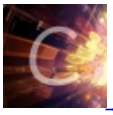
嵌入式Linux misc 设备驱动

原创

木士易 于 2022-01-07 15:23:51 发布 383 收藏 3
分类专栏： 嵌入式 misc 设备驱动 文章标签： linux
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/yangxueyangxue/article/details/122364935
版权


嵌入式 同时被 2 个专栏收录
44 篇文章 3 订阅
订阅专栏


misc 设备驱动
1 篇文章 0 订阅
订阅专栏

misc 设备驱动简介

那么杂项设备驱动是属于我们 linux 三大设备驱动的哪一项呢？ 由于linux 驱动倾向于分层设计， 所以每个具体的设备都可以找到它归属的类型， 从而可以套到它相应的架构里面去， 我们只需要实现它最底层的那部分。 但是也有部分字符设备， 确实不知道它属于哪种类型， 一般推荐大家采用 miscdevice 的框架结构。 misc 的意思是混合的杂项的， 所以 misc 设备驱动也叫做杂项设备驱动， 当板子上的某个设备没有办法分类时， 就可以用 misc 设备驱动。 它的注册跟使用比较的简单，所以比较适用于功能简单的设备。 正因为简单， 所以它通常嵌套在 platform 总线驱动中， 配合总线驱动达到更复杂， 多功能的效果。 杂项设备是字符设备的一种， 杂项设备可以自动生成设备节点。

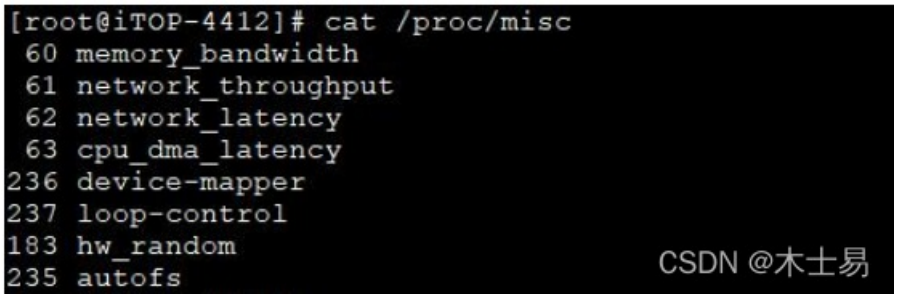
在学习 misc 设备驱动之前， 先来了解几个基础概念。

概念 1 设备节点

我们可以启动我们的开发板， 进入到 dev 目录下， dev 目录下全部都是生成的设备节点， 如下图所示：

我们的系统里面有很多杂项设备。 我们可以输入以下命令来查看， 如下图所示：

cat /proc/misc



概念 2 杂项设备的优点

杂项设备除了比字符设备代码简单， 还有别的区别吗？ 所有的 misc 设备驱动的主设备号都为 10， 不同的设备使用不同的从设备号。 主设设备号相同就可以节省内核的资源， 在内核中大概可以找到 200 多处使用 miscdevice 框架结构的驱动。

概念 3 主设备号和次设备号的概念

设备号包含主设备号和次设备号， 设备号是计算机识别设备的一种方式， 主设备号相同的就被视为同一类设备， 主设备号在 Linux 系统里面是唯一的， 次设备号不一定唯一。 主设备号可以比做成电话号码的区号。 比如北京的区号是 010， 次设备号可以比作成电话号码。

主设备号可以通过以下命令来查看， 前面的数字就是主设备号， 如下图所示：

cat /proc/devices

misc 设备用 miscdevice 结构体表示， miscdevice 结构体的定义在内核源码具体定义在 include/linux/miscdevice.h 中， 内容如下：

```
struct miscdevice {
    int minor; //次设备号
    const char *name; //设备节点的名字
    const struct file_operations *fops; //文件操作集
    struct list_head list;
    struct device *parent;
    struct device *this_device;
    const struct attribute_group **groups;
    const char *nodename;
    umode_t mode;
};
```

当我们创建一个 misc 设备的 miscdevice 结构体时， 需要我们指定 minor、 name 和 fops 这三个成员变量。 minor 表示次设备号， 需要用户设置， 在 Linux 内核中有一些预定义的 misc 设备的次设备号， 定义在 include/linux/miscdevice.h 文件中， 如下所示：

```
#define PSMOUSE_MINOR 1
#define MS_BUSMOUSE_MINOR 2 /* unused */
#define ATIXL_BUSMOUSE_MINOR 3 /* unused */
/*#define AMIGAMOUSE_MINOR 4 FIXME OBSOLETE */
#define ATARIMOUSE_MINOR 5 /* unused */
#define SUN_MOUSE_MINOR 6 /* unused */
......
#define MISC_DYNAMIC_MINOR 255
```

设置子设备号时要注意不要重复使用其他设备的子设备号。 可以从这些预定义的子设备号中选择一个， 也可以自定义。name 就是这个 misc 设备的名字， 当设备注册成功后， 会在/dev 目录下自动生成一个名为 name 的设备文件。 fops 就是这个 misc 设备的操作集合。

当创建好 miscdevice 结构体后， 使用 misc_register 函数向系统中注册一个 misc 设备， 函数原型如下：

函数	int misc_register(struct miscdevice * misc)
参数 misc	之前创建好的 miscdevice 结构体
返回值	成功返回 0， 失败返回负数。

在设备驱动的卸载函数中， 使用 misc_deregister 函数来注销掉 misc 设备。 函数原型如下

函数	int misc_deregister(struct miscdevice *misc)
参数 misc	要注销的 miscdevice 结构体。
返回值	无

在 miscdevice 结构体的第四行， 它指向了一个 file_operation 的结构体。 file_operations 文件操作集在定义在 include/linux/fs.h 下面， 如下图所示。

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*iterate) (struct file *, struct dir_context *);
rminal  int (*iterate_shared) (struct file *, struct dir_context *);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **, void **);
        long (*fallocate)(struct file *file, int mode, loff_t offset,
                          loff_t len);
        void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
        unsigned (*mmap_capabilities)(struct file *);
#endif
        ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
                        loff_t, size_t, unsigned int);
        int (*clone_file_range)(struct file *, loff_t, struct file *, loff_t,
                        u64);
        ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *,
                        u64);
} __randomize_layout;
```

CSDN @木士易

file_operations 中的成员函数实际是由 drivers/char/misc.c 中 misc 驱动核心层的 misc_fops 成员函数间接调用的。 file_operations 结构体里面的结构体成员都对应一个调用。 简单介绍一下其中比较常用的函数：
llseek()函数用来修改一个文件的当前的读写位置， 并将新位置返回。
read()函数用来从设备中读取数据， 成功时返回读取到的字节数， 出错返回一个负值。
write()函数用来向设备发送数据， 成功时返回该函数写入的字节数。
poll()函数用于查询设备是否可以进行非阻塞读写。
unlock_ioctl()函数提供设备相关控制命令的实现。
mmap()函数将设备内存映射到进程的虚拟地址空间中。
open()函数用于打开设备文件。
release()函数用于关闭设备文件。
注册杂项设备有一个通用的思路和方法， 这里给大家总结为三个步骤：
填充 miscdevice 这个结构体
填充 file_operations 这个结构体
注册杂项设备并生生成设备节点。

实验程序

添加头文件

```
/*注册杂项设备头文件*/
#include <linux/miscdevice.h>
/*注册设备节点的文件结构体*/
#include <linux/fs.h>
```

填充 miscdevice 结构体

```
struct miscdevice misc_dev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "hello_misc",
    .fops = &misc_fops,
};
```

上述代码第 2 行的 minor 为 MISC_DYNAMIC_MINOR， miscdevice 核心层会自动找一个空闲的次设备号，否则用 minor 指定的次设备号。 上述代码第 3 行 name 是设备的名称， 自定义为"hello_misc"

填充 file_operations 结构体

```
struct file_operations misc_fops={
    .owner = THIS_MODULE
};
```

THIS_MODULE 宏是什么意思呢？ 它在 include/linux/module.h 里的定义是
#define THIS_MODULE (&__this_module)
它是一个 struct module 变量， 代表当前模块， 可以通过 THIS_MODULE 宏来引用模块的 struct module 结构， 比如使用 THIS_MODULE->state 可以获得当前模块的状态。 这个 owner 指针指向的就是你的模块。

注册杂项设备并生成设备节点

在 misc_init()函数中填充 misc_register()函数注册杂项设备， 并判断杂项设备是否注册成功。

```
static int misc_init(void){
    int ret;
    ret = misc_register(&misc_dev); //注册杂项设备
    if(ret<0) //判断杂项设备是否注册成功
    {
       printk("misc registe is error \n"); //打印杂项设备注册失败
    }
    printk("misc registe is succeed \n"); //打印杂项设备注册成功
    return 0;
}

在 misc_exit（） 函数中填充 misc_deregister()函数注销杂项设备。

static void misc_exit(void){
    misc_deregister(&misc_dev); //注销杂项设备
    printk("misc goooodbye! \n"); //打印杂项设备注销成功
}
```

完整的代码如下图所示：

```
/*
* @Descripttion: 最简单的杂项设备驱动
* @version:
* @Author: topeet
*/
#include <linux/init.h> //初始化头文件
#include <linux/module.h> //最基本的文件， 支持动态添加和卸载模块。
#include <linux/miscdevice.h> /*注册杂项设备头文件*/
#include <linux/fs.h> /*注册设备节点的文件结构体*/

struct file_operations misc_fops=
{ //文件操作集
    .owner = THIS_MODULE
};
struct miscdevice misc_dev =
{ //杂项设备结构体
    .minor = MISC_DYNAMIC_MINOR, //动态申请的次设备号
    .name = "hello_misc", //杂项设备名字是 hello_misc
    .fops = &misc_fops, //文件操作集
};

static int misc_init(void)
{ //在初始化函数中注册杂项设备
    int ret;
    ret = misc_register(&misc_dev);
    if(ret<0)
    {
        printk("misc registe is error \n");
    }
    printk("misc registe is succeed \n");
    return 0;
}

static void misc_exit(void)
 { //在卸载函数中注销杂项设备
    misc_deregister(&misc_dev);
    printk(" misc goooodbye! \n");
}

module_init(misc_init);
module_exit(misc_exit);
MODULE_LICENSE("GPL");
```

现在最简单的杂项设备的驱动就写完了， 那么接下来我们可以把这个驱动编译一下， 然后放到开发板上面运行。 编译驱动， 可以将它编译进内核里面， 也可以将它编译成模块。

编译驱动程序

Makefile 为：

```
obj-m += misc.o #先写生成的中间文件的名字是什么， -m 的意思是把我们的驱动编译成模块
KDIR:=/home/topeet/driver/imx6ull/linux-imx-rel_imx_4.1.15_2.1.0_ga/
PWD?=$(shell pwd) #获取当前目录的变量
all:
    make -C $(KDIR) M=$(PWD) modules #make 会进入内核源码的路径， 然后把当前路径下的代码编译成
模块
```

驱动编译成功生成了 ko 文件， 如下图所示：

运行测试

进入到共享目录， 加载驱动模块如图所示：

cd imx6ull/

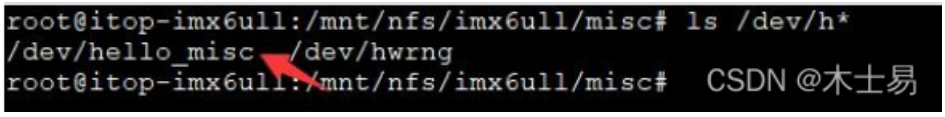
ls

cd misc/

insmod misc.ko

驱动加载成功后， 输入以下命令， 查看注册的设备节点是否存在， 如下图所示， 设备节点存在。

ls /dev/h*



输入以下命令拆卸驱动模块,如下图所示：

rmmod misc