FPGA 嵌入式系统的 Linux 设备驱动程序开发①

姚 铭，黄林健

(厦门大学 信息科学与技术学院，厦门 361005)

摘 要：基于 FPGA 嵌入式系统，在 PowerPC 架构的 Linux2.6 操作系统环境下，对通用输入输出接口(GPIO)控制器的驱动，采用平台设备机制进行中断控制管理。通过该管理机制，将 GPIO 设备本身的资源注册进内核，由内核统一管理。在参照 Linux2.6 内核源码有关平台设备驱动的基础上，编写和测试了 GPIO 设备的驱动程序。该驱动程序已在 Xilinx 公司 FPGA 开发板 ML403 上验证，并且稳定运行。

关键字：PowerPC；平台设备；中断控制；设备驱动

Linux Device Drivers Development Based on FPGA Embedded System

YAO Ming, HUANG Lin-Jian

(School of Information Science & Technology, Xiamen University, Xiamen 361005, China)

Abstract: Under the PowerPC architecture Linux2.6 operating system based on FPGA embedded system, the platform devices mechanism is used to control the driver of GPIO controller by interrupt. With this mechanism, the GPIO registers its own resources into the kernel, and generally managed by the kernel. In reference to the Linux2.6 kernel source about platform device driver, compiling and testing of the GPIO device driver. The driver has been verificated in the Xilinx company's FPGA development board ML403, and operates stably.

Keywords: powerPC; platform device; interrupt control; device driver

从 Linux 2.6 起引入了平台设备机制，即 platform device driver 机制，Linux 中大部分设备驱动都可以使用这套机制[1]。和传统的 device driver 机制(通过 driver register 函数进行注册)相比，十分明显的优势在于 platform 机制将设备本身的资源注册进内核，由内核统一管理，在驱动程序中使用这些资源时通过 platform device 提供的标准接口进行申请并使用[2]。这样提高了驱动和资源管理的独立性，并且拥有较好的可移植性和安全性。文中讨论的 GPIO 设备具有双重身份：平台设备与混杂设备(miscdevice)。平台设备意味着 GPIO 控制器设备是属于平台的独立模块；混杂设备(即主设备号为 10)是一种特殊的字符型设备，描述了 GPIO 控制器的访问方式是顺序的[1]。

1 Linux中平台设备驱动开发流程

ML403 开发板采用 vertex-4 系列 FPGA，集成了 PowerPC405 硬核，带有内核管理单元（MMU），因此可以在该开发板上运行 Linux2.6 操作系统。在嵌入式 Linux2.6 操作系统中，通过 Platform 机制，对外设进行管理。开发设备驱动的流程如图 1 所示:

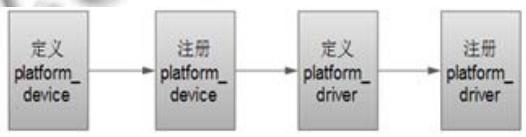

图 1 Platform 机制开发设备驱动流程图

1.1 定义 platform_device

在 Linux 2.6 内核中 platform 设备用结构体 platform_device 来描述，该结构体定义在 kernel\include\linux\platform_device.h 中：

```
struct platform_device {
const char * name;   //平台设备的设备名
```

① 收稿时间:2010-09-08;收到修改稿时间:2010-11-11

```
u32 id;                 //平台设备的设备 ID
struct device dev;      //设备结构体
u32 num_resources; //平台设备使用的各类资源数量
struct resource * resource;//资源
};
```

该结构一个重要的元素是 resource，它存入了最为重要的设备资源信息。在嵌入式开发工具 EDK 中生成 BSP(板级支持包)的时候有一个设备参数头文件 xparameter.h，里面定义了相关设备的设备数量、地址资源、中断资源和时钟资源等。在添加平台设备信息的时候需要用到该头文件中定义的地址信息和中断信息，Xilinx 公司的 Virtex-4 平台设备是在 kernel/arch/ppc/syslib/virtex_devices.c 中定义的，在编写驱动之前，需要在该文件中添加有关 GPIO 控制器的设备定义：

```
/*
* ML300/ML403 Gpio Device: shortcut macro for single instance
*/
#define XPAR_GPIO(num) { \
        .name = "xilinx_gpio", \
        .id = num, \
         .dev.platform_data=XPAR_GPIO_##num##_IS_DUAL, \
        .num_resources = 2, \
        .resource = (struct resource[]) { \
            { \
                .start=XPAR_GPIO_##num##_BASEADDR, \
                .end=XPAR_GPIO_##num##_HIGHADDR, \
                .flags = IORESOURCE_MEM, \
            }, \
            { \
                .start=XPAR_INTC_0_GPIO_##num##_VEC_ID, \
                .flags = IORESOURCE_IRQ, \
            }, \
    }, \
}
        /* GPIO instances */
#if defined(XPAR_GPIO_0_BASEADDR)
   XPAR_GPIO(0),
#endif
#if defined(XPAR_GPIO_1_BASEADDR)
   XPAR_GPIO(1),
#endif
#if defined(XPAR_GPIO_2_BASEADDR)
   XPAR_GPIO(2),
#endif
```

上述的代码定义了 GPIO 设备名称——xilinx_gpio，XPAR_GPIO 平台设备结构中 name 元素和设备驱动的 platform_driver 结构体中的 driver.name 必须是相同的。这是因为在平台设备驱动注册时会对所有已注册的 platform_device 中的 name 和当前注册的 platform_driver 的 driver.name 进行比较，使得 platfrom_device 和 platform_driver 建立关联，只有找到相同的名称的 platfomr_device 才能注册成功。在平台设备的描述中 GPIO 设备定义了 2 个资源，一个是 I/O 空间资源，描述了 GPIO 控制器设备所占用的总线地址范围，IORESOURCE_MEM 表示第 1 组描述的是内存类型的资源信息；另一个是中断资源，描述了设备的中断号，IORESOURCE_IRQ 表示第 2 组描述的是中断资源信息，设备驱动会根据类型来获取相应的资源信息。本文共用到三个 GPIO 设备 XPAR_GPIO(0)，XPAR_GPIO(1)，XPAR_GPIO(2)。

1.2 注册 platform_device

virtex_devices.c 中的 platform_device 是在系统启动时，使用 virtex_init(void)函数进行注册。

同时被注册还有很多 virtex 平台的设备，该函数是在系统初始化阶段调用，驱动注册时需要匹配内核中所有已注册的设备名，因此 platform_device 设备的注册过程必须在相应设备驱动加载之前被调用。

1.3 定义 platform_driver

与平台设备对应的平台设备驱动程序由 struct platform_driver 描述:

```
struct platform_driver {
int (*probe)(struct platform_device *);     //探测
int (*remove)(struct platform_device *);  //移除
void (*shutdown)(struct platform_device *);//关闭
int (*suspend)(struct platform_device *, pm_message_t state);     //挂起
```

```
int (*resume)(struct platform_device *);  //恢复
struct device_driver driver;
};
```

GPIO 的驱动程序中结构体 struct platform_driver 主要实现了 xgpio_driver 的探测和移除函数。代码如下：

```
static struct platform_driver xgpio_driver = {
    .probe          = xgpio_probe,
    .remove   = xgpio_remove,
    .driver         = {
        .name     = xilinx_gpio,
        .bus  = &platform_bus_type,
        .owner    = THIS_MODULE,
    }
```

1.4 注册 platform_driver

最后需要调用 platform_driver_register()函数注册平台设备驱动，在注册成功后会调用 platform_driver 结构元素 probe 函数指针，进入 probe 函数后，需要获取设备的资源信息。注册平台设备驱动的实现函数如下：

```
static int __init xgpio_init(void)
{
    return platform_driver_register (&xgpio_
driver);
}
```

2 GPIO控制器设备驱动

Linux 是保护模式的操作系统，内核和应用程序分别运行在完全分离的虚拟地址空间,用户空间的进程一般不能直接访问硬件。设备驱动充当了硬件和应用软件之间的纽带，它与底层硬件直接打交道，按照硬件设备的具体工作方式读写设备寄存器，完成设备的轮询、中断处理、DMA 通信，进行物理内存向虚拟内存的映射，最终使通信设备能收发数据，使显示设备能否显示文字和画面，使存储设备能够记录文件和数据[1]。

2.1 GPIO 控制器的平台设备驱动函数实现

使用 platform_driver_register(&xgpio_driver)注册 GPIO 设备驱动成功后，利用系统探测函数 probe()，获取设备需要的资源信息。在探测函数中，需要通过 platform_get_resource()函数分别获得 GPIO 内存和 IRQ 资源：

```
struct resource * platform_get_resource(struct
platform_device *dev, unsigned int type, unsigned int
num);
```

根据参数 type 所指定的类型，IORESOURCE_MEM 和 IORESOURCE_IRQ 来获取指定的资源。

驱动程序中相应代码为：

```
regs_res = platform_get_resource(pdev,
IORESOURCE_MEM, 0);
irq_res = platform_get_resource(pdev,
IORESOURCE_IRQ, 0);
```

在获取资源成功后，驱动程序会申请内核空间和 I/O 空间，将物理地址映射到虚拟地址以及申请中断等。在 Linux 内核空间申请内存的主要函数是 kmalloc()、kzalloc()。由这两个函数申请的内存位于物理内存映射区域，在物理上也是连续的。它们与真实的物理地址只有一个固定的偏移，存在较简单的转换关系。

驱动程序中与内存申请有关的程序代码：

```
xgpio_inst = kmalloc(sizeof(struct xgpio_instance),
GFP_KERNEL);
miscdev = kmalloc(sizeof(struct miscdevice),
GFP_KERNEL);
```

第一个参数是分配的空间大小，第二个标志表示是在内核空间的进程中申请内存。GFP_KERNEL 标志申请内存时，若暂时不能满足，则进程会睡眠引起阻塞。使用 kmalloc()、kzalloc()申请的内存要用 kfree()释放。

2.1.1 申请 I/O 内存空间和映射物理内存：

GPIO 设备控制器有一组寄存器用于读写设备和获取设备状态，即控制寄存器、状态寄存器和数据寄存器。这些寄存器位于 I/O 内存空间[3]。首先需要调用 request_mem_region()申请资源，接着将寄存器地址通过 ioremap()将物理地址映射到内核空间虚拟地址，之后才可以调用编程接口访问这些设备的寄存器。访问完成后用 iounmap()对申请的内核虚拟地址进行释放，并释放申请的 I/O 内存资源。GPIO 控制器驱动程序的相关代码如下：

```
/*申请 I/O 内存资源 */
request_mem_region(regs_res->start,  remap_size,
```

```
DRIVER_NAME);
    ioremap(regs_res->start, remap_size);
    /*映射物理地知道虚拟地址*/
```

2.1.2 申请中断：

request_irq(irq_res->start,xgpio_interrupt,0,"XGPIO", xgpio_inst)

irq_res->start 是要申请的硬件中断号；xgpio_interrupt 是向系统登记的中断处理函数，是一个回调函数，中断发生时，系统调用这个函数，dev_id 参数(即 xgpio_inst)将被传递。

2.1.3 释放虚拟地址和内存资源

```
static int xgpio_remove(struct platform_device *pdev) {  iounmap(xgpio_inst->base_address);
    release_mem_region(xgpio_inst->phys_addr, xgpio_inst->remap_size);
    kfree(xgpio_inst);
    return 0;  /*success*/}
```

2.2 **GPIO 控制器的字符型设备接口实现**

在 Linux 的文件操作系统调用中，字符型设备一般涉及到打开，读写和关闭文件等操作。在控制器驱动程序中要给内核提供 file_operations 结构，才能为设备驱动提供用户调用的接口，定义如下：

```
static struct file_operations xgpio_fops = {
.owner = THIS_MODULE,
.read = xgpio_read,
.open = xgpio_open,
.release = xgpio_release,};
```

当系统启动后，GPIO 控制器被初始化，申请资源和内核 I/O 内存空间。用户调用 open 函数打开 GPIO 设备时，系统调用了 xgpio_open()函数，主要完成使能中断等功能。在打开设备后，返回一个文件指针，可以用这个文件指针对设备进行一系列操作。当用户调用 read()函数对控制器进行读取的时候，系统调用了 xgpio_read()函数，读取 GPIO 设备数据寄存器的值。当用户调用 close()函数关闭 GPIO 设备时，系统调了用 xgpio_release()函数，禁止中断。

2.2.1 GPIO 控制器 open()函数的实现

在打开 GPIO 控制器后，依据 GPIO 数据文档，向 GPIO 全局中断使能寄存器 GIER 写入 0x80000000，向中断使能寄存器 IER 写入 0x00000003 来使能中断：xgpio_open()函数实现代码如下

```
static int xgpio_open(struct inode *inode, struct file *file)
{
  XIo_Out32((int) xgpioinst->v_addr + XGPIO_GIER_OFFSET, 0x80000000);  /* 全局中断使能 */
  XIo_Out32((int) xgpioinst->v_addr + XGPIO_IER_OFFSET, 0x00000003);  /* 使能 GPIO 中断 */
  return 0;
}
```

2.2.2 GPIO 控制器 read()函数的实现

当用户空间调用 read()函数的时，调用 put_user 函数实现内核空间数据到应用程序的传递，将内核空间传递给用户空间的数据。xgpio_read()函数实现代码如下：

```
static ssize_t xgpio_read(struct file *file, char *buf, size_t count, loff_t * ppos)
{
    if(put_user(gpio_value, (int*)buf))
     return - EFAULT;
    else
     return sizeof(unsigned int);
 }
```

2.2.3 GPIO 控制器 close()函数的实现

在关闭 GPIO 控制器后，向 GPIO 全局中断使能寄存器 GIER 写入 0x0，向中断使能寄存器 IER 写入 0x0 来禁止中断。xgpio_release()函数实现代码如下：

```
static int xgpio_release(struct inode *inode, struct file *file)
{
XIo_Out32((int) xgpioinst->v_addr + XGPIO_GIER_OFFSET, 0x0);  /* 禁止全局中断*/
XIo_Out32((int) xgpioinst->v_addr + XGPIO_IER_OFFSET, 0x0);  /* 禁止 GPIO 中断*/
return 0;
}
```

3 设备驱动添加到嵌入式Linux内核中

嵌入式 Linux 设备驱动程序编写完成后，需要将驱动程序加到内核中[4]，这要求修改嵌入式 Linux 的源

代码，然后重新编译内核。步骤如下：

3.1 将设备驱动文件拷贝到/linux/driver/char 目录下

3.2 在/linux/driver/char 目录下 Makefile 中增加如下代码

obj-$(CONFIG_ XMU_GPIO) +=xgpio；

在/linux/driver/char 目录下 Kconfig 中增加如下代码：

```
Config   XMU_GPIO
    tristate “XMU_GPIO”
    depends on XILINX_DRIVERS
    select XILINX_EDK
    help
    This option enables support for Xilinx GPIO.
```

3.3 重新编译内核，进入 Linux 目录，执行以下代码

#make menuconfig

在 Character Devices-->中找到＜＞XMU_GPIO 选中为加载模块的形式：＜*＞XMU_GPIO，然后保存退出。

#make

这样得到的内核包含了用户的设备驱动程序。

4 GPIO驱动程序的测试

在应用程序中利用函数 open()系统调用 xgpio_open()函数来使能 GPIO 中断，当中断发生时，执行中断处理程序；应用程序执行 read()函数时，系统调用了 xgpio_read()函数读取 GPIO 数据寄存器的值；当应用程序执行 close()函数时，系统调用 xgpio_release()函数，屏蔽 GPIO 中断。此时，驱动程序测试结束。

5 结语

Linux2.6 内核引入的平台设备机制，使得内核对设备的管理更加简便。本文介绍了基于 PowerPC 架构的嵌入式 Linux 平台设备驱动的一般设计方法。在基于 FPGA 的嵌入式系统中，外设通过 GPIO 的 IP 核与 CPU 的互连，因此，本文介绍的设备驱动程序的设计方法，具有的一定的通用性，对底层驱动程序开发人员有较好的参考价值。此外，在 Linux 系统中，字符设备和块设备都被映射到文件系统的文件和目录，很好地体现了“一切都是文件”的思想。所有的字符设备和块设备都被统一地呈现给用户，通过文件系统的调用接口 read()、write()等函数即可访问字符设备和块设备[1]。

参考文献

1 宋宝华.Linux 设备驱动开发详解.北京:人民邮电出版社, 2008.

2 Linux 系列教材编写组.Linux 操作系统分析与实践.北京:清华大学出版社, 2008.

3 周立功,陈明计,陈渝.ARM 嵌入式 Linux 系统构建与驱动程序开发范例.北京:北京航空航天大学出版社, 2006.

4 董志国,李式巨.嵌入式 Linux 设备驱动开发.计算机工程与设计, 2006,20(27):3737-3740.

（上接第 108 页）

本系统的车流量统计部分还可以根据不同的要求统计不同的时段和不同路况下的交通流量情况，例如当给定道路的方向后，就可以统计出在某一给定时段中向东、西、南、北不同方向行驶的车流量的变化规律，这样，能够给宏观的交通调控和调度一个合理的统计依据。这样，使得下一步让系统根据道路的车流量状况自动进行交通灯的调度成为可能。

参考文献

1 张超.视频监控中的图像匹配和运动目标检测[硕士学位论文].武汉:华中科技大学,2008.34－37.

2 贺春林.一种基于视频的车辆检测算法.计算机科学,2005,32(5):243－245.

3 张玲,易卫明,何伟,郭磊民,陈丽敏.一种基于视频的车辆检测新方法.信息与电子工程,2006,4(4):264－267.

4 王圣男,郁梅,蒋刚毅.智能交通系统中基于视频图像处理的车辆检测与跟踪方法综述.计算机应用研究,2005,9:11－12.

5 Bertozzi M, Broggi A, Castelluccio S. A real time oriented system for vehicle detection. Journal of System Architecture, 1997,43(3):317－325.