第一次实验成绩	第二次实验成绩	实验总成绩

哈爾濱工業大學（威海）

Harbin Institute of Technology at Weihai

《嵌入式系统原理》实验报告

(IMX6 嵌入式实验箱-计算机科学与技术)

姓　　名 蒲海博

学　　号 2021211041

班　　级 2104102

（请双面打印）

实验内容

第一次实验 嵌入式 LINUX 开发环境建立实验

(对应实验指导书第一章实验二、三、四、五)

实验一 U-Boot 移植与编译、Linux 内核移植与编译、根文件系统实验

实验二 烧写实验

第二次实验 嵌入式 LINUX 设备驱动程序实验

（对应实验指导书第二章实验六、七、八）

实验三 LED 驱动及控制实验

实验四 八段双数码管实验

实验五 点阵控制实验

注：在实验内容及步骤部分要提供截图或照片

实验一 U-Boot 移植与编译、Linux 内核移植与编译、根文件系统实验

同组人：蒲海博，聂硕，朱昱安　　日期:2024.5.18

1、实验内容及步骤

(1)u-boot 移植与编译

$ source /opt/poky/1.7/environment-setup-cortexa9hf-vfp-neon-poky-linux-gnueabi

$ make

注意：Linux 平时处于 x86 程序编译环境，前面的命令建立了编译 arm 程序的 环境，并且仅在当前终端窗口中生效。

/opt/poky/1.7/environment-setup-cortexa9hf-vfp-neon-poky-linux-gnue abi 为 imx6 自带脚本配置，主要内容如下:

export ARCH=arm //为 arm 架构

export CROSS_COMPILE=arm-poky-linux-gnueabi- //交叉编译器命令前缀 编译成功则会得到 u-boot.imx 文件。

在 arch/arm/imx-common/Makefile 文 件 中

可 以 看 到 u-boot.imx 由 u-boot.bin 生 成

(2)Linux 内核移植与编译

1、使用 vim 编辑器手动编写实验代码 helloworld.c

$ cd drivers/char

$ vim helloworld.c

helloworld.c 参考内容如下：

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL"); //驱动程序入口函数
static int hello_init(void) {
printk(KERN_ALERT"###############Hello,
2021211040ns,2021211041-phb,2021211054-zya############\n");
 return 0;
}
//驱动程序出口函数 static void hello_exit(void) {
 printk(KERN_ALERT "###############Goodbye, world#########\n");
return 0;
}
module_init(hello_init);
module_exit(hello_exit);
```

对程序做更改，在输出信息中加入自己的学号和英文名字，验证时使用。

本实验只在编写简单的驱动程序并加入到 Linux 内核目录树中。该驱动程序是向终端输出相关程序信息。

2、进入实验内核源码目录修改 drivers/char/目录下的 Kconfig 文件，按照 Kconfig

语法添加 helloworld 程序的菜单支持。

$ vim Kconfig

例如：在 Kconfig 文件中最后一行 endmenu 的前面添加如下：

```
config HELLO_MODULE
    bool "Hello World Test"
    help
        This is a demo to test kernel experiment On IMX6.
```

注意 config HELLO_MOULDE 段与前后段要有空格或换行隔开，且 bool 和 help 变量要与行开头有 TAB 符号位隔开。

3、进入实验内核源码目录修改 drivers/char/目录下的 Makefile 文件，按照内核中 Makefile 语法添加 helloworld 程序的编译支持：

$ vi Makefile

在 Makefile 中最后一行添加如下代码。

```
obj-$(CONFIG_HELLO_MODULE) += helloworld.o
```

4、进入内核源码主目录，运行 make menuconfig 配置内核对 helloworld 程序的支持。

$ make menuconfig

将 hello world test 静态编译进内核

5、查看内核的 emmc 支持

文件 drivers/mmc/core/mmc.c,保证第 297 行的值为 8

6、重新编译内核

在内核源码的顶层目录下编译内核

```
$ source /opt/poky/1.7/environment-setup-cortexa9hf-vfp-neon-poky-linux-gnueabi
$ make zImage
$ make imx6dl-sabresd.dtb
```

编译成功后会在源码目录的 arch/arm/boot/ 目录下生成内核压缩文件 zImage，在源码目录的 arch/arm/boot/dts/下生成设备树文件 imx6dl-sabresd.dtb

(3)根文件系统实验

实验目录： /home/uptech/fsl-6dl-source/rootfs

进入实验目录，查看文件系统。

```
$ cd /home/uptech/fsl-6dl-source/rootfs
$ ls rootfs.tar.bz2
$ mkdir rootfs
$ cd rootfs
$ tar -xjvf ../rootfs.tar.bz2
$ ls bin dev home media opt run sys unit_tests var boot etc lib mnt proc sbin tmp usr
```

当前 rootfs 目录就是文件系统，可以在此目录下创建自己的目录，注意不要和系统目录或文件冲突，这里创建一个 test 目录和 a.c。

```
$ mkdir test
$ touch test/a.c
$ ls bin dev home media opt run sys tmp usr boot etc lib mnt proc sbin test unit_tests
```

var

在 etc/init.d/rc.local 文件中添加一行定义目标机的 IP：

ifconfig eth0 192.168.50.100 netmask 255.255.255.0

最后重新删除原来的压缩包，并生成自己的根文件系统压缩包：

$ rm ../rootfs.tar.bz2 $ tar -cjvf ../rootfs.tar.bz2 *

2、遇到的问题及解决方法

(1)在 uptech 用户系统下没有权限无法访问环境变量

sudo -s 获得 root 权限

source /etc/bash.bashrc 获得环境变量

即可访问 source 资源

(2) 在 linux 内核编译过程中生成新的编译文件时出错

Vim 修改文件时直接关闭没有自动保存, wq 保存并退出

3、实验结论及分析

```
imx sema4 driver is registered.
#################Hello, 2021211040-ns, 2021211041-phb, 2021211054-zya#############
[drm] Initialized drm 1.1.0 20060810
[drm] Initialized vivante 1.0.0 20120216 on minor 0
brd: module loaded
loop: module loaded
at24 0-0051: 128 byte 24c01 EEPROM, writable, 1 bytes/write
```

在 Linux 系统中，引导加载程序（Bootloader）、内核（Kernel）和根文件系统（Root Filesystem）是启动和运行操作系统所需的三个基本组件。

这次实验成功设置了交叉编译环境，并使用该环境编译了 U-Boot。最终生成了 u-boot.imx 文件。交叉编译环境的配置至关重要，确保了编译工具链和目标架构的一致性。u-boot.imx 是从 u-boot.bin 生成的，表明 U-Boot 编译过程顺利，符合预期。

成功编写并编译了一个简单的 Linux 内核模块 helloworld，并将其集成到内核中。编译后的内核文件 zImage 和设备树文件 imx6dl-sabresd.dtb 生成成功。

编写内核模块并配置内核是一项基础且重要的技能，掌握后有助于理解内核模块加载和设备驱动程序的开发。通过 make menuconfig 配置内核选项，确保了新模块的正确集成。内核重新编译过程顺利，验证了环境配置和编译过程的正确性。

成功解压、修改并重新打包了根文件系统。新的根文件系统包含了自定义的目录和配置文件，验证了对文件系统的操作权限。根文件系统的解压和打包操作验证了对文件系统内容的修改能力。添加自定义目录和文件，确保系统目录不冲突，是系统定制的重要步骤。修改 rc.local 文件来配置网络设置，可以使系统在启动时自动配置网络，增强了系统的可用性。

通过这些实验，掌握了 U-Boot 的移植与编译、Linux 内核模块的编写与集成、以及根文件系统的修改与重打包的基本技能。这些技能是嵌入式系统开发中的核心内容，为进一步开发复杂的嵌入式系统奠定了坚实的基础。实验过程中遇到的问题及其解决方法，进一步强化了对环境配置和编译流程的理解。

实验二 烧写实验

同组人：蒲海博，聂硕，朱昱安　　日期:2024.5.18

1、实验内容及步骤

1 准备好烧写文件

要烧写的文件已经在前面 3 个实验中生成，包括编译生成的 u-boot 镜像文件 u-boot-imx6dlsabresd_sd.imx（u-boot.imx 改名）、编译内核生成的镜像文件 zImage 和设备树文件 zImage-imx6dl-sabresd.dtb（imx6dl-sabresd.dtb 改名）、根文件系统的压缩文件 rootfs.tar.bz2。

2 拨码开关：

此处的拨码开关位于核心板上，on 表示 1 off 表示 0 ，有两种模式：

（1）USB OTG 模式 :bit[1:8] = 0 0 0 0 1 1 0 0

（2）EMMC 模式 :bit[1:8] = 1 1 0 1 0 1 1 0

烧写程序时需要按照 USB OTG 的方式拨动拨码开关，正常运行要处于 EMMC 模式。

3 正确连接 mini usb 烧写线

将 USB 线一端插入 PC 的 USB 接口，另一端接到 imx6 平台的核心板上的 USB OTG 接口。

给 IMX6 平台插上 12V DC 电源，上电开机。

4 运行 mfgtools 软件就行烧写

将已准备好的 4 个要烧写文件拷贝到要烧写烧写工具 mfgtools 下的目录 mfgtools\Profiles\Linux\OS Firmware\files 中

进入烧写工具主目录：

双击 mfgtool2-yocto-mx6-sabresd-emmc

单击 start，等待烧写完毕单击 stop 即可完成。

5 烧写完成，拨码启动 程序烧写大约需要 10 分钟，烧写完毕，按“stop”并退出，断电关机，按照 EMMC 模式拨动拨码开关，上电即可验收烧写系统的正确性。

烧写成功后启动嵌入式系统教学科研平台(IMX6 核心)部分系统，可以在串口终端中查看到 Linux 内核在启动过程中打印出来的如下信息

```
Serial: imx driver
2020000.serial: ttymxc0 at MMIO 0x2020000 (irq = 58, base_baud = 5000000) is a IMX
console [ttymxc0] enabled
21e8000.serial: ttymxc1 at MMIO 0x21e8000 (irq = 59, base_baud = 5000000) is a IMX
21f0000.serial: ttymxc3 at MMIO 0x21f0000 (irq = 61, base_baud = 5000000) is a IMX
21f4000.serial: ttymxc4 at MMIO 0x21f4000 (irq = 62, base_baud = 5000000) is a IMX
serial: Freescale lpuart driver
##############Hello, world############
```

ppdev: user-space parallel port driver
s3c-uart.0: ttySAC0 at MMIO 0x7f005000 (irq = 37) is a S3C
s3c-uart.1: ttySAC1 at MMIO 0x7f005400 (irq = 38) is a S3C
s3c-uart.2: ttySAC2 at MMIO 0x7f005800 (irq = 39) is a S3C
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
loop: loaded (max 8 devices)
nbd: registered device at major 43
dm9000 Ethernet Driver
eth0: dm9000 at c7866000,c7866002 IRQ 78 MAC: 00:22:12:34:56:90
S3C IrDA driver, (c) 2006 Samsung Electronics
Linux video capture interface: v2.00 ……
……
同样进入 **ARM** 系统后也可以通过 **dmesg** 命令查看内核启动信息。
 # dmesg ……
……
S3C_LCD clock got enabled :: 133.000 Mhz
Window[0]- FB1 : map_video_memory: clear ff600000:00096000
FB1 : map_video_memory: dma=57100000 cpu=ff600000 size=00096000
Console: switching to colour frame buffer device 80x30
fb-1069494052: frame buffer device
lp: driver loaded but no devices found
####################Hello, world####################
kzkuan___gpio_leds_init
kzkuan___gpio_leds_probe
ledtestleds initialized
kzkuan___gpio_uart485_init
kzkuan___gpio_uart485_probe
kzkuan___gpio_bt_init
imx sema4 driver is registered.
[drm] Initialized drm 1.1.0 20060810
[drm] Initialized vivante 1.0.0 20120216 on minor 0
brd: module loaded
loop: module loaded
at24 0-0051: 128 byte 24c01 EEPROM, writable, 1 bytes/write
OF: no ranges; cannot translate
…... ……
2、遇到的问题及解决方法
(1)在开启虚拟机时由于接口连接上的虚拟机无法进行烧写
选择连接主机即可继续进行
(2)在超级终端无法输入 **root** 查看文件
串口设置出错，不应选择硬件而应选择无

3、实验结论及分析

```
Poky (Yocto Project Reference Distro) 1.7 imx6dlsabresd /dev/ttymxc0

imx6dlsabresd login: root
root@imx6dlsabresd:~# ls
imx6_V1_0
root@imx6dlsabresd:~# ls /home/
adfonts  root
root@imx6dlsabresd:~# ls /
bin   dev  home  lost+found  mnt  proc  sbin  test  unit_tests  var
boot  etc  lib   media       opt  run   sys   tmp   usr
root@imx6dlsabresd:~# _
```

成功编译并烧写了 **Linux** 内核，系统启动正常。根文件系统生成的 **test** 文件烧写结果正确，系统启动后能够正确识别并使用。

本次实验展示了通过 **USB OTG** 模式进行嵌入式系统烧写的方法，这是嵌入式系统开发中常见的一种方式。

通过实践，体会到了烧写嵌入式系统的一些关键步骤和注意事项，例如正确设置拨码开关、配置串口等。

老师还介绍了通过 **SD** 卡进行烧写的另一种方法，拓展了我们对嵌入式系统烧写方式的理解。

通过这次实验，我们不仅掌握了烧写嵌入式 **Linux** 系统的基本方法，还对嵌入式系统的设计和开发有了更深的认识。这些经验将对今后的嵌入式开发工作有很大的帮助。

实验三 LED 驱动及控制实验

同组人：蒲海博，聂硕，朱昱安　　日期:2024.5.26

1、实验内容及步骤

实验目录：

/home/uptech/fsl-6dl-sabresd/kernel-3.14.28（内核目录）

/ home/uptech/exp/diver/02_leds/ （应用程序目录）

首先在内核中添加 LED 设备模块驱动并烧写至实验箱 （注意内核实际上 已经按要求配置并烧写到目标机了，不需要重新烧），步骤如下：

1、配置内核：

$ cd /home/uptech/fsl-6dl-source/kernel-3.14.28

$ make menuconfig Device Driver--> Character devices--> [*]IMX6 leds test

2、进入宿主机中 IMX6 型光盘内核目录：

$ source /opt/poky/1.7/environment-setup-cortexa9hf-vfp-neon-poky-linux-gnueabi

3、编译项目 $ make $ make imx6dl-sabresd.dtb 编译成功后会在源码目录的 arch/arm/boot/ 目录下生成内核压缩文件 zImage，在源码目录的 arch/arm/boot/dts/下生成设备树文件 imx6dl-sabresd.dtb。

用前面讲述的方法更新内核，这里不在赘述。

编译 LED 应用测试程序

1、进入实验目录： $ cd /home/uptech/exp/driver/02_led/

2、清除中间代码，重新编译

$ source /opt/poky/1.7/environment-setup-cortexa9hf-vfp-neon-poky-linux-gnueabi

$ make clean

$ make

当前目录下生成可执行程序 ledtest

NFS 挂载实验目录测试

1、启动 IMX6 型实验系统，连好网线、串口线。通过串口终端挂载宿主机实 验目录。 （此时主机 IP 应为 192.168.50.128，目标机 IP 应为 192.168.50.100）

 # mount -t nfs 192.168.50.128:/home/uptech /mnt/

2、进入串口终端的 NFS 共享实验目录。 # cd /mnt/exp/driver/02_led/

3、执行应用程序测试该驱动及设备

 # ./ledtest /dev/ledtest 1 0

 # ./ledtest /dev/ledtest 1 1

 # ./ledtest /dev/ledtest 1 2

 # ./ledtest /dev/ledtest 1 3

2、遇到的问题及解决方法

(1)在 uptech 用户系统下没有权限无法访问环境变量

sudo -s 获得 root 权限

source /etc/bash.bashrc 获得环境变量

即可访问 source 资源

3、代码分析及实验结论

Linux 系统下，应用程序不可直接操作底层硬件寄存器，必须经过驱动层来完成对硬件的操作。

驱 动 程 序分析 ：

/home/uptech/fsl-6dl-source/kernel-3.14.28/drivers/char/imx6-leds.c

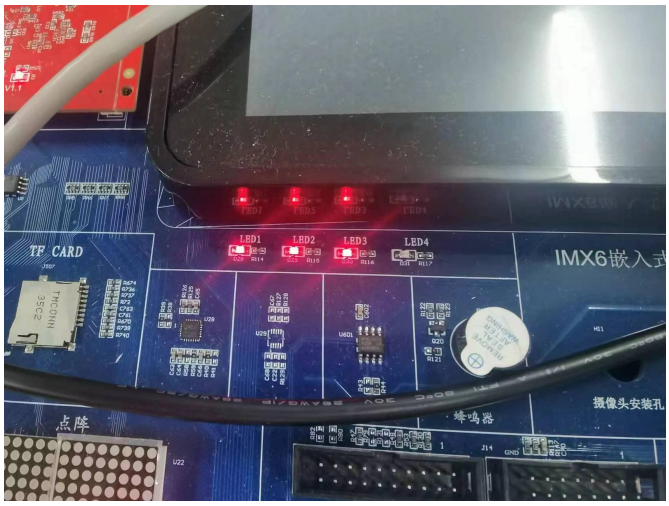
```
#define DEVICE_NAME "ledtest"//定义设备名称
#define DEVICE_MAJOR 231 //主设备号
#define DEVICE_MINOR 0 //次设备号
struct cdev *mycdev;//字符型设备指针
struct class *myclass;//自定义类
dev_t devno;
static unsigned int led_table [4] = {};//4 个 led 数组
static long uptech_leds_ioctl( //ioctl 给应用层使用，主要控制引脚的高低
struct file *file,
unsigned int cmd,
unsigned long arg)
{
switch(cmd) {
case 1:
if (arg < 0 || arg > 4) {//IOCTL 接参数
return -EINVAL;
}
gpio_request(led_table[arg],"ledCtrl");//注册一个引脚
gpio_direction_output(led_table[arg],0);//将引脚设置输出模式，并拉低
gpio_free(led_table[arg]);//释放一个引脚
break;
case 0:
if (arg < 0 || arg > 4) {
return -EINVAL;
}
gpio_request(led_table[arg],"ledCtrl");
gpio_direction_output(led_table[arg],1);//拉高
gpio_free(led_table[arg]);
break;
default:
return -EINVAL;
}
return 0;
}
/*led 结构体主要说明有哪些功能，这里有 ioctl 功能*/
static struct file_operations uptech_leds_fops = {
.owner = THIS_MODULE,
```

```
.unlocked_ioctl = uptech_leds_ioctl,
};
static int uptech_leds_init(void)//模块程序的初始化，注册字符设备
{
int err;
devno = MKDEV(DEVICE_MAJOR, DEVICE_MINOR);//注册一个设备号
mycdev = cdev_alloc();
cdev_init(mycdev, &uptech_leds_fops);//初始化
err = cdev_add(mycdev, devno, 1);//增加  char  字符
if (err != 0)
printk("Exynos4412 leds device register failed!\n");
myclass = class_create(THIS_MODULE, "ledtest");//创建一个设备文件
if (IS_ERR(myclass)) {
printk("Err: failed in creating class.\n");
return -1;
}
device_create(myclass,NULL,
MKDEV(DEVICE_MAJOR,DEVICE_MINOR), NULL,
DEVICE_NAME);
printk(DEVICE_NAME "leds initialized\n");
return 0;
}
static int gpio_leds_probe(struct platform_device *pdev)
//初始化各个引脚状态
{
unsigned int i;
struct device *dev = &pdev->dev;
struct device_node *of_node;
of_node = dev->of_node;
if (!of_node) {
return -ENODEV;
}
led_table[0] = of_get_named_gpio(of_node,"gpio0",0);
//获取设备树中  gpio0
led_table[1] = of_get_named_gpio(of_node,"gpio1",0);
led_table[2] = of_get_named_gpio(of_node,"gpio2",0);
led_table[3] = of_get_named_gpio(of_node,"gpio3",0);
if(!gpio_is_valid(led_table[0])||!gpio_is_valid(led_table[1])||!gpio_is_v
alid(led_table[2])||!gpio_is_valid(led_table[3]))
{
return -ENODEV;
}
```

```
for (i = 0; i < 5; i++) {
gpio_request(led_table[i],"ledCtrl");
gpio_direction_output(led_table[i],1);
gpio_free(led_table[i]);
}
printk("\n\n\nkzkuan___%s\n\n\n",__func__);
uptech_leds_init();
return 0;
}
static void uptech_leds_exit(void)//
class_destroy(myclass);
}
static int gpio_leds_remove(struct platform_device *pdev)//模块移除函数
{
uptech_leds_exit();
return 0;
}
/*匹配设备树中的信息*/
static struct of_device_id gpio_leds_of_match[] = {
{ .compatible = "fsl,gpio-leds-test", },
{ },
};
MODULE_DEVICE_TABLE(of, gpio_leds_of_match);
/*led  结构体*/
static struct platform_driver gpio_leds_device_driver = {
.probe = gpio_leds_probe,//probe  入口，指针函数
.remove = gpio_leds_remove,//移除入口
.driver = {
.name = "gpio-leds-test",
.owner = THIS_MODULE,
.of_match_table = of_match_ptr(gpio_leds_of_match),
}
};
static int __init gpio_leds_init(void)
{
return platform_driver_register(&gpio_leds_device_driver);
// 注册  platform  设备
}
static void __exit gpio_leds_exit(void)
{
platform_driver_unregister(&gpio_leds_device_driver);//删除设备
}
```

module_init(gpio_leds_init);//模块入口函数
module_exit(gpio_leds_exit);//模块出口函数
添加设备树文件：
在内核路径 arch/arm/boot/dts/imx6qdl-sabresd.dtsi
gpio-leds-test {
compatible = "fsl,gpio-leds-test";
//fsl 厂商，gpio-leds-test 匹配驱动文件
pinctrl-names = "default";//配置名称
pinctrl-0 = <&pinctrl_gpio_leds_test>;//引脚的配置，设置成 gpio
gpio0 = <&gpio6 7 0>;//我们要的引脚在设备树里声明，驱动会调用 gpio0
gpio1 = <&gpio6 15 0>;
gpio2 = <&gpio6 8 0>;
gpio3 = <&gpio6 10 0>;
};
……pinctrl_gpio_leds_test: gpio_ledsgrp {//设备树中引脚的配置
fsl,pins = <
MX6QDL_PAD_NANDF_CLE__GPIO6_IO07 0x80000000
//定义成 gpio 模式
MX6QDL_PAD_NANDF_CS2__GPIO6_IO15 0x80000000
MX6QDL_PAD_NANDF_ALE__GPIO6_IO08 0x80000000
MX6QDL_PAD_NANDF_RB0__GPIO6_IO10 0x80000000
>;
};
MX6QDL_PAD_NANDF_CLE__GPIO6_IO07 就是把 NANDF_CLE 配置成 GPIO6_IO7 功能，引脚复用。NANDF_CLE 其他功能的定义可以查看
arch/arm/boot/dts/IMX6DL-pinfunc.h，里面是 IMX6DL 引脚的所有配置功能。
应用程序分析：
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
int main(int argc ,char* argv[])//主函数
{
int fd;
int ret;
fd = open(argv[1],O_RDWR,0777);//打开设备文件
if(fd<0)
{
printf("open device %s err",argv[1]);
return -1;
}

```
ret =ioctl(fd,*argv[2]-'0',*argv[3]-'0');//传参，调用驱动里的 ioctl
if(ret <0)
{
prin("ioctl err\n");
return -1;
}
}
```



通过本次实验，成功编写并烧写了 LED 设备驱动，验证了驱动的正确性。

通过测试程序，成功控制了 LED 的点亮和熄灭，验证了驱动的功能。

实验过程中，掌握了在 Linux 系统下编写和调试驱动程序的基本方法，以及如何通过设备树配置硬件资源。

总体而言，本次实验达到了预期目标，为今后在嵌入式 Linux 系统上进行更复杂的驱动开发打下了坚实基础。

实验四 八段双数码管实验

同组人：蒲海博，聂硕，朱昱安　　日期:2024.5.26

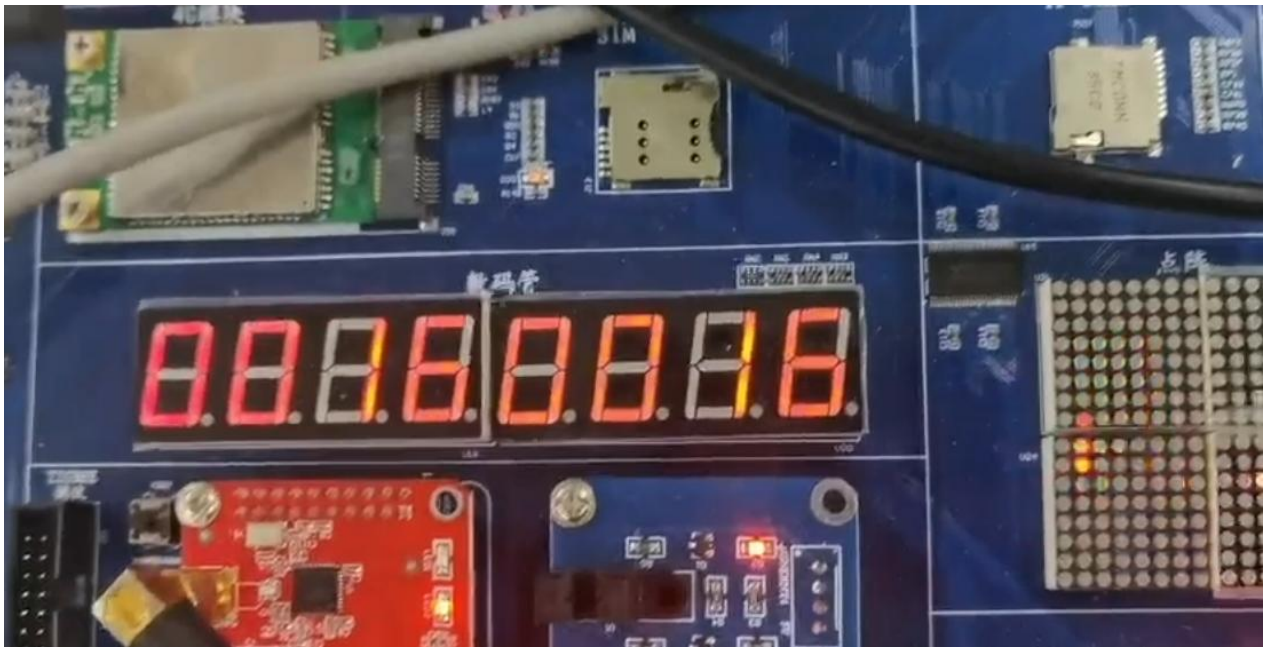
1、实验内容及步骤
实验目录：/home/uptech/exp/module/01_ledSegmentDisplays
主机 Linux 上编译应用测试程序
1、进入实验目录：
$ cd /home/uptech/exp/module/01_ledSegmentDisplays
$ ls
Makefile demo demo.c demo.o
2、清除中间代码，重新编译
$ source /opt/poky/1.7/environment-setup-cortexa9hf-vfp-neon-poky-linux-gnueabi
$ make clean
rm -f test_led *.elf *.gdb *.o
$ make
$ ls
Makefile demo demo.c demo.o
当前目录下生成可执行程序 demo。
目标机上 NFS 挂载实验目录并运行
1、启动 IMX6 实验系统， 连好网线、 串口线。 通过串口终端挂载宿主机实验目录。
 # mount -t nfs 192.168.50.128:/home/uptech /mnt/
2、进入串口终端的 NFS 共享实验目录。
 # cd /mnt/exp/module/01_ledSegmentDisplays/
 # ls
Makefile demo demo.c demo.o
3、执行应用程序测试该驱动及设备
 # ./demo
实验平台上的八字数码管将递增显示 0000-9999
注意：若实验平台上的启动时运行了/home/root/imx6_V1_0，会与实验程序冲突，请使用 ps -x 命令找到进程号，并使用 kill 命令杀死

2、遇到的问题及解决方法
(1)设备文件不可访问或不存在
确保设备文件/dev/mem 存在并有适当的访问权限
(2)物理地址映射失败
检查物理地址和映射参数，确保内存映射的正确性。

3、代码分析及实验结论
#include <stdio.h>
#include <fcntl.h>

```
#include <syspes.h>
#include <sys/time.h>
#include <unistd.h>
#include <string.h>
#include <sysan.h>
#include <stdlib.h>
unsigned char tube[] ={0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90,0x7f,
0xff};
unsigned char addr[] = {0x11,0x22,0x44,0x88};
int main(int argc, char *argv[]){
int num=0;
int i=0,j=0;
int mem_fd;
unsigned char *cpld;
//下面 2 条语句完成了物理地址 0x8000000 到逻辑地址的映射。
mem_fd = open("/dev/mem", O_RDWR);
cpld = (unsigned char*)mmap(NULL,(size_t)0x10,PROT_READ | PROT_WRITE |
PROT_EXEC,MAP_SHARED,mem_fd,(off_t)(0x8000000));
if(cpld == MAP_FAILED)
return;
while(1)
{
for(j=0;j<245*4;j++)
{
switch(i)
{
case 0:
*(cpld+(0xe6<<1)) = addr[i]; //数码管地址 (0xe6<<1)为地址
*(cpld+(0xe4<<1)) = tube[num%10]; //数码管个位 (0xe4<<1)为地址
break;
case 1:
*(cpld+(0xe6<<1)) = addr[i]; //数码管地址
41
*(cpld+(0xe4<<1)) = tube[(num%100)/10]; //数码管十位
break;
case 2:
*(cpld+(0xe6<<1)) = addr[i]; //数码管地址
*(cpld+(0xe4<<1)) = tube[(num%1000)/100]; //数码管百位 2
break;
case 3:
*(cpld+(0xe6<<1)) = addr[i]; //数码管地址
*(cpld+(0xe4<<1)) = tube[num/1000]; //数码管千位
```

```
break;
default:break;
}
usleep(1000);
if((++i)==4)
i=0;
}
if((++num) == 10000)
num = 0;
}
munmap(cpld,0x10);
```



实验成功地编写并执行了八字数码管驱动的测试程序。

程序能够正确地递增显示 0000 到 9999，验证了驱动程序和硬件接口的正确性。

通过本次实验，进一步理解了嵌入式系统中驱动程序和用户空间程序的交互，以及如何通过内存映射访问硬件资源，为更复杂的嵌入式开发奠定了基础。

实验五 点阵控制实验

同组人：蒲海博，聂硕，朱昱安　　日期:2024.5.26

1、实验内容及步骤

实验目录：/home/uptech/exp/module/02_Matrix

编译应用测试程序

1、进入实验目录：

$ cd /home/uptech/exp/module/02_Matrix/

$ ls

Makefile demo demo.c demo.o

2、清除中间代码，重新编译

$ source /opt/poky/1.7/environment-setup-cortexa9hf-vfp-neon-poky-linux-gnueabi

$ make clean

rm -f test_led *.elf *.gdb *.o

$ make

$ ls

Makefile demo demo.c demo.o d

当前目录下生成可执行程序 demo。 NFS 挂载实验目录测试

1、启动 IMX6 实验系统， 连好网线、 串口线。 通过串口终端挂载宿主机实验目录。

 # mount -t nfs 192.168.50.128:/home/uptech /mnt/

2、进入串口终端的 NFS 共享实验目录。

 # cd /mnt/exp/module/02_Matrix/

 # ls

Makefile demo demo.c demo.o

3、执行应用程序测试该驱动及设备

 # ./demo

实验平台上的点阵数码管将显示“恭喜发财”。**

2、遇到的问题及解决方法

(1)物理地址映射失败

检查物理地址和映射参数，确保内存映射的正确性。

(2)程序冲突

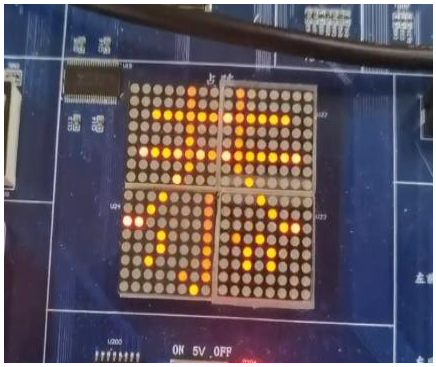
确保没有其他占用相关资源的程序在运行，使用 **ps** 和 **kill** 命令管理进程。

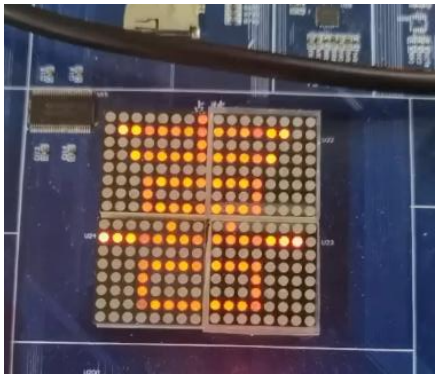
3、代码分析及实验结论

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
```

```
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <sys/select.h>
struct typFNT_GB16 // 汉字字模数据结构
{
char Index[2]; // 汉字内码索引
char Msk[32]; // 点阵码数据
};
struct typFNT_GB16 GB_16[] = // 数据表
{
"",
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0
0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x
00, "恭",
0x04,0x00,0x04,0x20,0x22,0x24,0x19,0x24,0x00,0xA4,0x40,0x7F,0x80,0x24,0x7F,0x2
4,0x00,0x24,0x08,0x7F,0x30,0xA4,0x09,0x24,0x32,0x24,0x04,0x24,0x04,0x20,0x00,0
x00, "喜",
0x02,0x00,0x02,0x02,0x02,0x0A,0x7A,0xEA,0x4A,0xAA,0x4B,0xAA,0x4A,0xAA,0x4A,0xAF,0x4A,0xAA,0x4B,0xAA,0x4A,0xAA,0x7A,0xEA,0x02,0x0A,0x02,0x02,0x0
2,0x00,0x00,0x00, "发",
0x40,0x00,0x20,0x10,0x10,0x3E,0x88,0x10,0x87,0x10,0x41,0xF0,0x46,0x9F,0x28,0x9
0,0x10,0x90,0x28,0x92,0x27,0x94,0x40,0x1C,0xC0,0x10,0x40,0x10,0x00,0x10,0x00,0
x00, "财",
0x80,0x00,0x43,0xFE,0x20,0x02,0x18,0x02,0x07,0xFA,0x08,0x02,0x73,0xFE,0x20,0x
00,0x08,0x10,0x06,0x10,0x41,0x90,0x80,0x70,0x7F,0xFF,0x00,0x10,0x00,0x10,0x00,0
x00,
"",
0x03,0x00,0x0E,0x80,0x18,0x00,0x17,0xFE,0x2F,0xFE,0x6F,0xFE,0xF0,0x00,0xF8,0
x08,0xFD,0xDE,0x7E,0xEA,0x3F,0x76,0x1F,0xBE,0x00,0xC0,0x00,0x62,0x00,0x34,0
x00,0x18
};
int main(int argc, char *argv[])
{
int mem_fd,i,j,z;
unsigned short *cpld;
mem_fd = open("/dev/mem", O_RDWR);
cpld = (unsigned short*)mmap(NULL,(size_t)0x20,PROT_READ | PROT_WRITE
|PROT_EXEC,MAP_SHARED,mem_fd,(off_t)(0x8000000));
if(cpld == MAP_FAILED)
46
return;
while(1)
```

```
{
for(z=0;z<sizeof(GB_16)/34;z++) //6 个汉字字模
for(j=0;j<16;j++)
{
for(i=0;i<16;i++)
{
if(i < 16-j)
*(cpld+((0xc0+i)<<1)) = ((GB_16[z].Msk[(i+j)*2]<<8)+(GB_16[z].Msk[(i+j)*2
+1])); //数码管
else
*(cpld+((0xc0+i)<<1)) = ((GB_16[z+1].Msk[(i-(16-j))*2]<<8)+
(GB_16[z+1].Msk[(i-(16-j))*2+1])); //数码管
}
usleep(200000);
}
}
munmap(cpld,0x20);
close(mem_fd);
return 0;
}
```





实验成功地编写并执行了点阵数码管驱动的测试程序。
程序能够正确地循环显示“恭喜发财”，验证了驱动程序和硬件接口的正确性。
通过本次实验，进一步理解了嵌入式系统中驱动程序和用户空间程序的交互，以及如何通过内存映射访问硬件资源，为更复杂的嵌入式开发奠定了基础。