实验五 ZLG7290驱动编写实验（内核态）

实验步骤1 内容简介

* 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

实验步骤2 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

实验环境

硬件：装有Linux操作系统的开发板

软件：Ubuntu16/18/20，putty，ssh

实验原理：

1. **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1。

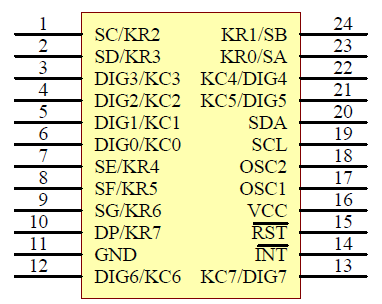


图 1 ZLG7290引脚

表 11.4说明了ZLG7290各引脚的功能。

|  |  |  |
| --- | --- | --- |
| 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 |

表1 引脚功能

下面两幅图显示了数码管与字母数字的对应关系。

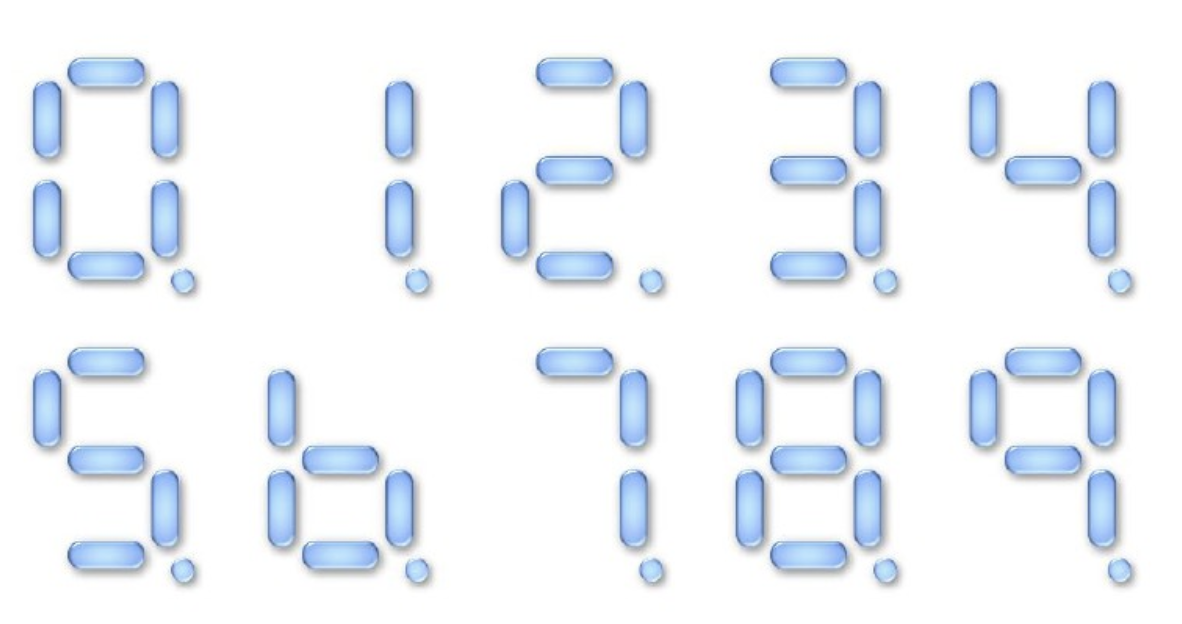


图2 数码管-数字对照图

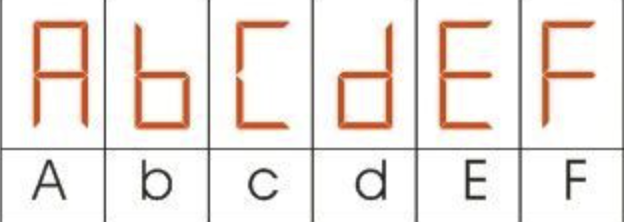


图3 数码管-字母对照图

1. **字符设备**

字符设备是指只能一个字节一个字节读写的设备，不能随机读取设备内存中的某一数据，读取数据需要按照先后数据。字符设备是面向流的设备，常见的字符设备有鼠标、键盘、串口、控制台和LED设备等。本次实验就是编写一个字符设备的驱动程序。下图是字符设备、字符设备驱动与用户空间访问该设备的程序三者之间的关系。

图示

描述已自动生成

图4 字符设备、字符设备驱动与用户空间访问该设备的程序

1. **设备模型**

设备模型由 总线（bus\_type） + 设备（device） + 驱动（device\_driver） 组成，在该模型下，所有的设备通过总线连接起来，即使有些设备没有连接到一根物理总线上，linux为其设置了一个内部的、虚拟的platform总线，用以维持总线、驱动、设备的关系。

因此，对于实现一个linux下的设备驱动，可以划分为两大步：

1. 设备注册
2. 驱动注册

本次实验需要实现的是基于i2c总线的zlg7290字符设备的驱动程序。i2c设备驱动中有以下常用的数据结构：

**i2c\_adapter**：每个i2c\_adapter对应一个物理上的i2c控制器，在i2c总线驱动probe 函数中动态创建，通过i2c\_add\_adapter注册到i2c\_core。

**i2c\_client**：代表一个挂载到i2c总线上的i2c从设备，它包含该设备所需要的数据：

该i2c从设备所依附的i2c控制器 struct i2c\_adapter \*adapter；

该i2c从设备的驱动程序struct **i2c\_driver** \*driver；

该i2c从设备的访问地址unsigned short addr；

该i2c从设备的名称name；

1. **驱动加载的流程**

当用户在终端执行insmod指令的时候，系统会调用module\_init()所绑定的初始化函数，在该模块初始化函数中，会对设备和驱动进行注册。如果能成功完成注册，将顺序执行到init函数中编写的业务逻辑模块。当用户在终端执行rmmod指令，系统会调用module\_exit()所绑定的模块退出函数，在该函数中需要注销之前的设备和驱动。

1. **驱动代码**

阅读ZLG7290数码管驱动的代码程序清单1.1，了解其实现的具体方法。

**程序清单1.1 myzr\_zlg7290.c**

/\*

\* BUAA LDMC

\*/

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/i2c.h>

#include <linux/input.h>

#include <linux/delay.h>

#include <linux/slab.h>

#include <linux/interrupt.h>

#include <linux/irq.h>

#include <linux/gpio.h>

#include <linux/fs.h>

#include <linux/cdev.h>

#include <linux/platform\_device.h>

#include <linux/module.h>

#include <linux/cdev.h>

#include <linux/fs.h>

#include <linux/poll.h>

#include <linux/sched.h>

#define ZLG7290\_NAME "zlg7290"

#define ZLG7290\_LED\_NAME "zlg7290\_led"

#define REG\_DP\_RAM0 0x10 //第一个数码管的寄存器

#define REG\_DP\_RAM1 0x11 //第二个数码管的寄存器

#define REG\_DP\_RAM2 0x12 //第三个数码管的寄存器

#define REG\_DP\_RAM3 0x13 //第四个数码管的寄存器

#define REG\_DP\_RAM4 0x14

#define REG\_DP\_RAM5 0x15

#define REG\_DP\_RAM6 0x16

#define REG\_DP\_RAM7 0x17

#define ZLG7290\_LED\_MAJOR 800 //主设备号

#define ZLG7290\_LED\_MINOR 0 //从设备号

#define ZLG7290\_LED\_DEVICES 1 //设备数量

#define WRITE\_DPRAM \_IO('Z', 0)

// 数码管设备结构体

struct zlg7290

{

struct i2c\_client \*client;

struct input\_dev \*input;

struct delayed\_work work;

unsigned long delay;

struct cdev cdev;

};

// 数码管设备结构体全局实例

struct zlg7290 \*ptr\_zlg7290;

// 数码管位图，每一个bit代表一根灯管，置1:亮，置0:灭

unsigned char bit\_map[8] = {

0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01

};

// 数码管显示0-F 16进制字符的位图数据

//Todo： 例如：0xfc 代表 “0”，0x0c 代表 “1”

unsigned char hex\_map[16] = {

0xfc, 0x0c, // 补全

};

// 向i2c总线写数据

// 考点：分析各个参数取值的含义

/\* 参数：

\* zlg7290 ：数码管结构体指针

\* len ：待写入的字节数（以byte为单位）

\* retlen ： 返回写入的字节数（read函数中使用）（注意是个指针）

\* buf： 待写入数据指针（寄存器地址和待写数据）

\*/

static int zlg7290\_hw\_write(struct zlg7290 \*zlg7290, int len, size\_t \*retlen, char \*buf)

{

struct i2c\_client \*client = zlg7290->client;

int ret;

//i2c\_msg

/\* 参数：

\* client->addr : 芯片地址

\* 0 ：flag位，0代表write，I2C\_M\_RD代表read

\* len ： 待写入的字节数

\* buf ： 待写入数据指针（寄存器地址和待写数据）

\*/

struct i2c\_msg msg[] = {

{ client->addr, 0, len, buf},

};

//Todo：调用i2c\_transfer函数向总线上写数据，注意参数

if (ret < 0)

{

dev\_err(&client->dev, "i2c write error!\n");

return -EIO;

}

\*retlen = len;

return 0;

}

// 从i2c总线读数据

static int zlg7290\_hw\_read(struct zlg7290 \*zlg7290, int len, size\_t \*retlen, char \*buf)

{

//Todo：仿照zlg7290\_hw\_write写出read函数

/\*步骤：

\*1. 构造i2c\_msg msg[]（由于read需要两次通信，因此需要构造两次参数的值）

第一个i2c\_msg用来发送需要读取的从设备目标寄存器的地址，buf保存待读寄存器的地址

第二个i2c\_msg用来设置接收从设备目标寄存器返回数据的buf以及数据长度，buf保存从寄存器内读到的值的地址

\*2. 调用i2c\_transfer(注意参数如何填写)

\*3. 错误处理

\*/

}

// 打开数码管设备

static int zlg7290\_led\_open(struct inode \*inode, struct file \*file)

{

return 0;

}

// 释放数码管设备

static int zlg7290\_led\_release(struct inode \*inode, struct file \*file)

{

return 0;

}

// 向上的接口，用户态通过调用ioctl系统函数，隐式调用该函数

/\* 参数：

\* filp ：文件结构体，用户态为文件描述符

\* cmd：

\* arg： 待写数据（long在系统中占8bytes，对应8个寄存器，arg高位对应低地址寄存器）

\*/

static long zlg7290\_led\_ioctl(struct file \*filp, unsigned int cmd, unsigned long arg)

{

return 0;

}

// file\_operations: 把系统调用和驱动程序关联起来的关键数据结构

static struct file\_operations zlg7290\_led\_fops = {

.owner = THIS\_MODULE,

.open = zlg7290\_led\_open,

.release = zlg7290\_led\_release,

.unlocked\_ioctl = zlg7290\_led\_ioctl,

};

// 注册数码管小灯设备主设备号：MAJOR，从设备号：MINOR

static int register\_zlg7290\_led(struct zlg7290 \*zlg7290)

{

struct cdev \*zlg7290\_cdev;

int ret;

dev\_t devid;

devid = MKDEV(ZLG7290\_LED\_MAJOR, ZLG7290\_LED\_MINOR);

ret = register\_chrdev\_region(devid, ZLG7290\_LED\_DEVICES, ZLG7290\_LED\_NAME);

if (ret < 0) {

dev\_err(&zlg7290->client->dev, "register chrdev fail!\n");

return ret;

}

zlg7290\_cdev = &zlg7290->cdev;

cdev\_init(zlg7290\_cdev, &zlg7290\_led\_fops);

zlg7290\_cdev->owner = THIS\_MODULE;

ret = cdev\_add(zlg7290\_cdev, devid, 1);

if (ret < 0) {

dev\_err(&zlg7290->client->dev, "cdev add fail!\n");

goto err\_unreg\_chrdev;

}

return 0;

err\_unreg\_chrdev:

unregister\_chrdev\_region(devid, ZLG7290\_LED\_DEVICES);

return ret;

}

// 从系统中注销设备

static int unregister\_zlg7290\_led(struct zlg7290 \*zlg7290)

{

cdev\_del(&zlg7290->cdev);

unregister\_chrdev\_region(MKDEV(ZLG7290\_LED\_MAJOR, ZLG7290\_LED\_MINOR), ZLG7290\_LED\_DEVICES);

return 0;

}

// 探测函数，如果驱动匹配到了目标设备，总线会自动回调probe函数

static int

zlg7290\_probe(struct i2c\_client \*client, const struct i2c\_device\_id \*id)

{

struct input\_dev \*input\_dev;

int ret = 0;

if (!i2c\_check\_functionality(client->adapter, I2C\_FUNC\_SMBUS\_BYTE)) {

dev\_err(&client->dev, "%s adapter not supported\n",

dev\_driver\_string(&client->adapter->dev));

return -ENODEV;

}

ptr\_zlg7290 = kzalloc(sizeof(struct zlg7290), GFP\_KERNEL);

input\_dev = input\_allocate\_device();

if (!ptr\_zlg7290 || !input\_dev) {

ret = -ENOMEM;

goto err\_free\_mem;

}

input\_dev->name = client->name;

input\_dev->phys = "zlg7290-keys/input0";

input\_dev->dev.parent = &client->dev;

input\_dev->id.bustype = BUS\_I2C;

input\_dev->id.vendor = 0x0001;

input\_dev->id.product = 0x0001;

input\_dev->id.version = 0x0001;

input\_dev->evbit[0] = BIT\_MASK(EV\_KEY);

ret = input\_register\_device(input\_dev);

if (ret) {

dev\_err(&client->dev, "unable to register input device\n");

goto err\_unreg\_dev;

}

ptr\_zlg7290->client = client;

ptr\_zlg7290->input = input\_dev;

i2c\_set\_clientdata(client, ptr\_zlg7290);

ret = register\_zlg7290\_led(ptr\_zlg7290);

if (ret < 0)

return ret;

printk("probe done! \n");

return 0;

err\_unreg\_dev:

input\_unregister\_device(input\_dev);

input\_dev = NULL;

err\_free\_mem:

input\_free\_device(input\_dev);

kfree(ptr\_zlg7290);

return ret;

}

// 释放函数，如果匹配到的设备从总线移除了，总线会自动回调remove函数

static int zlg7290\_remove(struct i2c\_client \*client)

{

struct zlg7290 \*zlg7290 = i2c\_get\_clientdata(client);

unregister\_zlg7290\_led(zlg7290);

printk("remove done! \n");

return 0;

}

// 匹配设备和驱动

// 考点：为什么需要i2c\_device\_id？

// 提示：一个设备只有一个驱动，一个驱动可以对应多个设备

static const struct i2c\_device\_id zlg7290\_id[] = {

{ZLG7290\_NAME, 0 },

{ }

};

MODULE\_DEVICE\_TABLE(i2c, zlg7290\_id);

#ifdef CONFIG\_OF

static const struct of\_device\_id zlg7290\_dt\_ids[] = {

{ .compatible = "myzr,zlg7290", },

{ }

};

MODULE\_DEVICE\_TABLE(of, zlg7290\_dt\_ids);

#endif

// 驱动程序结构体（重要）

// 考点：请分析init，probe，remove和exit函数的执行顺序

// 提示：可通过printk来分析

static struct i2c\_driver zlg7290\_driver= {

.driver = {

.name = ZLG7290\_NAME,

.owner = THIS\_MODULE,

.of\_match\_table = of\_match\_ptr(zlg7290\_dt\_ids),

},

.probe = zlg7290\_probe,

.remove = zlg7290\_remove,

.id\_table = zlg7290\_id

};

// 初始化模块

static int \_\_init zz\_init(void)

{

//module\_i2c\_driver宏展开

i2c\_register\_driver(THIS\_MODULE, &zlg7290\_driver);

//内核态i2c\_transfer通信

//Todo：保持8个数码管内容相同（只使用到4个），循环显示bit\_map和hex\_map中的所有字符

//提示： 调用zlg7290\_hw\_write函数

int i;

int j;

ssize\_t len = 0;

unsigned char write\_val[2] = {0};

/\* write\_val[0] : 待写入寄存器的地址

\* write\_val[1] : 待写入寄存器的数据

\* 例如：write\_val[0] = REG\_DP\_RAM0; //写0号寄存器

\* write\_val[1] = 0xfc; //写入字符“0”

\*/

for(i=0; i<8; i++){ //外层循环：bit\_map的长度

for(j = 0; j < 8; j++){// 内层循环 ： 8个寄存器对应8个数码管，开发板上只使用到了低4位寄存器

//Todo：

msleep(1); //睡眠1毫秒

}

msleep(1000); //睡眠1秒

}

for(i=0; i<16; i++){ //外层循环：hex\_map的长度

for(j = 0; j < 8; j++){// 内层循环 ： 8个寄存器对应8个数码管，开发板上只使用到了低4位寄存器

//Todo：

msleep(1); //睡眠1毫秒

}

msleep(1000); //睡眠1秒

}

for(j = 0; j < 8; j++){

//Todo： 最后要将所有数码管置零

msleep(1);

}

return 0;

}

// 卸载模块

static void \_\_exit zz\_exit(void)

{

i2c\_del\_driver(&zlg7290\_driver);

printk("886, The module has been removed\n");

}

module\_init(zz\_init);

module\_exit(zz\_exit);

MODULE\_AUTHOR("buaa ldmc");

MODULE\_DESCRIPTION("Keypad & Leds driver for ZLG7290");

MODULE\_LICENSE("GPL");

* **基础实验：**

**补全myzr\_zlg7290.c代码，实现内核态的数码管亮灭，包括zlg\_hw\_write函数、bit\_map[]、hex\_map[]、以及zz\_init函数中的逻辑，不包含zlg7290\_hw\_read函数。**

**实验操作：**

**使用Makefile编译驱动模块，将生成的.ko文件上传到开发板，加载数码管驱动myzr\_zlg7290.ko，加载成功后会有相应输出，并观察到数码管的亮灭。**

**root@myimx6ek314: /mnt/app# insmod myzr\_zlg7290.ko**

**intput: zlg7290 as /devices/soc0/soc.0/2100000.aips-bus/21a4000.i2c/i2c-1/1-0038/input/input2**

* **扩展实验：**

**补全zlg7290\_hw\_read函数并调用，实现终端与数码管的同步输出。要求：**

1. **以字符的形式输出，如数码管显示‘C’，则终端输出‘C’。（5分）**
2. **遍历字母表。（5分）**

* **评分细则：**

1. **安装模块，观察小灯的亮灭。（70分）**
2. **完成扩展实验（5+5分）**
3. **参与度/答辩成绩（20分）**

* **答辩内容：**

1. **参考源代码中的考点部分**
2. **驱动程序中各函数的作用**