> 作 者：道哥，10+年嵌入式开发老兵，专注于：C/C++、嵌入式、Linux。
>
> 关注下方公众号，回复【书籍】，获取 Linux、嵌入式领域经典书籍；回复【PDF】，获取所有原创文章(PDF 格式)。

目录

别人的经验，我们的阶梯！

大家好，我是道哥，今天我为大伙儿解说的技术知识点是：【驱动层中，如何发送信号给应用程序】。

在上一篇文章中，我们讨论的是：在应用层如何发送指令来控制驱动层的 GPIOLinux驱动实践：如何编写【GPIO】设备的驱动程序？。控制的方向是从应用层到驱动层：

那么，如果想让程序的执行路径从下往上，也就是从驱动层传递到应用层，应该如何实现呢？

最容易、最简单的方式，就是通过发送信号！

这篇文章继续以完整的代码实例来演示如何实现这个功能。

kill 命令和信号

使用 kill 命令发送信号

关于 Linux操作系统的信号，每位程序员都知道这个指令：使用 kill 工具来“杀死”一个进程：

```
$ kill -9 <进程的 PID>
```

这个指令的功能是：向指定的某个进程发送一个信号 9，这个信号的默认功能是：是停止进程。

虽然在应用程序中没有主动处理这个信号，但是操作系统默认的处理动作是终止应用程序的执行。

除了发送信号 9，kill 命令还可以发送其他的任意信号。

在 Linux 系统中，所有的信号都使用一个整型数值来表示，可以打开文件 /usr/include/x86_64-linux-gnu/bits/signum.h(你的系统中可能位于其他的目录) 查看一下，比较常见的几个信号是：

```
/* Signals.  */
```

```
#define SIGINT    2 /* Interrupt (ANSI).  */
#define SIGKILL   9 /* Kill, unblockable (POSIX).  */
#define SIGUSR1   10  /* User-defined signal 1 (POSIX).  */
#define SIGSEGV   11  /* Segmentation violation (ANSI).  */
#define SIGUSR2   12  /* User-defined signal 2 (POSIX).  */
...
...
#define SIGSYS    31  /* Bad system call.  */
#define SIGUNUSED 31

#define _NSIG   65  /* Biggest signal number + 1
           (including real-time signals).  */

/* These are the hard limits of the kernel.  These values should not be
   used directly at user level.  */
#define __SIGRTMIN  32
#define __SIGRTMAX  (_NSIG - 1)
```

信号 9 对应着 SIGKILL，而信号11（SIGSEGV）就是最令人讨厌的Segmentfault！

这里还有一个地方需要注意一下：实时信号和非实时信号，它俩的主要区别是：

1. 非实时信号：操作系统不确保应用程序一定能接收到(即：信号可能会丢失);
2. 实时信号：操作系统确保应用程序一定能接收到;

如果我们的程序设计，通过信号机制来完成一些功能，那么为了确保信号不会丢失，肯定是使用实时信号的。

从文件 signum.h 中可以看到，实时信号从 __SIGRTMIN（数值：32） 开始。

多线程中的信号

我们在编写应用程序时，虽然没有接收并处理 SIGKILL 这个信号，但是一旦别人发送了这个信号，我们的程序就被操作系统停止掉了，这是默认的动作。

那么，在应用程序中，应该可以主动声明接收并处理指定的信号，下面就来写一个最简单的实例。

> 在一个应用程序中，可能存在多个线程;
>
> 当有一个信号发送给此进程时，所有的线程都可能接收到，但是只能有一个线程来处理;
>
> 在这个示例中，只有一个主线程来接收并处理信号;

信号注册和处理函数

按照惯例，所有应用程序文件都创建在 ~/tmp/App 目录中。

```
// 文件：tmp/App/app_handle_signal/app_handle_signal.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <signal.h>
```

```
// 信号处理函数
static void signal_handler(int signum, siginfo_t *info, void *context)
{
  // 打印接收到的信号值
    printf("signal_handler: signum = %d \n", signum);
}

int main(void)
{
  int count = 0;
  // 注册信号处理函数
  struct sigaction sa;
  sigemptyset(&sa.sa_mask);
  sa.sa_sigaction = &signal_handler;
  sa.sa_flags = SA_SIGINFO;
  sigaction(SIGUSR1, &sa, NULL);
  sigaction(SIGUSR2, &sa, NULL);

  // 一直循环打印信息，等待接收发信号
  while (1)
  {
    printf("app_handle_signal is running...count = %d \n", ++count);
    sleep(5);
  }

  return 0;
}
```

这个示例程序接收的信号是 SIGUSR1 和 SIGUSR2，也就是数值 10 和 12。

编译、执行：

```
$ gcc app_handle_signal.c -o app_handle_signal
$ ./app_handle_signal
```

此时，应用程序开始执行，等待接收信号。

在另一个终端中，使用kill指令来发送信号SIGUSR1或者SIGUSR2。

> kill 发送信号，需要知道应用程序的 PID，可以通过指令: ps -au | grep app_handle_signal 来查看。

```
captain@ubuntu:app_handle_signal$ ps -au | grep app_handle_signal
captain   14788  0.0  0.0   4352   656 pts/20   S+   12:32   0:00 ./app_handle_signal
captain   14796  0.0  0.0  15964  1032 pts/21   S+   12:32   0:00 grep --color=auto app_handle_signal
```

其中的15428就是进程的PID。

执行发送信号SIGUSR1指令：

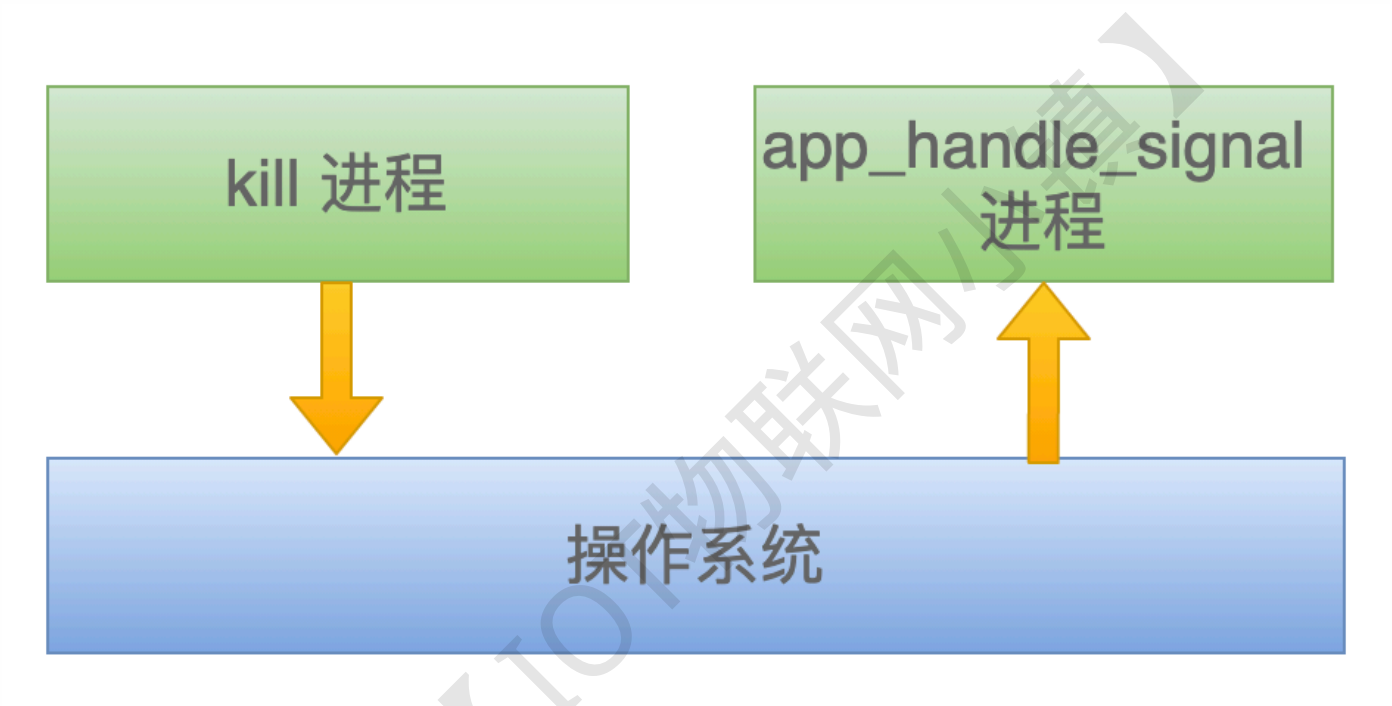
```
$ kill -10 15428
```

此时，在应用程序的终端窗口中，就能看到下面的打印信息：

```
captain@ubuntu:app_handle_signal$ ./app_handle_signal
app_handle_signal is running...count = 1
app_handle_signal is running...count = 2
app_handle_signal is running...count = 3
signal_handler: signum = 10
```

说明应用程序接收到了 SIGUSR1 这个信号！

注意：我们是使用kill命令来发送信号的，kill也是一个独立的进程，程序的执行路径如下：



在这个执行路径中，我们可控的部分是应用层，至于操作系统是如何接收kill的操作，然后如何发送信号给app_handle_signal 进程的，我们不得而知。

下面就继续通过示例代码来看一下如何在驱动层主动发送信号。

驱动程序代码示例：发送信号

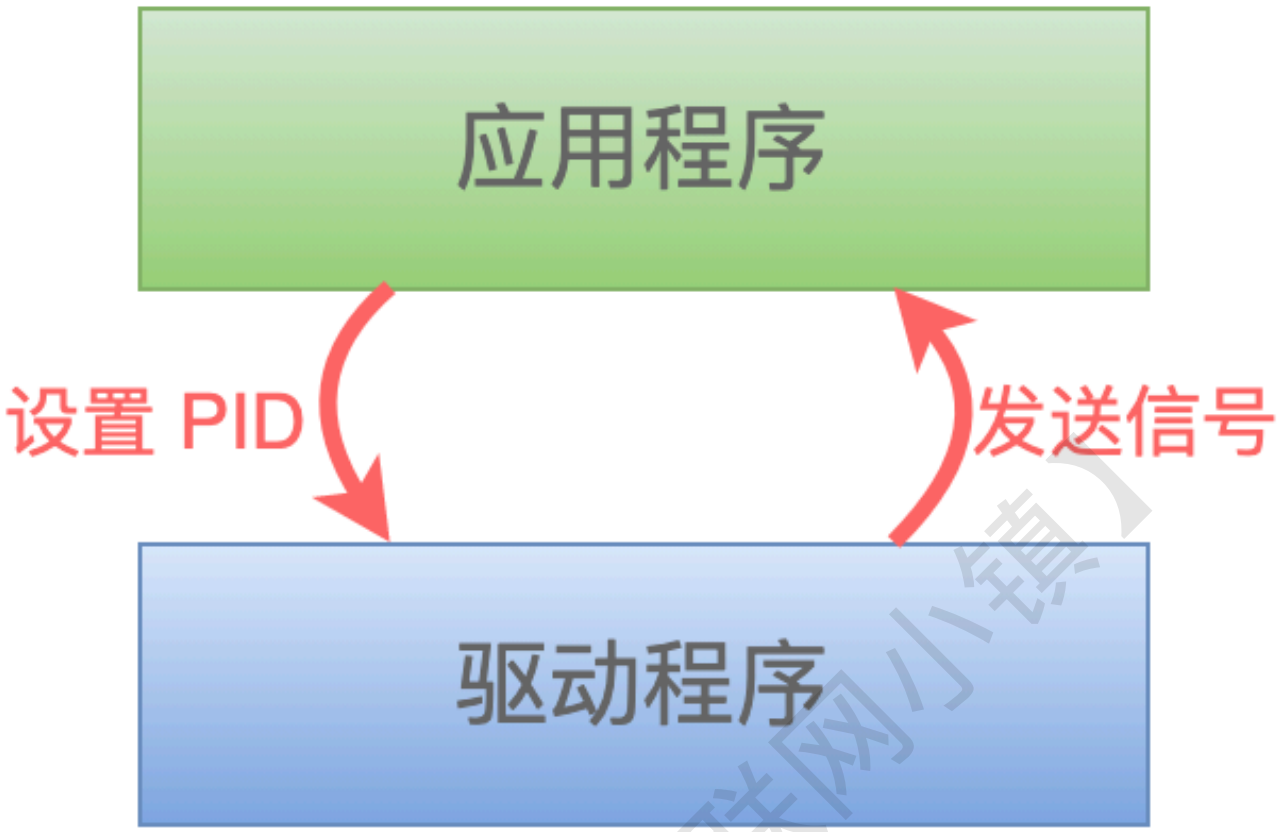
功能需求

在刚才的简单示例中，可以得出下面这些信息：

1. 信号发送方：必须知道向谁[PID]发送信号，发送哪个信号;
2. 信号接收方：必须定义信号处理函数，并且向操作系统注册：接收哪些信号;

发送方当然就是驱动程序了，在示例代码中，继续使用 SIGUSR1 信号来测试。

那么，驱动程序如何才能知道应用程序的PID呢？可以让应用程序通过oictl函数，把自己的PID主动告诉驱动程序：



驱动程序

这里的示例代码，是在上一篇文章的基础上修改的，改动部分的内容，使用宏定义 MY_SIGNAL_ENABLE 控制起来，方便查看和比较。

> 以下所有操作的工作目录，都是与上一篇文章相同的，即：~/tmp/linux-4.15/drivers/。

```
$ cd ~/tmp/linux-4.15/drivers/
$ mkdir my_driver_signal
$ cd my_driver_signal
$ touch my_driver_signal.c
```

my_driver_signal.c 文件的内容如下(不需要手敲，文末有代码下载链接):

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/ctype.h>
#include <linux/device.h>
#include <linux/cdev.h>

// 新增的头文件
#include <asm/siginfo.h>
#include <linux/pid.h>
#include <linux/uaccess.h>
```

```
#include <linux/sched/signal.h>
#include <linux/pid_namespace.h>

// GPIO 硬件相关宏定义
#define MYGPIO_HW_ENABLE

// 新增部分，使用这个宏控制起来
#define MY_SIGNAL_ENABLE

// 设备名称
#define MYGPIO_NAME     "mygpio"

// 一共有４个GPIO
#define MYGPIO_NUMBER   4

// 设备类
static struct class *gpio_class;

// 用来保存设备
struct cdev gpio_cdev[MYGPIO_NUMBER];

// 用来保存设备号
int gpio_major = 0;
int gpio_minor = 0;

#ifdef MY_SIGNAL_ENABLE
// 用来保存向谁发送信号，应用程序通过 ioctl 把自己的进程 ID 设置进来。
static int g_pid = 0;
#endif

#ifdef MYGPIO_HW_ENABLE
// 硬件初始化函数，在驱动程序被加载的时候(gpio_driver_init)被调用
static void gpio_hw_init(int gpio)
{
  printk("gpio_hw_init is called: %d. \n", gpio);
}

// 硬件释放
static void gpio_hw_release(int gpio)
{
  printk("gpio_hw_release is called: %d. \n", gpio);
}

// 设置硬件GPIO的状态，在控制GPIO的时候(gpio_ioctl)被调研
static void gpio_hw_set(unsigned long gpio_no, unsigned int val)
{
  printk("gpio_hw_set is called. gpio_no = %ld, val = %d. \n", gpio_no, val);
}
#endif

#ifdef MY_SIGNAL_ENABLE
// 用来发送信号给应用程序
```

```
static void send_signal(int sig_no)
{
  int ret;
  struct siginfo info;
  struct task_struct *my_task = NULL;
  if (0 == g_pid)
  {
    // 说明应用程序没有设置自己的 PID
      printk("pid[%d] is not valid! \n", g_pid);
      return;
  }

  printk("send signal %d to pid %d \n", sig_no, g_pid);

  // 构造信号结构体
  memset(&info, 0, sizeof(struct siginfo));
  info.si_signo = sig_no;
  info.si_errno = 100;
  info.si_code = 200;

  // 获取自己的任务信息，使用的是 RCU 锁
  rcu_read_lock();
  my_task = pid_task(find_vpid(g_pid), PIDTYPE_PID);
  rcu_read_unlock();

  if (my_task == NULL)
  {
      printk("get pid_task failed! \n");
      return;
  }

  // 发送信号
  ret = send_sig_info(sig_no, &info, my_task);
  if (ret < 0)
  {
         printk("send signal failed! \n");
  }
}
#endif

// 当应用程序打开设备的时候被调用
static int gpio_open(struct inode *inode, struct file *file)
{

  printk("gpio_open is called. \n");
  return 0;
}

#ifdef MY_SIGNAL_ENABLE
static long gpio_ioctl(struct file* file, unsigned int cmd, unsigned long arg)
{
  void __user *pArg;
```

```
  printk("gpio_ioctl is called. cmd = %d \n", cmd);
  if (100 == cmd)
  {
    // 说明应用程序设置进程的 PID
    pArg = (void *)arg;
    if (!access_ok(VERIFY_READ, pArg, sizeof(int)))
    {
        printk("access failed! \n");
        return -EACCES;
    }

    // 把用户空间的数据复制到内核空间
    if (copy_from_user(&g_pid, pArg, sizeof(int)))
    {
        printk("copy_from_user failed! \n");
        return -EFAULT;
    }

    printk("save g_pid success: %d \n", g_pid);
    if (g_pid > 0)
    {
      // 发送信号
      send_signal(SIGUSR1);
      send_signal(SIGUSR2);
    }
  }

  return 0;
}
#else
// 当应用程序控制GPIO的时候被调用
static long gpio_ioctl(struct file* file, unsigned int val, unsigned long gpio_no)
{
  printk("gpio_ioctl is called. \n");

  if (0 != val && 1 != val)
  {
    printk("val is NOT valid! \n");
    return 0;
  }

  if (gpio_no >= MYGPIO_NUMBER)
  {
    printk("dev_no is invalid! \n");
    return 0;
  }

  printk("set GPIO: %ld to %d. \n", gpio_no, val);

#ifdef MYGPIO_HW_ENABLE
  gpio_hw_set(gpio_no, val);
#endif
```

```
  return 0;
}
#endif

static const struct file_operations gpio_ops={
  .owner = THIS_MODULE,
  .open  = gpio_open,
  .unlocked_ioctl = gpio_ioctl
};

static int __init gpio_driver_init(void)
{
  int i, devno;
  dev_t num_dev;

  printk("gpio_driver_init is called. \n");

  // 动态申请设备号(严谨点的话，应该检查函数返回值)
  alloc_chrdev_region(&num_dev, gpio_minor, MYGPIO_NUMBER, MYGPIO_NAME);

  // 获取主设备号
  gpio_major = MAJOR(num_dev);
  printk("gpio_major = %d. \n", gpio_major);

  // 创建设备类
  gpio_class = class_create(THIS_MODULE, MYGPIO_NAME);

  // 创建设备节点
  for (i = 0; i < MYGPIO_NUMBER; ++i)
  {
    // 设备号
    devno = MKDEV(gpio_major, gpio_minor + i);

    // 初始化cdev结构
    cdev_init(&gpio_cdev[i], &gpio_ops);

    // 注册字符设备
    cdev_add(&gpio_cdev[i], devno, 1);

    // 创建设备节点
    device_create(gpio_class, NULL, devno, NULL, MYGPIO_NAME"%d", i);
  }

#ifdef MYGPIO_HW_ENABLE
  for (i = 0; i < MYGPIO_NUMBER; ++i)
  {
    // 初始硬件GPIO
    gpio_hw_init(i);
  }
#endif
```

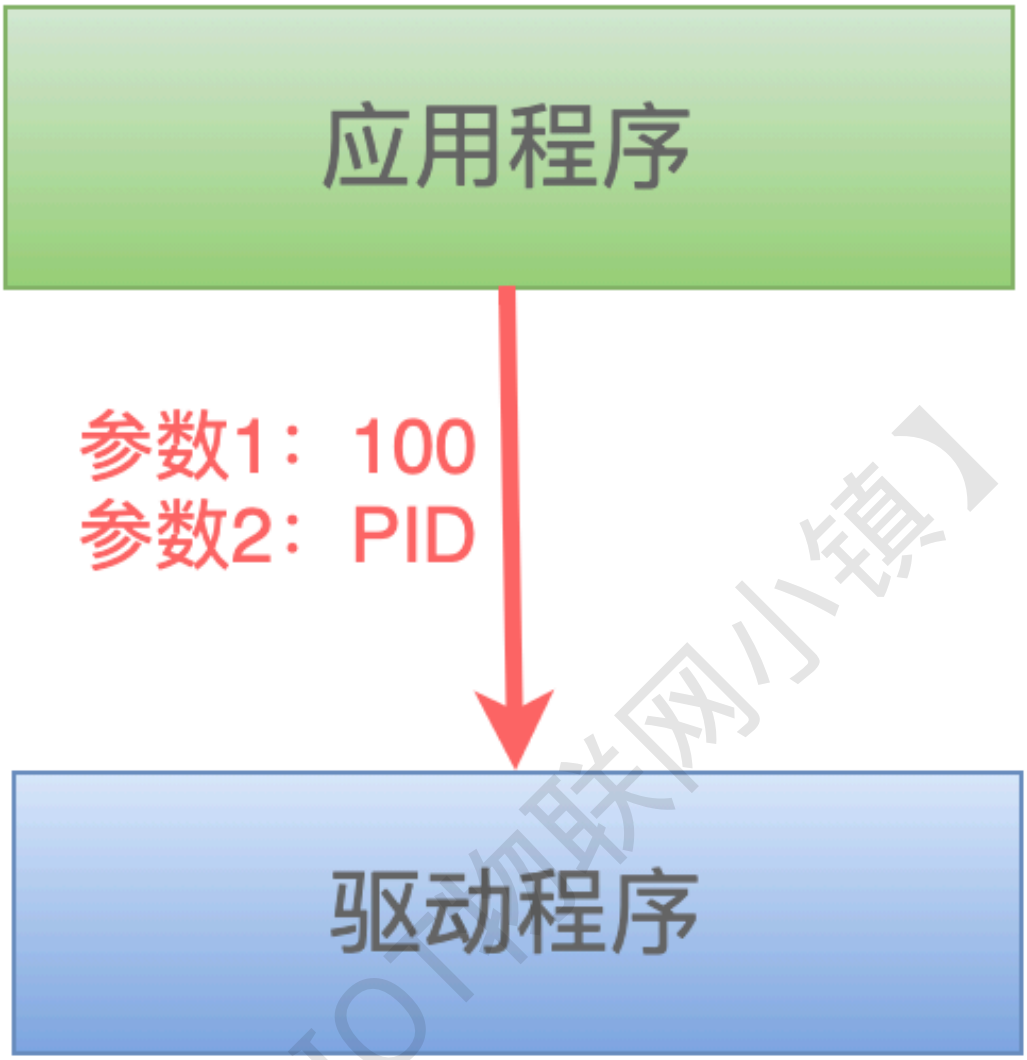
```
  return 0;
}

static void __exit gpio_driver_exit(void)
{
  int i;
  printk("gpio_driver_exit is called. \n");

  // 删除设备节点
  for (i = 0; i < MYGPIO_NUMBER; ++i)
  {
    cdev_del(&gpio_cdev[i]);
    device_destroy(gpio_class, MKDEV(gpio_major, gpio_minor + i));
  }

  // 释放设备类
  class_destroy(gpio_class);

#ifdef MYGPIO_HW_ENABLE
  for (i = 0; i < MYGPIO_NUMBER; ++i)
  {
    gpio_hw_release(i);
  }
#endif

  // 注销设备号
  unregister_chrdev_region(MKDEV(gpio_major, gpio_minor), MYGPIO_NUMBER);
}

MODULE_LICENSE("GPL");
module_init(gpio_driver_init);
module_exit(gpio_driver_exit);
```

这里大部分的代码，在上一篇文章中已经描述的比较清楚了，这里把重点关注放在这两个函数上：gpio_ioctl 和 send_signal。

（1）函数 gpio_ioctl

当应用程序调用 ioctl() 的时候，驱动程序中的 gpio_ioctl 就会被调用。

这里定义一个简单的协议：当应用程序调用参数中 cmd 为 100 的时候，就表示用来告诉驱动程序自己的 PID。



驱动程序定义了一个全局变量 g_pid，用来保存应用程序传入的参数PID。

> 需要调用函数 copy_from_user(&g_pid, pArg, sizeof(int))，把用户空间的参数复制到内核空间中;

成功取得PID之后，就调用函数 send_signal 向应用程序发送信号。

> 这里仅仅是用于演示目的，在实际的项目中，可能会根据接收到硬件触发之后再发送信号。

（2）函数 send_signal

这个函数主要做了3件事情：

> 1. 构造一个信号结构体变量：struct siginfo info;
> 2. 通过应用程序传入的 PID，获取任务信息：pid_task(find_vpid(g_pid), PIDTYPE_PID);
> 3. 发送信号：send_sig_info(sig_no, &info, my_task);

驱动模块 Makefile

```
$ touch Makefile
```

内容如下：

```
ifneq ($(KERNELRELEASE),)
  obj-m := my_driver_signal.o
else
  KERNELDIR ?= /lib/modules/$(shell uname -r)/build
  PWD := $(shell pwd)
default:
  $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
  $(MAKE) -C $(KERNEL_PATH) M=$(PWD) clean
endif
```

编译驱动模块

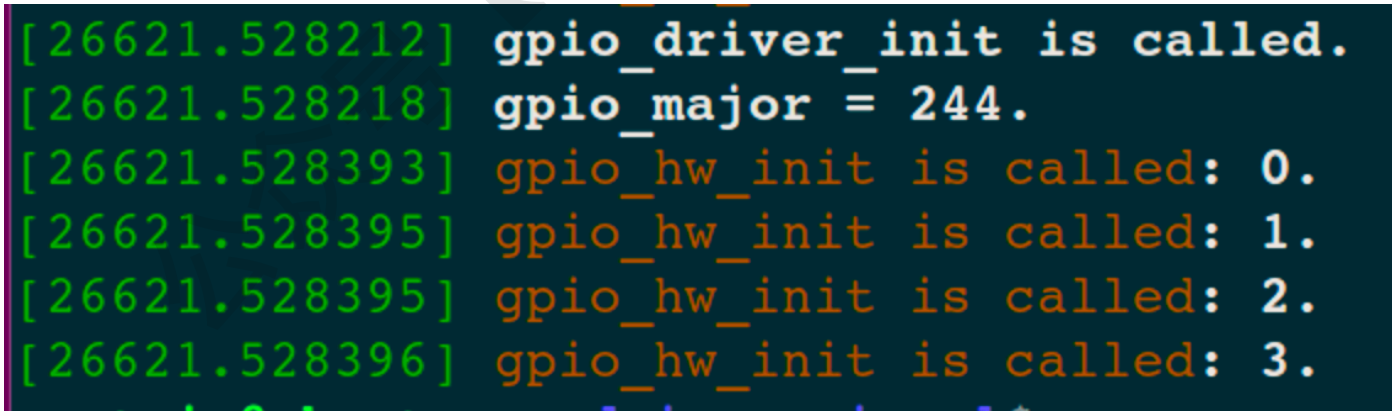
```
$ make
```

得到驱动程序： my_driver_signal.ko 。

加载驱动模块

```
$ sudo insmod my_driver_signal.ko
```

通过 dmesg 指令来查看驱动模块的打印信息：



因为示例代码是在上一篇GPIO的基础上修改的，因此创建的设备节点文件，与上篇文章是一样的：

```
captain@ubuntu:my_driver_signal$ ll /dev/mygpio*
crw------- 1 root root 244, 0 Dec  5 19:40 /dev/mygpio0
crw------- 1 root root 244, 1 Dec  5 19:40 /dev/mygpio1
crw------- 1 root root 244, 2 Dec  5 19:40 /dev/mygpio2
crw------- 1 root root 244, 3 Dec  5 19:40 /dev/mygpio3
```

应用程序代码示例：接收信号

注册信号处理函数

> 应用程序仍然放在 ~/tmp/App/ 目录下。

```
$ mkdir ~/tmp/App/app_mysignal
$ cd ~/tmp/App/app_mysignal
$ touch mysignal.c
```

文件内容如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <signal.h>

#define MY_GPIO_NUMBER    4

char gpio_name[MY_GPIO_NUMBER][16] = {
  "/dev/mygpio0",
  "/dev/mygpio1",
  "/dev/mygpio2",
  "/dev/mygpio3"
};

// 信号处理函数
static void signal_handler(int signum, siginfo_t *info, void *context)
{
  // 打印接收到的信号值
    printf("signal_handler: signum = %d \n", signum);
    printf("signo = %d, code = %d, errno = %d \n",
           info->si_signo,
           info->si_code,
           info->si_errno);
}
```

```
int main(int argc, char *argv[])
{
  int fd, count = 0;
  int pid = getpid();

  // 打开GPIO
  if((fd = open("/dev/mygpio0", O_RDWR | O_NDELAY)) < 0){
    printf("open dev failed! \n");
    return -1;
  }

  printf("open dev success! \n");

  // 注册信号处理函数
  struct sigaction sa;
  sigemptyset(&sa.sa_mask);
  sa.sa_sigaction = &signal_handler;
  sa.sa_flags = SA_SIGINFO;

  sigaction(SIGUSR1, &sa, NULL);
  sigaction(SIGUSR2, &sa, NULL);

  // set PID
  printf("call ioctl. pid = %d \n", pid);
  ioctl(fd, 100, &pid);

  // 休眠1秒，等待接收信号
  sleep(1);

  // 关闭设备
  close(fd);
}
```

可以看到，应用程序主要做了两件事情：

（1）首先通过函数 `sigaction()` 向操作系统注册了信号 SIGUSR1 和 SIGUSR2，它俩的信号处理函数是同一个：`signal_handler()`。

> 除了 sigaction 函数，应用程序还可以使用 signal 函数来注册信号处理函数；

（2）然后通过 `ioctl(fd, 100, &pid);` 向驱动程序设置自己的 PID。

编译应用程序：

```
$ gcc mysignal.c -o mysignal
```

执行应用程序：

```
$ sudo ./mysignal
```

根据刚才驱动程序的代码，当驱动程序接收到设置PID的命令之后，会立刻发送两个信号：

```
if (g_pid > 0)
{
        // 发送信号
        send_signal(SIGUSR1);
        send_signal(SIGUSR2);
}
```

先来看一下 dmesg 中驱动程序的打印信息：

```
[27557.970763] gpio_open is called.
[27557.970829] gpio_ioctl is called. cmd = 100
[27557.970831] save g_pid success: 6259
[27557.970832] send signal 10 to pid 6259
[27557.970835] send signal 12 to pid 6259
```

可以看到：驱动把这两个信号(10 和 12)，发送给了应用程序(PID=6259)。

应用程序的输出信息如下：

```
open dev success!
call ioctl. pid = 6259
signal_handler: signum = 12
signo = 12, code = 200, errno = 100
signal_handler: signum = 10
signo = 10, code = 200, errno = 100
```

可以看到：应用程序接收到信号 10 和 12，并且正确打印出信号中携带的一些信息!

------ End ------

文中的测试代码，已经放在网盘了。

在公众号【IOT物联网小镇】后台回复关键字：1205，即可获取下载地址。

谢谢！

推荐阅读

【1】《Linux 从头学》系列文章

【2】C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻

【3】原来gdb的底层调试原理这么简单

【4】内联汇编很可怕吗？看完这篇文章，终结它！

其他系列专辑：精选文章、应用程序设计、物联网、 C语言。

星标公众号，第一时间看文章！

C/C++、物联网、嵌入式、Lua语言

Linux 操作系统、应用程序开发设计

扫码关注公众号

道哥 个人微信

喜欢请分享，满意点个赞，最后点在看。