Lab 7：字符设备驱动程序

GPIO

GPIO 引脚是树莓派与外部世界之间的物理接口，树莓派可以通过这些引脚连接外部电路来控制和监视外部世界，我们称之为物理计算。在 Pi3 上有 40 个 GPIO 的引脚，它们提供不同的功能，在树莓派网站有详细的介绍。

对于 GPIO 的控制和访问，树莓派平台有丰富的库函数提供支持，Wiring Pi 和 RPi.GPIO 等都提供了很好的封装，同时 Linux 内核中也包含了 GPIO 的驱动程序，可以很方便的使用。

Wiring Pi

WiringPi 是应用于树莓派平台的 GPIO 控制库函数，它遵守 GUN Lv3 并且适用于 C/C++ 开发。

Blink 相当于 GPIO 接口的世界里的 “Hello World”，因为可以以最简单的程序和电路让你了解发生了什么。

编写一个测试程序：

```
#include <wiringPi.h>
int main (void)
{
  wiringPiSetup () ;
  pinMode (0, OUTPUT) ;
  for (;;)
  {
    digitalWrite (0, HIGH) ; delay (500) ;
    digitalWrite (0,  LOW) ; delay (500) ;
  }
  return 0 ;
}
```

编译并执行：

```
gcc -Wall -o blink blink.c -lwiringPi
sudo ./blink
```

连接电路，将一个 LED 灯与 GPIO 引脚相连并接地，如果一切正常，就可以看到 LED 每秒闪烁一次。

MAX7219

MAX7219 是一种集成化的串行输入输出共阴极显示驱动器，它连接微处理器与 8 位数字的 7 段数字 LED 显示，也可以连接条线图显示器或者64个独立的LED。

寻址

数据寄存器由一个在片上的 8*8 的双向 SRAM 来实现。它们可以直接寻址，每个数据都可以独立的修改或保存。控制寄存器包括编码模式、显示亮度、扫描限制、关闭模式以及显示检测五个寄存器。

串行地址格式

DIN 引脚串行输入数据，CLK 上升沿控制每一位的写入。

```
void Write_Max7219_byte(uchar DATA) {
    for (int i = 0; i < 8 ; i++) {
        digitalWrite(Max7219_pinCLK, LOW);
        digitalWrite (Max7219_pinDIN, DATA & 0x80) ;
        DATA = DATA << 1;
        digitalWrite(Max7219_pinCLK, HIGH);
    }
}
```

每次输入的数据分为 8 位选址和 8 位数据，16 位数据在 CS 上升沿被写入数据寄存器或控制寄存器。

```
//功能：向 MAX7219 写入数据
//参数：地址，数据
void Write_Max7219(uchar address, uchar dat) {
    digitalWrite (Max7219_pinCS, LOW) ;
    Write_Max7219_byte(address);           //写入地址
    Write_Max7219_byte(dat);               //写入数据
    digitalWrite (Max7219_pinCS, HIGH) ;
}
```

初始化

程序开始，首先写控制寄存器，设置一些必要的参数：

```
void Init_MAX7219(void) {
    Write_Max7219(0x09, 0x00);       //译码方式：BCD码
    Write_Max7219(0x0a, 0x03);       //亮度
    Write_Max7219(0x0b, 0x07);       //扫描界限；8个数码管显示
    Write_Max7219(0x0c, 0x01);       //掉电模式：0，普通模式：1
    Write_Max7219(0x0f, 0x00);       //显示测试：1；测试结束，正常显示：0
}
```

显示字库

使用 8*8 点阵字模，循环显示 0-9/a-z：

```
int  main() {
    // 配置 GPIO
    wiringPiSetup () ;
    pinMode (Max7219_pinCLK, OUTPUT) ;
    pinMode (Max7219_pinCS, OUTPUT) ;
    pinMode (Max7219_pinDIN, OUTPUT) ;

    // 配置 MAX7219
    Init_MAX7219();

    // 循环显示字符
    while (1) {
        for (int j = 0; j < 36; j++) {
            for (int i = 0; i < 8; i++)
                Write_Max7219(i + 1, displ[j][i]);
            delay(1000);
        }
    }
    return 0;
}
```

GPIO 驱动

Linux 内核中的驱动程序实现了一套函数，内核中的其他部分可以利用它们来建立使用 GPIO 某种具体的硬件设备的驱动程序。

引入头文件:

```
#include <linux/gpio.h>
```

每个 GPIO 引脚被赋予唯一的正整数值，可以测试某个引脚是否可用：

```
int gpio_is_valid(int number);
```

如果测试某个引脚不在可用状态，首先要分配这个引脚：

```
int gpio_request(unsigned gpio, const char *label);
```

之后要规定引脚的方向，即输入还是输出：

```
/* set as input or output, returning 0 or negative errno */
int gpio_direction_input(unsigned gpio);
int gpio_direction_output(unsigned gpio, int value);
```

读取或设置引脚有两种方式，其中一种是利用自旋锁的方式，这种方式不会进入休眠状态：

```
/* GPIO INPUT:  return zero or nonzero */
int gpio_get_value(unsigned gpio);

/* GPIO OUTPUT */
void gpio_set_value(unsigned gpio, int value);
```

或者另一种允许睡眠的方式：

```
/* GPIO INPUT:  return zero or nonzero, might sleep */
int gpio_get_value_cansleep(unsigned gpio);

/* GPIO OUTPUT, might sleep */
void gpio_set_value_cansleep(unsigned gpio, int value);
```

在某个 GPIO 使用完毕后，可以释放之前请求的引脚：

```
/* release previously-claimed GPIO */
void gpio_free(unsigned gpio);
```

设备驱动

为系统添加一个设备驱动程序，直接访问 GPIO 控制寄存器，能将写入设备文件中的字符在矩阵上显示出来。

我们以内核模块的方式将驱动程序插入到内核，设备相当于 `/dev/` 文件夹下的一个文件，向该文件中写入的内容会自动传送给设备驱动程序进行处理，设备驱动程序负责控制外部设备并交换数据。

由于内核态只能调用内核中的函数，正好 Linux 内核中就自带了 GPIO 驱动函数，可以很方便地使用。

模块初始化

在内核模块加载回调函数中，需要进行设备的注册和 GPIO 端口的初始化：

```
static int __init hello_init(void) {
    misc_register(&MAX7219_misc_device);
    gpio_request(Max7219_pinDIN, "sysfs");
    gpio_direction_output(Max7219_pinDIN, 0);
    gpio_request(Max7219_pinCS, "sysfs");
    gpio_direction_output(Max7219_pinCS, 1);
    gpio_request(Max7219_pinCLK, "sysfs");
    gpio_direction_output(Max7219_pinCLK, 0);
    MAX7219_init();
    printk("MAX7219 device has been registed.\n");
    return 0;
}
```

注册设备

在 Linux 驱动中把无法归类的五花八门的设备定义为杂项设备，杂项设备是在嵌入式系统中用得比较多。

杂项设备结构体声明在 `include/linux/miscdevice.h` 中：

```
struct miscdevice  {
    int minor;
    const char *name;
    const struct file_operations *fops;
    struct list_head list;
    struct device *parent;
    struct device *this_device;
    const struct attribute_group **groups;
    const char *nodename;
    umode_t mode;
};
```

定义一个杂项设备结构体，设置设备名称为 MAX7219，其中 file_operations 结构表示注册的文件操作接口。

```
static struct miscdevice MAX7219_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "MAX7219",
    .fops = &MAX7219_fops
};
```

向设备输入时会调用 `const struct file_operations *fops;` 结构中注册的文件操作接口，将 write 操作注册为新的功能函数：

```
static struct file_operations MAX7219_fops = {
    .owner = THIS_MODULE,
    .write = MAX7219_write,
    .llseek = noop_llseek
};
```

`int misc_register(struct miscdevice *misc)` 函数注册一个设备，传入设备相关信息的结构体:

Write 操作

每次向设备文件的输入会调用文件操作结构中的 write 函数，因此我们覆盖这个函数，设置环形缓冲区，在该函数中将每次输入的字符存入缓冲区，并调用定时显示函数，将缓冲区中的字符输出到 LED 显示：

```
static int MAX7219_write(struct file *file, const char __user *buffer,
size_t count, loff_t *ppos) {
    int i;
    for (i = 0; i < count; i++) {
        ptr_inc(&tail);
        if (tail == head)
            ptr_inc(&head);
        queue[tail] = buffer[i];
    }
    display();
    return count;
}
```

显示字符

字符显示的方式与之前一样，根据 MAX7219 的规则操纵 GPIO 端口，此时应该使用内核中的 GPIO 驱动函数：

```
void Write_Max7219_byte(uchar DATA) {
    uchar i;
    for (i = 0; i < 8 ; i++) {
        gpio_set_value(Max7219_pinCLK, LOW);
        gpio_set_value (Max7219_pinDIN, DATA & 0x80) ;
        DATA = DATA << 1;
        gpio_set_value(Max7219_pinCLK, HIGH);
    }
}
void Write_Max7219(uchar address, uchar dat) {
    gpio_set_value(Max7219_pinCS, LOW);
    Write_Max7219_byte(address);
    Write_Max7219_byte(dat);
    gpio_set_value(Max7219_pinCS, HIGH);
}

void Render_MAX7219(unsigned char* matrix) {
    int i;
    for (i = 0; i < 8; i++) {
        Write_Max7219(i + 1, matrix[i]);
    }
}
```

加载模块

编写 Makefile，利用 `make` 命令编译内核模块，`insmod` 命令将模块插入内核，可以看到模块加载时的输出。

测试

向 `/dev/MAX7219` 文件中写入数据并保存，LED 会自动显示输入的字符串，至此设备驱动添加成功。