一.嵌入式linux设备驱动开发相关内容
<<Linux设备驱动程序>>第三版
<<Linux内核设计与实现>>第三版
<<跟我一起写Makefile>>电子档

关于SecureCRT远程登录linux系统的配置过程：
打开快速连接->协议：ssh2
主机名：192.168.1.8
用户名：tarena
->保存->输入密码->配置SecureCRT->会话选项->仿真->终端：ANSI
使用颜色方案选中
->外观：设置自己喜欢的字体Courier New
字符编码：UTF-8
保存
->重新SecureCRT即可

面试题：如何开发一个linux硬件设备驱动？
友情提示：嵌入式linux系统一旦运行起来以后,要花更多的时间
和精力放在开发板上的外设硬件的设备驱动程序上,
如果这个外设有了驱动,需要进行测试,测试的前提是
你要看得懂；
如果这个外设没有驱动,要进行这个外设硬件的设备驱动开发

1. 设备驱动概念
一个驱动的关键两个内容：
1.将硬件的整个操作过程进行封装
2.必须能够给用户提供访问操作硬件的接口(函数)
将来用户调用函数能够随便访问硬件

2.linux系统的两个空间(两种状态)：用户空间和内核空间(了解即可)
用户空间：
又称用户态
包含的软件就是各种命令,各种应用程序,各种库,各种配置服务等
用户空间的软件在运行的时候,CPU的工作模式为USER模式
用户空间的软件不能访问硬件设备的物理地址,如果要访问硬件物理地址
必须将硬件外设的物理地址映射到用户空间的虚拟地址上
用户空间的软件不能直接访问内核空间的代码,地址和数据
用户空间的软件如果进行非法的内存访问,不会导致操作系统崩溃
但是应用软件会被操作系统干掉(例如：*(int *)0=0)
用户空间的软件类似论坛的普通用户
用户空间的虚拟地址空间大小为3G(0x00000000~0xBFFFFFFF)
内核空间：
又称内核态
内核空间的软件就是内核源码(zImage)
内核代码运行时,CPU的工作模式为SVC模式
内核空间代码同样不能访问硬件外设的物理地址,必须将物理地址
映射到内核空间的虚拟地址上
内核代码如果进行非法的内存访问,操作系统会直接崩溃(吐核)
(例如：*(int *)0=0)
内核空间的软件类似论坛的管理员
内核空间的虚拟地址空间大小为1G(0xC0000000~0xFFFFFFFF)

3.linux系统设备驱动分类
字符设备驱动
字符设备访问时按照字节流形式访问
例如：LED,按键,UART接口设备(BT,GPS,GPRS),触摸屏

LCD,声卡,摄像头,各种传感器

块设备驱动

块设备访问时按照一定的数据块进行访问,数据块比如为512字节,1KB字节

例如：硬盘,U盘,SD卡,TF卡,Nor,Nand等

网络设备驱动

网络设备驱动访问时需要配合TCP/IP协议栈进行访问

驱动一般由芯片厂家提供,驱动开发者需要进行移植

例如：DM9000网卡基地址

4.linux字符设备驱动开发相关内容

4.0.明确不管什么驱动,它们都是内核程序

4.1.明确linux内核程序编程基本框架

回忆：应用程序编程框架

vim helloworld.c

```
#include <stdio.h> //标准C头文件

//main：程序的入口函数
int main(int argc, char *argv[])
{
	//标准C库函数
	printf("hello,world\n");

	//程序的出口
	return 0;
}
```

内核程序参考代码：

```
mkdir /opt/drivers/day01/1.0 -p
cd /opt/drivers/day01/1.0
vim helloworld.c 添加第一个内核程序
#include <linux/init.h>
#include <linux/module.h>
static int helloworld_init(void)
{
	printk("hello,world\n");
	return 0;
}
static void helloworld_exit(void)
{
	printk("good bye world!\n");
}
module_init(helloworld_init);
module_exit(helloworld_exit);
MODULE_LICENSE("GPL");
```

说明：

1.内核程序使用的头文件位于linux内核源码中(/opt/kernel)

2.内核程序的入口函数需要使用module_init宏进行修饰,例如
helloworld_init函数就是此内核程序的入口函数,将来加载
安装驱动时(insmod),内核会调用此函数;
此函数的返回值必须为int型,执行成功返回0,执行失败返回负值

3.内核程序的出口函数需要使用module_exit宏进行修饰,例如
helloworld_exit函数就是此内核程序的出口函数,将来卸载
驱动(rmmod)时,内核会调用此函数

4.任何一个内核程序源码(.c结尾)必须添加MODULE_LICENSE("GPL")

这句话,就是告诉内核,此内核程序同样遵循GPL协议,否则后果很严重
5.内核打印函数使用printk,此函数定义不再C库中,而是在内核源码中
6.结论：编译内核程序肯定需要关联内核源码

4.2.内核程序的编译
回顾：应用程序的编译
gcc -o helloworld helloworld.c
编写Makefile,make编译即可
内核程序编译：
回顾led_drv.c编译步骤：
1.静态编译
拷贝内核程序到内核源码中
修改Kconfig
修改Makfile
make menuconfig //选择为*
make zImage (led_drv.c包含在zImage里面)
2.模块化编译
拷贝内核程序到内核源码中
修改Kconfig
修改Makfile
make menuconfig //选择为M
make zImage
make modules //将led_drv.c->led_drv.ko
insmod
rmmod
3.模块化编译方法2：
思想就是无需把内核程序拷贝到内核源码中
无需修改Kconfig和Makefile
无需make menuconfig
无需make zImage
只需一个小小的Makeifle即可搞定：
死记一下参考代码：
```
cd /opt/drivers/day01/1.0
vim Makefile 添加如下内容：
obj-m += helloworld.o  #采用模块化编译,helloworld.c->helloworld.ko
#执行命令make all或者make,执行对应的命令make -C ...
all:
    make -C /opt/kernel SUBDIRS=$(PWD) modules
    #make -C /opt/kernel=cd /opt/kernel && make
    #SUBDIRS=/opt/drivers/day01/1.0,告诉内核源码,在内核源码之外还有一个目录作为子目录
    #modules：对1.0这个子目录下的内核程序采用模块化编译
clean：
    make -C /opt/kernel SUBDIRS=$(PWD) clean
    #将子目录1.0的程序进行make clean操作
保存退出
make //编译
ls
  helloworld.ko //编译成果
cp helloworld.ko /opt/rootfs/

开发板测试：
重启开发板,进入uboot,让内核加载采用tftp,让内核启动采用nfs
setenv bootcmd tftp 20008000 zImage \; bootm 20008000
```

```
setenv bootargs root=/dev/nfs nfsroot=192.168.1.8:/opt/rootfs ...
savenv
boot //启动
启动以后：
insmod helloworld.ko //安装内核程序,内核执行入口函数helloworld_init
lsmod //查看内核程序的安装信息
rmmod helloworld //卸载内核程序,内核执行出口函数helloworld_exit
```

4.2.linux内核程序编程之命令行传参

1.回忆应用程序的命令行传参

```
vim helloworld.c
#include <stdio.h>
int main(int argc, char *argv[])
{
      int a;
      int b;

      if (argc != 3) {
         printf("用法: %s num1 num2\n", argv[0]);
         return -1;
      }

      //"100"->100
      a = strtoul(argv[1], NULL, 0);
      b = strtoul(argv[2], NULL, 0);

      printf("a = %d, b = %d\n", a, b);
      return 0;
}
gcc -o helloworld helloworld.c
./helloworld 100 200
分析：
argc = 3
argv[0] = "./helloworld"
argv[1] = "100"
argv[2] = "200"
缺点：一旦程序启动,后序就没法再次传递新的参数

内核程序的命令行传参实现过程：
1.内核程序的命令行传参时,接收参数的内核程序变量必须是
  全局变量
2.变量的数据类型必须是基本的数据类型,结构体不行
3.如果要给内核程序的某个全局变量传递参数,需要内核程序
  显式的进行传参声明,传参声明的宏：
  module_param(name, type, perm)
  name：接收参数的内核全局变量名
  type: 变量的数据类型：
        bool invbool
        short ushort
        int uint
        long ulong
        charp(=char *)
        切记：内核不允许处理浮点数(float,double)
        例如：2.3*3.2
              23*32/100
              浮点数的运算放在用户空间的应用程序来进行
```

perm：变量的访问权限(rwx)
例如：0664
注意：不允许有可执行权限(x=1)

案例：编写内核程序,实现内核程序的命令行传参
实施步骤：
虚拟机执行：
1.mkdir /opt/drivers/day01/2.0 -p
cd /opt/drivers/day01/2.0
3.vim helloworld.c 添加如下内容
4.vim Makefile 添加如下内容
5.make
helloworld.ko
6.cp helloworld.ko /opt/rootfs/

linux系统调试宏：
__FILE__
__LINE__
__FUNCITION__/__func__
__DATE__
__TIME__

ARM板执行：
1.不传递参数
insmod helloworld.ko
lsmod
rmmod helloworld

2.加载安装内核程序时传递参数
insmod helloworld.ko irq=100 pstring=china
lsmod
rmmod helloworld

3.加载安装内核程序以后传递参数
insmod helloworld.ko irq=100 pstring=china
//读取文件irq的内容
cat /sys/module/helloworld/parameters/irq
ls /sys/module/helloworld/parameters/pstring //没有此文件
//向文件irq重新写入一个新内容
echo 20000 > /sys/module/helloworld/parameters/irq
rmmod helloworld

结论：
1.如果传参声明时,权限为非0,那么在/sys/...../parameters
会生成一个跟变量名同名的文件,文件内容就是变量的值
2.通过修改文件的内容就可以间接修改变量的值
3.如果权限为0,那么在/sys/.../parameters下就不会
生成同名的文件,这个变量的传参只能在程序加载时进行
4.注意：/sys/目录下所有的内容都是内核创建,存在于
内存中,将来如果没有内核程序加载以后传递参数的需求
权限必须一律给0,目的为了节省内存资源!

4.3.linux内核程序编程之内核符号导出
回忆：应用程序多文件之间的调用
参考代码：
mkdir /opt/drivers/day01/3.0

```
cd /opt/drivers/day01/3.0
vim test.h //声明
#ifndef __TEST_H
#define __TEST_H

extern void test(void);

#endif

vim test.c //定义
#include <stdio.h>
void test(void)
{
        printf("%s\n", __func__);
}

vim main.c //调用
#include <stdio.h>
#include "test.h"

int main(void)
{
        test();//调用
        return 0;
}

编译：
arm-linux-gcc -fpic -shared -o libtest.so test.c
arm-linux-gcc -o main main.c -L. -ltest
mkdir /opt/rootfs/home/applib
cp libtest.so /opt/rootfs/home/applib
cp main /opt/rootfs/home/applib

开发板测试：
export LD_LIBRARY_PATH=/home/applib:$LD_LIBRARY_PATH
/home/applib/main

内核程序多文件的调用实现过程：
1.内核程序多文件的调用实现过程和应用程序多文件的调用实现
  过程一模一样:该声明的声明,该定义的定义,该调用的调用
2.还需要显式的进行符号(函数名或者变量名)的导出
  导出符号的宏：
  EXPORT_SYMBOL(函数名或者变量名);
  或者
  EXPORT_SYMBOL_GPL(函数名或者变量名);
  前者导出的变量和函数,不管其他内核程序是否添加：MODULE_LICENSE("GPL")
  都能访问调用;
  后者导出的变量和函数,只能给那些添加了MODULE_LICENSE("GPL")的内核程序访问

案例：编写内核程序,掌握内核的符号导出知识点
实施步骤：
虚拟机执行：
mkdir /opt/drivers/day01/4.0
cd /opt/drivers/day01/4.0
vim test.h 添加如下内容
#ifndef __TEST_H
```

```
#define __TEST_H

//函数声明
extern void test(void);

#endif
```

保存退出

```
vim test.c 添加如下内容
#include <linux/init.h>
#include <linux/module.h>

//函数定义
void test(void)
{
        printk("%s\n");
}

//显式的进行导出
EXPORT_SYMBOL(test);

EXPORT_SYMBOL_GPL(test);

//添加遵循GPL协议的信息
MODULE_LICENSE("GPL");
```

保存退出

```
vim helloworld.c 添加如下内容
#include <linux/init>
#include <linux/module.h>

static int helloworld_init(void)
{
        test(); //调用
        printk("%s\n", __func__);
        return 0;
}

static void helloworld_exit(void)
{
        test(); //调用
        printk("%s\n", __func__);
}

module_init(helloworld_init);
module_exit(helloworld_exit);

MODULE_LICENSE("GPL");
```

保存退出

```
修改Makefile,添加对test.c的编译支持
obj-m += helloworld.o test.o
或者
obj-m += helloworld.o
```

```
obj-m += test.o

make //开始编译
test.ko helloworld.ko

mkdir /opt/rootfs/home/drivers/ //创建驱动目标文件的存放目录
cp *.ko /opt/rootfs/home/drivers/
```

开发板测试：
1.insmod /home/drivers/?
2.insmod /home/drivers/?
3.rmmod ?
4.rmmod ?

案例：利用EXPORT_SYMBOL_GPL进行符号导出,做对比测试