实验 15：Linux 下 GPIO 驱动程序编写实验@

一、实验目的

1.理解 Linux GPIO 驱动程序的结构、原理。
2.掌握 Linux GPIO 驱动程序的编程。
3.掌握 Linux GPIO 动态加载驱动程序模块的方法。

二、实验内容

1. 编写 GPIO 字符设备驱动程序。
2. 编写 Makefile 文件。
3. 编写测试程序。
4. 调试 GPIO 驱动程序和测试程序。

三、实验设备

1.硬件：PC 机，基于 ARM9 系统教学实验系统实验箱 1 台；网线；串口线，电压表。
2.软件：PC 机操作系统；Putty；服务器 Linux 操作系统；arm-v5t_le-gcc 交叉编译环境。
3.环境：ubuntu12.04.4；文件系统版本为 filesys_clwxl；烧写的内核版本为 uImage_slh_gpio，编译成的驱动模块为 davinci_dm365_gpios.ko，驱动源码见 GPIO 文件夹。

四、预备知识

1. C 语言的基础知识。
2. 软件调试的基础知识和方法。
3. Linux 基本操作。
4. Linux 驱动程序的编写。

五、实验说明

5.1 概述

在嵌入式系统中，常常有数量众多，但是结构却比较简单的外部设备/电路，对这些设备/电路有的需要 CPU 为之提供控制手段，有的则需要被 CPU 用作输入信号。而且，许多这样的设备/电路只要求一位，即只要有开/关两种状态就够了，例如灯的亮与灭。对这些设备/电路的控制，使用传统的串行口或并行口都不合适。所以在微控制器芯片上一般都会提供一个通用可编程 I/O 接口，即 GPIO（General Purpose Input Output）。

GPIO 的驱动主要就是读取 GPIO 口的状态，或者设置 GPIO 口的状态。就是这么简单，但是为了能够写好的这个驱动，在 LINUX 上作了一些软件上的分层。为了让其它驱动可以方便的操作到 GPIO，在 LINUX 里实现了对 GPIO 操作的统一接口，这个接口实则上就是 GPIO 驱动的框架。

在本实验中，将编写简单的 GPIO 驱动程序来控制 LCD 液晶屏屏幕的亮灭，然后动态加载模块，并编写测试程序，以验证驱动程序。

5.2 实现的功能

1> 设置对应的 GPIO 口为输出。

2> 设置对应的 GPIO 口为输入。

3> 设置对应的 GPIO 口为高电平。

4> 设置对应的 GPIO 口为低电平。

5> 获取对应的 GPIO 的状态。

5.3 基本原理

GPIO 驱动是 Linux 驱动开发中最基础、但却是很常用、很重要的驱动。比如要点亮一个 LED 灯、键盘扫描、输出高低电平等等。而 Linux 内核的强大之处在于对最底层的 GPIO 硬件操作层的基础上封装了一些统一的 GPIO 操作接口，也就是所谓的 GPIO 驱动框架。这样开发人员可以调用这些接口去操作设备的 I/O 口，不需要担心硬件平台的不同导致 I/O 口的不同，方便对各个模块进行控制。

GPIO 外设提供专用的可配置为输入或输出的通用引脚。当配置为一个输出，你可以写一个内部寄存器来控制输出引脚上的状态。当配置为一个输入时，你可以通过读取内部寄存器的状态来检测输入的状态。当配置为一个高电平时，可以通过改变内部寄存器的状态来改变引脚的状态为高电平。如表 1 所示：

表 1 GPIO 寄存器

GPIO 寄存器	实际执行函数	作用
DIRn	gpio_direction_output(arg,1)	设置该 GPIO 口为输出
DIRn	gpio_direction_input(arg)	设置该 GPIO 口为输入
SET_DATAn	gpio_set_value(arg,1)	设置该 GPIO 口为高电平
CLR_DATAn	gpio_set_value(arg,0)	设置该 GPIO 口为低电平
GET_DATA	gpio_get_value(arg)	获取该 GPIO 的状态

由于 TMS320DM365 芯片的管脚不是很多，所以大部分管脚都是复用的，需要对复用管脚进行有序的管理，保证系统正常稳定工作，而在应用层，也需要对 IO 管脚进行控制来实现一定功能。在进行 GPIO 驱动开发前，在内核中进行如下配置:

1> 在内核 linux-2-6-18_pro500/arch/arm/mach-davinci/board-dm365-evm.c 中的 davinci_io_init()函数进行配置。

```
davinci_cfg_reg(DM365_GPIO63, PINMUX_RESV);
```

2>在 DM365 板文件的系统启动函数内核 linux-2-6-18_pro500/include/asm/arch/mux.h 中结构体函数 enum davinci_dm365_index 添加并使能相应的 I/O 端口。

```
DM365_GPIO63,
```

davinci_cfg_reg（）函数的参数就是 enum davinci_dm365_index 枚举中的成员，davinci_cfg_reg（）函数根据获得的枚举成员参数，来到 const struct pin_config __initdata_or_moduldedavinci_dm365_pinmux[]数组中相应位置找到需要配置的引脚复用控制寄存器。

3> 在内核 linux-2-6-18_pro500/arch/arm/mach-davinci/mux_cfg.c 中 const struct pin_config __initdata_or_moduldedavinci_dm365_pinmux[]函数进行配置，用于对 GPIO 管脚的配置工作。

```
MUX_CFG("GPIO63",  2,  6,    1,  0,    0);
```

MUX-CFG 中内容是为了给 davinci_cfg_reg 函数提供需要的配置复用引脚的 MUX 控制寄存器的编号、寄存器对应的位偏移、位掩码、模式位值等，以便 davinci_cfg_reg 完成引脚功能的配置。在 MUX_CFG 中，第一个变量是 GPIO，用作索引；第二个参数表示管脚复用的寄存器号，根据 DM365 的 datasheet，在 DM365 上，有 PINMUX0~PINMUX4 总共 5 个寄存器对 IO 管脚复用配置。PINMUX2 寄存器的第 1,2 位定义了 GPIO63 管脚的复用情况；第三个参数表示偏移量，这里是 6 即 PINMUX2 的第 6 位起；第四个参数表示该寄存器对应位的掩码值，这里 1 表示 1 位，假如 3 的话在二进制中为 11 也即两位；第五个参数表示该管脚需设定的值，这里设为 0，即将其复用设置为 GPIO63；最后一个参数表示是否开启对管脚的调试，一般设置为 0，即该管脚不需要开启调试。

4> 在内核中配置好所需的 GPIO 后，重新编译进内核。

5.4 硬件平台框架

5.4.1 DM365 嵌入式处理器

TMS320DM365 是 TI 公司推出的一款基于 DaVinci 技术的高清视频处理器，它集成了一颗 ARM926EJ-S 核，通过其与视频协处理器以及丰富的外围设备的融合，为高清视频处理提供了很好的解决方案。DM365 的内部功能结构如图 1 所示。

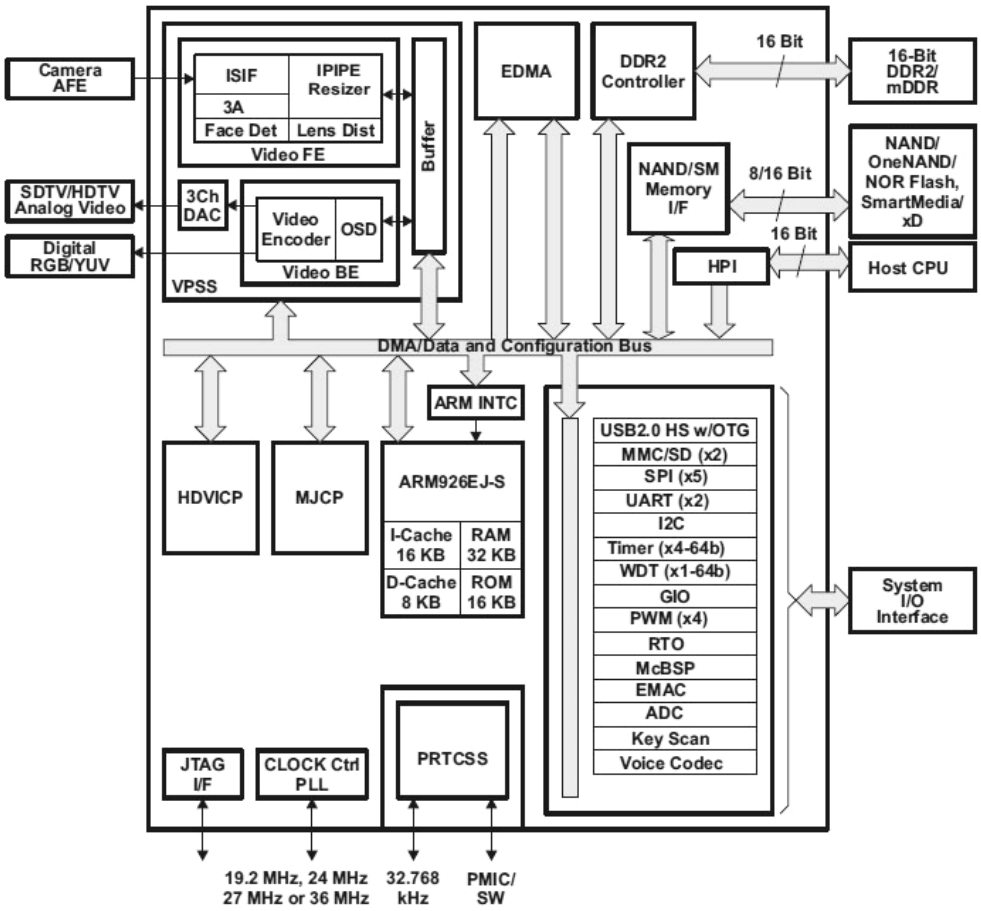

图 1 DM365 内部功能结构

DM365 片上系统主要由 ARM 子系统（ARM Subsystem，ARMSS）、视频处理子系统（Video Processing Subsystem，VPSS）、视频影像协处理器（Video Image Co-Processor，VICP）以及丰富的输入输出（I/O）外设组成。

DM365 的核心是一颗主频为 300MHz 的 ARM926EJ-S 精简指令集（RISC）处理器。该处理器拥有独立的 8KB 数据缓存和 16KB 的指令缓存，还集成了 32KB 的 RAM 和 16KB 的 ROM，内存管理单元（Memory Management Units，MMU）等。另外，流水线技术的运用，使得整个系统的控制更加高效。

输入输出（I/O）外设也非常丰富，存储接口包括 MMC/SD、DDR2 控制器、NAND/SM 接口等，连接接口包括 10/100M 以太网口、USB2.0（High Speed）等，通用 I/O 接口包括 SPI、UART、I2C 等。

ARM 子系统、视频处理子系统、视频图像协处理器和外设之间通过直接内存读取（Direct Memory Access，DMA）实现。DMA 允许不同速度的硬件在不占用 CPU 大量的终端负载的情况下进行通信。DMA 数据通信时，不需要程序控制，通过硬件自动完成。这样在进行大量数据通信时，减少系统对 CPU 资源的利用率，提升了系统性能。

5.4.2　在 DM365 上的 GPIO 寄存器

GPIO 外设模块寄存器如图 2 所示

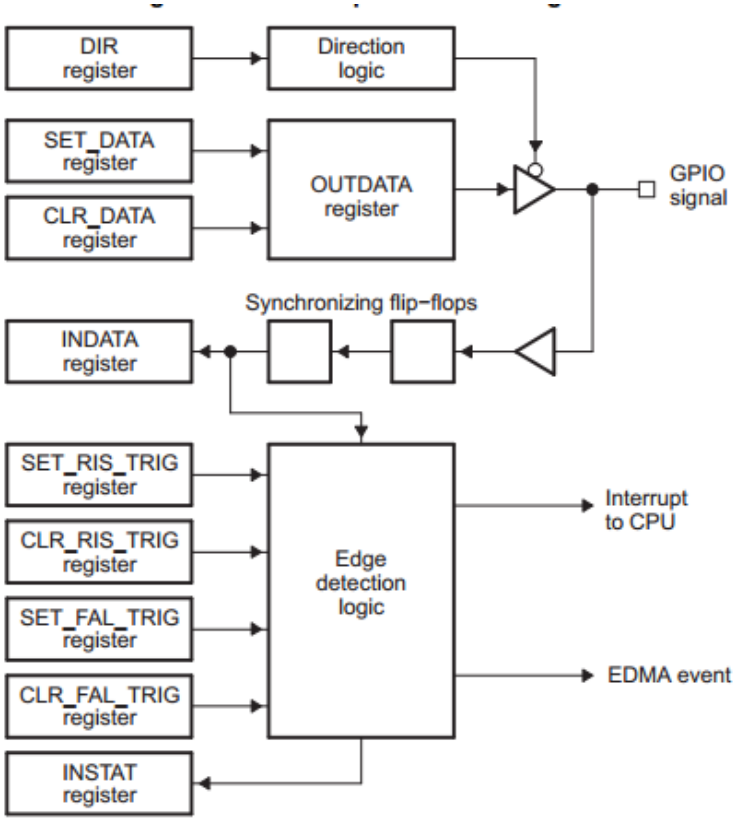

图 2 GPIO 模块寄存器图

5.4.3 总体硬件结构设计

硬件电路图如图 3 所示：

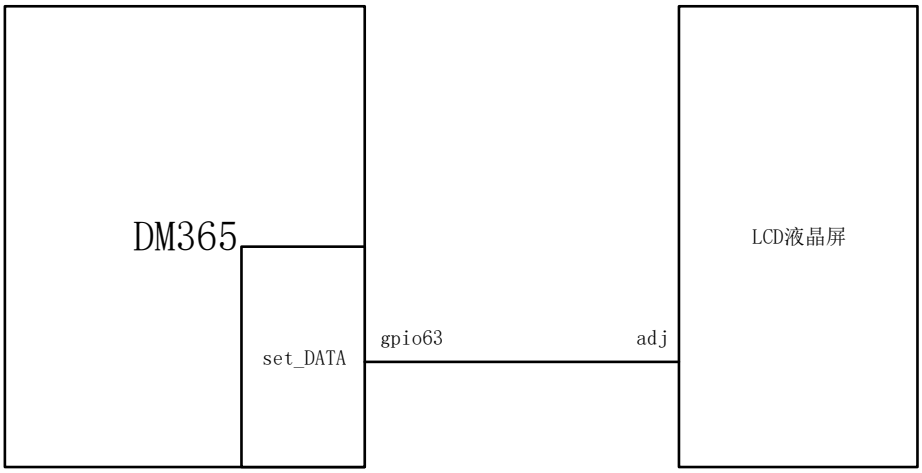

图 3　GPIO 控制 LCD 硬件电路图

LCD 屏幕和 DM365 主板的电路连接，主板 GPIO63 连接的是 LCD 液晶屏的 adj 引脚，adj 控制 LCD 的背光。当 GPIO63 为高电平时，LCD 屏幕背光灭，当 GPIO63 为低电平时，LCD 屏幕背光亮。

其中 DM365：主控 CPU。
DM365 寄存器 set_DATA 作用：控制 GPIO63 口输出高低电平。

5.5 软件框架

5.5.1 软件流程

软件设计流程图 4 如下：

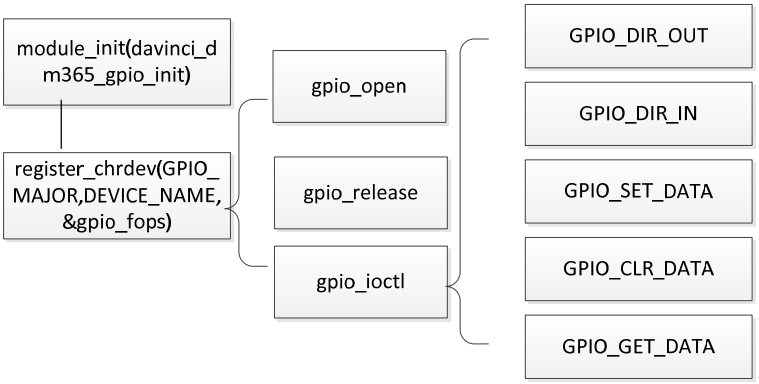

图 4　GPIO 驱动主要部分

GPIO 驱动主要代码如下：

```
static struct file_operations gpio_fops = {
.owner=THIS_MODULE,
.open=gpio_open,
.release=gpio_release,
.ioctl=gpio_ioctl
};
static int __init davinci_dm365_gpio_init (void) {
int ret;
ret=register_chrdev(GPIO_MAJOR, DEVICE, &gpio_fops);
…
}
```

GPIO 驱动加载后，调用字符设备驱动注册函数 register_chrdev，向 Linux 内核注册字符设备驱动，同时，在 gpio_fops 中提供 open、release、ioctl 方法，并为上层提供了表 2 下表所示的 ioctl 命令：

表 2 GPIO ioctl 命令

ioctl 命令	实际执行函数	作用
DEF_GPIO_DIR_OUT	gpio_direction_output(arg,1)	设置该 GPIO 口为输出
DEF_GPIO_DIR_IN	gpio_direction_input(arg)	设置该 GPIO 口为输入
DEF_GPIO_SET_DAT	gpio_set_value(arg,1)	设置该 GPIO 口为

A		高电平
DEF_GPIO_CLR_DATA	gpio_set_value(arg,0)	设置该 GPIO 口为低电平
DEF_GPIO_GET_DATA	gpio_get_value(arg)	获取该 GPIO 的状态

5.5.2 操作函数的接口函数和结构体

1> 驱动源码函数分析

函数：MODULE_LICENSE("Dual BSD/GPL");

功能：将模块的许可协议设置为 BSD 和 GPL 双许可,必不可少的。

函数：module_init(davinci_dm365_gpio_init);

功能：module_init 是内核模块一个宏。其用来声明模块的加载函数，也就是使用 insmod 命令加载模块时，调用的函数 davinci_dm365_gpio_init()。

函数: module_exit(davinci_dm365_gpio_exit);

功能：module_exit 是内核模块一个宏。其用来声明模块的释放函数，也就是使用 rmmod 命令卸载模块时，调用的函数 davinci_dm365_gpio_exit（）。

函数：static int __init davinci_dm365_gpio_init (void)

功能：加载函数调用驱动注册函数实现驱动程序在内核的注册，同时还有可能对设备进行初始化，在驱动程序加载被调用。

函数：static void __exit davinci_dm365_gpio_exit(void)

功能：卸载函数调用解除注册函数实现驱动程序在内核的中的解除注册，同时在驱动程序卸载时被调用。

函数：static struct file_operations gpio_fops = {

```
    .owner    =  THIS_MODULE,
    .open     =  gpio_open,
    .release =  gpio_release,
    .ioctl    =  gpio_ioctl
};
```

功能：这是名为 gpio_fops 的 file_operations 的结构体变量，并对其部分成员用 gpio_open（指定 gpio 设备的打开）、gpio_release（指定设备的释放内存和关闭）、gpio_ioctl（对设备的 I/O 通道进行管理）进行初始化，gpio_open、gpio_release、gpio_ioctl 函数分别对应 gpio_fops 的一个接口函数，构成字符设备驱动程序的主体。

参数：gpio 指设备名称(可以任取)。

函数：static int gpio_open(struct inode *inode, struct file *file)

功能：open()函数使用 MOD_INC_USE_COUNT 宏增加驱动程序打开的次数，以防止还有设备打开卸载驱动程序，如果是初次打开该设备，则对该设备进行初始化。

参数：inode:对应文件的的 inode 节点。

file: 设备的私有数据指针。

函数：static int gpio_release(struct inode *inode, struct file *file)

功能： release()函数使用 MOD_DEC_USE _COUNT 宏减少驱动程序打开的次数，以防止还有设备打开时卸载驱动程序。

参数：inode:关闭文件的的 inode 节点。

file: 设备的私有数据指针

函数：int gpio_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg)

功能：ioctl()函数对设备相关控制命令的实现,既不是读操作也不是写操作，调用成功返回一个非负值。

参数：inode:文件的的 inode 节点。

filp:是文件结构体指针

cmd: 用户程序定义对设备的 I/O 控制命令。

arg:对对应的 cmd 命令传入的参数。

2> 测试代码函数分析

函数：int main(int argc, char **argv)

功能：主函数，函数的入口。

参数：argc：是命令行总的参数个数。

argv：是 argc 个参数，其中第 0 个参数是程序的全名，以后的参数命令行后面跟的用户输入的参数。

六、实验步骤

本次实验共涉及 3 个文件的编写 davinci_dm365_gpios.c、gpios.c、Makefile。将执行下面几个步骤：

步骤 1：编译 GPIO 驱动（在服务器上进行）

首先在指定位置创建 GPIO 目录，进入 GPIO 目录创建驱动文件 davinci_dm365_gpios.c ，并编写驱动代码。

```
~$ mkdir GPIO
~$ cd GPIO
~$ vim davinci_dm365_gpios.c
```

参考驱动代码如下所示：

```
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/platform_device.h>
#include <asm/arch/gpio.h>
#include <linux/types.h>
#include <linux/cdev.h>
#include <asm/arch-davinci/gpio.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <asm/arch/hardware.h>
#include <asm/arch/gpio.h>

//设备名称，如果注册成功可通过 cat /proc/devices 查看到
#define DEVICE_NAME "dm365_gpio_experiment"
#define GPIO_MAJOR 0 //主设备号为 0 表示自动分配未使用的主设备号
#define NOT_MUX_GPIO '2'
//ioctl 的命令
#define DEF_GPIO_DIR_OUT    0x01
#define DEF_GPIO_DIR_IN     0x02
#define DEF_GPIO_SET_DATA 0x03
#define DEF_GPIO_CLR_DATA 0x04
#define DEF_GPIO_GET_DATA 0x05
static int gpio_open(struct inode *inode, struct file *file)
{
//      printk("open gpio\nhere is dm365 gpio driver\n");
        return 0;
}
static int gpio_release(struct inode *inode,struct file *filp)
{
        return 0;
}
int gpio_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg)
{
            int ret = 0;
            switch (cmd)
            {
                        case DEF_GPIO_DIR_OUT:
                                    printk("***dir out***cmd=%d,arg=%d\n", cmd,(unsigned)arg);
                                    gpio_direction_output(arg,1);
                                    break;
                        case DEF_GPIO_DIR_IN:
                                    printk("***dir in***cmd=%d,arg=%d\n", cmd,(unsigned)arg);
                                    gpio_direction_input(arg);
```

```
                                break;
                        case DEF_GPIO_SET_DATA:
                        case DEF_GPIO_CLR_DATA:
//                                printk("data=%d\n", cmd);
                                if (cmd == DEF_GPIO_SET_DATA)
                                        gpio_set_value(arg, 1);
                                else
                                        gpio_set_value(arg, 0);
//                                break;
                        case DEF_GPIO_GET_DATA:
                                ret = gpio_get_value(arg);
//                               printk("gpio%d = %d\n",arg,ret);
                                break;
                }
                return ret;
}

static struct file_operations gpio_fops = {
                .owner = THIS_MODULE,
                .open = gpio_open,
                .release=gpio_release,
                .ioctl = gpio_ioctl
};
static int __init davinci_dm365_gpio_init(void)
{
        int ret;
        ret = register_chrdev(GPIO_MAJOR, DEVICE_NAME, &gpio_fops);
        if(ret < 0)
        {
                printk("dm365_gpio register falid!\n");
                return ret;
        }
        printk ("dm365_gpio initialized\n");
        return ret;
}
static void __exit davinci_dm365_gpio_exit(void)
{
        unregister_chrdev(GPIO_MAJOR, DEVICE_NAME);
        printk("dm365_gpio exit\n");
}
module_init(davinci_dm365_gpio_init);
module_exit(davinci_dm365_gpio_exit);
MODULE_AUTHOR("jingwei,Song <Punuo Ltd.>");
MODULE_DESCRIPTION("Davinci DM365 gpio driver");
MODULE_LICENSE("GPL");
```

编写驱动程序编译成模块所需要的 Makefile 文件，执行

```
~$ vim Makefile
```

Makefile 参考代码如下：

```
KDIR:= /home/xxx/kernel-for-mceb/     //编译驱动模块依赖的内核路径，修改为使用服
                                        务器上内核文件所在的路径。xxx 是
                                        说明者使用的用户名，在 make 命令
                                        执行之前将代码中 xxx 替换为具体
                                        挂载的根文件系统目录的。
    CROSS_COMPILE       = arm_v5t_le-
    CC      = $(CROSS_COMPILE)gcc
    .PHONY: modules clean
    obj-m := davinci_dm365_gpios.o          //表明有个模块要从目标文件
                                            davinci_dm365_gpios.o 建立.
                                            在从目标文件建立后结果模
                                            块命名为 davinci_dm365_gpios.ko
    modules:
        make -C $(KDIR) M=`pwd` modules     //根据提供的内核生成
                                            davinci_dm365_gpios.o
    clean:
        make -C $(KDIR) M=`pwd` modules clean
```

执行 make 命令，成功后会生成 davinci_dm365_gpios.ko 等文件，如下所示,

```
~$ make
make -C /home/baozi/kernel-for-mceb/ M=`pwd` modules
make[1]:  正在进入目录 `/home/xxx/kernel-for-mceb'
  CC [M]    /home/xxx/GPIO/davinci_dm365_gpios.o
  Building modules, stage 2.
  MODPOST
  CC          /home/xxx/GPIO/davinci_dm365_gpios.mod.o
  LD [M]    /home/xxx/GPIO/davinci_dm365_gpios.ko
make[1]:正在离开目录 `/home/xxx/kernel-for-mceb'
~$ ls
davinci_dm365_gpios.c     davinci_dm365_gpios.mod.c    davinci_dm365_gpios.o
  Makefile     davinci_dm365_gpios.ko    davinci_dm365_gpios.mod.o
```

davinci_dm365_gpios.ko 文件生成成功后，将其复制到挂载的文件系统 modules 的目录下。

```
~$ cp davinci_dm365_gpios.ko /home/xxx/filesys_test/modules/
```

接着在 putty 端查看设备的 modules 目录

```
[root@zjut ~]# cd /modules/
```

```
[root@zjut modules]# ls
at24cxx.ko              i2c.ko                  rt5370sta.ko            rtutil5370ap.ko
davinci_dm365_gpios.ko  lcd.ko                  rt5572sta.ko            rtutil5572sta.ko
egalax_i2c.ko           ov5640_i2c.ko           rtnet5370ap.ko          ts35xx-i2c.ko
fm1188_i2c.ko           rt5370ap.ko             rtnet5572sta.ko         ttyxin.ko
```

接着加载模块如下所示：

```
[root@zjut modules]# insmod davinci_dm365_gpios.ko
[  824.850000] dm365_gpio initialized
```

接着创建测试代码 gpio.c

```
~$ vi gpios.c
```

测试代码参考如下所示：

（命令 sudo cp davinci_dm365_gpios.ko /home/stux/filesys_wlw/modules），以备步骤 2 使用。

步骤 2：通过 pc 的远程登陆软件 putty 登录板子，接着将实验所需要的内核文件提前放入到虚拟机上 ubuntu 系统下 tftp 服务器配置的文件夹中。使用的软件有 SSH Secure Shell 和 Putty。

1> 将板子接上网线，板子和电脑之间通过 USB 转串口线连接，安装 USB 转串口驱动，打开电脑设备管理器查看新加入 COM 端口，例如电脑占用的为 COM3，如下图 6 所示：

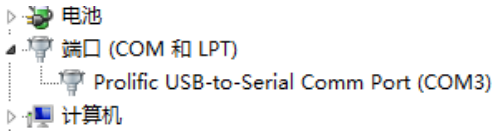

图 6 usb 转串口端口编号

2> 打开 putty 软件，接下来就是配置端口，如下图 7 所示的在 serial line 填写加入的 COM3，在 speed 填入 115200，在 Connection type 中选择 Serial，在 saved Sessions 填写加入的 COM3，点击 Save。下次做实验时，只需要选中新加入的设备对应的 COM 端口，鼠标双击或者点 open。

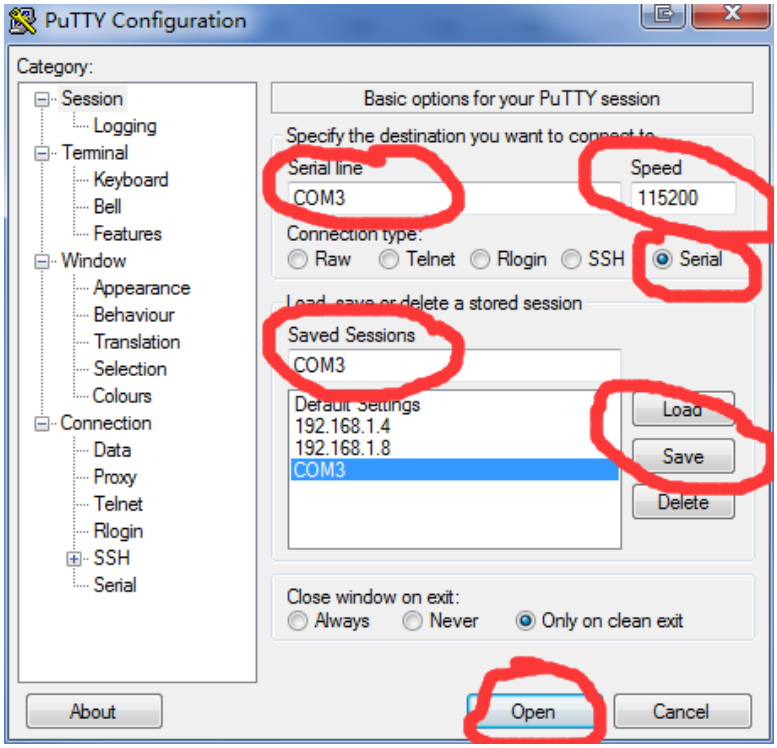

图 7　putty 设置界面

3> 打开 ssh 软件，进行如下配置，首先点击 Quick Connect，出现如下图 8 所示 Connect to Remote Host 选项卡，在 Host Name 输入服务器 IP，User Name 输入用户名，点击 Connect。

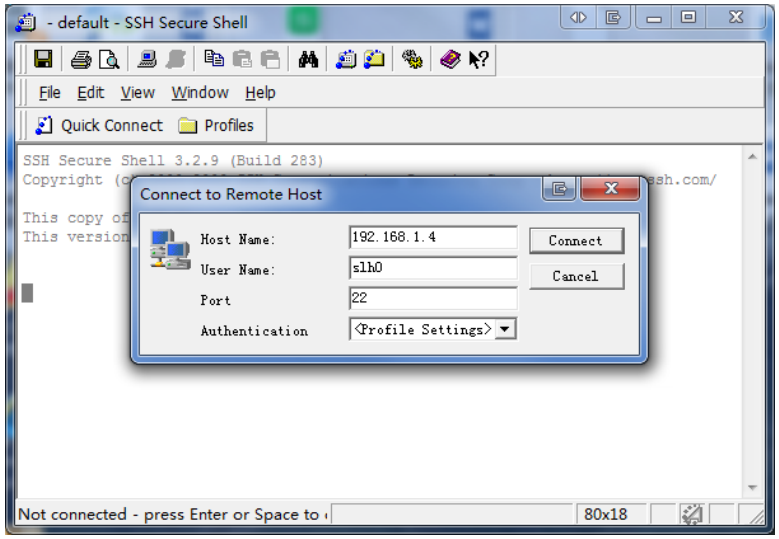

图 8　SSH 设置的界面

4> 出现 Enter Password 选项卡，输入用户名密码。登入成功如图 9：

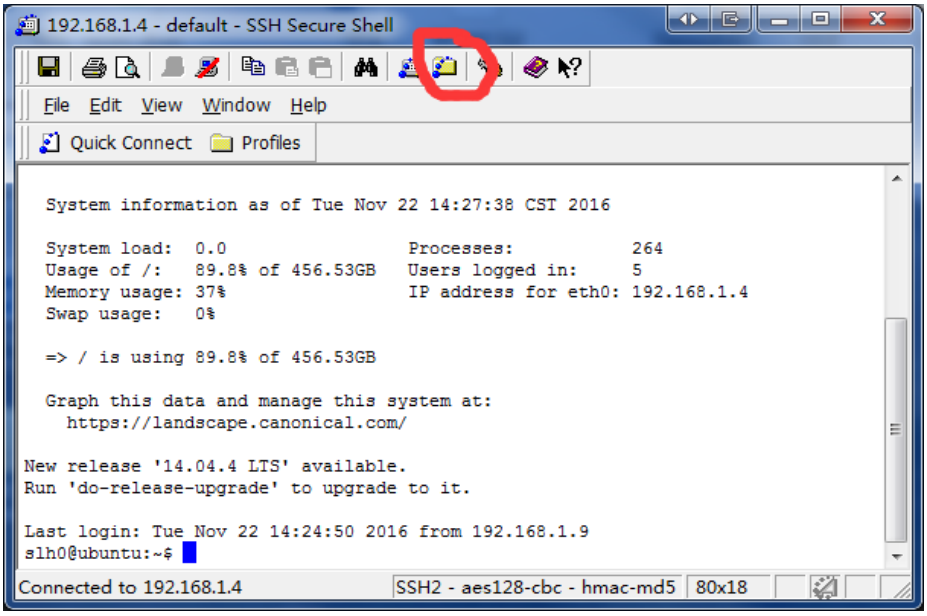

图 9　SSH 登入成功界面

5> 点击下图 10 所示红圈图标，出现如图 10 所示界面。在左侧红圈选择本地电脑存放内核的路径，在右侧红圈选择根目录（/tftpboot）。在左侧框内选中需要上传的文件，鼠标左键按住不放直接拖入或者右击选择 upload。

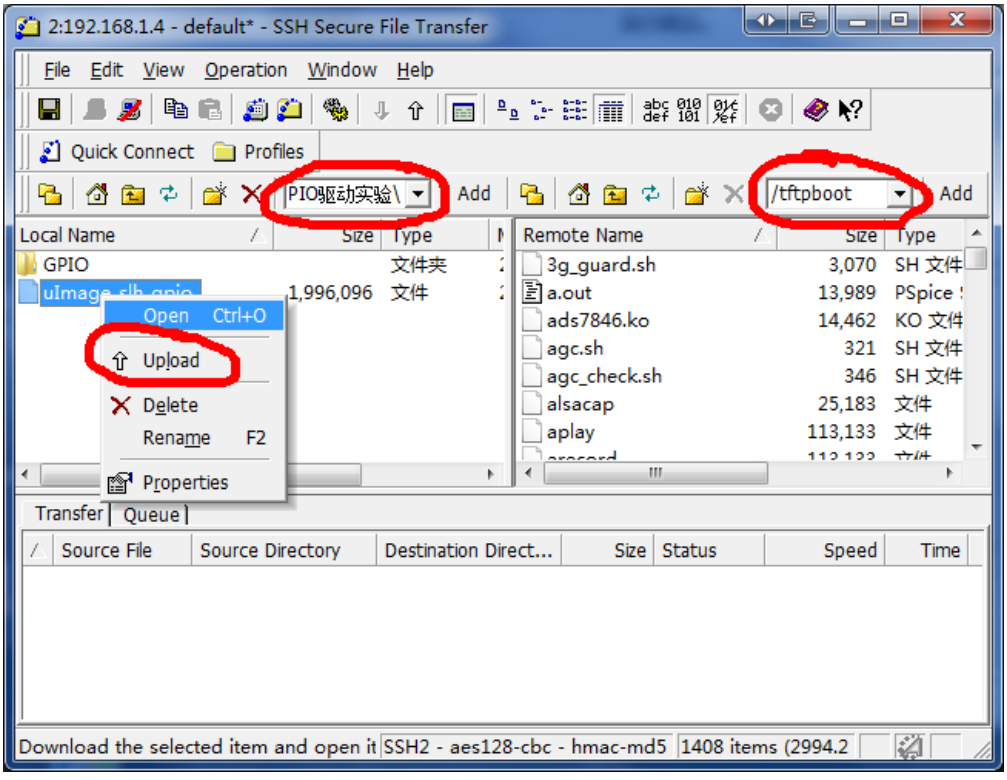

图 10　SSH 上传文件操作界面

步骤 3：动态加载模块 （以下操作在板子上进行）

（1）NFS 挂载启动

完成以上前期步骤后，正确连接好实验箱，开启电源，通过 NFS 作为启动方式挂载虚拟机上 filesys_test 文件系统作为根文件系统进行启动,

注：启动机器前要确保实验箱设备的 ip 地址和 NFS 服务器（该服务器一般和实验箱设备在同一网关下）的 ip 地址可以相互 ping 通。且实验像设备的 ip 地址是该网关下唯一的 ip 地址否则将会冲突报错。

登入板子后，在驱动模块加载之前查看设备节点使用命令 cat /proc/devices。例如图 12 所示：

```
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 14 sound
 21 sg
 29 fb
 81 video4linux
 89 i2c
 90 mtd
108 ppp
116 alsa
128 ptm
136 pts
180 usb
188 ttyUSB
189 usb_device
199 dm365_gpio
244 dm365mmap
245 edma
246 irqk
247 cmem
248 sr04
249 dht11
```

图 12　当前已经挂载的设备节点

（3）动态加载

进入文件系统的 modules 目录下，查看已经加载的模块 lsmod，如图 13 所示。运行模块加载命令 insmod davinci_dm365_gpios.ko，模块成功加载，提示信息 dm365_gpio initialized。(如果加载失败，可以用 lsmod 命令查看是否已存在模块 davinci_dm365_gpios.ko，若已存在，可用 rmmod davinci_dm365_gpios.ko 先卸载，再依照上述方式加载查看结果）。

步骤 4：手动创建设备节点（以下操作在板子上进行）

当使用 insmod 命令成功加载 davinci_dm365_gpios.ko 驱动模块后，再次通过命令 cat /proc/devices 查看已经注册的设备和设备号如图 14 所示，可以发现 dm365_gpio_experiment 驱动已经加载完成。

驱动模块的设备节点创建有俩种方式手动创建和自动创建，gpio 驱动程序为手动创建设备节点，是为了能够更好掌握节点创建的过程。mknod /dev/dm365_gpio c 243 0（ 其中，

dm365_gpio_experiment 是设备文件名，c 代表字符设备，243 代表上图中 dm365_gpio_experiment 对应的主设备号，0 表示次设备号，主设备号是 0 因此这个设备号是随机分配未使用的设备号，因此每次加载模块产生的设备号可能是不一样的。

此时生成一个设备节点如下图所示

图 13 加载 gpio 模块后当前已经加载的模块

步骤 5：编写测试程序（在虚拟机上进行），并进行调试（在实验箱上进行）

测试程序在文件下 GPIO/gpio/gpio.c 下，gpio.c.就是对应的测试文件查看实验测试程序。

在服务器编写测试程序的各函数后，接下来是使用交叉编译工具编译测试程序，并将编译后生成的可执行文件挂载到实验箱的板子上运行调试。

```
$ arm_v5t_le –gcc   gpio.c   - o   gpio
```

交叉编译生成可执行文件 gpio。编译成功后，可看见 gpio 文件，将可执行文件 gpio 复制到板子挂载的文件系统 /home/stx/filesys_wlw/opt/dm365。（x 表示学生 NFS 方式登录挂载的用户目录）交叉编译生成可执行文件 gpio。编译成功后，将可执行文件 gpio 复制到 /home/stux/filesys_wlw/opt/dm365 文件夹下，此时对实验箱进行操作进入/opt/dm365 目录下执行测试代码。

```
$ cp   gpio   /home/stx/filesys_wlw/opt/dm365

$ cd   /opt/dm365
```

执行如下命令 gpio 63 0/3。观察实验箱上液晶屏暗亮有没有达到实验预期结果。如图 15 所示：

```
# gpio 63 0 （实验箱的板子上运行）lcd 背光打开
# gpio 63 3（实验箱的板子上运行） lcd 背光关闭
```

```
[root@zjut modules]# insmod  davinci_dm365_gpios.ko
[ 1474.100000] dm365_gpio initialized
[root@zjut modules]# gpio 63 3
[ 1759.450000] ***dir out***cmd=1,arg=63
[root@zjut modules]# gpio 63 0
[root@zjut modules]#
```

图 15 输入控制 LCD 背光指令

```
#cat gpio.c
#include<stdio.h>
#include<stdlib.h>
#include<fcntl.h>
#include<sys/ioctl.h>
#define DEF_GPIO_DIR_OUT    0x01
#define DEF_GPIO_DIR_IN     0x02
#define DEF_GPIO_SET_DATA 0x03
#define DEF_GPIO_CLR_DATA 0x04
#define DEF_GPIO_GET_DATA 0x05
void main(int argc,char * argv[])
{
        FILE *fd;
        fd=open("/dev/dm365_gpio",O_RDWR,0);
        int gpio;
        int cmd;
        gpio=atoi(argv[1]);
        if(argc==1)
        printf("please input arg\n");
        else if(argc==2)
                ioctl(fd,DEF_GPIO_GET_DATA,gpio);
        else
        {
                cmd=atoi(argv[2]);
                switch(cmd)
                {
                        case 0:ioctl(fd,DEF_GPIO_CLR_DATA,gpio);break;
                        case 1:ioctl(fd,DEF_GPIO_SET_DATA,gpio);break;
                        case 2:ioctl(fd,DEF_GPIO_DIR_IN,gpio);break;
                        case 3:ioctl(fd,DEF_GPIO_DIR_OUT,gpio);break;
                        default: printf("wrong\n");
                }
        }
}
```

实验结束。