以下是凌阳教育的嵌入式linux培训资深讲师-徐老师提供的免费的linux驱动基础开发教程（linux驱动基础开发0- linux驱动基础开发3）理论加实例详解，重点红字标注。linux初学者必看！

一：linux驱动基础开发0--linux 设备驱动概述

目前，Linux软件工程师大致可分为两个层次：

（1）Linux应用软件工程师（Application Software Engineer）：
主要利用C库函数和Linux API进行应用软件的编写；
从事这方面的开发工作，主要需要学习：符合linux posix标准的API函数及系统调用，linux的多任务编程技巧：多进程、多线程、进程间通信、多任务之间的同步互斥等，嵌入式数据库的学习，UI编程：QT、miniGUI等。

（2）Linux固件工程师（Firmware Engineer）：
主要进行Bootloader、Linux的移植及Linux设备驱动程序的设计工作。

一般而言，固件工程师的要求要高于应用软件工程师的层次，而其中的Linux设备驱动编程又是Linux程序设计中比较复杂的部分，究其原因，主要包括如下几个方面：

1）设备驱动属于Linux内核的部分，编写Linux设备驱动需要有一定的Linux操作系统内核基础；需要了解部分linux内核的工作机制与系统组成。

2）编写Linux设备驱动需要对硬件的原理有相当的了解，大多数情况下我们是针对一个特定的嵌入式硬件平台编写驱动的，例如：针对特定的主机平台：可能是三星的2410、2440，也可能是atmel的，或者飞思卡尔的等等。

3）Linux设备驱动中广泛涉及到多进程并发的同步、互斥等控制，容易出现bug；因为linux本身是一个多任务的工作环境，不可避免的会出现在同一时刻对同一设备发生并发操作。

4）由于属于内核的一部分，Linux设备驱动的调试也相当复杂。linux设备驱动没有一个很好的IDE环境进行单步、变量查看等调试辅助工具；linux驱动跟linux内核工作在同一层次，一旦发生问题，很容易造成内核的整体崩溃。

本系列文章我们将一步步、深入浅出的介绍linux设备驱动编程中设计的一些问题及学习方法，希望对大家学习linux设备驱动有所帮助。

在任何一个计算机系统中，大至服务器、PC机、小至手机、mp3/mp4播放器，无论是复杂的大型服务器系统还是一个简单的流水灯单片机系统，都离不开驱动程序的身影，没有硬件的软件是空中楼阁，没有软件的硬件只是一堆废铁，硬件是底层的基础，是所有软件得以运行的平台，代码最终会落实到硬件上的逻辑组合。

但是硬件与软件之间存在一个驳论：为了快速、优质的完成软件功能设计，应用程序工程师不想也不愿关心硬件，而硬件工程师也很难有功夫去处理软件开发中的一些应用。例如软件工程师在调用printf的时候，不许也不用关心信息到底是通过什么样的处理，走过哪些通路显示在该显示的

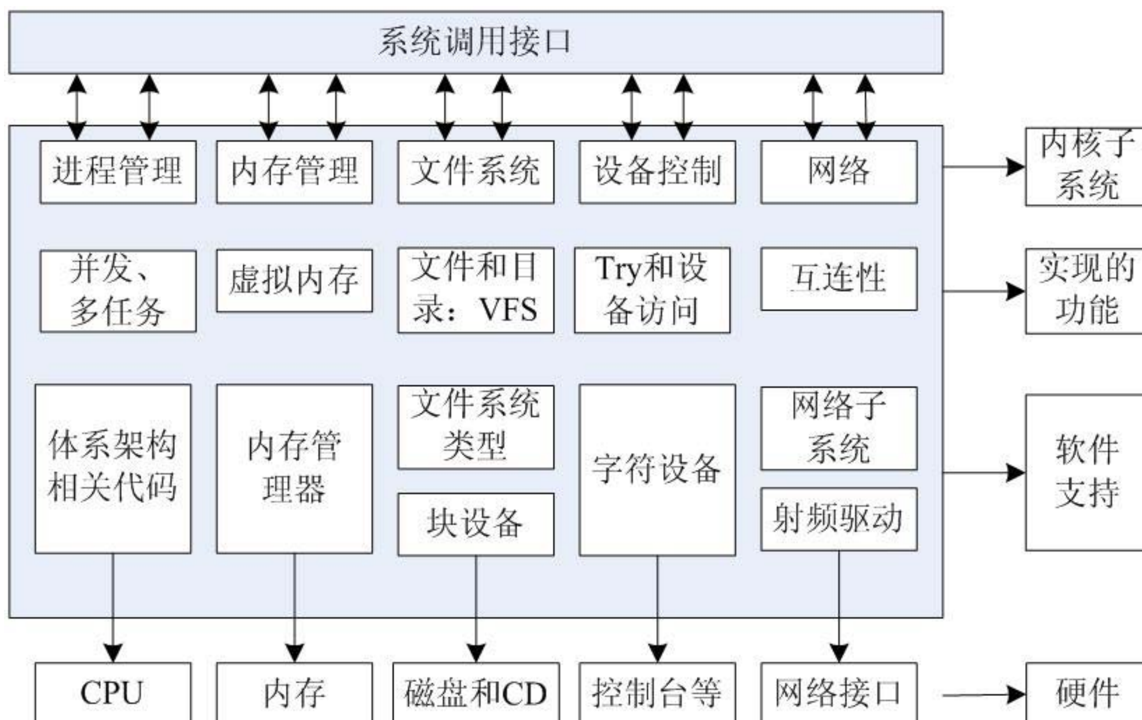
地方，硬件工程师在写完了一个4*4键盘驱动后，无需也不必管应用程序在获得键值后做哪些处理及操作。

也就是说软件工程师需要看到一个没有硬件的纯软件世界，硬件必须透明的提供给他，谁来实现这一任务？答案是驱动程序，驱动程序从字面解释就是：“驱使硬件设备行动”。驱动程序直接与硬件打交道，按照硬件设备的具体形式，驱动设备的寄存器，完成设备的轮询、中断处理、DMA通信，最终让通信设备可以收发数据，让显示设备能够显示文字和画面，让音频设备可以完成声音的存储和播放。

可见，设备驱动程序充当了硬件和软件之间的枢纽，因此驱动程序的表现形式可能就是一些标准的、事先协定好的API函数，驱动工程师只需要去完成相应函数的填充，应用工程师只需要调用相应的接口完成相应的功能。无论有没有操作系统，驱动程序都有其存在价值，只是在裸机情况下，工作环境比较简单、完成的工作较单一，驱动程序完成的功能也就比较简单，同时接口只要在小范围内符合统一的标准即可。但是在有操作系统的情况下，此问题就会被放大：硬件来自不同的公司、千变万化，全世界每天都会有大量的新芯片被生产，大量的电路板被设计出来，如果没有一个很好的统一标准去规范这一程序，操作系统就会被设计的非常冗余，效率会非常低。

所以无论任何操作系统都会制定一套标准的架构去管理这些驱动程序：linux作为嵌入式操作系统的典范，其驱动架构具有很高的规范性与聚合性，不但把不同的硬件设备分门别类、综合管理，并且针对不同硬件的共性进行了统一抽象，将其硬件相关性降到最低，大大简化了驱动程序的编写，形成了具有其特色的驱动组织架构。

下图反映了应用程序、linux内核、驱动程序、硬件的关系。



linux内核分为5大部分：多任务管理、内存管理、文件系统管理、设备管理、网络管理；
每一部分都有承上下的作用，对上提供API接口，提供给应用开发工程师使用；
对下通过驱动程序屏蔽不同的硬件构成，完成硬件的具体操作。

二：linux驱动基础开发 1——linux 设备驱动基本概念

学习linux设备驱动首先我们必须明确以下几个概念，为我们接下来学习linux驱动打下坚实的基础：

应用程序、库、内核、驱动程序的关系：

设备类型
设备文件、主设备号与从设备号
驱动程序与应用程序的区别
用户态与内核态
Linux驱动程序功能

一、应用程序、库、内核、驱动程序的关系

1）应用程序调用一系列函数库，通过对文件的操作完成一系列功能：

应用程序以文件形式访问各种硬件设备（linux特有的抽象方式，把所有的硬件访问抽象为对文件的读写、设置）

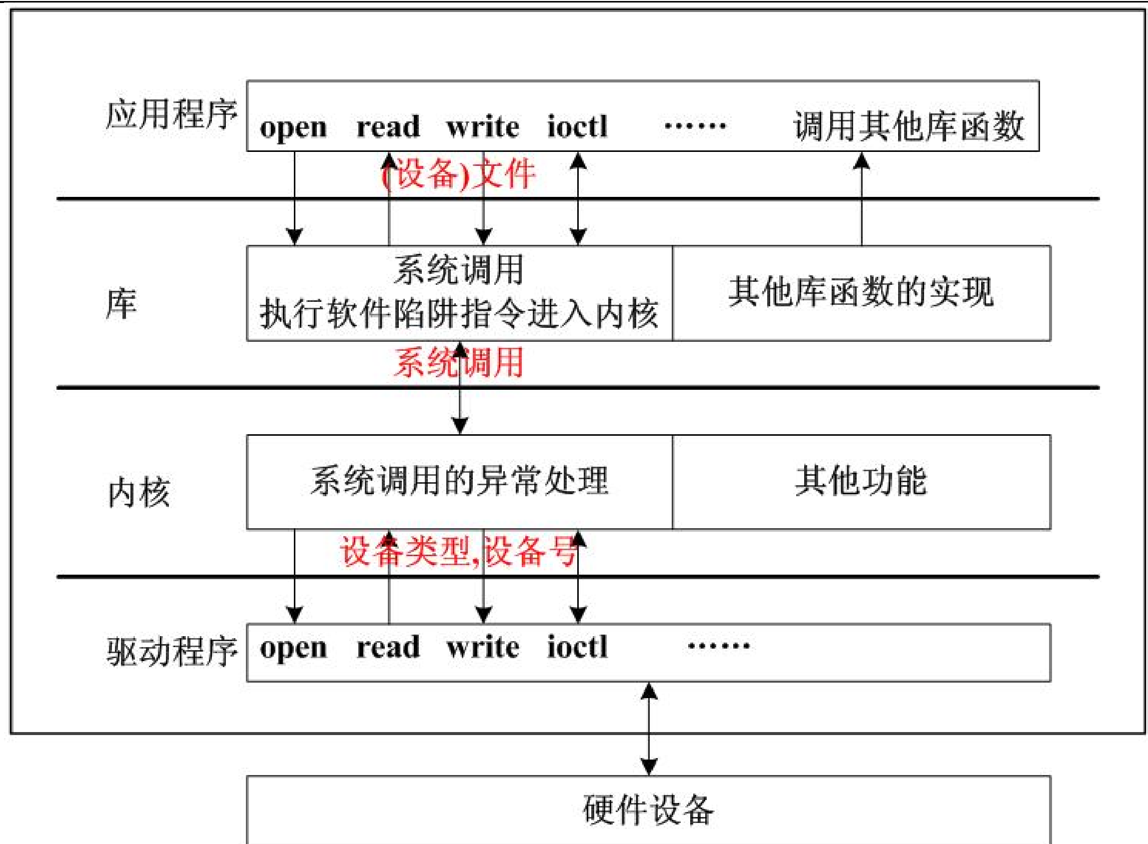
函数库：

部分函数无需内核的支持，由库函数内部通过代码实现，直接完成功能

部分函数涉及到硬件操作或内核的支持，由内核完成对应功能，我们称其为系统调用

2）内核处理系统调用，根据设备文件类型、主设备号、从设备号（后面会讲解），调用设备驱动程序；

3）设备驱动直接与硬件通信；



二、设备类型

硬件是千变万化的，没有八千也有一万了，就像世界上有三种人：男人、女人、女博士一样，linux做了一个很伟大也很艰难的分类：把所有的硬件设备分为三大类：字符设备、块设备、网络设备。

1）字符设备：字符（char）设备是个能够像字节流（类似文件）一样被访问的设备。

对字符设备发出读/写请求时，实际的硬件I/O操作一般紧接着发生；

字符设备驱动程序通常至少要实现open、close、read和write系统调用。

比如我们常见的lcd、触摸屏、键盘、led、串口等等，就像男人是用来干活的一样，他们一般对应具体的硬件都是进行出具的采集、处理、传输。

2）块设备：一个块设备驱动程序主要通过传输固定大小的数据（一般为512或1k）来访问设备。

块设备通过buffer cache（内存缓冲区）访问，可以随机存取，即：任何块都可以读写，不必考虑它在设备的什么地方。

块设备可以通过它们的设备特殊文件访问，但是更常见的是通过文件系统进行访问。

只有一个块设备可以支持一个安装的文件系统。

比如我们常见的电脑硬盘、SD卡、U盘、光盘等，就像女人一样是用来存储信息的。

3）网络接口：任何网络事务都经过一个网络接口形成，即一个能够和其他主机交换数据的设备。

访问网络接口的方法仍然是给它们分配一个唯一的名字（比如eth0），但这个名字在文件系统中不存在对应的节点。

内核和网络设备驱动程序间的通信，完全不同于内核和字符以及块驱动程序之间的通信，内核调用一套和数据包传输相关的函数（socket函数）而不是read、write等。

比如我们常见的网卡设备、蓝牙设备，就像女博士一样，数量稀少但又不可或缺。

linux中所有的驱动程序最终都能归到这三种设备中，当然他们之间也没有非常严格的界限，这些都是程序中对他们的划分而已，比如一个sd卡，我们也可以把它封装成字符设备去操作也是没有问题的。就像……

三、设备文件、主设备号、从设备号

有了设备类型的划分，那么应用程序应该怎样访问具体的硬件设备呢？

或者说已经确定他是一个男人了，那么怎么从万千世界中区分他与他的不同呢？

答案是：姓名，在linux驱动中也就是设备文件名。

那么重名怎么办？

答案是：身份证号，在linux驱动中也就是设备号（主、从）。

设备文件：

在linux 系统 中有一个约定俗成的说法：“一切皆文件”， 应用程序使用设备文件节点访问对应设备， Linux下的各种硬件设备以文件的形式存放于/dev目录下，可以使用ls /dev 查看 Linux把对硬件的操作全部抽象成对文件的操作 （open,read,write,close,…）

```
ous      loop3    parport0  ram14    sda   sg5       tty14  tty28
console  loop4    parport1  ram15    sda1  sg6       tty15  tty29
core     loop5    parport2  ram2     sda2  shm       tty16  tty3
disk     loop6    parport3  ram3     sda3  snapshot  tty17  tty30
fd       loop7    port      ram4     sda4  stderr    tty18  tty31
full     lp0      ppp       ram5     sda5  stdin     tty19  tty32
gpmctl   MAKEDEV  ptmx      ram6     sdb   stdout    tty2   tty33
hpet     mapper   pts       ram7     sdc   systty    tty20  tty34
initctl  md0      ram       ram8     sdd   tty       tty21  tty35
input    mem      ram0      ram9     sde   tty0      tty22  tty36
kmsg     net      ram1      ramdisk  sg0   tty1      tty23  tty37
log      null     ram10     random   sg1   tty10     tty24  tty38
```

每个设备文件都有其文件属性（c或者b），使用ls /dev -l 的命令查看， 表明其是字符设备或者块设备，网络设备没有在这个文件夹下，用来明其性别（男人、女人）

```
crw-rw-rw-  1 root root    1,   8 08-29 11:01 random
crw-------  1 root root  162,   0 08-29 11:01 rawctl
brw-------  1 root root    8,   2 08-29 19:01 root
crw-r--r--  1 root root   10, 135 08-29 19:01 rtc
brw-r-----  1 root disk    8,   0 08-29 19:01 sda
brw-r-----  1 root disk    8,   1 08-29 19:01 sda1
brw-r-----  1 root disk    8,   2 08-29 11:01 sda2
brw-r-----  1 root disk    8,   3 08-29 19:01 sda3
```

主设备号、从设备号

在设备管理中，除了设备类型外，内核还需要一对被称为主从设备号的参数，才能唯一标识一个设备，类似人的身份证号

主设备号：

用于标识驱动程序，相同的主设备号使用相同的驱动程序，例如：S3C2440 有串口、LCD、触摸屏三种设备，他们的主设备号各不相同；

从设备号：

用于标识同一驱动程序的不同硬件

例：PC的IDE设备，主设备号用于标识该硬盘，从设备号用于标识每个分区，2440有三个串口，每个串口的主设备号相同，从设备号用于区分具体属于那一个串口。

四、驱动程序与应用程序的区别

应用程序以main开始
驱动程序没有main，它以一个模块初始化函数作为入口
应用程序从头到尾执行一个任务
驱动程序完成初始化之后不再运行，等待系统调用
应用程序可以使用glibc等标准C函数库
驱动程序不能使用标准C库

五、用户态与内核态的区分

驱动程序是内核的一部分，工作在内核态
应用程序工作在用户态
数据空间访问问题
无法通过指针直接将二者的数据地址进行传递
系统提供一系列函数帮助完成数据空间转换
get_user
put_user
copy_from_user
copy_to_user

六、Linux驱动程序功能

对设备初始化和释放资源
把数据从内核传送到硬件和从硬件读取数据
读取应用程序传送给设备文件的数据和回送应用程序请求的数据
检测和处理设备出现的错误（底层协议）
用于区分具体设备的实例

三：linux驱动基础开发2——linux 驱动开发前奏（模块编程）

一、linux内核模块简介

linux内核整体结构非常庞大，其包含的组件也非常多。我们怎么把需要的部分都包含在内核中呢？

一种办法是把所有的需要的功能都编译到内核中。这会导致两个问题，一是生成的内核会很大，二是如果我们要在现有的内核中新增或删除功能，不得不重新编译内核，工作效率会非常的低，同时如果编译的模块不是很完善，很有可能会造成内核崩溃。

linux提供了另一种机制来解决这个问题，这种集中被称为模块，可以实现编译出的内核本身并不含有所有功能，而在这些功能需要被使用的时候，其对应的代码可以被动态的加载到内核中。

二、模块特点：

1）模块本身并不被编译入内核，从而控制了内核的大小。
2）模块一旦被加载，他就和内核中的其他部分完全一样。

注意：模块并不是驱动的必要形式：即：驱动不一定必须是模块，有些驱动是直接编译进内核的；同时模块也不全是驱动，例如我们写的一些很小的算法可以作为模块编译进内核，但它并不是驱动。就像烧饼不一定是圆的，圆的也不都是烧饼一样。

三、最简单的模块分析

1）以下是一个最简单的模块例子

```
1. #include <linux/init.h>          /* printk() */
2. #include <linux/module.h>        /* __init __exit */
3.
4. static int  __init  hello_init(void)       /*模块加载函数，通过 insmod
   命令加载模块时，被自动执行*/
5. {
6.   printk(KERN_INFO " Hello World enter\n");
7.   return 0;
8. }
9. static void  __exit  hello_exit(void)     /*模块卸载函数，当通过 rmmod
   命令卸载时，会被自动执行*/
10. {
11.   printk(KERN_INFO " Hello World exit\n ");
12. }
13.
14. module_init(hello_init);
15. module_exit(hello_exit);
16.
17. MODULE_AUTHOR("dengwei");             /*模块作者，可选*/
18. MODULE_LICENSE("Dual BSD/GPL");       /*模块许可证明，描述内核模块的许可
   权限，必须*/
19.
20. MODULE_DESCRIPTION("A simple Hello World Module"); /*模块说明，可选
   */
21. MODULE_ALIAS("a simplest module");                /*模块说明,可选
   */<span style="font-family:SimSun;font-size:18px;color:#FF0000;">
   <strong>
22. </strong></span>
```

2） 以下是编译上述模块所需的编写的makefile

```
1. obj-m :=hello.o                        //目标文件
2. #module-objs := file1.o file.o         //当模块有多个文件组成时,添加本句
3. KDIR :=/usr/src/linux                  //内核路径，根据实际情况换成自己的内
   核路径，嵌入式的换成嵌入式，PC 机的指定 PC 机路径
4. PWD := $(shell pwd)                    //模块源文件路径
5. all:
6.     $(MAKE)  -C  $(KDIR)  SUBDIRS=$(PWD)  modules
7.     @rm -rf *.mod.*
```

```
8.      @rm -rf .*.cmd
9.      @rm -rf *.o
10.      @rm -rf Module.*
11. clean:
12.      rm -rf *.ko
```

最终会编译得到：hello.ko文件
使用insmodhello.ko将模块插入内核，然后使用dmesg即可看到输出提示信息。

常用的几种模块操作：
insmod XXX.ko　　加载指定模块
lsmod　　　　　　　　　　列举当前系统中的所有模块
rmmod　XXX　　　　卸载指定模块（注意没有。ko后缀）
dmesg　　　　　　　　　　当打印等级低于默认输出等级时，采用此命令查看系统日志

3）linux内核模块的程序结构
1.模块加载函数：
Linux内核模块一般以__init标示声明，典型的模块加载函数的形式如下：

```
1. static int __init myModule_init(void)
2. {
3.     /* Module init code */
4.     PRINTK("myModule_init\n");
5.     return 0;
6. }
7. module_init(myModule_init);
```

模块加载函数的名字可以随便取，但必须以"module_init（函数名）"的形式被指定；
执行insmod命令时被执行，用于初始化模块所必需资源，比如内存空间、硬件设备等；
它返回整形值，若初始化成功，应返回0，初始化失败返回负数。
2.模块卸载函数
典型的模块卸载函数形式如下：

```
1. static void __exit myModule_exit(void)
2. {
3.     /* Module exit code */
4.     PRINTK("myModule_exit\n");
5.     return;
6. }
7. module_exit(myModule_exit);
```

模块卸载函数在模块卸载的时候执行，不返回任何值，需用"module_exit（函数名）"的形式被指定。

卸载模块完成与加载函数相反的功能：
若加载函数注册了XXX，则卸载函数应当注销XXX
若加载函数申请了内存空间，则卸载函数应当释放相应的内存空间
若加载函数申请了某些硬件资源（中断、DMA、I/O端口、I/O内存等），则卸载函数应当释放相应的硬件资源
若加载函数开启了硬件，则卸载函数应当关闭硬件。
其中__init 、__exit 为系统提供的两种宏，表示其所修饰的函数在调用完成后会自动回收内存，即内核认为这种函数只会被执行1次，然后他所占用的资源就会被释放。

3.模块声明与描述
在linux内核模块中，我们可以用MODULE_AUTHOR、MODULE_DESCRIPTION、MODULE_VERSION、MODULE_TABLE、MODULE_ALIA，分别描述模块的作者、描述、版本、设备表号、别名等。

```
1. MODULE_AUTHOR("dengwei");
2. MODULE_LICENSE("Dual BSD/GPL");
3. MODULE_DESCRIPTION("A simple Hello World Module");
4. MODULE_ALIAS("a simplest module");
```

四、有关模块的其它特性
1）模块参数：

我们可以利用module_param（参数名、参数类型、参数读写属性） 为模块定义一个参数，例如：

```
1.    static char *string_test = “this is a test”;
2. static num_test = 1000;
3. module_param (num_test,int,S_IRUGO);
4. module_param (steing_test,charp,S_ITUGO);
```

在装载模块时，用户可以给模块传递参数，形式为：“insmod 模块名 参数名=参数值”，如果不传递，则参数使用默认的参数值
参数的类型可以是：byte,short,ushort,int,uint,long,ulong,charp,bool;
权限：定义在linux/stat.h中，控制存取权限，S_IRUGO表示所有用户只读;
模块被加载后，在sys/module/下会出现以此模块命名的目录，当读写权限为零时：表示此参数不存在sysfs文件系统下的文件节点，当读写权限不为零时：此模块的目录下会存在parameters目录，包含一系列以参数名命名的文件节点，这些文件节点的权限值就是传入module_param（）的“参数读/写权限”，而该文件的内容为参数的值。
除此之外，模块也可以拥有参数数组，形式为：“module_param_array（数组名、数组类型、数组长、参数读写权限等）”，当不需要保存实际的输入的数组元素的个数时，可以设置“数组长”为0。
运行insmod时，使用逗号分隔输入的数组元素。
下面是一个实际的例子，来说明模块传参的过程。

```
1. #include <linux/module.h>     /*module_init()*/
2. #include <linux/kernel.h> /* printk() */
3. #include <linux/init.h>        /* __init __exit */
4.
5. #define DEBUG   //open debug message
```

```
6.
7. #ifdef DEBUG
8. #define PRINTK(fmt, arg...)     printk(KERN_WARNING fmt, ##arg)
9. #else
10. #define PRINTK(fmt, arg...)     printk(KERN_DEBUG fmt, ##arg)
11. #endif
12.
13. static char *string_test="default paramater";
14. static int num_test=1000;
15.
16. static int __init hello_init(void)
17. {
18.   PRINTK("\nthe  string_test is : %s\n",string_test);
19.   PRINTK("the  num_test is : %d\n",num_test);
20.   return 0;
21. }
22.
23. static void __exit hello_exit(void)
24. {
25.   PRINTK(" input paramater module exit\n ");
26. }
27.
28. module_init(hello_init);
29. module_exit(hello_exit);
30.
31. module_param(num_test,int,S_IRUGO);
32. module_param(string_test,charp,S_IRUGO);
33.
34. MODULE_AUTHOR("dengwei");
35. MODULE_LICENSE("GPL");
```

当执行 insmod hello_param.ko时，执行dmesg 查看内核输出信息：

```
1. Hello World enter
2. the test string is:  this is a test
3. the test num is : 1000
```

当执行insmod hello_param.ko num_test=2000 string_test="edit by dengwei"，执行dmesg 查看内核输出信息：

```
1. Hello World enter
```

```
2. the test string is:  edit by dengwei
3. the test num is : 2000
```

2）导出模块及符号的相互引用

Linux2.6内核的“/proc/kallsyms”文件对应内核符号表，它记录了符号以及符号所在的内存地址，模块可以使用下列宏导到内核符号表中。

EXPORT_SYMBOL（符号名）；   任意模块均可

EXPORT_SYMBOL_GPL（符号名）；   只使用于包含GPL许可权的模块

导出的符号可以被其它模块使用，使用前声明一下即可。

下面给出一个简单的例子：将add sub符号导出到内核符号表中，这样其它的模块就可以利用其中的函数

```
1. #include <linux/module.h>     /*module_init()*/
2. #include <linux/kernel.h> /* printk() */
3. #include <linux/init.h>        /* __init __exit */
4.
5. int add_test(int a ,int b)
6. {
7.     return a + b;
8. }
9.
10. int sub_test(int a,int b)
11. {
12.     return a - b;
13. }
14.
15. EXPORT_SYMBOL(add_test);
16. EXPORT_SYMBOL(sub_test);
17.
18. MODULE_AUTHOR("dengwei");
19. MODULE_LICENSE("GPL");
```

执行 cat/proc/kallsyms | grep test 即可找到以下信息，表示模块确实被加载到内核表中。

```
1.    f88c9008 r __ksymtab_sub_integar        [export_symb]
2. f88c9020 r __kstrtab_sub_integar         [export_symb]
3. f88c9018 r __kcrctab_sub_integar         [export_symb]
4. f88c9010 r __ksymtab_add_integar        [export_symb]
5. f88c902c r __kstrtab_add_integar          [export_symb]
6. f88c901c r __kcrctab_add_integar         [export_symb]
```

```
7. f88c9000 T add_tes                      [export_symb]
8. f88c9004 T sub_tes                      [export_symb]
9. 13db98c9 a __crc_sub_integar                 [export_symb]
10. e1626dee a __crc_add_integar                 [export_symb]
```

在其它模块中可以引用此符号

```
1.    #include <linux/module.h>    /*module_init()*/
2. #include <linux/kernel.h> /* printk() */
3. #include <linux/init.h>       /* __init __exit */
4.
5. #define DEBUG   //open debug message
6.
7. #ifdef DEBUG
8. #define PRINTK(fmt, arg...)     printk(KERN_WARNING fmt, ##arg)
9. #else
10.#define PRINTK(fmt, arg...)     printk(KERN_DEBUG fmt, ##arg)
11.#endif
12.
13.extern int add_test(int a ,int b);
14.extern int sub_test(int a,int b);
15.
16.static int __init hello_init(void)
17.{
18.    int a,b;
19.
20.   a = add_test(10,20);
21.   b = sub_test(30,20);
22.   PRINTK("the add test result is %d",a);
23.   PRINTK("the sub test result is %d\n",b);
24.   return 0;
25.}
26.
27.static void __exit hello_exit(void)
28.{
29.   PRINTK(" Hello World exit\n ");
30.}
31.
32.module_init(hello_init);
33.module_exit(hello_exit);
```

```
34.
35. MODULE_AUTHOR("dengwei");
36. MODULE_LICENSE("GPL");
```

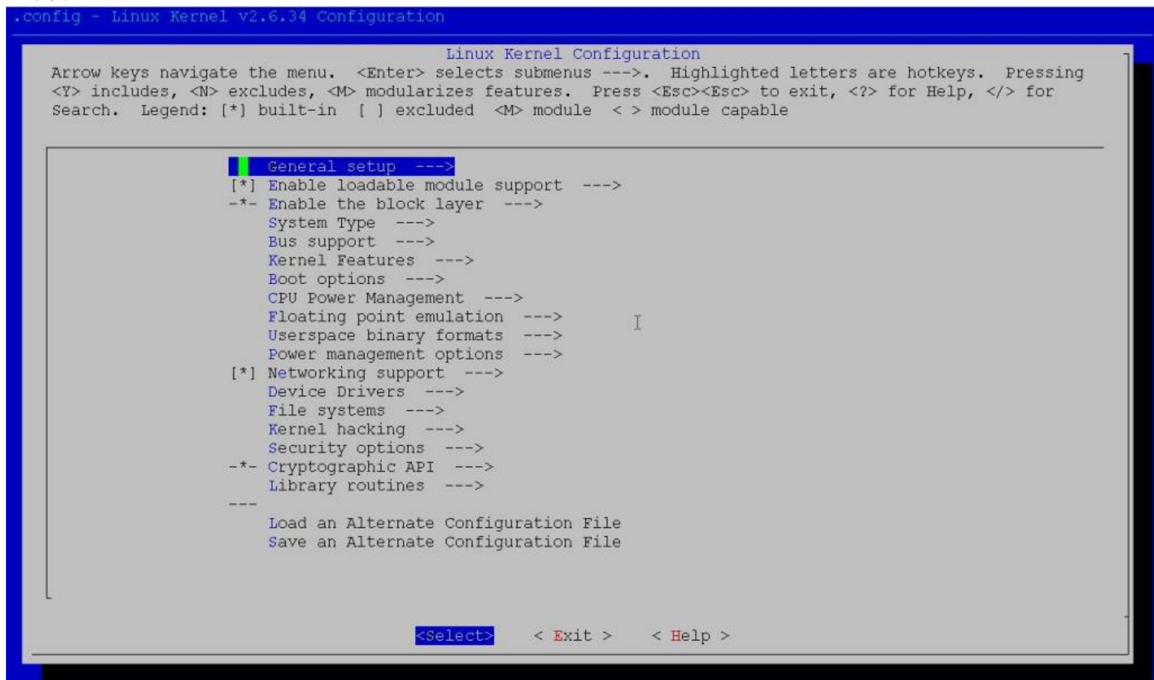
四：linux 驱动基础开发 3——linux 内核配置机制（make menuconfig、Kconfig、makefile）讲解

前面我们介绍模块编程的时候介绍了驱动进入内核有两种方式：模块和直接编译进内核，并介绍了模块的一种编译方式——在一个独立的文件夹通过 makefile 配合内核源码路径完成。

那么如何将驱动直接编译进内核呢？

在我们实际内核的移植配置过程中经常听说的内核裁剪又是怎么麽回事呢？

我们在进行 linux 内核配置的时候经常会执行 make menuconfig 这个命令，然后屏幕上会出现以下界面：



这个界面是怎么生成的呢？

跟我们经常说的内核配置与与编译又有什么关系呢？

下面我们借此来讲解一下 linux 内核的配置机制及其编译过程。

一、配置系统的基本结构

Linux 内核的配置系统由三个部分组成，分别是：

1、Makefile：分布在 Linux 内核源代码根目录及各层目录中，定义 Linux 内核的编译规则；

2、配置文件（config.in（2.4 内核，2.6 内核））：给用户提供配置选择的功能；

3、配置工具：包括配置命令解释器（对配置脚本中使用的配置命令进行解释）和配置用户界面（提供基于字符界面、基于 Ncurses 图形界面以及基于 Xwindows 图形界面的用户配置界面，各自对应于 Make config、Make menuconfig 和 make xconfig）。

这些配置工具都是使用脚本语言，如 Tcl/TK、Perl 编写的（也包含一些用 C 编写的代码）。本文并不是对配置系统本身进行分析，而是介绍如何使用配置系统。所以，除非是配置系统的维护者，一般的内核开发者无须了解它们的原理，只需要知道如何编写 Makefile 和配置文件就可以。

二、makefile menuconfig 过程讲解

当我们在执行 make menuconfig 这个命令时，系统到底帮我们做了哪些工作呢？
这里面一共涉及到了一下几个文件我们来一一讲解

Linux 内核根目录下的 scripts 文件夹
arch/$ARCH/Kconfig 文件、各层目录下的 Kconfig 文件
Linux 内核根目录下的 makefile 文件、各层目录下的 makefile 文件
Linux 内核根目录下的的。config 文件、arm/$ARCH/下的 config 文件
Linux 内核根目录下的 include/generated/autoconf.h 文件

1）scripts 文件夹存放的是跟 make menuconfig 配置界面的图形绘制相关的文件，我们作为使用者无需关心这个文件夹的内容

2）当我们执行 make menuconfig 命令出现上述蓝色配置界面以前，系统帮我们做了以下工作：

首先系统会读取 arch/$ARCH/目录下的 Kconfig 文件生成整个配置界面选项（Kconfig 是整个 linux 配置机制的核心），那么 ARCH 环境变量的值等于多少呢？

它是由 linux 内核根目录下的 makefile 文件决定的，在 makefile 下有此环境变量的定义：

```
export KBUILD_BUILDHOST := $(SUBARCH)
ARCH            ?= arm
CROSS_COMPILE   ?=/usr/local/arm/4.3.2/bin/arm-linux-
```

或者通过 make ARCH=arm menuconfig 命令来生成配置界面，默认生成的界面是所有参数都是没有值的

比如教务处进行考试，考试科数可能有外语、语文、数学等科，这里相当于我们选择了 arm 科可进行考试，系统就会读取 arm/arm/kconfig 文件生成配置选项（选择了 arm 科的卷子），系统还提供了 x86 科、milps 科等 10 几门功课的考试题

4） 假设教务处比较“仁慈”，为了怕某些同学做不错试题，还给我们准备了一份参考答案（默认配置选项），存放在 arch/$ARCH/configs 下，对于 arm 科来说就是 arch/arm/configs 文件夹：

5）

```
acs5k_defconfig              ep93xx_defconfig
acs5k_tiny_defconfig         eseries_pxa_defconfi
afeb9260_defconfig           ezx_defconfig
am200epdkit_defconfig        footbridge_defconfig
am3517_evm_defconfig         fortunet_defconfig
ams_delta_defconfig          g3evm_defconfig
ap4evb_defconfig             g4evm_defconfig
assabet_defconfig            h3600_defconfig
at572d940hfek_defconfig      h5000_defconfig
at91cap9adk_defconfig        h7201_defconfig
at91rm9200dk_defconfig       h7202_defconfig
```

此文件夹中有许多选项，系统会读取哪个呢？内核默认会读取 linux 内核根目录下。config 文件作为内核的默认选项（试题的参考答案），我们一般会根据开发板的类型从中选取一个与我们开发板最接近的系列到 Linux 内核根目录下（选择一个最接近的参考答案）

#cp arch/arm/configs/s3c2410_defconfig .config

4）config

假设教务处留了一个心眼，他提供的参考答案并不完全正确（。config 文件与我们的板子并不是完全匹配），这时我们可以选择直接修改。config 文件然后执行 make menuconfig 命令读取新的选项

但是一般我们不采取这个方案，我们选择在配置界面中通过空格、esc、回车选择某些选项选中或者不选中，最后保存退出的时候，Linux 内核会把新的选项（正确的参考答案）更新到。config 中，此时我们可以把。config 重命名为其它文件保存起来（当你执行 make distclean 时系统会把。config 文件删除），以后我们再配置内核时就不需要再去 arch/arm/configs 下考取相应的文件了，省去了重新配置的麻烦，直接将保存的。config 文件复制为。config 即可。

5）经过以上两步，我们可以正确的读取、配置我们需要的界面了

那么他们如何跟 makefile 文件建立编译关系呢？

当你保存 make menuconfig 选项时，系统会除了会自动更新。config 外，还会将所有的选项以宏的形式保存在

Linux 内核根目录下的 include/generated/autoconf.h 文件下

```
#define AUTOCONF_INCLUDED
#define CONFIG_VIDEO_V4L1_COMPAT 1
#define CONFIG_USB_LEGOTOWER_MODULE 1
#define CONFIG_HID_CHERRY 1
#define CONFIG_INET_XFRM_TUNNEL_MODULE 1
#define CONFIG_FRAME_WARN 1024
#define CONFIG_VIDEO_EM28XX_ALSA 1
#define CONFIG_TCP_CONG_ADVANCED 1
#define CONFIG_CPU_S3C244X 1
#define CONFIG_COMPAT_BRK 1
#define CONFIG_DEFAULT_SECURITY_DAC 1
#define CONFIG_TCP_CONG_SCALABLE_MODULE 1
#define CONFIG_VIDEO_HELPER_CHIPS_AUTO 1
#define CONFIG_IP6_NF_TARGET_REJECT_MODULE 1
#define CONFIG_NF_CT_ACCT 1
#define CONFIG_TCP_CONG_VENO_MODULE 1
#define CONFIG_CPU_COPY_V4WB 1
#define CONFIG_DEBUG_USER 1
```

内核中的源代码就都会包含以上。h 文件，跟宏的定义情况进行条件编译。

当我们需要对一个文件整体选择如是否编译时，还需要修改对应的 makefile 文件，例如：

我们选择是否要编译 s3c2410_ts.c 这个文件时，makefile 会根据 CONFIG_TOUCHSCREEN_S3C2410 来决定是编译此文件，此宏是在 Kconfig 文件中定义，当我们配置完成后，会出现在。config 及 autconf 中，至此，我们就完成了整个 linux 内核的编译过程。

```
obj-$(CONFIG_TOUCHSCREEN_PCAP)          += pcap_ts.o
obj-$(CONFIG_TOUCHSCREEN_PENMOUNT)      += penmount.o
obj-$(CONFIG_TOUCHSCREEN_S3C2410)       += s3c2410_ts.o
obj-$(CONFIG_TOUCHSCREEN_TOUCHIT213)    += touchit213.o
```

最后我们会发现，整个 linux 内核配置过程中，留给用户的接口其实只有各层 Kconfig、makefile 文件以及对应的源文件。

比如我们如果想要给内核增加一个功能，并且通过 make menuconfig 控制其声称过程

首先需要做的工作是：修改对应目录下的 Kconfig 文件，按照 Kconfig 语法增加对应的选项；

其次执行 make menuconfig 选择编译进内核或者不编译进内核，或者编译为模块，。config 文件和 autoconf.h 文件会自动生成；

最后修改对应目录下的 makefile 文件完成编译选项的添加；

最后的最后执行 make zImage 命令进行编译。

三、具体实例

下面我们以前面做过的模块实验为例，讲解如何通过 make menuconfig 机制将前面单独编译的模块编译进内核或编译为模块

假设我已经有了这么一个驱动：

modules.c

Step1：将 modules.c 拷到 drivers/char/目录下（这个文件夹一般存放常见的字符驱动）
Step2: vi driver/char/Kconfig,在
config DEVKMEM 后添加以下信息
config MODULES
tristate “modules device support”
default y
help
Say Y here,the modules will be build in kernel.
Say M here,the modules willbe build to modules.
Say N here,there will be nothing to be do.
Step3：make menuconfig
Device driver-character devices
[*]modules device suppor
Step4:vi driver/char/Makefile，在 js-rtc 后添加
obj-$（CONFIG_MODULES）+= modules.o
CONFIG_MODULES 必须跟上面的 Kconfig 中保持一致，系统会自动添加 CONFIG_前缀
modules.o 必须跟你加入的。c 文件名一致

最后执行：make zImage modules 就会被编译进内核中

第三步：
Step3：make menuconfig
Device driver-character devices
[M]modules device suppor
把星号在配置界面通过空格改为 M，最后执行 make modules，在 driver/char/目录下会生成一个 modules.ko 文件

跟我们前面讲的单独编译模块效果一样，也会生成一个模块，将它考入开发板执行 insmod moudles.ko，即可将生成的模块插入内核使用。

以上 linux 驱动基础开发系列教程由凌阳教育的嵌入式 linux 培训老师-徐老师提供！更多 linux 学习交流请关注>> http://bbs.sunplusedu.com/
http://blog.csdn.net/xdw1985829

：