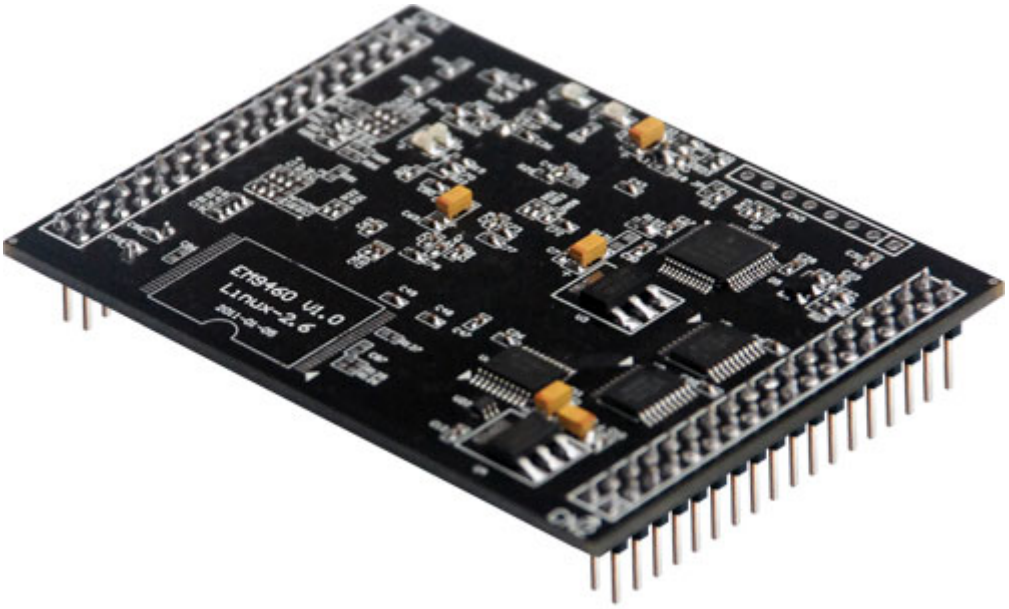
Emlinix



英利 Linux 工控主板应用程序编程手册

感谢您选择英利嵌入式 Linux 工控主板。

英利 EM9x60 系列工控主板包括五个型号：EM9160、EM9161、EM9260、EM9360 和 EM9460。为便于读者了解和使用英利产品，本手册中一些部分会以 EM9160 为例进行讲解；一些示例程序也会以 EM9160 命名。然而，本手册和上述示例程序完全适用于这五个产品。

英利 EM9x60 工控主板是面向工业自动化领域的高性价比嵌入式工控主板，其硬件核心为工业级的 ARM9 芯片 AT91SAM9260 和 AT91SAM9261（EM9161）。EM9x60 预装嵌入式 Linux-2.6 实时多任务操作系统，并针对板载的各个接口，提供了完整的接口底层驱动以及丰富的应用程序范例。用户可在此基础上，利用熟悉的各种软件工具直接开发自己的应用程序，以方便、快速地构成各种高性能工控产品。

本手册主要是为在英利嵌入式 Linux 工控主板上进行 Linux 应用程序开发的客户提供基本的编程指南。

此外，英利公司针对软硬件开发环境的配置编写有《英利 Linux 工控主板使用必读（EM9x60）》；针对工控主板和开发评估底板的使用编写有相应的使用手册。这些手册都包含在英利为用户提供的产品开发光盘里面，用户也可以登录英利公司网站下载相关资料的最新版本。

在使用英利产品进行应用开发的过程中，如果您遇到任何困难需要帮助，都可以通过以下三种方式寻求英利工程师的技术支持：

1、直接致电 **028-86180660　85329360**

2、发送邮件到技术支持邮箱**support@emlinix.com**

3、登录英利网站**www.emlinix.com**，在技术论坛上直接提问

另，本手册以及其它相关技术文档、资料均可以通过英利网站下载。

注：英利公司将会不断完善本手册的相关技术内容，请客户适时从公司网站下载最新

版本的手册，恕不另行通知。

再次谢谢您的支持！

目　　录

1 前言

Linux 操作系统是当前嵌入式系统中使用最为广泛的操作系统。一般来讲，要开发一款基于 Linux 的嵌入式产品，需要完成以下工作：

1、熟悉了解在 Linux 主机中开发的基本方法，这些方法通常以命令行方式为特征

2、从 Linux 社区下载与目标硬件相关的 Linux 代码，进行 Linux 平台移植工作

3、搭建合适的交叉编译工具链，完成应用程序的开发。包括以命令行方式，首先编写 Makefile 文件，然后通过执行 make 来调用 GCC 来实现各个 C 文件模块的编译以及最后的链接

要完成以上工作，企业不仅需要配备较高水平的软硬件工程师，而且开发周期通常会持续两年甚至更长的时间，这使得众多面对激烈市场竞争的中小企业，由于产品的开发周期以及相应风险的限制，不得不放弃 Linux 这样优秀的操作系统。

针对 Linux 应用中的这些问题，英利公司推出了预装嵌入式 Linux 操作系统的系列工控主板。作为一种高效、低成本的 Linux 产品解决方案，以 EM9x60 为核心的 Linux 应用平台是通过以下技术手段来满足客户需求的：

- EM9x60 的硬件接口非常丰富，包括以太网、串口、GPIO、USB、精简 ISA 扩展总线等资源。200MHz / 400MHz 主频的 32 位 ARM9 CPU 的处理能力，可轻松满足大多数嵌入式设备的功能需求，加之 EM9x60 低廉的价格，使其产品具有极高的性价比，确保基于 EM9x60 的智能产品在市场上的竞争力。
- EM9x60 上已经移植安装了 Linux-2.6.30 操作系统，包括所有接口的完整驱动程序，从而省去了客户为移植 Linux 平台所必需花费的大量人力物力，节约了开发时间，从而降低了开发风险。
- 为了让客户能把宝贵的时间花在自有产品的核心价值方面，我们选择并推荐客户采用 CodeSourcery 公司的 G++集成开发环境来开发自己的 Linux 应用程序。G++集成开发环境是一套完整的针对 Linux 运行平台的 C/C++开发工具，该工具的一个很大优点是可直接安装在 Windows 环境下。用户可在自己熟悉的 Windows 窗口环境下编写 Linux 的应用程序，然后通过 G++构造的交叉编译工具链，直接生成

可在嵌入式 Linux 环境下（这里为 EM9x60 的环境）运行的应用程序。这样客户原则上只需要学习 Linux 的应用程序的编程方法（对嵌入式应用来说，以多线程编程为特点），而无需学习常规 Linux 编程中复杂的命令行工具，就可完成自己的应用程序开发。

- 在嵌入式应用程序的编程中，可分为仅采用 C 开发或 C/C++开发两种风格。一般说来，对实现相同的功能，仅采用 C 编程对程序员的要求更高；而 C/C++编程，由于 C++提供了很好的面向对象设计的机制，使得应用程序设计变得更加简单，同时也更加安全。G++工具对 C++有完备的支持，这意味采用 C++来设计应用程序会以更短的时间开发出更加高效安全的代码，所以我们强烈建议客户采用 C++来设计自己的应用程序。

众所周知，在嵌入式产品的开发中，应用程序的开发是最为重要的部分，同时也是客户产品的核心价值所在。本手册的主要目的，是为那些采用 EM9x60 作为产品核心平台，并在 Windows 环境中直接开发 Linux 应用程序的客户提供一份详细的循序渐进的编程指南。因为我们认定采用此方法进行 Linux 嵌入式产品的开发最有利于客户以最小的代价、最低的风险、最快的速度把自己的产品推向市场。

我们认为阅读本手册的客户，已经根据英利 Linux 工控主板开发资料中的《英利 Linux 工控主板使用必读》中的相关内容，正确搭建了基于英利 Linux 工控主板评估套件的开发环境，包括硬件平台的连接安装、创建超级终端或安装好其他串口调试工具、配置 NFS 网络服务器、安装 CodeSourcery G++集成开发环境等；并在此环境中能正确运行“Hello”等基本测试程序。

本手册其他各章的内容如下：

第 2 章 G++集成开发环境入门，主要介绍如何创建应用程序工程（Project），包括导入现有的工程文件；添加或删除 C++文件；编译链接整个工程形成可执行文件；编译单个 C++文件等内容。

第 3 章 通过 NFS 进行应用程序的调试，一方面介绍 Windows 环境下网络文件服务器（NFS）工具 nfsAXe 的使用方法，另一方面介绍在 EM9x60 环境中，通过控制台窗口（console，通常为 Windows 的超级终端窗口）执行简单的 Linux 命令来调试 Linux 应用程

序的基本方法。

第 4 章 驱动程序及其调用方法 主要介绍针对英利 Linux 工控主板特色功能而编写的驱动程序的使用方法，包括精简 ISA 总线、GPIO、矩阵键盘、硬件中断等。

第 5 章到第 8 章，每一章将以一项英利 Linux 工控主板的典型应用功能为对象，详细介绍应用程序的实现方法，并特别注重使用 C++的面向对象的设计方法以及 Linux 线程的设计方法的介绍。各章的介绍内容为：

第 5 章介绍基于精简 ISA 总线驱动的 LCD 显示功能的实现；

第 6 章是以基本的 Linux 串口 API 函数为基础，构造串口类，支持多线程操作的方法；

第 7 章则是基于标准 Socket 的 API 函数，介绍支持多连接的 TCP 服务器的方法；

第 8 章介绍支持多个连接的客户端的方法。网络的多连接管理都是采用了 C++的类的手段来封装对象，从而简化了程序结构。

2 G++集成开发环境入门

Sourcery G++是针对嵌入式 C/C++应用编程的一套完整、专业的集成开发环境，其核心的编译器是 Linux 系统通用的 GCC 编译器，这也是 G++这一名称的由来。在常规的 Linux 应用程序开发中，一般是以命令行方式，首先编写 Makefile 文件，然后通过 make 工具来运行 GCC，完成对程序的编译链接的。这种编程方法由于门槛较高，使 Linux 的应用范围受到相当的限制，特别是对技术力量相对薄弱的中小企业更是如此。而当采用 G++集成开发环境时，用户已不再需要涉及复杂的 Makefile 文件的编写，可把精力集中在应用程序本身实现的功能上，从而大大加快了应用程序的开发进度。

Sourcery G++的 IDE 工具在 Windows 操作系统上的安装过程，请参考《英利 Linux 工控主板使用必读》一文，其中有详细的介绍。

在进行应用程序开发时，往往会根据不同的功能把应用程序划分成多个模块来进行设计，而每个模块通常对应一个源文件，这样有利于应用程序的管理和维护，对于具有多个模块的应用程序的管理一般是通过应用工程文件进行的，因此每一个应用程序对应着一个应用工程文件。本章主要是通过介绍创建、修改、编译应用程序工程文件的方法，对 Sourcery G++工具软件的使用进行详细说明。

2.1 Sourcery G++ IDE 下创建和管理 C++应用工程

对一个全新的 Linux 应用程序开发，用户可创建全新的工程文件，也可利用已有的工程文件进行必要的修改，来完成应用程序的开发。

创建一个新的工程文件

1、创建一个新的工程文件，是通过选择**File -> New -> Project…**来实现的，我们建议客户选择**C++ Project**，如图 2-1 所示。

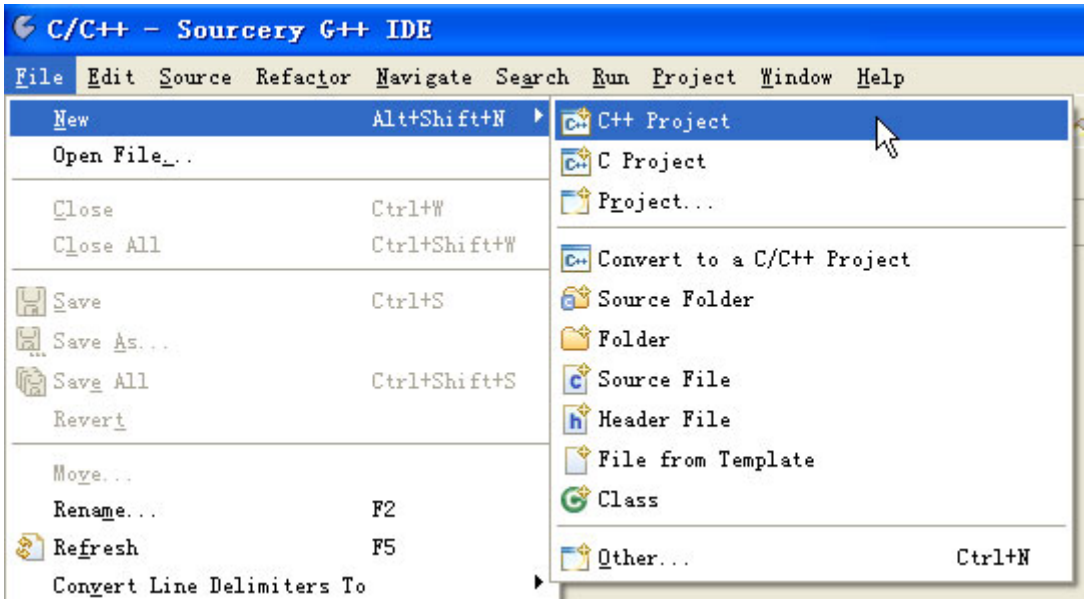


图 2-1 建立新的 C++工程文件

2、接下来输入工程文件的名称以及相应的位置路径，在这里建议不要选用系统的缺省路径，而是选择用户自行建立的 Linux 应用程序开发路径，以便于应用程序的管理。然后选择 **Project type: -> Executable -> Empty Project**，如图 2-2 所示。

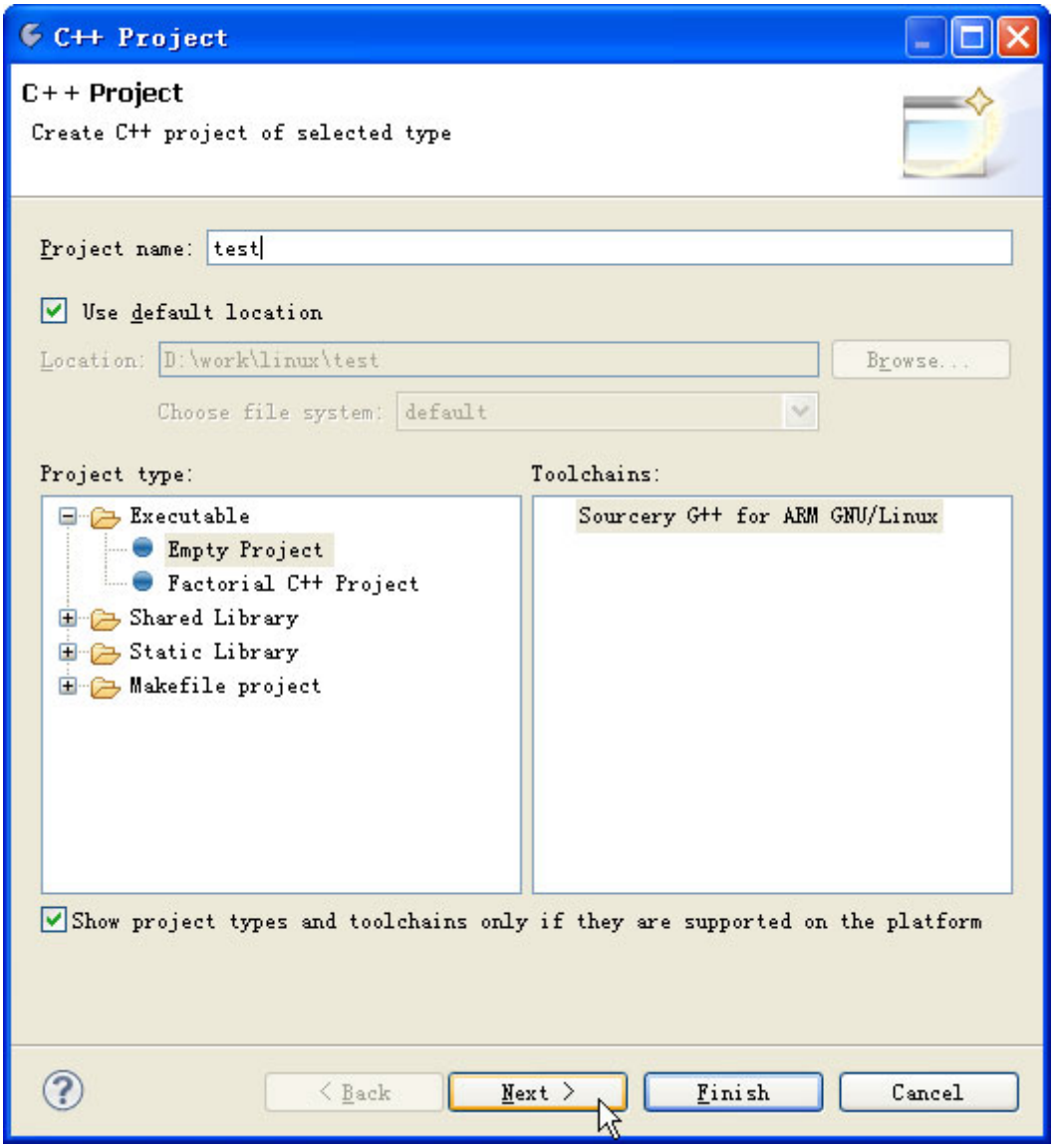


图 2-2 工程文件命名

3、点击 **Next** 进入 G++属性设置菜单，选择目标机的处理器类型、大端或小端模式以及浮点处理支持等。对于 EM9160 来说，这些设置均可以采用缺省配置。

4、然后直接点击 **Finish**，一个新的工程文件就建立完毕了。

打开已有的工程文件

1、选择**File -> Import...**，在**General**下选择**Existing Projects into Workspace**，如图 2-3 所示。

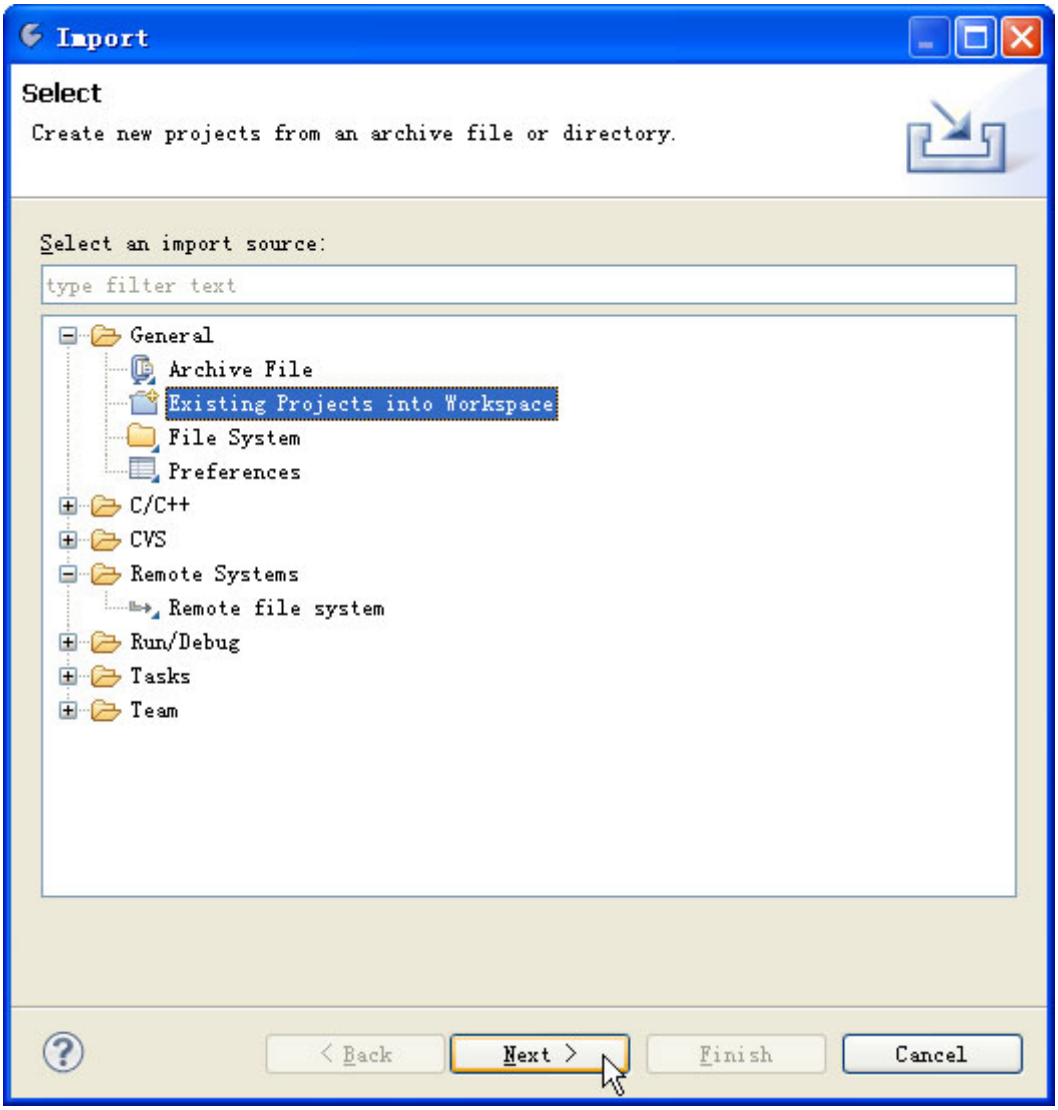


图 2-3 打开已有的工程文件

2、点击**Next**，通过**Browse…**选择已有的工程文件的目录，然后点击**Finish**打开图中显示的Step1_LCDTest应用工程，如图 2-4 所示。

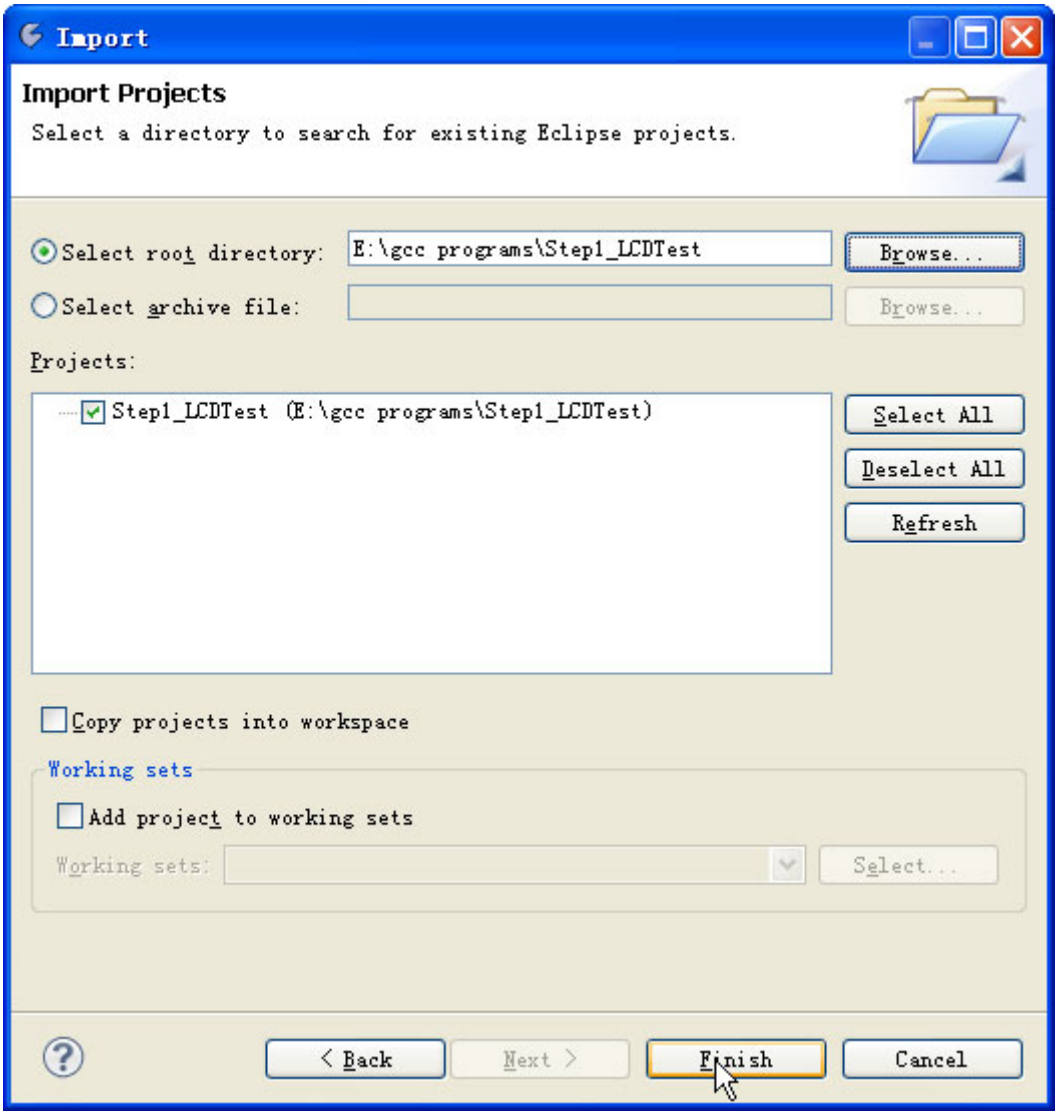


图 2-4　找到工程文件并打开

添加新的源文件

在Project Explorer视窗下选择需要添加源文件的工程，然后点击鼠标右键，选择**New -> Source File**或者**New -> Header File**，即可增加新的CPP或者H文件到该工程中。

编辑完成CPP文件或者H文件后，可以通过菜单中**File -> Save**或者通过按**Ctrl+S**键进行保存。

添加已有源文件

1、在Project Explorer视窗下，选择需要添加已有源文件的工程，然后点击鼠标右键，选择**Import...**，选择**General -> File System**，如图 2-5 所示。

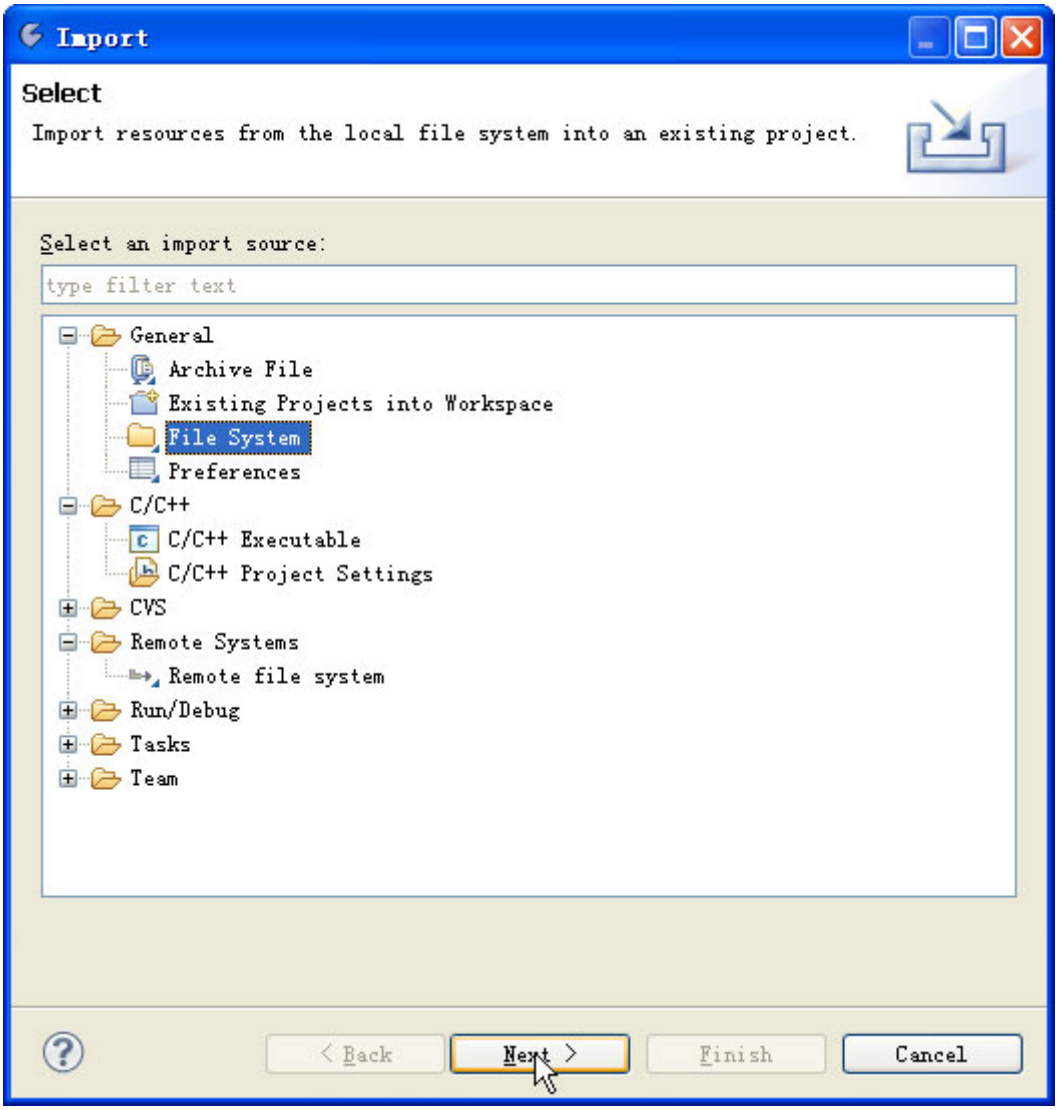


图 2-5 添加已有的源文件

2、通过打勾进行文件的选择，如图 2-6 所示。

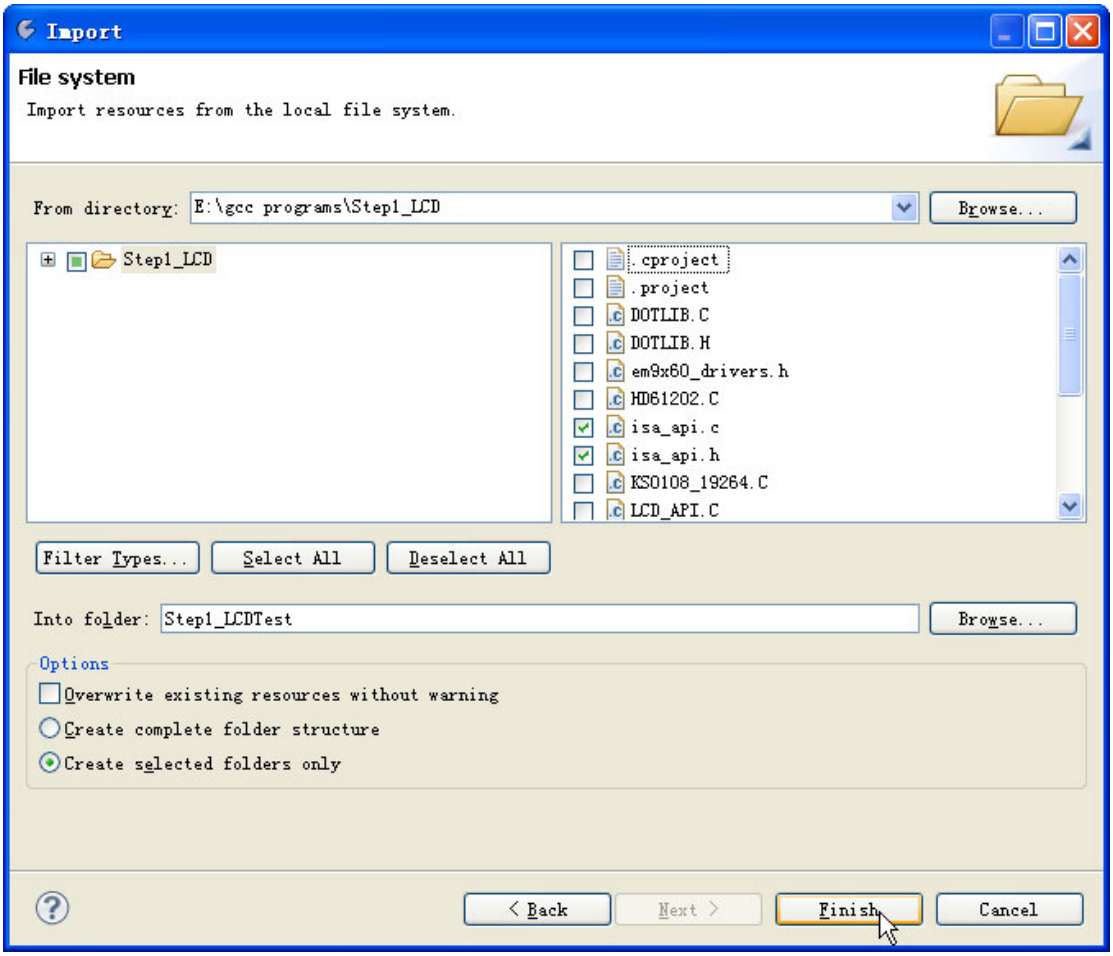


图 2-6　选择要添加的源文件

删除文件

删除工程文件：在该工程文件上点鼠标点键，然后选择**Delete**，如图 2-7 所示。

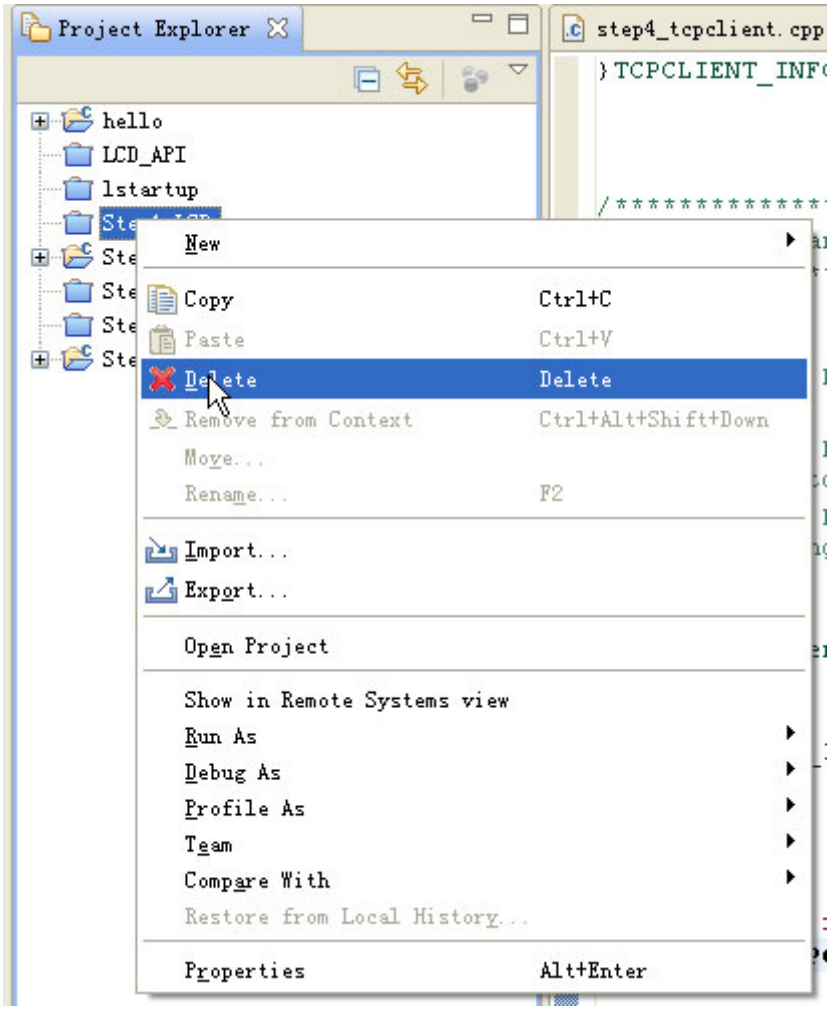


图 2-7 删除工程文件

删除工程文件中的源文件：将鼠标移至该文件处，然后点击鼠标右键，选择**Delete**，如图 2-8 所示。

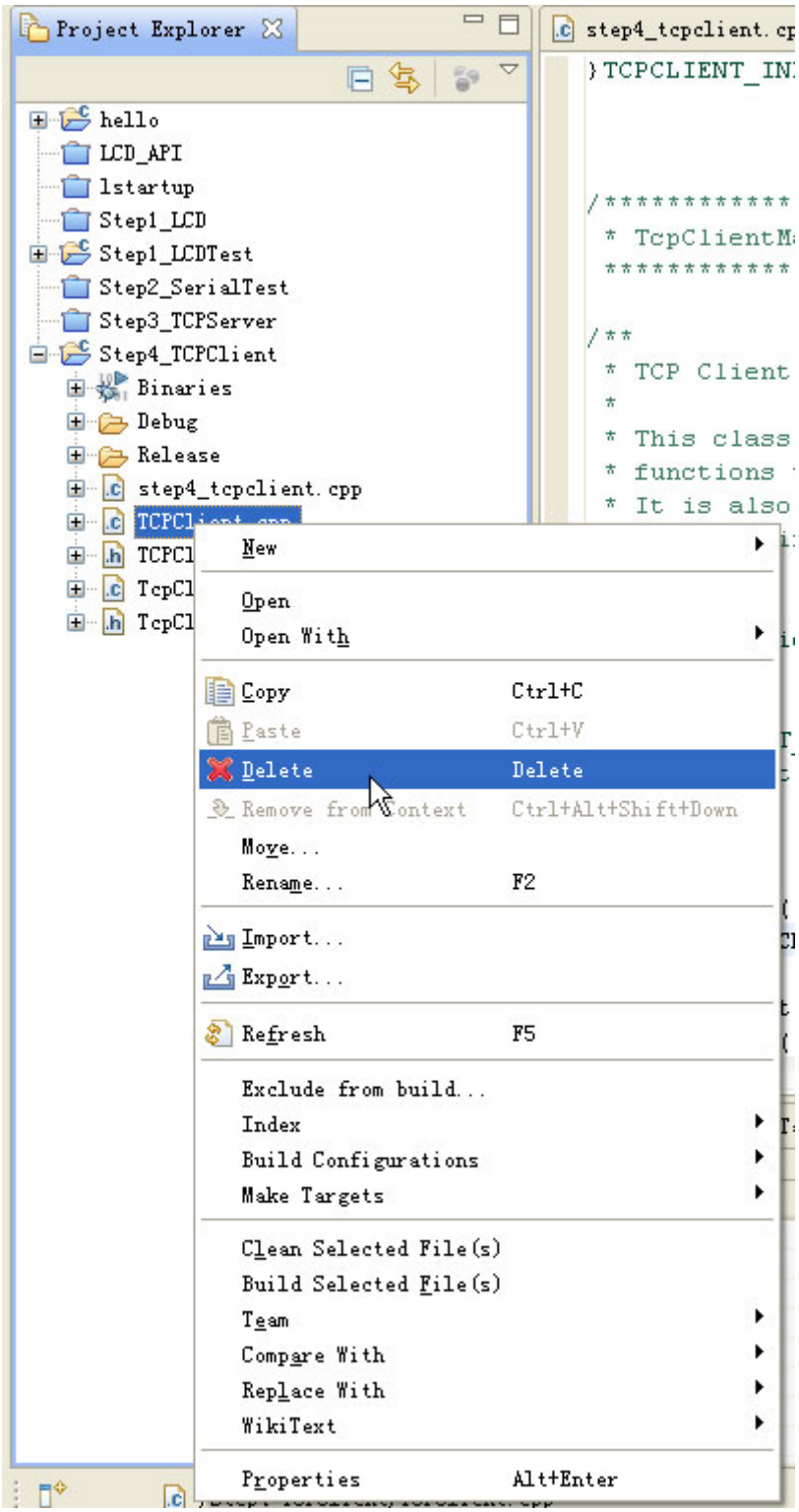


图 2-8　删除源文件

2.2 Sourcery G++ IDE 下编译 C++应用工程

工程文件编辑完成后，就在 Sourcery G++ IDE 环境下直接进行编译生成相应的可执行文件。

在Project Explorer窗口下选择需要编译的工程文件，如下例选择Step4_TCPClient工程，然后点击菜单中**Project -> Build Project**即可进行程序的编译，如图 2-9 所示。也可以选择使用工具栏中的  图标。编译输出的结果显示在Console窗口中，如果因为某种原因该窗口没有显示，可以通过选择**Window -> Show View -> Console**来实现。

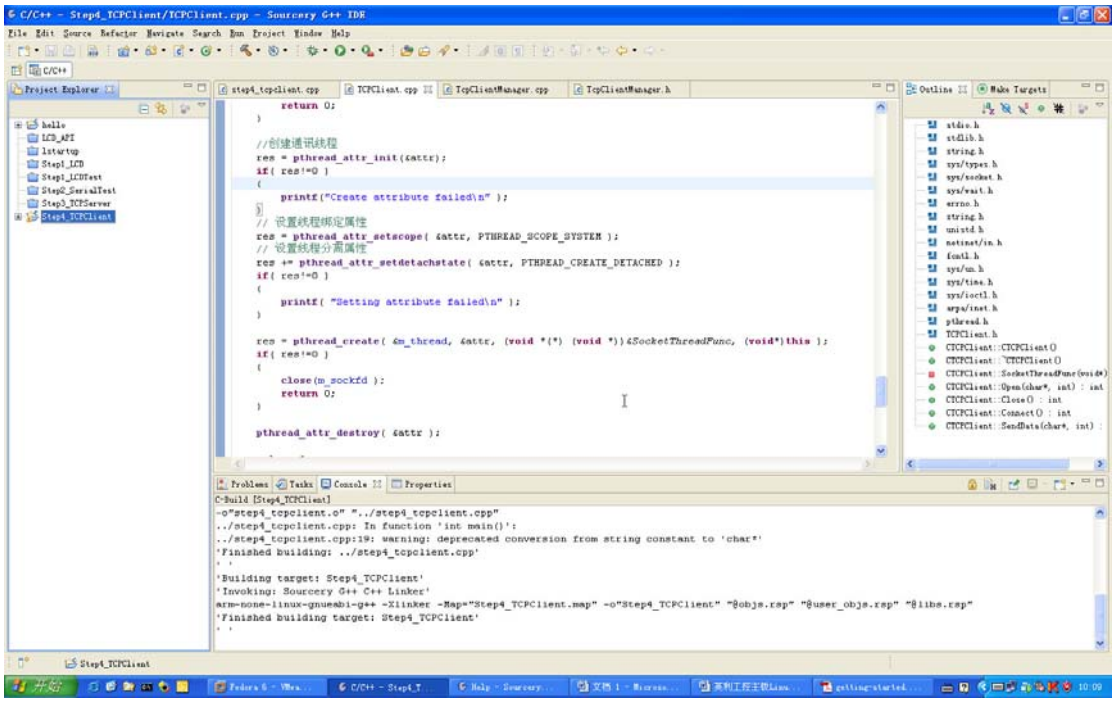


图 2-9 编译工程文件

编译模式的选择

选择应用工程编译的 Debug（调试版本）和 Release（发行版本）。在 Project Explorer 视窗下，选择需要调试的工程文件，然后点击鼠标右键，选择 **Build Configurations -> Set Active -> Release** 项，如图 2-10 所示。

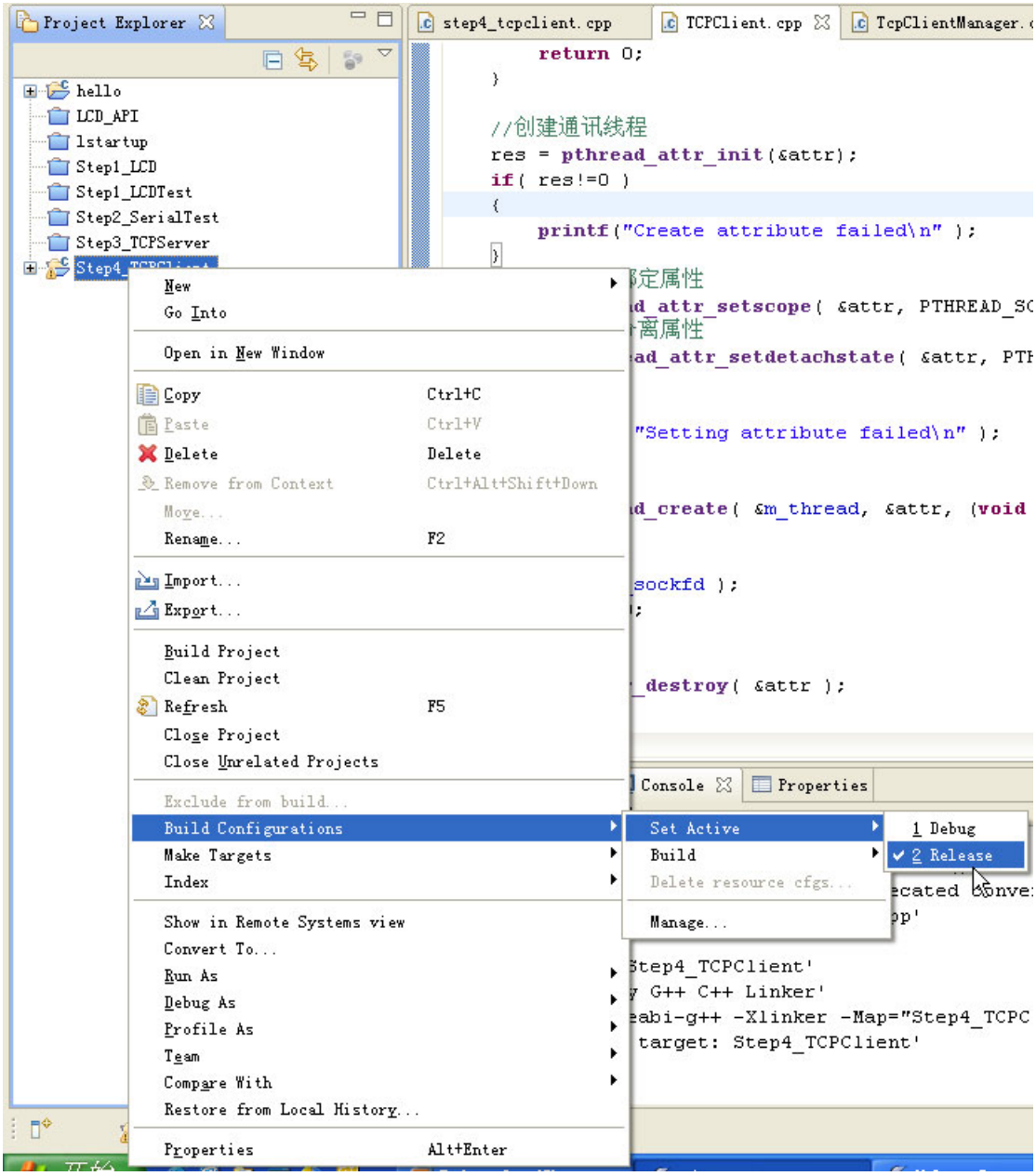


图 2-10　编译模式选择

编译属性的设置

设置应用工程编译属性，在实际应用中用得比较多的是应用工程需要 Link 专用的库文件，此时就需要对 C/C++的编译属性进行相应的设置。

在Project Explorer视窗下，选择需要设置的工程文件，然后点击鼠标右键，选择**Properties**项，在窗口中选择**C/C++ Build -> Settings -> Tool Settings -> Sourcery G++ C++ Linker -> Libraries**，如图 2-11 所示。其中的一个窗口用于指定库文件的名称，一个用于指定库文件的路径，对于系统中已有的库文件，就不需要指定路径；而对于用户专用的库文件，则需要指定路径。

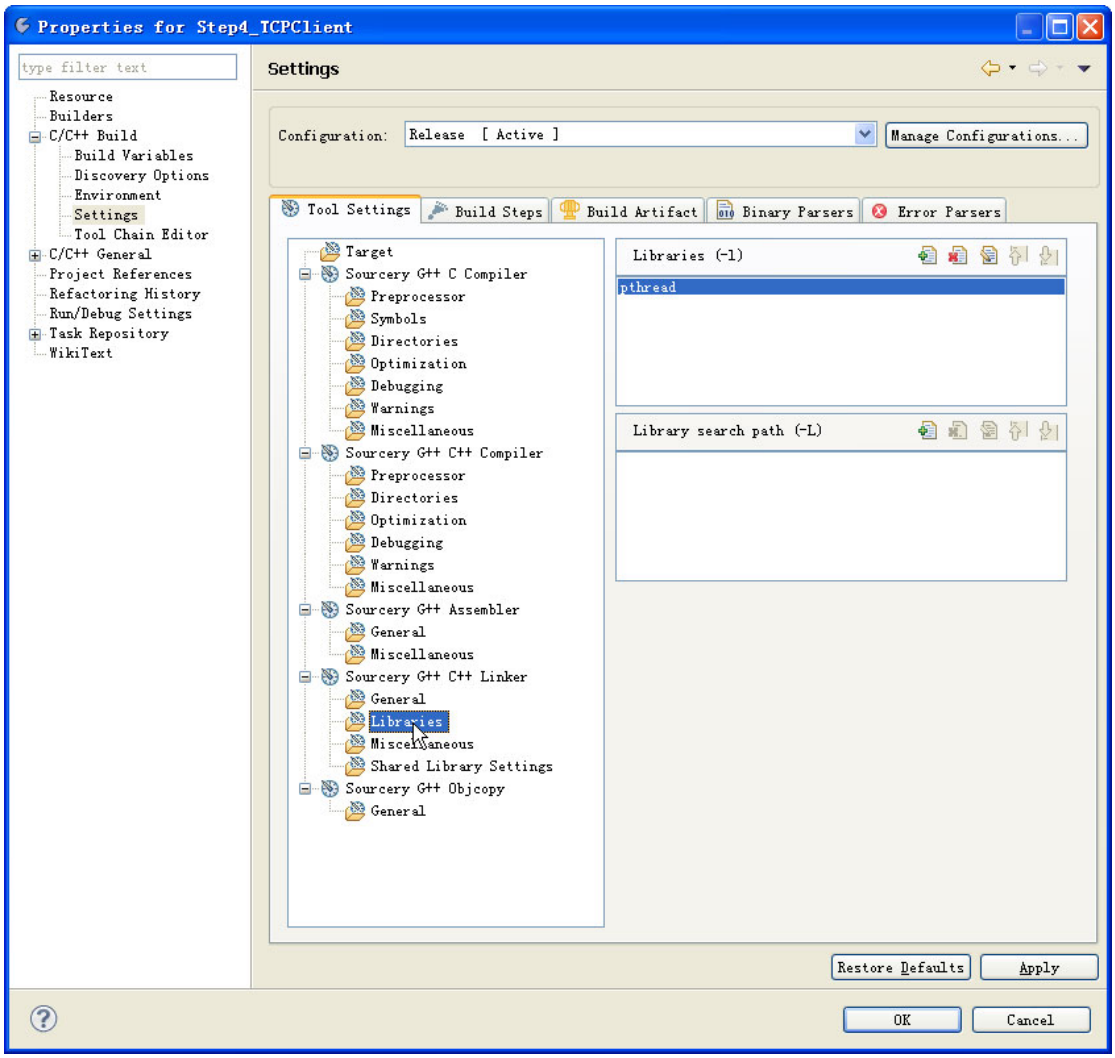


图 2-11 设置库连接

在进行 Linux 应用程序开发时，如果采用了多线程的操作，进行编译前，需要把线程库 libpthread 添加到链接选项中。注意，在 G++集成环境的缺省设置中没有链接此库。

如在 EM9160 的 LCD 显示程序中，如果用到英利封装的一个支持具有 C 风格的操作 LCD 相关的 API 函数库 libLCD_API.a，该库文件放置在相应的应用工程目录下，此时编译属性中还需要设置 libLCD_API.a 库文件所在的文件路径，其设置如图 2-12 所示。

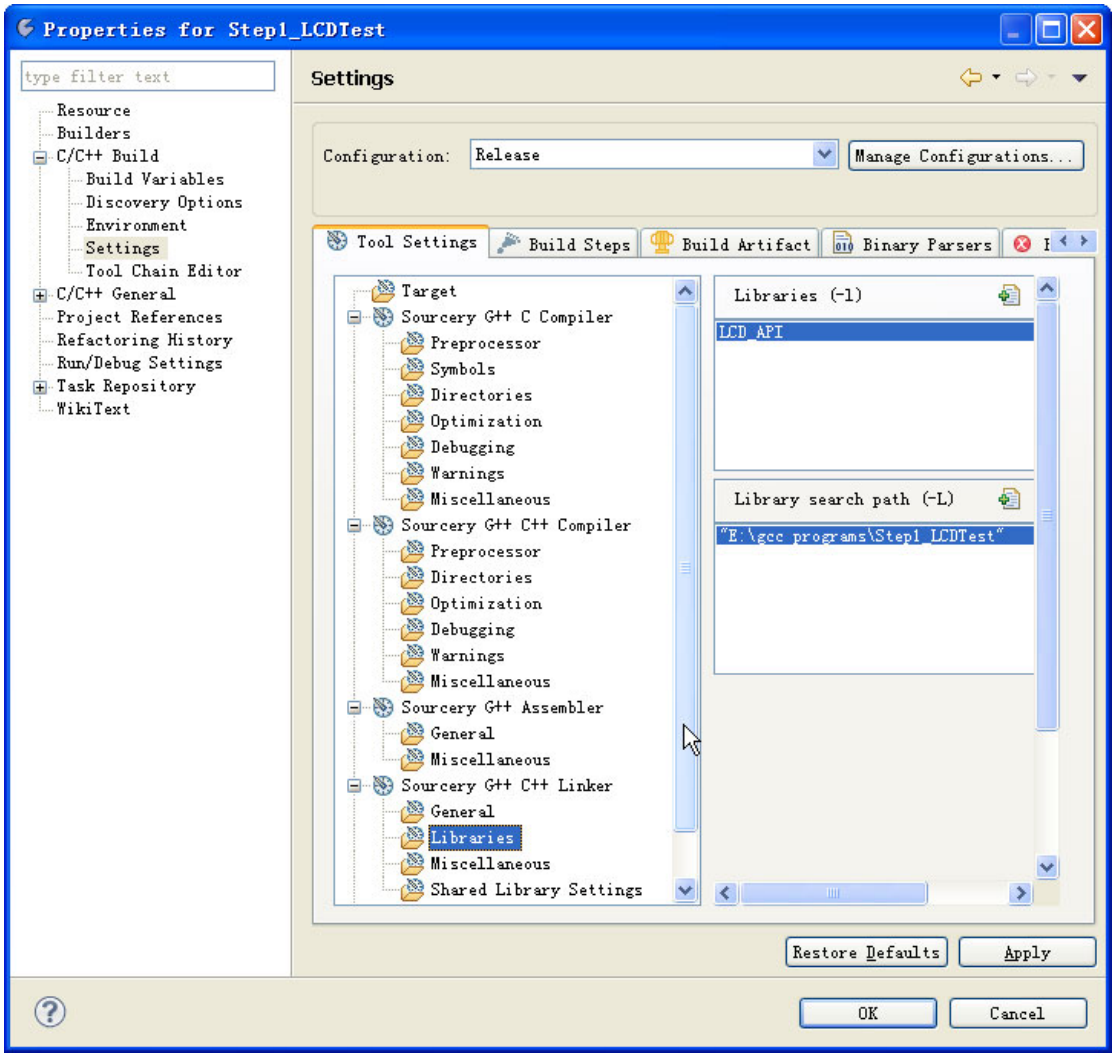


图 2-12　库文件属性设置

3 通过 NFS 进行应用程序调试

Linux 应用程序的调试，首先需要建立网络文件系统 NFS。所谓 NFS，是 Linux 操作系统支持的一种非常有特色的挂载式的文件系统，基本方法是在开发主机（Windows 环境）上运行一个 NFS 服务器，并把应用程序所在目录设置为完全共享目录；在 EM9x60 的控制台窗口（一般为 Windows 超级终端）执行 Linux 的 mount 命令，这时应用程序所在目录就被完全映射到了 EM9x60 的一个指定目录下。作为基本的调试方法，用户可在控制台窗口直接运行编译好的应用程序，通过代码中插入的 printf 语句来了解程序运行的相关状态。

Linux 环境还提供了称为 GDB 的调试工具，基本方法是在目标板 EM9x60 上运行 GDB 服务器，通过以太网进行源码的调试。但事实上由于 GDB 调试器的使用相当繁琐复杂，需要花大量时间去熟悉了解它，因此在 Linux 应用程序的开发中很少用到 GDB 调试器。

对绝大多数 Linux 应用开发，采用基于 NFS 的基本调试方法，已完全满足应用需求。一方面是因为有了 NFS，应用程序的修改、运行、终止（直接在控制台按 **Ctrl+C** 键即可终止当前应用程序）都非常方便，完全能保证调试进度。另一方面应用程序处于完全真实的运行环境中运行，一旦程序调试通过，程序的相关功能就是正确实现了，不会出现调试状态程序运行正常，而运行状态程序却有异常的问题。由于基于 NFS 的基本调试方法是如此简单，因此应用程序的调试问题就简单地转变成了 NFS 文件系统的搭建问题。为此本章主要介绍如何在 Windows 开发主机和 EM9x60 间建立 NFS 文件系统。

3.1 在 Windows 开发主机配置 NFS 服务器

NFS（网络文件系统）在进行嵌入式 Linux 应用开发上非常有意义，如果正确地配置了 NFS 文件系统，用户就可以将一个目录输出到 NFS 服务器，并可以将该目录挂载到一个远端客户端机器上，对于客户端的使用者来说，挂载的该目录就好像是本机的一个文件系统一样。

对于EM9x60，可以使用Windows环境下的Sourcery G++进行Linux应用程序的开发，并通过配置的NFS文件系统，将用户在Windows主机上的应用程序工作空间挂载到EM9x60上，这样的好处是在开发过程中，在开发主机上对应用程序文件的修改，会在EM9x60 主板上立即生效并可执行。为了达到这个目标，需要在Windows开发主机上安装并启动NFS服务器，NFS Server程序我们建议客户使用LabF.COM公司提供的nfsAxe。下载该工具的链接：**http://labf.com//download/nfsafx.html**。关于该软件的安装、NFS配置的说明请参见《英利Linux工控主板使用必读（EM9x60）》中的相关章节。

3.2 使用 NFS 在 EM9x60 主板上挂载应用程序工作目录

在 Windows 开发主机上启动 NFS Server，并将应用程序所在的目录定义为需要挂载的目录，如图 3-1 所示。

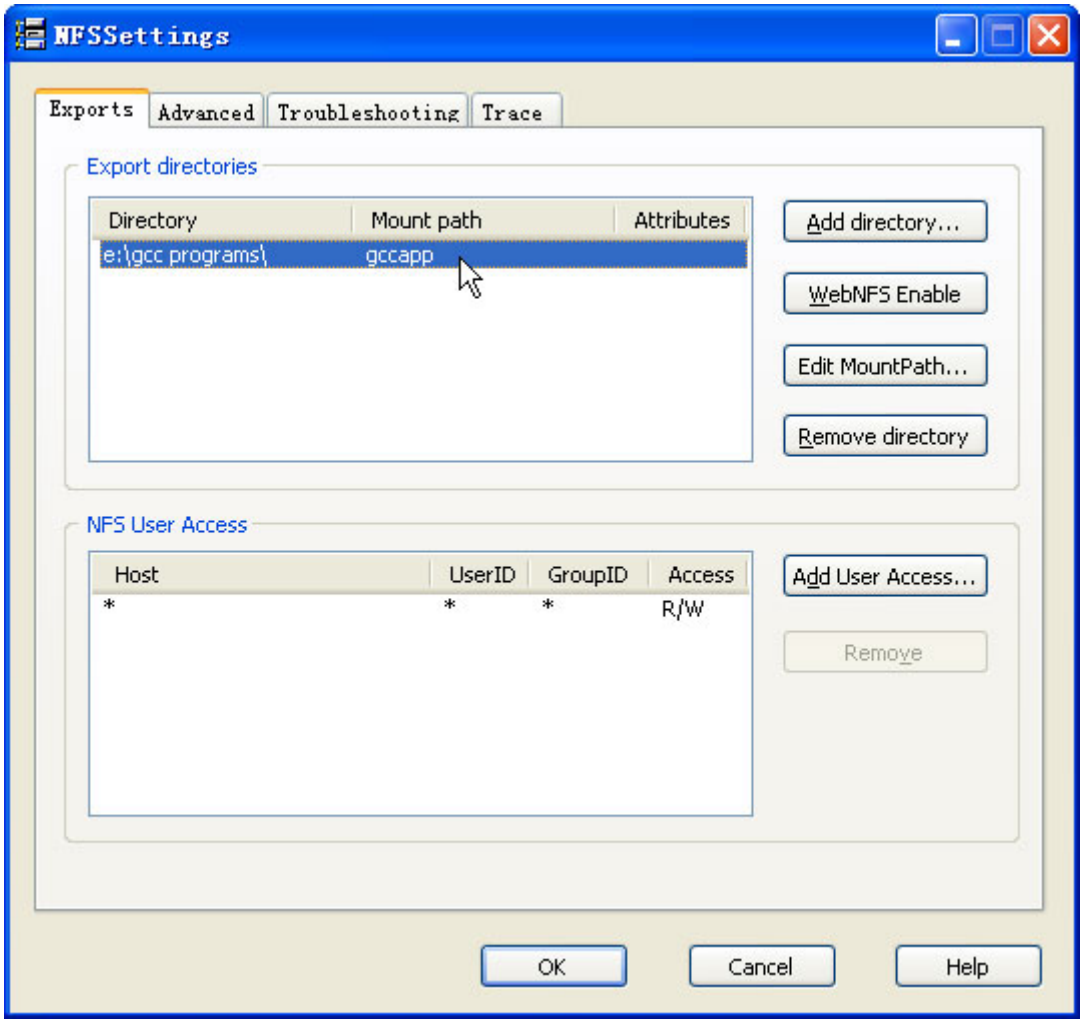


图 3-1 指定挂载目录

其中的 Mount path 可以自定义名称。

在 EM9x60 的调试模式下，系统将会自动执行挂载操作，将应用程序所在的目录空间自动挂载到 EM9x60 系统的/mnt/nfs 目录下。具体方法是在 userinfo.txt 文件中配置 Windows 开发主机的 IP 地址以及 NFS 服务器 Mount path 的名称如下（具体 IP 地址和 NFS 服务器名称由用户根据所在网络和自己的配置填写）：

[NFS Server]

IPAddress="192.168.201.170"

Mountpath="gccapp"

挂载成功后，对于 EM9x60 主板来说，Windows 主机上的应用程序目录空间，就像本机的一个文件夹一样，用户可以直接通过超级终端访问/mnt/nfs 目录下的文件。

用户也可以在 EM9x60 的控制台窗口运行挂载文件系统的 mount 指令进行 NFS 文件挂载：

[root@EM9x60]# **mount -t nfs -o nolock 192.168.201.170:gccapp /mnt/nfs**

注意在上述命令中，名称“gccapp”就是指向 Windows 开发主机上应用程序所在的目录。任何时候若不再需要使用，可执行 umount 指令卸载文件系统：

[root@EM9x60]# **umount /mnt/nfs**

3.3 应用程序测试运行

通过 NFS 挂载应用程序文件目录成功（即将 Windows 开发主机上的应用程序目录文件挂载到 EM9x60 的/mnt/nfs 目录下）以后，就可以直接在终端模式操作此目录，直接运行 Linux 应用程序。需要注意的是命令行前面要加上“**./**”，如运行 Step4_TCPClient 测试程序，在控制台窗口应输入“**./Step4_tcpclient**”。程序在运行的过程中，可以按 **Ctrl+C** 键立即终止运行，如图 3-2、3-3 所示。

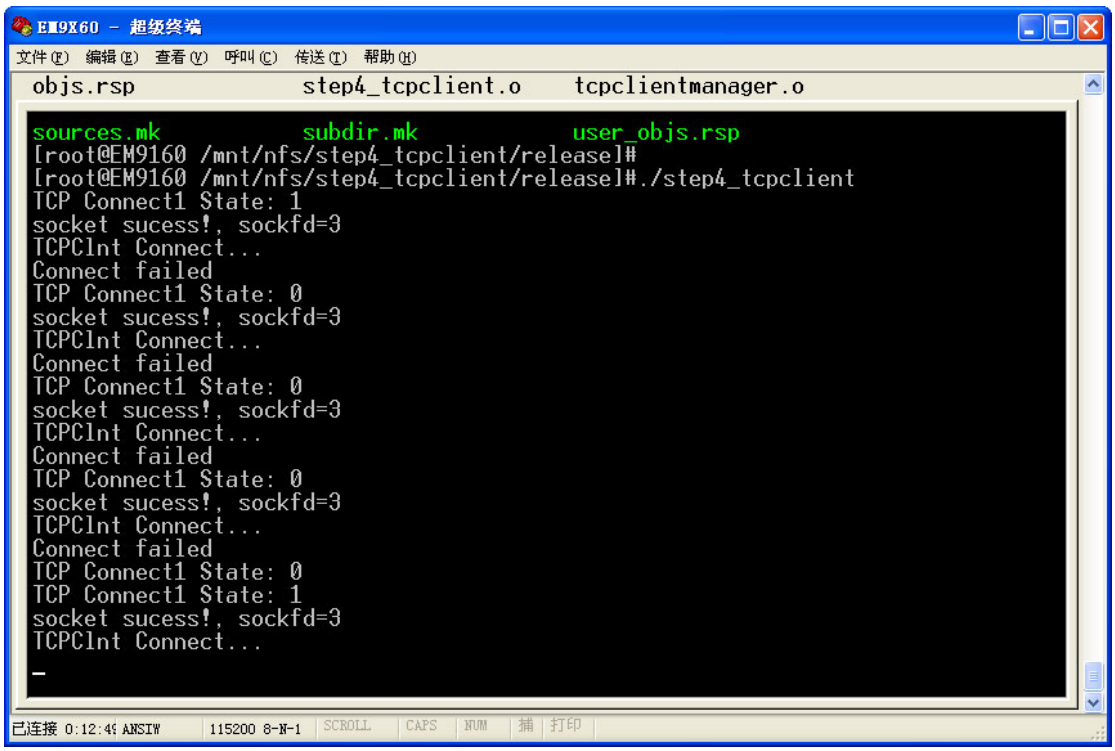


图 3-2 运行程序

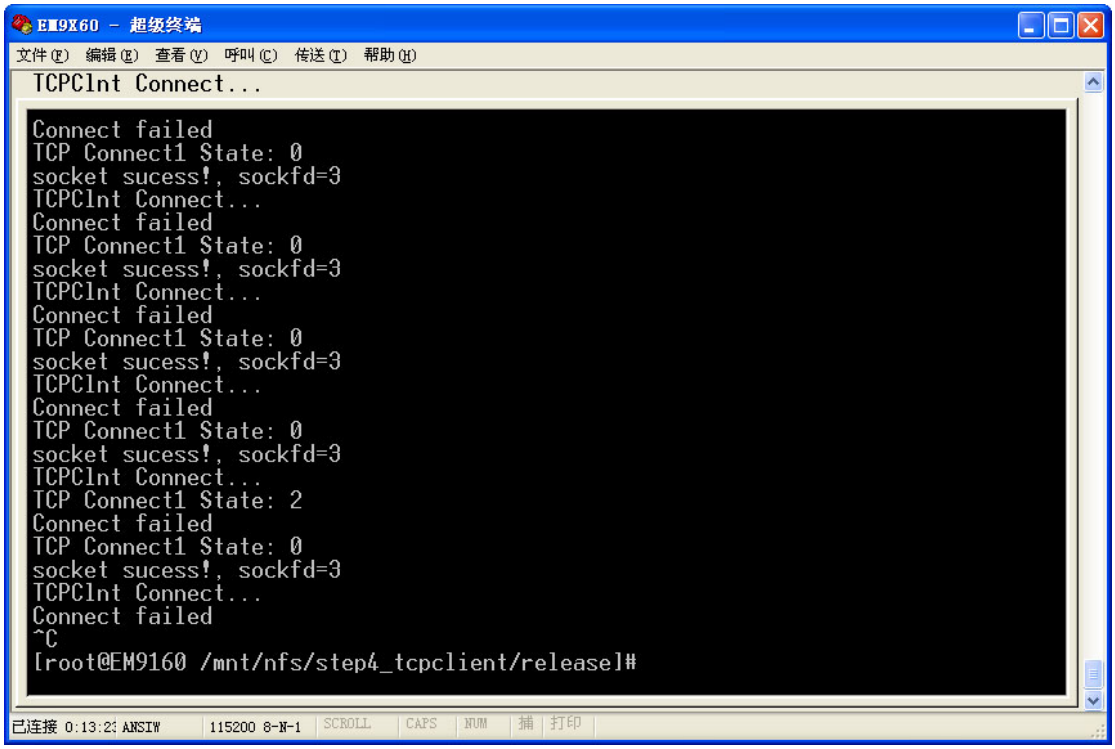


图 3-3 终止程序

3.4 应用程序 printf 语句的使用

采用终端测试运行对应用程序进行调试时，可以利用 printf 进行终端信息的输出。

查看变量值，如：

```
printf( "TCP Connect%d State: %d \n", i2+1, i3 );
```

查询程序运行的状态，如：

```
printf( "TCPClnt Connect...\n" );
printf( "TCPClnt Opened\n" );
```

在使用 printf 进行调试信息输出时，需要注意的是语句的结束处一定要加上换行符“**\n**”，否则程序将阻塞在该条语句中。

4 驱动程序及其调用方法

在 Linux 系统中，应用程序对所有硬件接口的操作都是通过调用相应的硬件设备驱动程序的 API 函数来实现的。针对嵌入式系统，最典型的设备当属以太网接口和异步串口（即通常所说的 RS232 和 RS485 接口）。对这些接口，Linux 都已有完善的 API 接口函数。本手册将在后续章节中详细介绍异步串口和以太网接口的编程方法，而在本章中则主要介绍针对英利 Linux 工控主板 EM9x60 硬件的专用设备驱动程序的调用方法。英利 Linux 工控主板 EM9x60 的专用设备驱动程序均采用 Linux 的字符型设备驱动程序模式构成。

4.1 Linux 驱动程序调用方法概述

在 Linux 环境中，所谓通过驱动程序来操作相应硬件接口，就是指应用程序打开特定文件名称的文件，然后通过常规的对文件读写或控制的方法，来实现对硬件接口的访问。在嵌入式系统中，对硬件的读写通常采用 ioctl 函数的形式，其典型的流程如下：

流程	实现功能	实现函数	备注
第一步	打开文件	fd = open(devname, …);	特定设备文件名称
第二步	访问接口	rc = ioctl(fd, cmd, …);	可能调用多次，多种 cmd
第三步	关闭文件	Close(fd);	不再操作，则关闭

在应用 EM9x60 的设备驱动程序中，最常用的函数是 ioctl(…)，其基本定义为：

```
int ioctl(int fd, int cmd, unsigned long arg);

输入参数 fd：  为打开文件获得的文件描述符
         cmd：  操作命令码，在 em9x60_drivers.h 头文件中以定义
         arg：  需要传递的参数指针，不同的命令必须严格匹配相应的参数类型

返回值 = 0：  调用成功
  = EBADF：  非法文件描述符
 = EFAULT：  arg 指向不可访问的内存空间
 = EINVAL：  非法的 cmd 或 arg 参数
= ENOTTY：  文件描述符未指向特殊的字符设备
```

EM9x60 的专用设备驱动程序都有特定的设备文件名称，列表如下：

EM9x60 驱动程序	设备名称	简要功能描述
精简 ISA 总线驱动程序	“/dev/em9x60_isa”	总线读写操作
GPIO 驱动程序	“/dev/em9x60_gpio”	每位 GPIO 可独立设置
矩阵键盘驱动程序	“/dev/em9x60_keypad”	支持 4x5 矩阵键盘
外部中断驱动程序	“/dev/em9x60_irq1” “/dev/em9x60_irq2”	实时响应外部中断信号
看门狗 WDT 驱动程序	“/dev/watchdog”	应用程序接管喂狗任务
CAN 接口驱动程序	“/dev/em9x60_can1” “/dev/em9x60_can2”	EM9260 EM9360 支持 CAN 接口通讯
系统信息配置驱动程序	“/dev/em9x60_sysinfo”	读取相关配置信息

EM9161 的专用设备驱动程序都有特定的设备文件名称，列表如下：

EM9161 驱动程序	设备名称	简要功能描述
精简 ISA 总线驱动程序	“/dev/em9161_isa”	总线读写操作
GPIO 驱动程序	“/dev/em9161_gpio”	每位 GPIO 可独立设置
外部中断驱动程序	“/dev/em9161_irq1”	实时响应外部中断信号
看门狗 WDT 驱动程序	“/dev/watchdog”	应用程序接管喂狗任务
CAN 接口驱动程序	“/dev/em9161_can1”	支持 CAN 接口通讯
系统信息配置驱动程序	“/dev/em9161_sysinfo”	读取相关配置信息

下面以 EM9160 为例说明各驱动程序的调用方法。

4.2 精简 ISA 总线驱动程序

EM9160 的精简 ISA 总线由 8 位数据总线、5 位地址总线以及片选和读写控制信号组成。用户在设计基于 EM9160 的嵌入式产品时，除了使用 EM9160 板上已提供的接口功能外，还有可能需要扩展一部分面向自己特定应用的专用接口电路，如 AD 数据采集、数字 IO 等等，这些接口电路的扩展都可以非常方便地通过精简 ISA 总线来实现；应用程序通过精简

ISA 总线的驱动程序就可对扩展的接口单元进行读写操作控制了。有关精简 ISA 总线的信号特性、总线时序以及电路扩展方法，用户可参考《EM9160 工控主板数据手册》和英利公司网站中关于 ISA 总线扩展应用的方案，本节主要介绍精简 ISA 总线驱动程序的使用方法。

打开精简 ISA 总线设备文件的方法为：

```
fd = open("/dev/em9x60_isa", O_RDWR);
```

EM9160 的精简 ISA 总线共支持 8 种类型的操作：

1、读取 LCD 总线时序类型（motorola 时序或 intel 时序）

2、设置 LCD 总线时序类型（motorola 时序或 intel 时序）

3、读 LCD 片选区某寄存器数据

4、写数据到 LCD 片选区某寄存器

5、读 CS0 片选区某寄存器数据

6、写数据到 CS0 片选区某寄存器

7、读 CS1 片选区某寄存器数据

8、写数据到 CS1 片选区某寄存器

相应的，在“em9x60_drivers.h”中定义了对应的 ioctl 命令：

```
#define EM9X60_ISA_IOCTL_READ_LCDTYPE     _IOR(EM9X60_MAGIC,  0x40, unsigned int)
#define EM9X60_ISA_IOCTL_WRITE_LCDTYPE    _IOW(EM9X60_MAGIC,  0x41, unsigned int)
#define EM9X60_ISA_IOCTL_READ_LCD         _IOWR(EM9X60_MAGIC, 0x42, struct isa_io)
#define EM9X60_ISA_IOCTL_WRITE_LCD        _IOW(EM9X60_MAGIC,  0x43, struct isa_io)
#define EM9X60_ISA_IOCTL_READ_CS0         _IOWR(EM9X60_MAGIC, 0x44, struct isa_io)
#define EM9X60_ISA_IOCTL_WRITE_CS0        _IOW(EM9X60_MAGIC,  0x45, struct isa_io)
#define EM9X60_ISA_IOCTL_READ_CS1         _IOWR(EM9X60_MAGIC, 0x46, struct isa_io)
#define EM9X60_ISA_IOCTL_WRITE_CS1        _IOW(EM9X60_MAGIC,  0x47, struct isa_io)
```

特别需要注意的是，精简 ISA 总线的读写涉及两个参数：地址偏移量和读写的数据，因此定义了专门的数据结构 struct isa_io 作为参数类型：

```
struct isa_io
{
    unsigned long    dwOffset;
    unsigned char    ucValue;
};
```

在进行具体读写操作时，需先设置参数，然后再调用 ioctl 函数。如对 CS1 片选段，地址偏移为 2 的地址（SA=0'b00010）写入数据 0x55：

```
struct isa_io      IsaArg;
IsaArg.dwOffset = 2;
IsaArg.ucValue    = 0x55;
rc = ioctl(fd, EM9X60_ISA_IOCTL_WRITE_CS1, &IsaArg);
```

EM9160 共有 5 位地址总线，所以对 CS0 和 CS1 的读写，其地址偏移量为 0-31；对 LCD，仅使用了 SA0-SA2，所以偏移量为 0-7。

EM9160 以精简 ISA 总线为基础，设置了专门的 LCD 读写控制信号，以支持嵌入式系统中常用的低成本点阵类型的单色 LCD 模块：

点阵格式	控制器类型	时序类型	简要说明
128×64	KS0108	Motorola	最常用 LCD 模块
240×128	T6963C	Intel	
320×240	R8835	Intel	
160×160	UC1698U	Intel	常用于电力集抄终端

对 motorola 时序，ioctl 的设置参数为 0；对 intel 时序设置参数为 1。设置 LCD 为 intel 时序的操作为：

```
unsigned int mode = 1;                    // 注意是 unsigned int，不能用 int!
rc = ioctl(fd, EM9X60_ISA_IOCTL_WRITE_LCDTYPE, &mode);
```

注意调用 ioctl 的参数的数据类型必须与定义完全一致。

4.3 GPIO 驱动程序

EM9160 最多可支持 16 位通用数字输入输出，即 GPIO，每位 GPIO 都可独立设置方向为输入或输出；独立设置输出电平。GPIO 上电初始化的状态均为输入，另外每位 GPIO 都带有 100K 的上拉电阻，所以在初始化状态下，每位 GPIO 管脚所呈现的电平均为高电平。Linux 应用程序若希望操作 GPIO，首先需要打开 GPIO 的设备文件：

```
fd = open("/dev/em9x60_gpio", O_RDWR);
```

然后根据 em9x60_drivers.h 中所列的 ioctl 命令进行相关操作。为了实现各位独立操作，EM9160 的 GPIO 驱动程序的 ioctl 函数使用一个 unsigned int 参数，其低 16 位 bit 分别对应 GPIO0-GPIO15，对任意 ioctl 命令，参数中对应 bit 位置 1，则命令对该位有效，否则无效。

对 GPIO 的操作可归为 6 种基本操作如下：

1、GPIO 输出使能：在任何时候 GPIO 的输入功能都是有效的。当执行了该项操作后，对应的 GPIO 位就为数字输出了，而应用程序仍然可以读取当前管脚的状态

2、GPIO 输出禁止：执行该操作后，对应 GPIO 只能作为数字输入管脚使用了

3、GPIO 输出置位：执行该操作后，对应的 GPIO 输出高电平

4、GPIO 输出清零：执行该操作后，对应的 GPIO 输出低电平

5、GPIO 三态状态：执行该操作后，对应的 GPIO 处于 OC 状态，当再对 GPIO 置位时，管脚呈三态；而对 GPIO 清零，对应 GPIO 输出低电平

6、读取 GPIO 状态：执行该操作后，返回参数的低 16 位对应各位 GPIO 当前管脚的电平状态

EM9160 的 GPIO 驱动程序为上述 6 种功能设置了对应的命令，定义如下：

```
#define EM9X60_GPIO_IOCTL_OUT_ENABLE     _IOW(EM9X60_MAGIC, 0x60, unsigned int)
#define EM9X60_GPIO_IOCTL_OUT_DISABLE    _IOW(EM9X60_MAGIC, 0x61, unsigned int)
#define EM9X60_GPIO_IOCTL_OUT_SET        _IOW(EM9X60_MAGIC, 0x62, unsigned int)
#define EM9X60_GPIO_IOCTL_OUT_CLEAR      _IOW(EM9X60_MAGIC, 0x63, unsigned int)
#define EM9X60_GPIO_IOCTL_OPEN_DRAIN     _IOW(EM9X60_MAGIC, 0x64, unsigned int)
#define EM9X60_GPIO_IOCTL_PIN_STATE      _IOR(EM9X60_MAGIC, 0x65, unsigned int)
```

具体操作 GPIO 的典型代码为：

```
unsigned int gpio_pins = GPIO1 | GPIO12;                    // 同时操作 2 位 GPIO;
rc = ioctl(fd, EM9X60_GPIO_IOCTL_OUT_SET, &gpio_pins);      // GPIO 置高电平
```

在上述操作中，对参数 gpio_pins 中没有置位的 GPIO，其状态保持不变。由于 EM9160 的部分 GPIO 管脚还复用了其他功能，如串口等。这样即使启动串口功能，驱动程序仍然可以操作其他 GPIO，而不会影响串口的功能。

4.4 矩阵键盘驱动程序

在嵌入式智能设备中，常常会用到矩阵键盘。对 EM9160 来说，最简单的矩阵键盘扩展电路就是利用 EM9160 的精简 ISA 总线，扩展一个数字输出端口，作为矩阵键盘的扫描输出；扩展一个数字输入端口，读取扫描状态。完成这样的扩展，只需要 3 片简单的 74's 系列芯片，英利公司的 ETA716 就是这样的一个扩展模块。EM9160 的矩阵键盘驱动程序就是基于 ETA716，实现 4×5 的矩阵键盘功能。

应用程序打开矩阵键盘驱动程序的方法还是打开相应的设备文件，如下：

```
fd = open("/dev/em9x60_keypad", O_RDONLY | O_NONBLOCK);
```

驱动程序一旦打开，将启动内部的扫描机制，对矩阵键盘进行周期检测，一旦有键按下，驱动程序将完成对按键的相关处理，并把得到的有效键码存储在内部的缓冲区中，键码缓冲区的深度为 16 个键值。应用程序可直接通过 read 函数读取键码。以下是一个典型的矩阵键盘测试程序：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <string.h>
#include "em9x60_drivers.h"


int main(int argc,char* argv[])
{
        int                     fd;
        int                     len;
        fd_set                  fs_read;
        int                     fs_sel = 1;
        struct timeval          time;
        KEY_CODE                KeyCode;

        printf("EM9x60 4x5 Keypad Test V1.0\r\n");
```

```
    fd = open("/dev/em9x60_keypad", O_RDONLY | O_NONBLOCK);
    if(fd < 0)
    {
        printf("can not open /dev/em9x60_keypad\n");
        return -1;
    }

    while(1)
    {
        FD_ZERO(&fs_read);
        FD_SET(fd,&fs_read);

        time.tv_sec  = 0;
        time.tv_usec = 50000;            // timeout = 50ms
        fs_sel = select(fd+1, &fs_read, NULL, NULL, &time);
        if(fs_sel)
        {
            // data available, so get it!
            len = read(fd, &KeyCode, sizeof(KeyCode));
            if(len > 0)
            {
                printf("len=%d KeyCode= 0x%x\r\n", len, KeyCode);
                if(KeyCode == 0x011b)
                {
                    break;
                }
                continue;
            }
        }
        // sleep(1);
    }

    close(fd);
    printf("close file\n");
    return 0;
}
```

ETA716 的输入输出寄存器的地址偏移量都是 0，但在实际用户产品中，很有可能需要把矩阵键盘的寄存器地址偏移量设置在其他位置上，为此 EM9160 的矩阵键盘驱动程序支持带参数加载，如输入寄存器偏移量为 2，输出寄存器偏移量为 3，加载驱动程序的命令则为：

[root@EM9x60]# **insmod em9x60_keypad.ko KinPortOffset=2 KoutPortOffset=3**

4.5 外部硬中断驱动程序

EM9160 支持 2 个独立的外部中断输入，中断信号的上升沿有效，即触发中断。其中 ISA_IRQ1（与 GPIO10 复用管脚 CN2.7#）对应设备文件“/dev/em9x60_irq1”，而 ISA_IRQ2（与 GPIO11 复用管脚 CN2.8#）对应设备文件“/dev/em9x60_irq2”。

EM9160 的中断驱动程序采用了 Linux 的异步通知的机制，即外部中断信号一旦触发中断驱动程序，驱动程序会主动向应用程序发送 SIGIO 信号（该信号为 Linux 系统预定义的信号），这样应用程序就不需要查询设备的状态，只需要简单响应 SIGIO 进行相关操作即可。在应用程序的相应函数中，可通过操作其他驱动程序的 API 函数来实现对硬件的操作，如精简 ISA 驱动或 GPIO 驱动等。有关应用程序响应中断驱动程序发出的 SIGIO 的方法，Linux 操作系统已提供了成熟的方法，即通过在应用程序初始化阶段调用：

```
Signal(SIGIO, em9x60_irq_handler);
```

来把响应函数 em9x60_irq_handler(int signum)与 Linux 信号 SIGIO 绑定。应用程序对硬件中断的具体响应操作代码则放在 em9x60_irq_handler 函数中。此外 EM9160 的中断驱动程序还在内部设置了一个中断次数的计数器，应用程序可以通过 ioctl 来读取计数值，读取后内部计数值自动清零。ioctl 命令定义如下：

```
#define EM9X60_IRQ_IOCTL_GET_COUNT    _IOR(EM9X60_MAGIC, 0x80, unsigned int)
```

以下范例程序是在 main 函数中 GPIO0 产生一个正脉冲，GPIO0 信号连接到 ISA_IRQ1 上，利用 GPIO0 的上升沿触发中断。在中断响应函数中用 GPIO1 产生一个脉冲，同时对中断计数。相关的主要代码如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>
#include "em9x60_drivers.h"

int GPIO_OutEnable(int fd, unsigned int dwEnBits);
int GPIO_OutDisable(int fd, unsigned int dwDisBits);
```

```
int GPIO_OpenDrainEnable(int fd, unsigned int dwODBits);
int GPIO_OutSet(int fd, unsigned int dwSetBits);
int GPIO_OutClear(int fd, unsigned int dwClearBits);
int GPIO_PinState(int fd, unsigned int* pPinState);

// 全局变量
int         nIrqCounter = 0;          // 中断响应函数计数
int         fd;                       // GPIO驱动文件描述符

// 接收到异步通知后的处理函数
void em9x60_irq_handler(int signum)
{
    int                 rc;

    // printf("receive a signal from /dev/em9x60_irq1,signalnum = %d\n",signum);
    nIrqCounter++;          // 中断计数

    rc = GPIO_OutSet(fd, GPIO1);
    if(rc < 0)
    {
        printf("GPIO_OutSet::failed %d\n", rc);
    }

    rc = GPIO_OutClear(fd, GPIO1);
    if(rc < 0)
    {
        printf("GPIO_OutClear::failed %d\n", rc);
    }
}

int main(int argc, char** argv)
{
    int                 rc;
    int                 i1;
    int                 irq_fd;
    int                 oflags;
    unsigned int        NumOfIrqs;
    unsigned int        SumNumOfIrqs;                   // 记录中断驱动相应的次数

    printf("Test IRQ Async Signation on EM9160\n");

    // open driver of GPIO
    fd = open("/dev/em9x60_gpio", O_RDWR);              // 打开GPIO驱动
    printf("open file = %d\n", fd);
```

```
irq_fd = open("/dev/em9x60_irq1", O_RDONLY);          // 打开中断驱动
if (irq_fd < 0)
{
        printf("can not open /dev/em9x60_irq1 device file!\n");
        close(fd);
        return -1;
}

rc = GPIO_OutEnable(fd,GPIO0 | GPIO1);                // set GPIO0 as output
if(rc < 0)
{
        printf("GPIO_OutEnable::failed %d\n", rc);
        return rc;
}

// 启动Linux异步信号通知机制
signal(SIGIO, em9x60_irq_handler);      // 让em9x60_irq_handler()处理SIGIO信号
fcntl(irq_fd, F_SETOWN, getpid());
oflags = fcntl(irq_fd, F_GETFL);
fcntl(irq_fd, F_SETFL, oflags | FASYNC);      // 使能驱动程序的异步通知功能

SumNumOfIrqs = 0;
for(i1 = 0; i1 < 1000; )
{
        // issue a positive pulse as irq pulse
        rc = GPIO_OutSet(fd, GPIO0);
        if(rc < 0)
        {
                printf("GPIO_OutSet::failed %d\n", rc);
                return rc;
        }

        rc = GPIO_OutClear(fd, GPIO0);
        if(rc < 0)
        {
                printf("GPIO_OutClear::failed %d\n", rc);
                return rc;
        }

        rc = ioctl(irq_fd, EM9X60_IRQ_IOCTL_GET_COUNT, &NumOfIrqs);
        if(rc < 0)
        {
                printf("EM9X60_IRQ_IOCTL_GET_COUNT::failed!\n");
```

```
                return rc;
            }

            SumNumOfIrqs += NumOfIrqs;
            printf("IrqSent = %d, ISRcount = %d, SignalCount = %d\n", (i1 + 1),
                SumNumOfIrqs, nIrqCounter);

        // sleep(1);            // 增加延时，测试不同频度的中断的延时情况
    }

    close(fd);
    printf("close file\n");
    return 0;
}
```

在上面的程序中，通过中断驱动内部计数值SumNumOfIrqs与应用程序响应计数值nIrqCounter的比较，可以判断中断是否有丢失。进一步用示波器观察GPIO0 和GPIO1，可以了解到Linux系统的中断延时情况：当ISA_IRQ中断周期为 1 秒时，从IRQ上升沿到应用程序响应函数的延时为 62us±3us。当中断周期加快到 250us水平，这时的延时为60us-80us。这个指标可满足绝大多数嵌入式系统的应用需求。

4.6 看门狗 WDT 驱动程序

EM9160 的看门狗驱动程序是基于CPU内部的WDT硬件单元而设计的，WDT超时时间为16 秒，当WDT发生超时时，将产生硬件的复位信号，复位EM9160，与上电复位的效果完全一样。在调试模式下，WDT硬件是关闭的；在运行模式下，WDT处于打开的模式。在应用程序没有打开WDT驱动程序时，Linux内核会按一定的时间间隔对WDT进行刷新，保证WDT不会发生超时；应用程序一旦打开WDT设备文件，Linux内核就不会再进行WDT刷新操作。应用程序可通过ioctl命令来执行对WDT的刷新操作，ioctl命令定义在em9x60_drivers.h中：

```
#define    WATCHDOG_IOCTL_BASE    'W'
#define    WDIOC_KEEPALIVE                _IOR(WATCHDOG_IOCTL_BASE, 5, int)
```

应用程序打开WDT设备文件的代码为：

```
fd = open("/dev/watchdog", O_RDONLY);
```

应用程序进行WDT刷新操作的代码为：

```
rc = ioctl(fd, WDIOC_KEEPALIVE, 0);
```

一般来讲，应用程序应在 8 秒内进行一次WDT刷新操作，以保证系统的正常运行。进行刷新操作的代码，应放在应用程序的管理线程循环中，以确保应用程序不会处于无意义运行，而WDT又不起作用。

4.7 系统配置信息驱动程序

系统配置信息驱动程序的主要作用是让应用程序能了解主板的相关硬件的情况，如板卡的类型、用户加密的ID号等等。我们会根据客户的需求不断地扩充该驱动程序的功能，为应用程序提供相关的主板管理信息。

系统配置信息驱动程序的ioctl命令也定义在em9x60_drivers.h中：

```
#define EM9X60_SYSINFO_IOCTL_GET_DBGSL          _IOR(EM9X60_MAGIC,  0x00, unsigned int)
#define EM9X60_SYSINFO_IOCTL_GET_BOARDTYPE  _IOR(EM9X60_MAGIC,  0x01, unsigned int)
```

典型的读取系统配置信息的程序如下：

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include "em9x60_drivers.h"

int main()
{
    int             fd;
    int             rc;
    unsigned int    nDBGSLState;
    unsigned int    nBoardType;

    // 打开系统配置信息设备文件
    fd = open("/dev/em9x60_sysinfo", O_RDWR);
    printf("open file = %d\n", fd);

    // 读取调试模式状态
```

```
        rc = ioctl(fd, EM9X60_SYSINFO_IOCTL_GET_DBGSL, &nDBGSLState);
        printf("ioctl rc = %d, nDBGSLState = 0x%x\n", rc, nDBGSLState);

        // 读取板卡类型编号
        rc = ioctl(fd, EM9X60_SYSINFO_IOCTL_GET_BOARDTYPE, &nBoardType);
        printf("ioctl rc = %d, nBoardType = 0x%x\n", rc, nBoardType);

        close(fd);
        printf("close file\n");
        return 0;
}
```

5 应用程序编程范例之一：LCD 显示

5.1 EM9x60 单色 LCD 显示

简述

EM9x60 主要应用之一就是可以作为智能终端的核心平台，智能终端总是带有一种显示单元。EM9x60 作为一种高效、低成本的产品解决方案，专门针对小型的单色 LCD 显示模块（分辨率通常在 128×64 至 320×240），实现了 LCD 显示驱动程序，应用程序可通过 ioctl 函数对 LCD 屏进行画点、画线的操作。LCD 设备文件名称为 “/dev/em9x60_lcd”，Linux 应用程序操作 LCD 屏时，需要首先打开 LCD 设备文件：

```
fd = open("/dev/em9x60_lcd", O_RDWR);
```

然后根据 em9x60_drivers.h 中所列的 ioctl 命令进行相关操作。EM9x60 LCD 驱动程序设置了以下命令，定义如下：

```
#define EM9X60_LCD_IOCTL_TYPE      _IOW(EM9X60_MAGIC,   0xd0, unsigned int)
#define EM9X60_LCD_IOCTL_LINE      _IOW(EM9X60_MAGIC,   0xd1, struct lcd_line)
#define EM9X60_LCD_IOCTL_BLOCK     _IOW(EM9X60_MAGIC,   0xd2, struct lcd_block)
#define EM9X60_LCD_IOCTL_CLEAR     _IO(EM9X60_MAGIC,    0xd3)
#define EM9X60_LCD_IOCTL_UPDATE    _IO(EM9X60_MAGIC,    0xd4)
```

其中用到两种数据结构，用于定义点、线、Bar 条以及数据块。

```
struct lcd_line
{
    unsigned int    type;          // = 0: point; = 1: line; = 2: bar
    unsigned int    x0;
    unsigned int    y0;
    unsigned int    x1;
    unsigned int    y1;
    unsigned int    color;         // = 0: write "0";   = 1: write "1", = 2: xor operation
};

struct lcd_block
{
    unsigned int    x0;
    unsigned int    y0;
    unsigned int    xsize;         // = 1 - 8; left alignment
    unsigned int    ysize;         // = 1 - 16
```

```
    unsigned char    data[16];        // block data to be copied
};
```

LCD API 函数的简介

为了方便客户使用，我们设计了一套通用的汉字及图形显示接口函数，这些接口函数定义和实现分别在 LCD_API.H、LCD_API.CPP 中。

以下将介绍 LCD_API.H 文件中定义的各个 LCD 操作函数，以及如何在应用程序中调用这些 API 函数，以实现对于 LCD 屏的显示操作。

EM9x60 提供的 LCD_API 函数库目前支持的 LCD 屏类型包括有：

LCD 控制器类型	显示分辨率	简要说明
KS0108	128×64	最常用的 LCD 模块
T6963C	240×128	具有较大的显示窗口
SED1335	320×240	
KS0108	192×64	
UC1698U	160×160	电力集抄终端标准显示屏

在头文件 LCD_API.H 中的对应定义如下。

```
#define LCD_12864       1       // LCD 128×64   KS0108控制器
#define LCD_240128      2       // LCD 240×128 T6963C控制器
#define LCD_320240      3       // LCD 320×240 SED1335控制器
#define LCD_19264       4       // LCD 192×64   KS0108控制器
#define LCD_160160      5       // LCD 192×64   UC1698U控制器
```

LCD_API 函数库提供了一系列对 LCD 屏进行画点、画线、画 Bar 条以及字符串（包括汉字和西文）显示的接口函数，一共包括了 13 个函数，关于各个函数的定义说明，可以参见 LCD_AP.H 头文件中的中文注释。其中的汉字支持为标准一级全汉字，为 16×16 点阵字模，西文（字母、数字和符号）为 8×14 点阵字模，这两个字模文件分别为 cclib、ascii.chr，放置在 EM9x60 工控主板的根文件系统**“/lib”**目录中。

下面以 Step1_LCDTest 为例，介绍利用 Sourcery G++工具进行应用程序开发时，如

何使用 LCD_API 函数库来实现对 LCD 的显示操作。

首先需要在应用工程项目文件 Step1_LCDTest 中导入 API 函数相关的几个文件：LCD_API.H、LCD_API.CPP 以及 DotLib.CPP（主要实现了西文、汉字显示功能）。在 Sourcery G++ IDE 下导入不仅仅是将这些文件加入到项目工程中，同时也会把这几个文件复制到项目工程所在的目录下。

在 Step1_LCDTest 中 Test_LCD12864.cpp 提供了一个调用 LCD_API 中相关函数进行 LCD 显示的示例，以下为测试代码：

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <math.h>
#include "lcd_api.h"

int main( )
{
    int    x, y;
    char        BufStr[200];

    // 对LCD屏进行初始化操作
    LCD_Init( LCD_12864 );
    // 设置为异或模式
    LCD_SetMode( 1 );       // set to XOR mode

    // 居中显示中文、字符
    strcpy( BufStr, "** 欢迎测试 **" );
    x = (128-strlen(BufStr)*8)/2;
    y = 8;
    LCD_WriteString( x, y, BufStr, TEXT_COLOR );

    strcpy( BufStr, "嵌入式网络模块" );
    x = (128-strlen(BufStr)*8)/2;
    y = 24;
    LCD_WriteString( x, y, BufStr, BK_COLOR );        // 反显
    sleep( 1 );

    // 画线
```

```
        LCD_DrawLine( 0, 42, 127, 42, TEXT_COLOR );
        LCD_DrawLine( 0, 63, 127, 63, TEXT_COLOR );

        // 画曲线
        for( x=0; x<128; x++ )
        {
            y = 53 - 10.0*sin( 2.0*M_PI*x/24.0 );
            LCD_PutPixel( x, y, TEXT_COLOR );
         }
        sleep( 1 );

        for( x=0; x<128; x++ )
        {
            y = 53 - 10.0*sin( 2.0*M_PI*x/24.0 );
            LCD_PutPixel( x, y, TEXT_COLOR );
         }
        sleep( 1 );

        LCD_FillBar( 0, 42, 127, 63, BK_COLOR );
        LCD_FillBar( 0, 42, 127, 63, TEXT_COLOR );
        LCD_FillBar( 0, 42, 127, 63, BK_COLOR );

        // 退出LCD显示
        LCD_DeInit( );
        return 0;
}
```

5.2 EM9161 彩色 LCD 显示

EM9161 其硬件核心为工业级的 ARM9 芯片 AT91SAM9261，该 CPU 带有 LCD 图形驱动控制器，可以支持 TFT 彩色液晶 LCD 显示，所能支持的最大显示分辨率为 800×600。常用的液晶 LCD 有：

名称	型号	分辨率
4.3" TFT 彩色 LCD	LR430	480×272
5.6" TFT 彩色 LCD	AT056TN52	640×480
7.0" TFT 彩色 LCD	CLAA070L	800×480
8.4" TFT 彩色 LCD	G084SN03	800×600
10.4" TFT 彩色 LCD	G104SN03	800×600

简述

在 Linux 操作系统中 ，系统专门为显示设备提供了一个帧缓冲（framebuffer）的接口，它将显示缓冲区抽象，屏蔽图像硬件的底层差异，允许上层应用程序在图形模式下直接对显示缓冲去进行读写操作，应用程序只需在显示缓冲区(framebuffer)中与对应的区域写入颜色值，对应的颜色会自动在屏幕上显示。

在应用程序中获取 framebuffer 的方式还是通过打开显示设备文件，在 EM9161 中其设备文件名称为"/dev/fb0"，Linux 应用程序操作 LCD 屏时，一般为以下几个步骤：

1、打开 LCD 设备文件：

```
fd = open("/dev/fb0", O_RDWR);
```

2、调用 ioct 函数获取当前屏幕的参数，如屏幕分辨率、每个像素点的比特数和偏移。应用程序可根据这些参数计算出屏幕缓冲区的大小

3、将屏幕缓冲区映射到用户空间

4、应用程序就可利用映射后的空间进行直接读/写缓冲区的操作，进行绘图和图片显示

```
// open lcd_fd
lcd_fd = open("/dev/fb0", O_RDWR );
if( !lcd_fd )
{
        printf( "Can not open framebuffer devic\n");
        return -1;
}

// get screen information
if( ioctl(lcd_fd, FBIOGET_VSCREENINFO, &lcd_info))
{
        printf( "GET_VSCREENINFO error\n");
        return -2;
}

printf("screen:%dx%d %dbpp\n", lcd_info.xres, lcd_info.yres, lcd_info.bits_per_pixel );

// screen size in bytes
lcd_size = lcd_info.xres * lcd_info.yres * 2;

// map the device to memory
lcd_fbp = (char*)mmap( 0, lcd_size, PROT_READ|PROT_WRITE, MAP_SHARED, lcd_fd, 0 );
if( (int)lcd_fbp==-1 )
{
        printf( "map framebuffer device to memory failed \n" );
```

```
        return -3;
    }
```

如上述代码所示，当获取到映射的 lcd_fbp 指针时，应用程序就可以直接操作 lcd_fbp 来实现对于 LCD 显示的操作。

LCD API 函数的简介

为了方便客户使用，我们设计了一套通用的汉字及图形显示接口函数，这些接口函数定义和实现分别在 lcd_graph.h lcd_graph.cpp 中，函数中包括画点、画线、画圆、矩形框、输出汉字、输出字符、显示 bmp 文件等操作。关于各个函数的定义说明，可以参见 lcd_graph.h 头文件中的中文注释。其中的汉字支持为标准一级全汉字，为 16×16 点阵字模，西文（字母、数字和符号）为 8×14 点阵字模，这两个字模文件分别为 cclib、ascii.chr，放置在 EM9161 嵌入式主板的根文件系统“**/lib**”目录中。

以 step1_lcdtest 为例，介绍利用 Sourcery G++工具进行应用程序开发时，如何使用 lcd_graph.h 函数库来实现对 LCD 的显示操作。

首先需要在应用工程项目文件 step1_lcdtest 中导入 API 函数相关的几个文件：lcd_graph.h、lcd_graph.cpp 以及 DotLib.CPP（主要实现了西文、汉字显示功能）。在 Sourcery G++ IDE 下导入不仅仅是将这些文件加入到项目工程中，同时也会把这几个文件复制到项目工程所在的目录下。以下为测试代码：

```
    i1 = initlcd( );

    MaxX = getmaxx();
    MaxY = getmaxy();

    // 显示中文、字符操作
    strcpy( BufStr, "** 欢迎测试**" );
    x = (MaxX-strlen(BufStr)*8)/2;
    y = 8;
    outchinesexy( x, y, BufStr );

    setcolor( rgb[BLACK] );
    setbkcolor( rgb[LIGHTCYAN] );
```

```
strcpy( BufStr, "嵌入式Linux主板" );
x = (MaxX-strlen(BufStr)*8)/2;
y = 24;
outchinesexy( x, y, BufStr );
sleep( 1 );
setbkcolor( rgb[BLACK] );

// 画线
setcolor( rgb[WHITE] );
line( 0, 42, MaxX-1, 42 );
line( 0, 63, MaxX-1, 63 );

// 画曲线
for( x=0; x<MaxX; x++ )
{
        y = 53 - 10.0*sin( 2.0*M_PI*x/24.0 );
        putpixel( x, y, rgb[LIGHTGREEN] );
 }
sleep( 1 );

setwritemode( XOR_PUT );
for( x=0; x<MaxX; x++ )
{
        y = 53 - 10.0*sin( 2.0*M_PI*x/24.0 );
        putpixel( x, y, rgb[LIGHTGREEN] );
 }
sleep( 1 );
setwritemode( COPY_PUT );

clearscreen( rgb[BLACK]);

line( 0, 0, MaxX-1, MaxY-1 );
line( 0, MaxY-1, MaxX-1, 0 );
setcolor( rgb[YELLOW]);

for( i1=0; i1<6; i1++)
{
        circle( MaxX/2-1, MaxY/2-1, 40+i1*20 );
}

setcolor( rgb[LIGHTMAGENTA]);
// setfnt( Font_16 );
strcpy( BufStr, "字符串点阵测试" );
x = (MaxX-strlen(BufStr)*8)/2;
```

```
y = 10;
outchinesexy( x, y, BufStr );

setfnt( Font_12 );
strcpy( BufStr, "字符串点阵测试" );
x = (MaxX-strlen(BufStr)*8)/2 + 8;
y = 30;
outchinesexy( x, y, BufStr );

setfnt( Font_5X7 );
strcpy( BufStr, "ASCII 5x7font test" );
x = (MaxX-strlen(BufStr)*6)/2 + 8;
y = 50;
outtextxy( x, y, BufStr );

sleep(2);
setcolor( rgb[LIGHTRED]);
setwritemode( XOR_PUT );
start = time(NULL);
// strcpy( BufStr, "欢迎测试嵌入式模块ABCDEFGHIJKL" );
for( i1=0; i1<9; i1++)
{
        bar( MaxX/2-30, MaxY/2-30, MaxX/2+30, MaxY/2+30 );
        sleep(1);
}

end = time(NULL);
printf("The pause used %f seconds.\n",difftime(end,start)/10);//

deinitlcd( );
```

6 应用程序编程范例之二：串口通讯

EM9160 提供了 6 个标准异步串口：ttyS1、ttyS2、ttyS3、ttyS4、ttyS5、ttyS6，其中 ttyS4、ttyS5、ttyS6 和 GPIO 的管脚复用，每个串口都有独立的中断模式，使得多个串口能够同时实时进行数据收发。各个串口的驱动已经包含在 Linux 操作系统的内核中，EM9160 在 Linux 系统启动完成时，各个串口已作为字符设备完成了注册加载，用户的应用程序可以以操作文件的方式对串口进行读写，从而实现数据收发的功能。

6.1 串口编程接口函数

在 Linux 中，所有的设备文件都位于“/dev”目录下，EM9160 上六个串口所对应的设备名依次为：“/dev/ttyS1”、“/dev/ttyS2”、“/dev/ttyS3”、“/dev/ttyS4”、“/dev/ttyS5”、“/dev/ttyS6”。

在 Linux 下操作设备的方式和操作文件的方式是一样的，调用 open( )打开设备文件，再调用 read( )、write( )对串口进行数据读写操作。这里需要注意的是打开串口除了设置普通的读写之外，还需要设置 O_NOCTTY 和 O_NDLEAY，以避免该串口成为一个控制终端，有可能会影响到用户的进程。如：

```
sprintf( portname, "/dev/ttyS%d", PortNo );              //PortNo为串口端口号，从1开始
m_fd = open( portname,O_RDWR | O_NOCTTY | O_NONBLOCK);
```

作为串口通讯还需要一些通讯参数的配置，包括波特率、数据位、停止位、校验位等参数。在实际的操作中，主要是通过设置 struct termios 结构体的各个成员值来实现，一般会用到的函数包括：

```
tcgetattr( ) ;
tcflush( );
cfsetispeed( );
cfsetospeed( );
tcsetattr( );
```

其中各个函数的具体使用方法这里就不一一介绍了，用户可以参考 Linux 应用程序开发的相关书籍，也可参看 Step2_SerialTest 中 Serial.cpp 模块中 set_port( )函数代码。

6.2 串口综合应用示例

Step2 _SerialTest 是一个支持异步串口数据通讯的示例，该例程采用了面向对象的 C++编程，把串口数据通讯作为一个对象进行封装，用户调用该对象提供的接口函数即可方便地完成串口通讯的操作。

CSerial 类介绍

利用上一小节中介绍的串口 API 函数，封装了一个支持异步读写的串口类 CSerial，CSerial 类中提供了 4 个公共函数、一个串口数据接收线程以及数据接收用到的数据 Buffer。

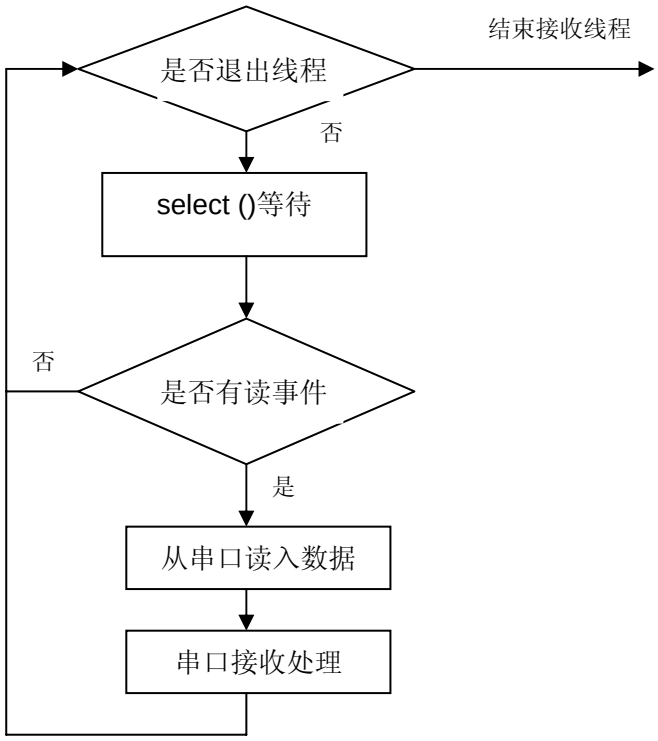
```
class CSerial
{
private:
    //通讯线程标识符ID
    pthread_t  m_thread;
    // 串口数据接收线程
    static int ReceiveThreadFunc( void* lparam );
public:
    CSerial();
    virtual ~CSerial();

    int          m_fd;                  // 已打开的串口文件描述符
    int     m_DatLen;
    char         DatBuf[1500];
    int     m_ExitThreadFlag;

    // 按照指定的串口参数打开串口，并创建串口接收线程
    int OpenPort( int PortNo, int baudrate, char databits, char stopbits, char parity );
    // 关闭串口并释放相关资源
    int ClosePort( );
    // 向串口写数据
    int WritePort( char* Buf, int len );
    // 接收串口数据处理函数
    virtual int PackagePro( char* Buf, int len );
};
```

OpenPort 函数用于根据输入串口参数打开串口，并创建串口数据接收线程。在 Linux 环境中是通过函数 pthread_create( )创建线程，通过函数 pthread_exit( )退出线程。Linux

线程属性存在有非分离（缺省）和分离两种，在非分离情况下，当一个线程结束时，它所占用的系统资源并没有被释放，也就是没有真正的终止；只有调用 pthread_join( )函数返回时，创建的线程才能释放自己占有的资源。在分离属性下，一个线程结束时立即释放所占用的系统资源。基于这个原因，在我们提供的例程中通过相关函数将数据接收线程的属性设置为分离属性。如：

```
// 设置线程绑定属性
res = pthread_attr_setscope( &attr, PTHREAD_SCOPE_SYSTEM );
// 设置线程分离属性
res += pthread_attr_setdetachstate( &attr, THREAD_CREATE_DETACHED );
```

ReceiveThreadFunc 函数是串口数据接收和处理的主要核心代码，在该函数中调用 select( )，阻塞等待串口数据的到来。对于接收到的数据处理也是在该函数中实现，在本例程中处理为简单的数据回发，用户可结合实际的应用修改此处代码，修改 PackagePro( )函数即可。流程如下：




```
int CSerial::ReceiveThreadFunc(void* lparam)
{
    CSerial *pSer = (CSerial*)lparam;

    //定义读事件集合
    fd_set fdRead;
```

```
    int ret;
        struct timeval    aTime;

        while( 1 )
        {
            //收到退出事件，结束线程
            if( pSer->m_ExitThreadFlag )
            {
                break;
            }
            FD_ZERO(&fdRead);
            FD_SET(pSer->m_fd,&fdRead);
            aTime.tv_sec = 0;
            aTime.tv_usec = 300000;
            ret = select( pSer->m_fd+1,&fdRead,NULL,NULL,&aTime );
            if (ret < 0 )
            {
                //关闭串口
                pSer->ClosePort( );
                break;
            }
            if (ret > 0)
            {
                //判断是否读事件
                if (FD_ISSET(pSer->m_fd,&fdRead))
                {
                    //data available, so get it!
                    pSer->m_DatLen = read( pSer->m_fd, pSer->DatBuf, 1500 );
                    // 对接收的数据进行处理，这里为简单的数据回发
                    if( pSer->m_DatLen > 0 )
                    {
                        pSer->PackagePro( pSer->DatBuf, pSer->m_DatLen);
                    }
                    // 处理完毕
                }
            }
        }
        printf( "ReceiveThreadFunc finished\n");
        pthread_exit( NULL );
        return 0;
    }
```

需要注意的是，select( )函数中的时间参数在 Linux 下，每次都需要重新赋值，否则会自动归 0。

CSerial 类的实现代码请参见 Serial.CPP 文件。

CSerial 类的调用

CSerial 类的具体使用也比较简单，主要是对于类中定义的 4 个公共函数的调用，以下为 Step2_SerialTest.cpp 中相关代码。

```
class CSerial    m_Serial;
int main( int argc,char* argv[] )
{
    int         i1;
    int          portno,    baudRate;
    char        cmdline[256];

    printf( "Step2_SerialTest V1.0\n" );
    // 解析命令行参数：串口号    波特率
    if( argc > 1 )          strcpy( cmdline, argv[1] );
    else                            portno = 1;
    if( argc > 2 )
    {
        strcat( cmdline, " " );
        strcat( cmdline, argv[2] );
        scanf( cmdline, "%d %d", &portno, &baudRate );
    }
    else
    {
        baudRate = 115200;
    }
    printf( "port:%d baudrate:%d\n", portno, baudRate);
    //打开串口相应地启动了串口数据接收线程
    i1 = m_Serial.OpenPort( portno, baudRate, '8', '1', 'N');
    if( i1<0 )
    {
        printf( "serial open fail\n");
        return -1;
    }
    //进入主循环，这里每隔1s输出一个提示信息
    for( i1=0; i1<10000;i1++)
    {
        sleep(1);
        printf( "%d \n", i1+1);
    }
```

```
    m_Serial.ClosePort( );
    return 0;
}
```

从上面的代码可以看出，程序的主循环只需要实现一些管理性的功能，在本例程中仅仅是每隔 1s 输出一个提示信息，在实际的应用中，可以把一些定时查询状态的操作、看门狗的喂狗等操作放在主循环中，这样充分利用了 Linux 多任务的编程优势，利用内核的任务调度机制，将各个应用功能模块化，以便于程序的设计和管理。这里顺便再提一下，在进行多个串口编程时，也可以利用本例程中的 CSerial 类为基类，根据应用需求派生多个 CSerial 派生类实例，每一个派生类只是重新实现虚函数 PackagePro(…)，这样每个串口都具有一个独立的串口数据处理线程，利用 Linux 内核的任务调度机制以实现多串口通讯功能。

Step2_SerialTest 的编译设置

在该例程中用到了线程操作函数，由于线程库不是缺省库，Sourcery G++编译可以通过，但是link会出错，需要配置Sourcery G++ IDE的编译参数，Linker链接中增加线程库。在Project Explorer视窗下，选择Step2_SerialTest工程文件，然后点击鼠标右键，选择**Properties**项，在窗口中选择**C/C++ Build** -> **Settings** -> **Tool Settings** -> **Sourcery G++ C++ Linker** -> **Libraries**，如图 5-2 所示。其中的一个窗口用于指定库文件的名称，一个用于指定库文件的路径，对于系统中已有的线程库lpthread文件，就不需要指定路径。

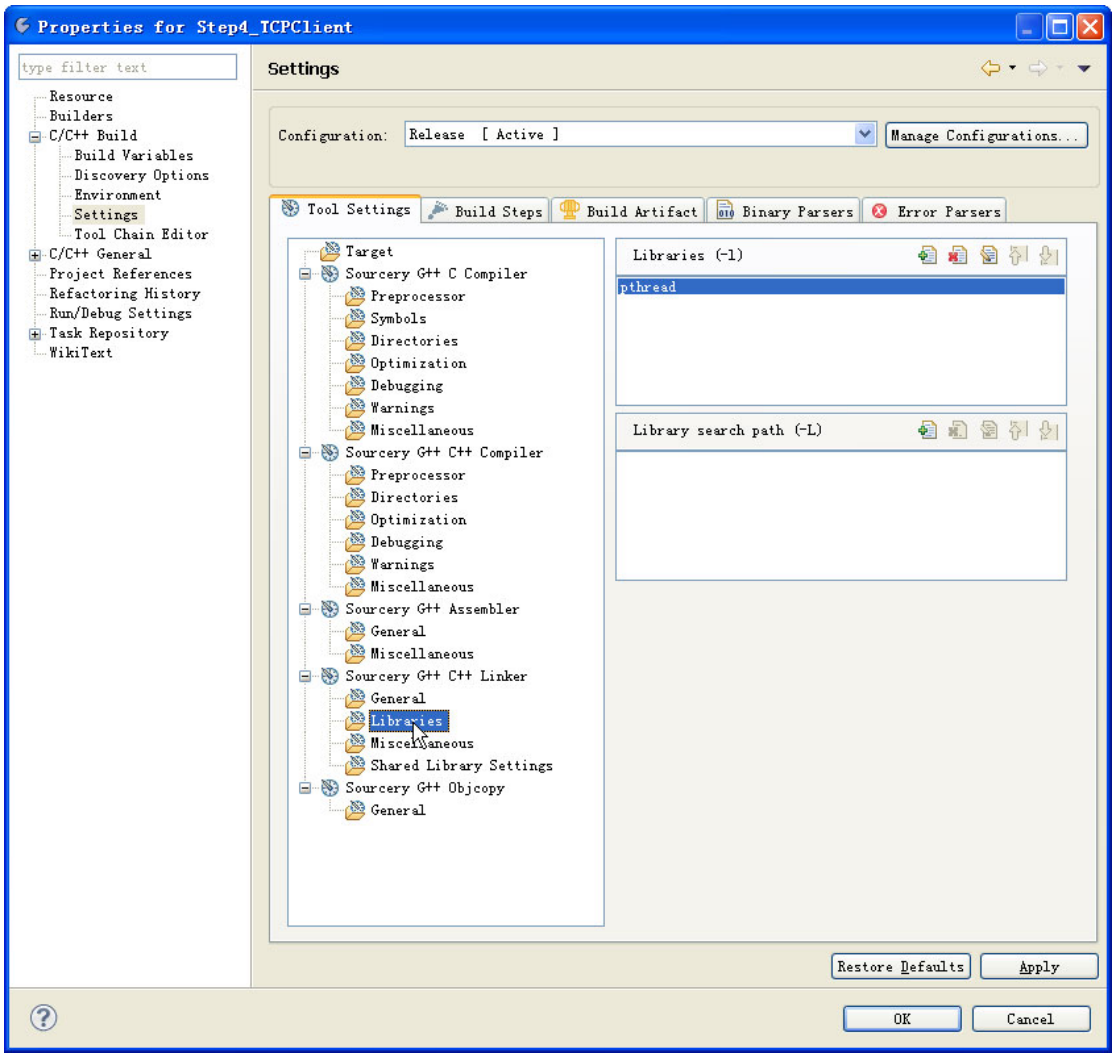

Properties for Step4_TCPClient
type filter text
Resource
Builders
C/C++ Build
Build Variables
Discovery Options
Environment
Settings
Tool Chain Editor
C/C++ General
Project References
Refactoring History
Run/Debug Settings
Task Repository
WikiText
Settings
Configuration: Release [Active]
Manage Configurations...
Tool Settings
Build Steps
Build Artifact
Binary Parsers
Error Parsers
Target
Sourcery G++ C Compiler
Preprocessor
Symbols
Directories
Optimization
Debugging
Warnings
Miscellaneous
Sourcery G++ C++ Compiler
Preprocessor
Directories
Optimization
Debugging
Warnings
Miscellaneous
Sourcery G++ Assembler
General
Miscellaneous
Sourcery G++ C++ Linker
General
Libraries
Miscellaneous
Shared Library Settings
Sourcery G++ Objcopy
General
Libraries (-l)
pthread
Library search path (-L)
Restore Defaults
Apply
OK
Cancel

图 6-1 链接库文件

7 应用程序编程范例之三：TCP 服务器

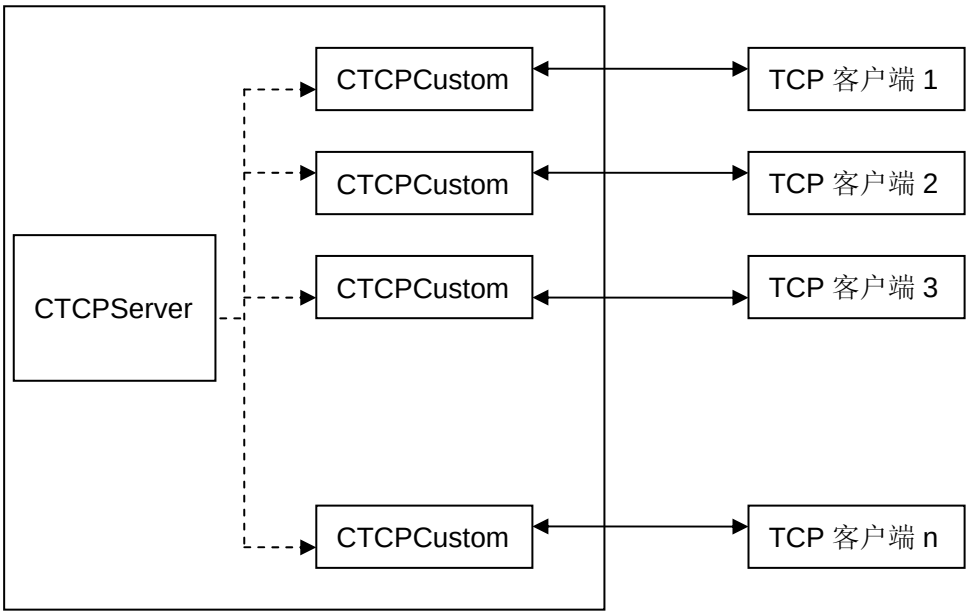
以太网在嵌入式领域的应用非常广泛，在工控领域中比较常见的就是利用 TCP/IP 协议进行数据通讯。EM9160 具有 10M/100M 自适应网络接口，非常适用于作为网络应用的开发平台。在网络应用中，编程显得尤为重要，在本章主要介绍 EM9160 作为 TCP 服务器方式的应用——支持多连接的 TCP 服务器示例程序：Step3_TCPServer。

7.1 TCP Socket 编程

在进行网络应用程序开发方面大多是采用套接字 Socket 技术，Linux 的系统平台上也是如此。Socket 编程的基本函数有 socket( )、bind( )、listen( )、accept( )、send( )、sendto( )、recv( )、recvfrom( )、connect( )等，各个函数的具体使用方法这里就不一一介绍了，用户可以参考 Linux 应用程序开发的相关书籍。

7.2 支持多连接的 TCP 服务器应用示例

Step3_TCPServe 是一个支持多个客户端的连接 TCPServer 示例，该例程采用了面向对象的 C++编程，创建了 CTCPServer 和 CTCPCustom 两个类，其中 CTCPServer 类负责侦听客户端的连接，一旦有客户端请求连接，它就负责接受此连接，并创建一个新的 CTCPCustom 类对象与客户端进行通讯，然后 CTCPServer 类接着监听客户端的连接请求，其流程如下：



CTCPServer 类

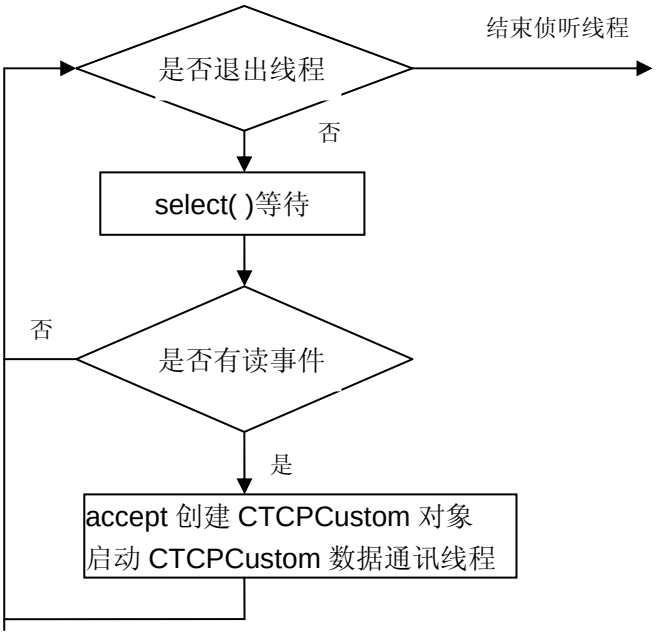
CTCPServer 类定义在 TCPServer.h 文件下，该类提供了 3 个公共函数，以及一个 Socket 侦听线程，公共的函数中 Open( )、Close( )用于启动或是关闭 TCP 服务。

```
class CTCPServer
{
private:
    pthread_t   m_thread;                          // 通讯线程标识符ID
    // Socket侦听线程
    static int SocketListenThread( void*lparam );
public:
    int             m_sockfd;                      // TCP服务监听socket
    int             m_ExitThreadFlag;
    int             m_LocalPort;                   // 设置服务端口号
    CTCPServer();
    virtual ~CTCPServer();
    int Open();                                    // 打开TCP服务
    int Close();                                   // 关闭TCP服务
    // 删除一个客户端对象连接 释放资源
    int RemoveClientSocketObject( void* lparam );
};
```

在Open( )函数中实现了打开套接字，将套接字设置为侦听套接字，并创建侦听客户端连接线程。在Linux应用程序中创建线程的方法在“6.2 串口综合应用示例”中有相关的说明，

在该例程中也是采取的同样方式。

SocketListenThread函数中调用select( )侦听客户端的TCP连接，流程如下：



同样的需要注意的是，select( )函数中的时间参数在 Linux 下每次都需要重新赋值，否则会自动归 0。CTCPServer 类的实现代码请参见 TCPServer.CPP 文件。

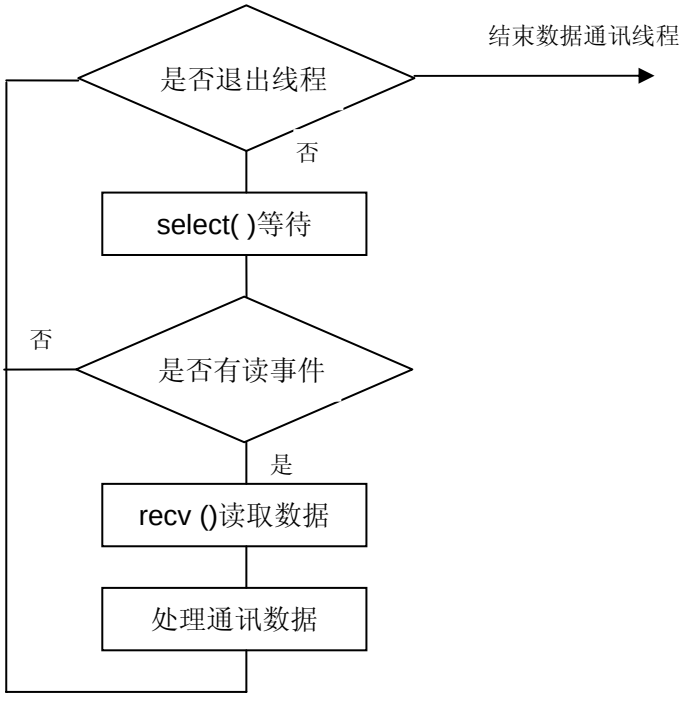
CTCPCustom类

CTCPCustom的定义在TCPCustom.h文件下。

```
class CTCPCustom
{
public:
	CTCPCustom();
	virtual ~CTCPCustom();
public:
	char	m_RemoteHost[100];			// 远程主机IP地址
	int	m_RemotePort;				// 远程主机端口号
	int	m_socketfd;				// 通讯socket
	int	m_SocketEnable;
	int	m_ExitThreadFlag;
	CTCPServer*	m_pTCPServer;
private:
```

```
        // 通讯线程函数
        pthread_t   m_thread;                                   // 通讯线程标识符ID
        static void* SocketDataThread(void* lparam);        // TCP连接数据通讯线程
    public:
        int RecvLen;
        char RecvBuf[1500];
        // 打开socket，创建通讯线程
        int Open(void* lparam);
        // 关闭socket，关闭线程，释放Socket资源
        int Close();
       // 向客户端发送数据
        int SendData(const char * buf , int len );
    };
```

其中的 SocketDataThread 函数是实现 TCP 连接数据通讯的核心代码，在该函数中调用 select( )等待 TCP 连接的通讯数据，对于接收的 TCP 连接数据的处理也是在该函数中实现，在本例程中处理为简单的数据回发，用户可结合实际的应用修改此处代码，流程如下：



CTCPServer 类的调用

CTCPSerer 类的具体使用也比较简单，主要是调用对于类中定义 Open 函数来启动各个 TCP 通讯线程，反而在主循环中需要实现的功能代码不多了，在本例程中仅仅为每隔 1s 输出提示信息。以下为 Step3_TCPServer.cpp 中的相关代码。

```
class CTCPServer    m_TCPServer;
int main()
{
        int i1;
        printf( "Step3_TCPTest V1.0\n" );
        // 给TCP服务器端口赋值
        m_TCPServer.m_LocalPort = 1001;
        // 创建Socket，启动TCP服务器侦听线程
        i1 = m_TCPServer.Open( );
        if( i1<0 )
        {
                printf( "TCP Server start fail\n");
                return -1;
        }
        // 进入主循环，主要是负责管理工作
        for( i1=0; i1<10000;i1++)                    // 实际应用时，可设置为无限循环
        {
                sleep(1);
                printf( "%d \n", i1+1);
        }
        m_TCPServer.Close( );
        return 0;
}
```

Step3_TCPServer 的编译设置

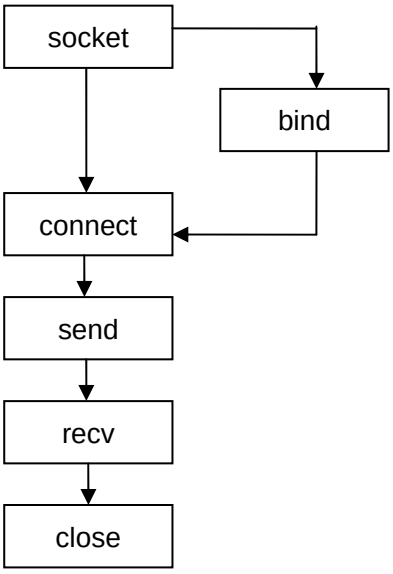
由于使用到 Linux 的线程功能，因此本实例与 Step2_SerialTest 实例程序一样，需要对缺省的编译设置进行修改，具体方法请参见“6.2　串口综合应用示例”相关内容。

8 应用程序编程范例之四：TCP 客户端

本章主要介绍 EM9160 作为 TCP 客户端方式的应用示例 Step4_TCPClient。

8.1 TCP 客户端 Socket 编程流程

在利用 Socket 进行 TCP 客户端编程时，建立 TCP 连接的过程一般比较简单，首先客户端调用 socket()函数建立流式套接字，然后调用 connect()函数请求服务器端建立 TCP 连接，成功建立连接后即可与服务器端进行 TCP/IP 数据通讯，流程如下：



8.2 TCPClient 应用示例

Step4_TCPClient 是一个具有自动管理功能的 TCP 客户端应用示例。作为 TCP 客户端主动和服务器端建立 TCP 连接的过程编程相对简单，直接调用相关的 Socket 函数即可，建立 TCP 连接的功能封装在 CTCPClient 类中。嵌入式的应用场合大多是处于长期运行无人值守的状态，可能会遇到需要一直保持 TCP 客户端连接的情况，Step4_TCPClient 例程基于这种需求，专门封装了一个 CTCPClientManager 管理类对 TCPClient 的连接进行自动管理，包括启动建立 TCP 的客户端连接、查询 TCP 连接的状态、添加多个 TCP 客户端连

接等功能。

CTCPClient 类

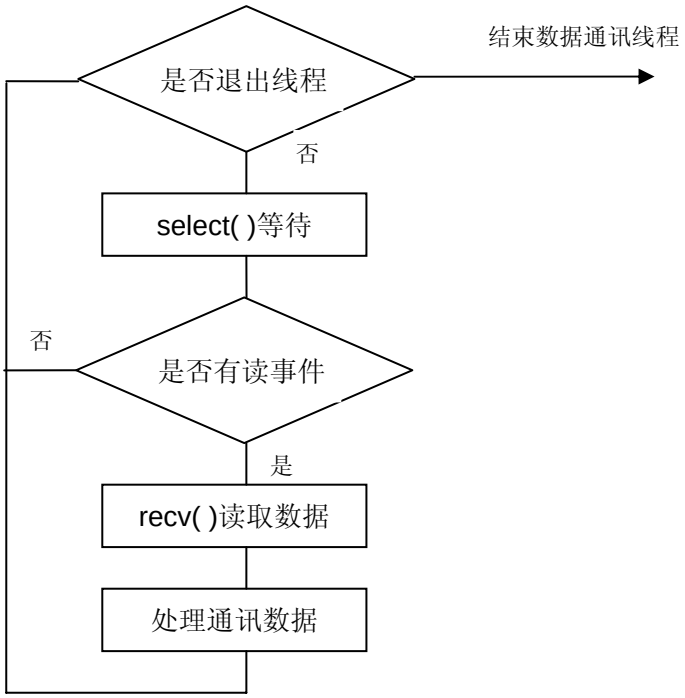
CTCPClient 类定义在 TCPClient.H 文件下，该类提供了 4 个公共函数，以及一个数据通讯线程，调用该类中的相关函数与 TCP 服务器端建立连接。

```
class CTCPClient
{
private:
    pthread_t   m_thread;                                   // 通讯线程标识符ID
    // 数据通讯处理线程函数
    static int SocketThreadFunc( void*lparam );
public:
    // TCP通讯Socket
    int             m_sockfd;
    int             m_sockclose;
    int             m_ExitThreadFlag;
    // 远程主机IP地址
    char        m_remoteHost[255];
    // 远程主机端口
    int         m_port;
    char RecvBuf[1500];
    int    m_nRecvLen;

public:
    CTCPClient();
    virtual ~CTCPClient( );
    // 打开创建客户端socket
    int Open( char* ServerIP, int ServerPort );
    // 关闭客户端socket
    int Close( );
    // 与服务器端建立连接  并创建数据通讯处理线程
    int Connect();
    // 向服务器端发送数据
    int SendData( char * buf , int len);
};
```

Open 函数执行创建打开 socket 操作，并设置远端 TCP 服务器的 IP 和端口。

Connect 函数调用 connect( )与远端 TCP 服务器建立连接，调用 select( )等待 TCP 连接的建立，TCP 连接建立成功，则创建 TCP 数据通讯处理线程。

SocketThreadFunc 函数是实现 TCP 连接数据通讯的核心代码，在该函数中调用 select( )，等待 TCP 连接的通讯数据，对于接收的 TCP 连接数据的处理也是在该函数中实现，在本例程中处理为简单的数据回发，用户可结合实际的应用修改此处代码。流程如下：



CTCPClientManager 类

TCP 客户端连接定义为四个状态：

```
enum   CONNSTATE{ csWAIT, csINIT, csCLOSED, csOPEN };
```

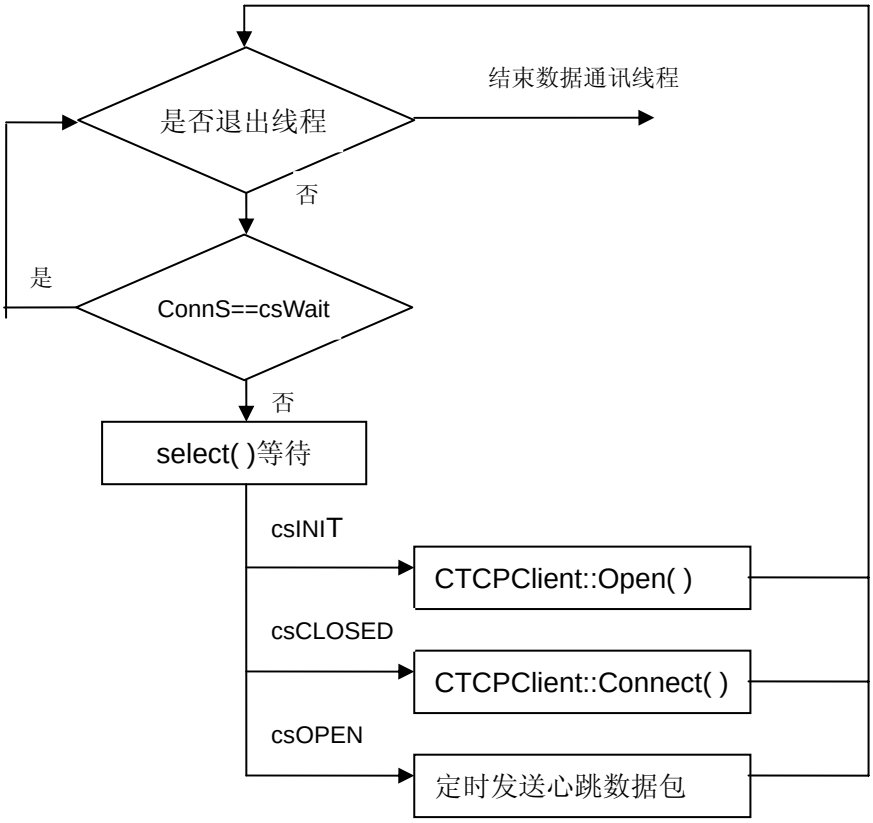
其中 csOPEN 表明 TCP 客户端连接建立。

CTCPClientManager 所封装的功能函数就是根据连接的各个状态对 TCP 客户端连接进行管理，CTCPClientManager 类定义在 TCPClientManager.H 文件下。

```
class CTCPClientManager
{
private:
    // TCPClient连接管理线程
    static int TCPClientThreadFunc( void* lparam );
public:
```

```
    TCPCLIENT_INFO          m_TCPClientInfo[TCPCLIENT_MAX_NUM];
    pthread_t               m_thread[TCPCLIENT_MAX_NUM];
    int                     m_nTCPClientNum;
public:
    CTCPClientManager( );
    ~CTCPClientManager();
    // 添加TCP客户端连接对象，输入参数为TCP服务器的IP和端口
    int AddTCPClientObject( char* pHostIP, int nHostPort );
    // 删除所有TCP客户端连接对象
    int DeleteAllTCPClient( );
    // 设置TCP客户端连接对象为csINIT状态
    int Open( int Idx );
    // 获取TCP客户端连接状态
    int   GetTCPClientState( int Idx );
    // 启动TCPClient连接管理操作，并创建TCPClient连接管理线程
    int Start( );
    // 关闭TCPClient连接管理操作
    int Stop( );
};
```

TCPClientThreadFunc 函数是实现对 TCP 连接状态管理操作的核心代码，由于 Linux 下 sleep 的最小单位为秒，对于毫秒级的延时等待，在该函数中利用调用 select( )设置相关的时间参数来实现。流程如下：



CTCPClientManager 类的调用

CTCPClientManager 类的具体使用过程：首先调用类中定义 AddTCPClientObject 加载 TCP 连接对象，然后调用类中定义 Start 函数来启动 TCP 连接自动管理线程，Open 函数将 TCP 连接状态设置为 csINIT 状态。本例程中主循环的操作为每隔 1s 查询 TCPClient 连接的状态，如果状态为 csWait，程序调用 Open 函数将其设置为 csINIT 状态，则 TCPC 连接管理线程将自动进行与 TCP 服务器端建立连接的操作。

以下为 Step4_TCPClient.cpp 中的相关代码。

```
class CTCPClientManager        TCPClntManager;
int main()
{
      int i1, i2, i3;
      // 添加一个TCP客户端连接对象
```

```
        TCPClntManager.AddTCPClientObject( "192.168.201.121", 1001 );
        // 启动TCPClient连接管理操作，并创建TCPClient连接管理线程
        TCPClntManager.Start();
        for( i1=0; i1<TCPClntManager.m_nTCPClientNum; i1++ )
        {
            // 设置TCP客户端连接初始状态，连接管理线程将自动进行TCP的连接操作
            TCPClntManager.Open(i1);
        }
        for(i1=0; i1<10000; i1++)
        {
            sleep(1);
            for( i2=0; i2<TCPClntManager.m_nTCPClientNum; i2++ )
            {
                // 查询TCP客户端连接状态
                i3 = TCPClntManager.GetTCPClientState(i2);
                printf( "TCP Connect%d State: %d \n", i2+1, i3 );
                if( i3==0 )
                {
                // 设置TCP客户端连接初始状态，连接管理线程将自动进行TCP连接操作
                    TCPClntManager.Open( i2 );
                }
            }
        }
        return 0;
}
```

附录 1 版本信息管理表

日期	版本	简要说明
2010 年 4 月	V1.0	创建本文档
2011 年 4 月	V2.0	增加对 EM9161 彩色显示说明