全志科技
Allwinner Technology

V40 项目

SPI 接口使用说明书 V1.0

文档履历

版本号	日期	制/修订人	内容描述
v1.0	2016-06-07		正式版本

目　　录

1. 概述

1.1. 编写目的

介绍 V40 sdk 配套的 Linux 内核中 SPI 子系统的接口及使用方法，为 SPI 设备驱动开发提供参考。

1.2. 适用范围

适用于 V40 sdk 配套的 Linux 3.10 内核。

1.3. 相关人员

SPI 设备驱动、SPI 总线驱动的开发/维护人员。

2. 模块介绍

2.1. 模块功能介绍

Linux 中 SPI 体系结构图 2.1 所示，图中用分割线分成了三个层次：

1. 用户空间，包括所有使用 SPI 设备的应用程序；
2. 内核，也就是驱动部分；
3. 硬件，指实际物理设备，包括了 SPI 控制器和 SPI 外设。

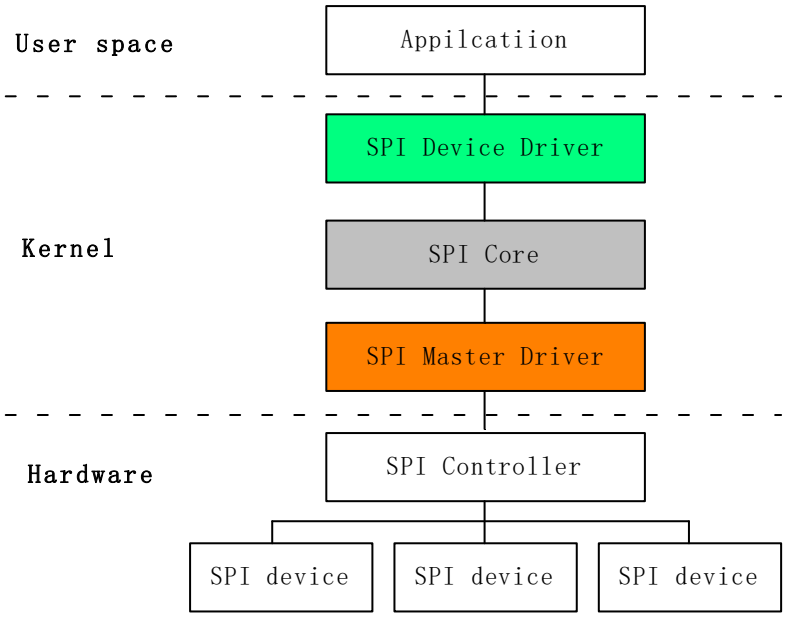

图 2.1 Linux SPI 体系结构图

其中，Linux 内核中的 SPI 驱动程序仅支持主设备，逻辑上又可以分为 3 个部分：

1. SPI 核心（SPI Core）：实现对 SPI 总线驱动及 SPI 设备驱动的管理；
2. SPI 总线驱动（SPI Master Driver）：针对不同类型的 SPI 控制器，实现对 SPI 总线访问的具体方法；
3. SPI 设备驱动（SPI Device Driver）：针对特定的 SPI 设备，实现具体的功能，包括 read，write 以及 ioctl 等对用户层操作的接口。

SPI 总线驱动主要实现了适用于特定 SPI 控制器的总线读写方法，并注册到 Linux 内核的 SPI 架构，SPI 外设就可以通过 SPI 架构完成设备和总线的适配。但是总线驱动本身并不会进行任何的通讯，它只是提供通讯的实现，等待设备驱动来调用其函数。

SPI Core 的管理正好屏蔽了 SPI 总线驱动的差异，使得 SPI 设备驱动可以忽略各种总线控制器的不同，不用考虑其如何与硬件设备通讯的细节。

2.2. 相关术语介绍

术语	解释说明
Sunxi	指 Allwinner 的一系列 SOC 硬件平台。
SPI	Serial Peripheral Interface，同步串行外设接口
SPI Master	SPI 主设备
SPI Device	指 SPI 外部设备

2.3. 模块配置介绍

2.3.1. Device Tree 配置说明

在 Device Tree 中配置参数相似，如下：

```
spi0: spi@01c05000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "allwinner,sun8i-spi";
        device_type = "spi0";
        reg = <0x0 0x01c05000 0x0 0x1000>;
        interrupts = <GIC_SPI 10 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&clk_pll_periph0>, <&clk_spi0>;
        clock-frequency = <100000000>;
        pinctrl-names = "default", "sleep";
        pinctrl-0 = <&spi0_pins_a &spi0_pins_b &spi0_pins_c>;
        pinctrl-1 = <&spi0_pins_d>;
        spi0_cs_number = <2>;
        spi0_cs_bitmap = <3>;
        status = "disabled";
};
```

其中：

1. compatible：表征具体的设备，用于驱动和设备的绑定；
2. reg：设备使用的地址；
3. interrupts：设备使用的中断；
4. clocks：设备使用的时钟；
5. pinctrl-0：设备使用的 PIN 脚；
6. cs-number：SPI 控制器的片选线数；
7. cs-bitmap：SPI 控制器的片选线的掩码；
8. status：是否使用该节点。

为了在 SPI 总线驱动代码中区分每一个 SPI 控制器，需要在 Device Tree 中的 aliases 节点中为每一个 SPI 节点指定别名：

```
aliases {
        spi0 = &spi0;;
        ...
};
```

别名形式为字符串“spi”加连续编号的数字，在 SPI 总线驱动程序中可以通过 of_alias_get_id()函数获取对应 SPI 控制器的数字编号，从而区别每一个 SPI 控制器。

对于 SPI 设备，可以把设备节点填充作为 Device Tree 中相应 SPI 控制器的子节点。SPI 控制器驱动的 probe 函数透过 spi_register_master()注册 master 的时候，会自动展开作为其子节点的 SPI 设备。

```
spi0: spi@01c05000{
    #address-cells = <1>;
    #size-cells = <0>;
    ...
    nor-flash@0 {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "atmel,at25df641";
        reg = <0>;          /* Chip select 0 */
        spi-max-frequency = <50000000>;
        mode = <0>;

        partition@0 {
            label = "NorFlash part0";
            reg = <0x00000000 0x0002000>;
        };
    };
};
```

其中:

1. compatible：表征具体的设备，用于驱动和设备的绑定；
2. reg：SPI 片选号，取决于硬件的 CS 连线；
3. spi-max-frequency：最大传输速度频率；
4. mode：SPI MOSI 的 GPIO 配置，取值范围为 0 到 3；
5. partition：分区信息。

2.3.2. menuconfig 配置说明

在命令行中进入内核根目录，执行 make ARCH=arm menuconfig 进入配置主界面，并按以下步骤操作：

首先，选择 Device Drivers 选项进入下一级配置，如下图所示：

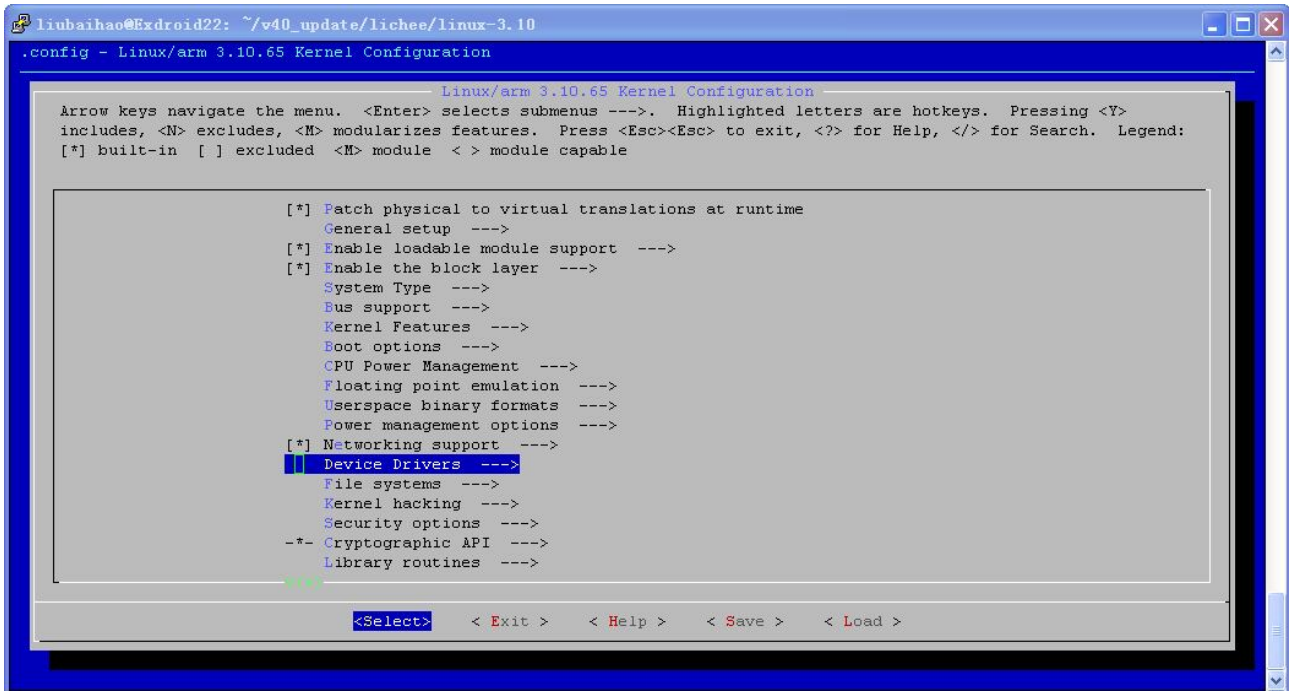

图 2.2 Device Drivers 配置选项

然后，选择 SPI support 选项，进入下一级配置，如下图所示：

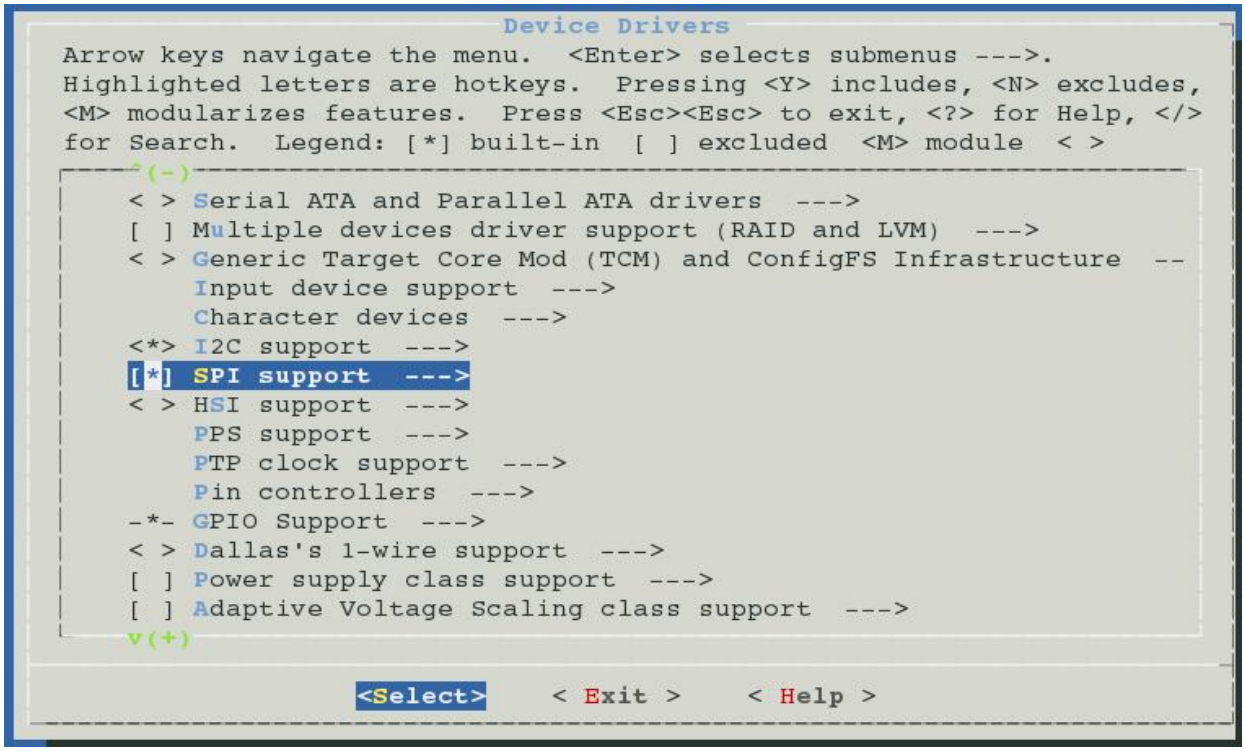

图 2.3 SPI support 配置选项

选择 SUNXI SPI Controller 选项，可选择直接编译进内核，也可编译成模块。如下图：

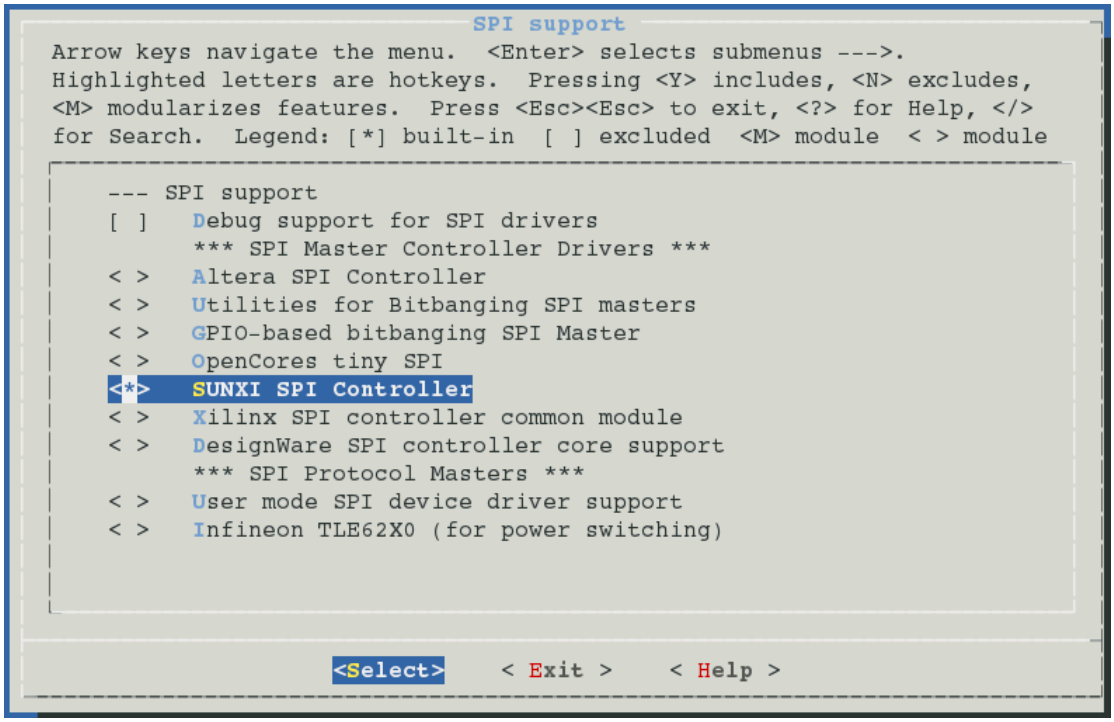

图 2.4 SUNXI SPI Controller 配置选项

如果当前的配置是给 FPGA 使用，因为 FPGA 板上只有一个 SPI 控制器，还需要多做一项配置，配置 SPI 外设要使用哪个 SPI 通道。如下图：

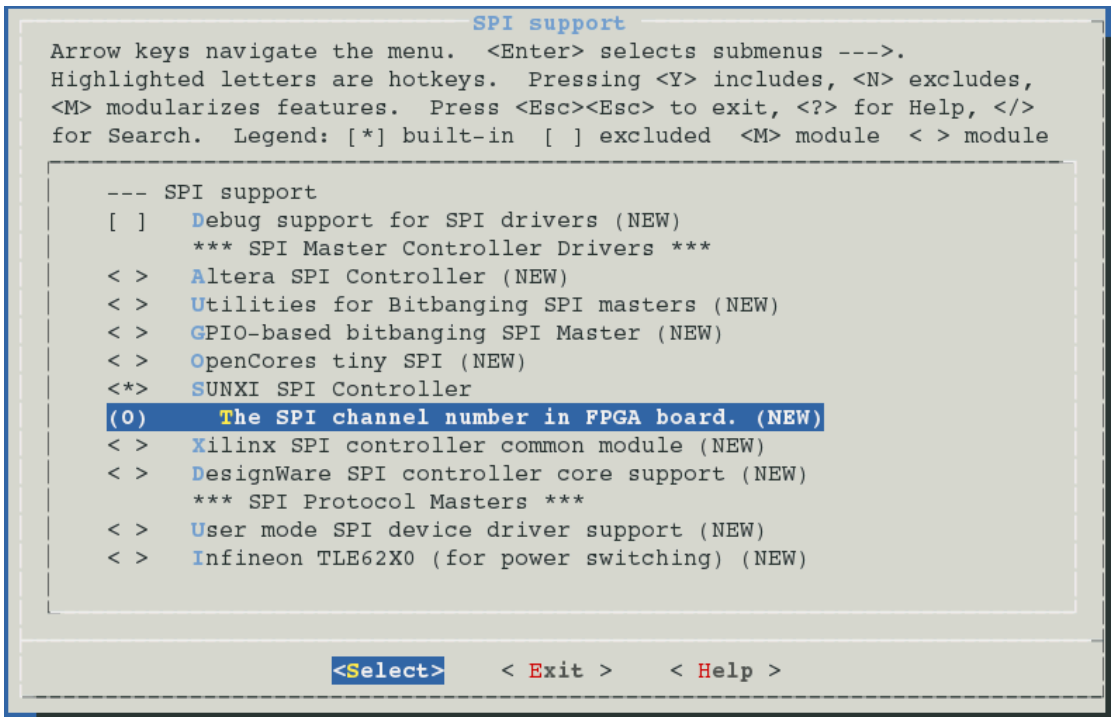

图 2.5 为 FGPA 板选择一个 SPI 通道

2.4. 源码结构介绍

SPI 总线驱动的源代码位于内核在 drivers/spi 目录下：

drivers/spi/

├── spi-sunxi.c // Sunxi 平台的 SPI 控制器驱动代码
├── spi-sunxi.h // 为 Sunxi 平台的 SPI 控制器驱动定义了一些宏、数据结构

3. 接口描述

SPI 总线驱动对外没有直接调用的接口，如图 2.1 所示，SPI 设备驱动都是通过 SPI Core 来使用 SPI 总线的读写功能。

3.1. 内部接口

总线驱动除了标准的 probe、remove、resume、suspend 接口外，和 SPI 控制器密切相关的内部接口有以下几个。

3.1.1. sunxi_spi_transfer()

【函数原型】：static int sunxi_spi_transfer(struct spi_device *spi, struct spi_message *msg)
【功能描述】：由 spi_asyn()调用此接口，启动一次 SPI 数据传输。
【参数说明】：spi，指向当前的 SPI 设备
msg，指向待处理的 SPI 消息
【返 回 值】：0，成功。

3.1.2. sunxi_spi_work()

【函数原型】：static void sunxi_spi_work(struct work_struct *work)
【功能描述】：SPI 数据传输的工作队列入口函数。完成所有 SPI 消息队列中的数据传输。
【参数说明】：adap，指向工作队列的控制结构信息
【返 回 值】：无

3.1.3. sunxi_spi_handler()

【函数原型】：static irqreturn_t sunxi_spi_handler(int irq, void *dev_id)
【功能描述】：处理 SPI 控制器产生的中断信号
【参数说明】：irq，中断号
dev_id，自定义的回调参数，在此接口中该参数类型是 struct sunxi_spi *
【返 回 值】：IRQ_HANDLED，中断已经出来；IRQ_NONE，不是此设备的中断。

3.1.4. sysfs 调试接口

SPI 总线驱动通过 sysfs 节点提供了几个在线调试接口：

1. /sys/devices/soc.2/1c68000.spi/info

此节点文件可以打印出当前 SPI 通道的一些硬件资源信息。操作效果图：

```
/ # cat /sys/devices/soc.2/1c68000.spi/info
pdev->id   = 0
pdev->name = 1c68000.spi
pdev->num_resources = 2
pdev->resource.mem = [0x0000000001c68000, 0x0000000001c68ff
pdev->resource.irq = 97
pdev->dev.platform_data.cs_bitmap = 1
pdev->dev.platform_data.cs_num    = 1
pdev->dev.platform_data.regulator = 0x          (null)
pdev->dev.platform_data.regulator_id =
```

2. /sys/devices/soc.2/1c68000.spi/status

此节点文件可以打印出当前 SPI 通道的一些运行状态信息，包括控制器的各寄存器值。操作效果图：

```
/ # cat /sys/devices/soc.2/1c68000.spi/status
master->bus_num = 0
master->num_chipselect = 1
master->dma_alignment  = 0
master->mode_bits = 15
master->flags = 0x0, ->bus_lock_flag = 0x0
master->busy = 0, ->running = 0, ->rt = 0
sspi->mode_type = 5 [Null]
sspi->irq = 97 [spi0]
sspi->cs_bitmap = 1
sspi->dma_tx.dir = 0 [DMA NULL]
sspi->dma_rx.dir = 0 [DMA NULL]
sspi->busy = 1 [Free]
sspi->result = 0 [Success]
sspi->base_addr = 0xffffff8000014000, the SPI control register:
[VER] 0x00 = 0x00090000, [GCR] 0x04 = 0x00000083, [TCR] 0x08 = 0x000001c4
[ICR] 0x10 = 0x00000000, [ISR] 0x14 = 0x00000032, [FCR] 0x18 = 0x00400001
[FSR] 0x1c = 0x00000000, [WCR] 0x20 = 0x00000000, [CCR] 0x24 = 0x00001000
[BCR] 0x30 = 0x00000000, [TCR] 0x34 = 0x00000000, [BCC] 0x38 = 0x00000000
[DMA] 0x88 = 0x000000a5, [TXR] 0x200 = 0x00000000, [RXD] 0x300 = 0x00000000
```

4. demo

4.1. drivers\spi\spi-tle62x0.c

此源文件为内核中自带的一个 SPI 设备驱动代码，其中通过 sysfs 方式提供读写操作，总体程序流程实现比较简单。

```
…
struct tle62x0_state {
	struct spi_device	*us;
	struct mutex		lock;
	unsigned int		nr_gpio;
	unsigned int		gpio_state;

	unsigned char		tx_buff[4];
	unsigned char		rx_buff[4];
};
…
static inline int tle62x0_write(struct tle62x0_state *st)
{
	unsigned char *buff = st->tx_buff;
	unsigned int gpio_state = st->gpio_state;

	buff[0] = CMD_SET;

	if (st->nr_gpio == 16) {
		buff[1] = gpio_state >> 8;
		buff[2] = gpio_state;
	} else {
		buff[1] = gpio_state;
	}

	dev_dbg(&st->us->dev, "buff %02x,%02x,%02x\n",
		buff[0], buff[1], buff[2]);

	return spi_write(st->us, buff, (st->nr_gpio == 16) ? 3 : 2);
}

static inline int tle62x0_read(struct tle62x0_state *st)
{
	unsigned char *txbuff = st->tx_buff;
	struct spi_transfer xfer = {
		.tx_buf		= txbuff,
		.rx_buf		= st->rx_buff,
		.len		= (st->nr_gpio * 2) / 8,
```

```
    };
    struct spi_message msg;

    txbuff[0] = CMD_READ;
    txbuff[1] = 0x00;
    txbuff[2] = 0x00;
    txbuff[3] = 0x00;

    spi_message_init(&msg);
    spi_message_add_tail(&xfer, &msg);

    return spi_sync(st->us, &msg);
}

static struct device_attribute *gpio_attrs[] = { …
};

static int __devinit tle62x0_probe(struct spi_device *spi)
{
    …
    mutex_init(&st->lock);

    ret = device_create_file(&spi->dev, &dev_attr_status_show);
    if (ret) {
        dev_err(&spi->dev, "cannot create status attribute\n");
        goto err_status;
    }

    for (ptr = 0; ptr < pdata->gpio_count; ptr++) {
        ret = device_create_file(&spi->dev, gpio_attrs[ptr]);
        if (ret) {
            dev_err(&spi->dev, "cannot create gpio attribute\n");
            goto err_gpios;
        }
    }
    …
}

static int __devexit tle62x0_remove(struct spi_device *spi)
{
    …
    return 0;
}

static struct spi_driver tle62x0_driver = {
```

```
	.driver = {
		.name	= "tle62x0",
		.owner	= THIS_MODULE,
	},
	.probe		= tle62x0_probe,
	.remove		= __devexit_p(tle62x0_remove),
};

static __init int tle62x0_init(void)
{
	return spi_register_driver(&tle62x0_driver);
}

static __exit void tle62x0_exit(void)
{
	spi_unregister_driver(&tle62x0_driver);
}

module_init(tle62x0_init);
module_exit(tle62x0_exit);
```

Declaration