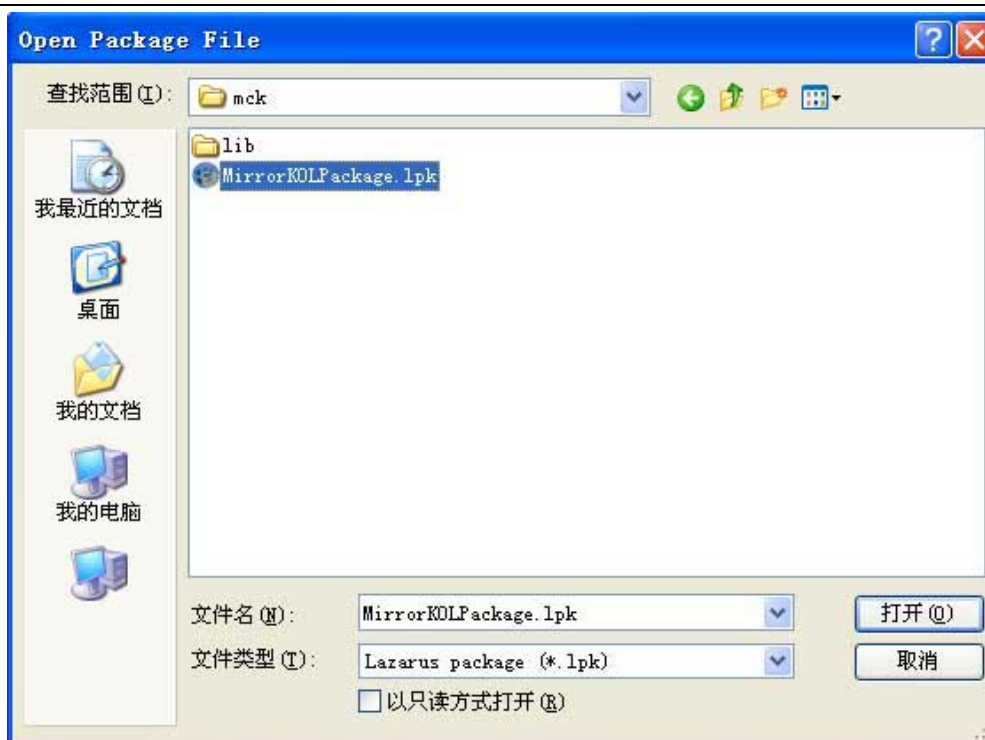


图 2.2-23

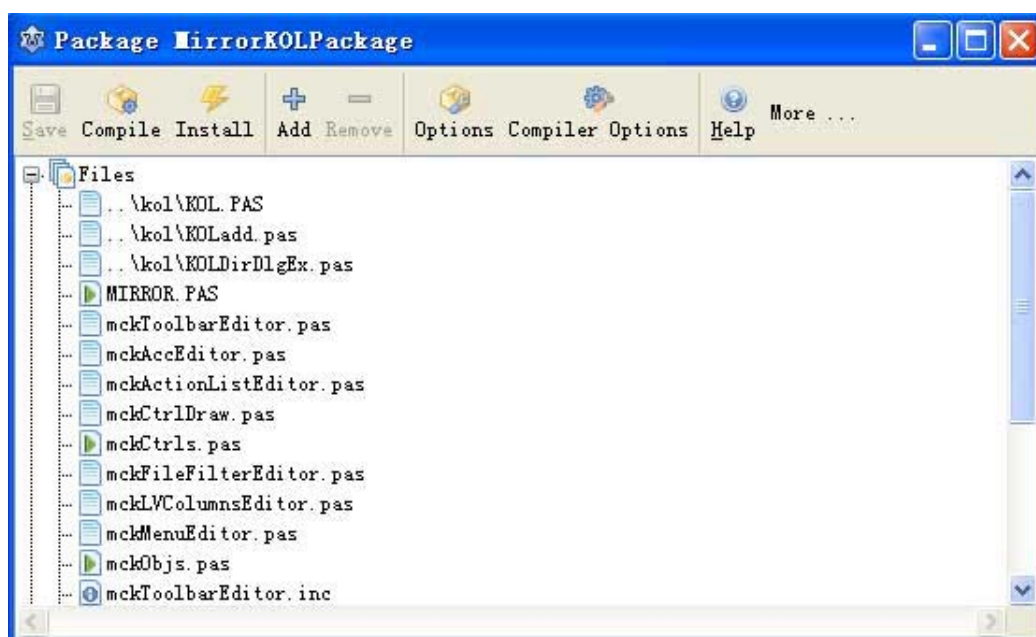


图 2.2-24



2.3.2 使用 KOL-CE 组件创建程序

在 LAZARUS 的 IDE 里点选单项目 Project -> New Project, 接着在弹出的菜单里选择“KOL Toolkit Application”, 如图 2.2-26 所示。

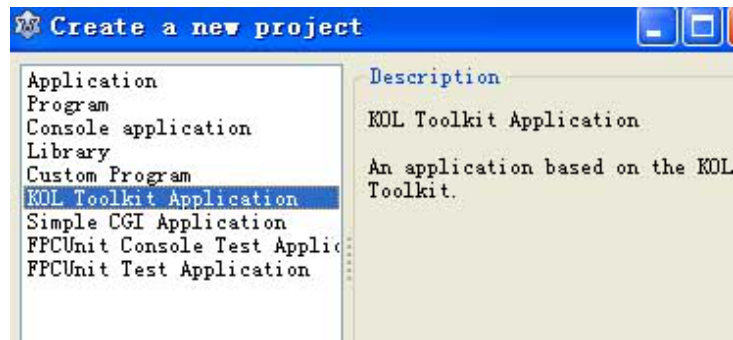


图 2.2-26

在图 2.2-27 里所看到的的就是新创建的 KOL Project, 有两个控件: KOLProject 和 KOLForm, 它们是 KOL Project 自动创建的。

虽然 KOL Project 也可以通过在 LAZARUS 的普通 Project 中添加 KOLProject 和 KOLForm 来创建, 但我们强烈推荐使用直接创建 KOL Project 的方式, 这样可以避免意外的发生。

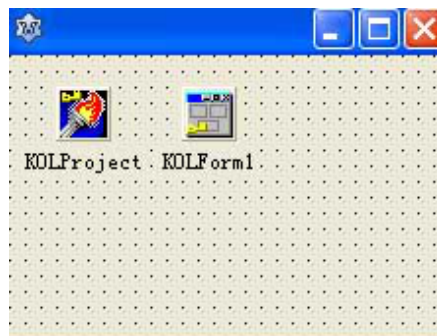


图 2.2-27

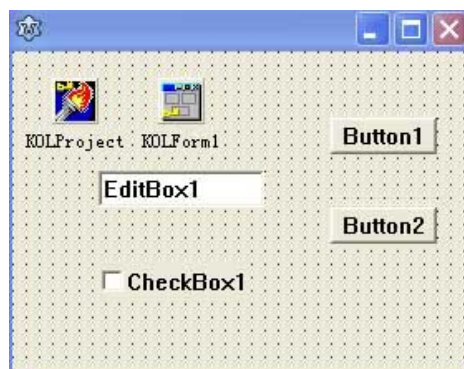


图 2.2-28

在组件栏的 KOL 分页里选择所需要的控件，将控件拖放到 Form 里，摆在合适的位置上，注意：一定要在 KOL 分页里选择控件，因为 KOL 对 LAZARUS 本身的可视化控件的支持有限，可能会产生兼容性的问题。

图 2.2-28 显示了所选取的 KOL 控件，与 LAZARUS 本身的可视化控件效果极其相似。

为了使界面更好看，需要做一些设置，选中界面里的 KOLForm 控件，修改其属性：

borderStyle 选择 FbsDialog，该设置是使程序在 WINCE 里能显示标题栏；

Caption 里填写 KOL test，这是标题栏的内容，可以任意填写。

图 2.2-29 显示了修改的属性。

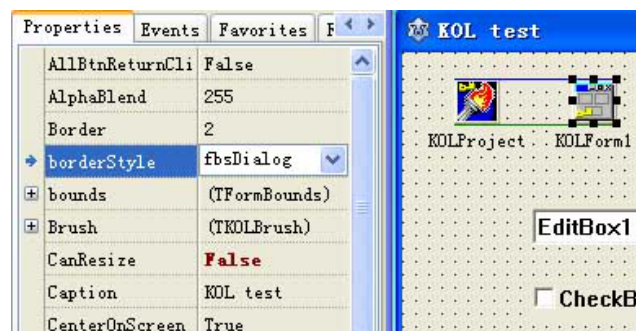


图 2.2-29

在使用可视化设计软件的时候，我们都习惯直接点击控件产生事件，IDE 自动产生事件代码框架，我们只需要往框架里添代码就可以了，这种操作是任何时候都有效的。

在 0.9.28.2 以上版本 LAZARUS 里，KOL-CE 的程序很例外，我们必须将所有控件的所有可能用到的事件先点击了，产生框架（注意：这个 Project 还没有做任何的保存！）。

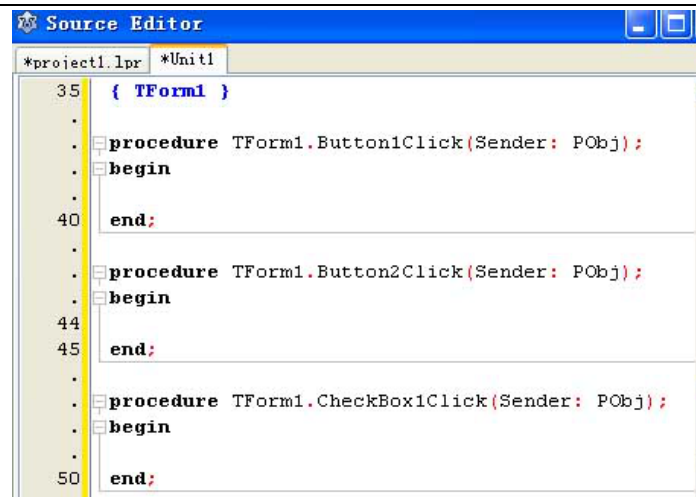


图 2.2-30

在图 2.2-30 中，我们选择了几个常用的事件，框架里是空白的，还没有任何的代码，确认所需要的事件足够了，保存工程！

添加简单的代码，以便于测试本程序的效果，选择 Button1 的双击事件，加入代码：

```
EditBox1.Text:='Hello KOL';
```

如图 2.2-31 所示。

接着按照上一节（2.2.2 编译 LAZARUS 的 WINCE 工程）的方法，生成执行文件，注意：该执行文件大小为 186K，体积轻盈很多啊；再按上一节（2.2.3 运行效果）的方法，在 WINCE 模拟器中运行该执行文件，效果如图 2.2-32 和图 2.2-33 所示。

总结：

- ① 利用 KOL 组件生成的程序，体积很小，尤其适合于硬件存储空间比较紧凑的场合；
- ② 创建 KOL 程序的方法比较特殊，在使用时要注意；
- ③ KOL 程序的 Form 不直接支持 Lazarus 的可视化控件，尽量全部使用 KOL 组件自身的可视化控件。
- ④ KOL 组件在 0.9.28.2 以上版本的 LAZARUS 里表现并不好，这点需要牢记。最理想的搭配是：LAZARUS 0.9.28.2 + KOL-CE 2.80.3。见图 2.2-34。

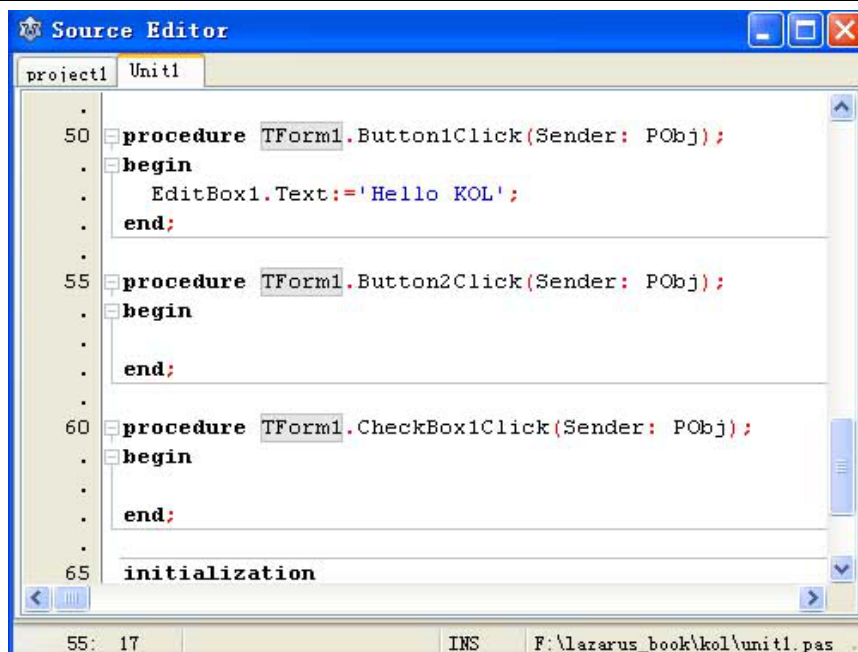


图 2.2-31



图 2.2-32



图 2.2-33

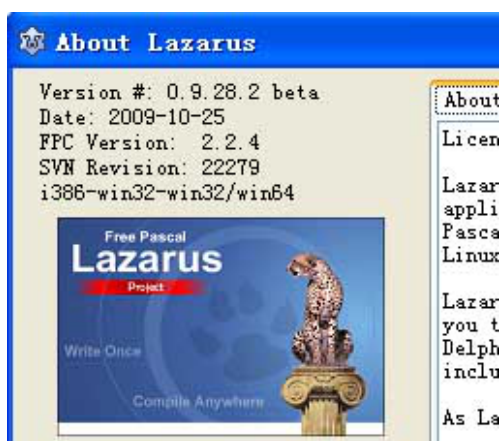


图 2.2-34

2.4 WINCE 程序的调试技术

在进行 WINCE 程序的调试前，我们需要进行如下的工作：

1、下载相关的软件

其中一个需要下载的是调试软件：

<ftp://ftp.freepascal.org/pub/fpc/contrib/cross/gdb-6.4-win32-arm-wince.zip>

该压缩文件解压后有 3 个文件，如下图：

名称 ▲	大小	类型
arm-wince-pe-stub.exe	11 KB	应用程序
gdb.exe	2,051 KB	应用程序
readme.txt	2 KB	文本文档

图 2.2-35

为了方便使用，将 gdb.exe 重新命名为： gdb_ce.exe ，完成后如下图：

名称 ▲	大小	类型
arm-wince-pe-stub.exe	11 KB	应用程序
gdb_ce.exe	2,051 KB	应用程序
readme.txt	2 KB	文本文档

图 2.2-36

接下来，需要给上面的文件找个家了，笔者在 ..\lazarus\mingw\bin 里安放了上面的文件。当然，你也可以将它们放在任何的文件夹里，不过在设置工程的属性时则需要做相应的改动。

在 ..\lazarus\mingw\bin 里已经有了一个 gdb.exe 程序，该程序是用于调试 windows 应用软件的，笔者将它重新命名为： gdb_win.exe，和上面的程序做个区别：

E:\lazarus\mingw\bin	
名称 ▲	
arm-wince-pe-stub.exe	
gdb_win.exe	
readme.txt	
gdb_ce.exe	

图 2.2-37

另外一个需要下载的是联机软件：

<http://www.microsoft.com/windowsmobile/en-us/help/synchronize/activesync45.mspx>

这个是专门用于 WINCE 设备和计算机联机的软件，通过该软件，可以将 windows 环境下的文件和 WINCE 设备里的文件保持同步状态。该软件直接安装使用。

然后到微软的网站上下载单机版的 WINCE 模拟器软件，下载地址：

3.0 版本：

<http://www.microsoft.com/downloads/zh-cn/details.aspx?FamilyID=A6F6ADAF-12E3-4B2F-A394-356E2C2FB114>

2.0 版本:

<http://www.microsoft.com/downloads/zh-cn/details.aspx?FamilyID=dd567053-f231-4a64-a648-fea5e7061303>

1.0 版本:

<http://www.microsoft.com/en-us/download/details.aspx?id=20259>

建议: 将 3 个版本的模拟器软件都下载, 先安装 1.0 版本, 因为只有此版本才有 WINCE 的镜像文件, 然后顺序安装 2.0 版本, 3.0 版本。

安装完毕后的文件夹如下图:

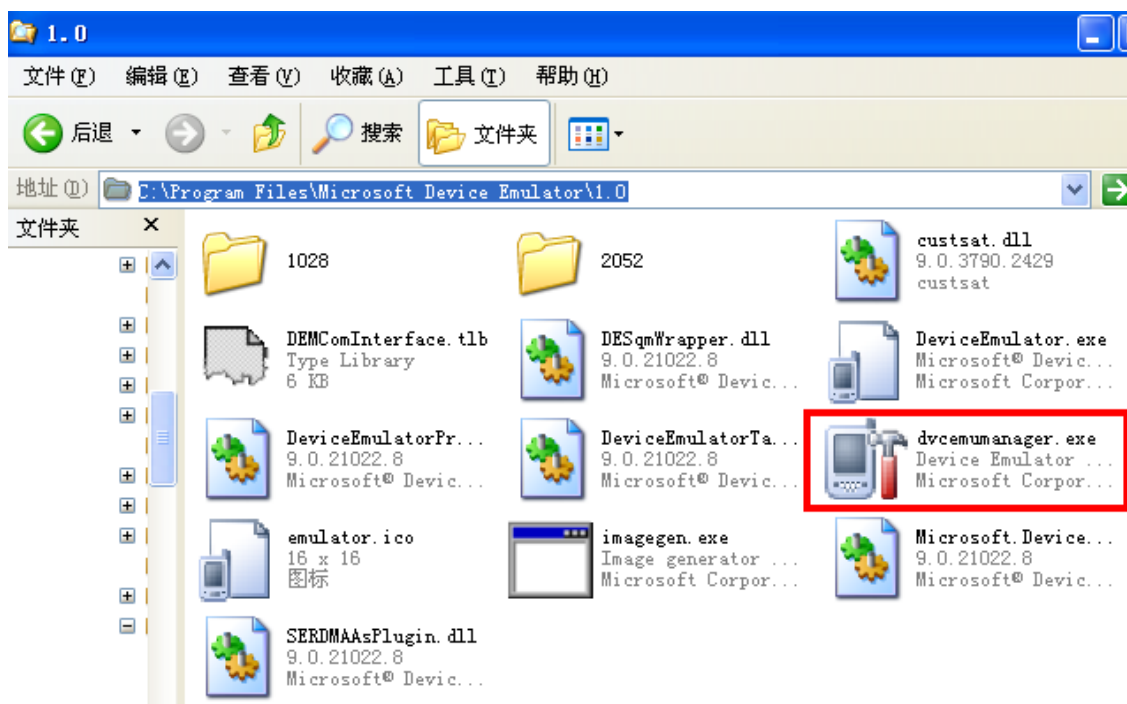


图 2.2-38

上图中红色框中的程序是关键! 通过它才能进行调试程序。

2、设置工程的属性

在 LAZARUS 的 IDE 菜单选 “Environment” —》“Options”; 接着在弹出的界面里选择 “Debugger”, 并在图 2.2-39 所示的红色框中选择前面的程序: gdb_ce.exe, 最后点 “OK” 结束设置。

经过此步骤, 使得 IDE 具有调试 WINCE 程序的功能, 可以使用 IDE 的 “Run” (按 F9 键)

功能来运行 WINCE 程序。

注意：没有经过这里的设置，直接使用“Run”（按 F9 键）功能是不成功的，IDE 弹出错误提示，如图 2.2-40 所示。

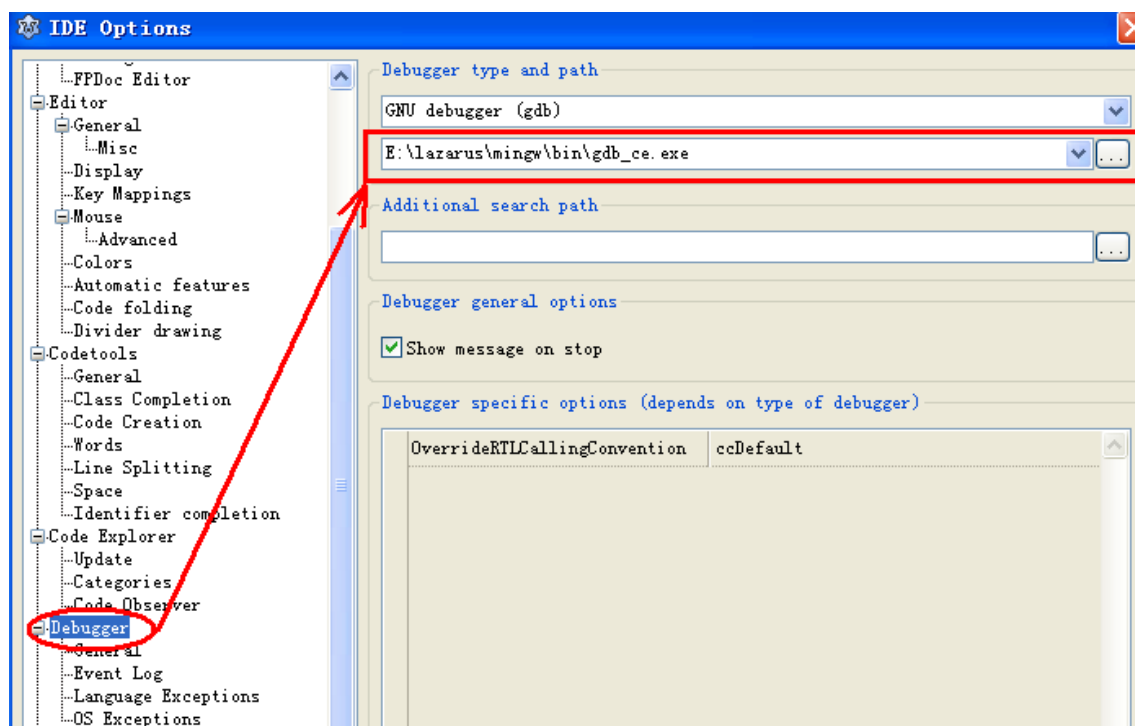


图 2.2-39

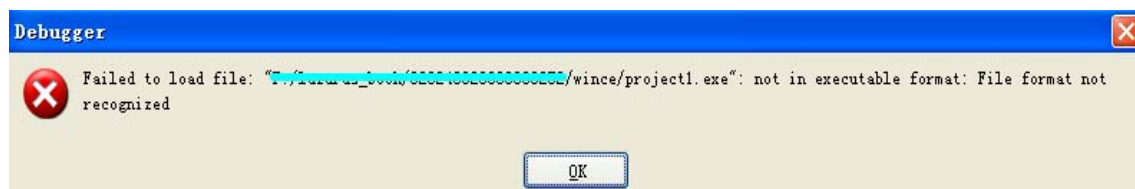


图 2.2-40

还有一个重要的属性需要设置，在 LAZARUS 的 IDE 菜单选“Environment”—“Options”；接着在弹出的界面里选择“Linking”，并勾选图 2.2-41 所示的红色选择项，此处的设置是生成供 GDB 使用的调试信息。

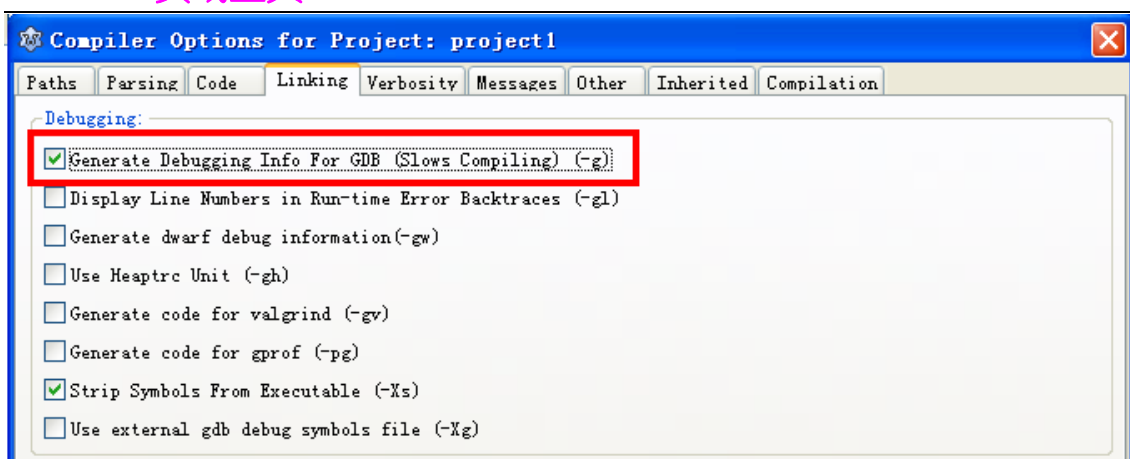


图 2.2-41

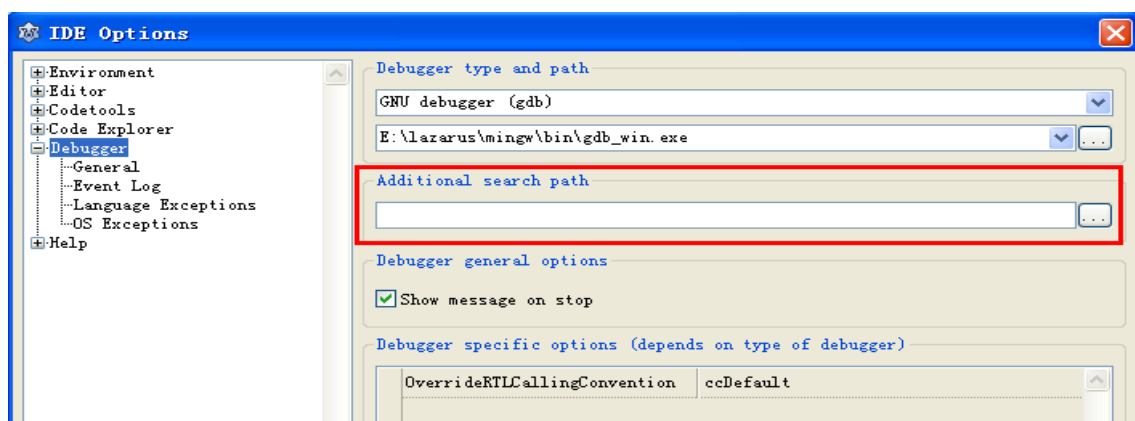


图 2.2-42

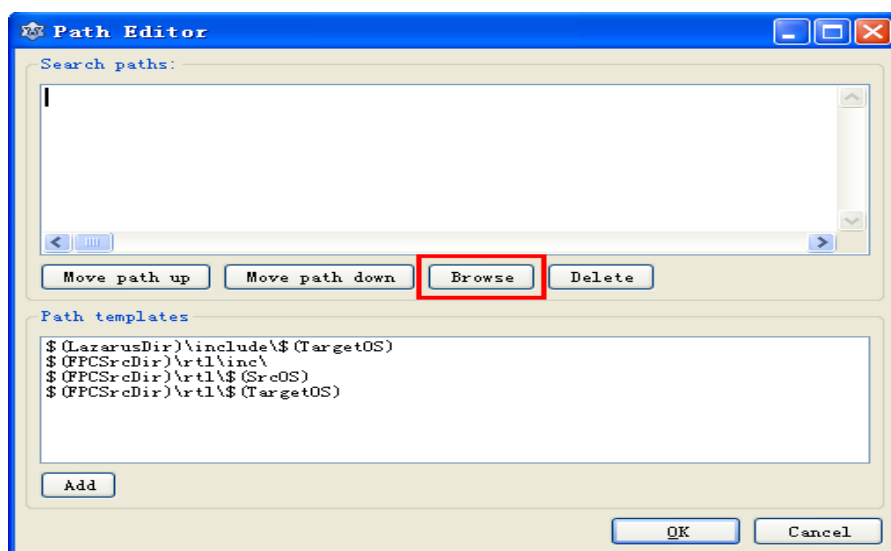


图 2.2-43



图 2.2-44

如果需要单步调试程序，则还需要进行图 2.2-22 至图 2.2-44 的设置，使得 GDB 能够找到调试的源文件的路径，从而进入正常的单步调试。

2.4.1 使用 WINCE 模拟器的调试技术

为了能在 WINCE 模拟器里进行程序的调试，还需要对模拟器做一些处理。

必须在模拟器和 PC 之间建立程序信息交流的桥梁，才可以进行调试。

1、建立新连接

按照前面的方法，启动模拟器，出现如图 2.2-45 所示的界面，双击左下角的启动按钮，选择“设置”—》“网络和拨号连接”。



图 2.2-45



图 2.2-46

图 2.2-47 出现“新建连接”的图标，点击该图标，进行相关的设置，具体的设置内容如图 2.2-48 和图 2.2-49 所示，新建连接的名称为“我的连接”。

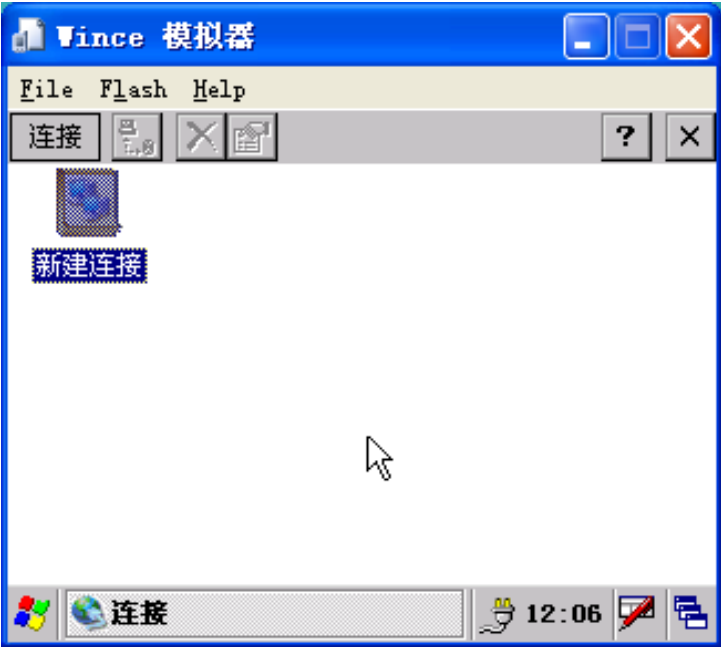


图 2.2-47



图 2.2-48

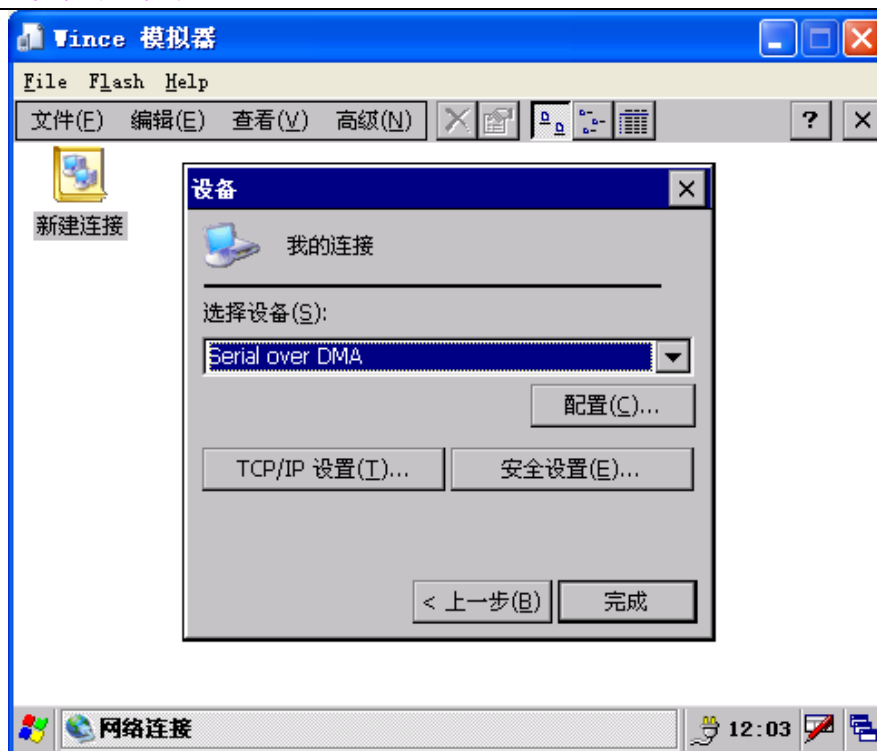


图 2.2-49

2、设置模拟器的连接方式

点击模拟器左下角的“开始”按钮（4色小旗标志），选择“设置”—“控制面板”，出现了图 2.2-50 的界面，在这个界面里，点击“PC 连接”的图标，出现图 2.2-51 的界面，点击“更改连接”，出现图 2.2-52 的界面，在下拉的列表里，选择前面刚建立的“我的连接”。

设置完毕后，模拟器不要关闭，保持运行状态。



图 2.2-50



图 2.2-51



图 2.2-52

3、设置同步软件

在 Windows 的状态栏右下角，可以看到同步软件图标是灰色的，如图 2.2-53，表示当前没有设备连接到计算机；点击该图标，弹出图 2.2-54 的界面，选择“连接设置”。

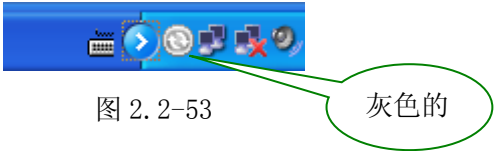


图 2.2-53

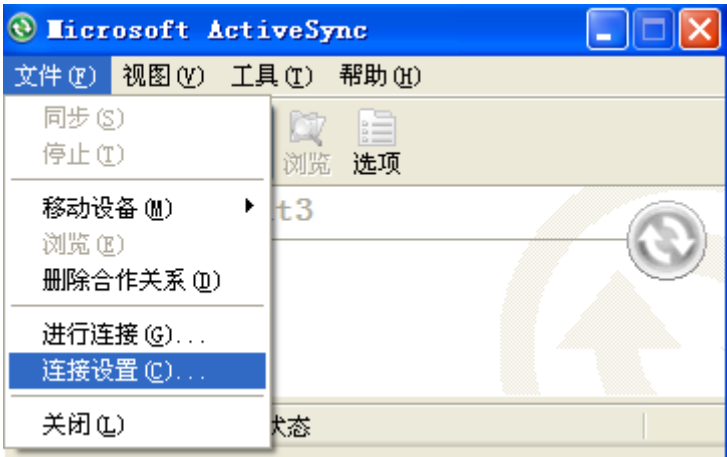


图 2.2-54

设置所有的参数如图 2.2-55，然后点“确定”按钮。

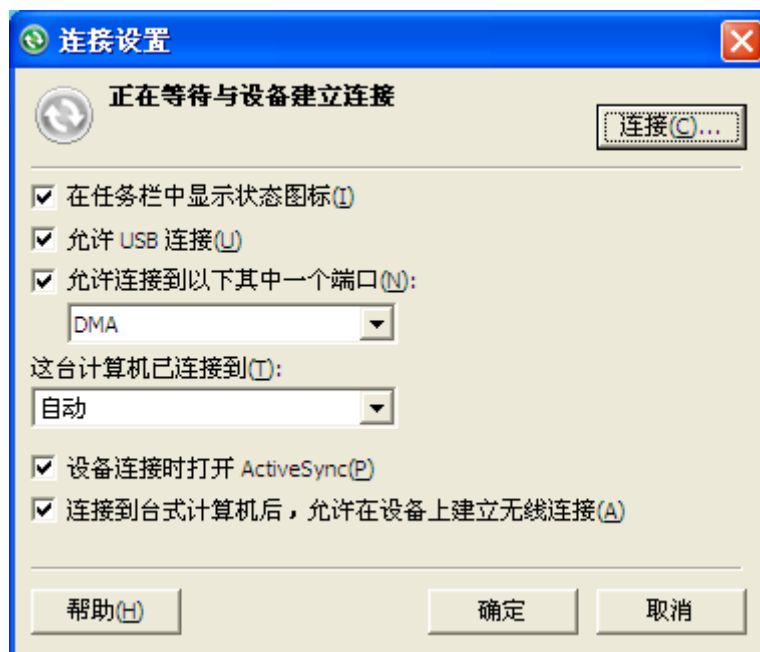


图 2.2-55

4、建立连接的通道

在前面的图 2.2-40 中，红色框住了一个重要的文件，文件名为：dvcemumanager.exe，单独抽取出来后，重新命名为：设备仿真器管理器.exe，该程序运行后如图 2.2-56。

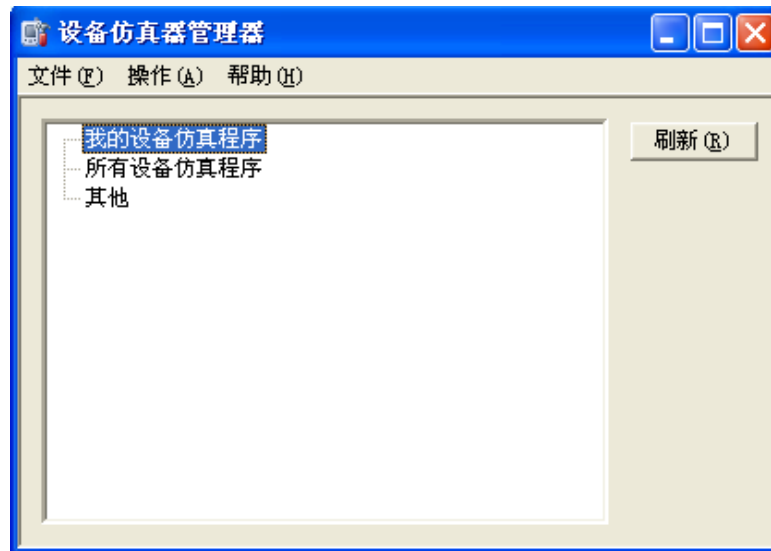


图 2.2-56

如图 2.2-57，在“设备仿真器管理器”里，点“刷新”按钮，带绿色三角符号的“Wince 模拟器”的程序标志出现在列表里了。在图 2.2-58 中，选中“Wince 模拟器”，然后点鼠标的右键，在弹出的菜单里点击“插入底座”，计算机的屏幕立即弹出图 2.2-59 的界面，我们只需要在弹出的界面里点“下一步”，从图 2.2-60 至图 2.2-61，直到出现“完成”按钮，点击该按钮，出现图 2.2-62 的界面，表示连接成功了。



图 2.2-57

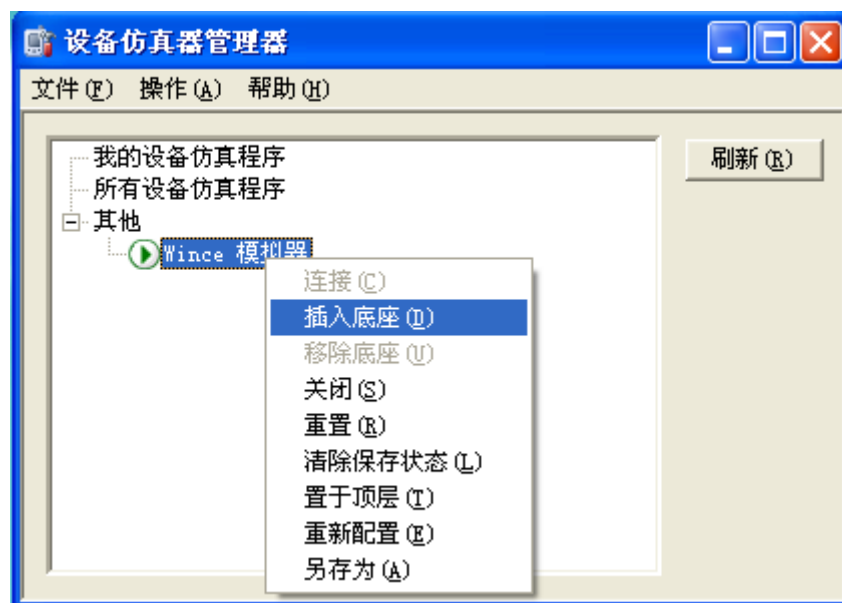


图 2.2-58

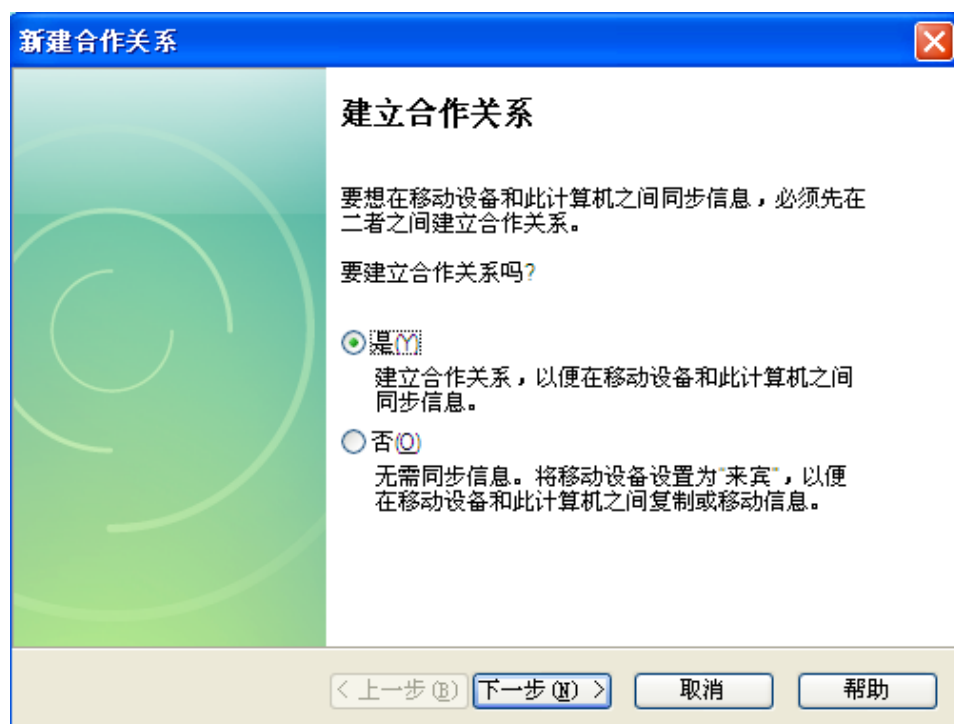


图 2.2-59

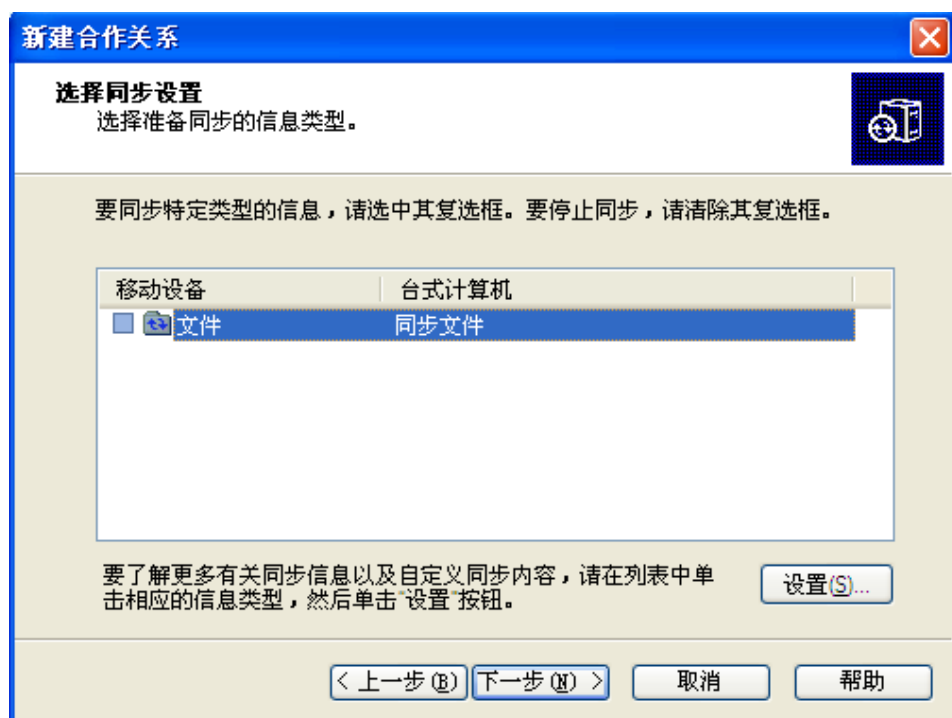


图 2.2-60

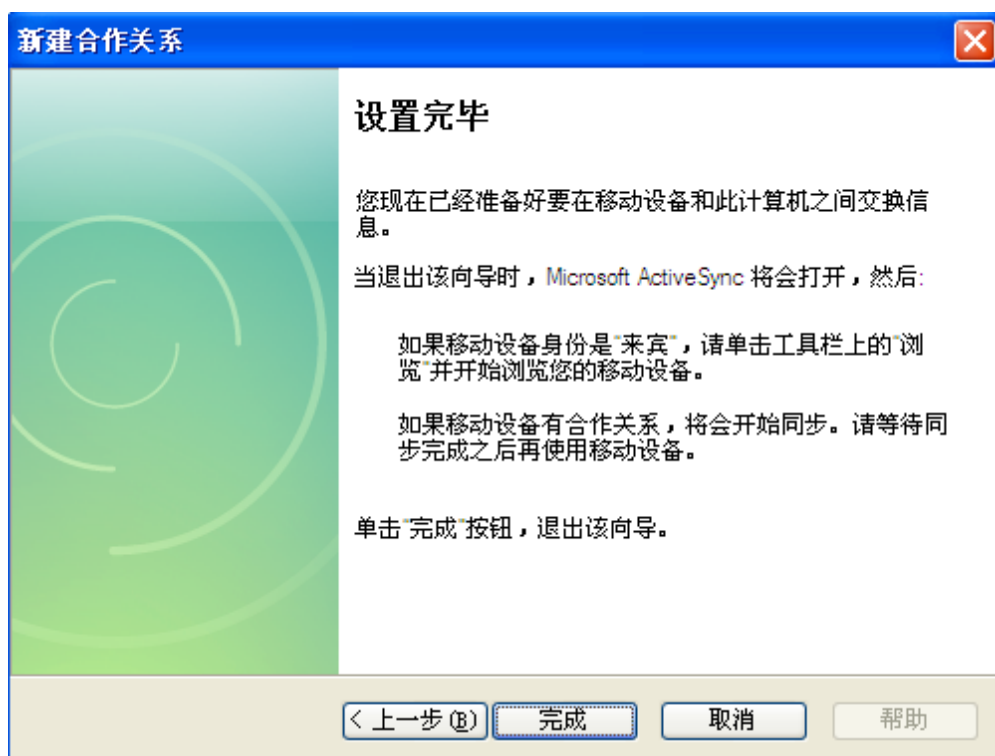


图 2.2-61

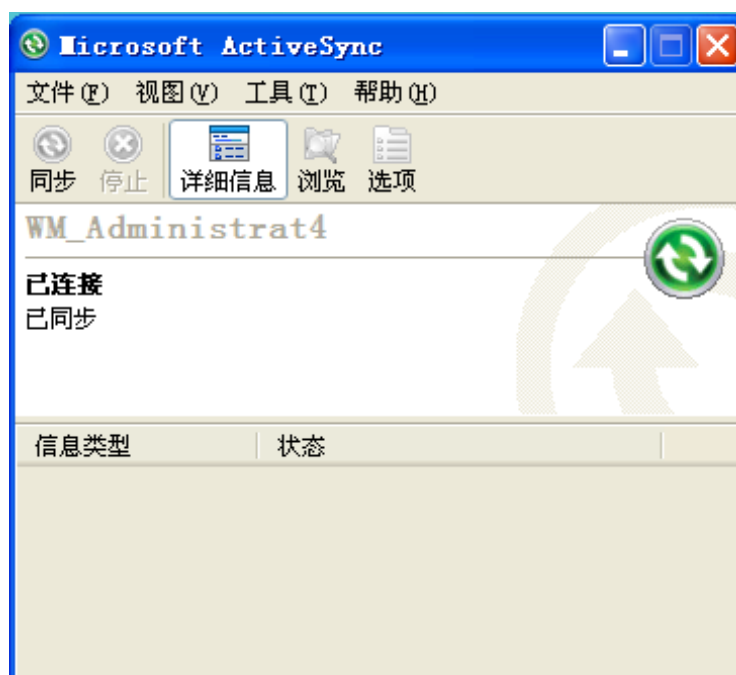


图 2.2-62

在图 2.2-63 里，红色框住的同步图标是绿色的，表示当前同步成功，需要时刻注意该同步图标，一旦变成灰色，则需要重新进行同步，以保证程序调试的正常进行。（即重新做一遍新连接的建立，从图 2.2-45 的步骤开始，至图 2.2-58 的步骤结束，这些设置的结果是无法保存的，只要模拟器一关闭，所有设置全部丢失。）



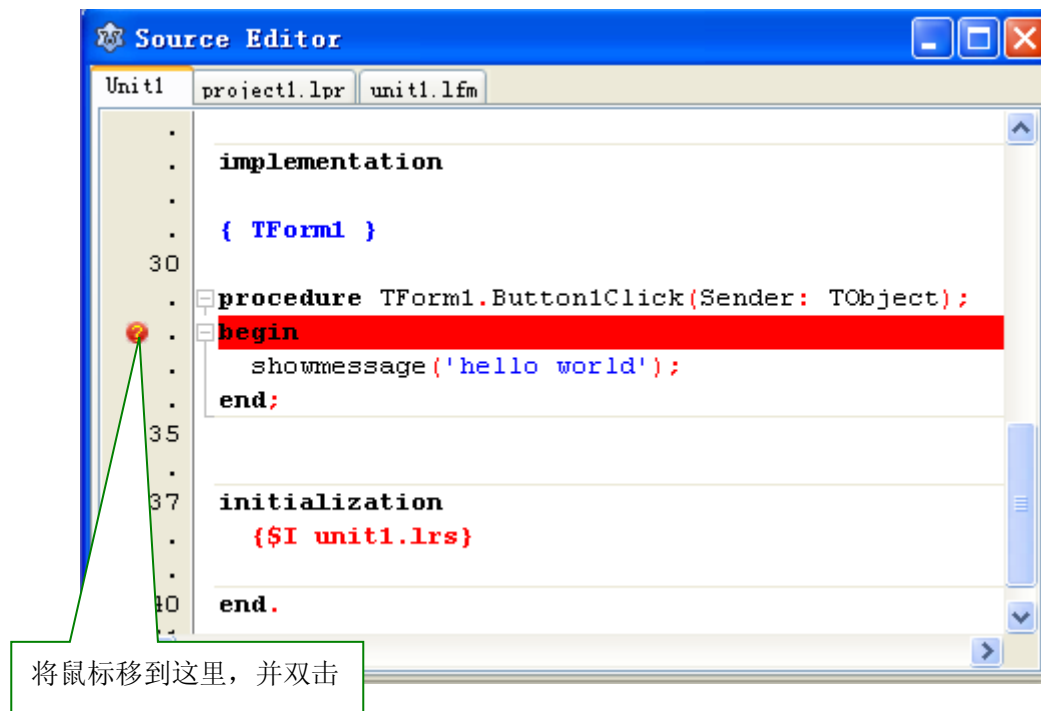
图 2.2-63

需要严重关注

5、调试代码

经过前面的处理，调试环境已经建立起来，下面要进行代码的调试了。

我们继续使用前面章节 2.2.1 的代码进行演示，在 LAZARUS 的工程编辑框里，光标移到第 32 行，并设置断点，然后在 IDE 下拉菜单里选择“Run”-->“Run to cursor”，或者直接按 F4 键，如图 2.2-64。



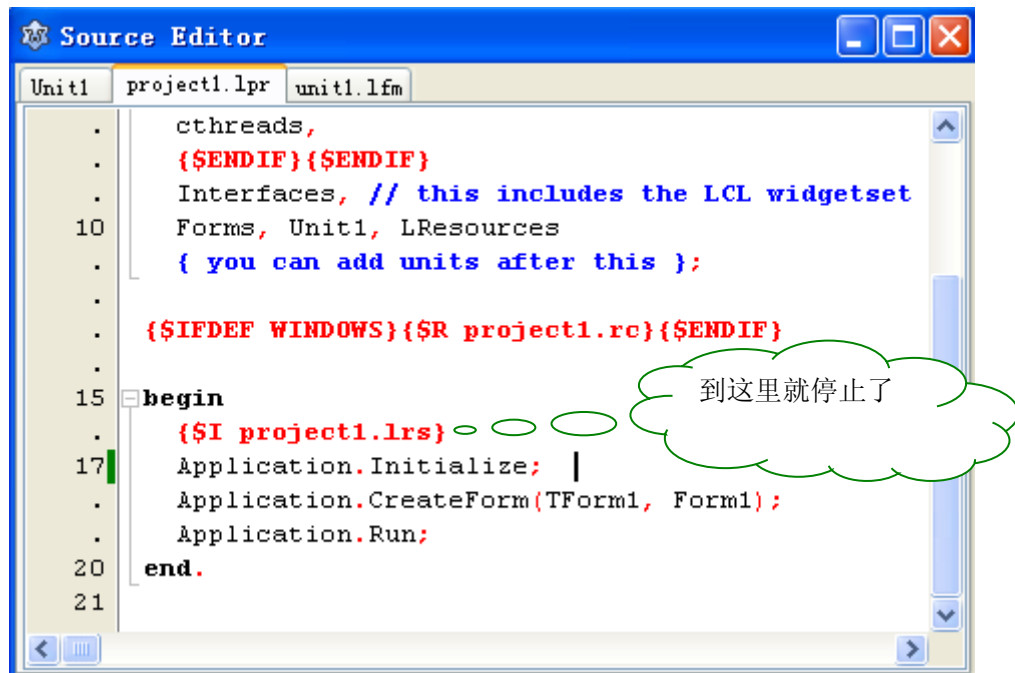


图 2.2-65

第一次单步调试的时候，Windows 会弹出如图 2.2-66 的警报：

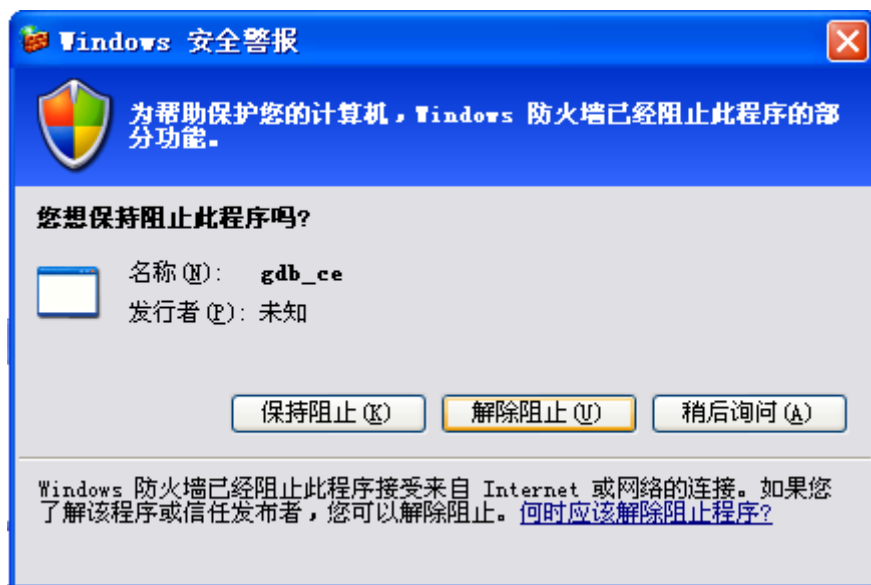


图 2.2-66

点选“解除阻止”就可以了。

在警报提示消失后，LAZARUS 的 IDE 在一段时间内，没有任何的动静，这个时候，切记你要有耐心，实在无聊，可以闭上眼睛，休息一会儿。。。。。

其实 LAZARUS 的 IDE 并没有偷懒，它在后台和模拟器交互信息呢。就在大家放松警惕的时候，LAZARUS 的 IDE 突然跳出图 2.2-67 的界面，同时模拟器的界面也变成图 2.2-68 所示。

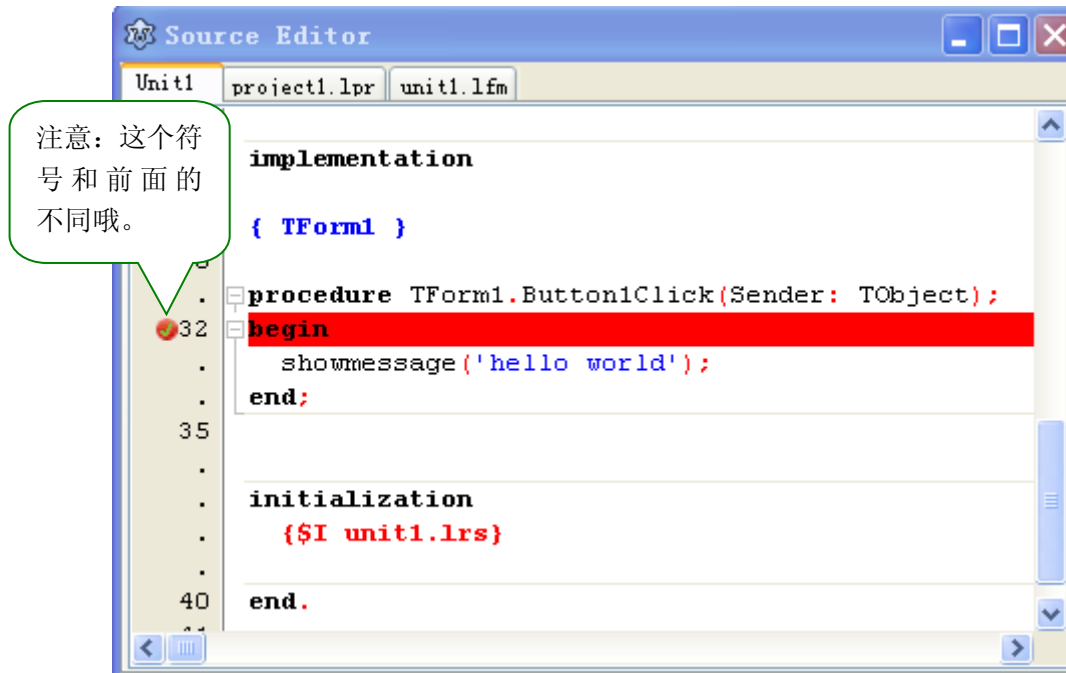


图 2.2-67

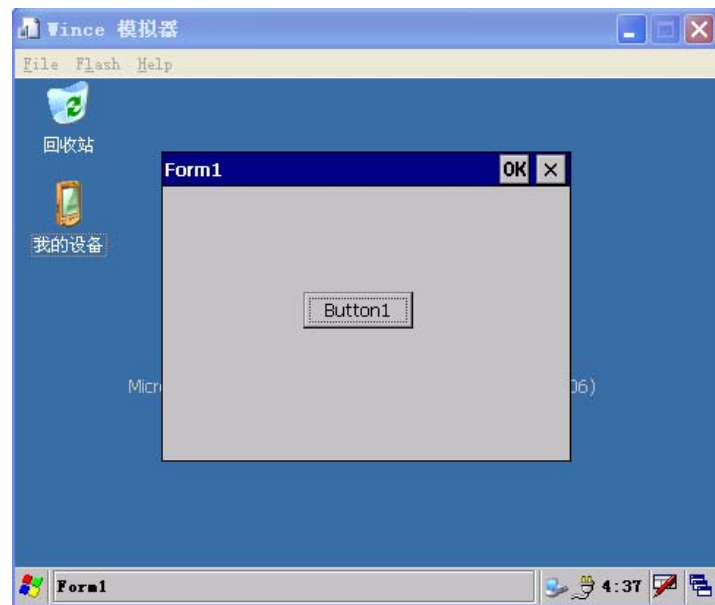


图 2.2-68

从上面的图可以看到程序已经跳到断点处了，等待下一步的操作。转到在模拟器的界面上，

点击“Button1”按钮，接着切换到 LAZARUS 的 IDE 界面，看到新变化了，见图 2.2-69。

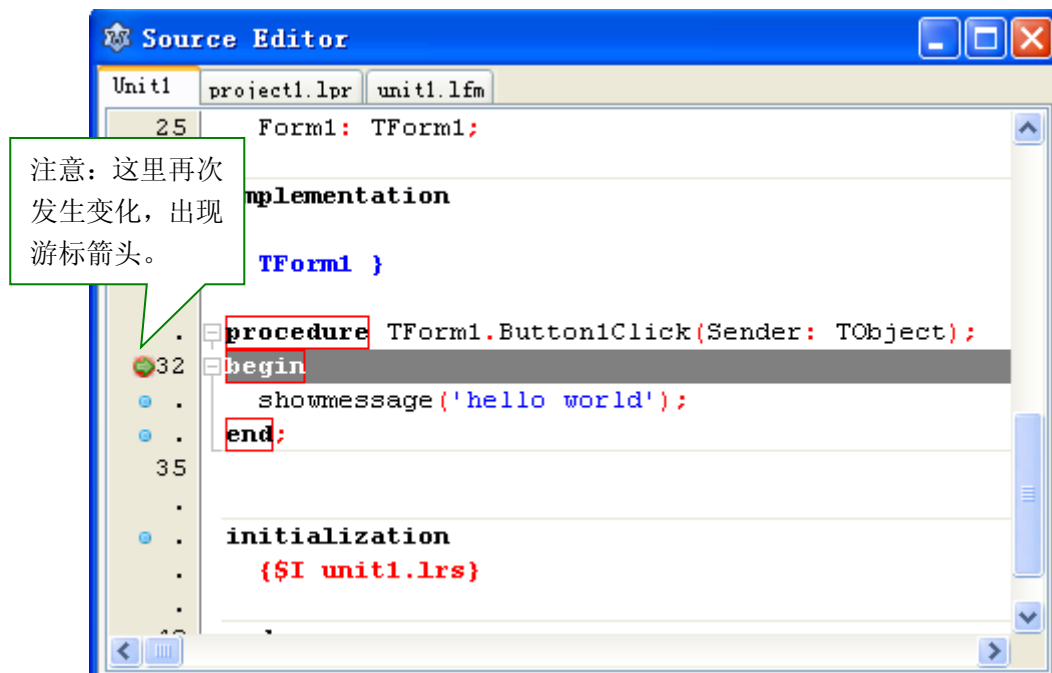


图 2.2-69

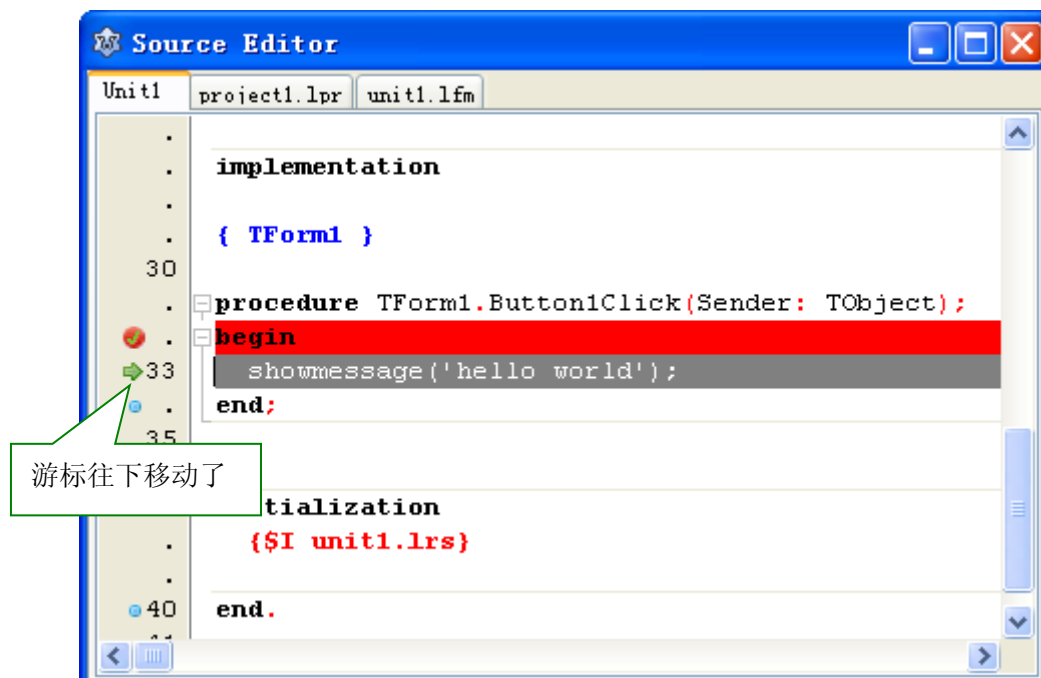


图 2.2-70

在 LAZARUS 的 IDE 里，按“F8”键单步跟踪，见图 2.2-70，再按一次“F8”键，接着在

图 2.2-71 显示了随后的变化，模拟器的界面亦同步变化位图 2.2-71。

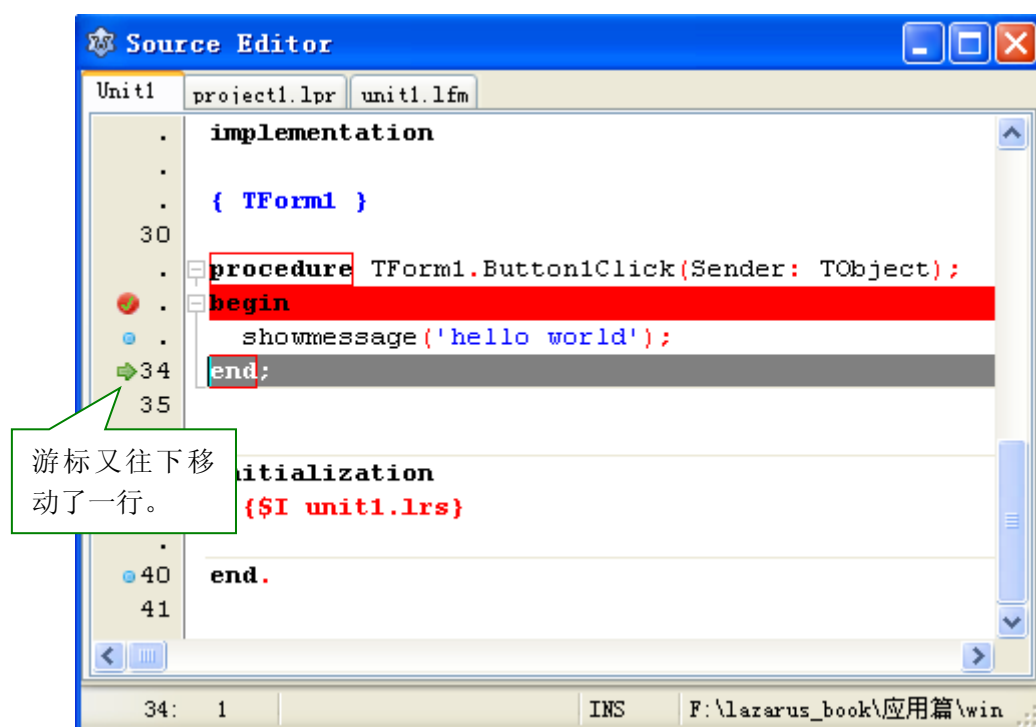


图 2.2-71

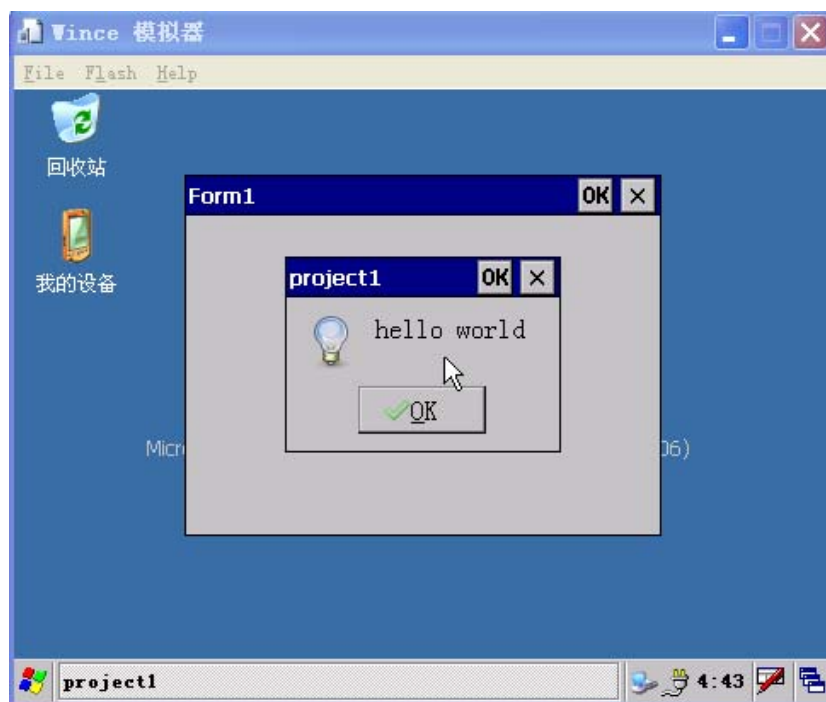


图 2.2-72

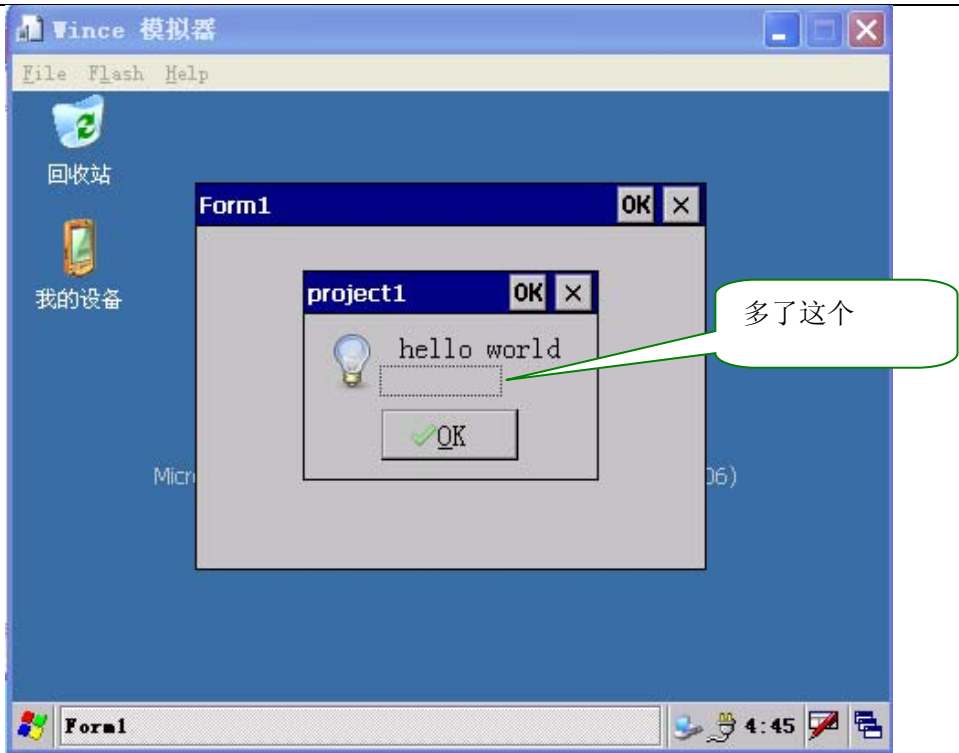
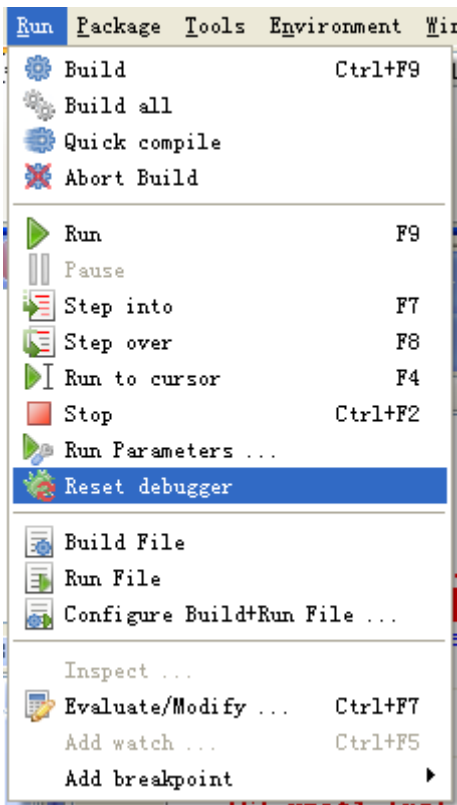


图 2.2-73



在图 2.2-72 中，点模拟器界面上的“OK”按钮，接下来的变化如图 2.2-73，而且 Windows 的焦点自动跳回到图 2.2-71 所示的地方。

前面只是演示简单的调试过程，根据实际的需要，还可以使用诸如添加断点、添加监测变量等常规的调试手段。

如果我们想结束或者中断调试，有什么办法呢？除了直接关闭模拟器和 LAZARUS 这种重口味的办法，我们可以在 IDE 里进行处理，如图 2.2-74：

- ① 在下拉菜单选择：“Run” --> “Stop”；
- ② 在下拉菜单选择：“Run” --> “Reset debugger”。

这两种方法不用重新设置模拟器，但都会引起模拟器出现图 2.2-75 所示的报警，该报警不影响使用。

图 2.2-74

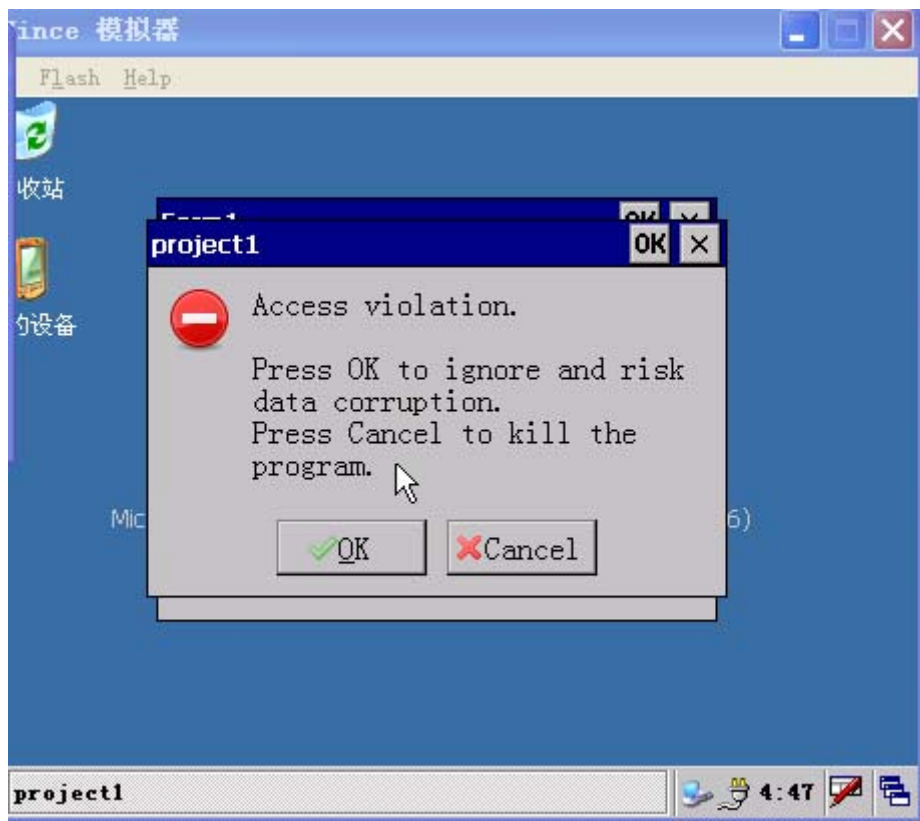


图 2.2-75

6、注意的事项

笔者在使用本调试方法的过程中发现，在模拟器运行通过的程序，在真实机器上未必能跑起来。故此，当程序交付客户之前，无论如何，一定要经过真实机器测试。

2.4.2 使用真实机器的调试技术

相对于模拟器的调试，真实机器的优势在于不用进行任何的设置，直接将机器的 USB 接口和计算机的 USB 接口连接在一起就可以了。如果中途连接失效，则重新拔插一下连接的 USB 线即可。

注意：必须先运行 Windows 的同步软件，然后再插入 USB 连接线。

使用真实机器进行调试，前面 2.4.1 中的 1~4 步骤省略，其余的步骤相同。

在真实机器里调试，速度比在模拟器中调试要来得快，而且更接近实际的使用，故此，在条件允许的情况下，推荐使用。

2.5 WINCE 编程点滴

1、必须清楚程序运行的环境

由于 WINCE 是可定制的操作系统，故此，每种机器上的 WINCE 都可能存在不同，极有可能导致 LAZARUS 编译出来的程序在某个机器能运行，到了其它机器却无法运行。没有做过 WINCE 定制的人可能以为是程序问题，其实情况不如此。

要避免这类现象，要么做好程序的全面测试；要么自己定制 WINCE，安装适合程序运行的组件。

2、注意编码的问题

WINCE 是 unicode 编码的操作系统，故此，所有的字符型都必须先转化为 Pwidechar 类型，然后再处理。

在 LAZARUS 的官方网站上有编码问题如何处理的详细讲解，感兴趣的可以到下面的网址：

http://wiki.freepascal.org/LCL_Unicode_Support

3、注意路径的问题

如果要在 WINCE 里处理文件或者目录，一定要使用绝对路径。

第 1 章

1. 文字图案生成器

1.1 原理

这款小工具改自 LAZARUS 中文社区（www.fpcn.com）中的一个程序“文字生成字符图案的小程序”，原作者为猫者。

我们在 BBS 上经常看到由某个特殊字符构成的有趣图形，如：

```

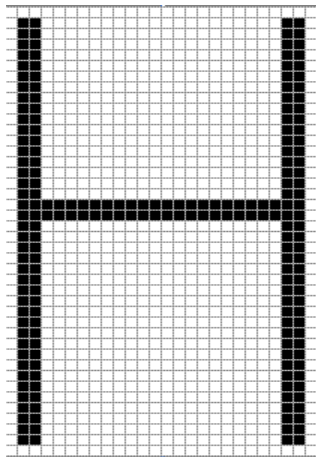
H   H   EEEEE   L       L       00
H   H   E       L       L       0 0
H   H   E       L       L       0 0
H   H   E       L       L       0 0
HHHHHH EEEEE   L       L       0 0
H   H   E       L       L       0 0
H   H   E       L       L       0 0
H   H   E       L       L       0 0
H   H   EEEEE   LLLLLLL LLLLLLL 00

```

这些图形可是能用文本编辑工具（例如，记事本）进行编辑的哦，在 BBS 或论坛上用这样的图形作为签名真是帅呆了！怎么样，想自己设计一个吗？本工具可以帮您实现梦想。

本工具的作用是将文本转化为字符图案，原理是使用 TImage 作为载体，将文字转换为图形点阵的信息，然后用特定的字符将图形点阵的信息表达出来。

为方便分析原理，将字体为的黑体的字母“H”放在 TImage 上，显示如下图。



TImage 其实是由网格（最小显示单位）组成的画布，网格上的黑点构成了我们肉眼看到的字母“H”。记录了每行的黑点位置的信息，我们称之为点阵信息。

本工具的作用就是将上图的黑点和空白点分别用特定的字符代替，然后输出该点阵信息。

1.2 关键代码

在 LAZARUS 的 IDE 里新建一个 Project，在 IDE 的控件栏里拖曳如下控件到窗体 Form 上：1 个 Image 控件，3 个 Edit 控件，1 个 Button 控件。

双击 Button 控件，在事件里输入以下代码：

```
1      var
2          I,J: Integer;
3          S: string;
4      begin
5          with Image1 do
6              begin
7                  Picture.Assign(nil);
8                  Canvas.Font.Assign(Edit1.Font);
9                  Picture.Bitmap.Width := Canvas.TextWidth(Edit1.Text);
10                 Picture.Bitmap.Height := Canvas.TextHeight(Edit1.Text);
11                 Canvas.TextOut(0, 0,Edit1.Text);
12                 Memo1.Clear;
13                 for J :=0 to Picture.Bitmap.Height - 1 do
14                     {每行的信息扫描}
15                     begin
16                         S := "";
17                         for I:=0 to Picture.Bitmap.Width - 1 do
18                             {每列的信息扫描}
```

```
19         if Picture.Bitmap.Canvas.Pixels[I,J] = clBlack then
20             S:= S + Edit2.Text
21             {如果该位置为黑点，则用前景字代替}
22         else S:= S + Edit3.Text;
23             {如果该位置为空白，则用背景字代替}
24         Memo1.Append(S); {写出一行的点阵信息}
25     end;
26 end;
27 end;
```

详细的代码及完整的工程见本书配套代码：TEXT2P.rar

1.3 界面效果

这款小工具运行的界面如下图所示。

聪明的读者可以在本工具上进行扩展，以满足各种不同的需求。如可改进为显示汉字，显示图形等等。

