

Linux 设备模型浅析之固件篇

本文属本人原创，欢迎转载，转载请注明出处。由于个人的见识和能力有限，不可能面面俱到，也可能存在谬误，敬请网友指出，本人的邮箱是 yzq.seen@gmail.com，博客是 <http://zhigiang0071.cublog.cn>。

Linux 设备模型，仅仅看理论介绍，比如 LDD3 的第十四章，会感觉太抽象不易理解，而通过阅读内核代码就更具体更易理解，所以结合理论介绍和内核代码阅读能够更快速的理解掌握 linux 设备模型。这一序列的文章的目的就是在于此，看这些文章之前最好能够仔细阅读 LDD3 的第十四章。固件 **firmware**，一般是在芯片中可运行的程序代码或配置文件，比如 CY68013 USB2.0 芯片，它里面有一个 8051 内核，可执行代码可以在线下载到其 **ram** 中运行。内核中 **firmware** 模块的作用是使得驱动程序可以调用其 **API** 获得用户空间的固件（一般存放在 **/lib/firmware** 目录下）再通过一定的通信方式（比如 **I2C**、**UART** 和 **USB** 等）下载到芯片里。以下就以 CY68013 为例子说明固件下载的原理。阅读这篇文章之前，最好先阅读文章《Linux 设备模型浅析之设备篇》、《Linux 设备模型浅析之驱动篇》和《Linux 设备模型浅析之驱动篇》。在文章的最后贴有一张针对本例的图片，可在阅读本文章的时候作为参照。

一、cy68013 的 8051 单片机程序是 intel 的 hex 格式，用 keil 软件编译生成。需要下载的可执行程序有两个，一个是"cy68013_loader.hex"程序，另一个是"cy68013_fw.hex"程序。前者是作为下载后者的守护程序，也就是说先下载"cy68013_loader.hex"程序到 cy68013 的 ram 中运行，然后其负责"cy68013_fw.hex"程序的下载运行及固化(可以固化到外接 i2c eeprom 中)。这两个固件存放在/lib/firmware 目录中。其中会用到两个 API，request_firmware()例程和 release_firmware()例程。前者前者的使用方式是 **request_firmware(&fw_entry, "cy68013_loader.hex", &cy68013->udev->dev)**，请求加载一个固件，用来获取 cy68013_loader.hex 固件的数据，数据保存在 fw_entry->data 中，大小是 fw_entry->size。后者的使用方式是 **release_firmware(fw_entry)**，释放获取的资源。request_firmware()例程的定义如下：

```
int request_firmware(const struct firmware **firmware_p, const char *name,
                    struct device *device)
{
    int uevent = 1;
    return _request_firmware(firmware_p, name, device, uevent); // uevent = 1, 会产生 uevent 事件
}
代码中,
```

1. 第一个形参 struct firmware 的定义如下:

```
struct firmware {
    size_t size;           // firmware 的大小
    const u8 *data;        // 指向保存 firmware 数据的 buffer
};
```

本例中，cy68013->udev->dev 生成的 sys 目录是/sys/devices/pci0000:00/0000:00:1d.7/usb2/2-1，后面生成的 firmware device 的目录在该目录下。下面分析 request_firmware()例程。

二、 request firmware()例程会睡眠，直到固件下载完成，其定义如下：

[illegible]

```

{
    struct device *f_dev;
    struct firmware_priv *fw_priv;
    struct firmware *firmware;
    struct builtin_fw *builtin;
    int retval;

    if (!firmware_p)
        return -EINVAL;

    // 分配 struct firmware 结构体，在 release_firmware() 例程中释放
    *firmware_p = firmware = kzalloc(sizeof(*firmware), GFP_KERNEL);
    if (!firmware) {
        dev_err(device, "%s: kmalloc(struct firmware) failed\n",
                __func__);
        retval = -ENOMEM;
        goto out;
    }

    // 先从内置在内核的固件中查找，如果找到了就返回
    for (builtin = __start_builtin_fw; builtin != __end_builtin_fw;
        builtin++) {
        if (strcmp(name, builtin->name)) // 通过名字查找
            continue;
        dev_info(device, "firmware: using built-in firmware %s\n",
                name);
        firmware->size = builtin->size;
        firmware->data = builtin->data;
        return 0;
    }

    if (uevent)
        dev_info(device, "firmware: requesting %s\n", name);

    // 这里做了比较多的工作，后面分析
    retval = fw_setup_device(firmware, &f_dev, name, device, uevent);
    if (retval)
        goto error_kfree_fw;

    fw_priv = dev_get_drvdata(f_dev);

    // 显然，前面已经设置为 1 了，所以会执行
    if (uevent) {
        /* 说明正在执行装载 firmware，设置一个定时器，在定时器超时回调例程中调用
        complete() 例程。*/
        if (loading_timeout > 0) {
            fw_priv->timeout.expires = jiffies + loading_timeout * HZ; // 默认是 10s
            add_timer(&fw_priv->timeout);
        }
    }
}

```

```

/* 通知用户空间，让 udev 或 mdev 处理，后者会再调用一个脚本来处理，该脚本
是/lib/udev/firmware.sh 文件，在后面会分析 */
kobject_uevent(&f_dev->kobj, KOBJ_ADD);
wait_for_completion(&fw_priv->completion); // 阻塞，等待用户空间程序处理完
set_bit(FW_STATUS_DONE, &fw_priv->status);
del_timer_sync(&fw_priv->timeout); // 删除定时器
} else
    wait_for_completion(&fw_priv->completion); // 等待用户空间程序处理完毕

mutex_lock(&fw_lock);
// 检查返回状态，看是否出现了问题
if (!fw_priv->fw->size || test_bit(FW_STATUS_ABORT, &fw_priv->status)) {
    retval = -ENOENT;
    release_firmware(fw_priv->fw);
    *firmware_p = NULL;
}
fw_priv->fw = NULL;
mutex_unlock(&fw_lock);
device_unregister(f_dev); // 删除掉之前注册的 f_dev
goto out;

error_kfree_fw:
    kfree(firmware);
    *firmware_p = NULL;
out:
    return retval;
}

```

代码中，

1. **fw_setup_device()**例程的定义如下：

```

static int fw_setup_device(struct firmware *fw, struct device **dev_p,
                           const char *fw_name, struct device *device,
                           int uevent)
{
    struct device *f_dev;
    struct firmware_priv *fw_priv;
    int retval;

    *dev_p = NULL;
    retval = fw_register_device(&f_dev, fw_name, device); // 注册 f_dev，后面分析
    if (retval)
        goto out;

    /* Need to pin this module until class device is destroyed */
    __module_get(THIS_MODULE); // 增加对本模块的引用

    fw_priv = dev_get_drvdata(f_dev); // 获取私有数据

    fw_priv->fw = fw; // 反向引用
}

```

/* 生成名为"data"的二进制属性文件，其位于/sys/devices/pci0000:00/0000:00:1d.7/usb2/2-1/firmware/2-1 目录下，可以用于读写 firmware 数据 */

```
retval = sysfs_create_bin_file(&f_dev->kobj, &fw_priv->attr_data);
if (retval) {
    dev_err(device, "%s: sysfs_create_bin_file failed\n", __func__);
    goto error_unreg;
}
```

/* 生成名为"loading"的属性文件，其位于/sys/devices/pci0000:00/0000:00:1d.7/usb2/2-1/firmware/2-1 目录下，用于控制 firmware 的加载过程 */

```
retval = device_create_file(f_dev, &dev_attr_loading);
if (retval) {
    dev_err(device, "%s: device_create_file failed\n", __func__);
    goto error_unreg;
}
```

// 设置可发送 uevent 事件，在《Linux 设备模型浅析之 uevent 篇》中曾分析过

```
if (uevent)
    f_dev->uevent_suppress = 0;
*dev_p = f_dev;
goto out;
```

```
error_unreg:
    device_unregister(f_dev);
out:
    return retval;
}
```

代码中，

1.1. dev_attr_loading 的定义如下：

static DEVICE_ATTR(**loading**, 0644, firmware_loading_show, firmware_loading_store)。显然是一个 device 类型的属性结构体，有 firmware_loading_show() 例程和 firmware_loading_store() 例程两个读写方法，分别用于读出加载的命令和写入加载的命令，定义如下：

```
static ssize_t firmware_loading_show(struct device *dev,
                                     struct device_attribute *attr, char *buf)
{
    struct firmware_priv *fw_priv = dev_get_drvdata(dev);
    // 如果是在 FW_STATUS_LOADING 状态，则变量 loading 为 1
    int loading = test_bit(FW_STATUS_LOADING, &fw_priv->status);
    return sprintf(buf, "%d\n", loading); // 最终会输出到用户空间
}
```

```
static ssize_t firmware_loading_store(struct device *dev,
                                     struct device_attribute *attr,
                                     const char *buf, size_t count)
{
    struct firmware_priv *fw_priv = dev_get_drvdata(dev);
    int loading = simple_strtol(buf, NULL, 10); // 将用户空间传递的值保存在 loading 变量中
```

```

switch (loading) {
case 1:      // 标志着开始加载固件，要清除之前所获取的资源
    mutex_lock(&fw_lock);
    if (!fw_priv->fw) { // 如果清除好了，则直接返回
        mutex_unlock(&fw_lock);
        break;
    }
    vfree(fw_priv->fw->data); // 释放存放固件数据的 buffer
    fw_priv->fw->data = NULL;
    fw_priv->fw->size = 0;
    fw_priv->alloc_size = 0;
    // 设置状态为 FW_STATUS_LOADING
    set_bit(FW_STATUS_LOADING, &fw_priv->status);
    mutex_unlock(&fw_lock);
    break;
case 0: // 停止加载固件
    // 如果正在 loading，则唤醒被阻塞的例程
    if (test_bit(FW_STATUS_LOADING, &fw_priv->status)) {
        complete(&fw_priv->completion);
        clear_bit(FW_STATUS_LOADING, &fw_priv->status);
        break;
    }
    /* fallthrough */
default:
    dev_err(dev, "%s: unexpected value (%d)\n", __func__, loading);
    /* fallthrough */
case -1: // 由于产生了错误，取消固件的加载，并丢弃任何已经加载的数据
    fw_load_abort(fw_priv);
    break;
}

return count;
}

```

1.2. fw_register_device()例程代码如下：

```

static int fw_register_device(struct device **dev_p, const char *fw_name,
                             struct device *device)
{
    int retval;
    // 获取 struct firmware_priv 大小的内存
    struct firmware_priv *fw_priv = kzalloc(sizeof(*fw_priv),
                                             GFP_KERNEL);

    // 获取 struct device 大小的内存
    struct device *f_dev = kzalloc(sizeof(*f_dev), GFP_KERNEL);

    *dev_p = NULL;
}

```

```

if (!fw_priv || !f_dev) {
    dev_err(device, "%s: kmalloc failed\n", __func__);
    retval = -ENOMEM;
    goto error_kfree;
}

init_completion(&fw_priv->completion);

// 用于生成名为"data"的二进制属性文件，该结构体后面分析
fw_priv->attr_data = firmware_attr_data_tmpl;

// 拷贝 fw_name 到 fw_priv->fw_id
strncpy(fw_priv->fw_id, fw_name, FIRMWARE_NAME_MAX);

// 设置软定时器回调例程，用于设定用户加载固件的时间，后面会分析该回调例程
fw_priv->timeout.function = firmware_class_timeout;
fw_priv->timeout.data = (u_long) fw_priv; // 设置私有数据，将传送给回调例程
init_timer(&fw_priv->timeout); // 初始化时钟

dev_set_name(f_dev, dev_name(device)); // 名字跟父设备的名字相同，也就是“2-1”
f_dev->parent = device; // 在本例中是 usb_device.dev
f_dev->class = &firmware_class; // 类型是 firmware_class，后面会分析
dev_set_drvdata(f_dev, fw_priv); // 保存私有数据到 f_dev
f_dev->uevent_suppress = 1;
/* 注册到设备模型中，生成/sys/devices/pci0000:00/0000:00:1d.7/usb2/2-1/firmware/2-1 目录，该例程的具体实现过程可参照《Linux 设备模型浅析之设备篇》。*/
retval = device_register(f_dev);
if (retval) {
    dev_err(device, "%s: device_register failed\n", __func__);
    goto error_kfree;
}
*dev_p = f_dev;
return 0;

error_kfree:
kfree(fw_priv);
kfree(f_dev);
return retval;
}

```

代码中，

1.2.1. 属性结构体 `firmware_attr_data_tmpl` 的定义如下：

```

static struct bin_attribute firmware_attr_data_tmpl = {
    .attr = {.name = "data", .mode = 0644},
    .size = 0,
    .read = firmware_data_read,
    .write = firmware_data_write,
};

```

显然，其定义了名为"data"的属性文件，有 `firmware_data_read()`和 `firmware_data_write()`两个读

写的方法，前者将 buffer 中的固件数据读出，后者将固件数据写到 buffer 中，都是被应用程序间接调用，分别定义如下：

```
static ssize_t
firmware_data_read(struct kobject *kobj, struct bin_attribute *bin_attr,
                    char *buffer, loff_t offset, size_t count)
{
    struct device *dev = to_dev(kobj);
    // 获取私有数据，在前面的 fw_register_device() 例程中保存了此私有数据
    struct firmware_priv *fw_priv = dev_get_drvdata(dev);
    struct firmware *fw;
    ssize_t ret_count;

    mutex_lock(&fw_lock);
    fw = fw_priv->fw;
    // 判读是否已经完成
    if (!fw || test_bit(FW_STATUS_DONE, &fw_priv->status)) {
        ret_count = -ENODEV;
        goto out;
    }
    // 从 fw->data 中将固件数据拷贝到 buffer，此 buffer 中的数据最终传递到用户空间
    ret_count = memory_read_from_buffer(buffer, count, &offset,
                                         fw->data, fw->size);
out:
    mutex_unlock(&fw_lock);
    return ret_count;
}

static ssize_t
firmware_data_write(struct kobject *kobj, struct bin_attribute *bin_attr,
                    char *buffer, loff_t offset, size_t count)
{
    struct device *dev = to_dev(kobj);
    struct firmware_priv *fw_priv = dev_get_drvdata(dev);
    struct firmware *fw;
    ssize_t retval;

    if (!capable(CAP_SYS_RAWIO))
        return -EPERM;

    mutex_lock(&fw_lock);
    fw = fw_priv->fw;
    // 判读是否已经完成
    if (!fw || test_bit(FW_STATUS_DONE, &fw_priv->status)) {
        retval = -ENODEV;
        goto out;
    }
    // 根据固件数据的实际大小重新分配内存
    retval = fw_realloc_buffer(fw_priv, offset + count);
    if (retval)
```

```

        goto out;

// 将固件数据拷贝到 fw->data 中
memcpy((u8 *)fw->data + offset, buffer, count);

fw->size = max_t(size_t, offset + count, fw->size);
retval = count;
out:
    mutex_unlock(&fw_lock);
    return retval;
}

```

1.2.2. firmware_class_timeout()超时回调例程定义如下:

```

static void firmware_class_timeout(u_long data)
{
    struct firmware_priv *fw_priv = (struct firmware_priv *) data;
    fw_load_abort(fw_priv);
}

```

代码中,

1.2.2.1. 调用了例程 fw_load_abort(), 其定义如下:

```

static void fw_load_abort(struct firmware_priv *fw_priv)
{
    set_bit(FW_STATUS_ABORT, &fw_priv->status); // 超时了, 所以 abort
    wmb();
    complete(&fw_priv->completion); // 唤醒阻塞的进程
}

```

1.2.3. firmware_class 的定义如下:

```

static struct class firmware_class = {
    .name          = "firmware", // 将会生成/sys/class/firmware 目录
    // 在《Linux 设备模型浅析之 uevent 篇》说明过, 在产生 uevent 事件时输出环境变量
    .dev_uevent    = firmware_uevent,
    .dev_release   = fw_dev_release, // 释放所有资源, 主要是释放内存
}

```

其在 firmware_class_init()例程中被初始化, 该例程定义如下:

```

static int __init firmware_class_init(void)
{
    int error;
    // 注册 firmware_class 并初始化, 生成/sys/class/firmware 目录
    error = class_register(&firmware_class);
    if (error) {
        printk(KERN_ERR "%s: class_register failed\n", __func__);
        return error;
    }
    /* 生成/sys/class/firmware/timeout 属性文件, 用于获取和设置固件下载超时时间, 后面分析 */
    error = class_create_file(&firmware_class, &class_attr_timeout);
    if (error) {

```



```

        printk(KERN_ERR "%s: class_create_file failed\n",
               __func__);
        class_unregister(&firmware_class);
    }
    return error;
}

```

代码中，

1.2.3.1. class_attr_timeout 定义如下：

static CLASS_ATTR(timeout, 0644, firmware_timeout_show, firmware_timeout_store)，显然定义了一个 class 类的属性结构体，firmware_timeout_show 例程和 firmware_timeout_store 例程分别定义如下：

```

static ssize_t firmware_timeout_show(struct class *class, char *buf)
{
    return sprintf(buf, "%d\n", loading_timeout); // 将变量 loading_timeout 值向用户空间传递
}
static ssize_t firmware_timeout_store(struct class *class, const char *buf, size_t count)
{
    // 将用户空间传递进来的值赋给变量 loading_timeout
    loading_timeout = simple_strtol(buf, NULL, 10);
    if (loading_timeout < 0)
        loading_timeout = 0;
    return count;
}

```

2. 现在说说之前提到过的 **firmware.sh** shell 脚本，其定义如下：

```
#!/bin/sh -e
```

```

// 固件存放的目录，通常是位于后者，/lib/firmware 中
FIRMWARE_DIRS="/lib/firmware/$(uname -r) /lib/firmware"

```

```

err() {
    echo "$@" >&2
    logger -t "${0##*/}[$$]" "$@" 2>/dev/null || true
}

```

/* 先判断"loading"属性文件是否存在。DEVPATH 是由内核通过 uevent 事件输出的环境变量(一个目录)，在本例中是/sys/devices/pci0000:00/0000:00:1d.7/usb2/2-1/firmware/2-1 目录。*/

```

if [ ! -e /sys$DEVPATH/loading ]; then
    err "udev firmware loader misses sysfs directory"
    exit 1
fi

```

// 从前面的定义中可看出，FIRMWARE_DIRS 有两个可能的目录，/lib/firmware/\$(uname -r) 和 /lib/firmware，所以做个遍历加载。

```

for DIR in $FIRMWARE_DIRS; do
    [ -e "$DIR/$FIRMWARE" ] || continue // 如果目录存在就执行
    echo 1 > /sys$DEVPATH/loading // 开始加载
done

```

```

        cat "$DIR/$FIRMWARE" > /sys$DEVPATH/data // 将固件数据写入到“data”属性文件中
        echo 0 > /sys$DEVPATH/loading // 停止加载
        exit 0 // 成功，则返回
done

// 如果执行到这里，说明没有找到固件，故写入-1
echo -1 > /sys$DEVPATH/loading
err "Cannot find firmware file '$FIRMWARE'"
exit 1

```

三、此时获取到了固件数据，但还要通过 USB 将其下载到 CY68013 芯片中运行。具体过程不在本文的内容范围内，就不予讲述分析了。

四、最后调用 release_firmware() 例程释放掉在 request_firmware() 例程中申请的资源，该例程的定义如下：

```

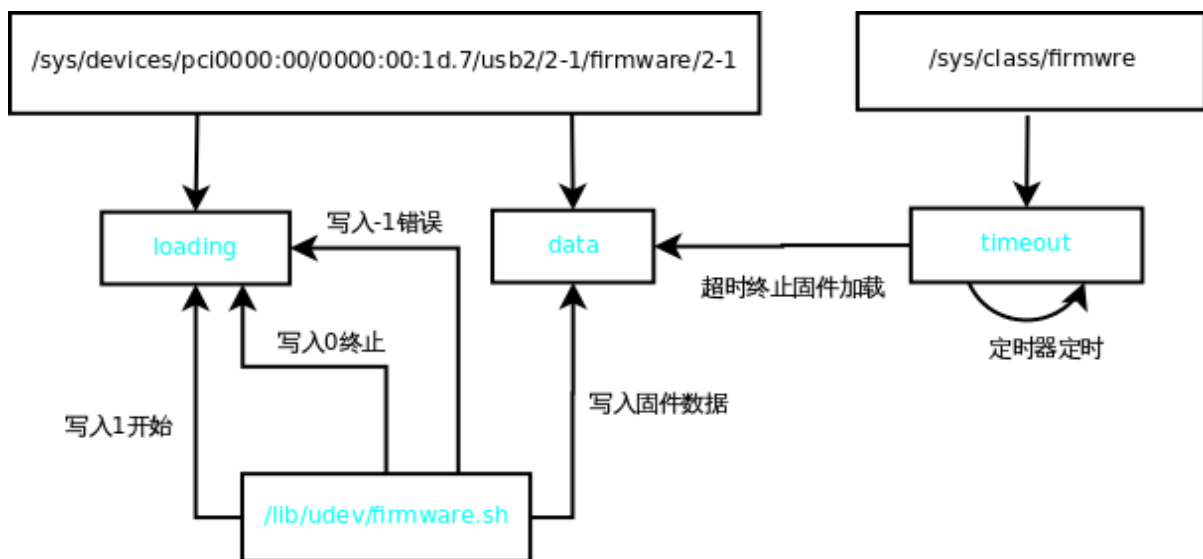
void
release_firmware(const struct firmware *fw)
{
    struct builtin_fw *builtin;

    if (fw) {
        for (builtin = __start_builtin_fw; builtin != __end_builtin_fw;
             builtin++) {
            if (fw->data == builtin->data)
                goto free_fw;
        }
        vfree(fw->data); // 如果固件不是内置在内核中，则释放掉存放固件的 fw->data
    free_fw:
        kfree(fw); // 释放掉结构体 fw
    }
}

```

至此，一次固件的加载就这样完成了。总体而言，固件的加载是通过 linux 设备模型产生一个 uevent 事件，通知用户空间的程序 udev 或 mdev 来实现。后者再调用脚本 firmware.sh（PC 机上）将固件数据通过属性文件“data”写入到内核中，然后驱动程序通过一定的通信方式将其下载到芯片中，比如本例中的 CY68013 芯片，PC 机就是通过 USB 总线下载到其中的。。

附图：



注：

1. 其中黑色字体的矩形表示是个文件夹；
2. 其中青色字体的矩形表示是个文件；