嵌入式笔记 - 第七章

> *本章主要介绍了Linux驱动*

Linux驱动的基本概念

设备驱动程序是应用程序和实际设备之间的一个软件层，它向下负责和硬件设备的交互，向上通过一个通用的接口挂接到文件系统上，从而使用户或应用程序可以按操纵普通文件的方式进行访问控制硬件设备。

驱动程序为硬件提供一个定义良好的内部接口；

驱动程序封装了硬件细节；

驱动程序为应用程序提供了访问设备的机制；

驱动程序是内核的一部分。

驱动的基本功能

对设备初始化和释放。

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。

检测错误和处理中断

驱动程序与应用程序的区别

应用程序是从头到尾执行单个任务;

驱动程序只是预先注册自己以便服务于将来的某个请求。

应用程序可以调用它未定义的函数，因为在连接过程能够解析外部引用从而使用适当的函数库;

驱动程序仅仅被连接到内核，因此它仅能调用由内核导出的函数，而没有任何可连接的库。

应用程序开发过程中的段错误是无害的，并且总是可以使用调试器跟踪到源代码中的问题所在;

驱动程序的一个错误即使不对整个系统是致命的，也至少会对当前进程造成致命错误。

应用程序运行于用户空间，处理器禁止其对硬件的直接访问以及对内存的未授权访问;内核模块运行于内核空间，可以进行所有操作；

用户进程调用驱动程序时,系统进入核心态,这时不再是抢先式调度.也就是说,系统必须在驱动程序的子函数返回后才能进行其他的工作。

Linux驱动的分类

字符设备

字符设备是以字节为单位逐个进行 I/O 操作的设备。 字符设备通常指像普通文件或字节流一样，以字节为单位顺序读写的设备， 如并口设备、虚拟控制台等。字符设备可以通过设备文件节点访问，它与普通文件之间的区别在于普通文件可以被随机访问（可以前后移动访问指针），而大多数字符设备只能提供顺序访问。

块设备

块设备主要是针对磁盘等慢速设备设计的，其目的是避免耗费过多的 CPU 时间来等待操作的完成。它利用一块系统内存作为缓冲区，当用户进程对设备进行读写请求时，驱动程序先查看缓冲区中的内容，如果缓冲区中的数据能满足用户的要求就返回相应的数据，否则就调用相应的请求函数来进行实际的 I/O 操作。
如硬盘，光驱，存储卡等。

网络设备

网络设备是一个能够和其他主机交换数据的设备，它通常是个物理设备，但也可能是个软件设备，如回环设备（loopback）。网络驱动程序负责驱动设备发送和接收数据包。

如网卡等。

设备文件与设备文件系统

Linux中，字符设备和块设备都是通过文件节点进行访问。

每个设备对应一个文件名，操作时对应各自的驱动程序。

设备号

内核还使用了一个主设备号和一个次设备号来唯一标识设备。

主设备号表明某一类设备，标识了设备所对应的驱动程序。

次设备号仅由驱动程序解释，一般用于识别在若干可能的硬件设备中，I/O 请求所涉及到的那个设备。

如系统中IDE硬盘的主设备号是3，而多个硬盘及各个分区分别赋予次设备号1，2，3...

设备节点

访问一个设备，需要制定一个设备的标识符。在linux系统中，这个标识符一般是位于/dev下的文件，称为**设备节点**。

创建设备节点 mknod

`mknod 设备名 设备类型 主设备号 次设备号`

Linux设备驱动与内核

在 Linux 中将驱动程序嵌入内核有两种方式：

1、静态编译链接进内核

采用静态编译链接方式，需要把驱动程序的源代码放在内核源代码目录“Drivers/”下，并修改 Makefile 文件。在编译链接内核的时候，驱动程序会作为内核的一部分链接到内核镜像文件中。这种方式会增加内核的大小，还要改动内核的源文件，而且不利于调试。

2、先编译成模块，再动态加载进内核

模块（module）在 Linux 中是一种已经编译好的目标文件，它可以被链接进内核从而生成可执行的机器代码。如果采用模块加载的方式，驱动程序首先会被编译成未链接的目标文件，具有 root 权限的用户在需要时可利用`insmod`命令将其动态的加载到内核中，而在不需要的时候可利用`rmmod`命令卸载该模块。`lsmod`列出当前系统中加载的模块。

访问Linux驱动

设备提供**dev文件系统节点**和**proc文件系统节点**

应用程序通过**dev文件节点访问驱动程序**

- 字符型驱动一般通过标准的文件I/O访问
- 块设备在上层加载文件系统，比如以FAT32的形式访问
- 网络设备通过SOCKET来访问

应用程序通过**proc文件节点**可以**查询设备驱动的信息**

proc文件系统

/proc文件系统是一个伪文件系统，是一种内核和内核模块向进程发送信息的机制。这个伪文件系统让用户可以和内核内部数据结构进行交互，获取有关进程的有用信息。

/proc存在于内存中而不是硬盘中。

系统中当前运行的每个进程都对应于/proc下一个目录，目录名为进程号。

加载模块成功后，可以在文件/proc/devices中获取主设备号。

Linux驱动的结构

Linux 的设备驱动程序按实现的功能大致可以分为如下几个部分：

- 驱动程序的注册与注销
- 设备的打开与释放
- 设备的读写操作
- 设备的控制操作
- 设备的中断和轮询处理

驱动程序的注册与注销

向系统增加一个驱动程序意味着要赋予它一个主设备号，这可以通过在驱动程序的初始化过程中调用 register_chrdev（）或register_blkdev（）来完成。

而在关闭字符设备或者块设备时，则需要通过调用 unregister_chrdev（）或 unregister_blkdev（）从内核中注销设备，同时释放占用的主设备号。

设备的打开

打开设备是通过调用 file_operations 结构中的函数 open()来完成的，它是驱动程

序用来为今后的操作完成初始化准备工作的。在大部分驱动程序中，open()通常需要完成下列工作：

1.检查设备相关错误，如设备尚未准备好等。

2.如果是第一次打开，则初始化硬件设备。

3.识别次设备号，如果有必要则更新读写操作的当前位置指针 f_ops。

4.分配和填写要放在 file->private_data 里的数据结构。

5.使用计数增1。

设备的关闭

释放设备是通过调用 file_operations 结构中的函数 release()来完成的，这个设备方法有时也被称为 close()，它的作用正好与 open()相反，通常要完成下列工作：

1.使用计数减1。

2.释放在 file->private_data 中分配的内存。

3.如果使用计数为0，则关闭设备。

设备的读写

字符设备的读写操作相对比较简单，直接使用函数 read()和 write()就可以了。但如果是块设备的话，则需要调用函数 block_read()和 block_write()来进行数据读写，这两个函数将在设备请求表中增加读写请求，以便 Linux 内核可以对请求顺序进行优化。由于是对内存缓冲区而不是直接对设备进行操作的，因此能够很大程度上加快读写速度。

设备的控制操作

除了读写操作外，应用程序有时还需要对设备进行控制，这可以通过设备驱动程序中的函数 ioctl()来完成。ioctl()的用法与具体设备密切关联，因此需要根据设备的实际情况进行具体分析。

设备的中断和轮询处理

对于不支持中断的硬件设备，读写时需要轮流查询设备状态，以便决定是否继续进行数据传输。如果设备支持中断，则可以按中断方式进行操作。

驱动程序开发

模块化驱动程序的设计和实现流程主要有编写**模块化编程子程序**、编写**自动配置和初始化子程序**、编写**服务于I/O请求的子程序**和编写**中断服务子程序**四个步骤。

模块化编程子程序

模块化编程子程序主要包括 init_module()函数和cleanup_module()函数。

init_module()函数在模块被**加载**到内核时调用，其主要负责向内核注册模块所提供的任何新功能。新功能可能是一个完整的驱动程序或者仅仅是一个新的软件抽象。对于每种新功能，init_module()函数会调用与其相对应的内核函数完成注册。
对于字符型设备，init_module()函数对应的内核函数的函数原型为：
int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);
其中，major是为设备驱动程序向系统申请的主设备号，如果为0，则系统为此驱动程序动态地分配一个主设备号。name是设备名，fops 是指向file_operations结构的指针。如果register_chrdev()操作成功，驱动程序会注册到内核中，设备名也会出现在/proc/devices文件里。成功向系统注册了设备驱动程序后，就可以使用**mknod**命令来将设备映射为一个设备文件，其它程序使用这个设备的时候只要对此设备文件进行操作就行了。

cleanup_module()函数在模块被**卸载**的时候调用，其主要负责从内核中注销模

块所提供的各种功能。cleanup_module()函数会调用与其相对应的内核函数完成注销，对于字符型设备，cleanup_module ()函数对应的内核函数的函数为unregister_chrdev()。

自动配置和初始化子程序

自动配置和初始化子程序一般在设备接入系统时或者加载设备驱动时调用，其主要负责检测所要驱动的硬件设备是否存在和是否能正常工作。如果该设备正常，则对这个设备及其相关的驱动程序需要的软件状态进行初始化。自动配置和初始化子程序还负责为驱动程序申请包括内存、中断、I/O 端口等资源，这些资源也可以在xxx_open()函数中申请。

服务于 I/O 请求的子程序

服务于 I/O 请求的子程序，又称为驱动程序的上半部分，调用这部分是由于系统调用的结果。这部分程序在执行的时候，系统仍认为是和进行调用的进程属于同一个进程，只是进程的运行状态由用户态变成了核心态，具有进行此系统调用的用户程序的运行环境。驱动程序所提供的与设备的打开、释放、读写和控制操作相对应的入口点函数都属于服务于 I/O 请求的子程序，并且通过file_operations结构向系统进行说明。
file_operations结构是一个函数指针表，这个表中的每一项都指向由驱动程序实现的处理相应请求的函数。

中断服务子程序

中断服务子程序，又称为驱动程序的下半部分。在 Linux系统中，并不是直接从中断向量表中调用设备驱动程序的中断服务子程序，而是由 Linux系统接收硬件中断，再由系统调用中断服务子程序。中断服务程序被调用的时候，不能依赖于任何

进程的状态，也就不能调用任何与进程运行环境相关的函数。
设备驱动程序在设备第一次打开、硬件被告知产生中断之前调用request_irq()函数来申请中断，在最后一次关闭、硬件设备被告知不再用中断处理器后调用free_irq()函数来释放中断。

应用程序对设备的操作

应用程序对一个设备文件进行操作的流程大致如下：
1.用户程序通过 Linux 所提供的系统调用如 open()、read()等进入内核。
2.内核里对应这些系统调用的函数是 sys_open()、sys_read()等，它们由内核空间里的虚拟文件系统层实现。虚拟文件系统 VFS是一个从普通文件和设备文件抽象出来的一个文件系统层，完成了进入具体的设备文件的操作之前的准备工作。
3.Linux内核将通过file_operations结构进入具体的设备文件的操作函数，这部分的函数就是由设备驱动程序所提供的。这些函数负责对设备硬件进行各种操作。

示例：一个简单的字符驱动

大多数字符设备比较简单，通常直接使用file_operations接口。
我们以一个虚拟的字符设备test_char为例，说明字符驱动的结构。

虚拟文件系统与设备驱动程序的接口

本设备涉及到读写和设备的打开关闭操作。因此可以定义以下file_opertions结构体变量。

```
struct file_operations testchar_fops{
    .read=test_read,
    .write=test_write,
```

```
    .open=test_open,
    .release=test_release
};
```

> *注意：定义结构体变量之前需要先实现用到的函数*

基本函数

```
static ssize_t test_read(struct file *filp, char __user *b
uf, size_t count, loff_t *l)
{   copy_to_user(buf, &val, sizeof(val));
    return sizeof(val);
}//copy_to_user(void __user *to, const void *from, unsigne
d long n);
Static ssize_t test_write(struct file *file, const char __
user *buf, size_t count, loff_t *l)
{   copy_from_user(&val, buf, sizeof(val);
    return sizeof(val);
}//copy_from_user(void *to, void __user *from, unsigned lo
ng n)
Static int test_open(struct inode *inode, struct file *fil
p)
{   try_module_get(THIS_MODULE);//模块计数加一
    return 0;
}
Static int test_release(struct inode *inode, struct file *
filp)
```

```
{   module_put(THIS_MODULE);//模块计数减一
    return 0;
}
```

加载和卸载

```
#define TEST_MAJOR 251
    #define TEST_NAME   “TEST_CHAR”
    int __init init_routine(void)
{   register_chrdev(TEST_MAJOR, TEST_NAME, &testchar_fops)
;
    return0;
}//int register_chrdev(unsigned int major, const char *nam
e, struct file_operations //*fops)
Void cleanup_routine(void)
{   unregister_chrdev(TEST_MAJOR,TEST_NAME);
}
    module_init(init_routine);
    module_exit(cleanup_routine);
```

Linux内核的中断处理

Linux内核对外部设备的中断处理在设备驱动程序里实现。

Linux2.6内核的工作推后执行机制

问题引入：

通常中断处理程序需要尽量做到短小和快速处理，以免中断阻塞时间过长，影响实时性和中断响应速度；
但是，有些设备产生中断后需要处理较多的与中断相关的工作，所以速度会比较慢。

处理方法：
将中断处理分为两个部分：
中断处理程序部分：只做必要的有严格时限的工作，如中断应答和复位硬件等，一般需要屏蔽中断的情况下处理
其他需要时间的不是很紧急的工作推后处理。

内核工作推后执行的机制主要有：

- 软中断
- tasklet
- 工作队列

软中断

通常用于执行频率很高的强实时性的场合。作为一种底层机制，很少由内核程序员直接使用。

tasklet

小任务，指一小段可以执行的代码，通常以函数的形式出现，对于io驱动程序，是实现工作推后执行的首选方法。
tasklet基于软中断机制实现，实际上只是一种软中断的应用和包装。

工作队列

每个工作队列有一个专门的线程，称之为工作队列线程。
工作队列中推后执行的工作由此线程完成，所有的工作在线程的上下文中被执行。因此，允许重新调度甚至睡眠。

```
struct work_struct{
    unsigned long pending;//工作是否等待处理
    struct list_head entry;//工作队列中的工作链表
    void(* func)(void *);//处理函数
    void *data;//处理函数的参数
    void *wq_data;//内部使用
    struct timer_list timer;//定时器
}
```

下面比较三种机制的差别与联系:
软中断：
1、软中断是在编译期间静态分配的。
2、最多可以有32个软中断。
3、软中断不会抢占另外一个软中断，唯一可以抢占软中断的是中断处理程序。 4、可以并发运行在多个CPU上（即使同一类型的也可以）。所以软中断必须设计为可重入的函数（允许多个CPU同时操作），因此也需要使用自旋锁来保护其数据结构。
5、目前只有两个子系直接使用软中断：网络和SCSI。

tasklet：
1、tasklet是使用两类软中断实现的：HI_SOFTIRQ和TASKLET_SOFTIRQ。
2、可以动态增加减少，没有数量限制。
3、同一类tasklet不能并发执行。

4、不同类型可以并发执行。

5、大部分情况使用tasklet。

工作队列：

1、由内核线程去执行，换句话说总在进程上下文执行。

2、可以睡眠，阻塞。