**构建Linux内核驱动demo子系统示例**



[luckyapple1028](https://me.csdn.net/luckyapple1028) 2016-02-09 15:51:25

<https://blog.csdn.net/luckyapple1028/article/details/50642906>

分类专栏： [Linux Kernel](https://blog.csdn.net/luckyapple1028/category_2927315.html) 文章标签： [linux](https://www.csdn.net/gather_2d/MtjaQg5sMDY0MC1ibG9n.html) [kernel](https://www.csdn.net/gather_2c/MtjaIg5sNTE1MjQtYmxvZwO0O0OO0O0O.html) [嵌入式](https://www.csdn.net/gather_20/MtjaMg0sNzg0NjUtYmxvZwO0O0OO0O0O.html) [框架](https://www.csdn.net/gather_23/MtTaEg0sMTc0OTQtYmxvZwO0O0OO0O0O.html)

版权

一般在编写嵌入式Linux内核驱动时，最简单的情况下往往只需要写一个简单的misc驱动，它仅需要兼容一种硬件外设和一种特定的芯片平台即可，这种驱动的最大缺点就是可扩展性和可移植性较差，往往在单板硬件上存在小幅的改动就需要更改驱动源代码，有时在甚至在硬件上增加了一个相同的外设时需要重新为其写一个几乎一模一样的驱动。

一个好的Linux内核驱动是要求在尽量小的改动下能够快速适配于不同的平台，且能够支持多设备。Linux内核针对没有挂接在物理总线（PCI、I2C和USB等等）上的嵌入式设备设计了一套platform总线驱动框架，这种框架能够很好的解决满足一般嵌入式外设驱动的要求。但如果存在较多功能类似但又不尽相同的的外设就需要为其设计对应的子系统来统一管理，Linux内核里存在许多各式各样的子系统，有相当复杂的也有比较简单的，它们不仅统一管理了不同类型的外设驱动也屏蔽了他们的差异并向用户空间提供了统一的接口。例如统一管理RTC驱动的RTC子系统、统一管理图形显示设备的framebuffer子系统，甚至非常庞大且复杂的网络子系统和文件系统子系统等等。

今次本文参考内核RTC子系统并提取出一个简单的demo驱动子系统框架示例程序，可适用于一些简单的Linux设备驱动开发。

示例环境：

交叉编译工具链：arm-bcm2708-linux-gnueabi-

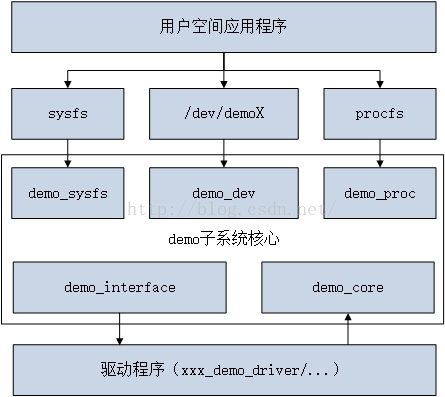
Linux内核：linux-rpi-4.1.y

单板硬件：树莓派b

源码链接：https://github.com/luckyapple1028/linux-demo-subsys-module

**一、demo子系统框架**

**框架结构图：**



示例程序列表：demo\_core.c、demo\_dev.c、demo\_interface.c、demo\_proc.c、demo\_sysfs.c、xxx\_demo\_driver.c、xxx\_demo\_device.c、demo-core.h、demo\_dev.h、demo.h

1、demo\_core

demo驱动程序的管理的核心，向设备驱动程序提供接口，完成驱动程序向内核的的注册和卸载。

2、demo\_dev

负责demo驱动的字符设备文件的管理，包括注册和注销字符设备，定义了包括read、write、ioctl等一系列设备控制接口。

3、demo\_interface.c

提供了demo设备iotcl控制的函数接口，负责对下层实际的驱动接口进行统一管理和调用。

4、demo\_proc.c

提供了proc文件系统的demo设备查询和控制接口。

5、demo\_sysfs.c

提供了sys文件系统的demo设备查询和控制接口，依赖于标准的Linux设备驱动模型。

6、xxx\_demo\_driver.c

具体外设的驱动程序，不同的外设针对自己的特点可分别实现，是真正具有差异性的部分，然后统一向demo子系统注册。本文中作为示例驱动依赖于platform驱动模型。

7、xxx\_demo\_device.c

具体外设，同本文中设备驱动xxx\_demo\_driver匹配，负责注册platform设备并向驱动程序传入具体的物理外设参数。（注：本文这里使用的是传统的方法，在新的Linux中可用Device Tree代替）。

**二、结构体**

**1、xxx\_demo**

struct xxx\_demo {

struct demo\_device \*demo; /\* demo子系统通用设备指针 \*/

struct timer\_list xxx\_demo\_timer;

unsigned long xxx\_demo\_data;

/\* something else \*/

};

xxx\_demo结构是具体的demo驱动的控制结构，不同的外设驱动不尽相同，可以包括自己需要的一些数据和特殊结构。例如本示例结构中包含了一个内核定时器结构和一个data数据。除了具体的驱动数据之外，最重要的是demo\_device指针，它是demo子系统的核心，负责了穿针引线的作用，在本驱动向demo子系统完成注册之后就会生成具体的对象了。下面来具体分析这个结构。

**2、demo\_device**

struct demo\_device

{

struct device dev;

struct module \*owner;

int id;

char name[DEMO\_DEVICE\_NAME\_SIZE];

const struct demo\_class\_ops \*ops;

struct mutex ops\_lock;

struct cdev char\_dev;

unsigned long flags;

unsigned long irq\_data;

spinlock\_t irq\_lock;

wait\_queue\_head\_t irq\_queue;

struct fasync\_struct \*async\_queue;

/\* some demo data \*/

struct demo\_data demo\_data;

};

该结构体中struct device dev 结构体为Linux设备驱动模型的设备结构，在向内核注册设备时需要进行初始化；struct module \*owner指针一般设置为THIS\_MODULE即可；id为demo设备号，由于子系统可支持多个demo驱动设备，顾需要进行编号便于管理；name为设备驱动的名字；struct demo\_class\_ops \*ops为demo驱动的控制函数集，为驱动程序向子系统注册的控制函数接口，在驱动进行注册时会进行赋值，以后由子系统负责调用具体的驱动功能接口；struce cdev char\_dev即是demo设备的字符设备结构体了；其他struct mutex ops\_lock和spinlock\_t irq\_lock锁分别用于保护iotcl操作和保护中断操作到的临界区数据，irq\_queue为等待队列，用于实现阻塞式的io操作，async\_queue用于信号式异步io操作，最后的demo\_data为demo子系统所管理的示例数据结构（该以上几项可更具驱动子系统的具体需要自行删减）。

**3、demo\_data**

struct demo\_data

{

unsigned long text\_data;

/\* something else \*/

};

该结构包含在demo\_device结构体中，包含了一些demo驱动中通用性的参数以及demo子系统为了屏蔽底层差异并向用户提供统一接口时需要而提取出来的需要统一管理的参数等等，这里仅示例性的列了一个demo\_data数据。

**4、demo\_class\_ops**

struct demo\_class\_ops {

int (\*open)(struct device \*);

void (\*release)(struct device \*);

int (\*ioctl)(struct device \*, unsigned int, unsigned long);

int (\*set\_data)(struct device \*, struct demo\_ctl\_data \*);

int (\*get\_data)(struct device \*, struct demo\_ctl\_data \*);

int (\*proc)(struct device \*, struct seq\_file \*);

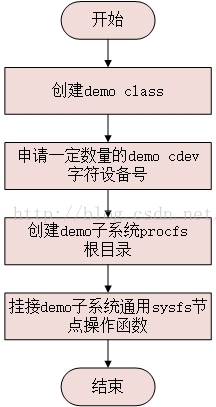
int (\*read\_callback)(struct device \*, int data);

};

该结构为demo驱动程序的注册函数接口，由具体的xxx\_demo驱动负责实现并向demo子系统注册，但并不是每个实例都需要实现，仅需要实现驱动所支持的功能即可，在demo子系统中会根据用户的需要进行调用。这里的open、release和ioctl接口都已经非常熟悉了，下面的set\_data和get\_data仅作为示例使用，分别用来设置和获取驱动中的具体数据，proc接口用于proc文件系统调用，后面会具体看到用法。

**三、demo子系统和驱动程序流程分析**

**1、demo子系统初始化**



这里demo子系统的初始化符合大多数Linux外设驱动子系统初始化方案，来简单走读一下代码：

/\* demo子系统初始化 \*/

static int \_\_init demo\_core\_init(void)

{

/\* 创建 demo class \*/

demo\_class = class\_create(THIS\_MODULE, "demo");

if (IS\_ERR(demo\_class)) {

pr\_err("couldn't create class\n");

return PTR\_ERR(demo\_class);

}

/\* demo 设备驱动初始化 \*/

demo\_dev\_init();

/\* demo proc初始化 \*/

demo\_proc\_init();

/\* demo sysfs初始化 \*/

demo\_sysfs\_init(demo\_class);

pr\_info("demo subsys init success\n");

return 0;

}

首先创建了一个class，具体作用详见Linux设备驱动模型，主要的用于在后续自动创建设备文件和在sysfs目录下创建控制节点。然后调用的demo\_dev\_init()函数：

void \_\_init demo\_dev\_init(void)

{

int err;

err = alloc\_chrdev\_region(&demo\_devt, 0, DEMO\_DEV\_MAX, "demo");

if (err < 0)

pr\_err("failed to allocate char dev region\n");

}

该函数向内核申请了主设备号和最大DEMO\_DEV\_MAX个次设备号，也即最多可以支持DEMO\_DEV\_MAX个demo字符设备，可以根据需要动态调整（占8位，上限255）。接着初始化函数调用demo\_proc\_init()函数：

void \_\_init demo\_proc\_init(void)

{

/\* 创建 demo proc 目录 \*/

demo\_proc = proc\_mkdir("driver/demo", NULL);

}

这个函数的作用非常简单就是在文件系统的/proc/driver/目录下创建demo子目录，后续驱动程序的proc节点都将保存在这个目录下。最后初始化函数调用demo\_sysfs\_init()函数：

void \_\_init demo\_sysfs\_init(struct class \*demo\_class)

{

/\* 绑定通用sys节点，在注册设备时会依次生成 \*/

demo\_class->dev\_groups = demo\_groups;

}

该函数绑定了sysfs节点操作函数的集合，它不会立即在/sys目录下生成节点，会在后面驱动程序注册设备时生成。这里的demo\_group是一个attribute函数指针数组，如下：

static struct attribute \*demo\_attrs[] = {

&dev\_attr\_demo\_name.attr,

&dev\_attr\_demo\_data.attr,

NULL,

};

ATTRIBUTE\_GROUPS(demo);

其中的dev\_attr\_demo\_name和dev\_attr\_demo\_data由宏DEVICE\_ATTR\_RO和DEVICE\_ATTR\_RW生成，他们分别定义了只读的和可读可写的attribute节点并绑定了对应的函数。其中个中细节详见《 [Linux设备驱动模块自加载示例与原理解析](http://blog.csdn.net/luckyapple1028/article/details/44261391)》。

这里的demo子系统初始化流程只是一个简单的框架模型，对于具体的设备子系统还会进行一些其他部件的初始化。分析完初始化流程后来简单看一下反初始化流程：

static void \_\_exit demo\_core\_exit(void)

{

demo\_proc\_exit();

demo\_dev\_exit();

class\_destroy(demo\_class);

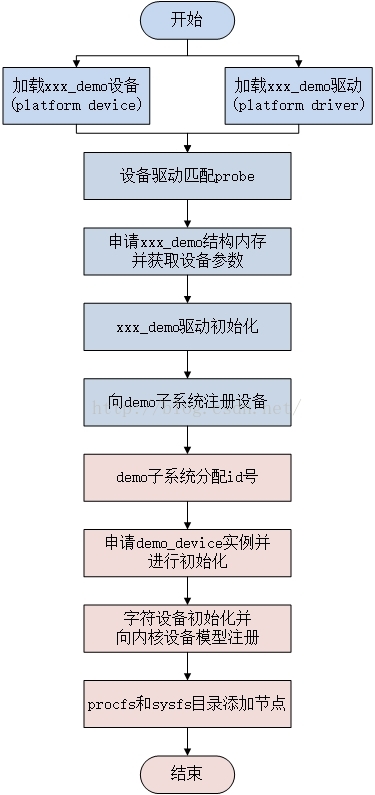
ida\_destroy(&demo\_ida);

pr\_info("demo subsys exit success\n");

}

反初始化流程可以视为初始化流程的一个逆过程，非常直白。下面来分析一个具体的驱动程序是如何完成初始化并注册到demo子系统中的。

**2、驱动注册总体流程**



具体驱动的初始化流程视具体的物理特性可以千变万化，对于依赖于I2C通信的外设可以通过I2C总线完成初始化、对于依赖于SPI通信的外设则可以通过SPI总线完成初始化，本文为了简单起见使用了虚拟的platform总线来进行驱动的匹配和注册操作，来详细分析一下代码：

xxx\_demo\_driver：

static struct platform\_driver xxx\_demo\_driver = {

.driver = {

.name = "xxx\_demo\_device",

.owner = THIS\_MODULE,

},

.probe = xxx\_demo\_driver\_probe,

.remove = xxx\_demo\_driver\_remove,

};

demo0\_device和demo1\_device：

static struct platform\_device demo0\_device = {

.name = "xxx\_demo\_device",

.id = 0,

.dev = {

.release = demo\_device\_release,

}

};

static struct platform\_device demo1\_device = {

.name = "xxx\_demo\_device",

.id = 1,

.dev = {

.release = demo\_device\_release,

}

};

xxx\_demo\_driver定义了probe函数xxx\_demo\_driver\_probe，它会在platform设备和platform驱动匹配时被platform\_drv\_probe()函数调用；platform驱动由驱动模块初始化函数xxx\_demo\_driver\_init()调用platform\_driver\_register(&xxx\_demo\_driver)函数注册；platform设备由驱动设备模块初始化函数demo\_device\_init()调用platform\_device\_register(&demoX\_device)函数注册，这里注册了两个xxx\_demo设备用来示例模拟多设备注册。值得说明的是，这里注册platform设备的方法是使用较为传统的方法，较为妥当的方式是使用Linux device tree来完成设备的注册（后续会补上）。下面来分析probe()函数：

static int xxx\_demo\_driver\_probe(struct platform\_device \*pdev)

{

struct xxx\_demo \*xxx\_demo = NULL;

int ret = 0;

/\* 申请驱动结构内存并保存为platform的私有数据 \*/

xxx\_demo = devm\_kzalloc(&pdev->dev, sizeof(struct xxx\_demo), GFP\_KERNEL);

if (!xxx\_demo)

return -ENOMEM;

platform\_set\_drvdata(pdev, xxx\_demo);

/\* 获取平台资源 \*/

/\* do something \*/

/\* 执行驱动相关初始化(包括外设硬件、锁、队列等)\*/

xxx\_demo->xxx\_demo\_data = 0;

/\* do something \*/

init\_timer(&xxx\_demo->xxx\_demo\_timer);

xxx\_demo->xxx\_demo\_timer.function = xxx\_demo\_time;

xxx\_demo->xxx\_demo\_timer.data = (unsigned long)xxx\_demo;

xxx\_demo->xxx\_demo\_timer.expires = jiffies + HZ;

add\_timer(&xxx\_demo->xxx\_demo\_timer);

/\* 向 demo 子系统注册设备 \*/

xxx\_demo->demo = devm\_demo\_device\_register(&pdev->dev, "xxx\_demo",

&xxx\_demo\_ops, THIS\_MODULE);

if (IS\_ERR(xxx\_demo->demo)) {

dev\_err(&pdev->dev, "unable to register the demo class device\n");

ret = PTR\_ERR(xxx\_demo->demo);

goto err;

}

return 0;

err:

del\_timer\_sync(&xxx\_demo->xxx\_demo\_timer);

return ret;

}

该函数首先向内核申请了xxx\_demo的结构实例内存空间，然后并将该结构体设置为了pedv的私有数据（注意这一步很关键，后面用于通过pdev查找对应的xxx\_demo结构实例）；然后可以获取一些platform resource及一些物理外设参数和资源（本示例程序中没有详细写出）；接下来就可以开始执行驱动躯体的初始化了，可以设置一些驱动外设的物理寄存器或者如同这里初始化可一个内核定时器，接下来最为关键的就是调用devm\_demo\_device\_register函数完成向demo子系统的注册工作。

/\* demo设备注册澹(使用devm机制) \*/

struct demo\_device \*devm\_demo\_device\_register(struct device \*dev,

const char \*name,

const struct demo\_class\_ops \*ops,

struct module \*owner)

{

struct demo\_device \*\*ptr, \*demo;

ptr = devres\_alloc(devm\_demo\_device\_release, sizeof(\*ptr), GFP\_KERNEL);

if (!ptr)

return ERR\_PTR(-ENOMEM);

/\* 注册 demo 设备 \*/

demo = demo\_device\_register(name, dev, ops, owner);

if (!IS\_ERR(demo)) {

\*ptr = demo;

devres\_add(dev, ptr);

} else {

devres\_free(ptr);

}

return demo;

}

该函数利用了内核的devm机制，它是一种错误回收机制，在初始化某步出错的时候不需要驱动程序员再逐次调用反向去初始化，下面把注意点放到最关键的demo\_device\_register()函数：

/\* demo设备注册 \*/

struct demo\_device \*demo\_device\_register(const char \*name, struct device \*dev,

const struct demo\_class\_ops \*ops,

struct module \*owner)

{

struct demo\_device \*demo;

int of\_id = -1, id = -1, err;

/\* 获取ID号 \*/

if (dev->of\_node)

of\_id = of\_alias\_get\_id(dev->of\_node, "demo");

else if (dev->parent && dev->parent->of\_node)

of\_id = of\_alias\_get\_id(dev->parent->of\_node, "demo");

if (of\_id >= 0) {

id = ida\_simple\_get(&demo\_ida, of\_id, of\_id + 1, GFP\_KERNEL);

if (id < 0)

dev\_warn(dev, "/aliases ID %d not available\n", of\_id);

}

if (id < 0) {

id = ida\_simple\_get(&demo\_ida, 0, 0, GFP\_KERNEL);

if (id < 0) {

err = id;

goto exit;

}

}

/\* 开始分配内存 \*/

demo = kzalloc(sizeof(struct demo\_device), GFP\_KERNEL);

if (demo == NULL) {

err = -ENOMEM;

goto exit\_ida;

}

/\* demo 结构初始化 \*/

demo->id = id;

demo->ops = ops;

demo->owner = owner;

demo->dev.parent = dev;

demo->dev.class = demo\_class;

demo->dev.release = demo\_device\_release;

mutex\_init(&demo->ops\_lock);

spin\_lock\_init(&demo->irq\_lock);

init\_waitqueue\_head(&demo->irq\_queue);

strlcpy(demo->name, name, DEMO\_DEVICE\_NAME\_SIZE);

dev\_set\_name(&demo->dev, "demo%d", id);

/\* 字符设备初始化 \*/

demo\_dev\_prepare(demo);

err = device\_register(&demo->dev);

if (err) {

put\_device(&demo->dev);

goto exit\_kfree;

}

/\* 字符设备、sysfs设备和proc设备注册添加 \*/

demo\_dev\_add\_device(demo);

demo\_sysfs\_add\_device(demo);

demo\_proc\_add\_device(demo);

dev\_notice(dev, "demo core: registered %s as %s\n", demo->name, dev\_name(&demo->dev));

return demo;

exit\_kfree:

kfree(demo);

exit\_ida:

ida\_simple\_remove(&demo\_ida, id);

exit:

dev\_err(dev, "demo core: unable to register %s, err = %d\n", name, err);

return ERR\_PTR(err);

}

该函数首先向dev（此处为pedv->dev）和dev->parent的of节点查找获取of\_id号，然后以该of\_id为基数获取一个新的id号，这里由于并不存在of\_node所以会直接从demo\_ida中获取空闲的ida，前文中注册的两个demo驱动设备，这里会分别分配0和1。这里的demo\_ida由宏定义：static DEFINE\_IDA(demo\_ida);

注册函数接下来向内核申请demo\_device结构的内存并进行赋值，分别赋值了id号、demo驱动操作函数指针ops，初始化了demo->dev结构，指定了parent为pedv->dev、class为demo\_class、并指定了结构释放回调函数，然后初始化了锁和等待队列。注意这里的函数指针ops指向了xxx\_demo\_driver中实现的xxx\_demo\_ops：

struct demo\_class\_ops xxx\_demo\_ops = {

.open = xxx\_demo\_open,

.release = xxx\_demo\_release,

.ioctl = xxx\_demo\_ioctl,

.set\_data = xxx\_demo\_set\_data,

.get\_data = xxx\_demo\_get\_data,

.proc = xxx\_demo\_proc,

.read\_callback = xxx\_demo\_read,

};

然后对设备名进行赋值，demo->name为“xxx\_demo”，demo->dev.name为“demoX”，后一个名字会作为/dev/目录设备文件、procfs节点文件及sysfs节点目录的名字。同样的可以针对不同的需求进行其他不同组件的初始化。接下来调用demo\_dev\_prepare函数根据分配的id号和字符设备号初始化字符设备结构：

void demo\_dev\_prepare(struct demo\_device \*demo)

{

if (!demo\_devt)

return;

if (demo->id >= DEMO\_DEV\_MAX) {

dev\_warn(&demo->dev, "%s: too many demo devices\n", demo->name);

return;

}

/\* 字符设备结构初始化 \*/

demo->dev.devt = MKDEV(MAJOR(demo\_devt), demo->id);

cdev\_init(&demo->char\_dev, &demo\_dev\_fops);

demo->char\_dev.owner = demo->owner;

}

可以看到字符设备号由子系统初始化时分配的主设备号和id号作为次设备好组成，然后绑定fops函数结构demo\_dev\_fops：

static const struct file\_operations demo\_dev\_fops = {

.owner = THIS\_MODULE,

.llseek = no\_llseek,

.read = demo\_dev\_read,

.poll = demo\_dev\_poll,

.unlocked\_ioctl = demo\_dev\_ioctl,

.open = demo\_dev\_open,

.release = demo\_dev\_release,

.fasync = demo\_dev\_fasync,

};

初始化完字符设备结构后接下来调用device\_register()函数向Linux内核注册设备，注册完成后就会在/sys目录下生成对应的目录并生成前文中绑定的attr属性文件demo\_data和 demo\_name。接着调用demo\_dev\_add\_device()函数向内核添加字符设备，添加完成后会在/dev目录下生成对应的/demoX设备节点。

void demo\_dev\_add\_device(struct demo\_device \*demo)

{

/\* 注册字符设备 \*/

if (cdev\_add(&demo->char\_dev, demo->dev.devt, 1))

dev\_warn(&demo->dev, "%s: failed to add char device %d:%d\n",

demo->name, MAJOR(demo\_devt), demo->id);

else

dev\_dbg(&demo->dev, "%s: dev (%d:%d)\n", demo->name,

MAJOR(demo\_devt), demo->id);

}

注册函数接着调用demo\_sysfs\_add\_device函数用来创建一些特殊性的attr节点。

void demo\_sysfs\_add\_device(struct demo\_device \*demo)

{

int err;

/\* 条件判断 \*/

/\* do something \*/

/\* 为需要的设备创建一些特殊的 sys 节点 \*/

err = device\_create\_file(&demo->dev, &dev\_attr\_demodata);

if (err)

dev\_err(demo->dev.parent,

"failed to create alarm attribute, %d\n", err);

}

该函数可以在进入后执行一些条件判断，用来判别是否需要创建attr属性文件。本示例程序中创建了一个dev\_attr\_demodata属性文件并绑定show和set函数为demo\_sysfs\_show\_demodata()和demo\_sysfs\_set\_demodata()。注册函数最后调用demo\_proc\_add\_device()在/proc/driver/demo目录下创建名为“demoX”（dev\_name(&demo->dev)）的文件并绑定文件操作函数：

void demo\_proc\_add\_device(struct demo\_device \*demo)

{

/\* 为新注册的设备分配 proc \*/

proc\_create\_data(dev\_name(&demo->dev), 0, demo\_proc, &demo\_proc\_fops, demo);

}

static const struct file\_operations demo\_proc\_fops = {

.open = demo\_proc\_open,

.read = seq\_read,

.llseek = seq\_lseek,

.release = demo\_proc\_release,

};

分析完xxx\_demo驱动程序的初始化流程，来简单看一下去初始化流程，去初始化流程在xxx\_demo驱动模块的remove接口中执行：

static int xxx\_demo\_driver\_remove(struct platform\_device \*pdev)

{

struct xxx\_demo \*xxx\_demo = platform\_get\_drvdata(pdev);

/\* 执行驱动相关去初始化(包括外设硬件、锁、队列等)\*/

del\_timer\_sync(&xxx\_demo->xxx\_demo\_timer);

/\* do something \*/

/\* 向 demo 子系统注销设备 \*/

devm\_demo\_device\_unregister(&pdev->dev, xxx\_demo->demo);

/\* 释放驱动结构内存 \*/

devm\_kfree(&pdev->dev, xxx\_demo);

return 0;

}

该函数首先执行具体驱动程序组件的去初始化和物理硬件功能的关闭（本示例程序销毁前文中初始化的内核定时器），然后调用devm\_demo\_device\_unregister()->devres\_release()函数向demo子系统执行注销流程，最后devm\_kfree释放驱动结构实例。

void devm\_demo\_device\_unregister(struct device \*dev, struct demo\_device \*demo)

{

int res;

/\* 注销 demo 设备 \*/

res = devres\_release(dev, devm\_demo\_device\_release,

devm\_demo\_device\_match, demo);

WARN\_ON(res);

}

该函数调用devm\_demo\_device\_match函数找到需要的demo\_device结构，然后调用devm\_demo\_device\_release执行demo设备注销

static void devm\_demo\_device\_release(struct device \*dev, void \*res)

{

struct demo\_device \*demo = \*(struct demo\_device \*\*)res;

demo\_device\_unregister(demo);

}

void demo\_device\_unregister(struct demo\_device \*demo)

{

if (get\_device(&demo->dev) != NULL) {

mutex\_lock(&demo->ops\_lock);

demo\_sysfs\_del\_device(demo);

demo\_dev\_del\_device(demo);

demo\_proc\_del\_device(demo);

device\_unregister(&demo->dev);

demo->ops = NULL;

mutex\_unlock(&demo->ops\_lock);

put\_device(&demo->dev);

}

}

这里的去初始函数同样非常的直观，首先获取设备（增加设备的引用计数），然后上锁并释放proc和sysfs属性文件，然后调用device\_unregister注销设备，最后put\_device释放设备引用计数，在设备引用计数降到0后会调用前面初始化时注册的release函数demo\_device\_release()：

static void demo\_device\_release(struct device \*dev)

{

struct demo\_device \*demo = to\_demo\_device(dev);

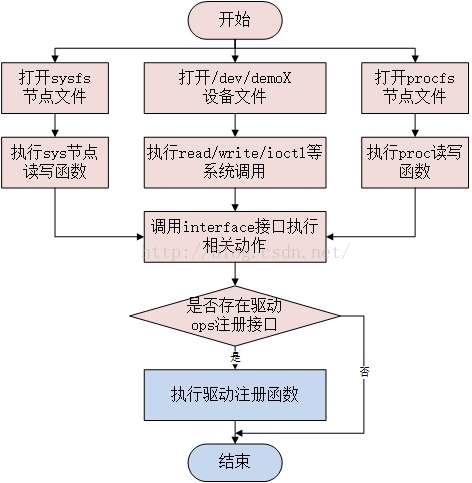
ida\_simple\_remove(&demo\_ida, demo->id);

kfree(demo);

}

这里回收了id号并释放demo\_device结构。驱动程序注册成功了后，下面来分析应用层是如何调用该驱动中的机制的。

**3、应用调用总体流程**



应用层可以通过设备文件/dev/demoX设备文件、procfs和sys属性文件同内核demo子系统进行交互。首先来看前面注册的demo\_dev\_fops函数集合中的open函数

static int demo\_dev\_open(struct inode \*inode, struct file \*file)

{

struct demo\_device \*demo = container\_of(inode->i\_cdev, struct demo\_device, char\_dev);

const struct demo\_class\_ops \*ops = demo->ops;

int err;

if (test\_and\_set\_bit\_lock(DEMO\_DEV\_BUSY, &demo->flags))

return -EBUSY;

file->private\_data = demo;

/\* 调用驱动层 open 实现 \*/

err = ops->open ? ops->open(demo->dev.parent) : 0;

if (err == 0) {

spin\_lock\_irq(&demo->irq\_lock);

/\* do something while open \*/

demo->irq\_data = 0;

spin\_unlock\_irq(&demo->irq\_lock);

return 0;

}

clear\_bit\_unlock(DEMO\_DEV\_BUSY, &demo->flags);

return err;

}

该函数首先找到对应的demo\_device结构实例，然后通过该结构就可以找到驱动程序注册的ops函数集合并尝试调用，由于本文中xxx\_demo\_driver驱动程序并没有实现该函数接口，因此不会调用，对于具体的驱动程序可以视情况进行实现。

另外值得注意的是，这里是如何找到对应的demo\_device结构实例的？若驱动注册了多个结构实例会有何影响呢？

关键就在本函数的第一条语句中，在inode的i\_cdev指针中保存了用户想要打开设备文件的cdev结构地址，然而该结构又包含在demo\_device结构中，通过它即可找到对应的demo\_device实例了。同时由于Linux进程在每打开一个文件后都会创建一个file结构，这里将找到的demo\_device实例绑定为该file结构的私有数据，以后用户空间对该设备节点的其他系统调用操作都将能够准确的找到该demo\_device结构，不会出现错误。例如，前文中注册了两个demo设备，在/dev目录下就会生成两个设备文件demo0和demo1。当用户程序open demo0，就会调用到这里的open函数并通过保存在该demo0文件中的设备号找到对应的cdev结构，最后通过该cedv结构找到对应的demo\_device结构，而不会对demo1存在影响。用户进程在open了设备后就可以调用read、ioctl等系统调用了，来分别看一下。

static ssize\_t demo\_dev\_read(struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*ppos)

{

struct demo\_device \*demo = file->private\_data;

DECLARE\_WAITQUEUE(wait, current);

unsigned long data;

ssize\_t ret;

/\* 对读取数据量进行保护 \*/

if (count != sizeof(unsigned int) && count < sizeof(unsigned long))

return -EINVAL;

/\* 等待数据就绪 \*/

add\_wait\_queue(&demo->irq\_queue, &wait);

do {

\_\_set\_current\_state(TASK\_INTERRUPTIBLE);

spin\_lock\_irq(&demo->irq\_lock);

data = demo->irq\_data;

demo->irq\_data = 0;

spin\_unlock\_irq(&demo->irq\_lock);

if (data != 0) {

ret = 0;

break;

}

if (file->f\_flags & O\_NONBLOCK) {

ret = -EAGAIN;

break;

}

if (signal\_pending(current)) {

ret = -ERESTARTSYS;

break;

}

schedule();

} while (1);

set\_current\_state(TASK\_RUNNING);

remove\_wait\_queue(&demo->irq\_queue, &wait);

/\* 从domo驱动层中读取数据并传输到应用层 \*/

if (ret == 0) {

if (demo->ops->read\_callback)

data = demo->ops->read\_callback(demo->dev.parent,

data);

if (sizeof(int) != sizeof(long) &&

count == sizeof(unsigned int))

ret = put\_user(data, (unsigned int \_\_user \*)buf) ?:

sizeof(unsigned int);

else

ret = put\_user(data, (unsigned long \_\_user \*)buf) ?:

sizeof(unsigned long);

}

return ret;

}

该read函数同时实现了同步非阻塞和同步阻塞式的接口。如果用户配置了O\_NONBLOCK，在数据没有ready的情况下就会直接返回失败，否则将使进程睡眠知道数据就绪。当数据就绪后就尝试调用驱动的read\_callback函数来获取数据，最后把数据传回给应用层。

static unsigned int demo\_dev\_poll(struct file \*file, poll\_table \*wait)

{

struct demo\_device \*demo = file->private\_data;

unsigned long data;

/\* 加入等待队列 \*/

poll\_wait(file, &demo->irq\_queue, wait);

/\* 读取数据并判断条件是否满足(若不满足本调用进程会睡眠) \*/

data = demo->irq\_data;

return (data != 0) ? (POLLIN | POLLRDNORM) : 0;

}

该接口实现了异步阻塞式接口，当用户看空间调用了poll、select或epoll接口就会在此处判断数据是否ready，若ready则以上系统调用直接返回，否则就会睡眠在此处准备的等待队列中（并非在此处睡眠）。

static long demo\_dev\_ioctl(struct file \*file,

unsigned int cmd, unsigned long arg)

{

struct demo\_device \*demo = file->private\_data;

struct demo\_ctl\_data demo\_ctl;

const struct demo\_class\_ops \*ops = demo->ops;

void \_\_user \*uarg = (void \_\_user \*) arg;

int err = 0;

err = mutex\_lock\_interruptible(&demo->ops\_lock);

if (err)

return err;

switch (cmd) {

case DEMO\_IOCTL\_SET:

/\* 进程权限限制(可选), 详见capability.h \*/

if (!capable(CAP\_SYS\_RESOURCE)) {

err = -EACCES;

goto done;

}

mutex\_unlock(&demo->ops\_lock);

if (copy\_from\_user(&demo\_ctl, uarg, sizeof(demo\_ctl)))

return -EFAULT;

/\* demo 示例设置命令函数 \*/

return demo\_test\_set(demo, &demo\_ctl);

case DEMO\_IOCTL\_GET:

mutex\_unlock(&demo->ops\_lock);

/\* demo 示例获取命令函数 \*/

err = demo\_test\_get(demo, &demo\_ctl);

if (err < 0)

return err;

if (copy\_to\_user(uarg, &demo\_ctl, sizeof(demo\_ctl)))

return -EFAULT;

return err;

default:

/\* 尝试使用驱动程序的 ioctl 接口 \*/

if (ops->ioctl) {

err = ops->ioctl(demo->dev.parent, cmd, arg);

if (err == -ENOIOCTLCMD)

err = -ENOTTY;

} else

err = -ENOTTY;

break;

}

done:

mutex\_unlock(&demo->ops\_lock);

return err;

}

该ioctl接口首先上锁，然后如果是执行写操作的情况判断用户进程的权限，最后调用demo\_interface.c中的接口函数demo\_test\_set()和demo\_test\_get()执行具体的操作。

int demo\_test\_set(struct demo\_device \*demo, struct demo\_ctl\_data \*demo\_ctl)

{

int err = 0;

err = mutex\_lock\_interruptible(&demo->ops\_lock);

if (err)

return err;

if (demo->ops == NULL)

err = -ENODEV;

else if (!demo->ops->set\_data)

err = -EINVAL;

else {

/\* do somerhing \*/

demo->demo\_data.text\_data = demo\_ctl->data;

/\* 调用驱动层接口 \*/

err = demo->ops->set\_data(demo->dev.parent, demo\_ctl);

}

mutex\_unlock(&demo->ops\_lock);

return err;

}

int demo\_test\_get(struct demo\_device \*demo, struct demo\_ctl\_data \*demo\_ctl)

{

int err = 0;

err = mutex\_lock\_interruptible(&demo->ops\_lock);

if (err)

return err;

if (demo->ops == NULL)

err = -ENODEV;

else if (!demo->ops->get\_data)

err = -EINVAL;

else {

/\* do somerhing \*/

demo\_ctl->data = demo->demo\_data.text\_data;

/\* 调用驱动层接口 \*/

err = demo->ops->get\_data(demo->dev.parent, demo\_ctl);

}

mutex\_unlock(&demo->ops\_lock);

return err;

}

本示例程序这两个函数实现的较为简单，分别是设置和返回demo\_data中的text\_data值，然后如果驱动程序实现了自己的get\_data和set\_data函数接口则会调用它们，这里的思想有点类似面向对象中的派生。在前文中已经看到xxx\_demo\_driver驱动程序中已经实现了这两个接口：

static int xxx\_demo\_set\_data(struct device \*dev, struct demo\_ctl\_data \*ctrl\_data)

{

struct xxx\_demo \*xxx\_demo = dev\_get\_drvdata(dev);

printk(KERN\_INFO "xxx demo set data\n");

xxx\_demo->xxx\_demo\_data = ctrl\_data->data;

return 0;

}

static int xxx\_demo\_get\_data(struct device \*dev, struct demo\_ctl\_data \*ctrl\_data)

{

struct xxx\_demo \*xxx\_demo = dev\_get\_drvdata(dev);

printk(KERN\_INFO "xxx demo get data\n");

ctrl\_data->data = xxx\_demo->xxx\_demo\_data;

return 0;

}

驱动程序中的这两个函数会设置和获取驱动程序自己的xxx\_demo\_data，来替代demo子系统中的通用demo\_data，如果此处没有实现这两个接口，则不会改变demo子系统中的值。然后再来看一下fops中的最后一个release接口：

static int demo\_dev\_release(struct inode \*inode, struct file \*file)

{

struct demo\_device \*demo = file->private\_data;

/\* do something while exit \*/

/\* 调用驱动层 release 实现 \*/

if (demo->ops->release)

demo->ops->release(demo->dev.parent);

clear\_bit\_unlock(DEMO\_DEV\_BUSY, &demo->flags);

return 0;

}

这个接口会在应用程序调用close(fd)时调用执行，首先执行一些通用的release操作，然后同样的如果驱动程序实现了自己的release接口则调用驱动自己的接口实现驱动的close操作。下面来看一下用户通过procfs和demo子系统交互的流程，首先来看proc文件系统的接口demo\_proc\_fops

static const struct file\_operations demo\_proc\_fops = {

.open = demo\_proc\_open,

.read = seq\_read,

.llseek = seq\_lseek,

.release = demo\_proc\_release,

};

这里几个函数的实现基于seq\_file子系统，在demo\_proc\_open中创建了seq\_operations结构和seq\_file结构实例并绑定了show函数为demo\_proc\_show，当用户读取/proc/driver/demo/demoX时，会调用seq\_read()->demo\_proc\_show()函数：

static int demo\_proc\_show(struct seq\_file \*seq, void \*offset)

{

int err = 0;

struct demo\_device \*demo = seq->private;

const struct demo\_class\_ops \*ops = demo->ops;

/\* 输出需要的subsys proc信息 \*/

seq\_printf(seq, "demo\_com\_data\t: %ld\n", demo->demo\_data.text\_data);

seq\_printf(seq, "\n");

/\* 输出驱动层proc信息 \*/

if (ops->proc)

err = ops->proc(demo->dev.parent, seq);

return err;

}

这里可以依次调用seq\_printf输出一些子系统方面的通用信息，如果驱动程序也需要输出并提供了proc接口则调用它：

static int xxx\_demo\_proc(struct device \*dev, struct seq\_file \*seq)

{

struct xxx\_demo \*xxx\_demo = dev\_get\_drvdata(dev);

seq\_printf(seq, "xxx\_demo\_data\t: %ld\n", xxx\_demo->xxx\_demo\_data);

seq\_printf(seq, "\n");

return 0;

}

通过该proc接口，应用程序可以非常方便的获取内核demo子系统和驱动上的信息。最后来看下面来看一下用户通过sysfs和demo子系统的交互流程，前文中驱动注册流程中注册了dev\_attr\_demo\_name和dev\_attr\_demo\_data两个通用的属性文件和一个

dev\_attr\_demodata特殊属性文件。其中dev\_attr\_demo\_name属性文件是只读的，所绑定的show接口函数为

static ssize\_t

demo\_name\_show(struct device \*dev, struct device\_attribute \*attr, char \*buf)

{

return sprintf(buf, "%s\n", to\_demo\_device(dev)->name);

}

static DEVICE\_ATTR\_RO(demo\_name);

该函数将设备的名字输出到用户空间。另一个dev\_attr\_demo\_data属性文件为可读可写的，其绑定的show接口和store接口为：

static ssize\_t

demo\_data\_show(struct device \*dev, struct device\_attribute \*attr, char \*buf)

{

return sprintf(buf, "%ld\n", to\_demo\_device(dev)->demo\_data.text\_data);

}

static ssize\_t

demo\_data\_store(struct device \*dev, struct device\_attribute \*attr,

const char \*buf, size\_t n)

{

struct demo\_device \*demo = to\_demo\_device(dev);

unsigned long val = simple\_strtoul(buf, NULL, 0);

if (val >= 4096 || val == 0)

return -EINVAL;

demo->demo\_data.text\_data = (unsigned long)val;

return n;

}

这里分别设置和输出子系统中的demo\_data。最后来看dev\_attr\_demodata的store接口：

static ssize\_t

demo\_sysfs\_set\_demodata(struct device \*dev, struct device\_attribute \*attr,

const char \*buf, size\_t n)

{

struct demo\_device \*demo = to\_demo\_device(dev);

struct demo\_ctl\_data demo\_ctl;

unsigned long val = 0;

ssize\_t retval;

val = simple\_strtoul(buf, NULL, 0);

if (val >= 4096 || val == 0)

retval = -EINVAL;

/\* 调用interface接口写入驱动数据 \*/

demo\_ctl.data = (unsigned long)val;

retval = demo\_test\_set(demo, &demo\_ctl);

return (retval < 0) ? retval : n;

}

这里接收到用户输入的数据后然后封装调用interface中的demo\_test\_set接口，就同ioctl相同，通过sysfs接口用户也可以非常方便的同驱动驱动程序交互。

**三、demo驱动和子系统演示**

首先使用如下Makefile程序进行编译：

ifneq ($(KERNELRELEASE),)

obj-m := demo.o

demo-objs := demo\_core.o demo\_dev.o demo\_interface.o demo\_proc.o demo\_sysfs.o

obj-m += xxx\_demo\_driver.o

obj-m += xxx\_demo\_device.o

else

KDIR := /home/apple/raspberry/build/linux-rpi-4.1.y

all:prepare

make -C $(KDIR) M=$(PWD) modules ARCH=arm CROSS\_COMPILE=arm-bcm2708-linux-gnueabi-

cp \*.ko ./release/

prepare:

mkdir release

modules\_install:

make -C $(KDIR) M=$(PWD) modules\_install ARCH=arm CROSS\_COMPILE=arm-bcm2708-linux-gnueabi-

clean:

rm -f \*.ko \*.o \*.mod.o \*.mod.c \*.symvers modul\*

rm -f ./release/\*

endif

编译后在relseae目录下得到3个模块ko文件demo.ko、xxx\_demo\_device.ko和xxx\_demo\_driver.ko，将他们拷贝到树莓派中依次加载：

root@apple:~# insmod demo.ko

root@apple:~# insmod xxx\_demo\_driver.ko

root@apple:~# insmod xxx\_demo\_device.ko

root@apple:~# lsmod

**Module                  Size  Used by**

**xxx\_demo\_device         1604  0**

**xxx\_demo\_driver         2935  0**

**demo                    9989  1 xxx\_demo\_driver**

加载完成后可以在/dev/目录下看到生成的设备文件：

root@apple:/dev# ls /dev/demo\*    

**/dev/demo0  /dev/demo1**

然后在/proc/driver/demo/目录下看到生成的属性文件：

root@apple:/dev# ls /proc/driver/demo/demo\*

**/proc/driver/demo/demo0  /proc/driver/demo/demo1**

读取其中一个就够可以看到输出了：

root@apple:/dev# cat /proc/driver/demo/demo0  

**demo\_com\_data   : 0**

**xxx\_demo\_data   : 206**

这里的xxx\_demo\_data会一直持续累加（约每1s累加1）

最后在/sys/class/demo目录下可以看到生成的两个目录，他们是指向device目录下对应的链接文件：

root@apple:/sys/class/demo# ls

**demo0  demo1**

root@apple:/sys/class/demo# ls -l

**total 0**

**lrwxrwxrwx 1 root root 0 Jan  1 01:26 demo0 -> ../../devices/platform/xxx\_demo\_device.0/demo/demo0**

**lrwxrwxrwx 1 root root 0 Jan  1 01:27 demo1 -> ../../devices/platform/xxx\_demo\_device.1/demo/demo1**

打开其中一个可以看到：

root@apple:/sys/class/demo/demo0# ls

**demo\_data  demo\_name  demodata  dev  device  subsystem  uevent**

其中demo\_data、demo\_name和demodata就是前文中程序生成的属性文件了，可以读取和写入数值

root@apple:/sys/class/demo/demo0# cat demodata

**0**

root@apple:/sys/class/demo/demo0# echo 100 > demo\_data

root@apple:/sys/class/demo/demo0# cat demodata

**100**

**四、总结**

最后总结一下，设计demo子系统的核心用意就在于能够提取出不同demo驱动中相似的部分进行归一化。底层可以有xxx\_demo\_driver、yyy\_demo\_driver、zzz\_demo\_drive等等，他们主要都是为Linux应用提供一个特定的服务，但是它们可能拥有不同的通信总线，不同的寄存器配置，甚至功能也有细微的差别，但是只要能够设计出类似上文中的这么一个demo子系统就可以把这些驱动外设的差异屏蔽起来，子系统要求驱动程序用一套标准的接口与其对接（不支持的功能可以不用实现），这样子系统就可以对这些驱动进行归一化的管理，结构更为清晰，层次更为分明，代码的重复率大大降低，最终为应用程序提供一套标准的接口，应用层序的设计也可以大大的简化。