目 录

[第1章 SylixOS开发基础知识 1](#_Toc469063996)

[1.1 SylixOS介绍 1](#_Toc469063997)

[1.2 RealEvo-IDE介绍 1](#_Toc469063998)

[1.3 RealEvo IDE常用工程介绍 1](#_Toc469063999)

[1.4 建立SylixOS Base工程 2](#_Toc469064000)

[1.5 SylixOS第一个应用程序helloworld 5](#_Toc469064001)

[1.6 RealEvo-IDE工程导入 7](#_Toc469064002)

[第2章 ARM汇编基础实验 9](#_Toc469064003)

[实验一 ARM汇编编程实验 10](#_Toc469064004)

[实验目的 10](#_Toc469064005)

[实验内容 10](#_Toc469064006)

[实验环境 10](#_Toc469064007)

[实验原理 10](#_Toc469064008)

[实验步骤 13](#_Toc469064009)

[扩展实验 13](#_Toc469064010)

[第3章 文件操作实验 14](#_Toc469064011)

[实验一 shell命令操作文件实验 15](#_Toc469064012)

[实验目的 15](#_Toc469064013)

[实验内容 15](#_Toc469064014)

[实验环境 15](#_Toc469064015)

[实验原理 15](#_Toc469064016)

[实验步骤 15](#_Toc469064017)

[扩展实验 17](#_Toc469064018)

[实验二 文件操作实验 18](#_Toc469064019)

[实验目的 18](#_Toc469064020)

[实验内容 18](#_Toc469064021)

[实验环境 18](#_Toc469064022)

[实验原理 18](#_Toc469064023)

[实验步骤 19](#_Toc469064024)

[扩展实验 20](#_Toc469064025)

[实验三 文件目录操作实验 21](#_Toc469064026)

[实验目的 21](#_Toc469064027)

[实验内容 21](#_Toc469064028)

[实验环境 21](#_Toc469064029)

[实验原理 21](#_Toc469064030)

[实验步骤 22](#_Toc469064031)

[扩展实验 23](#_Toc469064032)

[第4章 时间操作实验 24](#_Toc469064033)

[实验一 shell命令时间操作实验 25](#_Toc469064034)

[实验目的 25](#_Toc469064035)

[实验内容 25](#_Toc469064036)

[实验环境 25](#_Toc469064037)

[实验原理 25](#_Toc469064038)

[实验步骤 25](#_Toc469064039)

[扩展实验 26](#_Toc469064040)

[实验二 时间操作实验 27](#_Toc469064041)

[实验目的 27](#_Toc469064042)

[实验内容 27](#_Toc469064043)

[实验环境 27](#_Toc469064044)

[实验原理 27](#_Toc469064045)

[实验步骤 29](#_Toc469064046)

[扩展实验 29](#_Toc469064047)

[第5章 多线程实验 30](#_Toc469064048)

[实验一 线程管理实验 31](#_Toc469064049)

[实验目的 31](#_Toc469064050)

[实验内容 31](#_Toc469064051)

[实验环境 31](#_Toc469064052)

[实验原理 31](#_Toc469064053)

[实验步骤 33](#_Toc469064054)

[扩展实验 33](#_Toc469064055)

[实验二 线程同步实验—信号量 34](#_Toc469064056)

[实验目的 34](#_Toc469064057)

[实验内容 34](#_Toc469064058)

[实验环境 34](#_Toc469064059)

[实验原理 34](#_Toc469064060)

[实验步骤 37](#_Toc469064061)

[扩展实验 38](#_Toc469064062)

[实验三 线程同步实验—互斥锁 39](#_Toc469064063)

[实验目的 39](#_Toc469064064)

[实验内容 39](#_Toc469064065)

[实验环境 39](#_Toc469064066)

[实验原理 39](#_Toc469064067)

[实验步骤 41](#_Toc469064068)

[扩展实验 41](#_Toc469064069)

[实验四 线程同步实验—条件变量 42](#_Toc469064070)

[实验目的 42](#_Toc469064071)

[实验内容 42](#_Toc469064072)

[实验环境 42](#_Toc469064073)

[实验原理 42](#_Toc469064074)

[实验步骤 44](#_Toc469064075)

[扩展实验 45](#_Toc469064076)

[第6章 多进程实验 46](#_Toc469064077)

[实验一 进程管理实验 47](#_Toc469064078)

[实验目的 47](#_Toc469064079)

[实验内容 47](#_Toc469064080)

[实验环境 47](#_Toc469064081)

[实验原理 47](#_Toc469064082)

[实验步骤 49](#_Toc469064083)

[扩展实验 49](#_Toc469064084)

[实验二 进程间通信实验—共享内存 50](#_Toc469064085)

[实验目的 50](#_Toc469064086)

[实验内容 50](#_Toc469064087)

[实验环境 50](#_Toc469064088)

[实验原理 50](#_Toc469064089)

[实验步骤 53](#_Toc469064090)

[扩展实验 53](#_Toc469064091)

[实验三 进程间通信实验—消息队列 54](#_Toc469064092)

[实验目的 54](#_Toc469064093)

[实验内容 54](#_Toc469064094)

[实验环境 54](#_Toc469064095)

[实验原理 54](#_Toc469064096)

[实验步骤 57](#_Toc469064097)

[扩展实验 57](#_Toc469064098)

[第7章 串口收发实验 58](#_Toc469064099)

[实验一 串口收发实验 59](#_Toc469064100)

[实验目的 59](#_Toc469064101)

[实验内容 59](#_Toc469064102)

[实验环境 59](#_Toc469064103)

[实验原理 59](#_Toc469064104)

[实验步骤 63](#_Toc469064105)

[扩展实验 63](#_Toc469064106)

[第8章 网络通信实验 64](#_Toc469064107)

[实验一 UDP实验 65](#_Toc469064108)

[实验目的 65](#_Toc469064109)

[实验内容 65](#_Toc469064110)

[实验环境 65](#_Toc469064111)

[实验原理 65](#_Toc469064112)

[实验步骤 67](#_Toc469064113)

[扩展实验 68](#_Toc469064114)

[实验二 TCP客户端实验 69](#_Toc469064115)

[实验目的 69](#_Toc469064116)

[实验内容 69](#_Toc469064117)

[实验环境 69](#_Toc469064118)

[实验原理 69](#_Toc469064119)

[实验步骤 72](#_Toc469064120)

[扩展实验 73](#_Toc469064121)

[实验三 TCP服务器实验 74](#_Toc469064122)

[实验目的 74](#_Toc469064123)

[实验内容 74](#_Toc469064124)

[实验环境 74](#_Toc469064125)

[实验原理 74](#_Toc469064126)

[实验步骤 76](#_Toc469064127)

[扩展实验 77](#_Toc469064128)

[实验四 简易web服务器实验 78](#_Toc469064129)

[实验目的 78](#_Toc469064130)

[实验内容 78](#_Toc469064131)

[实验环境 78](#_Toc469064132)

[实验原理 78](#_Toc469064133)

[实验步骤 88](#_Toc469064134)

[扩展实验 90](#_Toc469064135)

[第9章 GPIO驱动实验 91](#_Toc469064136)

[实验一 GPIO驱动编写实验 92](#_Toc469064137)

[实验目的 92](#_Toc469064138)

[实验内容 92](#_Toc469064139)

[实验环境 92](#_Toc469064140)

[实验原理 92](#_Toc469064141)

[实验步骤 95](#_Toc469064142)

[扩展实验 96](#_Toc469064143)

[实验二 GPIO输出实验 97](#_Toc469064144)

[实验目的 97](#_Toc469064145)

[实验内容 97](#_Toc469064146)

[实验环境 97](#_Toc469064147)

[实验原理 97](#_Toc469064148)

[实验步骤 100](#_Toc469064149)

[扩展实验 100](#_Toc469064150)

[实验三 GPIO输入实验 101](#_Toc469064151)

[实验目的 101](#_Toc469064152)

[实验内容 101](#_Toc469064153)

[实验环境 101](#_Toc469064154)

[实验原理 101](#_Toc469064155)

[实验步骤 105](#_Toc469064156)

[扩展实验 105](#_Toc469064157)

[第10章 LED驱动实验 106](#_Toc469064158)

[实验一 内核模块基本实验 107](#_Toc469064159)

[实验目的 107](#_Toc469064160)

[实验内容 107](#_Toc469064161)

[实验环境 107](#_Toc469064162)

[实验原理 107](#_Toc469064163)

[实验步骤 107](#_Toc469064164)

[扩展实验 109](#_Toc469064165)

[实验二 GPIO内核态输出实验 110](#_Toc469064166)

[实验目的 110](#_Toc469064167)

[实验内容 110](#_Toc469064168)

[实验环境 110](#_Toc469064169)

[实验原理 110](#_Toc469064170)

[实验步骤 112](#_Toc469064171)

[扩展实验 112](#_Toc469064172)

[实验三 GPIO内核态输入实验 113](#_Toc469064173)

[实验目的 113](#_Toc469064174)

[实验内容 113](#_Toc469064175)

[实验环境 113](#_Toc469064176)

[实验原理 113](#_Toc469064177)

[实验步骤 116](#_Toc469064178)

[扩展实验 117](#_Toc469064179)

[实验四 LED驱动编写实验 118](#_Toc469064180)

[实验目的 118](#_Toc469064181)

[实验内容 118](#_Toc469064182)

[实验环境 118](#_Toc469064183)

[实验原理 118](#_Toc469064184)

[实验步骤 126](#_Toc469064185)

[扩展实验 126](#_Toc469064186)

[第11章 I2C驱动实验 127](#_Toc469064187)

[实验一 I2C驱动编写实验 128](#_Toc469064188)

[实验目的 128](#_Toc469064189)

[实验内容 128](#_Toc469064190)

[实验环境 128](#_Toc469064191)

[实验原理 128](#_Toc469064192)

[实验步骤 138](#_Toc469064193)

[扩展实验 138](#_Toc469064194)

[实验二 ZLG7290驱动编写实验 139](#_Toc469064195)

[实验目的 139](#_Toc469064196)

[实验内容 139](#_Toc469064197)

[实验环境 139](#_Toc469064198)

[实验原理 139](#_Toc469064199)

[实验步骤 162](#_Toc469064200)

[扩展实验 162](#_Toc469064201)

[实验三 EEPROM驱动编写实验 163](#_Toc469064202)

[实验目的 163](#_Toc469064203)

[实验内容 163](#_Toc469064204)

[实验环境 163](#_Toc469064205)

[实验原理 163](#_Toc469064206)

[实验步骤 176](#_Toc469064207)

[扩展实验 176](#_Toc469064208)

[第12章 SD卡驱动实验 178](#_Toc469064209)

[实验一 SD卡驱动编写实验 179](#_Toc469064210)

[实验目的 179](#_Toc469064211)

[实验内容 179](#_Toc469064212)

[实验环境 179](#_Toc469064213)

[实验原理 179](#_Toc469064214)

[实验步骤 188](#_Toc469064215)

[扩展实验 188](#_Toc469064216)

[第13章 framebuffer编程实验 189](#_Toc469064217)

[实验一 framebuffer实验 190](#_Toc469064218)

[实验目的 190](#_Toc469064219)

[实验内容 190](#_Toc469064220)

[实验环境 190](#_Toc469064221)

[实验原理 190](#_Toc469064222)

[实验步骤 194](#_Toc469064223)

[扩展实验 194](#_Toc469064224)

[第14章 共享库实验 195](#_Toc469064225)

[实验一 共享库基础实验 196](#_Toc469064226)

[实验目的 196](#_Toc469064227)

[实验内容 196](#_Toc469064228)

[实验环境 196](#_Toc469064229)

[实验原理 196](#_Toc469064230)

[实验步骤 196](#_Toc469064231)

[扩展实验 199](#_Toc469064232)

[实验二 zlib库使用实验 200](#_Toc469064233)

[实验目的 200](#_Toc469064234)

[实验内容 200](#_Toc469064235)

[实验环境 200](#_Toc469064236)

[实验原理 200](#_Toc469064237)

[实验步骤 200](#_Toc469064238)

[扩展实验 202](#_Toc469064239)

[第15章 Qt图形界面编程实验 203](#_Toc469064240)

[实验一 Qt工程导入 204](#_Toc469064241)

[实验目的 204](#_Toc469064242)

[实验内容 204](#_Toc469064243)

[实验环境 204](#_Toc469064244)

[实验原理 204](#_Toc469064245)

[实验步骤 204](#_Toc469064246)

[扩展实验 206](#_Toc469064247)

[实验二 Qt下GPIO输出实验 207](#_Toc469064248)

[实验目的 207](#_Toc469064249)

[实验内容 207](#_Toc469064250)

[实验环境 207](#_Toc469064251)

[实验原理 207](#_Toc469064252)

[实验步骤 207](#_Toc469064253)

[扩展实验 209](#_Toc469064254)

[实验三 Qt下GPIO输入实验 210](#_Toc469064255)

[实验目的 210](#_Toc469064256)

[实验内容 210](#_Toc469064257)

[实验环境 210](#_Toc469064258)

[实验原理 210](#_Toc469064259)

[实验步骤 210](#_Toc469064260)

[扩展实验 213](#_Toc469064261)

[附录A 推荐阅读书目 214](#_Toc469064262)

[附录B SylixOS ARM BSP简介 215](#_Toc469064263)

[附录C SylixOS系统调试及固化 216](#_Toc469064264)

[附录D Shell命令说明 220](#_Toc469064265)

# SylixOS开发基础知识

## SylixOS介绍

SylixOS是目前国内功能最为完善的实时嵌入式操作系统，符合IEEE1003（IEC9945）POSIX操作系统规范，同时支持IEEE1003.1b针对于实时操作系统的扩展标准。实时性与知名的VxWorks操作系统相当，同时又可兼容大部分Linux、UNIX、BSD等POSIX兼容系统应用程序与中间件。SylixOS目前由北京翼辉信息技术有限公司负责维护升级，同时提供完善的开发、调试、仿真、部署环境。

SylixOS操作系统以开放源代码形式存在，开源版权遵循GPL规范，方便学习和研究。本实验指导书是针对SylixOS的Cortex-A9教学验证平台（以下称验证平台）编写，但本书中的方法在RealEvo-Simulator（与RealEvo-IDE一起提供的SylixOS虚拟机）和其他平台上同样适用。

## RealEvo-IDE介绍

SylixOS开发使用由北京翼辉信息技术有限公司开发的RealEvo-IDE集成开发环境。

RealEvo-IDE开发环境可以在http://www.acoinfo.com/上申请下载。

RealEvo-IDE是一套完备的集成开发环境，上面集成的工具能完整覆盖软件设计、编译、调试及性能分析等研发过程，非常稳定易用。

本书中的所有例程均在RealEvo-IDE下编译测试，在进行本书实验前需要下载并安装RealEvo-IDE。

SylixOS集成开发环境安装包文件名一般为SylixOS IDE xxx.iso(xxx为版本号)，使用虚拟光驱软件DTLite加载此文件后会看到如图 1.1所示内容，双击“InstallWizard.exe”可以安装RealEvo-IDE系列集成开发环境。SylixOS IDE xxx.iso文件中doc目录下有《RealEvo系列软件注册步骤.pdf》说明文档，根据文档中的说明可以完成RealEvo-IDE的安装和注册。

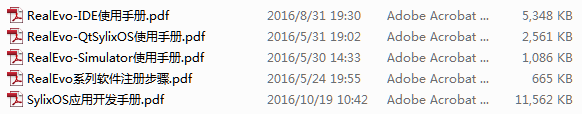


图 1.1 SylixOS IDE光盘内的文档

图 1.1所示的内容中，文件夹“RealEvo-Simulator”和“RealEvo-IDE”下面分别有相应软件的使用说明文档《RealEvo-Simulator使用手册》和《RealEvo-IDE使用手册》。手册中对RealEvo-IDE的使用有细致和全面的说明，通过对手册中内容的练习可以全面掌握对RealEvo-IDE的使用。

注：为了保证实验顺利进行，您务必要根据《RealEvo-IDE使用手册》熟悉RealEvo-IDE基本操作。

## RealEvo IDE常用工程介绍

SylixOS软件开发共分为三层（分别对应RealEvo-IDE里面三种不同类型的工程）：

* 操作系统内核开发 -- SylixOS Base工程
* BSP软件开发； -- SylixOS BSP工程
* 应用软件开发； -- SylixOS APP工程

其中操作系统内核是高度抽象的软件集合，里面集成了操作系统所有组件，包括SylixOS内核、文件系统、Posix支持、网络协议栈及组件。这些软件符合标准而且与硬件无关，可以提供给所有平台使用；

BSP软件是与硬件相关的软件的集合，将所有硬件相关的细节封装在BSP层，使得操作系统内核和应用程序与硬件实现解耦；

应用软件是用户针对不同使用场景和需求而开发的软件，通过调用SylixOS操作系统的标准编程接口，即可调用硬件功能；更换硬件平台后，应用软件不需要修改，只要重新编译即可实现原有的功能。

## 建立SylixOS Base工程

在开发应用程序前，需要建立一个SylixOS Base工程，为应用程序提供操作系统标准编程接口支持。

RealEvo IDE安装并注册成功之后，即可在主菜单File栏目找到建立SylixOS Base工程的入口，具体操作如下图所示：

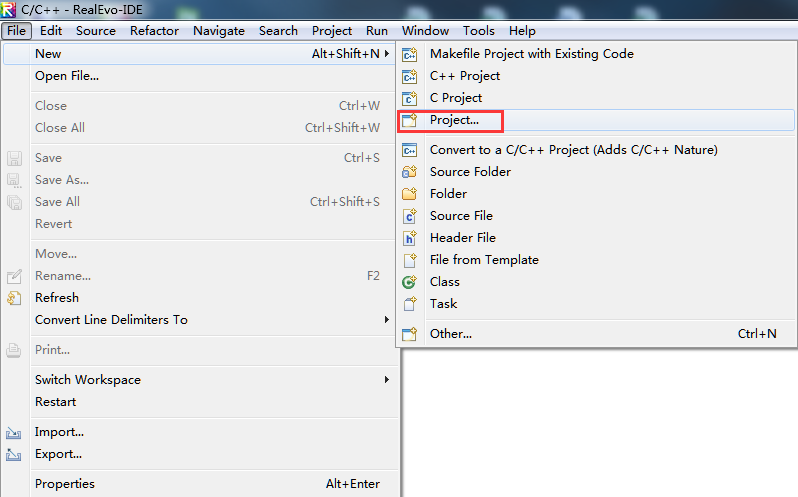


图1.4.1 选择Project

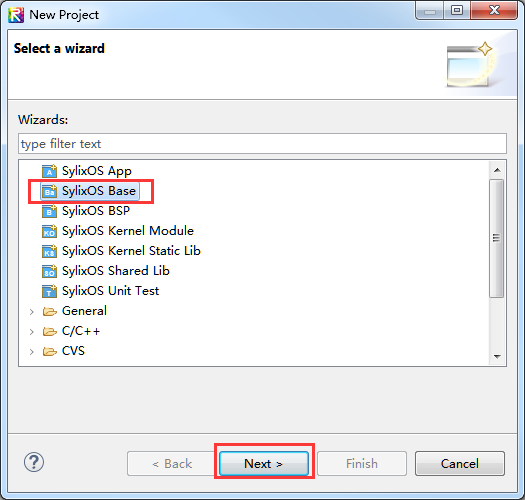


图1.4.2 选择SylixOS Base

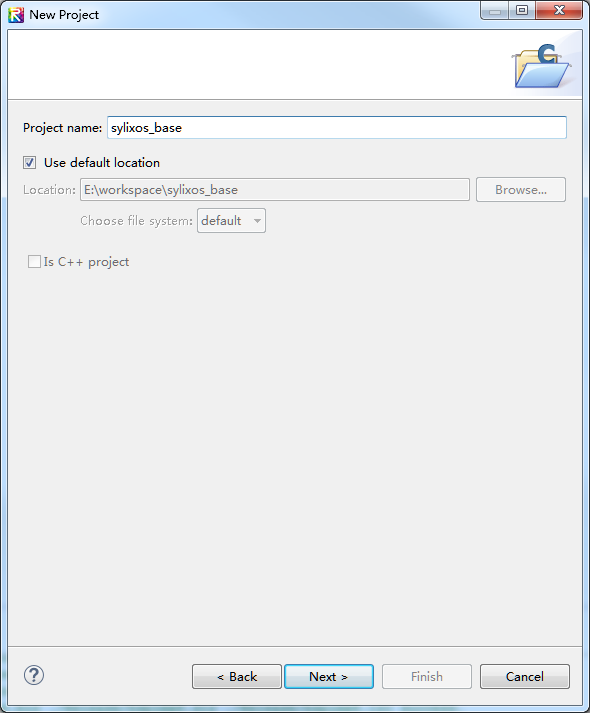


图1.4.3 填写工程名

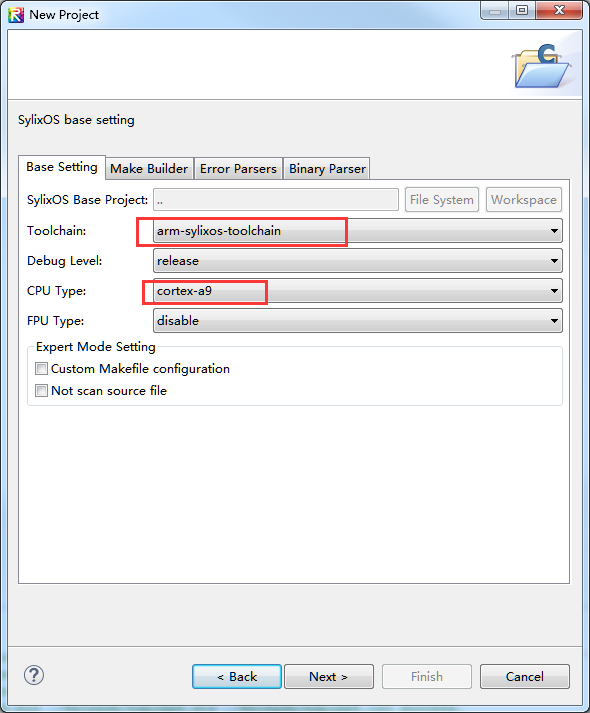


图1.4.4 选择工具链和CPU类型

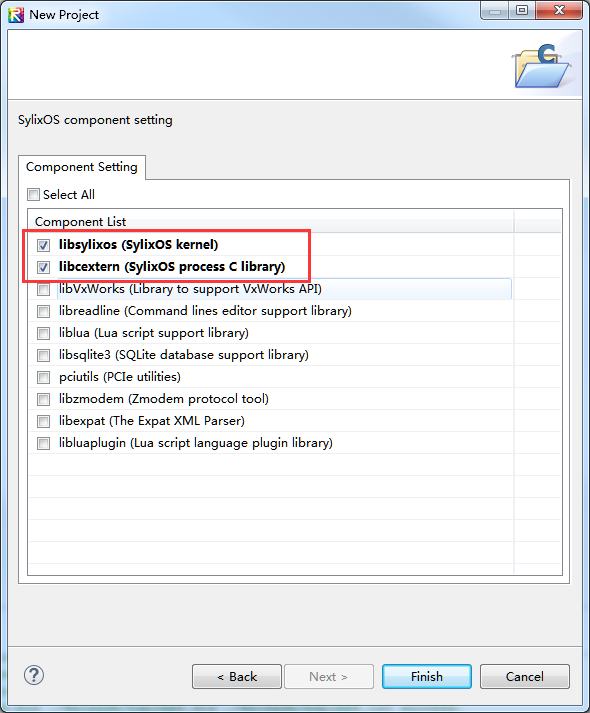


图1.4.5 选择操作系统组件

## SylixOS第一个应用程序helloworld

建立SylixOS Base工程后，可以建立应用程序开始应用软件开发。本文通过一个helloworld工程，演示应用程序创建的流程。

打开RealEvo-IDE，选择“File”→“New”→“Project”打开SylixOS工程建立对话框，选择SylixOS App工程，点击“Next”并输入工程名称“helloworld”。

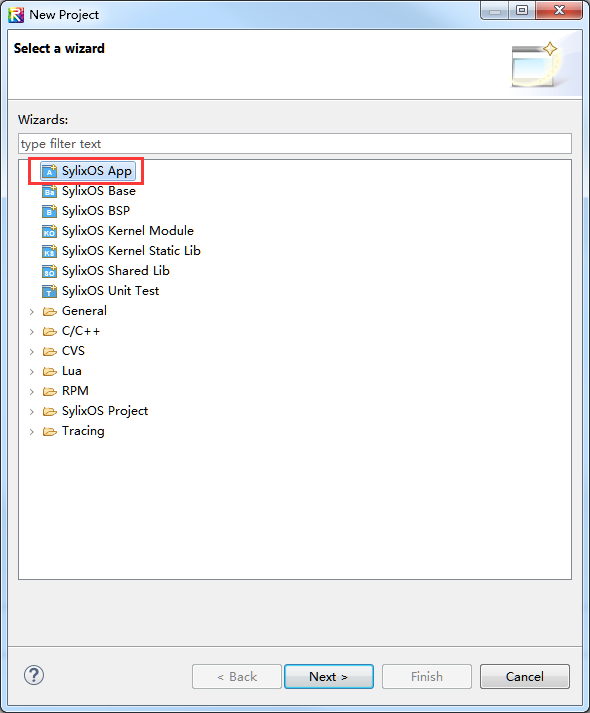


图 1.7 helloworld工程建立对话框

点击“Next”会显示“SylixOS app project setting”界面，选择base工程目录后。点击“Finish”完成工程创建。

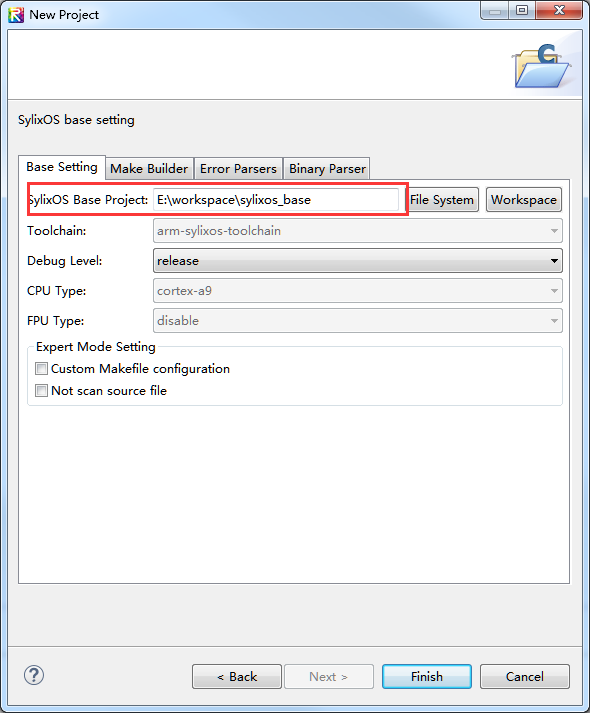


图 1.8 选择base工程目录

工程创建后将自动生成模板代码，这里不做任何更改，选择工程右键→“Build Project”对工程进行编译。编译完成后将生成名为helloworld的可执行程序文件。

将helloworld程序上传到验证平台中，这里我们需要设置工程上传路径，选择工程右键→“Properties”→“SylixOS Setting” →“Device Setting” →“New Device”，在弹出对话框的中填写验证平台的IP地址，其他保持默认配置。点击“OK”返回。

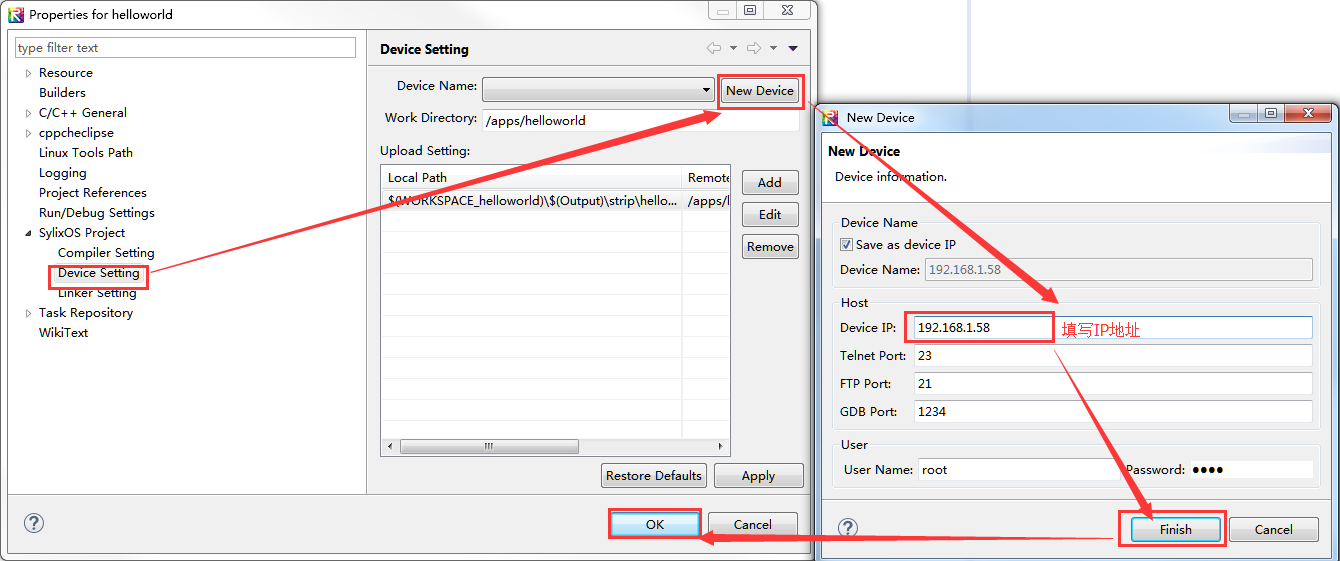


图1.9 绑定设备IP

选择工程右键→“SylixOS”→“Upload”，此操作会将helloworld文件上传到验证平台中，并在RealEvo-IDE的“Console”中输出上传结果信息。如果提示失败，则需要检查网线连接、IP地址设置。

在控制台中，将路径切换到/apps/helloworld下，会看到helloworld程序文件，执行helloworld程序会输出程序执行结果。

[root@sylixos\_station:/apps/helloworld]# ls

helloworld

[root@sylixos\_station:/apps/helloworld]# ./helloworld

Hello SylixOS!

## RealEvo-IDE工程导入

本书配套有完整的实验相关工程文件，这些工程文件在拷贝到读者电脑上后，需要进行一些设置修改才能够使用。

将包含完整工程文件的工作空间（sylixos\_example文件夹）拷贝到一个不含中文字符的路径下。打开RealEvo-IDE，在Workspace Launcher对话框中使用“Browse”按钮选择相应文件夹并打开。任何一个工程打开后直接编译会出现错误，这里需要修改工程“Base Project Path”的设置，选择工程右键 →“Properties”→“SylixOS Project”，设置界面如图 1.10所示，点击按钮“WorkSpace”会在弹出对话框中列出本工作空间下所有工程，选择“sylixos\_base”工程，点击“OK”返回。本工作空间的所有工程在编译之前，都需要设置“Base Project Path”。sylixos\_base工程本身的路径由于始终是设置自己的所在目录，所以在工程导入后sylixos\_base工程可以直接编译。

如果导入单个工程，将工程拷贝到工作空间中，使用“File”→“Import”→“General” →“Existing Projects into Workspace”导入RealEvo-IDE，随后使用上述方法更改其“Base Project Path”设置即可。

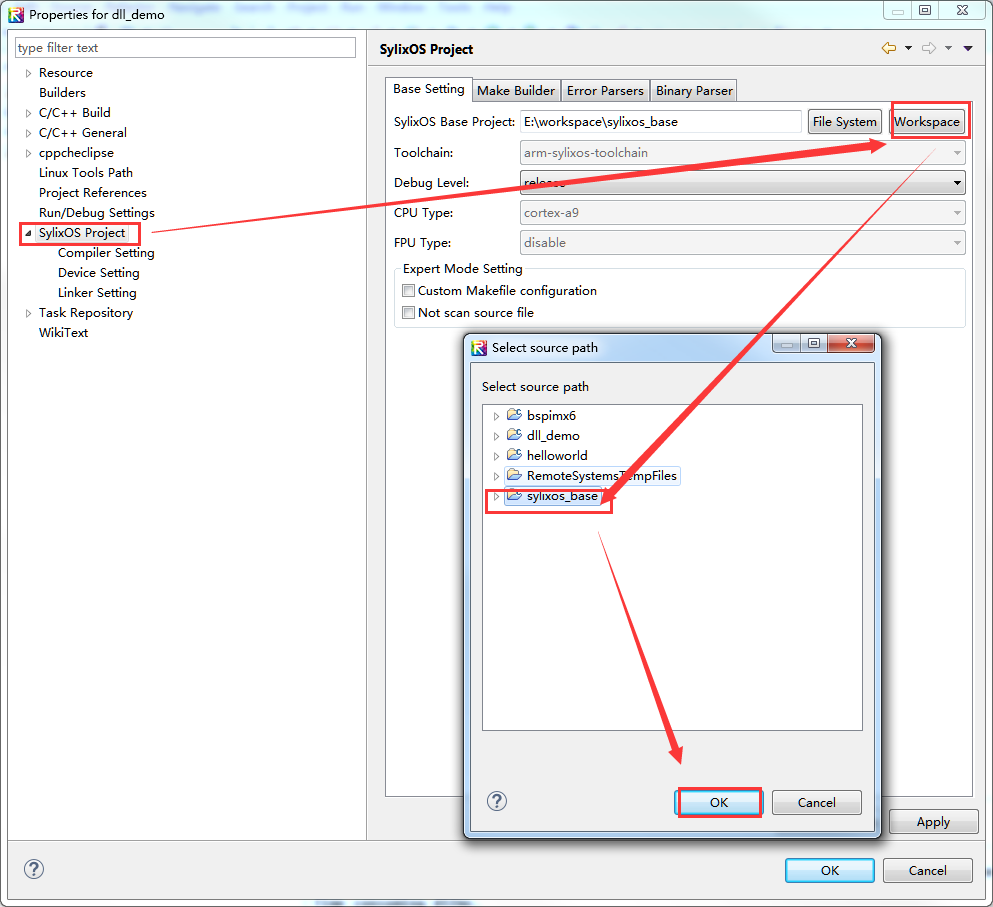


图 1.10 更改Base Project Path设置

# ARM汇编基础实验

在软件行业，绝大部分工作都是使用高级语言完成，但在嵌入式领域中仍然会时常用到汇编编程。结合实际工作需要，本章主要讲解汇编与C语言混合编程的方法。

## 实验一 ARM汇编编程实验

### 实验目的

* 了解ARM处理器的常用汇编指令和语法
* 了解C语言与汇编混合编程基础
* 掌握SylixOS下汇编代码编写特点，能够阅读并理解SylixOS中的汇编代码

### 实验内容

* 阅读汇编程序代码了解汇编指令的功能和使用
* 阅读实验原理掌握C语言与汇编混合编程的方法
* 在RealEvo-IDE下编译asm\_program工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

当前嵌入式领域主要使用C语言进行程序设计，而本实验指导书之所以介绍ARM汇编原因如下：

* 教育不能与培训类比，教育着眼于一个人更长期的发展；
* 汇编语言可以帮助我们更好理解计算机系统内部的运行，帮助我们更好的学习C语言；
* 在操作系统中涉及CPU启动、硬件驱动的很多地方还只能够靠汇编完成。

ARM汇编学习的主要难点是ARM汇编指令繁多，在简单了解汇编语法后，根据汇编程序翻阅ARM汇编指令介绍文档是学习ARM汇编非常高效的一种方法。《ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition》的A8章节详细说明了每一个ARM-V7架构处理器支持的汇编指令的功能和使用。在学习的时候也可以参考本书推荐的书目《ARM嵌入式系统开发：软件设计与优化》中文版本。

在实际工程应用中ARM汇编常常与C语言混合编程，为了实现ARM汇编与C语言程序的相互调用，子程序需要遵循ATPCS规则，基本的ATPCS规定有：

* 各寄存器的使用规则及其相应的名称；
* 数据栈的使用规则；
* 参数传递的规则。

简而言之，如果函数有不多于四个参数，使用ARM寄存器R0-R3进行传递，若参数多于四个则借助栈，函数返回值通过R0返回。

在SylixOS下实现C语言调用汇编程序需要两个步骤：

* 在C程序中声明函数原型，并使用extern标记为外部实现；
* 在汇编程序包含头文件arch/assembler.h，使用EXPORT\_LABEL(func)标记函数名。

本实验使用ARM汇编实现冒泡排序功能函数，并由C语言实现对该函数的调用，汇编部分代码如程序清单2.1 所示，详细说明参考注释。

程序清单2.1 冒泡排序算法函数sort汇编代码(sort.S)

#ifndef ASSEMBLY

#define ASSEMBLY 1

#endif

#include <arch/assembler.h>

FILE\_BEGIN()

EXPORT\_LABEL(sort) ;/\* 声明导出的函数名 \*/

FUNC\_DEF(sort) ;/\* 函数名 \*/

;/\* 入口参数 \*/

;/\* R0 <== buffer 首地址 \*/

;/\* R1 <== bufferlen 数据个数 \*/

SUBS R1, R1, #1 ;/\* 第一层循环次数 R1 \*/

CMP R1, #0 ;/\* 判断buffer长度为1 则退出 \*/

BLE EXIT

L0:

MOV R2, #0

MOV R3, R1 ;/\* 第二层循环次数 R3 \*/

L1:

LDR R6, [R0, R2] ;/\* R6 <= [R0+R2] \*/

ADD R4, R2, #4

LDR R7, [R0, R4] ;/\* R7 <= [R0+R2+1] \*/

CMP R6, R7

BLT L2

STR R7, [R0, R2] ;/\* R6 < R7 交换 \*/

STR R6, [R0, R4]

L2:

ADD R2, R2, #4

SUBS R3, R3, #1 ;/\* 二层循环控制 \*/

BNE L1

SUBS R1, R1, #1 ;/\* 一层循环控制 \*/

BNE L0

EXIT:

MOV PC, LR ;/\* 函数返回 \*/

FUNC\_END()

FILE\_END()

C语言中对函数sort进行声明后，可以像调用其他C语言函数一样使用冒泡排序功能函数sort。C语言部分程序代码如程序清单2.2所示：

程序清单2.2 冒泡排序算法函数调用(asm\_program.c)

#include <stdio.h>

#include <stdlib.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 汇编代码程序声明

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

extern int sort (int \*buf,int buflen); /\* 汇编代码程序声明 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: showdata

\*\* 功能描述: 通过控制台输出 buffer[] 中的数据

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void showdata (int \*buf, int buflen)

{

int i;

printf("buffer data:\n");

for(i = 0; i < buflen; i++) {

printf("%4d", \*(buf + i));

}

printf("\n");

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main()

{

int data;

int buffer[256]; /\* 数据缓冲区 \*/

int bufferlen = 0; /\* 数据缓冲区数据个数 \*/

printf("data input :\n");

while (1) { /\* 键盘输入整型数 ==〉 data \*/

scanf("%d", &data);

buffer[bufferlen++] = data;

if((data == -1) || (bufferlen >= 256)) { /\* 输入数据为 -1 结束输入 \*/

bufferlen--;

break;

}

}

printf("data input end!\n");

showdata(buffer, bufferlen); /\* 将输入数据输出到控制台 \*/

sort(buffer, bufferlen); /\* 对输入数据排序 \*/

showdata(buffer, bufferlen); /\* 输出排序数据 \*/

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开，编译等级选择debug模式编译asm\_program工程，将生成的工程文件上传到验证平台并执行；

[root@sylixos\_station:/apps/asm\_program]# ./asm\_program

1. 根据程序执行后输出的提示信息，输入一组数据，最后一个数据为-1标识输入结束，随后程序会输出获取的数据和经过排序后的数据，结果如下：

[root@sylixos\_station:/apps/asm\_program]# ./asm\_program

data input :

8 6 9 3 7 2 1 3 -1

data input end!

buffer data:

8 6 9 3 7 2 1 3

buffer data:

1 2 3 3 6 7 8 9

[root@sylixos\_station:/apps/asm\_program]#

### 扩展实验

验证平台的BSP工程bspimx6中有汇编代码文件startup.S，仔细阅读该文件，理解其中的汇编指令和代码功能。

# 文件操作实验

本章主要介绍在SylixOS下对普通文件（磁盘文件）操作方法和文件操作相关函数的使用。SylixOS中文件的概念与UNIX系统类似，它不仅仅是指普通文件，还包含设备文件、管道文件等等。虽然类型不同，但这些文件的操作方法函数是相同或相似的，操作步骤也是基本相同的。

## 实验一 shell命令操作文件实验

### 实验目的

* 掌握常用文件操作shell命令的使用
* 能够借助SylixOS自身的帮助，使用其他的shell命令

### 实验内容

* 使用vi编辑文件，向文件中写入一组数据并查看
* 使用shell命令创建文件、文件夹，实现对文件的拷贝、删除等操作

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

shell是操作系统的“外壳”程序，它向使用者提供了一个基于命令行的使用界面，也可称作命令解析器，系统开发人员常使用此接口来操作计算机。几乎所有的操作系统都包含shell程序，例如Linux系统中较为常见的shell是bash程序，Windows系统中的shell程序是cmd.exe。SylixOS也不例外，SylixOS包含自己的shell程序：ttinyShell。ttinyShell程序是系统开发人员操作SylixOS操作系统最为简单与便捷的接口，它与Linux系统shell规则非常相似，有区别的是ttinyShell运行在内核空间，它不是一个应用程序，所以ttinyShell不仅可以运行应用程序，而且内部内建了很多已经固化在SylixOS内核里的常用命令。

作为一款嵌入式操作系统，SylixOS有极好的可扩展性和可裁剪性，当内核不同或裁剪配置不同时，shell内建的命令也会有些许差别。在控制台中输入“help”命令，会列出此版本内核支持的内建shell命令及命令使用的相关参数。 若要查询特定shell内建命令xxx的使用方法，使用命令“help xxx”，控制台会输出此shell内建命令的使用提示信息。

vi是Unix/Linux下的标准文本编辑器，随着嵌入式Linux的广泛使用，vi编辑器得到了大量普及。SylixOS中也加入了对vi编辑器的支持，可以通过vi编辑器来编辑SylixOS文件系统内的文本文件或脚本。

### 实验步骤

将验证平台的调试串口、网口与PC主机的串口、网口进行连接（网口连接可以直接相连也可以通过交换机）。在PC主机上打开putty软件，验证平台上电，这时会在putty显示界面输出信息（即控制台），通过此控制台可以输入shell命令。

1. 查看当前目录，切换到“/apps”目录下：

[root@sylixos\_station:/]# ls

tmp var root home apps

sbin bin usr lib qt

etc boot usb yaffs2 proc

media mnt dev

[root@sylixos\_station:/]# cd apps

[root@sylixos\_station:/apps]#

如果当前目录下有“apps”目录，使用命令“cd apps”可以直接切换到apps目录。如果当前目录是在“/usr”等其他目录，这些目录下没有“apps”目录，需要使用“cd /apps”命令。“cd”命令的参数可以是相对路径也可以是绝对路径，具体使用需要根据实际情况和个人习惯而定。

1. 创建文件并写入一组数据

* 输入“vi file.txt”创建一个名为file.txt的文件
* 使用“插入/Insert”键，将vi切换到插入模式，输入一串任意字符
* 使用“退出/Esc”键，将vi切换到命令行模式，输入“:wq”命令保存文件并退出
* 使用“ls”查看当前目录下文件，会发现多出一个“file.txt”文件
* 查看文件是否写入flash，将验证平台重新上电，切换到“/apps”目录下，查看文件内容

[root@sylixos\_station:/apps]# cat file.txt

hello this is a file on disk.

[root@sylixos\_station:/apps]#

1. 创建一文件目录，并将“file.txt”文件拷贝到新建目录下

[root@sylixos\_station:/apps]# mkdir file /\* 创建文件目录 \*/

[root@sylixos\_station:/apps]# cp file.txt /apps/file /\* 拷贝文件 \*/

copy complete. size:30(Bytes) time:0(s) speed:30(Bps) /\* 提示信息 \*/

[root@sylixos\_station:/apps]# cd file /\* 进入新建目录 \*/

[root@sylixos\_station:/apps/file]# ls /\* 查看文件存在 \*/

file.txt

[root@sylixos\_station:/apps/file]# cat file.txt /\* 验证文件内容 \*/

hello this is a file on disk.

[root@sylixos\_station:/apps/file]#

1. 将创建的文件目录和文件删除

[root@sylixos\_station:/apps/file]# ls /\* 确认文件存在 \*/

file.txt

[root@sylixos\_station:/apps/file]# rm file.txt /\* 删除文件 \*/

[root@sylixos\_station:/apps/file]# ls /\* 确认文件删除 \*/

[root@sylixos\_station:/apps/file]# cd ../ /\* 返回上一级目录 \*/

[root@sylixos\_station:/apps]# rm file /\* 删除目录 \*/

[root@sylixos\_station:/apps]# cd file /\* 确认目录删除 \*/

cd : error : No such file or directory

[root@sylixos\_station:/apps]#

1. 使用Telnet实现SylixOS控制台

当前无论是笔记本电脑还是PC主机均很少有串口，在调试时往往需要一个USB转RS232功能的转换器。但在很多场合使用USB转RS232并不是很方便，SylixOS还支持Telnet连接功能，可以通过网络建立SylixOS控制台。确认当前网络连接正常，并已知当前验证平台的IP地址，打开putty软件，在启动界面进行如图 3.1设置，完成后点击“Open”按钮。

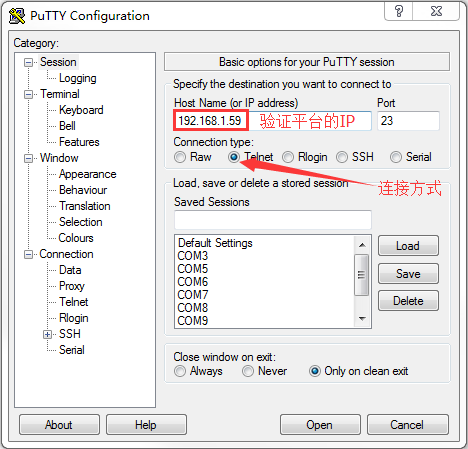


图 3.1 putty使用Telnet连接的设置

在进入putty界面后，要求输入用户名（login）和密码（password），这里均为“root”。输入后即打开SylixOS控制台。

login: root

password: /\* 密码不回显 \*/

... /\* 省略了一部分字符 \*/

SylixOS license: Commercial & GPL.

SylixOS kernel version: 1.3.4 NeZha(b)

CPU : Freescale i.MX6Q (Cortex-A9 Max@1.2GHz NEON VFPv3)

CACHE : 64KBytes L1-Cache(D-32K/I-32K), 1MBytes L2-Cache

PACKET : SylixOS EVM

ROM SIZE: 0x00002000 Bytes (0x00000000 - 0x00001fff)

RAM SIZE: 0x40000000 Bytes (0x10000000 - 0x4fffffff)

BSP : BSP version 0.8.0

[root@sylixos\_station:/]#

### 扩展实验

参考附录C中的说明，将本实验没有用到的shell文件命令进行熟悉和使用。

## 实验二 文件操作实验

### 实验目的

* 了解SylixOS下文件操作流程
* 掌握文件操作的函数用法

### 实验内容

* 阅读file\_operator.c文件，了解文件的操作流程和文件操作函数的使用
* 在RealEvo-IDE下编译file\_operator工程，并将程序上传到验证平台运行查看运行效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

在SylixOS中与文件操作相关的5个基本函数为open、write、read、close、ioctl。open函数用来打开或创建一个文件，并返回文件描述符，剩余4个函数都需要open函数返回的文件描述符作为输入参数。write/read函数分别向打开的文件中写入/读取数据。close函数是将打开的文件关闭，释放打开文件占用的系统资源。ioctl函数是文件的杂项函数，有很多操作难以归入write/read操作的项，可以放在ioctl函数里实现，在使用设备文件时会经常使用。文件操作流程如下：

* 使用open函数打开文件；
* 使用write/read函数对文件进行读写操作；
* 使用close函数关闭文件。

本实验例程中，先创建一个文件，并向文件中写入一组数据，随后关闭文件。再次打开创建的文件，读取文件中的数据并通过控制台输出显示。

**代码分析：**

程序清单3.1 file\_operator.c代码

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <fcntl.h>

#include <errno.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

char str[] = "Hello, welcome to SylixOS world!"; /\* 写入文件的内容 \*/

char filename[] = "hello.txt"; /\* 创建文件名称 \*/

int fd;

int ret;

char buf[128];

/\*

\* 以只写方式打开文件，若没有文件就创建一个文件 权限模式 mode=0x664

\*/

fd = open(filename, O\_WRONLY | O\_CREAT,

S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP | S\_IROTH);

if (fd < 0) {

printf("open file \"%s\" failed\n", filename);

return (-1);

}

ret = write(fd, str, sizeof(str)); /\* 文件写入数据 \*/

if (ret < 0) {

printf("write to file failed.\n");

close(fd);

return (-1);

}

printf("write %d bytes to \"%s\".\n", ret, filename);

fsync(fd); /\* 将文件从缓存写入磁盘 \*/

close(fd); /\* 关闭文件 \*/

fd = open(filename, O\_RDONLY); /\* 只读方式打开文件 \*/

if (fd < 0) {

printf("open file \"%s\" failed\n", filename);

return (-1);

}

ret = read(fd, buf, sizeof(buf)); /\* 读取文件的内容并显示 \*/

if (ret < 0) {

printf("read from file failed.\n");

close(fd);

return (-1);

}

printf("read %d bytes from file \"%s\",

data=\"%s\"\n", ret, filename, buf);

close(fd); /\* 关闭文件 \*/

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开随书工程，选择工程file\_operator右键→“Build Project”编译工程；
2. 选择工程file\_operator右键→“SylixOS”→“Upload Setting”设置验证平台ip地址和需要上传的程序文件路径；
3. 选择工程file\_operator右键→“SylixOS”→“Upload”向验证平台上传程序文件，在RealEvo-IDE开发环境下方提示文件传输成功信息时，说明上传成功。若失败需要检查设置和网线连接。默认传输路径为/apps/工程名称/工程名称；
4. 在控制台中将工作目录切换到“/apps/file\_operator”，执行程序file\_operator；

[root@sylixos\_station:/apps/file\_operator]# ./file\_operator

write 33 bytes to "hello.txt".

read 33 bytes from file "hello.txt", data="Hello, welcome to SylixOS world!"

### 扩展实验

文件的删除可以使用remove/unlink函数，remove/unlink函数的输入参数是文件名称。创建SylixOS App工程并添加代码使之能够将实验中新建的文件删除。

## 实验三 文件目录操作实验

### 实验目的

* 了解SylixOS下文件目录的概念和操作流程
* 掌握文件目录的创建、列举的相关函数和使用

### 实验内容

* 阅读dir\_operator.c文件，了解文件目录的创建和文件目录下文件遍历的操作方法
* 在RealEvo-IDE下编译dir\_operator工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

与文件目录操作相关函数有mkdir、rmdir、opendir、readdir、closedir等。其中mkdir函数用来创建一个目录，rmdir函数用来删除一个目录，opendir、readdir、closedir函数用来打开目录、获取当前目录下的信息、关闭目录等操作，可以类比文件操作的open、read、close函数。

本实验例程首先遍历当前目录下的文件和文件目录，并将其输出到控制台显示。随后创建一个名为“newdir”的文件目录，再次遍历当前目录下的文件和文件目录，输出信息与创建文件目录前的输出进行对比。

**代码分析：**

程序清单 3.2 dir\_operator.c代码

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/stat.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: printdir

\*\* 功能描述: 遍历当前目录下的文件和文件目录并通过控制台输出显示

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void printdir (void)

{

DIR \*dir;

struct dirent \*entry;

struct stat statbuf;

dir = opendir("./"); /\* 打开文件目录 \*/

if (!dir) {

printf("failed to open current directory\n");

return;

}

/\*

\* 循环读取当前目录下的文件和目录，直至全部读取完毕

\*/

while (1) {

entry = readdir(dir); /\* 读取目录下的文件 \*/

if (entry == NULL) { /\* 若没有新文件，返回 \*/

break;

} else {

lstat(entry->d\_name, &statbuf); /\* 获取文件信息 \*/

if (S\_ISDIR(statbuf.st\_mode)) { /\* 判断文件类型 \*/

printf("dir: %s\n", entry->d\_name); /\* 文件目录 \*/

} else {

printf("file: %s\n", entry->d\_name); /\* 其他类型文件 \*/

}

}

}

closedir(dir); /\* 关闭文件目录 \*/

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

char sz\_filepath[] = "./newdir"; /\* 创建的目录名称 \*/

int ret;

printdir(); /\* 输出当前目录下的信息 \*/

/\*

\* 在当前目录下创建一个文件目录

\*/

ret = mkdir(sz\_filepath, S\_IRWXU | S\_IRWXG | S\_IROTH | S\_IXOTH);

if (ret == 0) {

printf("create file path OK\n");

} else {

printf("create file path failed\n");

}

printdir(); /\* 输出当前目录下的信息 \*/

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开随书工程，选择工程dir\_operator右键→“Build Project”编译工程；
2. 选择工程dir\_operator右键→“SylixOS”→“Upload Setting”设置验证平台ip地址和需要上传的程序文件路径；
3. 选择工程dir\_operator右键→“SylixOS”→“Upload”向验证平台传送程序文件，在RealEvo-IDE下方提示文件传输成功信息时，说明文件传送成功。若失败需要检查相关设置和网线连接。默认传输路径为/apps/工程名称/工程名称；
4. 在验证平台控制台切换到目录“/apps/dir\_operator”下，执行程序dir\_operator。

[root@sylixos\_station:/apps/dir\_operator]# ./dir\_operator

file: dir\_operator

create file path OK

dir: newdir

file: dir\_operator

[root@sylixos\_station:/apps/dir\_operator]#

上述输出中，“file：”标记当前目录下文件名称，“dir：”标记当前目录下目录名称。

在没有创建“newdir”目录前，当前目录下只有“dir\_operator”文件，在“create file path OK”之后输出创建“newdir”目录后的结果，这时多出“newdir”目录。

### 扩展实验

编程实现遍历一个目录下的所有文件、子目录和子目录下的文件。

# 时间操作实验

本章主要介绍时间相关概念、时间读取和时间设置。通过两个实验，分别演示使用shell命令进行时间操作和使用函数在程序中进行时间操作。

在SylixOS中时间分为系统时间和RTC时间。系统时间是SylixOS系统中使用的时间，RTC时间是RTC（Real Time Clock）外设的时间，RTC时间是依靠外部晶振直接驱动，并使用电池保证在系统掉电情况下仍能够正常计时。一般在SylixOS系统启动时、BSP程序会将系统时间与RTC时间进行一次同步，并在系统运行中每间隔一段时间进行一次系统时间与RTC时间的同步。

## 实验一 shell命令时间操作实验

### 实验目的

* 掌握使用shell命令对系统时间和RTC时间设置、读取的方法

### 实验内容

* 使用shell命令完成对系统时间和RTC的读取、设置、同步

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

SylixOS与时间相关的shell命令只有两个：date、hwclock。

date命令用来显示和设置系统时间，其使用如下：

* date：用来显示当前时间
* date -s 23:06:06：设置当前系统时间
* date -s 20110217：设置当前系统日期

hwclock命令用来查看RTC时间，实现RTC时间与系统时间之间的同步，其使用如下：

* hwclock --show：显示当前的RTC时间
* hwclock --hctosys：修改系统时间，使之与当前RTC时间相同
* hwclock --systohc：修改RTC时间，使之与当前系统时间相同

如果在使用中忘记了命令的使用方法，help date可以查看date命令的使用提示，hwclock也可以使用同样的方法查看使用提示。

### 实验步骤

1. 使用date命令查看当前的系统时间；

[root@sylixos\_station:/]# date

Sat Jun 27 11:50:39 2015 /\* 当前的系统时间 \*/

1. 修改系统时间，分别设置了时间和日期；

[root@sylixos\_station:/]# date -s 22:02:12

Sat Jun 27 22:02:12 2015

[root@sylixos\_station:/]# date -s 20120222

Wed Feb 22 22:02:21 2012

可以看到当前时间是2012年2月22号22点02分21秒。

1. 此时使用hwclock查看rtc时间；

[root@sylixos\_station:/]# hwclock --show

Sun Jun 07 11:57:29 2015 /\* 时间仍然是设置前的时间，与系统时间并不一致\*/

1. 执行同步命令，使当前硬件时间与系统时间保持了一致。

[root@sylixos\_station:/]# hwclock --systohc

[root@sylixos\_station:/]# hwclock --show

Thu Feb 22 22:05:36 2012

### 扩展实验

参考其他书籍，考虑为什么要区分系统时间和RTC时间，时区对时间操作会有何影响并编程验证。

## 实验二 时间操作实验

### 实验目的

* 掌握使用程序读取、设置系统时间和RTC时间的方法
* 了解shell命令在程序中的使用方法

### 实验内容

* 阅读clock\_demo.c文件，了解与时间相关函数的使用和程序设计
* 在RealEvo-IDE下编译clock\_demo工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

在程序操作中，RTC时间是不可见的，即无法使用系统接口函数直接设置SylixOS下的RTC时间。设置RTC时间，操作流程与shell命令下流程类似，在设置系统时间后，将系统时间同步到RTC时间以完成对RTC时间的设置。

SylixOS下对系统时间操作遵循POSIX编程标准，在标准中有多个与时间相关的变量或变量结构体。对系统时间的操作均是以这些变量为基础。

1. time\_t变量

time\_t在SylixOS下是一个64位的整数，它的含义是表示从一个基准时间（1970年1月1日0时0分0秒）到某一时间点经过的秒数，一般称为日历时间（calendar time），日历时间与基准时间相加就是UTC时间（世界标准时间）。需要注意的是在Linux系统中此变量与CPU体系结构有关，在32位CPU中，time\_t是32位变量，这就意味着在32位的Linux系统中能够表示的时间上限为2038年1月18日19时14分07秒，而在SylixOS中不存在此问题。

1. tm结构体

tm结构体用来标识符合人阅读理解习惯的时间参数结构体，一般称为分解时间（broken-down time）。

struct tm{

int tm\_sec; /\* 代表目前秒数, 正常范围为0-59 \*/

int tm\_min; /\* 代表目前分数, 范围0-59 \*/

int tm\_hour; /\* 从午夜算起的时数, 范围为0-23 \*/

int tm\_mday; /\* 目前月份的日数, 范围01-31 \*/

int tm\_mon; /\* 代表目前月份, 从一月算起, 范围从0-11 \*/

int tm\_year; /\* 从1900 年算起至今的年数 \*/

int tm\_wday; /\* 一星期的日数, 从星期一算起, 范围为0-6 \*/

int tm\_yday; /\* 从今年1 月1 日算起至今的天数, 范围为0-365 \*/

int tm\_isdst; /\* 夏令时的旗标 \*/

};

系统提供了两种时间变量之间的转换函数localtime和mktime，分别实现从timer\_t转换到tm和从tm转换到time\_t。localtime是不可重入函数，因此系统又提供了一个可重入版本函数

localtime\_r，在工程使用中推荐使用localtime\_r函数。

1. timespec结构体

POSIX标准操作系统中最小的时间单位是：纳秒，为了配合最高精度时间函数使用，对时间设置和读取还需要单独的结构体timespec：

struct timespec {

time\_t tv\_sec; /\* time\_t结构体 \*/

LONG tv\_nsec; /\* 纳秒的定义 \*/

};

对时间进行设置和读取的函数为clock\_settime函数和clock\_gettime函数，虽然POSIX还定义了time函数，但不建议使用。

本实验例程全面演示了时间相关函数的使用：

* 使用clock\_gettime函数获取当前时间；
* 使用localtime\_r函数转换成分解时间并显示，这种格式在工程中会经常用到；
* 使用ctime函数转换成输出字符显示，与上一步骤对比，观察效果是否一致；
* 将时间增加1小时，使用mktime函数转换成日历时间；
* 使用clock\_settime函数设置时间；
* 再次读取系统时间，确认时间设置正常；
* 使用system函数在程序中调用shell命令，将系统时间同步到RTC时间。

程序清单4.1 clock\_demo.c

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

int ret;

struct tm tm\_temp;

struct tm \*tm\_time = &tm\_temp;

struct timespec settime;

ret = clock\_gettime(CLOCK\_REALTIME, &settime); /\* 读取系统时间 \*/

if (ret < 0) {

printf("read time error");

}

localtime\_r(&(settime.tv\_sec), tm\_time); /\* 转换时间表示格式 \*/

printf("%d-%2d-%2d ", (1900 + tm\_time->tm\_year), /\* 显示日期 \*/

(1 + tm\_time->tm\_mon), tm\_time->tm\_mday);

printf("%d:%d:%d", tm\_time->tm\_hour, /\* 显示时间 \*/

tm\_time->tm\_min, tm\_time->tm\_sec);

printf(" %d\n", tm\_time->tm\_wday); /\* 显示星期数 \*/

/\*

\* 使用 ctime 直接显示，与 localtime\_r 函数的转换做对比

\*/

printf(ctime(&(settime.tv\_sec)));

printf("\n");

tm\_time->tm\_hour = tm\_time->tm\_hour + 1; /\* 在当前时间上加 1 小时 \*/

settime.tv\_sec = mktime(tm\_time); /\* 转换时间表示格式 \*/

settime.tv\_nsec = 0;

ret = clock\_settime(CLOCK\_REALTIME, &settime); /\* 设置系统时间 \*/

if (ret < 0) {

printf("set time error");

}

ret = clock\_gettime(CLOCK\_REALTIME, &settime); /\* 读取系统时间进行验证 \*/

if (ret < 0) {

printf("read time error");

}

printf(ctime(&(settime.tv\_sec)));

printf("\n");

system("hwclock --systohc"); /\* 将时间同步到 RTC \*/

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译工程clock\_demo，将生成的程序文件上传到验证平台；
2. 进入SylixOS控制台，执行clock\_demo程序；

[root@sylixos\_station:/apps/clock\_demo]# ./clock\_demo

2015- 6- 7 4:36:15 0 /\* tm结构体输出结果 \*/

Sun Jun 07 04:36:15 2015 /\* ctime直接输出结果 \*/

Sun Jun 07 05:36:15 2015 /\* 时间设置后的结果 \*/

[root@sylixos\_station:/apps/clock\_demo]#

1. 将验证平台下电，并重新上电，使用date命令查看当前时间。

[root@sylixos\_station:/]# date

Sun Jun 07 05:37:10 2015

时间保持一致（秒数和分数因为时间走动会发生变化），说明完成了对RTC时钟的设置。

注：当验证平台RTC电池没有电的时候，系统掉电后无法保持设置时间。

### 扩展实验

使用本实验介绍的函数，如何实现类似shell命令的功能。

# 多线程实验

本章主要讲解线程的概念和线程间的同步方式。实验一主要介绍线程的概念和线程的创建，实验二、实验三、实验四分别介绍了信号量、互斥锁、条件变量的作用和使用。

## 实验一 线程管理实验

### 实验目的

* 掌握线程的概念
* 掌握线程创建、退出函数的使用方法

### 实验内容

* 阅读实验原理，了解线程的概念和作用
* 阅读thread\_create.c源代码，掌握线程管理相关函数的使用
* 在RealEvo-IDE下编译thread\_create工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

**线程的介绍**

线程是允许应用程序并发执行多个任务的一种机制，是程序运行后的任务处理单元，也是SylixOS操作系统任务调度的最小单元。在多核CPU中，同时可以有多个线程在执行，实现真正意义上的并行处理。

线程入口函数是一个能够完成特定任务的函数，因此线程入口函数的编写上与普通函数没有太多区别。

**线程常用函数**

SylixOS支持pthread接口，pthread（POSIX Thread）定义了一套管理线程的API函数。这里简要介绍常用的线程相关变量和函数。

1. 线程标识（线程ID）

每个线程都有其对应的线程ID，线程ID使用pthread\_t数据类型表示，在特定线程内，使用pthread\_self函数可以获取自身的线程ID。

pthread\_t pthread\_self(void);

1. 创建线程函数

int pthread\_create( pthread\_t \*thread,

const pthread\_attr\_t \*attr,

void \*(\*start\_routine)(void\*),

void \*arg);

* 创建线程失败，函数返回非0的错误代码，成功返回0；
* \*thread pthread\_t类型的缓冲区，保存一个线程的线程ID；
* \*attr 设置线程属性，设置为NULL标识创建的新线程使用默认属性；
* \*(\*start\_routine) 线程入口函数函数名
* \*arg 向所创建线程传入的参数

1. 自行退出当前线程函数

void pthread\_exit(void \*status);

* \*status 指定线程的返回值

此函数在线程内部调用，当某线程需要退出时，在线程内的实体函数调用此函数即可退出当前线程。使用return会隐式调用此函数退出当前线程。

1. 线程退出等待函数

int pthread\_join(pthread\_t thread, void \*\*ppstatus);

此函数用来将调用函数的线程挂起，直至指定线程退出。

* thread 已经被创建的线程ID
* ppstatus 保持线程终止时的返回值

**代码分析**

本例程实验的基本功能是创建5个子线程，主线程创建5个子线程后自行退出，各个子线程在输出自己线程的入口函数参数后退出。

首先包含相关头文件：

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

然后创建线程入口函数，主要功能是向控制台输出自身线程参数，随后退出当前线程。

程序清单5.1 线程入口函数

void \*print\_hello (void \*arg)

{

int i = (int)arg;

printf("it's me, thread %d!\n", i);

pthread\_exit(NULL); /\* 线程退出 \*/

return (NULL);

}

在main函数（主线程）中创建线程，完成线程创建后，退出本线程。

备注：SylixOS默认主线程退出时会自动等待所有子线程都执行完毕，进程方可退出，所以在主线程调用exit或return退出，不影响子线程继续执行。另外可使用vprocExitModeSet函数设置为主函数退出时进程立即结束（此方式不安全）。

程序清单5.2 创建线程main函数

int main (int argc, char \*argv[])

{

int ret;

int i;

pthread\_t tid[5];

for (i = 0; i < 5; i++) { /\* 创建 5 个线程 \*/

printf("in main: creating thread %d\n", i);

ret = pthread\_create(&tid[i], NULL, print\_hello, (void \*)i);

if (ret != 0) {

printf("create thread failed\n");

exit(-1);

}

}

printf("in main: exit!\n");

pthread\_exit(NULL); /\* 创建线程后，主线程退出 \*/

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译工程thread\_create，将生成的程序文件上传到验证平台中；
2. 在验证平台中执行程序thread\_create，控制台输出程序执行结果；

[root@sylixos\_station:/apps/thread\_create]# ./thread\_create

in main: creating thread 0

in main: creating thread 1

in main: creating thread 2

in main: creating thread 3

in main: creating thread 4

in main: exit!

it's me, thread 0!

it's me, thread 1!

it's me, thread 2!

it's me, thread 3!

it's me, thread 4!

[root@sylixos\_station:/apps/thread\_create]#

1. 在程序清单5.2主线程退出前，加入下列代码，再次运行查看程序执行效果。

pthread\_join(threads[0], NULL);

### 扩展实验

线程间如何进行大数据的交换，并编程验证。

## 实验二 线程同步实验—信号量

### 实验目的

* 了解线程中信号量的概念和作用
* 掌握信号量的相关函数及其使用方法

### 实验内容

* 阅读thread\_semaphore.c文件，了解信号量相关函数及在多线程中的使用方法
* 在RealEvo-IDE下编译thread\_semaphore工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

信号量是一个在进程和线程中都可以使用的同步机制。信号量类似于一个通知，某个线程发出一个通知，等待此通知的线程收到通知后，会执行预先设置的工作。当接收通知的线程没有收到通知前，会处于阻塞状态。

信号量可以连续发送多次，处理线程同样也会处理多次。信号量实质是一个计数器，信号量发送一次，计数值增加1，信号量每获取一次，计数值就减1，当计数值为0时，等待信号量线程阻塞。等待信号量过程中，还可以设置等待时间，超过设定时间，等待信号量的线程就不会继续等待，而是继续执行后续任务。

信号量分为命名信号量和未命名信号量，本实验介绍使用未命名信号量实现线程间的通信，

未命名信号量的使用流程如下：

* 使用sem\_t声明一个信号量；
* 使用sem\_init函数初始化信号量；
* 使用sem\_post函数发送信号量；
* 在等待信号量的线程内使用sem\_wait函数等待信号量，若需要设置等待期限需要使用sem\_timewait/sem\_reltimedwait\_np函数；
* 信号量使用完毕需要使用sem\_destroy函数将信号量销毁。

根据信号量的特点和使用流程，本实验例程中创建三个线程，分别是线程1、2、3，线程1向线程2、3分别发送信号量。其中线程1向线程2发送两次信号量，线程1向线程3发送3次信号量。在线程2、3中获取相关信号量。当线程2、3获取信号量的次数等于或小于线程1发送信号量次数的时候，程序会正常退出，当线程2获取信号量的次数大于线程1发送的信号量的时候，线程会被阻塞，无法退出。当线程3获取信号量的次数多于发送的信号量次数，线程3会等待10s，随后退出。

程序清单5.3 thread\_semaphore.c

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <semaphore.h>

#include <sys/time.h>

#include <pthread.h>

#include <time.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

函数声明和信号量声明

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void \*semsend(void \*arg); /\* 线程 \*/

void \*semrecv(void \*arg);

void \*semwait(void \*arg);

sem\_t sem1\_2; /\* 信号量 \*/

sem\_t sem1\_3;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

pthread\_t tid1, tid2, tid3;

int ret;

printf("this is main thread!\n");

ret = sem\_init(&sem1\_2, 0, 0); /\* 信号量 sem1\_2 初始化 \*/

if (ret != 0) {

printf("sem1\_2 init failed!\n");

return (-1);

}

ret = sem\_init(&sem1\_3, 0, 0); /\* 信号量 sem1\_3 初始化 \*/

if (ret != 0) {

printf("sem1\_3 init failed\n");

return (-1);

}

pthread\_create(&tid1, NULL, semsend, NULL); /\* 创建 \*/

pthread\_create(&tid2, NULL, semrecv, NULL);

pthread\_create(&tid3, NULL, semwait, NULL);

pthread\_join(tid1, NULL); /\* 等待线程结束 \*/

pthread\_join(tid2, NULL);

pthread\_join(tid3, NULL);

sem\_destroy(&sem1\_2); /\* 销毁信号量 \*/

sem\_destroy(&sem1\_3);

return (0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: semsend

\*\* 功能描述: 线程1 发送信号量线程

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void \*semsend (void \*arg)

{

printf("this is thread 1!\n");

sleep(2);

sem\_post(&sem1\_2); /\* 发送信号量 \*/

sem\_post(&sem1\_2);

sem\_post(&sem1\_3); /\* 发送信号量 \*/

sem\_post(&sem1\_3);

sem\_post(&sem1\_3);

printf("thread 1 completed!\n");

return (NULL);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: semrecv

\*\* 功能描述: 线程2 获取线程1发送的信号量

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void \*semrecv (void \*arg)

{

printf("this is thread 2!\n");

sem\_wait(&sem1\_2); /\* 等待信号量 \*/

sem\_wait(&sem1\_2);

printf("thread 2 completed!\n");

return (NULL);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: semwait

\*\* 功能描述: 线程3 获取线程1发送的信号量，若10s后还没接收到信号量，线程自动退出

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void \*semwait (void \*arg)

{

struct timespec ts;

int s;

printf("this is thread 3!\n");

sem\_wait(&sem1\_3); /\* 等待信号量 \*/

sem\_wait(&sem1\_3);

ts.tv\_nsec = 0;

ts.tv\_sec = time(NULL) + 10; /\* 设置 10s 钟等待超时 \*/

s = sem\_timedwait(&sem1\_3, &ts);

if(s == -1) {

printf("thread 3 wait timeout\n");

pthread\_exit(NULL);

}

printf("thread 3 completed!\n");

return (NULL);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译thread\_semaphore工程，将生成的可执行程序文件上传到验证平台；
2. 在验证平台上执行程序，控制台会输出程序执行结果；

[root@sylixos\_station:/apps/thread\_semaphore]# ./thread\_semaphore

this is main thread!

this is thread 1!

this is thread 2!

this is thread 3!

thread 1 completed!

thread 2 completed!

thread 3 completed!

[root@sylixos\_station:/apps/thread\_semaphore]#

1. 修改程序清单5.3中函数semsend，删除一行代码；

sem\_post(&sem1\_3);

重新编译执行程序，线程3等待超时。

[root@sylixos\_station:/apps/thread\_semaphore]# ./thread\_semaphore

this is main thread!

this is thread 1!

this is thread 2!

this is thread 3!

thread 1 completed!

thread 2 completed!

thread 3 wait timeout

[root@sylixos\_station:/apps/thread\_semaphore]#

1. 修改程序清单5.3中函数semsend，删除一行代码。

sem\_post(&sem1\_2);

重新编译并执行程序，程序无法正常退出（线程2无法正常退出）。

[root@sylixos\_station:/apps/thread\_semaphore]# ./thread\_semaphore

this is main thread!

this is thread 1!

this is thread 2!

this is thread 3!

thread 1 completed!

thread 3 wait timeout

### 扩展实验

使用信号量同步与使用全局变量同步有何区别，并编程验证。

## 实验三 线程同步实验—互斥锁

### 实验目的

* 了解互斥锁的概念和作用
* 掌握互斥锁相关函数和使用方法

### 实验内容

* 阅读thread\_mutex.c文件，了解互斥锁概念和相关函数使用方法
* 在RealEvo-IDE下编译thread\_mutex工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

互斥锁是避免在同一时间共享资源被多个线程同时访问。互斥锁在使用时类似一把锁，在访问共享资源前对其上锁，在访问完成后，将此资源解锁以便其他线程可以访问。

任何试图访问已经被上锁资源的的线程都会被阻塞，直至访问的资源被解锁。如果该资源解锁时有多个试图访问资源的线程被阻塞，那么只有一个线程会被唤醒访问共享资源，SylixOS唤醒原则采用基于优先级的判断，优先级高的优先被唤醒。

互斥锁在使用中根据初始化方式不同分为静态初始化和动态初始化，动态初始化互斥锁使用流程：

* 使用pthread\_mutex\_t声明互斥锁全局变量；
* 使用pthread\_mutex\_init初始化互斥锁；
* 线程中使用pthread\_mutex\_lock和pthread\_mutex\_unlock，对资源进行上锁和解锁；
* 不再需要互斥锁时，使用pthread\_mutex\_destroy销毁互斥锁；

使用互斥锁的时候需要避免出现死锁的情况。

本实验创建两个子线程，两个子线程共同访问同一个资源，通过互斥锁实现他们对资源访问的同步。

程序清单5.4 thread\_mutex.c

#include <stdio.h>

#include <string.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

全局变量声明

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

pthread\_mutex\_t lock; /\* 互斥锁 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: do\_print

\*\* 功能描述: 线程函数实体

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void\* do\_print (void \*arg)

{

int id = (int)arg;

int i;

pthread\_mutex\_lock(&lock); /\* 对资源上锁 \*/

printf("job %d started\n", id);

for (i = 0; i < 5; i++) {

printf("job %d printing\n", id); /\* 输出提示信息 \*/

sleep(1); /\* 延时1s \*/

}

printf("job %d finished\n", id);

pthread\_mutex\_unlock(&lock); /\* 对资源解锁 \*/

return (NULL);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

int i;

int ret;

pthread\_t tid[2];

if (pthread\_mutex\_init(&lock, NULL) != 0) { /\* 初始化互斥锁 \*/

printf("mutex init failed!\n");

return (-1);

}

for (i = 0; i < 2; i++) { /\* 创建线程 \*/

ret = pthread\_create(&(tid[i]), NULL, do\_print, (void \*)i);

if (ret != 0) {

printf("can't create thread !");

return (-1);

}

}

pthread\_join(tid[0], NULL); /\* 等待线程结束，并回收资源\*/

pthread\_join(tid[1], NULL);

pthread\_mutex\_destroy(&lock); /\* 销毁互斥锁 \*/

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译工程thread\_mutex，并将生成的程序文件上传到验证平台；
2. 在验证平台上执行程序thread\_mutex，控制台输出执行结果；

[root@sylixos\_station:/apps/thread\_mutex]# ./thread\_mutex

job 0 started

job 0 printing

job 0 printing

job 0 printing

job 0 printing

job 0 printing

job 0 finished

job 1 started

job 1 printing

job 1 printing

job 1 printing

job 1 printing

job 1 printing

job 1 finished

[root@sylixos\_station:/apps/thread\_mutex]#

1. 将程序清单5.4中的互斥锁相关代码屏蔽掉，再次编译运行程序，对比程序两次执行的效果，对互斥锁会有形象认识。

### 扩展实验

什么情况下会出现死锁，使用程序模拟死锁情形。

## 实验四 线程同步实验—条件变量

### 实验目的

* 了解条件变量的概念和作用
* 掌握条件变量相关函数及其使用方法

### 实验内容

* 阅读thread\_cond.c文件，了解条件变量概念和相关函数使用方法
* 在RealEvo-IDE下编译thread\_cond工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

互斥锁能够解决多线程中访问资源冲突的问题，其状态非常简单，只有上锁和解锁两种状态。当线程之间有其他需要同步的条件时，使用互斥锁会十分不方便。比如将线程设置为等待某一个条件满足后开始工作，这个时候使用互斥锁，只能频繁的查询该状态，这样会宝贵的CPU资源。这时使用条件变量是一个更合适的解决方案。

条件变量能够给多个线程提供一个会合的场所，条件变量与互斥锁一起使用，允许线程以无竞争的方式等待特定的条件发生。条件变量本身需要使用互斥锁保护，在改变条件变量之前需要使用互斥锁对其上锁。条件变量的使用流程与互斥锁类似：

* 使用pthread\_cond\_t声明全局条件变量；
* 使用pthread\_cond\_init函数初始化条件变量；
* 使用pthread\_cond\_wait函数阻塞等待条件变量的线程；
* 在其他线程里面使用pthread\_cond\_signal发送条件变量；
* 不再使用条件变量时，使用pthread\_cond\_destroy函数销毁条件变量。

条件变量同样分为静态初始化和动态初始化两种方式，本实验例程使用静态初始化的方式演示互斥锁和条件变量的使用。与动态初始化相比，静态初始化最大的局限性是其属性只能为默认属性，而动态初始化可以设置自定义属性。但静态初始化使用要更简单。

本实验例程中创建3个线程，线程1、2对一个全局变量进行加1操作，当全局变量的数值大于10的时候，线程1或2会发送条件变量给线程3，线程3会将此全局变量设置为0。延时一段时间后，三个线程自动退出。

程序清单5.5 thread\_cond.c

#include <stdio.h>

#include <string.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 全局变量声明

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int sum = 0;

pthread\_mutex\_t sumlock = PTHREAD\_MUTEX\_INITIALIZER; /\* 互斥锁 \*/

pthread\_cond\_t cond\_sum\_ready = PTHREAD\_COND\_INITIALIZER; /\* 条件变量 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: condsend

\*\* 功能描述: 发出条件变量的线程

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void \*condsend (void \*arg)

{

int i;

int num = (int)arg;

for (i = 0; i < 10; i++) {

pthread\_mutex\_lock(&sumlock); /\* 上锁 \*/

sum++;

printf("t%d: read sum value = %d\n", num + 1, sum);

if (sum >= 10) { /\* 满足条件发出条件变量 \*/

pthread\_cond\_signal(&cond\_sum\_ready);

}

pthread\_mutex\_unlock(&sumlock); /\* 解锁 \*/

/\*

\* 延时1s，以更好的观察线程调度情况

\*/

sleep(1);

}

return (NULL);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: condrecv

\*\* 功能描述: 等待条件变量线程

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void \*condrecv (void \*arg)

{

int i;

pthread\_mutex\_lock(&sumlock); /\* 上锁 \*/

for (i = 0;i < 2; i++) {

while (sum < 10) { /\* 等待条件变量 \*/

pthread\_cond\_wait(&cond\_sum\_ready, &sumlock);

}

sum = 0;

printf("t3: clear sum value\n");

}

pthread\_mutex\_unlock(&sumlock); /\* 解锁 \*/

return (NULL);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

int ret;

int i;

pthread\_t tid[3];

for (i = 0; i < 2; i++) { /\* 创建发出条件变量线程 \*/

ret = pthread\_create(&(tid[i]), NULL, condsend, (void \*)i);

if (ret != 0) {

printf("can't create thread !");

return (-1);

}

}

ret = pthread\_create(&(tid[2]), NULL, condrecv, NULL);/\* 创建等待条件变量线程 \*/

if (ret != 0) {

printf("can't create thread !");

return (-1);

}

for (i = 0; i < 3; i++) {

pthread\_join(tid[i], NULL); /\* 等待线程结束，并回收资源\*/

}

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译工程thread\_cond，将生成的程序文件上传到验证平台；
2. 在验证平台上执行程序thread\_cond，控制台输出程序执行结果；

[root@sylixos\_station:/apps/thread\_cond]# ./thread\_cond

t1: read sum value = 1

t2: read sum value = 2

t1: read sum value = 3

t2: read sum value = 4

t1: read sum value = 5

t2: read sum value = 6

t1: read sum value = 7

t2: read sum value = 8

t1: read sum value = 9

t2: read sum value = 10

t3: clear sum value

t1: read sum value = 1

t2: read sum value = 2

t1: read sum value = 3

t2: read sum value = 4

t1: read sum value = 5

t2: read sum value = 6

t1: read sum value = 7

t2: read sum value = 8

t1: read sum value = 9

t2: read sum value = 10

t3: clear sum value

[root@sylixos\_station:/apps/thread\_cond]#

### 扩展实验

对比互斥锁、条件变量静态初始化和动态初始化的区别，对程序运行有何影响并编程验证之。

# 多进程实验

本章主要讲述进程相关概念、进程的创建、退出和进程间的通信。针对进程的创建和进程间通信设计了3组实验，实验一主要讲述进程的创建，实验二和实验三讲述了进程间通信常用的2种方式。

## 实验一 进程管理实验

### 实验目的

* 了解SylixOS下进程的概念和原理
* 掌握进程创建、退出相关函数及其使用方法

### 实验内容

* 阅读proc\_manage.c文件，了解进程相关函数及使用方法
* 在RealEvo-IDE下编译proc\_manage工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

进程是一个可执行程序的实例。一个编译后的可执行程序在操作系统中运行后，就成为了一个进程，如果执行了两次，就创造了两个进程。从资源管理的角度看，一个进程会独占一些资源，如地址空间、程序代码、文件描述符表和程序运行的其他必要数据集合。其他进程中的函数无法访问此进程的资源，同样此进程的函数也无法访问其他进程中的资源。比如进程1打开了串口1，那么进程2就无法使用进程1打开的串口文件描述符向串口1收发数据。

SylixOS进程的实现方式与Linux/Windows有所不同，这些不同是为了保证SylixOS的实时性原则。但在使用方法上与POSIX标准进程没有太大区别。除不支持fork函数外，支持其他POSIX标准的进程管理相关函数。

进程在操作系统中使用进程号（进程ID）表示，进程ID是一个非负整数，用以唯一标识系统中的某个进程。

进程管理主要是指进程的创建和退出。在Linux/Unix中，进程的创建往往通过fork函数和exec函数配合使用来完成，fork函数从一个进程（父进程）复制出一个新的进程（子进程），随后子进程调用exec函数装载一个新的可执行文件替换当前进程代码，从而实现一个进程启动另外一个进程。但SylixOS中并不支持fork函数，因此在SylixOS系统中启动一个进程需要使用posix\_spawn函数。对大多数应用来讲posix\_spawn与fork+exec在功能上没有什么区别。

备注：posix\_spawn系列函数为POSIX实时扩展协议部分标准接口规范。

int posix\_spawn( pid\_t \*pid,

const char \*path,

const posix\_spawn\_file\_actions\_t \*file\_actions,

const posix\_spawnattr\_t \*attrp,

char \*const argv[],

char \*const envp[]);

* 函数执行成功返回值为0，执行不成功返回错误号
* pid 存放子进程的进程ID
* path 可执行程序文件路径
* file\_actions 子进程启动时需要执行文件操作集合，可为NULL
* attrp 设定子进程创建属性，可为NULL
* argv 子进程的执行参数
* envp 子进程环境变量

waitpid函数（wait系列函数之一，详见sys/wait.h）是与posix\_spawn函数经常配合使用的一个函数，waitpid用来等待一个特定的子进程结束，回收子进程的资源并返回子进程结束状态。

pid\_t waitpid( pid\_t pid,

int \*stat\_loc,

int options );

* 函数返回值为已停止子进程的进程ID，函数执行错误返回-1
* pid 标识等待结束的子进程ID，支持以下几种方式：

> 0 等待指定PID的子进程

= 0 子进程中组ID相同的

= -1 等待任意子进程

< -1子进程中组ID为pid绝对值的子进程

* stat\_loc 返回进程结束的状态
* options 功能选项位，由位掩码组成，支持的掩码位有：

WNOHANG 不阻塞，如果有子进程结束则回收，无则直接退出

WUNTRACED 若子进程进入暂停状态，则马上返回，但子进程的结束状态不予以理会。WIFSTOPPED(status)宏确定返回值是否对应与一个暂停子进程

与进程退出相关的函数有kill和exit，其中kill函数是用来向进程发送信号，其中有些信号可以杀死进程，而exit是实现进程自身退出的API。这里我们主要介绍exit的使用，exit的使用比较简单，通常情况下在需要退出进程的地方使用exit函数即可。

本实验例程使用posix\_spawn函数启动一个helloworld进程，随后输出父进程和子进程的进程ID，并在实验结束时验证exit函数的作用。

程序清单6.1 proc\_manage.c代码

#include <spawn.h>

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 全局变量

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

extern char \*\*environ;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

int ret ;

int stat ;

pid\_t pid ;

pid\_t child\_pid ;

char \* args[3] = {"hello", "-l", NULL};

posix\_spawnattr\_t attr ;

posix\_spawn\_file\_actions\_t fact ;

posix\_spawnattr\_init(&attr); /\* 初始化 spawn 函数参数 \*/

posix\_spawn\_file\_actions\_init(&fact);

ret = posix\_spawn(&child\_pid, "./helloworld", /\* 创建子进程 \*/

&fact, &attr, args, environ);

if (ret < 0) {

printf("child process create faile");

return (-1);

}

printf("pid=%d, child pid = %d\n", /\* 输出子父进程 ID \*/

getpid(), child\_pid);

pid = waitpid(child\_pid, &stat, 0); /\* 等待子进程结束 \*/

printf("%d stat is %d\n", pid, stat);

posix\_spawnattr\_destroy(&attr);

posix\_spawn\_file\_actions\_destroy(&fact);

exit(0);

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译工程proc\_manage，将生成的程序文件上传到验证平台；
2. 将第1章实验程序文件helloworld拷贝到proc\_manage程序所在目录；
3. 在验证平台上执行程序proc\_manage，控制台输出程序执行结果；

[root@sylixos\_station:/apps/proc\_manage]# ls

helloworld proc\_manage

[root@sylixos\_station:/apps/proc\_manage]# ./proc\_manage

pid=15,child pid = 16

Hello World!

16 stat is 0

[root@sylixos\_station:/apps/proc\_manage]#

Hello World！字符是程序helloworld输出。

### 扩展实验

结合文件操作实现程序的自动升级，比如使用SD卡实现自动升级SylixOS中的应用程序。

## 实验二 进程间通信实验—共享内存

### 实验目的

* 了解共享内存、信号量的概念和作用
* 掌握共享内存、信号量相关函数和使用方法

### 实验内容

* 阅读proc\_mmap\_client.c、proc\_mmap\_server.c文件，了解共享内存、信号量相关函数和使用方法
* 在RealEvo-IDE下编译proc\_mmap\_client和proc\_mmap\_server工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

共享内存是允许两个或多个进程读写一个共享的物理存储区域。因为数据不需要在进程间进行复制，因此共享内存是速度最快的进程间通信方式。但共享内存本身也存在一些不足，由于共享内存的数据会被两个或多个进程操作，如果两个进程同时向共享内存写入数据，会产生冲突。而且当一个进程向共享内存区域写入数据后，另外的进程也无法“知道”共享内存的数据发生了变化。因此共享内存通常会与另一种进程通信方式如：信号量组合使用。由信号量实现两个进程的同步，共享内存实现数据的交换。

信号量分为互斥信号量与普通信号量，互斥信号量用来解决多个进程（或线程）对共享数据对象访问时的互斥问题；普通信号量（计数信号量或二值信号量）用来解决（进程）线程直接同步通信问题。可以把信号量理解为是一个特殊的变量，程序对其访问都是原子操作，且只允许对它进行发送和获取操作。当信号量被发送一次，其数值就增加1，当信号量大于0的时候，等待此信号量的进程会从阻塞状态转为就绪状态，并将信号量减1。进程间通过信号量来实现简单的同步。信号量有命名信号量和匿名信号量，本实验使用命名信号量实现进程间的同步。

共享内存的使用流程：

* 创建打开共享内存文件并设置文件大小；
* 将内存文件映射到实际的物理内存上；
* 创建信号量实现使用共享内存通信的两个进程间的同步；
* 对内存进行读写操作；
* 取消共享内存的内存映射、关闭共享内存文件。

本实验例程使用共享内存实现两个进程间的数据交换，使用信号量实现两个进程的同步。本实验进程中一个作为服务器、一个作为客户端，服务器接受客户端传递的数据。

**代码分析—client**

收到控制台输入数据后，将数据写入共享内存，并通过信号量通知server进程读取。

程序清单6.2 proc\_mmap\_client.c代码

#include <stdio.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <sys/mman.h>

#include <unistd.h>

#include <semaphore.h>

#include <string.h>

#define MAPSIZE 100 /\* 共享内存大小 \*/

#define MAPNAME "test" /\* 共享内存名称 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

int shmfd;

char \*ptr;

sem\_t \*semfd;

printf("client input:");

fflush(stdout); /\* 刷新缓冲区，将字符输出 \*/

shmfd = shm\_open(MAPNAME, O\_RDWR, 0); /\* 创建共享内存文件 \*/

if (shmfd < 0) {

printf("open shared memory error\n");

return (-1);

}

ftruncate(shmfd, MAPSIZE); /\* 设置共享内存大小 \*/

semfd = sem\_open(MAPNAME, 0); /\* 创建信号量 \*/

if (semfd == SEM\_FAILED) {

printf("open semaphore error\n");

return (-1);

}

/\*

\* 将共享内存文件映射的实际物理内存

\*/

ptr = mmap(NULL, MAPSIZE, PROT\_READ | PROT\_WRITE, MAP\_SHARED, shmfd, 0);

if (ptr == MAP\_FAILED) {

printf("mmap failed");

return (-1);

}

fgets(ptr, MAPSIZE, stdin); /\* 等待控制台的数据输入 \*/

sem\_post(semfd); /\* 发送信号量 \*/

munmap(ptr, MAPSIZE); /\* 取消共享内存映射 \*/

close(shmfd); /\* 关闭共享内存文件 \*/

sem\_close(semfd); /\* 关闭信号量 \*/

sleep(1);

return (0);

}

**代码分析—server**

接收到cilent进程发送的信号量后，读取共享内存的数据并输出显示。

程序清单 6.3 proc\_mmap\_server.c

#include <stdio.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <sys/mman.h>

#include <unistd.h>

#include <semaphore.h>

#include <string.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 宏定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define MAPSIZE 100 /\* 共享内存大小 \*/

#define MAPNAME "test" /\* 共享内存名称 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

int shmfd;

char \*ptr;

sem\_t \*semfd;

shmfd = shm\_open(MAPNAME, O\_RDWR | O\_CREAT, 0644); /\* 创建共享内存 \*/

if (shmfd < 0) {

printf("open shared memory error\n");

return (-1);

}

ftruncate(shmfd, MAPSIZE); /\* 设置共享内存大小 \*/

semfd = sem\_open(MAPNAME, O\_CREAT, 0644, 0); /\* 创建信号量 \*/

if (semfd == SEM\_FAILED) {

printf("open semaphore error\n");

return (-1);

}

/\*

\* 将共享内存文件映射的实际物理内存

\*/

ptr = mmap(NULL, MAPSIZE, PROT\_READ | PROT\_WRITE, MAP\_SHARED, shmfd, 0);

if (ptr == MAP\_FAILED) {

printf("mmap failed");

return (-1);

}

strcpy(ptr, "\0");

/\*

\* 等待信号量,将共享内存里面的信息输出到控制台

\*/

sem\_wait(semfd);

printf("\nserver recv:%s", ptr);

strcpy(ptr, "\0");

munmap(ptr, MAPSIZE); /\* 取消共享内存映射 \*/

close(shmfd); /\* 关闭共享内存文件 \*/

sem\_close(semfd); /\* 关闭信号量 \*/

sem\_unlink(MAPNAME); /\* 取消信号量 \*/

shm\_unlink(MAPNAME); /\* 取消共享内存 \*/

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译工程proc\_mmap\_server、proc\_mmap\_client，将生成的程序文件上传到验证平台。修改上传文件目录，将文件传送目录均设置为/apps/proc\_mmap；
2. 首先执行程序proc\_mmap\_server，执行命令添加“&”字符，表示在后台运行，随后执行程序proc\_mmap\_client；
3. 根据提示字符输入数据，server会输出接收到的数据。

[root@sylixos\_station:/apps/proc\_mmap]# ./proc\_mmap\_server &

[root@sylixos\_station:/apps/proc\_mmap]# ./proc\_mmap\_client

client input:hello this is mmap client

server recv:hello this is mmap client

[root@sylixos\_station:/apps/proc\_mmap]#

### 扩展实验

不使用信号量同步，有无其他同步共享内存的方式，效率与信号量有何区别，编程验证之。

## 实验三 进程间通信实验—消息队列

### 实验目的

* 了解消息队列的概念和作用
* 掌握消息队列相关函数和使用方法

### 实验内容

* 阅读proc\_msg\_client.c、proc\_msg\_server.c文件，了解消息队列相关函数及使用方法
* 在RealEvo-IDE下编译proc\_msg\_client和proc\_msg\_server工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

消息队列是内核提供给各个进程之间传递消息的通信机制。消息队列的使用流程：

* 使用mq\_open 函数创建消息队列，创建队列时会设置消息队列的操作权限和消息关联的键；
* 使用mq\_send 函数向消息队列发送一个消息；
* 使用mq\_receive 函数从消息队列中读取一个消息；
* 如果不再需要消息通信，使用mq\_close 函数关闭此消息队列；
* 使用mq\_unlink函数删除此消息队列。

本实验使用消息实现两个进程间的通信，进程分为服务器和客户端。客户端工程用来发送消息，服务器进程接收消息并将接收到的消息内容输出显示，当消息内容是“exit”时，两个进程退出。

在客户端程序里，是打开消息队列而非创建，在退出后对应操作是关闭消息队列。消息队列的创建和删除均在服务器程序里进行。

**程序代码—client**

用来接收控制台输入数据，并将数据发送到消息队列。

程序清单 6.4 proc\_msg\_client.c

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <sched.h>

#include <mqueue.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 宏定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define QUEUE\_NAME "test\_queue"

#define MAX\_SIZE 1024

#define MSG\_STOP "exit"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*\*argv)

{

int ret;

mqd\_t mq;

char buffer[MAX\_SIZE];

mq = mq\_open(QUEUE\_NAME, O\_WRONLY); /\* 打开一个消息队列 \*/

if (mq == MQ\_FAILED) {

printf("open msg faile");

return (-1);

}

printf("send to server (enter \"exit\" to stop it):\n");

/\*

\* 循环接收数据并发送，获取数据 exit 后退出程序

\*/

while (1) {

printf("client input: ");

fflush(stdout); /\* 刷新缓冲区 \*/

memset(buffer, 0, MAX\_SIZE);

fgets(buffer, MAX\_SIZE, stdin); /\* 获取控制台输入 \*/

ret = mq\_send(mq, buffer, MAX\_SIZE, 0); /\* 通过消息队列发送数据 \*/

if (ret < 0) {

printf("message send faile");

break;

}

/\* 检验是否是 exit \*/

ret = strncmp(buffer, MSG\_STOP, strlen(MSG\_STOP));

if (ret == 0) {

break;

}

sched\_yield();

}

mq\_close(mq); /\* 关闭消息队列 \*/

return (0);

}

**程序代码—server**

接收消息并显示消息内容。

程序清单 6.5 proc\_msg\_server.c

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <mqueue.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 宏定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define QUEUE\_NAME "test\_queue"

#define MAX\_SIZE 1024

#define MSG\_STOP "exit"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*\*argv)

{

mqd\_t mq;

char buffer[MAX\_SIZE + 1];

ssize\_t bytes\_read;

int ret;

struct mq\_attr attr;

attr.mq\_flags = 0;

attr.mq\_maxmsg = 10; /\*消息队列容纳 10 个消息 \*/

attr.mq\_msgsize = MAX\_SIZE; /\* 每条消息的最大字节数 \*/

attr.mq\_curmsgs = 0;

mq = mq\_open(QUEUE\_NAME, O\_CREAT | O\_RDONLY, /\* 创建一个消息队列 \*/

0644, &attr);

if (mq == MQ\_FAILED) {

printf("open msg failed\n");

return (-1);

}

while (1) { /\* 从消息队列读取消息 \*/

bytes\_read = mq\_receive(mq, buffer, MAX\_SIZE, NULL);

if (bytes\_read <= 0) {

printf("message recv faile\n");

break;

}

buffer[bytes\_read] = '\0';

ret = strncmp(buffer, MSG\_STOP, strlen(MSG\_STOP));/\* 检验是否是 exit \*/

if (ret == 0) {

break;

}

printf("received: %s\n", buffer);

}

mq\_close(mq); /\* 关闭消息队列 \*/

mq\_unlink(QUEUE\_NAME); /\* 删除消息队列 \*/

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译工程proc\_msg\_server、proc\_msg\_client，将生成的程序文件上传到验证平台，修改上传文件目录，将文件均传送到/apps/proc\_msg目录；
2. 首先执行程序proc\_msg\_server，执行命令添加“&”字符，让其在后台运行，随后执行程序proc\_msg\_client；
3. 根据提示字符输入数据，server输出其接收到的数据。

[root@sylixos\_station:/apps/proc\_msg]# ./proc\_msg\_server &

[root@sylixos\_station:/apps/proc\_msg]# ./proc\_msg\_client

send to server (enter "exit" to stop it):

client input: hello this is msg from client

received: hello this is msg from client

client input: exit

[root@sylixos\_station:/apps/proc\_msg]#

### 扩展实验

消息队列与共享内存相比有什么优点和不足，编程验证之。

# 串口收发实验

串口是指RS232-C接口，串口几乎是计算机领域最简单的通信接口，也是使用最广泛的通信接口。虽然现在在PC主机上已经很少看到，但在嵌入式领域中通常是必备接口。在实际应用中，由于电气特性不同，串口有多种应用方式，如TTL电平、标准RS232电平、RS422差分电平等，但对软件编程基本没有影响。

## 实验一 串口收发实验

### 实验目的

* 了解串口使用需要的配置和通信流程
* 掌握串口参数配置函数和使用方法

### 实验内容

* 阅读uart\_test.c代码，了解串口通信需要设置的参数和相关函数使用方法
* 在RealEvo-IDE下编译uart\_test工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty、串口调试助手

### 实验原理

在SylixOS中串口是以文件的方式被管理，操作流程与普通文件操作流程相似。由于串口通信需要配置一些与通信相关的参数 ，串口编程还需要一些额外的函数配合。

串口操作流程为：

* 使用open函数打开串口设备，串口设备常用/dev/ttyS\*表示；
* 设置串口的波特率；
* 设置串口的数据位、停止位、校验等；
* 使用read/write函数对串口进行读写操作；
* 串口使用完毕后，使用close函数关闭设备。

本实验例程完成一个串口基本的收发通信功能，当串口接收到外部输入数据后，会再将数据通过本串口发出。本例程中串口参数的配置相关代码封装成了独立的函数，这些函数可以在其它工程中直接使用。

程序清单 7.1 uart\_test.c文件的main函数

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <errno.h>

#include <string.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

int ret;

int nread;

int fd\_tty;

char buff[512];

/\*

\* 参数个数小于2则返回

\*/

if (argc < 2) {

printf("useage: %s dev\n", argv[0]);

return (-1);

}

fd\_tty = open(argv[1], O\_RDWR); /\* 打开串口 \*/

if (fd\_tty < 0) {

printf("open device %s failed\n", argv[1]);

return (-1);

}

ret = set\_speed(fd\_tty, 115200); /\* 设置串口波特率 \*/

if (ret != 0) {

close(fd\_tty);

return (-1);

}

ret = set\_parity(fd\_tty, 8, 1, 'n'); /\*数据位 8，停止位 1，无校验\*/

if (ret != 0) {

close(fd\_tty);

return (-1);

}

/\*

\* 在 while 循环中，不断的检查数据，若收到数据，将数据通过此串口发送回主机

\*/

while(1) {

nread = read(fd\_tty,buff, 255);

if (nread > 0) { /\* 接收数据 \*/

write(fd\_tty, buff, nread); /\* 将数据通过原串口输出 \*/

}

}

sync();

close(fd\_tty);

return (0);

}

设置波特率的函数set\_speed原型如程序清单 7.2，代码中有详细注释，可以详细了解串口波特率的设置流程和方法。

程序清单 7.2 uart\_test.c串口速率设置函数set\_speed

int set\_speed (int fd, int speed)

{

int ret;

struct termios opt;

ret = tcgetattr(fd, &opt); /\*获取串口的 termios 信息 \*/

if (ret != 0) {

printf("Get Serial speed failed");

return (1);

}

tcflush(fd, TCIOFLUSH); /\* 刷新输入输出数据 \*/

cfsetispeed(&opt, speed); /\* 设置输入波特率 \*/

cfsetospeed(&opt, speed); /\* 设置输出波特率 \*/

/\*

\* 用于设置终端的相关参数, TCSANOW 不等数据传输完毕就立即改变属性

\*/

ret = tcsetattr(fd, TCSANOW, &opt);

if (ret != 0) {

printf("Setup Serial speed failed");

return (2);

}

return (0);

}

设置数据位等参数的函数set\_parity详细代码如程序清单 7.3。

程序清单 7.3 uart\_test.c串口参数设置函数set\_parity

int set\_parity (int fd, int databits, int stopbits, int parity)

{

int ret;

struct termios options;

ret = tcgetattr(fd, &options);

if (ret != 0) {

printf("Setup Serial failed");

return(1);

}

options.c\_cflag &= ~CSIZE;

switch (databits) { /\* 设置数据位数 \*/

case 5:

options.c\_cflag |= CS5;

break;

case 6:

options.c\_cflag |= CS6;

break;

case 7:

options.c\_cflag |= CS7;

break;

case 8:

options.c\_cflag |= CS8;

break;

default:

printf("Unsupported data size\n");

return (2);

}

switch (parity) { /\* 设置校验方式 \*/

case 'n':

case 'N':

options.c\_cflag &= ~PARENB;

options.c\_iflag &= ~INPCK;

break;

case 'o':

case 'O':

options.c\_cflag |= (PARODD | PARENB);

options.c\_iflag |= INPCK;

break;

case 'e':

case 'E':

options.c\_cflag |= PARENB;

options.c\_cflag &= ~PARODD;

options.c\_iflag |= INPCK;

break;

default:

printf("Unsupported parity\n");

return (3);

}

switch (stopbits) { /\* 设置停止位 \*/

case 1:

options.c\_cflag &= ~CSTOPB;

break;

case 2:

options.c\_cflag |= CSTOPB;

break;

default:

printf("Unsupported stop bits\n");

return (4);

}

tcflush(fd, TCIOFLUSH); /\* 清除串口收发缓冲区 \*/

/\*

\* 用于设置终端的相关参数,TCSANOW：不等数据传输完毕就立即改变属性

\*/

ret = tcsetattr(fd, TCSANOW, &options);

if (ret != 0) {

printf("Setup Serial failed");

return (5);

}

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译uart\_test工程，将生成的工程文件上传到验证平台并执行；

[root@sylixos\_station:/apps/uart\_test]# ./uart\_test /dev/ttyS1

1. 将与PC主机串口连接的DB9更换到验证平台另外一个串口DB9上，由于验证平台会把收到的数据发送回PC主机，因此向控制台中输入任意字符，都会被回显到控制台上；
2. 将串口线与验证平台断开，在控制台中输入任意字符，都不会有反应，证明上一步骤中的字符确实为验证平台自身串口返回数据；
3. 程序设计没有添加自动退出的功能，回到SylixOS的shell控制台，使用“Ctrl+C”可以退出串口测试例程。

### 扩展实验

编程实现串口shell服务程序，支持常用的ls/mkdir/rmdir/cd/rm/cp等功能。

# 网络通信实验

以太网接口是目前应用最为广泛的高速通信接口，其通信速度通常为10Mbps、100Mbps、1000Mbps。一般以太网接口配合TCP/IP协议使用（也可以用于其他协议栈）。

在本章中使用TCP/IP协议栈进行以太网通信，但对TCP/IP协议内部原理不做深入分析，主要讲解TCP/IP协议的应用，TCP/IP协议相关知识读者可以参考其他专业书籍，在不同的操作系统下TCP/IP协议的应用接口基本相同（socket）。本章设计了四个实验，分别讲述了UDP、TCP客户端、TCP服务器、web服务器的编程和使用。

## 实验一 UDP实验

### 实验目的

* 了解UDP通信的特点和工作流程
* 掌握socket接口中UDP操作相关函数及使用方法

### 实验内容

* 阅读udp\_test.c内的程序代码，了解UDP通信流程及socket函数使用方法
* 在RealEvo-IDE下编译udp\_test工程，将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

UDP是不可靠的通信方式，它不保证通信的可靠性与有序性，其突出优点是数据传输效率高，节约系统资源，可以在开销很小的情况下实现单点对多点的通信。尤其是UDP较高的传输效率在很多对简单的丢包不敏感的场合有着非常大的优势。

UDP本身不分为客户端和服务端模型、但为了程序设计方便，在逻辑上我们一般分为客户端和服务端模型。服务器主要包含建立套接字、将套接字与IP地址绑定、读写数据、关闭套接字等过程。客户端包含建立套接字、读写数据、关闭套接字等过程。

图 8.1 UDP通信流程图

UDP通信流程如图 8.1所示，首先调用socket函数创建套接字，然后调用bind函数绑定本地地址，随后使用sendto、recvfrom函数发送和接收数据。图 8.1中可以看到UDP的服务器编程和客户端编程并没有本质区别，实现的工作内容基本相同。在UDP通信流程中，调用bind函数的过程可以省略（协议栈将随机分配本地端口号），建立套接字后，可以直接进行数据收发。

socket编程分为阻塞模式和非阻塞模式。所谓阻塞模式是当套接字缓冲区内没有数据可读时，recvfrom函数会让其所在线程进入睡眠状态，直到数据到来。线程睡眠后，CPU可以去处理其他任务，因此阻塞模式下能节约系统资源，在SylixOS中默认是以阻塞方式工作。但有些情况下需要以非阻塞方式工作，比如线程在执行监督任务，需要返回套接字缓冲区有无数据状态或者使用select函数与其他文件描述符进行复用阻塞。socket阻塞与非阻塞模式可以使用ioctl/fcntl等函数设置。

本实验例程实现了一个基本的UDP通信，程序运行后，使用UDP协议发送一组数据，随后等待接收从以太网上发送过来的UDP数据包。接到UDP数据包后将数据显示到控制台上，随后程序退出。

**代码分析：**

程序清单 8.1 udp\_test.c

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

宏定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define SERVER\_IP "192.168.1.55" /\* PC主机的IP地址 \*/

#define SERVER\_PORT (8000) /\* PC主机的IP端口 \*/

#define BUFF\_SIZE (1024 \* 4) /\* 通信缓冲区长度 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

int ret;

int sockfd;

struct sockaddr\_in server\_addr;

char test\_str[BUFF\_SIZE] = "this is udp\_test";

socklen\_t addr\_len = sizeof(server\_addr);

/\*

\* 建立UDP的socket(套接字)

\*/

sockfd = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP);

if (sockfd < 0) {

printf("socket error");

return (-1);

}

/\*

\* 设置IP地址、端口，

\*/

(void)memset(&server\_addr, 0, sizeof(server\_addr));

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = inet\_addr(SERVER\_IP);

server\_addr.sin\_port = htons(SERVER\_PORT);

/\*

\* 发送数据包

\*/

ret = sendto(sockfd, test\_str, strlen(test\_str), 0,

(struct sockaddr \*)&server\_addr, addr\_len);

if (ret < 0) {

printf("send data error");

close(sockfd);

return (-1);

}

(void)memset(test\_str, 0, BUFF\_SIZE); /\* 将接收缓冲区清零 \*/

/\*

\* 接收数据，

\*/

ret = recvfrom(sockfd, test\_str, BUFF\_SIZE, 0,

(struct sockaddr \*)&server\_addr, &addr\_len);

if (ret < 0) {

close(sockfd);

return (-1);

}

/\*

\* 将接收到的字符和字符的来源输出显示

\*/

printf("get echo from %s:%hu\n", inet\_ntoa(server\_addr.sin\_addr),

ntohs(server\_addr.sin\_port));

printf("%s\n", test\_str);

close(sockfd);

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译工程udp\_test，将生成的序文件上传到验证平台；
2. 在PC主机上打开“网络调试助手”，设置相关参数；

* 协议类型：UDP
* 本地IP地址：PC主机的IP地址
* 本地端口号：与程序中设置一直，默认8000

完成参数设置后，点击“连接按钮”，如图 8.2。



图 8.2 网络调试助手界面

1. 在验证平台上运行udp\_test程序，此时在PC主机的网络调试助手中会收到验证平台发送信息如图 8.2；
2. 点击“发送”按钮，验证平台控制台输出接收数据信息；

[root@sylixos\_station:/apps/udp\_test]# ./udp\_test

get echo from 192.168.1.55:8000

hello this is pc

[root@sylixos\_station:/apps/udp\_test]#

### 扩展实验

参考多线程实验例程，将本实验中的socket数据接收工作放在一个单独线程里实现。

## 实验二 TCP客户端实验

### 实验目的

* 了解TCP客户端的特点和工作流程
* 掌握TCP客户端的设计和使用方法

### 实验内容

* 阅读tcp\_clien.c程序代码，了解TCP客户端的设计方法和socket函数使用
* 使用RealEvo-IDE编译tcp\_client工程，将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

TCP在两个端点（应用程序）之间提供了可靠的、面向连接通信通道。在开始通信之前，TCP需要在两个端点间建立连接。在通信过程中，TCP接收到一数据包会向发送者发送确认信息，告知发送者数据已经传送完成。当数据包有错误，该TCP数据包会被丢失，接收者也不会发送任何信息，发送者在一段时间内没有收到接收者的确认，会重传此数据包。

TCP通信还具备排序功能、流量控制、拥塞控制。排序功能能够保证传输的数据能够以正确的顺序在目的地接收。流量控制是为了防止一个快速的发送者将一个慢速的接收者压垮。拥塞控制是防止一个快速的发送者压垮整个网络。

TCP协议的通信流程如所示，一个完整的TCP通信链路需要有TCP服务器和TCP客户端，而且TCP服务器要先于TCP客户端执行。在TCP通信之前，需要TCP服务器与TCP客户端建立链接，但TCP通信效率低于UDP。TCP最为突出的优点是其通信的可靠性，虽然使用UDP加自定义的校验协议也可以提高通信的可靠性，这样的效率和通信的可靠性都不如TCP协议本身。

如图 8.3所示，TCP通信中TCP客户端的编程较为简单，建立套接字后，调用connect函数请求与TCP服务器建立连接。TCP使用的收发函数一般是recv、send，相对recvfrom、sendto前者使用更为简单。

****

图 8.3 TCP通信流程图

本实验例程实现一个基本的TCP客户端功能，程序运行后向服务器发送建立连接请求信号，建立连接后发送一组数据，随后等待TCP服务器回复数据，接收到TCP服务器回复数据后，程序退出。

**代码分析：**

程序清单 8.2 tcp\_client.c

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <errno.h>

#include <stdbool.h>

#include <stdint.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

宏定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define SERVER\_IP "192.168.1.55" /\* 服务器的IP地址 \*/

#define SERVER\_PORT (8000) /\* 服务器的TCP/IP端口 \*/

#define BUFF\_SIZE (1024 \* 4)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

int ret;

int sockfd;

struct sockaddr\_in server\_addr;

char test\_str[BUFF\_SIZE] = "this is tcp client";

/\*

\* 建立套接字

\*/

sockfd = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

if (sockfd < 0) {

printf("socket error");

return (-1);

}

/\*

\* 设置连接的服务器 IP 地址、端口号

\*/

(void)memset(&server\_addr, 0, sizeof(server\_addr));

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = inet\_addr(SERVER\_IP);

server\_addr.sin\_port = htons(SERVER\_PORT);

/\*

\* 向服务器发送连接请求，若没有服务器，会返回错误。

\*/

ret = connect(sockfd, (struct sockaddr \*)&server\_addr, sizeof(server\_addr));

if (ret < 0) {

printf("connect error");

close(sockfd);

return (-1);

}

/\*

\* 与服务器建立连接后，发送默认字符

\*/

ret = send(sockfd, test\_str, strlen(test\_str), 0);

if (ret < 0) {

printf("send data error");

close(sockfd);

return (-1);

}

(void)memset(test\_str, 0, BUFF\_SIZE);

/\*

\* 接受到服务器下发数据后，将其显示出来，

\*/

ret = recv(sockfd, test\_str, BUFF\_SIZE, 0);

if (ret < 0) {

printf("receive data error");

close(sockfd);

return (-1);

}

printf("%s\n", test\_str);

close(sockfd);

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译工程tcp\_client，将工程可执行文件上传到验证平台；
2. 在PC主机上打开“网络调试助手”，设置相关参数；

* 协议类型：TCP Server
* 本地IP地址：PC主机的IP地址
* 本地端口号：与程序中设置一致，默认8000
* 完成参数设置后，点击“连接按钮”，如图 8.4

1. 在验证平台上运行tcp\_client程序，此时在PC主机的网络调试助手中会收到验证平台发送信息如图 8.4；

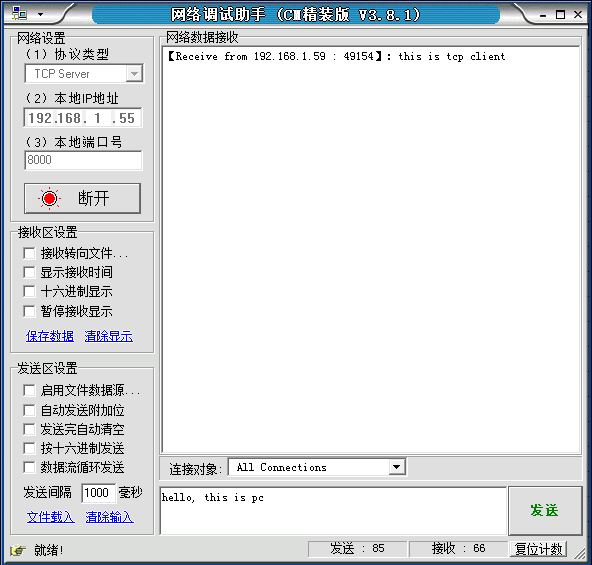


图 8.4 网络调试助手界面

1. 点击“发送”按钮，会看到验证平台控制台输出接收数据信息。

[root@sylixos\_station:/apps/tcp\_client]# ./tcp\_client

hello this is pc

[root@sylixos\_station:/apps/tcp\_client]#

### 扩展实验

参考多线程实验例程，将本实验中的socket数据接收工作放在一个单独线程里实现。

## 实验三 TCP服务器实验

### 实验目的

* 了解TCP服务器的特点和工作流程
* 掌握TCP服务器的设计和使用方法

### 实验内容

* 阅读tcp\_server.c程序代码，了解TCP服务器的设计方法和函数使用
* 在RealEvo-IDE下编译tcp\_server工程，将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

TCP的通信流程如图 8.3所示。相比较TCP客户端，TCP服务器的实现要稍复杂些，主要原因是TCP服务器要能够同时连接多个客户端，而客户端只需要考虑连接一个服务器。在TCP服务器的编程实现上，始终会有一个任务在监听网络，等待TCP客户端发送的连接请求，在与一个TCP客户端建立连接后，会开启单独的任务处理此TCP客户端的通信，原有任务继续监听连接请求。

在很多Linux/Unix编程中，使用多任务（进程或者线程）方式实现TCP服务器的设计。本实验采用多线程的方式，实现TCP服务器的设计。

本实验例程实现了一个具备基本功能的TCP服务器，在程序执行后，会监听socket接口的连接请求，当接收到一个连接请求后，创建一个线程来管理此连接。在主线程中会继续监听其他连接请求。在连接处理线程中，服务器会将接收到的数据再发送回客户端（echo），实现基本的TCP收发通信。

**代码分析：**

程序清单 8.3 tcp\_server.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <arpa/inet.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <unistd.h>

#include <pthread.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

宏定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define PORT 8000 /\* 服务器的端口号 \*/

#define BUFSIZE 512

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

int ret;

int sockfd;

struct sockaddr\_in server\_addr;

sockfd = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP); /\* 创建套接字 \*/

if (sockfd < 0) {

printf("socket failed");

return (-1);

}

server\_addr.sin\_family = AF\_INET; /\* 设置通信参数 \*/

server\_addr.sin\_port = htons(PORT);

server\_addr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

ret = bind(sockfd, (struct sockaddr\*)&server\_addr, sizeof(server\_addr));

if (ret < 0) {

printf("bind error!\n");

return (-1);

}

listen(sockfd, 128); /\* 监听 \*/

while (1) {

int newfd = accept(sockfd, NULL, NULL); /\* 建立连接 \*/

pthread\_t tid;

ret = pthread\_create(&tid, NULL, tcp\_fun, (void\*)newfd);

if (ret != 0) {

printf("create tcp sever thread failed");

close(newfd);

break;

}

}

close(sockfd);

return (0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: \*tcp\_fun

\*\* 功能描述: 服务器建立连接的线程

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void \*tcp\_fun (void \*arg)

{

int ret;

char buf[BUFSIZE + 1];

int new\_fd = (int)arg;

while (1) {

ret = read(new\_fd, buf, BUFSIZE); /\* 读取套接字接收数据 \*/

if (ret <= 0) {

printf("client close !");

break;

}

buf[ret] = '\0';

printf("%s\n", buf); /\* 显示接收数据 \*/

write(new\_fd, buf, strlen(buf)); /\* 将收到的数据返回去 \*/

}

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译工程tcp\_server工程，将工程可执行文件上传到验证平台；
2. 在PC主机上打开“网络调试助手”，设置相关参数；

* 协议类型：TCP Client
* 本地IP地址：PC主机的IP地址，
* 本地端口号：与程序中设置一致，默认8000
* 完成参数设置后，点击“连接”按钮，如图 8.5 TCP 客户端的配置

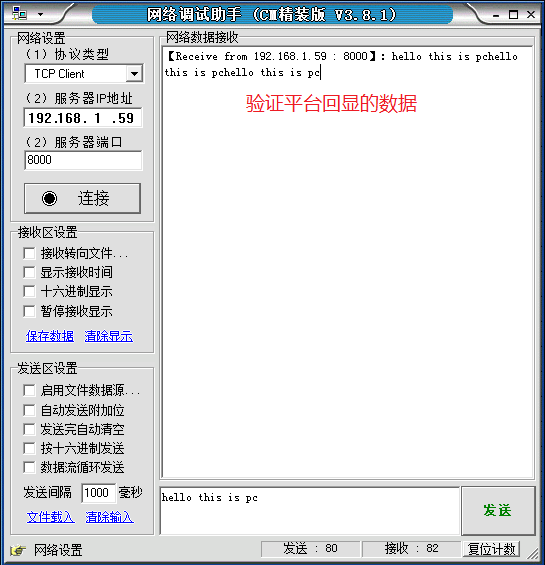


图 8.5 TCP 客户端的配置

1. 在验证平台上执行程序tcp\_server，之后点击图 8.5 TCP 客户端的配置中的“连接”按钮。建立连接“连接”按钮会字样变为“断开”，颜色也会变为红色；
2. 点击“发送”按钮，在网络调试助手上会看到返回数据，如图 8.5 TCP 客户端的配置所示，在验证平台控制台同样也会有字符回显，如果点击“断开”按钮，控制台会有提示信息。

[root@sylixos\_station:/apps/tcp\_server]# ./tcp\_server

hello this is pc

hello this is pc

hello this is pc

client close !

### 扩展实验

使用多个客户端与服务器同时建立连接，系统响应会有何变化。

## 实验四 简易web服务器实验

### 实验目的

* 了解web服务器的实现原理
* 了解简易web服务器的设计和使用流程

### 实验内容

* 阅读web\_server.c程序代码，了解web服务器的实现原理
* 在RealEvo-IDE下编译web\_server工程，将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

web服务器又称网页服务器，web服务器用来传送页面使浏览器可以浏览服务器上的网页。在嵌入式系统中实现web服务器可以对设备进行远程监控，而且不依赖任何辅助软件。web服务器和浏览器之间使用HTTP协议传输数据，HTTP默认使用TCP协议的80端口。

在浏览器中输入一个网址，点击回车之后，浏览器会向相应主机发送一段报文，报文使用HTTP协议。服务器启动后会一直监听TCP的80端口，当监听到客户端发来的请求后，与客户端建立连接，接收客户端发过来的请求报文。服务器成功接收到浏览器发送的报文后，会进行解析，根据报文内容发送信息，信息同样是以HTTP协议发送。比如一般情况下发送的是\*.html文件，浏览器接收到\*.html文件后，会根据\*.html文件的内容在浏览器中显示相应数据。若解析\*.html文件内容时遇到<img src=..>标签，浏览器会再次向web服务器发送请求报文，要求获取一个图像文件，web服务器会调用相应函数查找该图片并将图片文件打包发送给浏览器。如此反复，浏览器会呈现一个完整的web网页，如图 8.6所示。



图 8.6 web技术实现示意图

一个简单web服务器的实现包含如下三种技术：

* TCP/IP协议的TCP通信编程；
* HTTP协议（超文本传输协议）；
* HTML（超文本标记语言）。

**HTTP协议的主要特点**

HTTP采用请求/响应的握手方式，其通信流程为：建立连接→发出请求信息→接收响应信息→关闭连接。该协议的主要特点如下：

* 支持客户端/服务器模式；
* 简单快速：客户向服务器请求服务时，只需传送请求方法和路径。常用请求方法有GET、HEAD、POST，每种方法规定了客户端与服务器联系的类型；
* 灵活：HTTP允许传输任意类型的数据对象，传输的类型由Content-Type标记；
* 无连接：每次连接只处理一个请求，服务器处理完客户端请求并收到客户端应答后即断开连接；
* 无状态：HTTP协议对事务处理没有记忆，即服务器不会记录曾经访问过的用户，也不会记录某一客户访问次数。

**HTML简介**

HTML是一个纯文本文件，由“显示内容”和“控制语句”组成。控制语句描述了现实内容以何种形式在浏览器中显示。HTML语言有如下特点：

* HTML是纯文本文件，可以使用各种文字编辑器编辑；
* HTML文件扩展名为.htm或.html；
* 与浏览器程序参数文件关联，由浏览器编译执行。执行顺序至上而下，依次进行；
* 因浏览器支持或表现的方式不一，相同的文件在不同的浏览器中表现可能有差异；
* HTML文件由标签（tag）所构成的元素（element）组成。

在浏览器中打开任意一个网页，点击网页→右键选择“查看源代码”既可以查看实现该网页的HTML源码。

本实验对httpd进行了精简，并添加部分注释，通过阅读程序清单 8.4代码，可以了解web服务器的实现原理。此web服务器功能较简单，只能实现静态网页显示。

程序清单 8.4 web\_server.c

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <errno.h>

#include <sys/stat.h>

#include <dirent.h>

#include <signal.h>

#include <unistd.h>

#include <ctype.h>

#include "pthread.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 宏定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef O\_BINARY

#define O\_BINARY 0

#endif

#define DEBUG

#define SERVER\_PORT 80

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 定义全局变量

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int KEY\_QUIT = 0;

int TIMEOUT = 30;

char referrer[128];

char copybuf[16384];

int content\_length;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: copy

\*\* 功能描述: 发送文件

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int copy (FILE \*read\_f, FILE \*write\_f)

{

int ret;

int wrote;

ret = fread(copybuf, 1, sizeof(copybuf), read\_f);

while (ret) {

wrote = fwrite(copybuf, ret, 1, write\_f);

if (wrote < 1) {

return (-1);

}

ret = fread(copybuf, 1, sizeof(copybuf), read\_f);

}

return (0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: print\_header

\*\* 功能描述: 发送 http 协议数据头

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int print\_header (FILE \*f, int content\_type)

{

fprintf(f, "HTTP/1.0 200 OK\n"); /\* 响应 http 报文的状态行 \*/

switch (content\_type) {

case 't':

fprintf(f, "Content-type: text/plain\n"); /\* 发送文本文件信息 \*/

break;

case 'g':

fprintf(f, "Content-type: image/gif\n"); /\* 发送 gif 格式图片信息 \*/

break;

case 'j':

fprintf(f, "Content-type: image/jpeg\n"); /\* 发送 jpeg 格式图片信息 \*/

break;

case 'h':

fprintf(f, "Content-type: text/html\n"); /\* 发送 html 文件 \*/

break;

}

fprintf(f, "Server: uClinux-httpd 0.2.2\n"); /\* 发送服务器版本信息 \*/

fprintf(f, "Expires: 0\n"); /\* 发送文件永不过期 \*/

fprintf(f, "\n");

return (0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: send\_jpeg

\*\* 功能描述: 发送 jpeg 图像文件

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int send\_jpeg (FILE \*f, char \*name)

{

FILE \*infile;

if (!(infile = fopen(name, "r"))) {

fprintf(stderr, "Unable to open JPEG file %s, %d\n", name, errno);

fflush(f);

return (-1);

}

print\_header(f, 'j');

copy(infile, f);

fclose(infile);

return (0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: send\_gif

\*\* 功能描述: 发送gif图像文件

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int send\_gif (FILE \*f, char \*name)

{

FILE \*infile;

if (!(infile = fopen(name, "r"))) {

fprintf(stderr, "Unable to open GIF file %s, %d\n", name, errno);

fflush(f);

}

print\_header(f, 'g');

copy(infile, f);

fclose(infile);

return (0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: send\_dir

\*\* 功能描述: 发送当前目录信息

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int send\_dir (FILE \*f, char \*name)

{

DIR \*dir;

struct dirent \*dirent;

if ((dir = opendir(name)) == 0) {

fprintf(stderr, "Unable to open directory %s, %d\n", name, errno);

fflush(f);

return (-1);

}

print\_header(f, 'h');

fprintf(f, "<H1>Index of %s</H1>\n\n", name);

if (name[strlen(name)-1] != '/') {

strcat(name, "/");

}

dirent = readdir(dir);

while (dirent) {

fprintf(f, "<p><a href=\"/%s%s\">%s</a></p>\n",

name, dirent->d\_name, dirent->d\_name);

dirent = readdir(dir);

}

closedir(dir);

return (0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: send\_html

\*\* 功能描述: 发送html文件内容

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int send\_html (FILE \*f, char \*name)

{

FILE \*infile;

if (!(infile = fopen(name, "r"))) {

fprintf(stderr, "Unable to open HTML file %s, %d\n", name, errno);

fflush(f);

return (-1);

}

print\_header(f, 'h');

copy(infile, f);

fclose(infile);

return (0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: send\_text

\*\* 功能描述: 发送纯文本文件内容

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int send\_text(FILE \*f, char \*name)

{

FILE \*infile;

if (!(infile = fopen(name, "r"))) {

fprintf(stderr, "Unable to open text file %s, %d\n", name, errno);

fflush(f);

return (-1);

}

print\_header(f, 't');

copy(infile, f);

fclose(infile);

return (0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: parse\_req

\*\* 功能描述: 解析客户请求

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int parse\_req(FILE \*f, char \*r)

{

char \*bp;

struct stat stbuf;

char \*arg;

char \*c;

#ifdef DEBUG

printf("req is '%s'\n", r); /\* 打印请求报文 \*/

#endif

while (\*(++r) != ' ');

while (isspace(\*r)) {

r++;

}

while (\*r == '/') {

r++;

}

bp = r;

while (\*r && (\*(r) != ' ') && (\*(r) != '?')) {

r++;

}

#ifdef DEBUG

printf("bp='%s' %x, r='%s' \n", bp, \*bp,r);

#endif

if (\*r == '?')

{

char \*e;

\*r = 0;

arg = r + 1;

e = strchr(arg, ' ');

if (e) {

\*e = '\0';

}

} else {

arg = 0;

\*r = 0;

}

c = bp;

if (c[0] == 0x20){

c[0]='.';

c[1]='\0';

}

if(c[0] == '\0') {

strcat(c,".");

}

if (c && !stat(c, &stbuf)) {

if (S\_ISDIR(stbuf.st\_mode))

{

char \*end = c + strlen(c);

strcat(c, "/SylixOS-wiki.html"); /\* 发送的 htm /html 文件 \*/

if (!stat(c, &stbuf)) {

send\_html(f, c);

} else {

\*end = '\0';

send\_dir(f, c);

}

} else if (!strcmp(r - 4, ".gif")) { /\* 根据不同的报文发送文件 \*/

send\_gif(f, c);

} else if (!strcmp(r - 4, ".jpg") || !strcmp(r - 5, ".jpeg")) {

send\_jpeg(f, c);

} else if (!strcmp(r - 4, ".htm") || !strcmp(r - 5, ".html")) {

send\_html(f, c);

} else {

send\_text(f, c);

}

} else {

print\_header(f, 'h');

fprintf(f, "<html><head><title>404 File Not Found</title></head>\n");

fprintf(f, "<body>The requested URL was not found

on this server</body></html>\n");

}

return (0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: handle\_connect

\*\* 功能描述: 客户端连接处理

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int handle\_connect(int fd)

{

FILE \*f;

char buf[160];

char buf1[160];

f = fdopen(fd, "a+");

if (!f) {

fprintf(stderr, "httpd: Unable to open httpd input fd, error %d\n", errno);

close(fd);

return (0);

}

setbuf(f, 0);

if (!fgets(buf, 150, f)) {

fprintf(stderr, "httpd: Error reading connection, error %d\n", errno);

fclose(f);

return (0);

}

#ifdef DEBUG

printf("buf = '%s'\n", buf);

#endif

referrer[0] = '\0';

content\_length = -1;

while (fgets(buf1, 150, f) && (strlen(buf1) > 2)) {

#ifdef DEBUG

printf("Got buf1 '%s'\n", buf1);

#endif

if (!strncasecmp(buf1, "Referer:", 8)) {

char \*c = buf1 + 8;

while (isspace(\*c)) {

c++;

}

strcpy(referrer, c);

} else if (!strncasecmp(buf1, "Referrer:", 9)) {

char \* c = buf1 + 9;

while (isspace(\*c)) {

c++;

}

strcpy(referrer, c);

} else if (!strncasecmp(buf1, "Content-length:", 15)) {

content\_length = atoi(buf1 + 15);

}

}

if (ferror(f)) {

fprintf(stderr, "http: Error continuing reading connection,

error %d\n", errno);

fclose(f);

return (0);

}

parse\_req(f, buf);

fflush(f);

fclose(f);

return (1);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(int argc, char \*argv[])

{

int fd, s;

unsigned int len;

volatile int true = 1;

struct sockaddr\_in ec;

struct sockaddr\_in server\_sockaddr;

signal(SIGCHLD, SIG\_IGN);

signal(SIGPIPE, SIG\_IGN);

chroot("./");

printf("starting httpd...\n");

if((s = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) == -1) {

perror("Unable to obtain network");

exit(1);

}

if((setsockopt(s, SOL\_SOCKET, SO\_REUSEADDR, (void \*)&true,

sizeof(true))) == -1) {

perror("setsockopt failed");

exit(1);

}

server\_sockaddr.sin\_family = AF\_INET;

server\_sockaddr.sin\_port = htons(SERVER\_PORT);

server\_sockaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

if(bind(s, (struct sockaddr \*)&server\_sockaddr,

sizeof(server\_sockaddr)) == -1) {

perror("Unable to bind socket");

exit(1);

}

/\*

\* Arbitrary, 8 files/page, 3 clients

\*/

if(listen(s, 8\*3) == -1) {

perror("Unable to listen");

exit(4);

}

/\*

\* Wait until producer and consumer finish.

\*/

printf("wait for connection.\n");

while (1) {

len = sizeof(ec);

fd = accept(s, (void \*)&ec, &len);

if(fd == -1) {

exit(5);

close(s);

}

handle\_connect(fd);

}

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译工程web\_server，将生成的可执行文件上传到验证平台；
2. 使用浏览器打开一简单网页，随后保存该网页到本地电脑，保存后会有一\*.htm文件和一文件夹，使用FTP工具将两个文件拷贝到web\_server相同目录下；本实验使用文件如下：

[root@sylixos\_station:/apps/web\_server]# ls

web\_server SylixOS-wiki.html SylixOS-wiki\_files

[root@sylixos\_station:/apps/web\_server]#

其中SylixOS-wiki.html、SylixOS-wiki\_files是保存的网页。

注：若保存的网页名称不同，需要修改程序中的“strcat(c, "/SylixOS-wiki.html");”代码。

1. 使用ftp工具将html文件传输到验证平台：

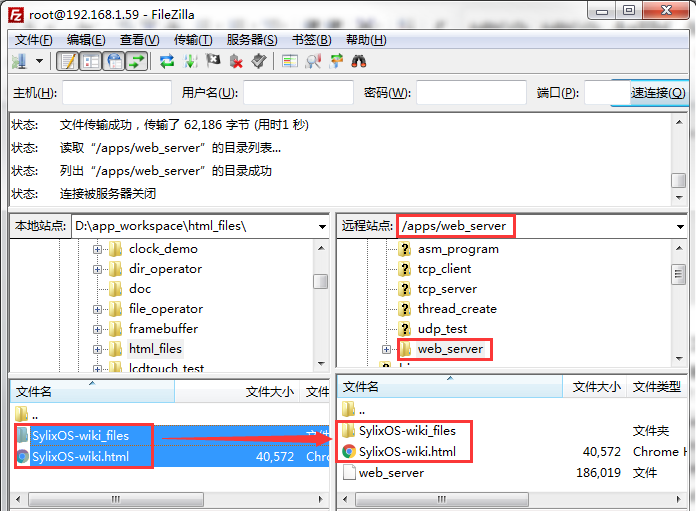


图8.7 用ftp传输html文件

1. 在验证平台上执行程序web\_server，随后打开浏览器，输入<http://192.168.1.59（IP>地址与运行web\_server的验证平台IP地址相同）并回车，等待片刻浏览器会显示网络页面。如图 8.8：



图 8.8 web\_server实现网页局部截图

验证平台控制台中也会输出获取的报文信息如下所示：

[root@sylixos\_station:/apps/web\_server]# ./web\_server

starting httpd...

wait for connection.

buf = 'GET /favicon.ico HTTP/1.1

'

Got buf1 'Host: 192.168.1.59

'

Got buf1 'Connection: keep-alive

'

Got buf1 'User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/31.0.1650.63 Safari/537.36

'

Got buf1 'Accept-Encoding: gzip,deflate,sdch

'

Got buf1 'Accept-Language: zh-CN,zh;q=0.8

'

req is 'GET /favicon.ico HTTP/1.1

'

bp='favicon.ico HTTP/1.1

' 66, r=' HTTP/1.1

### 扩展实验

在实际工程应用中往往会使用独立的web服务器，GoAhead目前已经被移植到了SylixOS，使用GoAhead实现动态网页和密码登录。

# GPIO驱动实验

GPIO是嵌入式领域使用最频繁也是使用方式最多样的硬件设备。本章通过对GPIO驱动介绍和GPIO的实际操作，介绍在SylixOS操作系统下对硬件设备操作特点和思想。实验中讲述操作GPIO的方法和思想在Linux等其他一些嵌入式操作系统上也同样适用。

## 实验一 GPIO驱动编写实验

### 实验目的

* 了解GPIO驱动的实现原理和特点
* 掌握编写GPIO驱动的方法

### 实验内容

* 阅读芯片数据手册，了解芯片IMX6Q的GPIO功能和工作流程
* 通过阅读实验原理，学习SylixOS下GPIO驱动的特点和编写方法
* GPIO驱动的编译使用

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

**IMX6Q的GPIO说明**

芯片IMX6Q用户手册（IMX6DQRM.pdf）Chapter 4: External Signals and Pin Multiplexing和Chapter 28：General Purpose Input/Output中详细介绍了芯片GPIO的功能、使用流程和相关寄存器。手册中介绍的内容依次如下：

* 简要介绍了GPIO与其他接口管脚的复用；
* GPIO管脚的芯片电气特性，如电压、输出/输出电流等；
* 简要介绍IMX6Q中GPIO控制器框图；
* 详细介绍GPIO的管脚复用、可调电气参数、对应寄存器等如图 9.1所示；

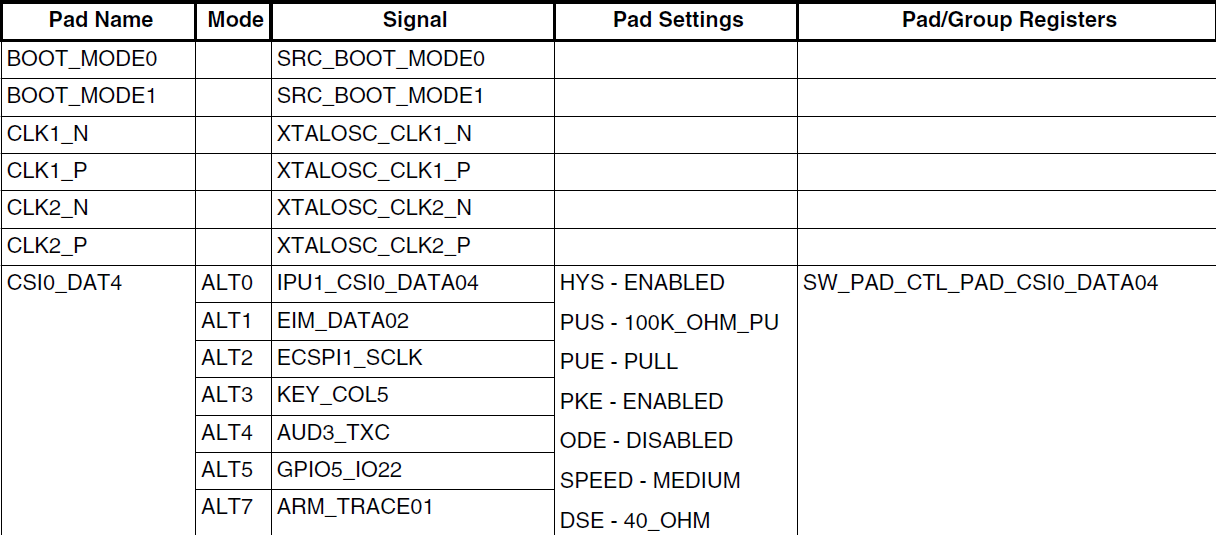


图 9.1 GPIO管脚复用和默认状态

注：图 9.1是整个管脚复用说明表格的一部分。

* GPIO控制寄存器详细说明。

每组GPIO包含8个控制器寄存器如图 9.2，每组GPIO有32路GPIO管脚。不同厂家的GPIO功能、管理方式、分组情况均有差异。在SylixOS系统中，会对不同厂家的GPIO使用无差别的管理方式，所有的GPIO使用对应编号操作，因此在应用编程上均是相同的。

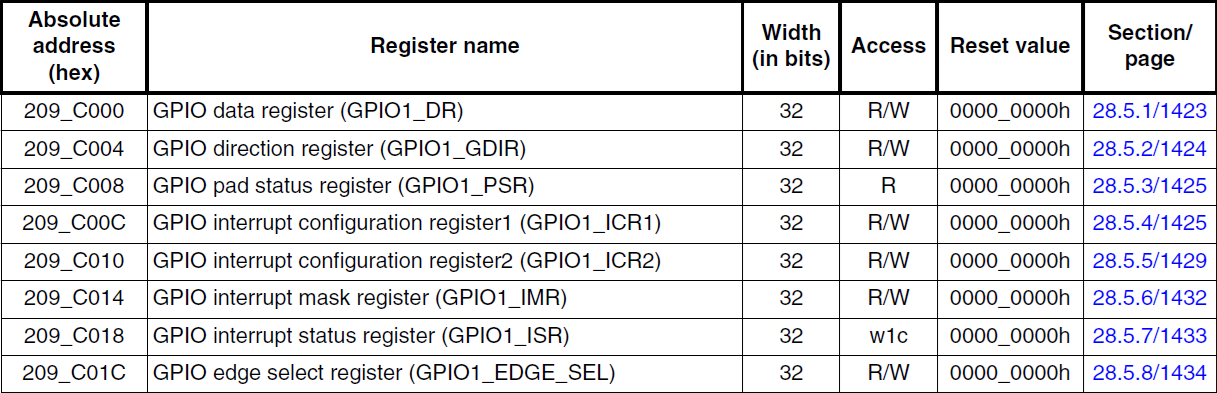


图 9.2 GPIO控制寄存器

**GPIO驱动函数**

SyixOS将GPIO控制器功能抽象成了lw\_gpio\_chip结构体，该结构体包含了GPIO所有可能的操作：

typedef struct lw\_gpio\_chip {

CPCHAR GC\_pcLabel;

LW\_LIST\_LINE GC\_lineManage;

INT (\*GC\_pfuncRequest)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset);

VOID (\*GC\_pfuncFree)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset);

INT (\*GC\_pfuncGetDirection)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset);

INT (\*GC\_pfuncDirectionInput)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset);

INT (\*GC\_pfuncGet)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset);

INT (\*GC\_pfuncDirectionOutput)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset,

INT iValue);

INT (\*GC\_pfuncSetDebounce)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset,

UINT uiDebounce);

INT (\*GC\_pfuncSetPull)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset,

UINT uiType);

VOID (\*GC\_pfuncSet)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset, INT iValue);

INT (\*GC\_pfuncSetupIrq)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset,

BOOL bIsLevel, UINT uiType);

VOID (\*GC\_pfuncClearIrq)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset);

irqreturn\_t (\*GC\_pfuncSvrIrq)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset);

UINT GC\_uiBase;

UINT GC\_uiNGpios;

struct lw\_gpio\_desc \*GC\_gdDesc;

ULONG GC\_ulPad[16]; /\* 保留未来扩展 \*/

} LW\_GPIO\_CHIP;

typedef LW\_GPIO\_CHIP \*PLW\_GPIO\_CHIP;

结构体lw\_gpio\_chip成员函数和变量的功能说明如下：

INT (\*GC\_pfuncRequest)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset);

* 功能：从SylixOS系统中申请一个GPIO；
* pgchip： GPIO控制器抽象结构体指针；
* uiOffset：GPIO编号。

VOID (\*GC\_pfuncFree)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset);

* 功能：释放正在使用的GPIO，如果当前是中断模式，则取消中断；
* pgchip： GPIO控制器抽象结构体指针；
* uiOffset：GPIO编号。

INT (\*GC\_pfuncGetDirection)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset);

* 功能：获得指定GPIO方向，1表示输出，0表示输入；
* pgchip： GPIO控制器抽象结构体指针；
* uiOffset：GPIO编号。

INT (\*GC\_pfuncDirectionInput)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset);

* 功能：设置指定GPIO为输入模式；
* pgchip：GPIO控制器抽象结构体指针；
* uiOffset：GPIO编号。

INT (\*GC\_pfuncGet)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset);

* 功能：获得指定GPIO的状态，1表示高电平，0表示低电平；
* pgchip：GPIO控制器抽象结构体指针；
* uiOffset：GPIO编号。

INT (\*GC\_pfuncDirectionOutput)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset,

INT iValue);

* 功能：设置指定GPIO为输出模式；
* pgchip：GPIO控制器抽象结构体指针；
* uiOffset：GPIO编号；
* iValue：输出值。

INT (\*GC\_pfuncSetDebounce)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset,

UINT uiDebounce);

* 功能：设置指定GPIO去抖参数；
* pgchip：GPIO控制器抽象结构体指针；
* uiOffset：GPIO编号；
* uiDebounce：去抖参数。

INT (\*GC\_pfuncSetPull)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset,

UINT uiType);

* 功能：设置GPIO上下拉；
* pgchip：GPIO控制器抽象结构体指针；
* uiOffset：GPIO编号；
* uiType：上下拉类型（0表示开路，1表示上拉，2表示下拉）。

VOID (\*GC\_pfuncSet)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset, INT iValue);

* 功能：设置指定GPIO的状态；
* pgchip：GPIO控制器抽象结构体指针；
* uiOffset：GPIO编号；
* iValue：值（1表示高电平，0表示低电平）。

INT (\*GC\_pfuncSetupIrq)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset,

BOOL bIsLevel, UINT uiType);

* 功能：设置指定GPIO的外部中断输入模式；
* pgchip： GPIO控制器抽象结构体指针；
* uiOffset：GPIO编号；
* bIsLevel：中断触发方式（1表示电平触发，0表示边沿触发）；
* uiType：触发类型（如果是电平触发，1表示高电平触发，0表示低电平触发；如果是边沿触发，1表示上升沿触发，0表示下降沿触发，2表示双边沿触发）。

VOID (\*GC\_pfuncClearIrq)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset);

* 功能：当GPIO为外部中断输入模式时，清除中断；
* pgchip ： GPIO控制器抽象结构体指针；
* uiOffset ：GPIO编号。

irqreturn\_t (\*GC\_pfuncSvrIrq)(struct lw\_gpio\_chip \*pgchip, UINT uiOffset);

* 功能：判断GPIO是否发生中断，返回LW\_IRQ\_HANDLED表示发生了中断，LW\_IRQ\_NONE表示没有发生中断；
* pgchip： GPIO控制器抽象结构体指针；
* uiOffset：GPIO编号。

GPIO驱动实现代码见文件imx6q\_gpio.c。lw\_gpio\_chip诸多成员函数不是每个都必须实现，一般GPIO驱动只是实现了LW\_GPIO\_CHIP中部分函数，如果某一功能没有实现，设置为LW\_NULL，如IMX6Q实现功能：

static LW\_GPIO\_CHIP \_G\_imx6qGpioChip = {

.GC\_pcLabel = "IMX6Q GPIO",

.GC\_pfuncRequest = imx6qGpioRequest,

.GC\_pfuncFree = imx6qGpioFree,

.GC\_pfuncGetDirection = imx6qGpioGetDirection,

.GC\_pfuncDirectionInput = imx6qGpioDirectionInput,

.GC\_pfuncGet = imx6qGpioGet,

.GC\_pfuncDirectionOutput = imx6qGpioDirectionOutput,

.GC\_pfuncSetDebounce = imx6qGpioSetDebounce,

.GC\_pfuncSetPull = LW\_NULL,

.GC\_pfuncSet = imx6qGpioSet,

.GC\_pfuncSetupIrq = imx6qGpioSetupIrq,

.GC\_pfuncClearIrq = imx6qGpioClearIrq,

.GC\_pfuncSvrIrq = imx6qGpioSvrIrq,

};

**GPIO驱动使用**

驱动完成之后需要安装到SylixOS系统中，这个过程包含两个工作： 安装GPIO库和安装GPIO驱动。安装方式十分简单，在文件bspInit.c的函数halDrvInit中调用gpioInit函数和imx6qGpioDrv函数即可。

imx6qGpioDrv();

### 实验步骤

1. 编译串口工程所在BSP工程bspimx6，生成bspimx6.bin文件；
2. 参考SylixOS系统部署附录C SylixOS系统部署章节，将文件bspimx6.bin上传到验证平台上，在系统调试时并不需要将文件部署到文件系统上，完成文件上传后，输入命令“run disable\_cache ; go 0x10000000”启动SylixOS系统。
3. 将当前控制台工作路径切换到/dev下，使用“ls”命令查看当前的GPIO设备编号。

### 扩展实验

在SylixOS网站上下载其他的bsp工程，阅读其GPIO驱动的编写，对比学习。

## 实验二 GPIO输出实验

### 实验目的

* 理解SylixOS下GPIO编号的设计和计算方法
* 掌握GPIO操作函数和使用方法

### 实验内容

* 阅读实验原理，掌握对GPIO编号的计算方法
* 阅读gpio\_out.c程序代码，了解GPIO的操作流程和相关函数使用
* 在RealEvo-IDE下编译gpio\_out工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

对GPIO输出的操作，主要包含两个方面

* 需要操作哪个GPIO，如何标识这个GPIO；
* 需要GPIO输出哪种电平状态，如何设置GPIO的输出电平状态。

GPIO输出最常用的场合是控制LED灯的亮灭来指示设备状态，通过LED的亮灭可以直观的观察GPIO高低电平的变化。查看验证平台的原理图（在名为《my-i.mx6\_sylixos\_rev.d.pdf》的PDF文档里面有LED的原理图）如图 9.3所示。

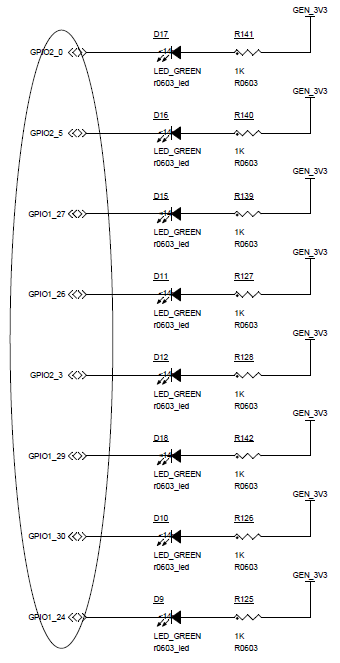


图 9.3 LED部分原理图

通过原理图可以看到，连接LED的GPIO为GPIO2\_0、GPIO2\_5、GPIO1\_27、GPIO1\_26、GPIO2\_3、GPIO1\_29、GPIO1\_30、GPIO1\_24，GPIO输出低电平点亮LED。

如果程序中直接使用GPIO2\_0类似的标识进行操作，会带来一些问题，不同芯片的GPIO定义一般不相同，使用芯片的标识编程，程序中会参杂大量与硬件相关的代码，不利于软件的移植。为此SylixOS对所有的GPIO进行了统一编号，具体实施的编号有BSP里面的GPIO驱动定义。

IMX6Q的GPIO总共分7个组，每组32个管脚。 每个GPIO编号在文件bspimx6/SylixOS/driver/gpio/imx6q\_gpio.h中提供了计算的宏定义，其他电路板中GPIO的编号也会在类似头文件中定义。

#define IMX6Q\_GPIO\_NUMR(bank, gpio) (32 \* (bank - 1) + (gpio))

根据这些宏定义，可以计算出对应GPIO在SylixOS中的编号，当然也可以直接使用宏定义。GPIO的编号及操作总结如表 9.1所示。

表 9.1 GPIO输出说明

|  |  |  |  |
| --- | --- | --- | --- |
| 丝印 | GPIO名称 | GPIO编号 | 说明 |
| LED1 | GPIO1\_24 | 24 | 1-LED灭  0-LED亮 |
| LED2 | GPIO1\_30 | 30 |
| LED3 | GPIO1\_26 | 26 |
| LED4 | GPIO1\_27 | 27 |
| LED5 | GPIO2\_5 | 37 |
| LED6 | GPIO2\_0 | 32 |
| LED7 | GPIO1\_29 | 29 |

常用GPIO的操作函数并不是标准的文件IO函数open、read、write等，系统针对GPIO的操作特点将open、read、write等函数进行了进一步封装，在使用上更加方便。

1. 打开GPIO设备函数

fd = gpiofd(led\_num, O\_WRONLY, GPIO\_FLAG\_OUT\_INIT\_LOW);

* fd 打开GPIO设备后的文件描述符，若返回值小于0，说明打开设备失败
* led\_num GPIO编号
* O\_WRONLY GPIO文件打开的方式，这里是只写
* GPIO\_FLAG\_OUT\_INIT\_LOW GPIO打开后的默认状态，默认低电平

1. 设置GPIO输出高低电平函数

gpiofd\_write(fd, 0);

* fd gpiofd函数返回的GPIO文件描述符
* 0 GPIO输出为低电平，若设置1标识输出为高电平

1. 释放GPIO设备函数

close(fd);

* fd gpiofd函数返回的GPIO文件描述符

在GPIO设备操作完成后，务必要释放该设备，否则再次打开此GPIO会失败。

本实验例程通过控制GPIO的高低电平变化来实现LED闪烁，根据上述说明设计程序如程序清单 9.1。

**代码分析：**

程序清单 9.1 gpio\_out.c

#include <sys/gpiofd.h>

#include <unistd.h>

#include <fcntl.h>

#include <stdio.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

int led\_num = 24;

int i;

int fd;

/\*

\* 打开 GPIO 文件

\*/

fd = gpiofd(led\_num, O\_WRONLY, GPIO\_FLAG\_OUT\_INIT\_LOW);

if (fd < 0) {

printf("failed to open gpio %d\n", led\_num);

return (-1);

}

for (i = 0; i < 60; i++) {

/\*

\* 对 LED 进行闪烁，每次高低电平操作前，使用usleep进行延时500ms。

\*/

gpiofd\_write(fd, 0);

usleep(500 \* 1000);

gpiofd\_write(fd, 1);

usleep(500 \* 1000);

}

close(fd); /\* 关闭设备 \*/

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译工程gpio\_out，将生成的程序文件上传到验证平台；
2. 在控制台中，将工作目录切换到/apps/gpio\_out执行命令./gpio\_out，会看到验证平台LED1在闪烁。

### 扩展实验

编程实现验证平台的LED实现流水灯的效果。

## 实验三 GPIO输入实验

### 实验目的

* 理解GPIO输入的应用程序设计方法
* 掌握select函数的使用方法

### 实验内容

* 分析原理图，得到相关GPIO的操作编号
* 掌握select函数使用方法
* 在RealEvo-IDE下编译gpio\_in工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

GPIO连接外部按键，当按键按下时，会改变对应GPIO的电平状态，对GPIO电平状态的读取即对按键按下的响应。验证平台对GPIO连接按键的设计使用了IMX6Q芯片的行列扫描功能，如果本例程使用行列扫描功能，GPIO输入例程将不能够突出GPIO输入的特点。因此对其硬件的使用进行了如下修改。

* 验证平台上的按键，仅仅使用电路板外侧的一部分，如图 9.4；
* 将图 9.4中名KP\_ROW5的GPIO悬空，不做配置。

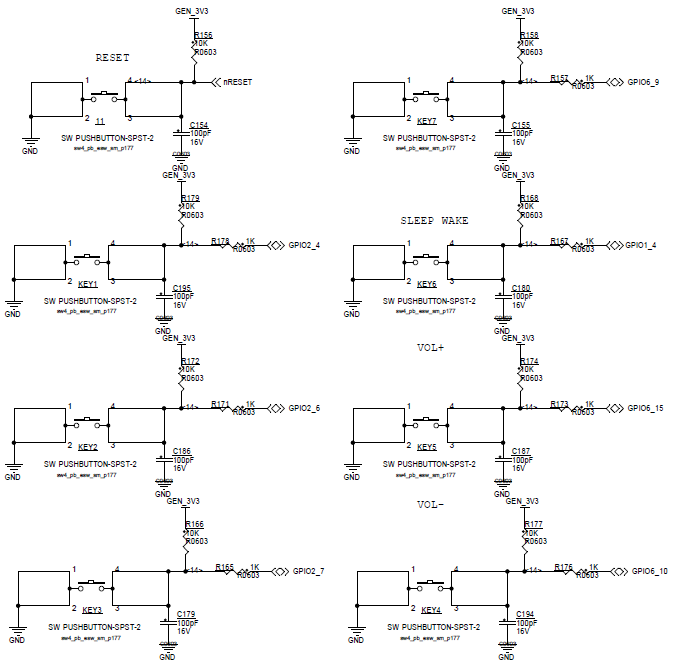


图 9.4 GPIO输入部分原理图

根据本章中实验二 GPIO输出实验的方法，计算出GPIO对应的编号并分析状态变化如表 9.2。

表 9.2 GPIO输入说明

|  |  |  |  |
| --- | --- | --- | --- |
| 丝印 | GPIO名称 | GPIO编号 | 说明 |
| KEY1 | GPIO2\_4 | 36 | 按键按下—0  按键松开—1 |
| KEY2 | GPIO2\_6 | 38 |
| KEY3 | GPIO2\_7 | 39 |
| KEY4 | GPIO6\_10 | 170 |
| KEY5 | GPIO6\_15 | 175 |
| KEY6 | GPIO1\_4 | 4 |
| KEY7 | GPIO6\_9 | 169 |

对GPIO输入电平状态获取，最简单的方式是在一个循环里不断的读取当前GPIO状态，GPIO读取操作与输出操作基本一致，只是用GPIO读函数替代了GPIO的写函数，读取GPIO电平状态函数为：

gpiofd\_read(keyfd, &ucvalue);

* keyfd 是gpiofd打开GPIO设备后返回文件描述符
* ucvalue 是gpiofd\_read函数返回的GPIO状态

由于gpiofd\_read函数立即返回电平状态不具备阻塞能力。所以应用程序读取带有中断能力的GPIO常用做法是使用select函数进行阻塞操作，不仅仅是GPIO，在其他设备输入操作均可以使用select函数。

**select函数介绍**

select函数是用来监视多个文件IO的状态的IO多路复用函数，诸如GPIO输入、网络数据输入/输出、串口输入/输出等操作，将相关设备打开返回的文件描述符装载入select函数，就可以实现对这些设备接口信息输入/输出的监视。select函数及使用方法如下：

1. 声明一个文件集结构体变量，用来添加文件描述符。

fd\_set fdset;

1. 使用专用宏定义，将文件集变量fdset清空

FD\_ZERO(&fdset);

1. 将文件描述符添加到文件集fdset

FD\_SET(keyfd, &fdset);

* keyfd 打开GPIO后返回的文件描述符。如果需要一次监控多个设备，这里就需要执行多次添加操作

使用FD\_CLR宏可以从文件集里删除某一个文件描述符。参数与FD\_SET宏一致。

1. 使用select函数监控设备

ret = select(keyfd + 1, /\* 文件集内数值最大的文件描述符+1 \*/

&fdset, /\* 转载文件描述符的文件集 \*/

NULL, /\* 填写NULL即可 \*/

NULL, /\* 填写NULL即可 \*/

NULL); /\* 等待时间设置，NULL表示无限等待 \*/

1. 在select函数之后添加处理代码

if (FD\_ISSET(keyfd, &fdset)) {

/\*

\* 文件描述符为keyfd的设备响应处理

\*/

}

本实验例程为方便观察实验效果，每按下特定按键后，与之相邻的LED状态就会取反，反复按下按键，LED就会一亮一灭闪烁。

**代码分析：**

程序清单 9.2 gpio\_in.c

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/select.h>

#include <sys/gpiofd.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

int ret;

fd\_set fdset;

uint8\_t value;

int led\_num = 24; /\* gpio LED 的编号 \*/

int key\_num = 36; /\* gpio KEY 的编号 \*/

int ledfd, keyfd;

/\*

\* 打开 LED 的 GPIO 文件

\*/

ledfd = gpiofd(led\_num, O\_WRONLY, GPIO\_FLAG\_OUT\_INIT\_LOW);

if (ledfd < 0) {

printf("gpiofd create error!\n");

return (-1);

}

/\*

\* 打开 KEY 的 GPIO 文件，配置为下降沿触发中断

\*/

keyfd = gpiofd(key\_num, O\_RDONLY, GPIO\_FLAG\_TRIG\_FALL);

if (keyfd < 0) {

printf("gpiofd create error!");

return (-1);

}

/\*

\* 调用 select 等待按键被按下

\*/

FD\_ZERO(&fdset);

while (1) {

FD\_SET(keyfd, &fdset);

ret = select(keyfd + 1, &fdset, NULL, NULL, NULL);

if (ret == 1) {

gpiofd\_read(keyfd, &value);

printf("value : %d\n", value);

/\*

\* 对 LED 进行取反操作

\*/

gpiofd\_read(ledfd, &value);

value = !value;

gpiofd\_write(ledfd, value);

/\*

\* 0.5s消抖处理

\*/

usleep(500 \* 1000);

} else if (ret < 0) {

printf("select error!\n");

break;

}

}

close(ledfd);

close(keyfd);

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译工程gpio\_in，将生成的程序文件上传到验证平台；
2. 在控制台中，将工作目录切换到/apps/gpio\_in执行命令./gpio\_in，反复按下Key1会观察到LED1亮灭变化。

### 扩展实验

编程实现所有按键状态的读取，并在每次读到有按键输入时，切换其对应LED灯的开关状态。例如，按下KEY2时，打开LED2；再按一次KEY2，关闭LED2；其他按键如此类推。

# LED驱动实验

内核模块是能够在系统内核空间运行的一段代码，在本质上与内核其他程序并没有区别，内核模块中所使用的函数也必须是内核态的函数，而不能够使用用户态的函数。

SylixOS中内核模块的概念和实现方式与Linux下相同，但内核模块的装载和卸载命令有所区别。

内核模块不仅可以有效的减少内核静态程序体积，在驱动调试方面也有极重要的意义，对驱动程序的调试可以不必频繁重启系统，也不用每次编译系统内核。修改驱动程序后，只需要卸载原有的驱动模块，然后装载修改后的驱动模块即可。因此内核模块一个很重要的应用是编写驱动程序。

本章通过4个实验，逐步深入的演示内核模块的使用，在实验四给出了一个完整的LED内核模块驱动例程。

## 实验一 内核模块基本实验

### 实验目的

* 掌握内核模块工程设计的流程
* 对内核模块的功能实现和使用有形象的认识

### 实验内容

* 完成一个内核模块工程的创建、编译、装载和卸载

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

内核模块文件开始，需要使用宏定义“\_\_SYLIXOS\_KERNEL”声明当前文件为内核模块。

任何一个内核模块工程或文件都要包含两个头文件“SylixOS.h”和“module.h”，前者是SylixOS系统统一的头文件，包含系统内核各个模块功能定义，后者则包含内核模块装载器相关参数与API。

装载一个内核模块的shell命令是modulereg，在装载模块时，会自动调用内核模块内的module\_init函数。在内核模块中需要进行的初始化工作可以放在module\_init函数内。编写驱动程序时需要的接口初始化、注册设备这些操作都需要放在此函数内。

卸载一个内核模块的shell命令是moduleunreg，在卸载模块时，会自动调用内核模块内的module\_exit函数。内核模块中一些任务结束后需要释放系统资源，释放系统资源操作可以放在module\_exit内完成。

一个内核模块工程中可能有很多函数，但不是所有的函数都能够被其他模块或程序使用，只有被宏“LW\_SYMBOL\_EXPORT”声明的函数才能够被其他模块或程序使用。

编译内核模块的SylixOS内核版本与验证平台运行的SylixOS内核版本需要保持一致，否则会装载失败。

### 实验步骤

1. 创建内核模块工程

打开RealEvo-IDE，工作空间切换到随书工程工作空间，“File” →“New” →“Project”创建内核模块工程，选择“SylixOS Kernel Module”。如图 10.1所示：

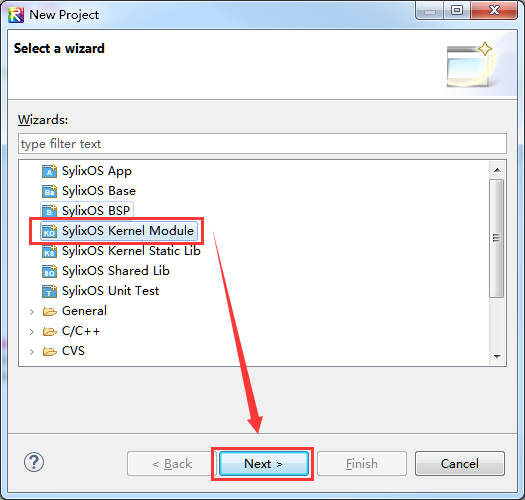


图 10.1 内核模块工程类型选择

点击“Next”按钮，添加工程命名modu\_demo，点击“Next”按钮，选择“SylixOS base project”路径。

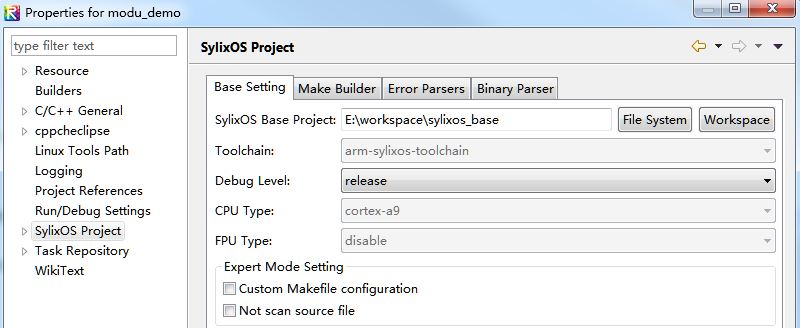


图 10.2 设置工程Base路径

之后点击“Finish”按钮，完成工程创建。创建后的工程，会自动生成modu\_demo.c文件，文件包含有默认代码。本实验例程中不对生成的代码做任何修改。

程序清单 10.1 内核模块工程生成的默认代码

#define \_\_SYLIXOS\_KERNEL

#include <SylixOS.h>

#include <module.h>

/\*

\* SylixOS call module\_init() and module\_exit() automatically.

\*/

void module\_init (void)

{

printk("hello\_module init!\n");

return 0;

}

void module\_exit (void)

{

printk("hello\_module exit!\n");

}

/\*

\* module export symbols

\*/

LW\_SYMBOL\_EXPORT void hello\_module\_func (void)

{

printk("hello\_module\_func() run!\n");

}

1. 装载内核模块

编译工程modu\_demo，将生成的程序文件modu\_demo.ko上传到验证平台中。需要注意，内核模块工程的“Upload Setting”中默认路径是/lib/modules/modu\_demo.ko。

在控制台中，将当前工作路径切换到/lib/modules下，装载内核模块

[root@sylixos\_station:/lib/modules]# modulereg modu\_demo.ko

hello\_module init！

module modu\_demo.ko registor ok, handle : 0x2468a298

* 执行装载命令后，输出信息是module\_init函数的打印信息
* 装载成功后，会返回handle，此handle在模块卸载时会使用

1. 卸载内核模块

[root@sylixos\_station:/lib/modules]# moduleunreg 0x2468a298

hello\_module exit！

### 扩展实验

建立一个内核模块工程，在新内核模块工程中调用本实验内核模块中的函数。

## 实验二 GPIO内核态输出实验

### 实验目的

* 了解内核态函数与用户态函数的区别
* 了解GPIO操作的内核态函数及其使用方法

### 实验内容

* 阅读文件gpio\_out\_modu.c，了解GPIO内核态函数和使用
* 在RealEvo-IDE下编译gpio\_out\_modu工程，将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

**什么是内核态函数**

在SylixOS中，对物理设备的操作提供了内核态函数和用户态函数，所有的物理设备在用户态下，均提供统一的文件IO操作接口，使用open、read、write、close、ioctl等函数进行操作。而内核态函数针对不同设备往往有不同的函数接口，内核态函数主要是为编写驱动程序等内核功能模块提供的。

**内核态函数介绍**

内核态下操作GPIO的输出与用户态下的流程是一致的，只是所使用的函数不同。

1. 申请（打开）一个GPIO设备

iError = API\_GpioRequestOne(iLedNum, LW\_GPIOF\_OUT\_INIT\_LOW, "LED");

* 返回值 函数执行成功返回ERROR\_NONE
* iLedNum 需要操作的GPIO编号
* LW\_GPIOF\_OUT\_INIT\_LOW 设置GPIO输出并默认输出低电平
* “LED” 打开后设备标识字符，并无标准要求

1. 设置GPIO输出数值

API\_GpioSetValue(iLedNum, 0);

* iLedNum 需要操作的GPIO编号
* 0 设置GPIO输出电平为低电平，若设置1输出为高电平

1. 释放GPIO

API\_GpioFree(iLedNum);

* iLedNum 需要操作的GPIO编号

在内核态下操作GPIO的函数均使用GPIO的编号，不会像用户态函数那样使用文件描述符。

**内核态函数使用**

在用户态函数编程下，均使用“SylixOS App”类型的工程创建，内核态函数是不能够在用户态程序下使用的。因此本实验创建的工程类型是“SylixOS Kernel Module”（内核模块）。由于程序是在内核执行，为了避免程序任务对内核其他任务的阻塞干扰，需要创建独立的线程。这里创建线程的函数同样也是内核态线程创建函数。

本实验例程与用户态下GPIO输出实验例程功能相同。

**代码分析：**

程序清单 10.2 gpio\_out\_modu.c

#define \_\_SYLIXOS\_KERNEL

#include <SylixOS.h>

#include <module.h>

#include <system/device/gpio/gpioLib.h>

#include <sys/select.h>

#include <sys/gpiofd.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: ledThread

\*\* 功能描述: LED thread

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static VOID ledThread (VOID)

{

INT iLedNum = 24;

INT i;

INT iError;

/\*

\* 打开 GPIO 设备

\*/

iError = API\_GpioRequestOne(iLedNum, LW\_GPIOF\_OUT\_INIT\_LOW, "LED");

if (iError != ERROR\_NONE) {

printk("failed to open gpio %d\n", iLedNum);

return ;

}

for (i = 0; i < 60; i++) { /\* 使 LED 闪烁 \*/

API\_GpioSetValue(iLedNum, 0);

API\_TimeMSleep(500);

API\_GpioSetValue(iLedNum, 1);

API\_TimeMSleep(500);

}

API\_GpioFree(iLedNum);

}

void module\_init (void)

{

printk("hello\_module init!\n");

API\_ThreadCreate("t\_led", (PTHREAD\_START\_ROUTINE)ledThread,

LW\_NULL, LW\_NULL);

}

void module\_exit (void)

{

printk("hello\_module exit!\n");

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译gpio\_out\_modu工程，将编译后的程序上传到验证平台。需要注意程序编译后生成的是有后缀名的文件gpio\_out\_modu.ko，文件的上传路径也不再是/apps目录，而是/lib/module目录；
2. 装载模块（相当于执行程序）；

[root@sylixos\_station:/lib/modules]# modulereg gpio\_out\_modu.ko

hello\_moudle init!

gpio\_out\_modu.ko register ok, handle 0x2468a440

提示成功后，会看到验证平台LED在闪烁。

1. 卸载模块。

在LED停止闪烁后，需要将内核模块从内核卸载掉。

[root@sylixos\_station:/lib/modules]# moduleunreg 0x2468a440

### 扩展实验

无。

## 实验三 GPIO内核态输入实验

### 实验目的

* 了解中断的概念和作用
* 掌握中断实现相关函数及使用方法

### 实验内容

* 阅读gpio\_in\_modu.c文件，了解使用内核态函数设计中断程序的方法
* 在RealEvo-IDE下编译gpio\_in\_modu工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

内核态下的GPIO输入操作与用户态下差别较大，在内核态下操作GPIO输入无法使用select函数，也没有类似select函数的内核态函数。这里使用更接近硬件本身工作流程的中断概念。中断是CPU本身提供的一种用来响应外部状态变化的机制。

在内核态下进行GPIO中断的操作流程和函数如下

1. 打开（申请）GPIO，设置GPIO状态输入

iError = API\_GpioRequestOne(iKeyGpioNum, LW\_GPIOF\_IN, "KEY");

* 函数执行成功，返回ERROR\_NONE
* iKeyGpioNum 用来触发中断的GPIO编号
* LW\_GPIOF\_IN 设置GPIO的属性，输入
* “KEY” 设备打开的后的标识

1. 设置中断方式为下降沿触发，并申请中断号

iIrqNum = API\_GpioSetupIrq(iKeyGpioNum, LW\_FALSE, 0);

* 返回GPIO的中断号
* iKeyGpioNum 触发中断的GPIO编号
* LW\_FALSE 边缘触发
* 0 下降沿触发

1. 绑定中断号与中断处理函数

iError = API\_InterVectorConnect((ULONG)iIrqNum,

(PINT\_SVR\_ROUTINE)GpioIsr,

(PVOID)iKeyGpioNum,

"GpioIsr");

* 函数执行成功返回ERROR\_NONE。
* iIrqNum 触发中断的中断号
* GpioIsr 中断响应函数
* iKeyGpioNum 触发中断的GPIO编号，作为参数传入中断处理函数GpioIsr
* “GpioIsr” 中断函数标识。

1. 使能GPIO中断

API\_InterVectorEnable(iIrqNum);

完成以上操作后，当GPIO的管脚电平由高变低产生中断时，系统会调用名为GpioIsr的函数。中断处理函数会抢占其他正在运行的线程，因此中断处理函数需要尽可能的简短。GPIO中断处理函数设计一般规则如下：

1. 判断触发中断的触发源（有可能多个GPIO共享一个中断向量号），在本实验中判断是否是该GPIO触发的中断

irqreturn = API\_GpioSvrIrq(iGpioNum);

* 函数返回值为LW\_IRQ\_NONE 说明不是本GPIO产生的中断，其他值则表示是编号为iGpioNum的GPIO触发的中断
* iGpioNum 设置中断响应的GPIO编号，此参数在中断响应函数绑定时传入

1. 清除中断，如果不清除中断，下次中断将无法正常响应等现象

API\_GpioClearIrq(iGpioNum);

1. 添加中断的其他处理代码

**代码分析**

本实验使用的工程类型同样是“SylixOS Kernel Module”。实验例程实现的功能与第9章中的实验三 GPIO输入实验相同。

程序清单 10.3 gpio\_in\_modu.c

#define \_\_SYLIXOS\_STDIO

#define \_\_SYLIXOS\_KERNEL

#include <SylixOS.h>

#include <module.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 宏定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define LED\_NUM 24

#define KEY\_NUM 36

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: GpioIsr

\*\* 功能描述: GPIO 的中断响应函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static irqreturn\_t GpioIsr (INT iGpioNum, ULONG ulVector)

{

API\_GpioClearIrq(iGpioNum); /\* 清除中断 \*/

if (API\_GpioGetValue(LED\_NUM)) { /\* 对 LED 进行取反操作 \*/

API\_GpioSetValue(LED\_NUM, 0);

} else {

API\_GpioSetValue(LED\_NUM, 1);

}

return 0;

} /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: \_\_keyThread

\*\* 功能描述: KEY thread

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static PVOID \_\_keyThread (PVOID pvArg)

{

INT iError;

INT iIrqNum;

/\*

\* 申请 LED 的 GPIO

\*/

iError = API\_GpioRequestOne(LED\_NUM, LW\_GPIOF\_OUT\_INIT\_LOW, "LED");

if (iError != ERROR\_NONE) {

printk("failed to request gpio %d!\n", LED\_NUM);

return (NULL);

}

/\*

\* 申请 KEY 的 GPIO

\*/

iError = API\_GpioRequestOne(KEY\_NUM, LW\_GPIOF\_IN, "KEY");

if (iError != ERROR\_NONE) {

printk("failed to request gpio %d!\n", KEY\_NUM);

return (NULL);

}

/\*

\* 获得 KEY 的 GPIO 的中断号，配置为下降沿触发中断

\*/

iIrqNum = API\_GpioSetupIrq(KEY\_NUM, LW\_FALSE, 0);

if (iIrqNum == PX\_ERROR) {

printk("failed to setup gpio %d irq!\n", KEY\_NUM);

return (NULL);

}

/\*

\* 绑定 KEY 的 GPIO 的中断响应函数

\*/

iError = API\_InterVectorConnect((ULONG)iIrqNum,

(PINT\_SVR\_ROUTINE)GpioIsr,

(PVOID)KEY\_NUM,

"GpioIsr");

if (iError != ERROR\_NONE) {

printk("failed to connect GpioIsr!\n");

return (NULL);

}

/\*

\* 使能 KEY 的 GPIO 的中断

\*/

API\_InterVectorEnable(iIrqNum);

API\_TimeSSleep(60);

/\*

\* 释放 GPIO

\*/

API\_InterVectorDisconnect((ULONG)iIrqNum,

(PINT\_SVR\_ROUTINE)GpioIsr,

(PVOID)KEY\_NUM);

API\_GpioFree(LED\_NUM);

API\_GpioFree(KEY\_NUM);

return (NULL);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: module\_init

\*\* 功能描述: 模块安装函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void module\_init (void)

{

printk("hello\_module init!\n");

API\_ThreadCreate("t\_Key", (PTHREAD\_START\_ROUTINE)\_\_keyThread, LW\_NULL, LW\_NULL);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: module\_exit

\*\* 功能描述: 模块退出函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void module\_exit (void)

{

printk("hello\_module exit!\n");

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译gpio\_in\_modu工程，将编译后的程序文件上传到验证平台。需要注意的是程序编译后生成的是有后缀的文件gpio\_in\_modu.ko，文件的上传路径也不再是/apps目录，而是/lib/module目录；
2. 装载内核模块（执行程序）；

[root@sylixos\_station:/lib/modules]# modulereg gpio\_in\_modu.ko

hello\_moudle init!

gpio\_in\_modu.ko register ok, handle 0x2468a440

1. 验证中断响应；

不断按下Key1，观察LED1的反应，LED1会在按压Key1时闪烁。

1. 卸载内核模块。

程序执行时间为60s，60s后需要将内核模块从内核卸载掉。

[root@sylixos\_station:/lib/modules]# moduleunreg 0x2468a440

### 扩展实验

无。

## 实验四 LED驱动编写实验

### 实验目的

* 了解以内核模块的方式编写驱动的方法及驱动的使用

### 实验内容

* 阅读led\_drv\_modu.c文件，了解内核模块编写驱动的方法
* 阅读led\_drv\_use.c文件，掌握使用文件IO函数操作LED的方法
* 在RealEvo-IDE打开并编译led\_drv\_modu、led\_drv\_use工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

**LED驱动编写**

通过前面一些GPIO相关实验，详细介绍了GPIO在用户态和内核态的操作，通过这些实验相信读者已经对如何控制LED的亮灭了深入的认识。本实验所讲述的LED与前面章节中的LED在概念上有所区别。前面实验中如果要控制LED需要查找对应的GPIO、计算GPIO的编号、随后使用GPIO操作函数通过控制GPIO的输出控制LED的亮灭。在本章LED是类似第7章的串口概念，LED作为一个独立的设备存在于SylixOS中，对LED的操作可以不需要了解GPIO等内容，只需要使用open、ioctl、close这些函数即可。

LED不具备输入输出功能，因此在驱动的实现中不需要设计read、write函数。本实验的LED驱动，主要实现四个功能函数\_ledOpen、\_ledClose、\_ledIotcl、\_ledLstat。

在ledDrv函数里面，会将led相关操作函数与标准文件操作接口进行绑定。ledDevCreate函数实现在SylixOS下创建LED设备。以上两个函数会在module\_init里调用，即在LED驱动内核模块注册时，会绑定文件操作函数并向SylixOS注册LED设备。同样在内核模块卸载时，需要删除已经注册了的LED设备。

在打开设备后，\_ledOpen函数会创建一个线程\_ledThread处理与LED相对于的GPIO操作，使用ioctl进行的设置会通过消息队列传入\_ledThread线程。详细代码如程序清单 10.4。

程序清单 10.4 led\_drv\_use.c

#define \_\_SYLIXOS\_KERNEL

#include <SylixOS.h>

#include <module.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

LED 简易操作宏

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define \_\_LED\_ON(pled) gpioSetValue(pled->LED\_uiGpio,

pled->LED\_bIsOutPutHigh ? 1 : 0)

#define \_\_LED\_OFF(pled) gpioSetValue(pled->LED\_uiGpio,

pled->LED\_bIsOutPutHigh ? 0 : 1)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* LED 任务默认优先级

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define \_\_LED\_THREAD\_PRIO LW\_PRIO\_HIGH

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

全局变量

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static INT \_G\_iLedDrvNum = 0; /\* LED 主设备号 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

设备结构

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typedef struct \_\_led\_dev {

LW\_DEV\_HDR LED\_devhdr; /\* 设备头 \*/

LW\_LIST\_LINE\_HEADER LED\_fdNodeHeader;

LW\_OBJECT\_HANDLE LED\_hThread; /\* 闪烁线程 \*/

LW\_OBJECT\_HANDLE LED\_hSemaphoreB; /\* 信号量 \*/

UINT LED\_uiGpio; /\* GPIO \*/

BOOL LED\_bIsOutPutHigh; /\* 是否输出高电平为点亮 \*/

BOOL LED\_bQuit; /\* 内线线程退出通知标志 \*/

time\_t LED\_time; /\* 设备创建时间 \*/

} \_\_LED\_DEV;

typedef \_\_LED\_DEV \*\_\_PLED\_DEV;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: \_\_ledThread

\*\* 功能描述: LED thread

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static VOID \_\_ledThread (\_\_PLED\_DEV pled)

{

ULONG ulMsTime;

for (;;) {

if (API\_SemaphoreBPendEx(pled->LED\_hSemaphoreB, LW\_OPTION\_WAIT\_INFINITE,

(PVOID \*)&ulMsTime) == ERROR\_NONE) {

if (pled->LED\_bQuit) {

break;

}

\_\_LED\_ON(pled);

API\_TimeMSleep(ulMsTime);

\_\_LED\_OFF(pled);

} else {

printk(KERN\_ERR "\_\_ledThread(): led semaphore error\n");

break;

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: \_\_ledOpen

\*\* 功能描述: LED 打开

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static LONG \_\_ledOpen (\_\_PLED\_DEV pled, CHAR \*pcName, INT iFlags, INT iMode)

{

LW\_CLASS\_THREADATTR threadattr;

PLW\_FD\_NODE pfdnode;

BOOL bIsNew;

if (pcName == LW\_NULL) {

\_ErrorHandle(ERROR\_IO\_NO\_DEVICE\_NAME\_IN\_PATH);

return (PX\_ERROR);

} else {

pfdnode = API\_IosFdNodeAdd(&pled->LED\_fdNodeHeader, (dev\_t)pled, 0,

iFlags, iMode, 0, 0, 0, LW\_NULL, &bIsNew);

if (pfdnode == LW\_NULL) {

printk(KERN\_ERR "\_\_ledOpen(): failed to add fd node!\n");

return (PX\_ERROR);

}

if (LW\_DEV\_INC\_USE\_COUNT(&pled->LED\_devhdr) == 1) {

if (gpioRequestOne(pled->LED\_uiGpio,

(pled->LED\_bIsOutPutHigh ? LW\_GPIOF\_OUT\_INIT\_LOW :LW\_GPIOF\_OUT\_INIT\_HIGH),

"led")) {

LW\_DEV\_DEC\_USE\_COUNT(&pled->LED\_devhdr);

API\_IosFdNodeDec(&pled->LED\_fdNodeHeader, pfdnode, NULL);

return (PX\_ERROR);

}

pled->LED\_bQuit = LW\_FALSE;

threadattr = API\_ThreadAttrGetDefault();

threadattr.THREADATTR\_pvArg = (VOID \*)pled;

threadattr.THREADATTR\_ucPriority = \_\_LED\_THREAD\_PRIO;

threadattr.THREADATTR\_ulOption |= LW\_OPTION\_OBJECT\_GLOBAL;

pled->LED\_hSemaphoreB = API\_SemaphoreBCreate("sem\_led", LW\_FALSE,

LW\_OPTION\_DEFAULT,

LW\_NULL);

pled->LED\_hThread = API\_ThreadCreate("t\_led", (PTHREAD\_START\_ROUTINE)\_\_ledThread, &threadattr, LW\_NULL);

}

return ((LONG)pfdnode);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: \_\_ledClose

\*\* 功能描述: LED 关闭

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static INT \_\_ledClose (PLW\_FD\_ENTRY pfdentry)

{

\_\_PLED\_DEV pled = (\_\_PLED\_DEV)pfdentry->FDENTRY\_pdevhdrHdr;

PLW\_FD\_NODE pfdnode = (PLW\_FD\_NODE)pfdentry->FDENTRY\_pfdnode;

if (pfdentry && pfdnode) {

API\_IosFdNodeDec(&pled->LED\_fdNodeHeader, pfdnode, NULL);

if (LW\_DEV\_DEC\_USE\_COUNT(&pled->LED\_devhdr) == 0) {

pled->LED\_bQuit = LW\_TRUE;

API\_SemaphoreBPostEx(pled->LED\_hSemaphoreB, (VOID \*)0);

API\_ThreadJoin(pled->LED\_hThread, LW\_NULL);

pled->LED\_hThread = 0;

API\_SemaphoreBDelete(&pled->LED\_hSemaphoreB);

pled->LED\_hSemaphoreB = 0;

gpioFree(pled->LED\_uiGpio);

}

}

return (ERROR\_NONE);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: \_\_ledIoctl

\*\* 功能描述: LED 控制

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static INT \_\_ledIoctl (PLW\_FD\_ENTRY pfdentry, INT iCmd, LONG lArg)

{

\_\_PLED\_DEV pled = (\_\_PLED\_DEV)pfdentry->FDENTRY\_pdevhdrHdr;

struct stat \*pstat;

switch (iCmd) {

case FIOFSTATGET:

pstat = (struct stat \*)lArg;

if (pstat) {

pstat->st\_dev = (dev\_t)pled;

pstat->st\_ino = (ino\_t)0;

pstat->st\_mode = (S\_IRUSR | S\_IRGRP | S\_IROTH);

pstat->st\_nlink = 1;

pstat->st\_uid = 0;

pstat->st\_gid = 0;

pstat->st\_rdev = 0;

pstat->st\_size = 0;

pstat->st\_blksize = 0;

pstat->st\_blocks = 0;

pstat->st\_atime = pled->LED\_time;

pstat->st\_mtime = pled->LED\_time;

pstat->st\_ctime = pled->LED\_time;

return (ERROR\_NONE);

}

return (PX\_ERROR);

case FIOSETFL:

if ((int)lArg & O\_NONBLOCK) {

pfdentry->FDENTRY\_iFlag |= O\_NONBLOCK;

} else {

pfdentry->FDENTRY\_iFlag &= ~O\_NONBLOCK;

}

return (ERROR\_NONE);

case 0:

API\_SemaphoreBPostEx(pled->LED\_hSemaphoreB, (VOID \*)lArg);

return (ERROR\_NONE);

case 1:

API\_ThreadSetPriority(pled->LED\_hThread, (UINT8)lArg);

return (ERROR\_NONE);

}

return (PX\_ERROR);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: \_\_ledLstat

\*\* 功能描述: 获得 LED 设备状态

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static INT \_\_ledLstat (PLW\_DEV\_HDR pDev, PCHAR pcName, struct stat \*pstat)

{

\_\_PLED\_DEV pled = (\_\_PLED\_DEV)pDev;

if (pstat) {

pstat->st\_dev = (dev\_t)pled;

pstat->st\_ino = (ino\_t)0;

pstat->st\_mode = (S\_IFCHR | S\_IRUSR | S\_IRGRP | S\_IROTH);

pstat->st\_nlink = 1;

pstat->st\_uid = 0;

pstat->st\_gid = 0;

pstat->st\_rdev = 0;

pstat->st\_size = 0;

pstat->st\_blksize = 0;

pstat->st\_blocks = 0;

pstat->st\_atime = pled->LED\_time;

pstat->st\_mtime = pled->LED\_time;

pstat->st\_ctime = pled->LED\_time;

return (ERROR\_NONE);

}

return (PX\_ERROR);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: ledDrv

\*\* 功能描述: 创建 LED 驱动程序

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT ledDrv (VOID)

{

struct file\_operations fileop;

if (\_G\_iLedDrvNum) {

return (ERROR\_NONE);

}

lib\_memset(&fileop, 0, sizeof(struct file\_operations));

fileop.owner = THIS\_MODULE;

fileop.fo\_create = \_\_ledOpen;

fileop.fo\_open = \_\_ledOpen;

fileop.fo\_close = \_\_ledClose;

fileop.fo\_lstat = \_\_ledLstat;

fileop.fo\_ioctl = \_\_ledIoctl;

\_G\_iLedDrvNum = iosDrvInstallEx2(&fileop, LW\_DRV\_TYPE\_NEW\_1);

DRIVER\_LICENSE(\_G\_iLedDrvNum, "Dual BSD/GPL->Ver 1.0");

DRIVER\_AUTHOR(\_G\_iLedDrvNum, "Han.hui");

DRIVER\_DESCRIPTION(\_G\_iLedDrvNum, "led driver.");

return (\_G\_iLedDrvNum > 0) ? (ERROR\_NONE) : (PX\_ERROR);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: ledDevCreate

\*\* 功能描述: 创建 LED 设备

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT ledDevCreate (CPCHAR cpcName, UINT uiGpio, BOOL bIsOutPutHigh)

{

\_\_PLED\_DEV pled;

pled = (\_\_PLED\_DEV)\_\_SHEAP\_ALLOC(sizeof(\_\_LED\_DEV));

if (!pled) {

\_DebugHandle(\_\_ERRORMESSAGE\_LEVEL, "system low memory.\r\n");

\_ErrorHandle(ERROR\_SYSTEM\_LOW\_MEMORY);

return (PX\_ERROR);

}

lib\_memset(pled, 0, sizeof(\_\_LED\_DEV));

pled->LED\_uiGpio = uiGpio;

pled->LED\_bIsOutPutHigh = bIsOutPutHigh;

pled->LED\_time = time(LW\_NULL);

if (API\_IosDevAddEx(&pled->LED\_devhdr, cpcName, \_G\_iLedDrvNum, DT\_CHR)

!= ERROR\_NONE) {

\_\_SHEAP\_FREE(pled);

\_DebugHandle(\_\_ERRORMESSAGE\_LEVEL, "system low memory.\r\n");

\_ErrorHandle(ERROR\_SYSTEM\_LOW\_MEMORY);

return (PX\_ERROR);

}

\_ErrorHandle(ERROR\_NONE);

return (ERROR\_NONE);

}

void module\_init (void)

{

ledDrv();

ledDevCreate("/dev/led1", IMX6Q\_GPIO\_NUMR(1, 24), LW\_FALSE);

ledDevCreate("/dev/led2", IMX6Q\_GPIO\_NUMR(1, 30), LW\_FALSE);

ledDevCreate("/dev/led3", IMX6Q\_GPIO\_NUMR(1, 26), LW\_FALSE);

ledDevCreate("/dev/led4", IMX6Q\_GPIO\_NUMR(1, 27), LW\_FALSE);

ledDevCreate("/dev/led5", IMX6Q\_GPIO\_NUMR(2, 5), LW\_FALSE);

ledDevCreate("/dev/led6", IMX6Q\_GPIO\_NUMR(2, 0), LW\_FALSE);

ledDevCreate("/dev/led7", IMX6Q\_GPIO\_NUMR(1, 29), LW\_FALSE);

}

void module\_exit (void)

{

PLW\_DEV\_HDR pdevHdr;

pdevHdr = API\_IosDevFind("/dev/led1", LW\_NULL);

if (pdevHdr) {

API\_IosDevDelete(pdevHdr);

}

pdevHdr = API\_IosDevFind("/dev/led2", LW\_NULL);

if (pdevHdr) {

API\_IosDevDelete(pdevHdr);

}

pdevHdr = API\_IosDevFind("/dev/led3", LW\_NULL);

if (pdevHdr) {

API\_IosDevDelete(pdevHdr);

}

pdevHdr = API\_IosDevFind("/dev/led4", LW\_NULL);

if (pdevHdr) {

API\_IosDevDelete(pdevHdr);

}

pdevHdr = API\_IosDevFind("/dev/led5", LW\_NULL);

if (pdevHdr) {

API\_IosDevDelete(pdevHdr);

}

pdevHdr = API\_IosDevFind("/dev/led6", LW\_NULL);

if (pdevHdr) {

API\_IosDevDelete(pdevHdr);

}

pdevHdr = API\_IosDevFind("/dev/led7", LW\_NULL);

if (pdevHdr) {

API\_IosDevDelete(pdevHdr);

}

API\_IosDrvRemove(\_G\_iLedDrvNum, LW\_TRUE); }

**LED驱动的使用**

LED驱动装载后，可以使用标准的文件操作函数对LED进行操作，如程序清单 10.5。

程序清单 10.5 led\_drv\_use.c

#include <stdio.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

int fb;

long time\_ms;

int i;

fb = open("/dev/led1", O\_RDWR, 0666);

if (fb < 0) {

printf("failed to open /dev/led2\n");

return (-1);

}

/\*

\* LED灯闪烁10次，每次亮100ms，1s后重新点亮。

\*/

for (i = 0; i < 10; i++) {

time\_ms = 100;

ioctl(fb, 0, time\_ms);

sleep(1);

}

close(fb);

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译工程led\_drv\_modu和led\_drv\_use，将生成的程序文件上传到验证平台；
2. 装载LED驱动内核模块；

[root@sylixos\_station:/lib/modules]# modulereg led\_drv\_modu.ko

module led\_drv\_modu.ko register ok, handle 0x2468c680

1. 将工作目录切换到/apps/led\_drv\_use，执行程序led\_drv\_use，会看到LED在闪烁，闪烁10次后退出。

### 扩展实验

本实验默认是IMX6Q芯片的LED驱动，根据上述介绍，将其移植到其它芯片上。

# I2C驱动实验

I2C是嵌入式系统中常用的板级通信总线，电源控制、EEPROM、RTC时钟芯片、摄像头初始化等均使用I2C总线。I2C驱动包括两个方面，一个是处理器的I2C总线控制器驱动，一个是通过I2C总线扩展的外设驱动。本章通过两个实验分别讲解两种驱动程序的设计。

## 实验一 I2C驱动编写实验

### 实验目的

* 了解I2C驱动的实现原理和特点
* 掌握编写I2C驱动的方法

### 实验内容

* 阅读芯片数据手册，了解芯片IMX6Q的I2C控制器功能和使用方法
* 阅读实验原理，学习SylixOS下I2C驱动的框架和编写特点
* I2C驱动的编译使用

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

**I2C串行总线概述**

I2C总线（I2C bus，Inter-IC bus）是PHILIPS公司于80年代推出的一种两线式串行总线，用于集成电路之间的互联通信，是微电子通信控制领域广泛采用的一种总线标准，是具备多主机系统所需的包括总线裁决和高低速器件同步功能的高性能串行总线。

I2C总线只有两根双向信号线。一根是数据线SDA，另一根是时钟线SCL。



图 11.1 I2C总线拓扑图

I2C总线通过上拉电阻接正电源。当总线空闲时，两根线均为高电平。连到总线上的任一器件输出低电平，都会使总线的信号拉低，即各器件的SDA及SCL都是“线与”关系。如图 11.2所示。

每个连接到I2C总线上的器件都有唯一的地址。主机与器件间的数据传送可以是由主机发送数据到其它器件，这时主机即为发送器。总线上接收数据的器件则为接收器。

在多主机系统中，可能同时有几个主机企图启动总线传送数据。为了避免混乱， I2C总线要通过总线仲裁，以决定由哪一台主机控制总线。

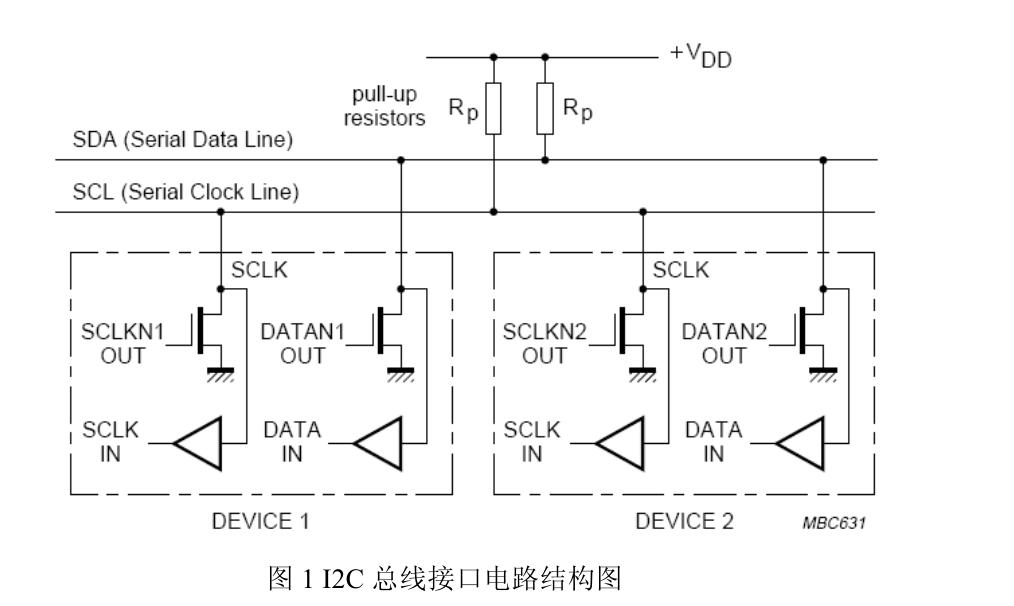


图 11.2 I2C “线与”实现原理

**I2C总线的数据传送**

* 数据位的有效性规定

I2C总线进行数据传送时，时钟信号为高电平期间，数据线上的数据必须保持稳定，只有在时钟线上的信号为低电平期间，数据线上的电平状态才允许变化。

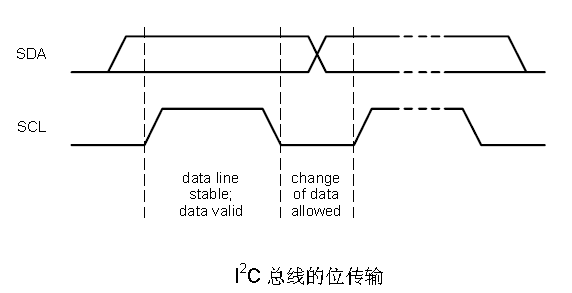


图 11.3 I2C数据位有效性示意图

* 起始和终止信号

SCL为高电平期间，SDA由高电平向低电平的变化表示起始信号；SCL线为高电平期间，SDA线由低电平向高电平的变化表示终止信号。

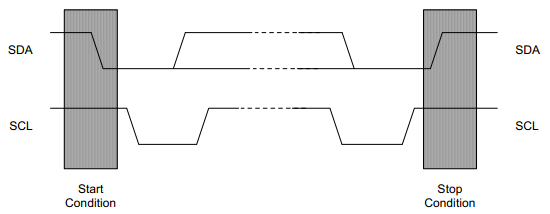


图 11.4 I2C信号示意图

起始和终止信号都是由主机发出的，在起始信号产生后，总线就处于被占用的状态；在终止信号产生后，总线就处于空闲状态。

* 数据传送格式

字节传送与应答：

每一个字节必须保证是8位长度。数据传送时，先传送最高位（MSB），每一个被传送的字节后面都必须跟随一位应答位（即一帧共有9位）。

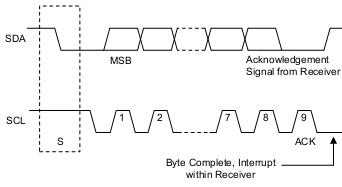


图 11.5 I2C字节传送

当主机接收数据时，它收到最后一个数据字节后，必须向从机发出一个结束传送的信号。这个信号是由对从机的“非应答”来实现的。然后，从机释放SDA线，以允许主机产生终止信号。

数据帧格式：

I2C总线上传送的数据信号是广义的，既包括地址信号，又包括真正的数据信号。

在起始信号后必须传送一个从机的地址（7位），第8位是数据的传送方向位（R/），用“0”表示主机发送数据（T），“1”表示主机接收数据（R）。每次数据传送总是由主机产生的终止信号结束。在总线的一次数据传送过程中，可以有以下几种组合方式：

* 主机向从机发送数据，数据传送方向在整个传送过程中不变：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| S | 从机地址 | 0 | A | 数据 | A | 数据 | A/Ā | P |

注：有阴影部分表示数据由主机向从机传送，无阴影部分则表示数据由从机向主机传送。

S表示起始信号，P表示终止信号。A表示应答，Ā表示非应答（高电平）。

* 主机在第一个字节后，立即由从机读数据

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| S | 从机地址 | 1 | A | 数据 | A | 数据 | Ā | P |

* 在传送过程中，当需要改变传送方向时，起始信号和从机地址都被重复产生一次，但两次读/写位正好反相。

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| S | 从机地址 | 0 | A | 数据 | A/Ā | S | 从机地址 | 1 | A | 数据 | Ā | P |

* 总线的寻址

I2C总线协议有明确的规定：采用7位的寻址字节（寻址字节是起始信号后的第一个字节）。

* 寻址字节的位定义

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 位 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  | 从机地址 | | | | | | | R/W |

D7～D1位组成从机的地址。D0位是数据传送方向位，为“0”时表示主机向从机写数据，为“1”时表示主机由从机读数据。

主机发送地址时，总线上的每个从机都将这7位地址码与自己的地址进行比较，如果相同，则认为自己正被主机寻址，根据R/W位将自己确定为发送器或接收器。

从机的地址由固定部分和可编程部分组成。在一个系统中可能希望接入多个相同的从机，从机地址中可编程部分决定了可接入总线该类器件的最大数目。如一个从机的7位寻址位有4位是固定位，3位是可编程位，这时仅能寻址8个同样的器件，即可以有8个同样的器件接入到该I2C总线系统中。

* 寻址字节中的特殊地址

固定地址编号0000和1111已被保留作为特殊用途。

表 11.1 I2C特殊地址说明

|  |  |  |  |
| --- | --- | --- | --- |
| 地址位 | | R/W | 意义 |
| 0 0 0 0 | 0 0 0 | 0 | 通用呼叫地址 |
| 0 0 0 0 | 0 0 0 | 1 | 起始地址 |
| 0 0 0 0 | 0 0 1 | X | CBUS地址 |
| 0 0 0 0 | 0 1 0 | X | 为不同总线保留地址 |
| 0 0 0 0 | 0 1 1 | X | 保留 |
| 0 0 0 0 | 1 X X | X |
| 1 1 1 1 | 1 X X | X |
| 1 1 1 1 | 0 X X | X | 十位从机地址 |

起始信号后的第一字节的8位为“0000 0000”时，称为通用呼叫地址。通用呼叫地址的用意在第二字节中加以说明。格式为：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 第一个字节（通用呼叫地址） | | | | | | | |  | 第二个字节 LSB | | | | | | | |  |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | A | X | X | X | X | X | X | X | B | A |

第二字节为 06H时，所有能响应通用呼叫地址的从机器件复位，并由硬件装入从机地址的可编程部分。能响应命令的从机器件复位时不拉低SDA和SCL线，以免堵塞总线。

第二字节为 04H时，所有能响应通用呼叫地址并通过硬件来定义其可编程地址的从机器件将锁定地址中的可编程位，但不进行复位。

如果第二字节的方向位B为“1”，则这两个字节命令称为硬件通用呼叫命令。

在这第二字节的高7位说明自己的地址。接在总线上的智能器件，如单片机或其他微处理器能识别这个地址，并与之传送数据。硬件主器件作为从机使用时，也用这个地址作为从机地址。格式为：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| S | 0000 0000 | A | 主机地址 | 1 | A | 数据 | A | 数据 | A | P |

在系统中另一种选择是系统复位时硬件主机器件工作在从机接收器方式，这时由系统中的主机先告诉硬件主机器件数据应送往的从机器件地址，当硬件主机器件要发送数据时就可以直接向指定从机器件发送数据了。

* 起始字节

起始字节是提供给没有I2C总线接口的单片机查询I2C总线时使用的特殊字节。

不具备I2C总线接口的单片机，必须通过软件不断地检测总线，以便及时地响应总线的请求。单片机的速度与硬件接口器件的速度就出现了较大的差别，为此，I2C总线上的数据传送要由一个较长的起始过程加以引导。

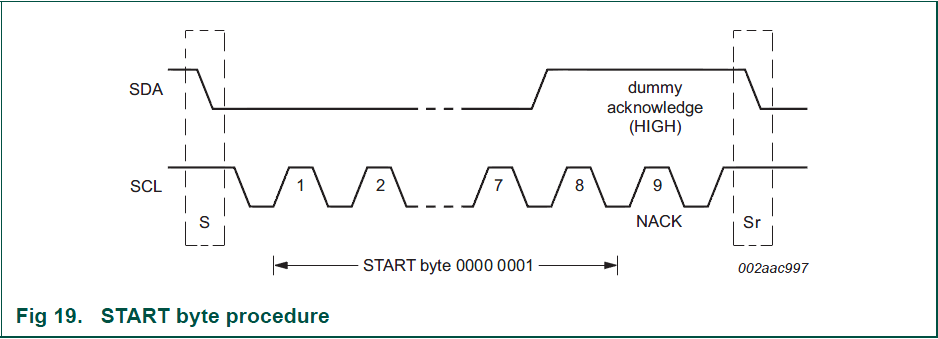


图 11.6 I2C起始字节

引导过程由起始信号、起始字节、应答位、重复起始信号（Sr）组成。

请求访问总线的主机发出起始信号后，发送起始字节（0000 0001），另一个单片机可以用一个比较低的速率采样SDA线，直到检测到起始字节中的7个“0”中的一个为止。在检测到SDA线上的高电平后，单片机就可以用较高的采样速率，以便寻找作为同步信号使用的第二个起始信号Sr。

在起始信号后的应答时钟脉冲仅仅是为了和总线所使用的格式一致，并不要求器件在这个脉冲期间作应答。

**IMX6Q的I2C总线说明**

在IMX6Q用户手册的Chapter 34\_I2CController(I2C)中详细说明。

驱动编写时需要重点阅读部分如下：

* IMX6Q的I2C总线

IMX6Q芯片有三路I2C总线，可在IMX6Q用户手册 Chapter4节查找。如表 11.2所示：

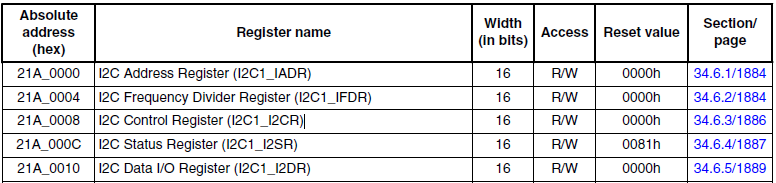
表 11.2 IMX6Q I2C信号说明

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Signal | I/O | Description | Pad | Type |
| I2C1\_SCL | Input/Output | I2C-Bus Interface1 Serial Clock Line | EIM\_D21 | muxed |
| I2C1\_SDA | Input/Output | I2C-Bus Interface1 Serial Data Line | EIM\_D28 | muxed |
| I2C2\_SCL | Input/Output | I2C-BUS Interface2 Serial Clock Line | KEY\_COL3 | muxed |
| I2C2\_SDA | Input/Output | I2C-BUS Interface2 Serial Data Line | KEY\_ROW3 | muxed |
| I2C3\_SCL | Input/Output | I2C-Bus Interface3 Serial Clock Line | GPIO\_3 | muxed |
| I2C3\_SDA | Input/Output | I2C-Bus Interface3 Serial Data Line | GPIO\_6 | muxed |

* IMX6Q的I2C总线控制器

以I2C1总线为例，说明IMX6Q的I2C总线控制器相关的特殊功能寄存器：

表 11.3 IMX6Q I2C控制器部分寄存器说明



注：特殊功能寄存器的详细说明请参考IMX6Q的芯片用户手册。

* IMX6Q的I2C总线操作流程

必须要在任何I2C总线发送或接收数据前被执行如下操作：

* 写自己的从机地址到I2C1\_IADR寄存器；
* 设置I2C1\_IFDR寄存器，定义SCL时钟周期；
* 设置I2C1\_I2CR使能串行输出，使能中断，设置主从设备模式。

对IMX6Q芯片来讲，一般情况下在I2C总线上总是做主机，因此这里主要介绍IMX6Q做主机时数据的发送和接收流程。

IMX6Q IIC中断模式收发数据流程图如图 11.7：

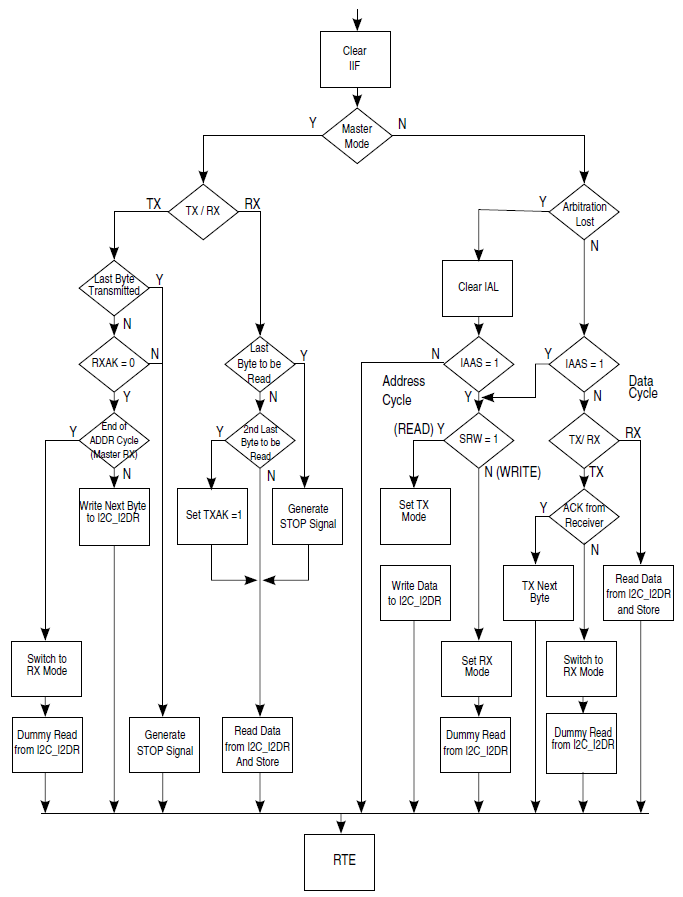


图 11.7 IMX6Q I2C中断模式收发数据流程图

IMX6Q IIC查询模式收发数据流程图如图 11.8：

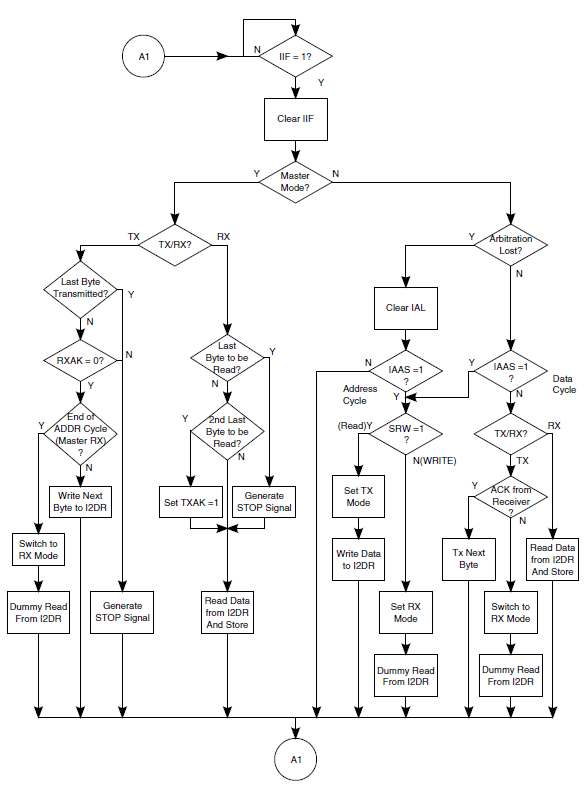


图 11.8 IMX6Q I2C查询模式收发数据流程图

**SylixOS对I2C总线的抽象**

一个I2C总线驱动可能同时应用在多个I2C控制器上（如上面的I2C1、I2C2、I2C3使用同一份I2C总线驱动代码），I2C总线驱动抽象为I2C 总线传输函数集：

typedef struct lw\_i2c\_funcs {

INT (\*I2CFUNC\_pfuncMasterXfer)(PLW\_I2C\_ADAPTER pi2cadapter,

PLW\_I2C\_MESSAGE pi2cmsg,

INT iNum);

/\* 适配器数据传输 \*/

INT (\*I2CFUNC\_pfuncMasterCtl)(PLW\_I2C\_ADAPTER pi2cadapter,

INT iCmd,

LONG lArg);

/\* 适配器控制 \*/

} LW\_I2C\_FUNCS;

typedef LW\_I2C\_FUNCS \*PLW\_I2C\_FUNCS;

I2C总线控制器（如上面的I2C1、I2C2、I2C3）被抽象为I2C总线适配器并作为I2C 总线传输函数集内的函数的第一个参数：

typedef struct lw\_i2c\_adapter {

LW\_BUS\_ADAPTER I2CADAPTER\_pbusadapter; /\* 总线节点 \*/

struct lw\_i2c\_funcs \*I2CADAPTER\_pi2cfunc; /\* 总线适配器操作函数 \*/

LW\_OBJECT\_HANDLE I2CADAPTER\_hBusLock; /\* 总线操作锁 \*/

ULONG I2CADAPTER\_ulTimeout; /\* 操作超时时间 \*/

INT I2CADAPTER\_iRetry; /\* 重试次数 \*/

LW\_LIST\_LINE\_HEADER I2CADAPTER\_plineDevHeader; /\* 设备链表 \*/

} LW\_I2C\_ADAPTER;

typedef LW\_I2C\_ADAPTER \*PLW\_I2C\_ADAPTER;

I2C总线上的从设备（如AT24C02芯片）被抽象为I2C设备：

typedef struct lw\_i2c\_device {

UINT16 I2CDEV\_usAddr; /\* 设备地址 \*/

UINT16 I2CDEV\_usFlag; /\* 标志, 仅支持 10bit 地址 \*/

#define LW\_I2C\_CLIENT\_TEN 0x10 /\* 与 LW\_I2C\_M\_TEN 相同 \*/

PLW\_I2C\_ADAPTER I2CDEV\_pi2cadapter; /\* 挂载的适配器 \*/

LW\_LIST\_LINE I2CDEV\_lineManage; /\* 设备挂载链 \*/

atomic\_t I2CDEV\_atomicUsageCnt /\* 设备使用计数 \*/

CHAR I2CDEV\_cName[LW\_CFG\_OBJECT\_NAME\_SIZE]; /\* 设备的名称 \*/

} LW\_I2C\_DEVICE;

typedef LW\_I2C\_DEVICE \*PLW\_I2C\_DEVICE;

I2C总线适配器与I2C设备之间的数据传输格式被抽象为I2C 总线传输控制消息：

typedef struct lw\_i2c\_message {

UINT16 I2CMSG\_usAddr; /\* 器件地址 \*/

UINT16 I2CMSG\_usFlag; /\* 传输控制参数 \*/

#define LW\_I2C\_M\_TEN 0x0010 /\* 使用 10bit 设备地址 \*/

#define LW\_I2C\_M\_RD 0x0001 /\* 为读操作, 否则为写 \*/

#define LW\_I2C\_M\_NOSTART 0x4000 /\* 不发送 start 标志 \*/

#define LW\_I2C\_M\_REV\_DIR\_ADDR 0x2000 /\* 读写标志位反转 \*/

#define LW\_I2C\_M\_IGNORE\_NAK 0x1000 /\* 忽略 ACK NACK \*/

#define LW\_I2C\_M\_NO\_RD\_ACK 0x0800 /\* 读操作时不发送 ACK \*/

#define LW\_I2C\_M\_RECV\_LEN 0x0400 /\* !目前不支持! \*/

UINT16 I2CMSG\_usLen; /\* 长度(缓冲区大小) \*/

UINT8 \*I2CMSG\_pucBuffer; /\* 缓冲区 \*/

} LW\_I2C\_MESSAGE;

typedef LW\_I2C\_MESSAGE \*PLW\_I2C\_MESSAGE;

**SylixOS的I2C总线操作函数**

INT API\_I2cLibInit(VOID);

* 功能：I2C总线相关的函数必须在调用API\_I2cLibInit函数完成相关初始化后才可以使用，在BSP工程bspimx6中的imx6q\_sylixos.c/boardBusInit(VOID) 函数调用该函数。

INT API\_I2cAdapterCreate(CPCHAR pcName,

PLW\_I2C\_FUNCS pi2cfunc,

ULONG ulTimeout,

INT iRetry);

* 功能：创建一个I2C总线适配器，同样是在BSP工程bspimx6中的imx6q\_sylixos.c/boardBusInit(VOID) 函数调用调用该函数。

PLW\_I2C\_DEVICE API\_I2cDeviceCreate(CPCHAR pcAdapterName,

CPCHAR pcDeviceName,

UINT16 usAddr,

UINT16 usFlag);

* 功能：创建一个I2C设备；
* pcAdapterName： 适配器的名字；
* pcDeviceName： 设备的名字（一般形如“/dev/eeprom”）；
* usAddr： 设备的从机地址。

INT API\_I2cDeviceTransfer(PLW\_I2C\_DEVICE pi2cdevice,

PLW\_I2C\_MESSAGE pi2cmsg,

INT iNum);

* 功能：I2C总线驱动程序完成与I2C设备的数据传输；
* pi2cdevice： I2C设备；
* pi2cmsg： I2C 总线传输控制消息数组的基地址；
* iNum： 消息的数目。

INT API\_I2cDeviceMasterSend(PLW\_I2C\_DEVICE pi2cdevice,

CPCHAR pcBuffer,

INT iCount);

* 功能：实现主机的数据发送。

INT API\_I2cDeviceMasterRecv(PLW\_I2C\_DEVICE pi2cdevice,

PCHAR pcBuffer,

INT iCount);

* 功能：实现主机的数据接收。

INT API\_I2cDeviceCtl(PLW\_I2C\_DEVICE pi2cdevice,

INT iCmd,

LONG lArg);

* 功能：对I2C总线设置参数，依赖于I2C 总线传输函数集的I2CFUNC\_pfuncMasterCtl函数实现，使用该函数时，I2CFUNC\_pfuncMasterCtl函数将被调用，命令iCmd和参数lArg也将传入I2CFUNC\_pfuncMasterCtl函数。

### 实验步骤

1. 编译I2C驱动所在BSP工程bspimx6，生成bspimx6.bin文件；
2. 参考1.4章节，将文件bspimx6.bin上传到验证平台上，在系统调试时并不需要将文件烧录到flash上，完成文件上传后，输入命令“go 0x10000000”启动SylixOS系统。
3. 在控制台中输入命令“buss”查看系统总线设备，

[root@sylixos\_station:/dev]# buss

BUS NAME DEV NUM

-------------- -------

/bus/sd/2 0

/bus/sd/3 1

/bus/i2c/2 0

/bus/i2c/1 1

/bus/i2c/0 1

[root@sylixos\_station:/dev]#

### 扩展实验

仔细阅读SylixOS中I2C驱动框架部分代码，总结分析SylixOS管理I2C总线的特点。

## 实验二 ZLG7290驱动编写实验

### 实验目的

* 了解ZLG7290的控制流程
* 掌握使用I2C总线读写ZLG7290的静态驱动程序设计方法

### 实验内容

* 阅读实验原理，了解ZLG7290的读写流程和I2C总线的使用方法
* 根据实验步骤完成对ZLG7290的读写程序设计和验证

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

**ZLG7290介绍**

ZLG7290 是广州周立功单片机发展有限公司自行设计的数码管显示驱动及键盘扫描管理芯片。能够直接驱动8位共阴极数码管（或64只独立的LED），同时还可以扫面管理多大64只按键。其中有8只按键可以作为功能键使用，就像电脑键盘上的Ctrl，Shift、Alt键一样。另外ZLG7290 内部还设有连击计数器，能够使某键按下后不松手而连续有效。该芯片为工业级芯片，抗干扰能力强，在工业测控中已有大量应用。该器件通过I2C总线接口进行操作，ZLG7290引脚图如图 11.9。

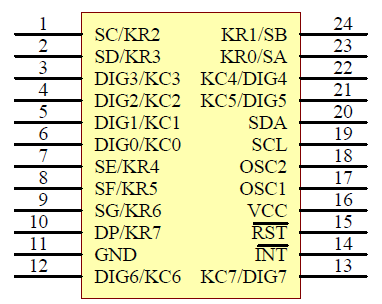


图 11.9 ZLG7290引脚

表 11.4说明了ZLG7290各引脚的功能。

表 11.4 ZLG7290引脚功能

|  |  |  |
| --- | --- | --- |
| 引脚序号 | 引脚名 | 功能 |
| 1 | SC/KR2 | 数码管c 段／键盘行信号2 |
| 2 | SD/KR3 | 数码管d 段／键盘行信号3 |
| 3 | DIG3/KC3 | 数码管位选信号3／键盘列信号3 |
| 4 | DIG2/KC2 | 数码管位选信号2／键盘列信号2 |
| 5 | DIG1/KC1 | 数码管位选信号1／键盘列信号1 |
| 6 | DIG0/KC0 | 数码管位选信号0／键盘列信号0 |
| 7 | SE/KR4 | 数码管e 段／键盘行信号4 |
| 8 | SF/KR5 | 数码管f 段／键盘行信号5 |
| 9 | SG/KR6 | 数码管g 段／键盘行信号6 |
| 10 | DP/KR7 | 数码管dp 段／键盘行信号7 |
| 11 | GND | 接地 |
| 12 | DIG6/KC6 | 数码管位选信号6／键盘列信号6 |
| 13 | DIG7/KC7 | 数码管位选信号7／键盘列信号7 |
| 14 | INT | 键盘中断请求信号，低电平（下降沿）有效 |
| 15 | RST | 复位信号，低电平有效 |
| 16 | Vcc | 电源，＋3.3～5.5V |
| 17 | OSC1 | 晶振输入信号 |
| 18 | OSC2 | 晶振输出信号 |
| 19 | SCL | I2C 总线时钟信号 |
| 20 | SDA | I2C 总线数据信号 |
| 21 | DIG5/KC5 | 数码管位选信号5／键盘列信号5 |
| 22 | DIG4/KC4 | 数码管位选信号4／键盘列信号4 |
| 23 | SA/KR0 | 数码管a 段／键盘行信号0 |
| 24 | SB/KR1 | 数码管b 段／键盘行信号1 |

**ZLG7290 驱动架构**

****

**ZLG7290驱动编写**

在ZLG7290硬件下创建了两个设备驱动分别为矩阵按键设备和数码管设备。ZLG7290主要通过zlg7290DevCreate函数把ZLG7290硬件设备绑定到由API\_I2cAdapterCreate函数创建的i2c总线适配器上。ZLG7290设备的读写通过alg7290ReadReg和zlg7290WriteCmd函数经过API\_I2cDeviceTransfer函数与i2c总线进行通信。

为了使驱动代码的可移植性的提高，把矩阵按键设备和数码管设备在实现设备文件操作控制块结构体file\_operations的成员函数之上进行了一次hal层的封装，这样当换用其他芯片的时候只需修改hal层的驱动函数即可，大大提高了可移植性。这里只介绍数码管的驱动实现，矩阵按键的驱动程序设计作为扩展实验课下自行完成。

数码管digitlcd实现设备文件操作控制块结构体file\_operations的成员函数，在digitlcd设备驱动中主要实现了digitLcdOpen、digitLcdClose、digitLcdRead、digitLcdWrite、digitLcdIoctl函数功能。

应用程序可以通过访问标准文件I/O函数来读写ZLG7290设备，据以上说明仔细阅读ZLG7290数码管驱动的代码程序清单 11.1，了解其实现的具体方法。

程序清单 11.1 digitlcd.c

#define \_\_SYLIXOS\_KERNEL

#include "config.h"

#include "SylixOS.h"

#include "stdlib.h"

#include "string.h"

#include "digitlcd.h"

#include "digitlcd\_hal.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Digital LCD （数码管） 控制器类型定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typedef struct {

LW\_DEV\_HDR DLCD\_devHdr; /\* 必须是第一个结构体成员 \*/

LW\_LIST\_LINE\_HEADER DLCD\_devHdr\_fdNodeHeader;

PVOID DLCD\_pvHwPrivate;

INT DLCD\_iSize;

} \_\_DLCD\_CTRL, \*\_\_PDLCD\_CTRL;

static UINT32 \_G\_iDigitLCDDrvNum = 0;

static \_\_DLCD\_CTRL \_G\_digitLcdCtrl;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

驱动程序IO接口实现

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: digitLcdOpen

\*\* 功能描述: 驱动open函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static LONG digitLcdOpen (\_\_PDLCD\_CTRL pCtrl, PCHAR pcName, INT iFlags, INT iMode)

{

PLW\_FD\_NODE pFdNode;

BOOL bIsNew;

if (pcName == LW\_NULL) {

\_ErrorHandle(ERROR\_IO\_NO\_DEVICE\_NAME\_IN\_PATH);

return (PX\_ERROR);

}

else {

pFdNode = API\_IosFdNodeAdd(&pCtrl->DLCD\_devHdr\_fdNodeHeader, (dev\_t)pCtrl, 0,iFlags, iMode, 0, 0, 0, LW\_NULL, &bIsNew);

if (pFdNode == LW\_NULL) {

printk(KERN\_ERR "digitLcdOpen(): failed to add fd node!\n");

return (PX\_ERROR);

}

LW\_DEV\_INC\_USE\_COUNT(&pCtrl->DLCD\_devHdr);

return ERROR\_NONE;

}

return (PX\_ERROR);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: digitLcdClose

\*\* 功能描述: 驱动close函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static INT digitLcdClose (PLW\_FD\_ENTRY pFdEntry)

{

\_\_PDLCD\_CTRL pCtrl = (\_\_PDLCD\_CTRL)pFdEntry->FDENTRY\_pdevhdrHdr;

LW\_DEV\_DEC\_USE\_COUNT(&pCtrl->DLCD\_devHdr);

return (ERROR\_NONE);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: digitLcdIoctl

\*\* 功能描述: 驱动Ioctl函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static INT digitLcdIoctl (PLW\_FD\_ENTRY pFdEntry, INT iCmd, LONG lArg)

{

\_\_PDLCD\_CTRL pCtrl = (\_\_PDLCD\_CTRL)pFdEntry->FDENTRY\_pdevhdrHdr;

INT \*iSize = (INT \*)lArg;

switch (iCmd) {

case FIONREAD: /\* 获取数码管数量 \*/

\*iSize = pCtrl->DLCD\_iSize;

break;

default:

\_ErrorHandle(ENOSYS);

return (PX\_ERROR);

}

return \*iSize;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: handleCharWriteRequest

\*\* 功能描述: 处理字符写入请求

\*\* 输　入 : pvBuf 写缓冲

\*\* 输　出 : ERROR\_CODE

\*\* 全局变量:

\*\* 调用模块:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static INT handleCharWriteRequest(\_\_PDLCD\_CTRL pCtrl, PVOID pvBuf)

{

INT i;

PUCHAR pucBuf = (PUCHAR) pvBuf;

for (i = 0; i < pCtrl->DLCD\_iSize; i++) {

digitLcdWriteChar(pCtrl->DLCD\_pvHwPrivate, i, pucBuf[i]); /\* 向数码管写入一个字符 \*/

}

return (ERROR\_NONE);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: handleDotWriteRequest

\*\* 功能描述: 处理字符写入请求

\*\* 输　入 : pvBuf 写缓冲

\*\* 输　出 : ERROR\_CODE

\*\* 全局变量:

\*\* 调用模块:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static INT handleDotWriteRequest(\_\_PDLCD\_CTRL pCtrl, PVOID pvBuf)

{

INT i;

PUCHAR pucBuf = (PUCHAR) pvBuf;

for (i = 0; i < pCtrl->DLCD\_iSize; i++) {

digitLcdWriteDot(pCtrl->DLCD\_pvHwPrivate, i, pucBuf[i]); /\* 向数码管开关一个点显示 \*/

}

return (ERROR\_NONE);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: digitLcdWrite

\*\* 功能描述: 驱动 write 函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static ssize\_t digitLcdWrite (PLW\_FD\_ENTRY pFdentry, PVOID pvBuf, size\_t stLen)

{

\_\_PDLCD\_CTRL pCtrl = (\_\_PDLCD\_CTRL)pFdentry->FDENTRY\_pdevhdrHdr;

PUCHAR pucBuf = (PUCHAR) pvBuf;

if (pvBuf != NULL && (stLen > 0)) { /\* 缓冲不为空 \*/

/\*

\* 检查请求类型

\*/

if (!strncmp(DLCD\_CHAR\_TAG, (const PCHAR)pucBuf, strlen(DLCD\_CHAR\_TAG))) {

handleCharWriteRequest(pCtrl, pucBuf + strlen(DLCD\_CHAR\_TAG));

}

else if (!strncmp(DLCD\_DOT\_TAG, (const PCHAR)pucBuf, strlen(DLCD\_DOT\_TAG))) {

handleDotWriteRequest(pCtrl, pucBuf + strlen(DLCD\_DOT\_TAG));

}

else {

goto \_\_DIGIT\_WRITE\_ERROR;

}

}

else {

goto \_\_DIGIT\_WRITE\_ERROR;

}

return (stLen);

\_\_DIGIT\_WRITE\_ERROR:

\_ErrorHandle(ERROR\_IOS\_DRIVER\_NOT\_SUP);

printk(KERN\_ERR "digitLcdWrite(): invalid parameters : %s.\n", pucBuf);

return (PX\_ERROR);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: handleCharReadRequest

\*\* 功能描述: 处理字符读取请求

\*\* 输　入 : pvBuf 写缓冲

\*\* 输　出 : ERROR\_CODE

\*\* 全局变量:

\*\* 调用模块:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static INT handleCharReadRequest(\_\_PDLCD\_CTRL pCtrl, PVOID pvBuf)

{

INT i;

PUCHAR pucBuf = (PUCHAR) pvBuf;

for (i = 0; i < pCtrl->DLCD\_iSize; i++) {

/\*

\* 从数码管读出一个字节

\* 最后缓冲按照"123a"格式返回

\*/

pucBuf[i] = digitLcdReadChar(pCtrl->DLCD\_pvHwPrivate, i);

}

return (ERROR\_NONE);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: handleDotReadRequest

\*\* 功能描述: 处理字符读取请求

\*\* 输　入 : pvBuf 写缓冲

\*\* 输　出 : ERROR\_CODE

\*\* 全局变量:

\*\* 调用模块:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static INT handleDotReadRequest(\_\_PDLCD\_CTRL pCtrl, PVOID pvBuf)

{

INT i;

PUCHAR pucBuf = (PUCHAR) pvBuf;

for (i = 0; i < pCtrl->DLCD\_iSize; i++) {

/\*

\* 从数码管读出一个DOT状态

\* 最后缓冲按照"1001"格式返回

\*/

pucBuf[i] = digitLcdReadDot(pCtrl->DLCD\_pvHwPrivate, i);

}

return (ERROR\_NONE);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: digitLcdRead

\*\* 功能描述: 驱动 read 函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static ssize\_t digitLcdRead (PLW\_FD\_ENTRY pFdentry, PVOID pvBuf, size\_t stLen)

{

\_\_PDLCD\_CTRL pCtrl = (\_\_PDLCD\_CTRL)pFdentry->FDENTRY\_pdevhdrHdr;

PUCHAR pucBuf = (PUCHAR) pvBuf;

if (pvBuf != NULL && (stLen >= (pCtrl->DLCD\_iSize + 1))) { /\* 缓冲不为空而且够空间 \*/

/\*

\* 检查请求类型

\*/

if (!strncmp(DLCD\_CHAR\_TAG, (const PCHAR)pucBuf, strlen(DLCD\_CHAR\_TAG))) {

handleCharReadRequest(pCtrl, pucBuf); /\* CHAR-开头 \*/

}

else if (!strncmp(DLCD\_DOT\_TAG, (const PCHAR)pucBuf, strlen(DLCD\_CHAR\_TAG))) {

handleDotReadRequest(pCtrl, pucBuf); /\* DOT-开头 \*/

}

else {

goto \_\_DIGIT\_READ\_ERROR;

}

}

else {

goto \_\_DIGIT\_READ\_ERROR;

}

return (pCtrl->DLCD\_iSize);

\_\_DIGIT\_READ\_ERROR:

\_ErrorHandle(ERROR\_IOS\_DRIVER\_NOT\_SUP);

printk(KERN\_ERR "digitLcdWrite(): invalid parameters : %s.\n", pucBuf);

return (PX\_ERROR);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: digitLcdDrv

\*\* 功能描述: 安装 Digit LCD 驱动函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT digitLcdDrv (VOID)

{

struct file\_operations fileop;

if (\_G\_iDigitLCDDrvNum) {

return (ERROR\_NONE);

}

lib\_memset(&fileop, 0, sizeof(struct file\_operations));

fileop.owner = THIS\_MODULE;

fileop.fo\_create = digitLcdOpen;

fileop.fo\_open = digitLcdOpen;

fileop.fo\_close = digitLcdClose;

fileop.fo\_ioctl = digitLcdIoctl;

fileop.fo\_write = digitLcdWrite;

fileop.fo\_read = digitLcdRead;

\_G\_iDigitLCDDrvNum = iosDrvInstallEx2(&fileop, LW\_DRV\_TYPE\_NEW\_1);

DRIVER\_LICENSE(\_G\_iDigitLCDDrvNum, "Dual BSD/GPL->Ver 1.0");

DRIVER\_AUTHOR(\_G\_iDigitLCDDrvNum, "Xu.GuiZhou");

DRIVER\_DESCRIPTION(\_G\_iDigitLCDDrvNum, "Digit LCD driver.");

return (\_G\_iDigitLCDDrvNum > 0) ? (ERROR\_NONE) : (PX\_ERROR);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: digitlcdDevRemove

\*\* 功能描述: 删除digitlcd设备

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

VOID digitlcdDevRemove (PCHAR pcName)

{

PLW\_DEV\_HDR pdevhdr;

PCHAR pcTail;

pdevhdr = iosDevFind(pcName, &pcTail);

if (!pdevhdr) {

return;

}

iosDevFileAbnormal(pdevhdr);

iosDevDelete(pdevhdr);

iosDrvRemove(\_G\_iDigitLCDDrvNum, LW\_FALSE);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: digitLcdDevCreate

\*\* 功能描述: 安装 Digit LCD 设备

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT digitLcdDevCreate(PCHAR pcName, PVOID pvHwPrivate)

{

\_\_PDLCD\_CTRL pCtrl = &\_G\_digitLcdCtrl; /\* 可扩展成支持多设备 \*/

if (pcName == NULL || (pvHwPrivate == NULL)) {

printk(KERN\_ERR "digitLcdDevCreate(): device name invalid!\n");

return (PX\_ERROR);

}

pCtrl->DLCD\_pvHwPrivate = pvHwPrivate; /\* 保存HW private信息 \*/

if (API\_IosDevAddEx(&pCtrl->DLCD\_devHdr, pcName, \_G\_iDigitLCDDrvNum, DT\_CHR) != ERROR\_NONE) {

printk(KERN\_ERR "digitLcdDevCreate(): can not add device : %s.\n", strerror(errno));

return (PX\_ERROR);

}

pCtrl->DLCD\_iSize = dlcdInit(pvHwPrivate); /\* 初始化数码管 \*/

return (ERROR\_NONE);

}

ZLG7290的驱动编写方法为内核模块驱动，其中digitLcdDrv函数和digitLcdDevCreate函数分别在zlg7290\_module.c文件中的module\_init函数中调用。

在程序清单 11.1中用到了读写请求函数所调用的读写数码管的字符或小数点的操作函数，在文件digitlcd\_hal.c中实现。函数具体实现如程序清单11.2所示。

程序清单 11.2 digitlcd\_hal.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 描 述: 数码管驱动硬件抽象层(HAL)接口实现

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define \_\_SYLIXOS\_KERNEL

#include "config.h"

#include "SylixOS.h"

#include "stdlib.h"

#include "string.h"

#include "digitlcd.h"

#include "digitlcd\_hal.h"

#include "zlg7290.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

宏定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define ASCII\_CHAR(x) ('0' + (x))

#define DLCD\_DISPLAY\_DEFAULT '0'

#define GET\_DOWNLOAD\_MAP\_IDX(x) ((x) & 0x3F)

#define CLR\_BIT(x, bit) ((x) & (~(bit)))

#define SET\_BIT(x, bit) ((x) | (bit))

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Digital LCD 可配置区域

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define DIGIT\_LCD\_SIZE\_DEFAULT ZLG7290\_DLCD\_NUM /\* 最多4个数码管字节显示 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 数码管字符译码表，向ZLG7290芯片写入索引值即可显示对应字符

0x1F表示无显示

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define ZLG7290\_DOWNLOAD\_MAX 32

UCHAR \_G\_ucDownloadMap[ZLG7290\_DOWNLOAD\_MAX] = {

'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',

'A', 'b', 'C', 'd', 'E', 'F', 'G', 'H', 'i', 'J',

'L', 'o', 'p', 'q', 'r', 't', 'U', 'y', 'c', 'h',

'T', CMD0\_NO\_DISPLAY

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: digitLcdReadChar

\*\* 功能描述: 读取第 uiIndex 个数码管的值

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

UCHAR digitLcdReadChar(PVOID pHwPrivate, UINT uiIndex)

{

PZLG7290\_DEV pCtrl = (PZLG7290\_DEV) pHwPrivate;

UINT uiDownloadIdx;

if (uiIndex >= ZLG7290\_DLCD\_NUM) {

\_ErrorHandle(ERROR\_IOS\_DRIVER\_NOT\_SUP);

printk(KERN\_ERR "digitLcdReadChar(): invalid parameters index: %d.\n", uiIndex);

return PX\_ERROR;

}

/\* 获取下载表里面的索引 \*/

uiDownloadIdx = GET\_DOWNLOAD\_MAP\_IDX(pCtrl->ZLG7290\_ucCmdBuf[uiIndex]);

return \_G\_ucDownloadMap[uiDownloadIdx];

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: digitLcdReadDot

\*\* 功能描述: 读取第 uiIndex 个点的状态

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

UCHAR digitLcdReadDot(PVOID pHwPrivate, UINT uiIndex)

{

PZLG7290\_DEV pCtrl = (PZLG7290\_DEV) pHwPrivate;

if (uiIndex >= ZLG7290\_DLCD\_NUM) {

\_ErrorHandle(ERROR\_IOS\_DRIVER\_NOT\_SUP);

printk(KERN\_ERR "digitLcdReadChar(): invalid parameters index: %d.\n", uiIndex);

return PX\_ERROR;

}

if ((pCtrl->ZLG7290\_ucCmdBuf[uiIndex] & CMD1\_DPRAM\_DP\_BIT) == 0) {

return ASCII\_CHAR(0);

}

else {

return ASCII\_CHAR(1);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: \_\_findCharInMap

\*\* 功能描述: 在下载字符表里面查找要写入的字符

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static UINT \_\_findCharInMap(UCHAR ucValue)

{

UINT i;

for (i=0; i < ZLG7290\_DOWNLOAD\_MAX; i++) {

if (\_G\_ucDownloadMap[i] == ucValue) {

return i;

}

}

return CMD0\_NO\_DISPLAY;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: digitLcdWriteChar

\*\* 功能描述: 写入第 uiIndex 个数码管

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT digitLcdWriteChar(PVOID pHwPrivate, UINT uiIndex, UCHAR ucValue)

{

PZLG7290\_DEV pCtrl = (PZLG7290\_DEV) pHwPrivate;

INT iDownloadIndex;

PUCHAR pucCmdBuf;

INTREG iregFlag;

if (uiIndex >= ZLG7290\_DLCD\_NUM) {

return PX\_ERROR;

}

/\*

\* 1. 在下载表里面找到字符对应索引

\* 2. 将索引写入到ZLG7290

\*/

iDownloadIndex = \_\_findCharInMap(ucValue);

LW\_SPIN\_LOCK\_QUICK(&pCtrl->DLCD\_lock, &iregFlag);

pucCmdBuf = &pCtrl->ZLG7290\_ucCmdBuf[uiIndex];

\*pucCmdBuf = (\*pucCmdBuf & 0xC0) + iDownloadIndex; /\* 更新Index，不修改DP和Flash \*/

LW\_SPIN\_UNLOCK\_QUICK(&pCtrl->DLCD\_lock, iregFlag);

zlg7290WriteCmd(pCtrl, ZLG7290\_CMDBUF, (CMD0\_DOWNLOAD\_FLAG | uiIndex), \*pucCmdBuf);

return ERROR\_NONE;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: digitLcdWriteDot

\*\* 功能描述: 设置第 uiIndex 个点开关

\*\* 输　入 : uiIndex - 数码管位置；ucValue - 设置的状态('0' 或者 '1')

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT digitLcdWriteDot(PVOID pHwPrivate, UINT uiIndex, UCHAR ucValue)

{

PZLG7290\_DEV pCtrl = (PZLG7290\_DEV) pHwPrivate;

INT iDownloadIndex;

INTREG iregFlag;

if (uiIndex >= ZLG7290\_DLCD\_NUM) {

return PX\_ERROR;

}

/\*

\* 1. 在下载表里面找到字符对应索引

\* 2. 将索引写入到ZLG7290

\*/

LW\_SPIN\_LOCK\_QUICK(&pCtrl->DLCD\_lock, &iregFlag);

iDownloadIndex = pCtrl->ZLG7290\_ucCmdBuf[uiIndex];

if (ucValue == ASCII\_CHAR(0)) {

iDownloadIndex = CLR\_BIT(iDownloadIndex, CMD1\_DPRAM\_DP\_BIT);

}

else {

iDownloadIndex = SET\_BIT(iDownloadIndex, CMD1\_DPRAM\_DP\_BIT);

}

pCtrl->ZLG7290\_ucCmdBuf[uiIndex] = iDownloadIndex;

LW\_SPIN\_UNLOCK\_QUICK(&pCtrl->DLCD\_lock, iregFlag);

zlg7290WriteCmd(pCtrl, ZLG7290\_CMDBUF,

(CMD0\_DOWNLOAD\_FLAG | uiIndex),

pCtrl->ZLG7290\_ucCmdBuf[uiIndex]);

return ERROR\_NONE;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: dlcdInit

\*\* 功能描述: 初始化 数码管功能

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT dlcdInit(PVOID pHwPrivate)

{

INT i;

PZLG7290\_DEV pCtrl = (PZLG7290\_DEV)pHwPrivate;

LW\_SPIN\_INIT(&pCtrl->DLCD\_lock); /\* 初始化自旋锁 \*/

memset(pCtrl->ZLG7290\_ucCmdBuf, DLCD\_DISPLAY\_DEFAULT, sizeof(pCtrl->ZLG7290\_ucCmdBuf));

for (i=0; i<ZLG7290\_DLCD\_NUM; i++) {

digitLcdWriteChar(pCtrl, i, DLCD\_DISPLAY\_DEFAULT); /\* 默认显示0 \*/

}

return ZLG7290\_DLCD\_NUM;

}

在程序清单 11.2中使用到的zlg7290WriteCmd等函数，在文件zlg7290.c中定义并实现。函数具体实现如程序清单 11.3所示

程序清单 11.3 zlg7290.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ZLG 7290芯片驱动（数码管及阵列按键功能）

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define \_\_SYLIXOS\_KERNEL

#include "config.h"

#include "SylixOS.h"

#include "string.h"

#include "sys/gpiofd.h"

#include "zlg7290.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

可配置区域

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define ZLG7290\_I2C\_ADDR 0x38

#define ZLG7290\_I2C\_BUS "/bus/i2c/1"

#define ZLG7290\_I2C\_DEVNAME "/dev/zlg7290"

#define ZLG7290\_IRQ IMX6Q\_GPIO\_NUMR(7,12)

#define ZLG7290\_RST IMX6Q\_GPIO\_NUMR(7,13)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

宏定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define ZLG7290\_I2C\_BUF\_MAX 8

#define CLR\_BIT(x, bit) ((x) & (~(bit)))

#define SET\_BIT(x, bit) ((x) | (bit))

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

全局变量

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static ZLG7290\_DEV \_G\_zlg7290[1];

static PZLG7290\_DEV \_G\_pCtrl = NULL;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: zlg7290ReadReg

\*\* 功能描述: isl1208 寄存器读函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT zlg7290ReadReg(PZLG7290\_DEV pCtrl, UINT8 ucReg, UINT8 \*ucBuf, UINT uiLen)

{

INT iError;

LW\_I2C\_MESSAGE i2cMsgs[2] = {

{

.I2CMSG\_usAddr = pCtrl->ZLG7290\_pI2cDev->I2CDEV\_usAddr,

.I2CMSG\_usFlag = 0,

.I2CMSG\_usLen = 1,

.I2CMSG\_pucBuffer = &ucReg,

}, {

.I2CMSG\_usAddr = pCtrl->ZLG7290\_pI2cDev->I2CDEV\_usAddr,

.I2CMSG\_usFlag = LW\_I2C\_M\_RD,

.I2CMSG\_usLen = uiLen,

.I2CMSG\_pucBuffer = ucBuf,

}

};

API\_TimeMSleep(1); /\* 避免连续读写太快 \*/

iError = API\_I2cDeviceTransfer(pCtrl->ZLG7290\_pI2cDev, i2cMsgs, 2);

if (iError < 0) {

printk(KERN\_ERR "zlg7290ReadReg(): failed to i2c transfer!\n");

return (PX\_ERROR);

}

return (ERROR\_NONE);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: zlg7290Write

\*\* 功能描述: zlg7290 寄存器写函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT zlg7290Write(PZLG7290\_DEV pCtrl, UINT8 ucReg, UINT8 \*ucBuf, UINT uiLen)

{

INT iError;

UINT8 i2cBuf[ZLG7290\_I2C\_BUF\_MAX];

LW\_I2C\_MESSAGE i2cMsgs[1] = {

{

.I2CMSG\_usAddr = pCtrl->ZLG7290\_pI2cDev->I2CDEV\_usAddr,

.I2CMSG\_usFlag = 0,

.I2CMSG\_usLen = uiLen + 1,

.I2CMSG\_pucBuffer = i2cBuf,

},

};

i2cBuf[0] = ucReg;

memcpy(&i2cBuf[1], &ucBuf[0], uiLen);

API\_TimeMSleep(1); /\* 避免连续读写太快 \*/

iError = API\_I2cDeviceTransfer(pCtrl->ZLG7290\_pI2cDev, i2cMsgs, 1);

if (iError < 0) {

printk(KERN\_ERR "zlg7290WriteReg(): failed to i2c transfer!\n");

return (PX\_ERROR);

}

return (ERROR\_NONE);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: zlg7290WriteCmd

\*\* 功能描述: zlg7290 寄存器写函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT zlg7290WriteCmd(PZLG7290\_DEV pCtrl, UINT8 ucReg, UINT8 ucCmd1, UINT8 ucCmd2)

{

INT iError;

UCHAR cmdBuf[2];

cmdBuf[0] = ucCmd1;

cmdBuf[1] = ucCmd2;

iError = zlg7290Write(pCtrl, ucReg, cmdBuf, sizeof(cmdBuf));

if (iError < 0) {

printk(KERN\_ERR "zlg7290WriteCmd(): failed to write cmd: %x:%x!\n",

ucCmd1, ucCmd2);

return (PX\_ERROR);

}

return (ERROR\_NONE);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: \_\_zlg7290IrqInit

\*\* 功能描述: 初始化设备IRQ

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static INT \_\_zlg7290IrqInit(PZLG7290\_DEV pCtrl, UINT uiIrq)

{

INT iError;

/\*

\* 设置中断引脚位下降沿触发

\*/

iError = API\_GpioRequestOne(uiIrq, GPIO\_FLAG\_IN | GPIO\_FLAG\_TRIG\_FALL, "zlg7290\_eint");

if (iError != ERROR\_NONE) {

printk(KERN\_ERR "\_\_zlg7290IrqInit(): failed to request gpio!\n");

goto \_\_error\_handle;

}

iError = API\_GpioSetupIrq(uiIrq, LW\_FALSE, 0);

if (iError == PX\_ERROR) {

printk(KERN\_ERR "\_\_zlg7290IrqInit(): failed to setup gpio irq!\n");

goto \_\_error\_handle;

}

pCtrl->ZLG7290\_uiIrqPin = uiIrq; /\* 记录当前中断管脚号 \*/

pCtrl->ZLG7290\_iVector = iError; /\* 记录当前中断向量 \*/

\_\_error\_handle:

return iError;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: \_\_zlg7290HwInit

\*\* 功能描述: 初始化 ZLG7290 芯片硬件寄存器

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static INT \_\_zlg7290HwInit (PZLG7290\_DEV pCtrl, UINT uiResetPin)

{

INT iError;

/\*

\* 设置中断引脚位下降沿触发

\*/

iError = API\_GpioRequestOne(uiResetPin, GPIO\_FLAG\_OUT\_INIT\_HIGH, "zlg7290\_reset");

if (iError != ERROR\_NONE) {

printk(KERN\_ERR "zlg7290HwInit(): failed to request gpio: %d \n", uiResetPin);

goto \_\_error\_handle;

}

API\_GpioSetValue(uiResetPin, 0);

API\_TimeMSleep(20); /\* 复位20ms \*/

API\_GpioSetValue(uiResetPin, 1);

API\_TimeMSleep(100); /\* 等待100ms \*/

pCtrl->ZLG7290\_uiResetPin = uiResetPin; /\* 记录当前复位管脚号 \*/

\_\_error\_handle:

return iError;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 函数名称: \_\_zlg7290I2cDevInit

\*\* 功能描述: 初始化 I2C 设备，用于和zlg7290通信

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static PLW\_I2C\_DEVICE \_\_zlg7290I2cDevInit(PZLG7290\_DEV pCtrl, PCHAR pI2cBusName)

{

if (pCtrl->ZLG7290\_pI2cDev == NULL) {

pCtrl->ZLG7290\_pI2cDev = API\_I2cDeviceCreate(pI2cBusName,

ZLG7290\_I2C\_DEVNAME,

ZLG7290\_I2C\_ADDR, /\* 芯片的设备地址 \*/

0);

}

return pCtrl->ZLG7290\_pI2cDev;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: zlg7290DevCreate

\*\* 功能描述: 初始化 ZLG7290 芯片，建立对应I2C总线设备

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

PZLG7290\_DEV zlg7290DevCreate (PCHAR pI2cBusName, UINT uiIrqPin, UINT uiResetPin)

{

if (\_G\_pCtrl != NULL) { /\* 可扩展成支持多设备 \*/

printk(KERN\_ERR "zlg7290DevCreate(): device already exist : %s\n", pI2cBusName);

return (NULL);

}

\_G\_pCtrl = &\_G\_zlg7290[0];

memset(\_G\_pCtrl, 0, sizeof(ZLG7290\_DEV));

if (\_\_zlg7290I2cDevInit(\_G\_pCtrl, pI2cBusName) == NULL) {

\_ErrorHandle(ERROR\_IO\_NO\_DEVICE\_NAME\_IN\_PATH);

printk(KERN\_ERR "zlg7290Init(): can not init I2C device\n");

return (NULL);

}

if (\_\_zlg7290IrqInit(\_G\_pCtrl, uiIrqPin) == PX\_ERROR) {

\_ErrorHandle(ERROR\_IO\_NO\_DEVICE\_NAME\_IN\_PATH);

printk(KERN\_ERR "zlg7290Init(): can not init IRQ: %d\n", uiIrqPin);

return (NULL);

}

\_\_zlg7290HwInit(\_G\_pCtrl, uiResetPin);

return \_G\_pCtrl;

}

其中zlg7290DevCreate函数在zlg7290\_module.c文件中的module\_init函数中调用。内核模块驱动函数程序清单11.4所示。

程序清单 11.4 zlg7290\_module.c

#define \_\_SYLIXOS\_KERNEL

#include <SylixOS.h>

#include <module.h>

#include "zlg7290.h"

#include "digitlcd.h"

#include "arraykey.h"

#define ZLG7290\_I2C\_BUS "/bus/i2c/1"

#define IMX6Q\_GPIO\_NUMR(bank, gpio) (32 \* (bank - 1) + (gpio))

#define ZLG7290\_IRQ IMX6Q\_GPIO\_NUMR(7,12)

#define ZLG7290\_RST IMX6Q\_GPIO\_NUMR(7,13)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 函数名称: module\_init

\*\* 功能描述: 模块安装函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int module\_init (void)

{

INT iRet;

ZLG7290\_DEV \*pzlg7290;

iRet = digitLcdDrv();

if (iRet != ERROR\_NONE) {

return (PX\_ERROR);

}

iRet = arrayKeyDrv();

if (iRet != ERROR\_NONE) {

return (PX\_ERROR);

}

pzlg7290 = zlg7290DevCreate(ZLG7290\_I2C\_BUS, ZLG7290\_IRQ, ZLG7290\_RST);

digitLcdDevCreate("/dev/digitLcd", pzlg7290);

// arrayKeyDevCreate("/dev/arrayKey", pzlg7290); //矩阵按键设备创建

return (ERROR\_NONE);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: zlg7290DevRemove

\*\* 功能描述: 删除zlg7290 数码管和矩阵按键设备

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void zlg7290DevRemove (void)

{

digitlcdDevRemove ("/dev/digitLcd");

// arrayKeyDevRemove ("/dev/arrayKey"); //矩阵按键设备删除

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 函数名称: module\_exit

\*\* 功能描述: 模块退出函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void module\_exit (void)

{

zlg7290DevRemove();

}

**ZLG7290驱动使用**

zlg7290内核模块驱动实现后，可以使用标准的文件IO函数对zlg7290进行操作，如程序清单 11.5所示。

程序清单 11.5 digitlcd\_test.c

#include <stdio.h>

#include <fcntl.h>

#include <string.h>

#include <pthread.h>

#define DIGITLCD\_DEVICE "/dev/digitLcd"

#define DOWNLOAD\_MAX 32

#define DLCD\_BUF\_MAX 16

#define DLCD\_NUM\_MAX 4

#define DLCD\_CHAR\_TAG "CHAR-"

#define DLCD\_DOT\_TAG "DOT-"

static UCHAR \_G\_ucDownloadMap[DOWNLOAD\_MAX] = {

'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',

'A', 'b', 'C', 'd', 'E', 'F', 'G', 'H', 'i', 'J',

'L', 'o', 'p', 'q', 'r', 't', 'U', 'y', 'c', 'h',

'T', 0xFF

};

void show\_char(int fd)

{

int ret;

int i = 0;

char \*pbuf;

char buf[DLCD\_BUF\_MAX];

for (i=0; i < DOWNLOAD\_MAX; i++) {

strcpy(buf, DLCD\_CHAR\_TAG);

pbuf = buf + strlen(DLCD\_CHAR\_TAG);

memset(pbuf, \_G\_ucDownloadMap[i], DLCD\_NUM\_MAX);

ret = write(fd, buf, (strlen(DLCD\_CHAR\_TAG) + DLCD\_NUM\_MAX));

if (ret > 0) {

printf("Show key: %c \n", \_G\_ucDownloadMap[i]);

}

else {

printf("write char error\n");

return;

}

sleep(1);

}

}

void show\_dot(int fd)

{

int ret;

int i = 0;

char \*pbuf;

char buf[DLCD\_BUF\_MAX];

memset(buf, 0, sizeof(buf));

for (i=0; i < DLCD\_NUM\_MAX; i++) {

strcpy(buf, DLCD\_DOT\_TAG);

pbuf = buf + strlen(DLCD\_DOT\_TAG);

memset(pbuf, '0', DLCD\_NUM\_MAX);

pbuf[i%4] = '1'; /\* Show this dot \*/ ret = write(fd, buf, (strlen(DLCD\_DOT\_TAG) + DLCD\_NUM\_MAX));

if (ret > 0) {

printf("Show dot: %s \n", pbuf);

}

else {

printf("write dot error\n");

return;

}

sleep(1);

}

}

void clear\_dot(int fd)

{

char \*pbuf;

char buf[DLCD\_BUF\_MAX];

strcpy(buf, DLCD\_DOT\_TAG);

pbuf = buf + strlen(DLCD\_DOT\_TAG);

memset(pbuf, '0', DLCD\_NUM\_MAX);

write(fd, buf, (strlen(DLCD\_DOT\_TAG) + DLCD\_NUM\_MAX));

}

int main (int argc, char \*argv[])

{

int fd;

fd = open(DIGITLCD\_DEVICE, O\_RDWR);

if (fd < 0) {

printf("can NOT open %s, error: %s\n", DIGITLCD\_DEVICE, strerror(errno));

return -1;

}

show\_char(fd);

show\_dot(fd);

clear\_dot(fd);

close(fd);

return 0;

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译，将生成的工程文件上传到验证平台，加载zlg7290\_dev驱动模块，查看、/dev目录会有digitLcd设备；

[root@sylixos\_station:/root]# modulereg /lib/modules/zlg7290\_dev.ko

module /lib/modules/zlg7290\_dev.ko register ok, handle: 0x12031030

[root@sylixos\_station:/root]# ls /dev/

log socket netevent arrayKey digitLcd

...

blk input pipe pty

1. 执行diditlcd\_test程序，串口终端会有以下提示。

[root@sylixos\_station:/root]# cd /apps/

[root@sylixos\_station:/apps]# ./digitlcd\_test/digitlcd\_test

Show key: 0

Show key: 1

Show key: 2

Show key: 3

Show key: 4

Show key: 5

Show key: 6

Show key: 7

Show key: 8

...

同时数码管会显示相应的字符。

### 扩展实验

实现设备的矩阵键盘驱动。

## 实验三 EEPROM驱动编写实验

### 实验目的

* 了解EEPROM的读写流程
* 掌握使用I2C总线读写EEPROM的程序设计方法

### 实验内容

* 阅读实验原理，了解EEPROM的读写流程和I2C总线的使用方法
* 根据实验步骤完成对EEPROM的读写程序设计和验证

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

EEPROM介绍

AT24C04是一4Kbits（512字节）容量串行EEPROM，AT24C04有16字节页写缓冲器和专门的写保护。该器件通过I2C总线接口进行操作，AT24C04引脚图如图 11.10。

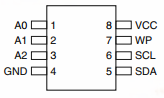


图 11.10 AT24C04引脚

表 11.4说明了AT24C04各引脚的功能。

表 11.5 AT24C04引脚功能

|  |  |
| --- | --- |
| 引脚名 | 功能 |
| A0到A2 | 地址 |
| SDA | 串行数据/地址 |
| SCL | 串行时钟输入 |
| WP | 写保护 |
| VCC/GND | 电源/接地 |

SDA：AT24C04双向串行数据管脚用于数据的发送接收；

SCL：AT24C04串行时钟输入管脚；

A0～A2：用于设置器件地址，悬空时引脚状态为0；

WP：WP = 1：器件为只读状态。WP = 0：器件可读可写。

EEPROM设备地址

对EEPROM读写前，需先发送设备地址以选择芯片，其中首部四比特的“1010”为固定值，A0到A2可以设置EEPROM器件地址，I2C总线上有多个EERPOM器件时，可通过设置不同地址进行读写，最后一比特为读写操作位，1表示读操作，0表示写操作，如下所示。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 0 | A2 | A1 | A0 | R/W |
| MSB LSB | | | | | | | |

注：对AT24Cxx系列器件的地址，不同容量器件地址定义有差异，详细说明需要查看说明书

EEPROM写操作

AT24C04的写操作有两种方式，分别是按字节写和按页写，按字节写时通常在向EEPROM发送设备地址并收到应答信号后，发送写字节地址再次收到ACK后开始写数据，最后发送停止位结束写操作。

写页时EEPROM可一次连续写入整页数据（一页为16字节）。其发地址过程与写字节时完全相同。不同的是，当写完一个数据字节后，处理器发不发停止状态，而是在应答信号后继续写入数据，每一个字节接收完毕后，EEPROM都返回一个ACK，一直到写完整页。如果页写时写入数超出该物理页边界，则超出数据将重新写入页首地址覆盖之前所写数据。

EEPROM读操作

读操作有当前地址读、随机读、多字节连续读三种方式。其发送的地址字节与写操作相同，只需把设备地址的最低位改为读。在当前地址读操作方式时无需发送读字节地址，每次只将当前地址所存数据读出，片内地址始终保持自加，直到读完整个EEPROM后又回到0地址。而随机读要先写读字节地址，然后才能读。多字节连续读操作既可以是当前地址读，也可以是随机地址读，每次处理器接收到一字节数据都返回一个ACK，EEPROM接收到此ACK后会自动地址加1，接着输出下一个字节数据，直到处理器返回NO ACK时，读过程结束。

EEPROM驱动

EEPROM驱动的编写同样是实现设备文件操作控制块结构体file\_operations的成员函数，在EEPROM设备驱动中主要实现了eepromOpen、eepromClose、eepromRead、eepromWrite、eepromIoctl函数功能，eepromIoctl函数用来设置EEPROM的读写模式，驱动实现了表 11.5所示几种读写模式。

表 11.6 eeprom驱动读写模式

|  |  |  |
| --- | --- | --- |
| 引脚名 | 功能 | 值 |
| EEPROM\_CTLCMD\_RBYTE | 读字节 | 100 + 1 |
| EEPROM\_CTLCMD\_WBYTE | 写字节 | 100 + 2 |
| EEPROM\_CTLCMD\_RPAGE | 读页 | 100 + 3 |
| EEPROM\_CTLCMD\_WPAGE | 写页 | 100 + 4 |

应用程序可以通过访问标准文件I/O函数来读写EEPROM设备，在读写EEPROM设备前，必须先设置读写模式，然后才能对设备进行读写操作。根据以上说明仔细阅读EEPROM驱动源代码程序清单 11.6，了解其实现的具体方法。

程序清单 11.6 eeprom\_dev.c

#define \_\_SYLIXOS\_KERNEL

#include <SylixOS.h>

#include <module.h>

#include <string.h>

#include "eeprom\_cmd.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 宏定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define EEPROM\_ADDR (0x50) /\* 从设备地址 \*/

#define EEPROM\_I2C\_NAME ("/bus/i2c/1") /\*　i2c 通道号 \*/

#define EEPROM\_DEV\_NAME ("/dev/eeprom") /\* eeprom 设备名 \*/

#define EEPROM\_DBG(fmt, arg...) printk(KERN\_INFO "%s: %d "fmt, \

\_\_func\_\_, \_\_LINE\_\_, ##arg)

#define EEPROM\_PAGE\_SZ (16) /\* 一页是 16字节数 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

eeprom 设备结构

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typedef struct eeprom {

LW\_DEV\_HDR EEP\_devhdr;

PLW\_I2C\_DEVICE EEP\_i2cdev; /\* I2C 设备 \*/

UINT8 EEP\_ucChildAddr; /\* eeprom 设备子地址 \*/

INT EEP\_iCmdWord; /\* 命令 \*/

time\_t EEP\_timeCreate; /\* 设备创建时间 \*/

} EEPROM;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

驱动号

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static INT \_\_G\_eepromDrvNum = -1;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: eepromPageWrite

\*\* 功能描述: 页的方式写 eeprom

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT eepromPageWrite (PLW\_I2C\_DEVICE pi2c, UINT8 ucPageAddr,

UINT8 \*ucPayload, INT iLen)

{

INT iRet;

UINT8 pucTxBuf[EEPROM\_PAGE\_SZ + 1];

if (!pi2c) {

return (PX\_ERROR);

}

if (iLen > EEPROM\_PAGE\_SZ) {

EEPROM\_DBG("write page toooo long.\n");

return (PX\_ERROR);

}

pucTxBuf[0] = ucPageAddr;

lib\_memcpy(pucTxBuf + 1, ucPayload, iLen);

LW\_I2C\_MESSAGE msgs[1] = {

{

.I2CMSG\_usAddr = pi2c->I2CDEV\_usAddr,

.I2CMSG\_usFlag = 0,

.I2CMSG\_usLen = EEPROM\_PAGE\_SZ + 1,

.I2CMSG\_pucBuffer = pucTxBuf,

}

};

iRet = API\_I2cDeviceTransfer(pi2c, msgs, 1);

if (iRet < 0) {

\_DebugFormat(\_\_PRINTMESSAGE\_LEVEL, "i2c msg write error\n\r");

return (PX\_ERROR);

}

// iRet = API\_I2cDeviceCtl(pi2c, I2C\_GET\_SNDMSGNR\_CMD, 0);

return (iRet);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: eepromPageRead

\*\* 功能描述: 页的方式读 eeprom

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT eepromPageRead (PLW\_I2C\_DEVICE pi2c, UINT8 ucPageAddr,

UINT8 \*ucBuf, INT iLen)

{

INT iRet;

if (!pi2c) {

EEPROM\_DBG("Invalid eeprom device.\n");

return (PX\_ERROR);

}

if (iLen > EEPROM\_PAGE\_SZ) {

EEPROM\_DBG("Read length great a page[%d]", EEPROM\_PAGE\_SZ);

return (PX\_ERROR);

}

LW\_I2C\_MESSAGE msgs[2] = {

{

.I2CMSG\_usAddr = pi2c->I2CDEV\_usAddr,

.I2CMSG\_usFlag = 0,

.I2CMSG\_usLen = 1,

.I2CMSG\_pucBuffer = &ucPageAddr,

},

{

.I2CMSG\_usAddr = pi2c->I2CDEV\_usAddr,

.I2CMSG\_usFlag = LW\_I2C\_M\_RD,

.I2CMSG\_usLen = EEPROM\_PAGE\_SZ,

.I2CMSG\_pucBuffer = ucBuf,

}

};

iRet = API\_I2cDeviceTransfer(pi2c, msgs, 2);

if (iRet < 0) {

EEPROM\_DBG("I2c msg PageRead error: %d\n", iRet);

return (PX\_ERROR);

}

return (iRet);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: eepromByteWrite

\*\* 功能描述: 字节的方式写 eeprom

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT eepromByteWrite (PLW\_I2C\_DEVICE pi2c, UINT8 ucByteAddr, UINT8 ucData)

{

INT iRet;

UINT8 pucRxBuf[2];

if (!pi2c) {

return (PX\_ERROR);

}

pucRxBuf[0] = ucByteAddr;

pucRxBuf[1] = ucData;

LW\_I2C\_MESSAGE msgs[] = {

{

.I2CMSG\_usAddr = pi2c->I2CDEV\_usAddr,

.I2CMSG\_usFlag = 0,

.I2CMSG\_usLen = 2,

.I2CMSG\_pucBuffer = pucRxBuf,

}

};

iRet = API\_I2cDeviceTransfer(pi2c, msgs, 1);

if (iRet < 0) {

EEPROM\_DBG("msg i2c read error: %d\n", iRet);

return (PX\_ERROR);

}

return (iRet - 1);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: eepromByteRead

\*\* 功能描述: 字节的方式读 eeprom

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT eepromByteRead (PLW\_I2C\_DEVICE pi2c, UINT8 ucByteAddr, UINT8 \*pucData)

{

INT iRet;

if (!pi2c) {

return (PX\_ERROR);

}

LW\_I2C\_MESSAGE msgs[] = {

{

.I2CMSG\_usAddr = pi2c->I2CDEV\_usAddr,

.I2CMSG\_usFlag = 0,

.I2CMSG\_usLen = 1,

.I2CMSG\_pucBuffer = &ucByteAddr,

}, {

.I2CMSG\_usAddr = pi2c->I2CDEV\_usAddr,

.I2CMSG\_usFlag = LW\_I2C\_M\_RD,

.I2CMSG\_usLen = 1,

.I2CMSG\_pucBuffer = pucData,

}

};

iRet = API\_I2cDeviceTransfer(pi2c, msgs, 2);

if (iRet < 0) {

EEPROM\_DBG("I2c msg read error: %d\n", iRet);

return (PX\_ERROR);

}

return (iRet - 1);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: eepromOpen

\*\* 功能描述: 驱动 open 函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LONG eepromOpen (EEPROM \*peeprom,

PCHAR pcName,

INT iFlags,

INT iMode)

{

if (!peeprom) {

\_ErrorHandle(EINVAL);

return (PX\_ERROR);

}

if (LW\_DEV\_INC\_USE\_COUNT(&peeprom->EEP\_devhdr) == 1) {

peeprom->EEP\_i2cdev = API\_I2cDeviceCreate(EEPROM\_I2C\_NAME,

pcName,

EEPROM\_ADDR,

0);

if (peeprom->EEP\_i2cdev == LW\_NULL) {

return (PX\_ERROR);

}

}

return ((LONG)peeprom);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: eepromClose

\*\* 功能描述: 驱动 close 函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT eepromClose (EEPROM \*peeprom)

{

if (!peeprom) {

\_ErrorHandle(EINVAL);

return (PX\_ERROR);

}

if (LW\_DEV\_DEC\_USE\_COUNT(&peeprom->EEP\_devhdr) == 0) {

API\_I2cDeviceDelete(peeprom->EEP\_i2cdev);

}

return (ERROR\_NONE);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 函数名称: eepromRead

\*\* 功能描述: 驱动 read 函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ssize\_t eepromRead (EEPROM \*peeprom, PVOID pvBuf, size\_t stLen)

{

INT iLen;

if (!peeprom) {

\_ErrorHandle(EINVAL);

return (PX\_ERROR);

}

if (peeprom->EEP\_iCmdWord == EEPROM\_CTLCMD\_RBYTE) {

iLen = eepromByteRead(peeprom->EEP\_i2cdev, peeprom->EEP\_ucChildAddr, pvBuf);

} else if (peeprom->EEP\_iCmdWord == EEPROM\_CTLCMD\_RPAGE) {

iLen = eepromPageRead(peeprom->EEP\_i2cdev, peeprom->EEP\_ucChildAddr, pvBuf, stLen);

} else {

iLen = 0;

}

return (iLen);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: eepromWrite

\*\* 功能描述: 驱动 write 函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ssize\_t eepromWrite (EEPROM \*peeprom, PVOID pvBuf, size\_t stLen)

{

INT iLen;

if (!peeprom) {

\_ErrorHandle(EINVAL);

return (PX\_ERROR);

}

if (peeprom->EEP\_iCmdWord == EEPROM\_CTLCMD\_WBYTE) {

iLen = eepromByteWrite(peeprom->EEP\_i2cdev, peeprom->EEP\_ucChildAddr, \*(UINT8 \*)pvBuf);

} else if (peeprom->EEP\_iCmdWord == EEPROM\_CTLCMD\_WPAGE) {

iLen = eepromPageWrite(peeprom->EEP\_i2cdev, peeprom->EEP\_ucChildAddr, pvBuf, stLen);

} else {

iLen = 0;

}

return (iLen);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: eepromIoctl

\*\* 功能描述: 驱动 ioctl 函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT eepromIoctl (EEPROM \*peeprom,

INT iCmd,

LONG lArg)

{

INT iError = ERROR\_NONE;

struct stat \*pstatGet;

switch (iCmd) {

case EEPROM\_CTLCMD\_RBYTE:

peeprom->EEP\_ucChildAddr = (UINT8)lArg;

peeprom->EEP\_iCmdWord = EEPROM\_CTLCMD\_RBYTE;

break;

case EEPROM\_CTLCMD\_WBYTE:

peeprom->EEP\_ucChildAddr = (UINT8)lArg;

peeprom->EEP\_iCmdWord = EEPROM\_CTLCMD\_WBYTE;

break;

case EEPROM\_CTLCMD\_RPAGE:

peeprom->EEP\_ucChildAddr = (UINT8)lArg;

peeprom->EEP\_iCmdWord = EEPROM\_CTLCMD\_RPAGE;

break;

case EEPROM\_CTLCMD\_WPAGE:

peeprom->EEP\_ucChildAddr = (UINT8)lArg;

peeprom->EEP\_iCmdWord = EEPROM\_CTLCMD\_WPAGE;

break;

case FIOFSTATGET:

pstatGet = (struct stat \*)lArg;

if (pstatGet) {

pstatGet->st\_dev = (dev\_t)peeprom;

pstatGet->st\_ino = (ino\_t)0; /\* 相当于唯一节点 \*/

pstatGet->st\_mode = 0444 | S\_IFCHR;

pstatGet->st\_nlink = 1;

pstatGet->st\_uid = 0;

pstatGet->st\_gid = 0;

pstatGet->st\_rdev = 1;

pstatGet->st\_size = 0;

pstatGet->st\_blksize = 0;

pstatGet->st\_blocks = 0;

pstatGet->st\_atime = peeprom->EEP\_timeCreate;

pstatGet->st\_mtime = peeprom->EEP\_timeCreate;

pstatGet->st\_ctime = peeprom->EEP\_timeCreate;

} else {

errno = EINVAL;

iError = (PX\_ERROR);

}

break;

default:

errno = ENOSYS;

iError = PX\_ERROR;

break;

}

return (iError);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: eepromDrvInstall

\*\* 功能描述: 驱动安装函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT eepromDrvInstall (VOID)

{

static struct file\_operations fileop;

if (\_\_G\_eepromDrvNum > 0) {

return (ERROR\_NONE);

}

fileop.owner = THIS\_MODULE;

fileop.fo\_open = eepromOpen;

fileop.fo\_close = eepromClose;

fileop.fo\_read = eepromRead;

fileop.fo\_write = eepromWrite;

fileop.fo\_ioctl = eepromIoctl;

\_\_G\_eepromDrvNum = iosDrvInstallEx2(&fileop, LW\_DRV\_TYPE\_ORIG);

DRIVER\_LICENSE(\_\_G\_eepromDrvNum, "Dual BSD/GPL->Ver 1.0");

DRIVER\_AUTHOR(\_\_G\_eepromDrvNum, "Zhao.JianPeng");

DRIVER\_DESCRIPTION(\_\_G\_eepromDrvNum, "eeprom driver.");

return ((\_\_G\_eepromDrvNum > 0) ? (ERROR\_NONE) : (PX\_ERROR));

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 函数名称: eepromDevCreate

\*\* 功能描述: 设备安装函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT eepromDevCreate (PCHAR pcName)

{

EEPROM \*peeprom;

INT iRet;

if (\_\_G\_eepromDrvNum < 0) {

\_ErrorHandle(EINVAL);

return (PX\_ERROR);

}

peeprom = \_\_SHEAP\_ALLOC(sizeof(EEPROM));

if (peeprom == LW\_NULL) {

\_ErrorHandle(ERROR\_SYSTEM\_LOW\_MEMORY);

return (PX\_ERROR);

}

lib\_bzero(peeprom, sizeof(EEPROM));

iRet = iosDevAddEx(&peeprom->EEP\_devhdr,

pcName,

\_\_G\_eepromDrvNum,

DT\_CHR);

if (iRet) {

\_\_SHEAP\_FREE(peeprom);

return (PX\_ERROR);

}

peeprom->EEP\_timeCreate = time(LW\_NULL);

return (ERROR\_NONE);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 函数名称: eepromDevRemove

\*\* 功能描述: 删除eeprom设备

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

VOID eepromDevRemove (VOID)

{

PLW\_DEV\_HDR pdevhdr;

PCHAR pcTail;

pdevhdr = iosDevFind(EEPROM\_DEV\_NAME, &pcTail);

if (!pdevhdr) {

return;

}

iosDevFileAbnormal(pdevhdr);

iosDevDelete(pdevhdr);

iosDrvRemove(\_\_G\_eepromDrvNum, LW\_FALSE);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 函数名称: module\_init

\*\* 功能描述: 模块安装函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT module\_init (VOID)

{

INT iRet;

iRet = eepromDrvInstall();

if (iRet != ERROR\_NONE) {

return (PX\_ERROR);

}

eepromDevCreate(EEPROM\_DEV\_NAME);

return (ERROR\_NONE);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: module\_exit

\*\* 功能描述: 模块退出函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

VOID module\_exit (VOID)

{

eepromDevRemove();

}

EEPROM驱动使用

EEPROM驱动装载后，可以使用标准的文件IO函数对EEPROM进行操作，如程序清单 11.3所示。

程序清单 11.7 eeprom\_drv\_use

#include <stdio.h>

#include <unistd.h>

#include "../../eeprom\_drv/src/eeprom\_cmd.h" /\* 驱动支持的命令头文件 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 宏定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define EEPROM\_CHILD\_ADDR (0x00)

#define EEPROM\_ONE\_PAGE (16)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: 程序主函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

int fd;

char write\_str[EEPROM\_ONE\_PAGE] = "sylixos eeprom.";

char read\_str[EEPROM\_ONE\_PAGE] = {0};

int ret;

fd = open("/dev/eeprom", O\_RDWR);

if (fd < 0) {

fprintf(stderr, "open %s failed.\n", "/dev/eeprom");

return (-1);

}

ret = ioctl(fd, EEPROM\_CTLCMD\_WPAGE, EEPROM\_CHILD\_ADDR);

if (ret < 0) {

fprintf(stderr, "ioctl error.\n");

return (-1);

}

ret = write(fd, write\_str, EEPROM\_ONE\_PAGE);

if (ret < 0) {

fprintf(stderr, "write eeprom failed.\n");

return (-1);

}

usleep(2000); /\*延时2ms （等待写入完成），否则导致读出错 \*/

ret = ioctl(fd, EEPROM\_CTLCMD\_RPAGE, EEPROM\_CHILD\_ADDR);

if (ret < 0) {

fprintf(stderr, "ioctl error.\n");

return (-1);

}

ret = read(fd, read\_str, EEPROM\_ONE\_PAGE);

if (ret < 0) {

fprintf(stderr, "read eeprom failed.\n");

return (-1);

}

close(fd);

fprintf(stdout, "read\_str: %s\n", read\_str);

return (0);

}

### 实验步骤

1. 使用RealEvo-IDE打开并编译eeprom\_drv工程，将生成的工程文件上传到开发板，注册内核模块，查看/dev目录会有eeprom设备；

[root@sylixos\_station:/lib/modules]# modulereg eeprom\_dev.ko

module eeprom\_dev.ko register ok, handle : 0x1204ae98

[root@sylixos\_station:/dev]# ls

eeprom log socket netevent backlight

...

pty

1. 使用RealEvo-IDE打开并编译eeprom\_drv\_use工程，将生成的工程文件上传到开发板并执行；

[root@sylixos\_station:/apps/eeprom\_drv\_use]# ./eeprom\_drv\_use

read\_str: sylixos eeprom.

1. 卸载eeprom驱动，并查看/dev目录，这时eeprom设备会消失；

[root@sylixos\_station:/apps/eeprom\_drv\_use]# moduleunreg 0x1204ae98

module /lib/modules/eeprom\_dev.ko unregister ok.

[root@sylixos\_station:/apps/eeprom\_drv\_use]# cd /dev

[root@sylixos\_station:/dev]# ls

log socket netevent backlight fb4

fb3 fb2 fb1 fb0 ttyS3

ttyS2 ttyS1 ttyS0 urandom random

shm rtc hotplug epollfd gpiofd

signalfd hstimerfd timerfd eventfd zero

null blk input pipe pty

1. 再次执行eeprom\_drv\_use程序，会提示设备打开失败。

[root@sylixos\_station:/apps/eeprom\_drv\_use]# ./eeprom\_drv\_use

open /dev/eeprom failed.

### 扩展实验

选择一款常见的I2C设备，模仿EEPROM驱动实现对该设备的驱动。如常见的陀螺仪芯片、加速计芯片等。

# SD卡驱动实验

SD卡是当前应用广泛的大容量可移动存储介质，SD卡控制器也几乎成为了性能稍高的嵌入式处理器的标准配置。SD卡协议本身比较复杂，但由于其标准化程度较高，SD卡相关驱动的移植和使用相对要简单很多。

## 实验一 SD卡驱动编写实验

### 实验目的

* 了解SD卡接口及特点
* 了解SD卡驱动的实现原理和特点
* 掌握编写SD卡驱动的方法

### 实验内容

* 阅读芯片数据手册，了解芯片IMX6Q的SD卡控制器功能和工作流程
* 阅读实验原理，学习SylixOS下SD驱动的编写方法
* SD卡驱动的编译使用

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

**SD协议简介**

SD协议由MMC发展而来，可在软件上完全兼容MMC协议。其定义了一套完整的物理层规范和总线通信协议，并且在协议层支持SPI传输模式，其目的是为了在没有SD控制器的芯片上也能使用SD设备，算是一种低成本解决方案。SD协议主要包含SD Memory（即SD存储卡）和SDIO规范，SDIO与USB类似，作为总线接口可支持WIFI、GPS等设备。

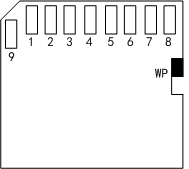


图 12.1 SD接口

表 12.1分别说明了SD各个引脚接口的功能：

表 12.1 SD接口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 接口编号 | 名称 | SD模式功能 | SPI模式功能 |
| 1 | DATA3/CS | 数据线3/卡插拔检测 | SPI从设备片选信号 |
| 2 | CMD/DI | 命令线 | MOSI |
| 3 | VSS1 | 电源地 | 电源地 |
| 4 | VDD | 电源正 | 电源正 |
| 5 | CLK | 时钟 | SCK |
| 6 | VSS2 | 电源地 | 电源地 |
| 7 | DATA0/DO | 数据线0 | MISO |
| 8 | DATA1/IRQ | 数据线1/SDIO中断 | 可作SDIO中断 |
| 9 | DATA2/NC | 数据线2 | 保留 |

从上表可以看出，在SPI模式下，由DATA1充当SDIO中断信号引脚，但SPI控制器本身无法捕捉该中断信号，因此SPI模式下要支持SDIO设备将对驱动程序有更多额外的需求，且不一定能很好地实现，软件设计中通常不考虑在SPI模式下支持SDIO。

**SD物理层定义**

* 传输位宽

在SD模式下，可使用一位（DATA0）或四位(DATA0~DATA3)数据线进行传输，该规定可最大限度地兼容不同的SD设备和SD控制器（两者均有可能仅支持单一的数据位宽）。

在SPI模式下，仅仅允许设备工作在一位数据位宽状态下。

* 传输速率

SD规范主要定义了3种传输速率：

* 低速：400KHZ时钟，通常用于设备初始化阶段；
* 全速：25MHZ时钟，用户设备正常工作阶段；
* 高速：50MHZ时钟，可支持高速设备。

SD规范中明确定义了设备和控制器必须支持低速和全速，高速模式为非必要实现，需要控制器和设备双方均支持才能正常工作。

* 设备类型

SD存储卡容量分为SDSC（最大2GB）、SDHC（最大32GB）、SDXC（最大2TB）。SD存储卡的传输速度和它的容量类型没有必然联系，如果认为SDHC就是所谓的高速卡是一个误区。SD存储卡的速度分为5类，class0并未规定具体的传输速度，由设备自定义，class2最低2MB/s，class4最低4MB/s，class6最低6MB/s，class10最低10MB/s。SDSC和SDHC速度最低为2MB/s，高速卡应该支持50MHZ时钟，速度达到25MB/s。

SDIO卡主要有单一的SDIO卡和组合SDIO卡（即包含SDIO功能和SD存储卡功能的设备）两大类，其传输速度与SD存储卡的定义相同。

* 操作电压

MMC规范定义的工作电压范围为1.65V~3.6V，但SD规范的电压范围为2.7V~3.6V，目前少有SD控制器支持MMC规定的1.65V~1.95V的低电压范围，因此不能支持只能工作在低电压范围的MMC卡。定义一个工作电压范围的目的是为了兼容已经存在的SD控制器和SD设备。

* 传输协议

如果把SD一次总线传输称作一次会话，那么一次会话由命令、应答和数据组成。由于SD是一个主从总线，命令总是由SD控制器发出，随后可能伴随设备对该命令的应答以及双方的数据传输。这三者均伴随CRC数据校验以保证传输的正确性，CRC计算由SD控制器处理。SD存储卡由不同的命令集分别处理设备初始化和设备数据传输等工作，SDIO卡在SD存储卡命令集的基础上，增加了一些额外的命令完成与SDIO设备相关的通信，因此SDIO卡可以看成是SD存储卡的一个扩展功能。

**SPI模式**

SPI模式下的总线协议是SD总线协议的一个子集。由于SPI控制器只能充当SD物理传输层的功能，因此定义了专供SPI模式下使用的命令集，这些命令同时也对应不同的应答格式。此模式下，一次SD总线会话将由多个SPI传输完成，这对软件设计有更多的需求。此外，还需要由软件完成CRC校验的处理，在初始化SD设备时可以关闭CRC功能。

**安全**

* 设备加密。SD存储卡可由特定的命令操作，让卡处于加密状态，此时该卡不能被读或写。设备是否支持加密功能由它是否支持该加密命令决定；
* 设备硬件写保护。SDHC和SDXC存储卡不支持该命令；
* 软件写保护。每一个SD卡上都有一个写保护开关（WP），事实上该开关与SD卡内部没有任何联系，但它提供了实现软件写保护的方法，如图 12.2。

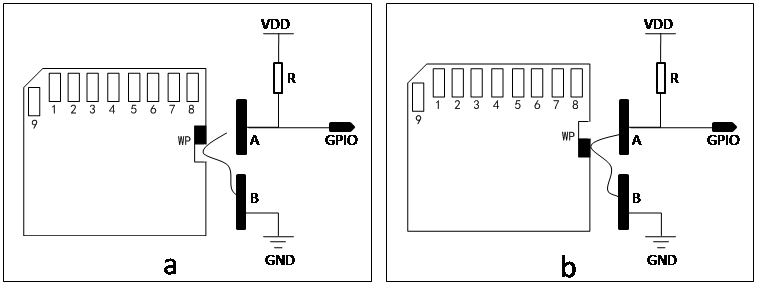


图 12.2 SD卡软件写保护示意图

如图 13.2 所示，写保护开关是一块可以拨动的滑块，当拨到上面时，A、B断开，此时GPIO为高电平；当拨到下面时，滑块将弹片顶起使A、B接通，此时GPIO为低电平。SD卡规定滑块拨到下面时表示写保护打开，因此可以总结为：当读到GPIO为低电平时，SD卡处于写保护状态，软件可根据此信息将SD卡标记只读模式；反之未写保护。

**总线对设备数量的支持**

SD总线为一主多从总线，即一条总线上可以挂接多个SD从设备，它们以不同的地址区分。但SylixOS SD协议实现上，并不支持一条SD总线上存在多个设备的情况，原因如下：

* SD卡接口完全兼容MMC卡，若一个总线上同时存在SD卡和MMC卡，且它们需要不同的电压支持时，不论是硬件还是软件上都难以实现兼容；
* 当一个总线上有多个SDIO设备时，无法区分产生的SDIO中断来自于哪个设备；
* 一个总线上的多个SD设备若需要不同的传输速率，SD规范中没有提供解决方案。SD规范中明确说明，在高速模式下，一个SD控制器只能驱动一个SD设备；
* SD规范没有对一个总线上挂接的设备最大数量进行定义，这使得应用具有不确定性；
* 到目前为止，实际工程应用中没有出现过一个SD总线挂接多个SDIO设备的情况。

注：Linux系统下一个SD总线也同样只能存在一个设备。

**SD总线会话时序**

一个SD总线会话由命令（Commond）、应答（Response）、和数据（Data）三个基本元素构成，SD总线会话可分为以下几类：

* 无应答或无数据

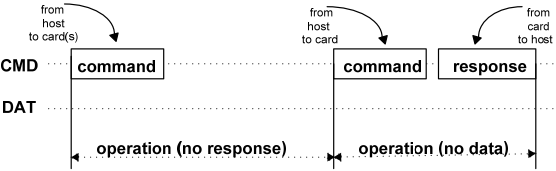


图 12.3 无数据的SD会话

一些SD会话只需控制器向设备发送单个命令即可，例如SD设备复位命令；有一些会伴随设备的应答信息但不涉及数据传输，例如获取设备的状态，读取设备的特殊信息等命令。从图 12.3可以看出，命令和应答均通过CMD信号线传输。

注：上图中并未给出的信息是，命令和应答均会伴随CRC校验。

* 读块数据

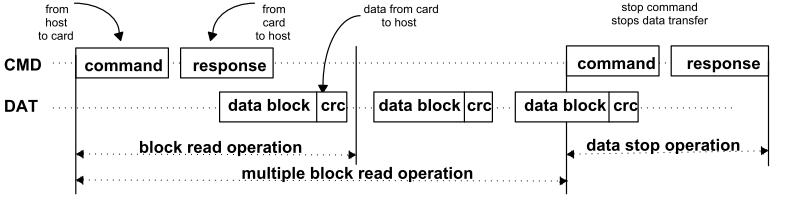


图 12.4 读块数据的SD会话

数据通过DAT信号线传输，分为单块传输和多块传输。上图中为数据读取时序，在单块数据读取时，控制器端首先发送单块数据读取命令，等待设备对此命令的应答，随后读取设备返回的数据，数据也通过CRC校验保证正确性；在多块数据读取时，也只需要向设备发送一个多块数据读取的命令即可，随后连续读取指定数量的数据块，数据传输完成后，需要向设备发送一个数据传输停止的命令。

* 写块数据

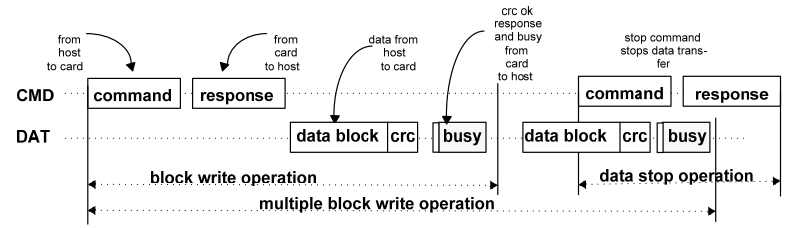


图 12.5 写块数据的SD会话

写数据块的时序与读数据块非常相似，只不过每次向设备发送数据后，都需要等待设备的忙状态。这是因为主机发送完数据后，并不能保证数据已经被写入SD物理介质中。SD自身在向内部物理介质写入数据时，设备会将自己置为忙状态，控制器端需要等待，直到忙状态解除，才能继续发送下一个数据块。同样数据传输完成，也需要发送数据传输停止命令。

**SylixOS SD协议栈总体结构**



图12.6 SylixOS的SD卡协议栈

Host层：硬件控制器抽象层，SD控制器在不同的硬件平台上有不同的实现。所有的控制器驱动都向上（Core层）提供统一的操作接口。SD Stack已经提供了符合SD规范的标准控制器SDHCI驱动，因此控制器驱动的编写相对比较简单。也可使用SPI传输，图 13.6的IF是interface的简写，即SD硬件接口。

Core层：封装了SD和SPI两种传输方式（Xfer为transfer的简写），让Client层只需要关心与具体设备相关的SD协议处理，而不必考虑底层的硬件情况。Core层同时还提供相关工具库供Client使用，sdLib是基于Core Xfer 传输对象封装的SD Memory相关协议操作库。sdioLib与sdLib相同，是针对SDIO设备操作的相关工具库，此库使得SDIO设备驱动的编写更简单。

Client层：实现具体的设备类协议。SD Memory 负责SD Memory 相关的协议处理（如初始化，块读写等），它同时完成与SylixOS块设备相关的接口创建（BLK\_DEV）。SDIO BASE是SDIO设备的基础驱动，它和SD Memory处于同一级，它不会直接去创建实际的SDIO设备， 主要是在完成SDIO的初始化后，使用特定的SDIO驱动创建对应的设备。之所以设计SDIO基础驱动，是为了使驱动框架更加统一，让SDIO设备驱动开发更简洁。

SDM：即SD DRV Manager，SD 驱动管理层，SD驱动包括SD Client驱动和Host驱动，这样两者的信息都由SDM管理维护，以此达到两者完全隔离的目的。

Client驱动通常由SylixOS系统提供，本章主要讲解Host层驱动，即硬件控制器驱动。

**SD总线抽象**

与SPI和I2C类似，SD总线抽象包含总线传输函数集合、SD总线适配器和SD传输消息等相关模块。一个SD总线适配器是负责处理SD数据传输的实体，即图 13.6的SD IF层。

在Base工程的SylixOS/system/device/sd/sdBus.h里定义了lw\_sd\_funcs结构，用以描述所有SD控制器的行为。

typedef struct lw\_sd\_funcs {

INT (\*SDFUNC\_pfuncMasterXfer)(PLW\_SD\_ADAPTER psdadapter,

struct lw\_sd\_device \*psddevice,

PLW\_SD\_MESSAGE psdmsg,

INT iNum);

INT (\*SDFUNC\_pfuncMasterCtl)(PLW\_SD\_ADAPTER psdadapter,

INT iCmd,

LONG lArg);

} LW\_SD\_FUNCS, \*PLW\_SD\_FUNCS;

SDFUNC\_pfuncMasterXfer为控制器处理应用层提交的传输请求，SDFUNC\_pfuncMasterCtl处理应用层提交的与控制器硬件本身相关的控制请求。这些请求包括打开/关闭控制器的电源，设置控制器的数据位宽，设置时钟频率等。SD总线适配器的结构与I2C、SPI相似：

typedef struct lw\_sd\_adapter {

LW\_BUS\_ADAPTER SDADAPTER\_busadapter; /\* 总线节点 \*/

struct lw\_sd\_funcs \*SDADAPTER\_psdfunc; /\* 总线适配器操作函数 \*/

LW\_OBJECT\_HANDLE SDADAPTER\_hBusLock; /\* 总线操作锁 \*/

INT SDADAPTER\_iBusWidth; /\* 总线位宽 \*/

#define SDBUS\_WIDTH\_1 0

#define SDBUS\_WIDTH\_4 2

#define SDBUS\_WIDTH\_8 4

#define SDBUS\_WIDTH\_4\_DDR 8

#define SDBUS\_WIDTH\_8\_DDR 16

LW\_LIST\_LINE\_HEADER SDADAPTER\_plineDevHeader; /\* 设备链表 \*/

} LW\_SD\_ADAPTER, \*PLW\_SD\_ADAPTER;

我们将一个物理的SD卡称作SD设备，使用lw\_sd\_device结构描述：

typedef struct lw\_sd\_device {

PLW\_SD\_ADAPTER SDDEV\_psdAdapter; /\* 从属的SD适配器 \*/

LW\_LIST\_LINE SDDEV\_lineManage; /\* 设备挂载链 \*/

atomic\_t SDDEV\_atomicUsageCnt; /\* 设备使用计数 \*/

UINT8 SDDEV\_ucType;

#define SDDEV\_TYPE\_MMC 0

#define SDDEV\_TYPE\_SDSC 1

#define SDDEV\_TYPE\_SDHC 2

#define SDDEV\_TYPE\_SDXC 3

#define SDDEV\_TYPE\_SDIO 4

#define SDDEV\_TYPE\_COMM 5

#define SDDEV\_TYPE\_MAXVAL 5 /\* 用于参数判断 \*/

UINT32 SDDEV\_uiRCA; /\* 设备本地地址 \*/

UINT32 SDDEV\_uiState; /\* 设备状态位标 \*/

#define SD\_STATE\_EXIST (1 << 0)

#define SD\_STATE\_WRTP (1 << 1)

#define SD\_STATE\_BAD (1 << 2)

#define SD\_STATE\_READONLY (1 << 3)

LW\_SDDEV\_CID SDDEV\_cid;

LW\_SDDEV\_CSD SDDEV\_csd;

LW\_SDDEV\_SCR SDDEV\_scr;

CHAR SDDEV\_pDevName[LW\_CFG\_OBJECT\_NAME\_SIZE];

PVOID SDDEV\_pvUsr; /\* 设备用户数据 \*/

} LW\_SD\_DEVICE, \*PLW\_SD\_DEVICE;

SD控制器与SD设备之间的通信（即SD会话）通过lw\_sd\_message结构描述：

typedef struct lw\_sd\_message {

LW\_SD\_COMMAND \*SDMSG\_psdcmdCmd; /\* 发送命令 \*/

LW\_SD\_DATA \*SDMSG\_psddata; /\* 数据传输属性 \*/

LW\_SD\_COMMAND \*SDMSG\_psdcmdStop; /\* 停止命令 \*/

UINT8 \*SDMSG\_pucRdBuffer; /\* 请求缓冲(读缓冲) \*/

UINT8 \*SDMSG\_pucWrtBuffer; /\* 请求缓冲(写缓冲) \*/

} LW\_SD\_MESSAGE, \*PLW\_SD\_MESSAGE;

**SD总线操作函数**

SD总线操作函数与SPI、I2C相似，它们主要函数为：

INT API\_SdLibInit(VOID);

INT API\_SdAdapterCreate(CPCHAR pcName, PLW\_SD\_FUNCS psdfunc);

PLW\_SD\_DEVICE API\_SdDeviceCreate(CPCHAR pcAdapterName, CPCHAR pcDeviceName);

INT API\_SdDeviceTransfer(PLW\_SD\_DEVICE psddevice,

PLW\_SD\_MESSAGE psdmsg,

INT iNum);

INT API\_SdDeviceCtl(PLW\_SD\_DEVICE psddevice,

INT iCmd, LONG lArg);

API\_SdLibInit初始化SD协议栈，该函数必须被首先调用。API\_SdAdapterCreate创建一个名为pcName的SD总线适配器，也是一个SD控制器驱动中最主要的部分。API\_SdDeviceCreate创建一个SD设备，应用层通过该设备对象与一个实际的SD卡通信。API\_SdDeviceTransfer和API\_SdDeviceCtl分别处理SD设备传输和总线控制操作。

**IMX6Q的SD控制器说明**

芯片IMX6Q用户手册（i.MX\_6Quad\_6Dual\_Reference\_Manual\_(Rev\_D)）中Chapter65\_Ultra Secured Digital Host Controller (uSDHC)中详细介绍了SD控制器。

实现一个完整的SD控制器驱动，需要创建SD总线适配器并实现符合SDM层规范的控制器驱动对象，将其加入SDM层。

IMX6Q的SD控制器是一个标准控制器，对于标准的SD控制器，在编写驱动时会方便很多。SDHCI作为一个标准，其功能以及寄存器结构都是确定的，仅有部分信息与具体平台有关，如控制器中断向量号、寄存器基地址等。SylixOS SD协议栈内部支持SDHCI驱动，其定义位于SylixOS/system/device/sdcard/host/sdhci.h，使用lw\_sdhci\_host\_attr结构描述不同硬件平台下SDHCI的特有属性信息：

typedef struct lw\_sdhci\_host\_attr {

/\*

\* SDHCIHOST\_pdrvfuncs 为控制器寄存器访问驱动函数

\* 通常情况下驱动程序可不提供, 内部根据

\* SDHCIHOST\_iRegAccessType 使用相应的默认驱动

\*

\* 但为了最大的适应性, 驱动程序也可使用自己的寄存器访问驱动

\*/

SDHCI\_DRV\_FUNCS \*SDHCIHOST\_pdrvfuncs; /\* 标准主控驱动函数结构指针 \*/

INT SDHCIHOST\_iRegAccessType; /\* 寄存器访问类型 \*/

#define SDHCI\_REGACCESS\_TYPE\_IO 0

#define SDHCI\_REGACCESS\_TYPE\_MEM 1

ULONG SDHCIHOST\_ulBasePoint; /\* 槽基地址指针 \*/

ULONG SDHCIHOST\_ulIntVector; /\* 控制器在 CPU 中的中断向量 \*/

UINT32 SDHCIHOST\_uiMaxClock; /\* 如果控制器没有内部时钟,用户 \*/

/\* 需要提供时钟源 \*/

/\*

\* 控制器怪异行为描述:

\* 有些控制器虽然按照 SDHCI 标准规范设计, 但在实现时存在一些

\* 不符合规范的地方(如某些寄存器的位定义, 某些功能不支持等).

\* SDHCIHOST\_pquirkop:

\* 主要针对不同的寄存器位设置, 由驱动编写者自行实现.

\* 如果该域为 NULL, 或相关的操作方法为 NULL, 则内部会使用

\* 对应的标准操作.

\* SDHCIHOST\_uiQuirkFlag:

\* 主要针对不同的特性开关, 如是否使用 ACMD12 等.

\*/

SDHCI\_QUIRK\_OP \*SDHCIHOST\_pquirkop;

UINT32 SDHCIHOST\_uiQuirkFlag;

#define SDHCI\_QUIRK\_FLG\_DONOT\_RESET\_ON\_EVERY\_TRANSACTION (1 << 0)

/\* 每一次传输前不需要复位控制器 \*/

#define SDHCI\_QUIRK\_FLG\_REENABLE\_INTS\_ON\_EVERY\_TRANSACTION (1 << 1) /\* 传输后禁止, 传输前使能中断 \*/

#define SDHCI\_QUIRK\_FLG\_DO\_RESET\_ON\_TRANSACTION\_ERROR (1 << 2) /\* 传输错误时复位控制器 \*/

#define SDHCI\_QUIRK\_FLG\_DONOT\_CHECK\_BUSY\_BEFORE\_CMD\_SEND (1 << 3) /\* 发送命令前不执行忙检查 \*/

#define SDHCI\_QUIRK\_FLG\_DONOT\_USE\_ACMD12 (1 << 4) /\* 不使用 Auto CMD12 \*/

#define SDHCI\_QUIRK\_FLG\_DONOT\_SET\_POWER (1 << 5) /\* 不操作控制器电源的开/关 \*/

#define SDHCI\_QUIRK\_FLG\_DO\_RESET\_AFTER\_SET\_POWER\_ON (1 << 6) /\* 当打开电源后执行控制器复位 \*/

#define SDHCI\_QUIRK\_FLG\_DONOT\_SET\_VOLTAGE (1 << 7) /\* 不操作控制器的电压 \*/

#define SDHCI\_QUIRK\_FLG\_CANNOT\_SDIO\_INT (1 << 8) /\* 控制器不能发出 SDIO 中断 \*/

#define SDHCI\_QUIRK\_FLG\_RECHECK\_INTS\_AFTER\_ISR (1 << 9) /\* 中断服务后再次处理中断状态 \*/

#define SDHCI\_QUIRK\_FLG\_CAN\_DATA\_8BIT (1 << 10) /\* 支持8位数据传输 \*/

#define SDHCI\_QUIRK\_FLG\_CAN\_DATA\_4BIT\_DDR (1 << 11) /\* 支持4位ddr数据传输 \*/

#define SDHCI\_QUIRK\_FLG\_CAN\_DATA\_8BIT\_DDR (1 << 12) /\* 支持8位ddr数据传输 \*/

#define SDHCI\_QUIRK\_FLG\_MMC\_FORCE\_1BIT (1 << 13) /\* MMC 卡强制使用1位总线 \*/

VOID \*SDHCIHOST\_pvUsrSpec; /\* 用户驱动特殊数据 \*/

} LW\_SDHCI\_HOST\_ATTR, \*PLW\_SDHCI\_HOST\_ATTR;

使用函数API\_SdhciHostCreate创建一个SDHCI标准控制器对象：

PVOID API\_SdhciHostCreate(CPCHAR pcAdapterName,

PLW\_SDHCI\_HOST\_ATTR psdhcihostattr)

API\_SdhciHostCreate会自动创建名称为pcAdapterName的SD总线适配器并创建SDM层驱动，并初始化SDHCI硬件以保证硬件控制器可用。但与控制其相关的端口初始化、时钟模块初始化等工作需要用户驱动提供。

IMX6Q的SD驱动所在目录为bsp工程的driver/sdi文件夹，文件imx6q\_sdi.c的 imx6qSdLibInit函数完成了整个SD系统的初始化工作，其实现如程序清单12.1：

程序清单12.1 imx6qSdLibInit函数代码

VOID imx6qSdLibInit (VOID)

{

API\_SdmLibInit();

API\_SdmSdioLibInit(); /\* SD 库初始化, 必须放在前面 \*/

API\_SdioBaseDrvInstall();

API\_SdMemDrvInstall(); /\* 安装应用层驱动 \*/

}

在初始化SD系统组件库后，分别安装了SD存储卡驱动和SDIO基础驱动。API\_SdmLibInit在内部已经调用了API\_SdLibInit函数，因此无需再次调用API\_SdLibInit了。

imx6qSdiDrvInstall安装SD控制器驱动，其函数实现位于imx6q\_sdi.c文件。SD初始化函数imx6qSdLibInit需要在bsp工程的../bsp/imx6q\_sylixos.c的boardDevInit函数中调用。

### 实验步骤

1. 编译工程所在BSP工程bspimx6，生成bspimx6.bin文件；
2. 参考1.4 SylixOS系统部署章节，将文件bspimx6.bin上传到验证平台上，在系统调试时并不需要将文件烧录到nandflash上，完成文件上传后，输入命令“go 0x10000000”启动SylixOS系统；
3. 插入SD卡，在SylixOS控制台中会输出系统信息，SD卡挂载目录/media/sdcard0。

[root@sylixos\_station:/]#

Block device /dev/blk/sdcard1 part 0 mount to /media/sdcard2 use vfat file system. /\* 插入 SD 后系统提示信息 \*/

SD Memory Information >>

Manufacturer : 0x02

OEM ID : TM

Media Type : SDHC(v2.0)

Product Name : SA08G

Product Vsn : v2.1

Serial Num : 456022cc

Date : 2016/04

Max Speed : 25MB/s

Capacity : 7420.000 MB

[root@sylixos\_station:/]# cd /media/sdcard2 /\* 进入 SD 卡目录 \*/

[root@sylixos\_station:/media/sdcard2]#

### 扩展实验

仔细阅读SD总线驱动原理介绍和BSP工程中SD驱动处理方式，总结SylixOS下对I2C、SPI、SD总线的管理特点。

# framebuffer编程实验

当前绝大多数开源图形界面库都是基于framebuffer编写，了解framebuffer的使用，可以方便的将一个图形界面库移植到SylixOS下，此外framebuffer还可以方便的用来测试硬件驱动。

## 实验一 framebuffer实验

### 实验目的

* 了解framebuffer的原理
* 了解基于framebuffer的图形绘制方法

### 实验内容

* 阅读framebuffer.c文件，了解framebuffer绘图方法和相关函数使用
* 在RealEvo-IDE下编译framebuffer工程，并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

framebuffer中文名称为帧缓冲，framebuffer是SylixOS系统为显示设备提供的一个标准设备接口模型，它将显示缓冲区抽象，屏蔽图形硬件的底层差异，允许上层应用程序在图形模式下直接对显示缓冲区进行读写操作。用户不必关心物理显示缓冲区的具体位置及存放方式，这些都由framebuffer设备驱动本身完成。对于framebuffer设备而言，只需要在framebuffer中与显示点对应的区域写入颜色值，对应的颜色会自动在屏幕上显示。

例如具备LCD控制器的处理器在打开framebuffer设备后，DMA控制器会持续将内存中的数据送到LCD控制器，LCD控制器将得到的数据配以时钟和场行信号送往LCD显示设备，LCD像素点根据得到的数据显示相应颜色组成显示图像。

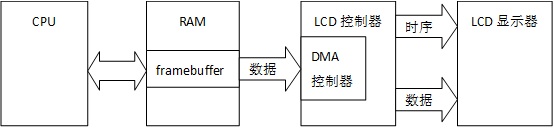


图 13.1 framebuffer显示原理

显示器一般是一块矩形区域，每个像素点都有其对应的二维坐标(x,y)。在内存中的一块存储区域往往是一维的一段线性地址（addr）。内存中的地址与显示器上的像素坐标对应关系一般如公式所示：

addr = y\*显示器一行的像素点 + x；(0< x <行像素点，0< y <列像素点 )

显示器的(0,0)坐标是在显示器的左上角，而不是数学中常用坐标系的左下角。

本实验例程首先输出当前framebuffer的参数，随后绘制白灰黑色条，最后通过操作绘制点的函数绘制一条红色直线。

**代码分析：**

程序清单 13.1 framebuffer.c文件中main函数代码

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/mman.h>

#include <system/device/graph/gmemDev.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: main

\*\* 功能描述: LCD 与触摸屏测试

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \*argv[])

{

int i;

int ret;

int fb\_fd;

LW\_GM\_SCRINFO scr\_info;

LW\_GM\_VARINFO var\_info;

void \*pframebuffer;

fb\_fd = open("/dev/fb0", O\_RDWR, 0666); /\* 打开 fb 设备 \*/

if (fb\_fd < 0) {

printf("failed to open /dev/fb0\n");

return (-1);

}

ret = ioctl(fb\_fd, LW\_GM\_GET\_SCRINFO, &scr\_info); /\* 获取 fb 的大小等参数 \*/

if (ret < 0) {

printf("failed to get /dev/fb0 screen info\n");

close(fb\_fd);

return (-1);

}

ret = ioctl(fb\_fd, LW\_GM\_GET\_VARINFO, &var\_info); /\* 获取 fb 行列信息等 \*/

if (ret < 0) {

printf("failed to get /dev/fb0 var info\n");

close(fb\_fd);

return (-1);

}

printf("the mem size :%d\n", /\* fb 占用内存的大小 \*/

scr\_info.GMSI\_stMemSize);

printf("the line\_length is :%d\n", /\* fb 每行像素点字节数 \*/

scr\_info.GMSI\_stMemSizePerLine);

printf("the xres is :%d\n", /\* fb 显示区域高度 \*/

(int)var\_info.GMVI\_ulXRes);

printf("the yres is :%d\n", /\* fb 显示区域宽度 \*/

(int)var\_info.GMVI\_ulYRes);

printf("bits\_per\_pixel is :%d\n", /\* fb 单像素点的 bits \*/

(int)var\_info.GMVI\_ulBitsPerPixel);

/\*

\* 将 framebuffer 物理内存映射到用户空间

\*/

pframebuffer = mmap(NULL, scr\_info.GMSI\_stMemSize,

PROT\_READ | PROT\_WRITE, MAP\_SHARED, fb\_fd, 0);

if (pframebuffer == MAP\_FAILED) {

printf("failed to mmap /dev/fb0\n");

close(fb\_fd);

return (-1);

}

memset(pframebuffer, 0xFF, /\* 屏幕全白 \*/

scr\_info.GMSI\_stMemSize);

sleep(2);

memset(pframebuffer, 0x88, /\* 屏幕半灰 \*/

scr\_info.GMSI\_stMemSize / 2);

sleep(2);

memset(pframebuffer, 0x00, /\* 屏幕 1/4 黑 \*/

scr\_info.GMSI\_stMemSize / 4);

for (i = 0;i < var\_info.GMVI\_ulXRes;i++) { /\* 绘制一条红色直线 \*/

draw\_pixel(pframebuffer, &scr\_info, &var\_info, i, 50, 0x00FF0000);

}

sleep(30);

/\*

\* 程序退出前，释放占用的相关资源

\*/

munmap(pframebuffer, scr\_info.GMSI\_stMemSize);

close(fb\_fd);

printf("framebuffer exit\n");

return (0);

}

能够绘制一个点的时候，理论上就可以绘制任意图形，因此在图形界面编程中，绘制点的函数非常重要，程序清单 13.2是绘制点的函数代码。

程序清单 13.2 framebuffer.c程序的绘制点的函数

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名称: draw\_pixel

\*\* 功能描述: LCD 绘制点

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static void draw\_pixel (void \*pframebuffer,

LW\_GM\_SCRINFO \*scr\_info,

LW\_GM\_VARINFO \*var\_info,

int x,

int y,

unsigned colidx)

{

if ( (x < 0) || (x >= var\_info->GMVI\_ulXResVirtual) ||

(y < 0) || (y >= var\_info->GMVI\_ulYResVirtual)) {

return;

}

switch (var\_info->GMVI\_ulBitsPerPixel) {

case 16:

{

uint16\_t \*pusFrameBuffer;

pusFrameBuffer = (uint16\_t \*)((uint8\_t \*)pframebuffer +

y \* scr\_info->GMSI\_stMemSizePerLine);

pusFrameBuffer[x] = ((colidx & 0xFF0000) >> 16 >> 3 << 11) |

((colidx & 0x00FF00) >> 8 >> 2 << 5) |

((colidx & 0x0000FF) >> 0 >> 3 << 0);

}

break;

case 24:

case 32:

{

uint32\_t \*puiFrameBuffer;

puiFrameBuffer = (uint32\_t \*)((uint8\_t \*)pframebuffer +

y \* scr\_info->GMSI\_stMemSizePerLine);

puiFrameBuffer[x] = colidx;

}

break;

}

}

### 实验步骤

1. 使用RealEvo-IDE打开工程framebuffer并编译，将生成的程序文件上传到验证平台；
2. 执行程序，在控制台会输出当前framebuffer设备的参数；

[root@sylixos\_station:/apps/framebuffer]# ./framebuffer

the mem size : :2457600

the line\_length is : :4096

the xres is :1024

the yres is :600

bits\_per\_pixel is :32

framebuffer exit

[root@sylixos\_station:/apps/framebuffer]#

1. 查看验证平台LCD显示器，显示器显示白、灰、黑三色条纹和一条红色直线。

### 扩展实验

了解市场常见的开源GUI库有哪些是基于framebuffer的，选择其中一个移植到到SylixOS系统上。

# 共享库实验

SylixOS下的共享库与Linux下的共享库概念相同，格式也类似。在Windows系统下实现共享库功能的技术被称为动态链接库，因此在一些书籍中会混合使用共享库和动态库名称，实质上指的是同一个事物。

本章设计了两个实验，实验一重点介绍了SylixOS共享库创建和使用的流程，实验二重点介绍了如何把一个第三方库编译成SylixOS下共享库。

## 实验一 共享库基础实验

### 实验目的

* 掌握共享库工程的设计流程
* 对共享库的功能实现和使用有形象的认识

### 实验内容

* 创建一个共享库工程和一个共享库的使用工程
* 使用RealEvo-IDE编译工程并将程序上传到验证平台运行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

“库”用于将函数打包在一个单元中，然后这些单元可以为其他程序所共享。库分两种，一种是静态库，静态库是在程序链接时链接进可执行程序文件里。另外一种是共享库，共享库会在加载应用程序时被加载，因此它只在程序运行时绑定，其他时刻是分离的。

使用静态库，每一个运行的程序都会有一份库的副本，这增加了对系统内存的占用。而使用共享库在内存中，代码段只有一个副本，并且一个应用程序可以加载很多个不同的共享库，在大型程序中可以极大的节约系统资源和方便程序模块化设计，因此共享库得到了广泛的使用。大多数第三方软件就是以共享库的方式提供使用。

在共享库的使用上，分为自动加载共享库和手动加载共享库。自动加载共享库是在应用程序编译时设置好依赖关系，装载器会自动加载应用程序所依赖的共享库；手动加载共享库是应用程序利用dlopen、dlclose等函数来加载一个共享库，并手动获取（dlsym）指定符号内存地址的方法。本实验使用的是自动加载共享库的方式。

在SylixOS等其他POSIX兼容系统中，共享库的命名方式以“lib”开头，以“.so”为后缀名。一个完整的共享库名称为lib\*.so。共享库的放置目录默认是/lib，对用户自己向SylixOS设备添加的共享库均建议放在默认目录下。

本实验通过演示一个最简单的共享库工程和共享库使用工程的创建，使读者对共享库的编写和使用有形象的认识。

### 实验步骤

1. 新建工程；

“File”→“New”→“Other”→“SylixOS Shared Lib” →“点击Next”工程名称dll\_demo

或者在空白区直接“右键”→“New”→“SylixOS Shared Lib” 工程名称填写dll\_demo。

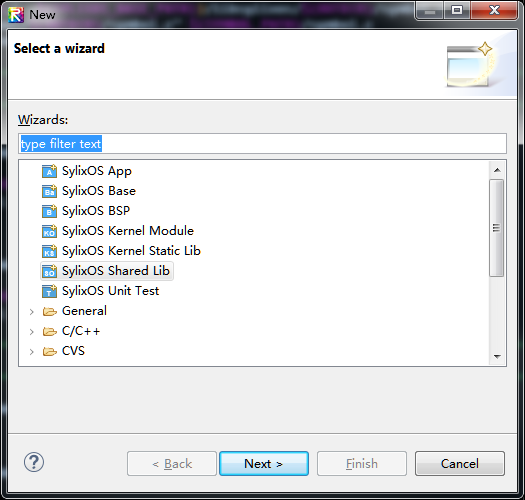


图 14.1 共享库工程创建选项

随后点击“Next”，填写工程名，设置配置信息，最后点击“Finish”完成工程创建。

1. 创建头文件；

“File”→“New”→“Header File”，创建头文件dll\_demo.h，然后在头文件里面填写对应.c文件中的函数声明。

1. 编译工程，生成libdll\_demo.so文件；
2. 建立动态链接库的使用工程；

“右键”→“New”→“SylixOS App”，填写工程名称dll\_demo\_use，点击“Finish”完成工程创建。

1. 在dll\_demo\_use.c文件中填写dll\_demo工程的头文件；
2. 在main函数中加入dll\_demo工程中的函数，如程序清单 14.1；

程序清单 14.1 dll\_demo\_use.c

#include "../../dll\_demo/src/dll\_demo.h"

int main (int argc, char \*argv[])

{

printf("Hello World!\n");

lib\_func (); /\* dll\_demo里的函数 \*/

return (0);

}

App工程配置选中工程“右键”→“properties”→“SylixOS Project” →“Linker Setting”如图15.2动态链接库的使用工程配置流程。

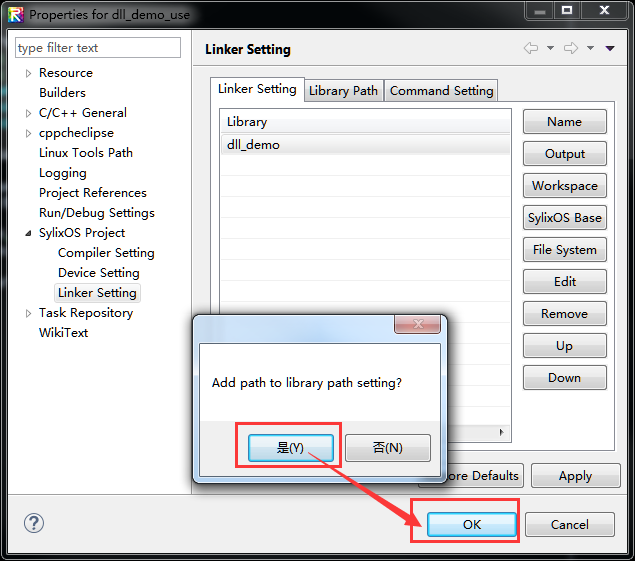
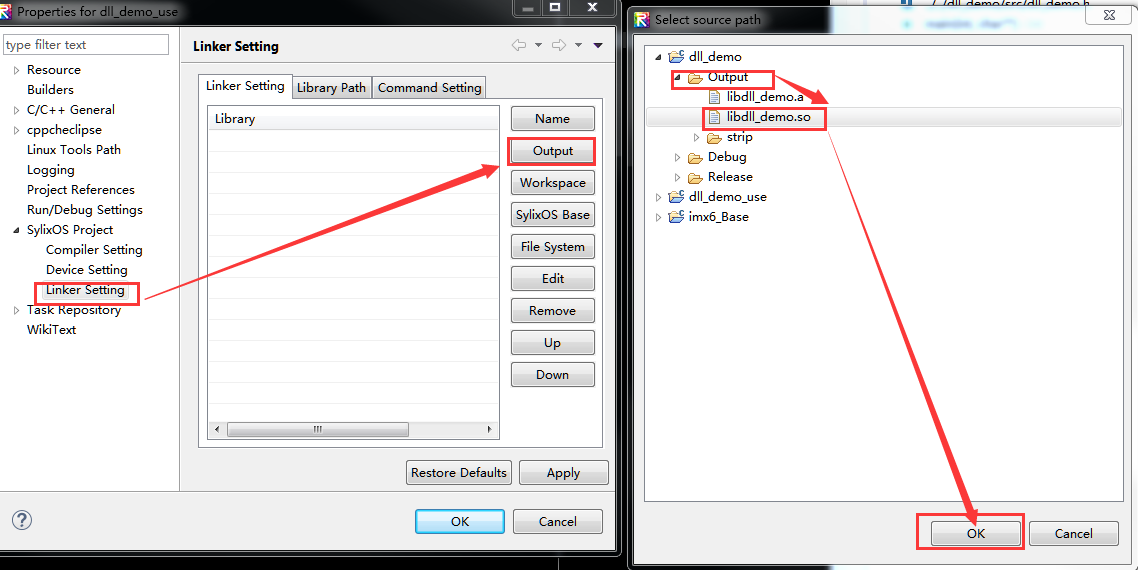


图14.2动态链接库的使用工程配置流程。

执行程序验证对共享库的调用。

将dll\_demo工程中的libdll\_demo.so文件上传到SylixOS系统的lib目录下，将应用程序dll\_demo\_use拷贝到/apps/dll\_demo\_use目录下运行。

[root@sylixos\_station:/apps/dll\_demo\_use]# ./dll\_demo\_use

Hello SylixOS!

lib\_func call!

[root@sylixos\_station:/apps/dll\_demo\_use]#

### 扩展实验

了解共享库手动加载方式并进行编程验证。

## 实验二 zlib库使用实验

### 实验目的

* 了解zlib的功能和使用方法
* 掌握简单的第三方软件转换成SylixOS系统下共享库的方法

### 实验内容

* 将zlib库编译成SylixOS系统下的共享库
* 编写一个简单的使用例程，实现对zlib库的验证

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

zlib是目前一款得到广泛使用文件压缩库，如Linux内核、OpenSSL、libpng、ffmpeg等开源工程均使用了zlib，粗略估计有数千种软件直接或间接依赖zlib提供的压缩库。zlib的广泛使用一方面说明了其技术上的优越和实用，另一方面也使使用zlib库的文件具备良好的通用性。

本实验完整的讲述了从zlib官方库中提取文件，加入SylixOS的共享库工程，编译出能够在SylixOS下使用的zlib共享库。随后编写一简单的测试程序，验证zlib库可用。

### 实验步骤

**移植zlib库**

从网站http://www.zlib.net/ 下载zlib源代码，本实验中使用了tar.xz格式。下载后文件名为zlib-1.2.8.tar.xz，使用普通的7zip解压软件解压到当前文件。解压后的文件所在文件夹名称为zlib-1.2.8。

    进入zlib-1.2.8文件夹，打开/watcom/watcom\_l.mak文件。会看到如下内容：

 C\_SOURCE = adler32.c compress.c crc32.c deflate.c &

gzclose.c gzlib.c gzread.c gzwrite.c &

infback.c inffast.c inflate.c inftrees.c &

trees.c uncompr.c zutil.c

 \*.c文件是实现zlib库压缩功能的文件。将zlib-1.2.8文件夹下中同名的\*.c文件拷贝到一新建文件夹zlib下，并将相关的\*.h文件也一并拷贝到zlib文件夹下。

    打开RealEvo-IDE，建立一个“SylixOS Shared  Lib”工程，命名为zlib\_dll，将上一步骤中zlib文件夹下的\*.c和\*.h文件拷贝到工程zlib\_dll的目录下，并将自动生成的zlib\_dll.c文件删除。

     编译生成libzlib\_dll.so文件至此我们已经完成zlib库在SylixOS下移植。

**zlib库的验证**

       新建“SylixOS App”类型工程zlib\_dll\_use，添加zlib头文件；

#include "../../zlib\_dll/src/zlib.h"

将生成的main函数更改如程序清单 14.2；

程序清单 14.2 zlib\_dll\_use.c

int main (int argc, char \*argv[])

{

unsigned char data\_in[] = "hello world! aaaaa bbbbb ccccc

ddddd 1234567890 notrecongen yes";

unsigned char buf[1024] = { 0 };

unsigned char data\_out[1024] = { 0 };

unsigned long srcLen = sizeof(data\_in);

unsigned long bufLen = sizeof(buf);

unsigned long dstLen = sizeof(data\_out);

printf("src string:%s\nlength:%ld\n", data\_in, srcLen);

compress(buf, &bufLen, data\_in, srcLen); /\* 压缩 \*/

printf("after compressed length:%ld\n", bufLen);

uncompress(data\_out, &dstLen, buf, bufLen); /\* 解压缩 \*/

printf("after uncompressed length:%ld\n",dstLen);

printf("uncompressed string:%s\n",data\_out);

}

App工程配置选中工程“右键”→“properties”→“SylixOS Project” →“Linker Setting”如图14.3动态链接库的使用工程配置流程。

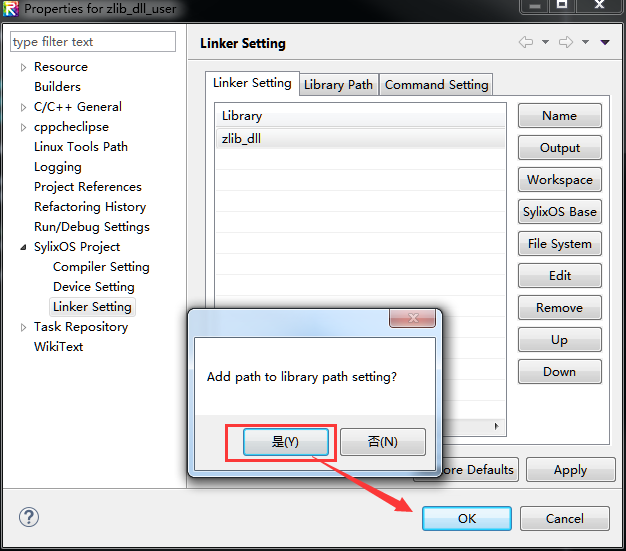
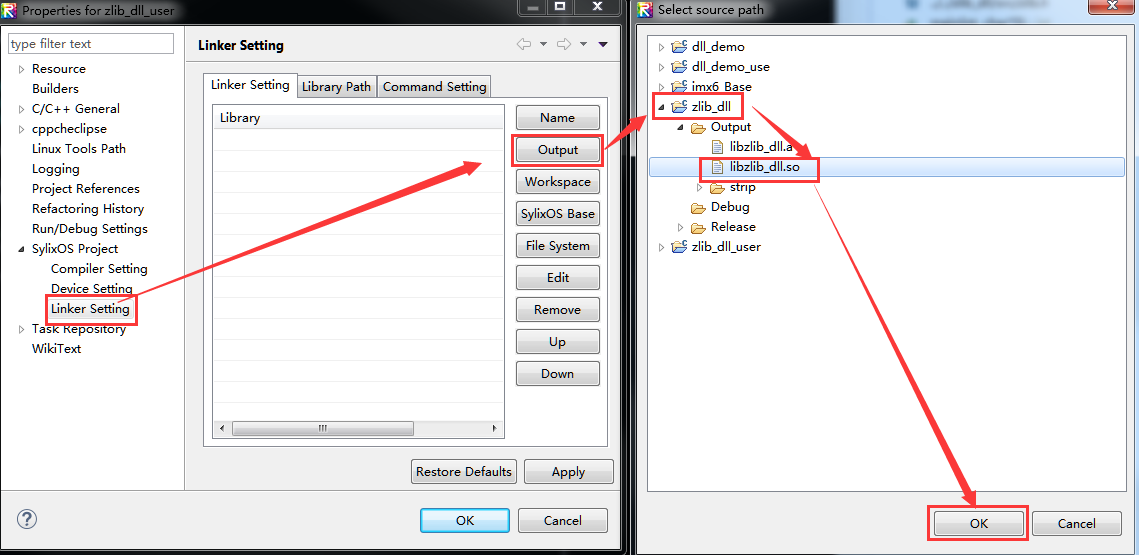


图14.3动态链接库的使用工程配置流程。

编译zlib\_demo工程

使用RealEvo-IDE里面的Upload工具，将libzlib.so文件上传到SylixOS设备中的/lib目录，zlib库使用程序上传到/apps/ zlib\_dll\_user，执行程序。

[root@sylixos\_station:/apps]# ./zlib\_dll\_user/zlib\_dll\_user

src string:hello world! aaaaa bbbbb ccccc ddddd 1234567890 notrecongen yes

length:64

after compressed length:62

after uncompressed length:64

uncompressed string:hello world! aaaaa bbbbb ccccc ddddd 1234567890 notrecongen yes

[root@sylixos\_station:/apps]#

说明zlib库可以使用，由于文本较小，压缩后只节约了2个字节。

### 扩展实验

在SylixOS官网上有很多第三方库，有些能够在RealEvo-IDE下编译，有些还需要在Linux下编译。读者任选其一，对其进行练习测试。

# Qt图形界面编程实验

图形界面编程是软件开发中一个很重要的内容，基于图形界面的软件已经形成了一个独立的产业，诞生了诸多开源和商业项目。本章介绍一款嵌入式领域广泛使用的大型图形系统Qt。

为提高Qt工程的开发效率，北京翼辉信息技术有限公司提供了配套的开发环境和组件。在SylixOS的安装光盘的文件夹中如图 1.1，文件夹“RealEvo-QtSylixOS”下有文档《RealEvo-QtSylixOS使用手册》，文档详细介绍了Qt在Windows下的开发环境的安装、SylixOS上支持Qt图形界面编程的方法。通过练习文档中的示例，能够掌握基本的Qt图形界面开发的方法。

注：在进行Qt相关实验前务必阅读《RealEvo-QtSylixOS使用手册》。

## 实验一 Qt工程导入

### 实验目的

* 了解Qt工程路径变更后需要更改的配置
* 掌握Qt工程工程路径变更后，更改设置进行编译的方法

### 实验内容

* 修改与实验配套的gpio\_out工程配置，使其能够在读者的PC主机上编译

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、RealEvo-QtSylixOS、putty

### 实验原理

同一个Qt工程会有两个相关文件夹，在拷贝工程时只需要拷贝与工程名称一致的文件夹即可，包含“Debug” 或“Release”关键字的文件夹不需要拷贝。

Qt工程在从一个主机拷贝到另外一个主机上后，会出现工程路径变化、Qt开发环境配置不一致等因素，使得拷贝后的Qt工程需要进行一些配置修改，才可以编译使用。

### 实验步骤

1. 打开工程gpio\_out

打开QtCreator，在开始界面中点击打开gpio\_out工程文件夹中的gpio\_out.pro文件，弹出设置对话框如图 15.1，选择“No”。

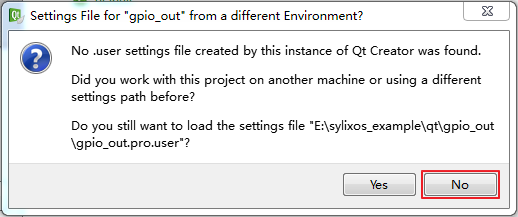


图 15.1 Qt设置更改提示对话框

随后会出现“Cofigure Project”界面如图 15.2，设置工程使用的SDK，

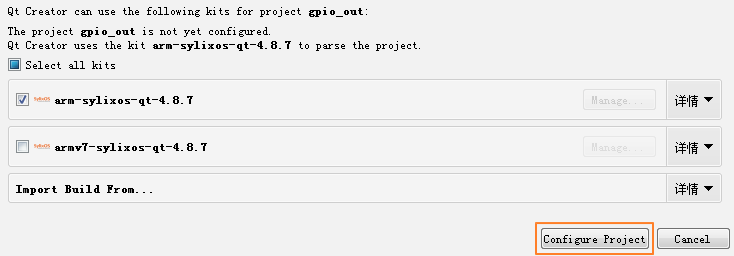


图 15.2 Qt SDK选择对话框

这里使用的是Cortex-A9的处理器属于ARM V7架构，选择armv7-sylixos-qt-4.8.7的SDK。然后点击“Configure Project”。

1. 修改设备配置

打开“工具”→“选项”对话框，设置SylixOS设备参数，如果没有SylixOS设备，使用“添加”按钮新建一个SylixOS设备并“开始向导”。

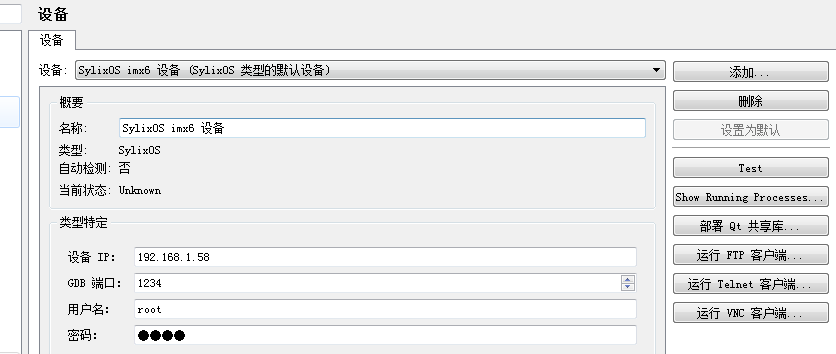


图 15.3 QtCreator下SylixOS设备设置

* 主机名：验证平台的ip地址
* 其他保持默认配置（用户名与密码均为：root）

1. 设置SylixOS base project路径

选择“构建和运行”→“构建套件”→“armv7-sylixos-qt-4.8.7”，在此处配置SylixOS base project路径，此路径是RealEvo-IDE工程下sylixos\_base路径。

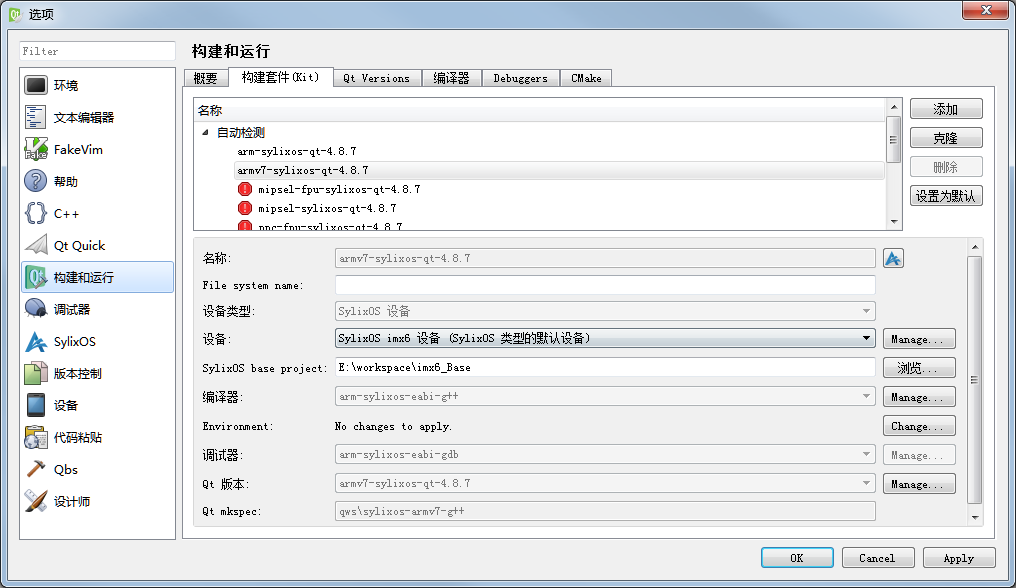


图 15.4 sysroot路径参数

1. 点击选择构建套件和Base的Debug Level（debug / release）相同的类型。



1. 点击对gpio\_out编译



1. 点击运行gpio\_out程序



### 扩展实验

将工程gpio\_in在PC主机上编译并部署到验证平台运行

## 实验二 Qt下GPIO输出实验

### 实验目的

* 掌握QtCreater程序的开发流程
* 掌握Qt下操作SylixOS硬件设备的方法

### 实验内容

* 创建一个Qt工程实现对验证平台GPIO的输出操作
* 将Qt工程部署到验证平台上执行查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、RealEvo-QtSylixOS、putty

### 实验原理

Qt是一个运行在SylixOS上的图形库。因此在Qt应用程序中操作SylixOS其他资源如硬件资源，与普通应用程序没有区别。

Qt应用使用C++语言编写，在文档组织和语法上与前面诸多章节仅仅使用C语言的工程会有所不同。

### 实验步骤

1. 创建Qt工程

打开QtCreator，使用开始界面的“New Project”按钮开启Qt工程创建向导，根据向导创建一个名为gpio\_out的工程；工程类型为“Qt Widgets Application”

1. 放置按钮控件

* 双击“界面文件”→“mainwindow.ui”文件，QtCreator会切换到界面编辑窗口
* 向主界面中放入“Push Button”控件

1. 修改按钮控件属性

在界面编辑窗口右下方属性编辑区域修改控件属性，完成后如图 15.5所示

* objectName：btnLed
* text：LED1
* font：14



图 15.5 属性设置后的按钮控件

1. 创建按钮响应函数

选择“LED操作”按钮右键→“转到槽”弹出设置对话框，如图 15.6。选择“clicked()”，单击“OK”

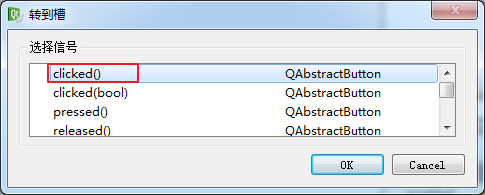


图 15.6 按钮响应函数设置对话框

1. 添加“LED操作”按钮相应代码

完成第4步后，窗口会自动打开“mainwindow.cpp”文件，并跳转到对应函数处。

void MainWindow::on\_btnLed\_clicked()

{

}

在函数中加入如程序清单 15.1所示的代码。

程序清单 15.1 void MainWindow::on\_btnLed\_clicked()函数代码

/\*

\* 打开 LED 的 GPIO 文件

\*/

if(status == 0) {

fd = gpiofd(gpioled, O\_WRONLY, GPIO\_FLAG\_OUT\_INIT\_LOW);

if (fd < 0) {

printf("gpiofd create error!");

return ;

}

status = 1;

}

/\*

\* 对gpio进行取反操作

\*/

if (g\_led\_flag) {

g\_led\_flag = 0;

gpiofd\_write(fd, 1);

QMessageBox::information(NULL, "GPIO Oper", "LED OFF");

} else {

g\_led\_flag = 1;

gpiofd\_write(fd, 0);

QMessageBox::information(NULL, "GPIO Oper", "LED ON");

}

添加需要的头文件和全局变量，在文件“mainwindow.cpp”开始位置添加头文件和全局变量：

#include <QMessageBox>

#include <stdio.h>

#include <unistd.h>

#include <pthread.h>

#include <sys/select.h>

#include <sys/gpiofd.h>

/\*

\*定义全局变量

\*/

int g\_led\_flag = 0;

1. 配置构建环境

* 点击选择构建套件和Base的Debug Level（debug / release）相同的类型。



1. 编译工程gpio\_out

* 点击编译工程gpio\_out



1. 运行程序gpio\_out

* 点击会开始程序部署和执行



* 若设备无法连接，参考本章实验二设置Qt运行设备

1. 控制LED亮灭

* gpio\_out程序运行后验证平台LCD会呈现软件设计界面，点击“LED1”按钮，可以看到验证平台板载LED亮灭交替。

### 扩展实验

在Qt下实现验证平台多个LED的控制。

## 实验三 Qt下GPIO输入实验

### 实验目的

* 掌握Qt下GPIO输入编程方法
* 掌握Qt多线程编程的方法

### 实验内容

* 创建名为gpio\_in的Qt工程，使用QThread多线程技术完成GPIO输入操作
* 将工程编译并部署到验证平台并执行，查看效果

### 实验环境

* 硬件：已经部署SylixOS操作系统的验证平台，一台PC主机
* 软件：RealEvo-IDE、putty

### 实验原理

相比GPIO的输出，输入操作会稍复杂一些，由于GPIO的输入事件是随机的，如果简单的使用等待读取的方式，会阻塞整个Qt进程（Qt主线程），使Qt界面“卡死”。所以在处理GPIO输入上，常用的做法是针对特定GPIO编写自定义按键的驱动，将异步事件与Qt的“信号/槽”机制绑定。这样当特定GPIO的管脚电平发生变化时，会产生一个特定信号传入Qt界面，Qt界面将自动调用信号绑定的“槽”函数实现对GPIO输入的响应。

本实验以Qt下多线程的方式实现对GPIO输入的响应。Qt自身提供了QThread类实现多线程的编程，这样在创建线程时可以使用QThread类也可以使用pthread，基于跨平台的考虑这里只介绍使用QThread类实现Qt下多线程的编程。在今后的Qt编程中也建议读者尽量使用与平台无关的函数，以充分体现Qt的跨平台特性。

QThread类的使用步骤

* 声明一个MyThread类继承QThread类；
* 在工程中对MyThread::run()函数填充，这里填充的是GPIO状态读取的代码，函数中需要调用emit 向主线程发送表示GPIO状态的信号；
* 将MyThread发送的信号与主线程中的控件操作函数使用信号/槽机制绑定。

### 实验步骤

1. 创建Qt工程

打开QtCreator，使用开始界面的“New Project”按钮开启Qt工程创建向导，根据向导创建一个名为gpio\_in的工程，工程类型为“Qt Widgets Application”；

1. 放置控件

* 双击“界面文件”→“mainwindow.ui”文件，QtCreator切换到界面编辑窗口
* 向主界面中放入“LCD Number”控件，保持默认配置，如图 15.7；



图 15.7 LCD Number控件

1. 添加“LCD Number”控件的信号槽函数

打开mainwindow.h文件，在MainWindow类中代码：

/\*

\* 添加界面主线程的信号槽函数，用来实现与线程的信号对接

\*/

private slots:

void lcdnumValSet(int num);

打开mainwindow.cpp文件，加入槽函数的实现：

void MainWindow::lcdnumValSet(int num)

{

ui->lcdNumber->display(num);

}

1. 添加QThread类声明和类的实现

打开mainwindow.h文件，添加QThread头文件。

#include <QThread> /\*添加qt线程的头文件\*/

在mainwindow.h文件下方添加MyThread类声明，如程序清单 15.2。

程序清单 15.2 声明MyThread类

class MyThread: public QThread

{

Q\_OBJECT

public:

MyThread();

void run();

signals:

void send(int num);

};

打开mainwindow.cpp文件，添加对MyThread类的实现，如程序清单 15.3。

程序清单 15.3 MyThread类函数实现

MyThread::MyThread()

{

}

void MyThread::run()

{

int inum = 10;

int ret;

int fd;

fd\_set fdset;

uint8\_t value;

int gpio\_in\_num = 36;

fd = gpiofd(gpio\_in\_num, 0, GPIO\_FLAG\_TRIG\_FALL | GPIO\_FLAG\_TRIG\_RISE);

if (fd <= 0) {

printf("gpiofd create error!");

return ;

}

FD\_ZERO(&fdset);

while (1) {

FD\_SET(fd, &fdset);

ret = select(fd + 1, &fdset, NULL, NULL, NULL);

if (ret == 1) {

gpiofd\_read(fd, &value);

printf("value : %d\n", value);

inum++;

/\*

\* 向主线程发送线程信号，发送后延时0.5s，做消抖处理

\*/

emit send(inum);

usleep(500 \* 1000);

} else {

break;

}

}

::close(fd);

exec();

}

在mainwindow.cpp文件开始位置添加GPIO操作的相关头文件。

#include <stdio.h>

#include <unistd.h>

#include <pthread.h>

#include <sys/select.h>

#include <sys/gpiofd.h>

1. 添加MyThread类初始化代码和信号槽绑定

mainwindow.cpp文件的QMainWindow函数下默认有代码：

ui->setupUi(this);

在其下方填写代码：

MyThread \*myth = new MyThread;

connect(myth,SIGNAL(send(int)),this,SLOT(lcdnumValSet(int)));

myth->start();

注： 信号与槽是Qt独有的事件通知机制，是Qt最伟大的发明之一。connect函数是信号与槽的连接函数，具体说明请参见Qt相关书籍。

1. 配置构建环境

* 点击选择构建套件和Base的Debug Level（debug / release）相同的类型。



1. 编译工程gpio\_in编译并部署执行

* 点击编译工程gpio\_in



1. 将gpio\_in工程部署到验证平台并执行。

* 点击按钮将gpio\_in工程部署到验证平台执行；



* 点击Key1按键，会发现Qt程序显示的数字在增长。

### 扩展实验

使用Qt编写一个SylixOS系统下的串口调试助手。

1. 推荐阅读书目

《Linux /UNIX系统编程手册》 孙剑 等译

《UNIX环境高级编程》 戚正伟 等译

《C++ GUI Qt 4编程（第二版）》 闫锋欣 等译

《ARM嵌入式系统开发—软件设计与优化》 沈建华 译

1. SylixOS ARM BSP简介

SylixOS BSP分为两大部分：

* 基础支持程序
* 驱动程序

其中“基础支持程序”主要是实现系统的启动流程与硬件板最基础的功能实现，例如中断控制器管理，TICK定时器管理，初始内存映射管理等等。“驱动程序”主要是实现操作系统对外部设备的管理。

此篇博客主要介绍SylixOS BSP文件的构成。SylixOS BSP可通过RealEvo-IDE创建，创建出来的BSP模板分为三个文件夹：bsp、driver、user。

* bsp 文件夹下的文件对应的就是“基础支持程序”；
* driver 文件夹下的文件对应的就是“驱动程序”；
* user 目录为 t\_main 内核线程实现，可以根据项目需要加入自己的代码。

bsp文件夹下存在至少5个源文件，他们分别是：config.h、startup.S、bspInit.c、bspLib.c、bspMap.h。当然SylixOS并不严格规定BSP必须这样写，只是长期以来形成的习惯。

* config.h 文件包含板子的基本配置情况，如果BSP包含对多个电路板的支持，则可以在bsp目录下再建立一些针对各种板子的文件夹，具体方法可参考TI 335x BSP；
* startup.S 文件是系统的启动入口；
* bspInit.c 文件是 bsp 初始化入口，其中初始化函数为 bspInit()，此函数会被 startup.S 文件调用；
* bspLib.c 文件是“基础支持程序”主要实现部分，包括中断管理，TICK初始化等等；
* bspMap.h 文件是操作系统内存布局设置与初始映射配置。

driver文件夹下就是“驱动程序”的实现，内部分为很多子文件夹，每一个文件夹都是指定设备的驱动程序，例如uart（或者 sio tty 等）是代表串口的驱动程序。

user目录包含t\_main.c文件，系统完成所有的初始化后进入多任务模式，其中t\_main是BSP创建的内核任务，SylixOS既可以像Linux一样动态装载用户进程，也可以像其他小型 RTOS一样直接将用户应用程序和BSP一起编译，运行在内核空间，如果选择将应用程序和BSP一起编译，t\_main任务可以认为是用户应用程序的入口，不过SylixOS还是不推荐这种方法，SylixOS推荐开发者像使用Linux或者Windows一样，将指定的应用编译成独立进程，通过 SylixOS装载器动态装载执行，方便升级与调试。

1. SylixOS系统调试及固化

SylixOS开发套件在出厂是已经部署SylixOS操作系统及其BSP固件，能满足一般的应用开发需求。如果用户需要学习底层驱动程序，可以使用套件里面的BSP工程源代码进行编译和调试。

将配套BSP工程“bspimx6”导入读者本机RealEvo-IDE中进行编译，编译后会根据编译选项在Debug或Release目录生成“bspimx6.bin”文件，此文件即为需要上传到验证平台的系统镜像文件。在部署SylixOS操作的时候需要用到两个工具软件“putty”和“tftpd32”。二者均在图 1.1中的“Tools”目录下与RealEvo-IDE一起提供。

SylixOS系统部署的详细步骤：

1. 将“Tools”目录下的“tftpd32.exe”软件拷贝到与“bspimx6.bin”文件同一目录下，双击打开，“tftpd32”会自动设置当前目录。此工具的作用是在当前主机上建立一个TFTP服务器，用来传输系统镜像文件（bspimx6.bin）。设置的“当前目录”是需要传输的系统镜像文件所在目录，“服务器地址”是本机与验证平台相连的网口对应IP地址，如果主机有多个网口，需要注意选择正确的IP地址。

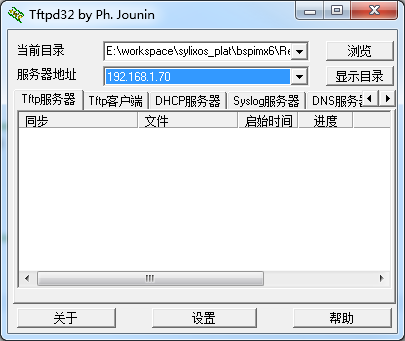


图 1.1 tftpd32启动后界面

1. 将“Tools”目录下的“putty.zip”文件拷贝到本机磁盘的任意一个目录下，解压文件，双击其中的“PUTTY.EXE”文件，打开putty软件；
2. 在左侧“Category”栏里面选择“Serial”，切换到“PuTTY Configuration”配置界面，如图 1.2所示，串口号需要根据本机情况填写，其他参数参考图 1.2设置即可；

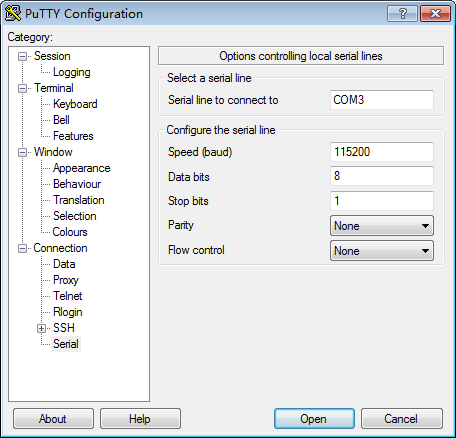


图 1.2 PuTTY串口通信参数配置界面

1. 点击“Session”选项，在putty切换后的界面中选择“Serial”复选框，单击“Open”打开putty控制台界面（文章后续简称控制台）；

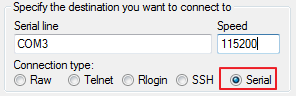


图 1.3 选择 Serial 复选框

1. 将主机的串口与验证平台的调试串口（与VGA相临的DB9）相连。随后给验证平台上电或复位，控制台会输出bootloader启动信息，在启动时敲击键盘的任意按键，会进入uboot的命令行界面；如图1.4.

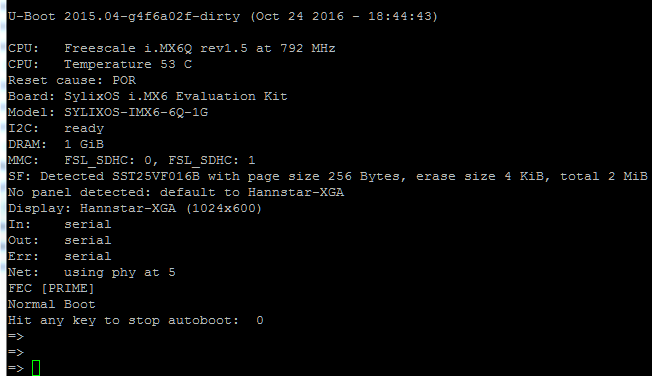


图 1.4 进入Uboot界面

1. 在控制台中输入“printenv”命令，会显示当前uboot中的相关变量配置，这里对我们有用的是网络配置相关内容，输出如下：

ipaddr=192.168.1.58

serverip=192.168.1.70

不同的PC主机的IP地址会不相同，这里需要根据读者主机的IP地址变化进行更改，更改命令如下：

setenv bootdelay 1 /\* 设置启动延时 \*/

setenv loadaddr 0x10000000 /\* 设置加载地址 \*/

setenv ipaddr 192.168.1.58 /\* 设置目标板 IP 地址 \*/

setenv serverip 192.168.1.70 /\* 设置主机的 IP 地址 \*/

setenv bootfile bspimx6.bin /\* 设置tftp下载的镜像名 \*/

setenv bootcmd 'tftp; dcache flush; dcache off; icache flush; icache off; go 0x10000000'

saveenv /\* 保存环境变量 \*/

注：执行saveenv后每次重启都会重新加载镜像，方便调试。如果不执行saveenv命令，每次加载镜像需重新输入以上命令。

1. 输入boot命令，或软重启开发板进行镜像的下载并启动。
2. 若上述参数配置无误，网线连接正常，镜像会下载到验证平台中。下载过程中会输出一串“#”提示下载进度；如图1.5所示。

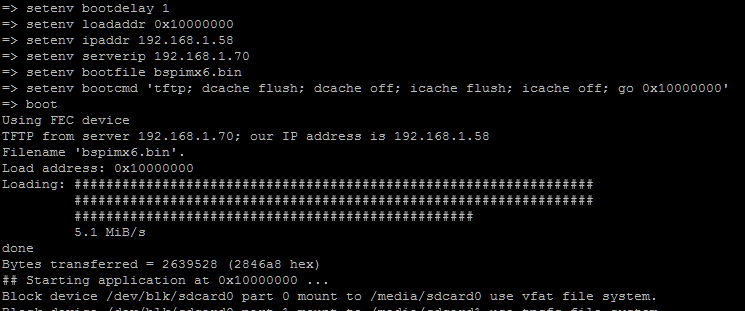


图 1.5 下载镜像并运行

1. 系统固化

⑴进入系统后把bspimx6.bin镜像拷贝到/boot目录，然后执行sync操作；

⑵重启进入uboot配置环境变量从emmc启动，如图1.6.

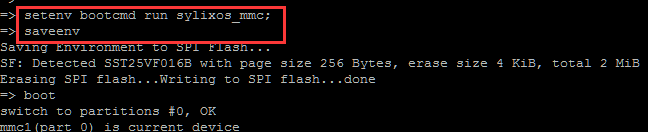


图 1.6 设置环境变量

1. Shell命令说明

**系统命令**

命令名 简要说明

help 显示ttinyShell所有内建命令列表，如果带有参数，则可显示以参数为名的命令介绍，例如help ls将介绍ls命令的介绍

free 查看系统当前内存信息

echo 回显命令，此命令将回显用户输入的参数

ts 查看系统中线程信息

tp 查看系统中被阻塞的线程信息

ss 查看系统中所有线程与中断系统堆栈使用情况

ps 查看系统所有进程的信息

ints 查看系统中断向量表信息

mems 查看操作系统内核内存堆与系统内存堆内存使用情况

zones 查看操作系统物理页面分区使用情况

env 查看操作系统全局环境变量表

varsave 保存当前的操作系统环境变量表。默认保存路径为/etc/profile

varload 从指定参数的文件中提取装载环境变量表，默认则从/etc/profile中提取

vardel 删除一个指定的系统环境变量

cpuus 测算当前cpu利用率

kill 向指定的线程或进程发送信号，默认发送SIGKILL信号

drvlics 显示系统中所有安装的驱动程序表

devs 显示系统中挂载的所有设备

buss 系统中挂载的所有总线信息

tty 显示当前shell终端对应的tty文件

clear 清除当前屏幕

aborts 显示当前操作系统异常处理统计信息

sprio 设置指定线程的优先级

renice 设置指定进程的优先级

hostname 显示或设置当前SylixOS镜像主机名

login 切换用户，重新登录

who 查看当前登录用户身份

shutdown 关闭或重启系统

monitor 启动、关闭或设置内核跟踪器

lspci 打印系统PCI总线与PCI设备相关信息

lsusb 打印系统USB总线与USB设备相关信息

which 检查参数指定的文件位置

exit 退出当前shell终端

**文件命令**

命令名 简要说明

ls 列出指定目录下的文件，默认为当前目录

ll 列出指定目录下的文件详细信息，默认为当前目录

files 列出系统内核中打开的文件信息（不包含进程打开的文件）

fdentrys 列出操作系统所有正在操作的文件信息（包含进程打开的文件）

sync 将所有系统缓存的文件、设备、磁盘信息全部写入到相应的物理设备中

logfileadd 向内核日志打印函数加入指定的内核文件描述符

logfileclear从内核日志打印文件表中清除指定的内核文件描述符

logfiles 显示内核日志打印文件列表

loglevel 显示或设置当前内核日志打印等级

cd 切换当前目录

pwd 查看当前工作目

df 查看指定目录的文件系统信息，默认为当前目录

tmpname 获得一个可以创建的临时文件名

mkdir 创建一个目录

mkfifo 创建一个命名管道，注意：只能在根文件系统设备下创建

rmdir 删除一个目录

rm 删除一个文件

mv 移动或重命名一个文件

cat 查看一个文件的内容

cp 拷贝一个文件

cmp 比较一个文件

dsize 计算一个指定的目录包含的所有文件信息

chmod 设置文件或目录的权限位

mkfs 格式化指定的磁盘

shfile 执行指定的shell脚本

mount 挂载一个卷

umount 卸载一个卷

showmount 查看系统中所有已经挂载的卷

ln 创建符号链接文件

dosfslabel 查看fat文件系统卷标

fatugid 设置fat文件系统用户与组id

mmaps 显示系统mmap信息

**网络命令**

命令名 简要说明

route 添加、删除、修改或查看系统路由表

netstat 查看网络状态

ifconfig 配置网络接口信息

ifup 使能一个网络接口

ifdown 禁能一个网络接口

ifrouter 设置默认路由接口

arp 添加、删除或查看ARP表

ping Ping命令

ping6 IPv6 Ping命令

tftpdpath 查看或设置tftp服务器本地路径

tftp 使用tftp命令接收或者发送一个文件

ftpds 显示ftp服务器信息

ftpdpath 查看或设置ftp服务器初始化路径

nat 启动、关闭或设置NAT虚拟网络地址服务

nats 查看当前NAT虚拟地址服务状态

npfs 查看网络数据包过滤器状态

npfruleadd 添加一条网络数据包滤波器规则

npfruledel 删除一条网络数据包过滤器规则

npfattach 在指定网络接口上使能网络数据包过滤器

npfdetach 在指定网络接口上禁能网络数据包过滤器

**时间命令**

命令名 简要说明

date 显示或设置系统时间

hwclock 显示或同步操作系统与硬件RTC时钟

**动态装载命令**

命令名 简要说明

debug 调试一个进程

dlconfig 配置动态链接器工作参数

modulereg 注册一个内核模块

moduleunreg 卸载一个内核模块

modulestat 查看一个内核模块或动态链接库文件信息

lsmod 查看系统装载的所有内核模块信息

modules 查看系统装载的所有内核模块与进程动态链接库信息

**其他命令**

命令名 简要说明

shstack 显示或者设置shell任务堆栈大小，设置仅对之后启动的shell有效

leakchkstart启动系统内存泄露跟踪器

leakchkstop 关闭系统内存泄露跟踪器

leakchk 内存泄露检查

xmodems 使用xmodem协议发送一个文件

xmodemr 使用xmodem协议接收一个文件

untar 解包或解压缩一个tar或tar.gz文件包

gzip 压缩或解压缩一个文件

vi 启动vi编辑器