全志科技
Allwinner Technology

H3 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	2015-01-10		正式版本

目　　录

1. 概述

1.1. 编写目的

介绍 H3 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 H3 sdk 配套的 Linux 3.4 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。

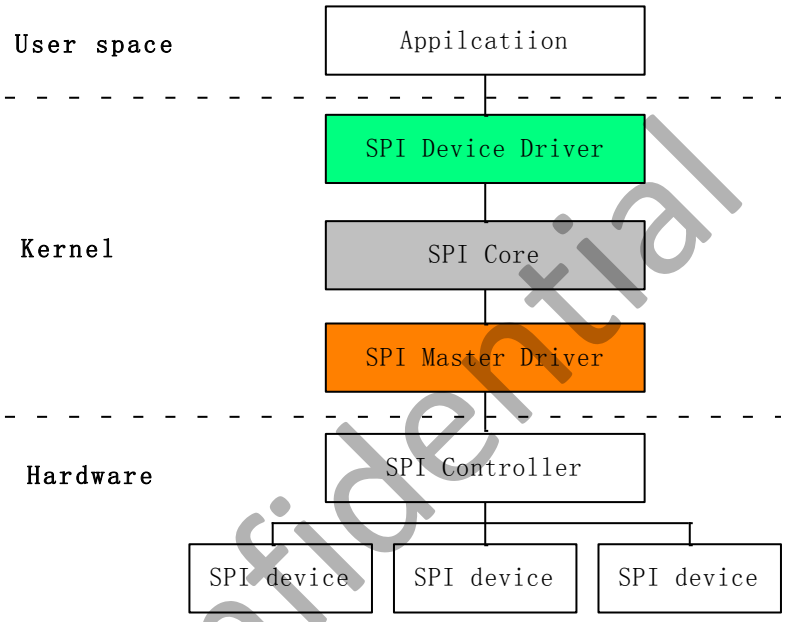

图 2.1 Linux SPI 体系结构图

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 sys_config.fex 配置说明

在不同的 Sunxi 硬件平台中，SPI 控制器的数目也不同，但对于每一个 SPI 控制器来说，在 sys_config.fex 中配置参数相似，如下：

```
[spi0]
spi_used            = 1
spi_cs_bitmap    = 1
spi_cs0             = port:PC27<3><1><default><default>
spi_sclk           = port:PC02<3><default><default><default>
spi_mosi           = port:PC00<3><default><default><default>
spi_miso           = port:PC01<3><default><default><default>
```

其中：

1. spi_used 置为 1 表示使能，0 表示不使能；
2. spi_cs_bitmap，由于 SPI 控制器支持多个 CS，这一个参数表示 CS 的掩码；
3. spi_cs0、spi_sclk、spi_mosi 和 spi_miso 用于配置相应的 GPIO。

对于 SPI 设备，还需要通过以下参数配置 SPI board info，这些信息会通过 SPI 子系统的接口 spi_register_board_info()在 SPI 总线驱动初始化前就注册到内核中。

```
[spi_devices]
spi_dev_num = 1

[spi_board0]
modalias            = "m25p32"
max_speed_hz    = 33000000
bus_num             = 0
chip_select         = 0
mode                = 0
```

其中:

1. spi_dev_num 表示 SPI 设备的数目，决定了下面有几个[spi_boardx]；

2. modalias，SPI 设备的名字，在做总线适配时会用到；
3. max_speed_hz，最大传输速度，单位是 Hz；
4. bus_num，SPI 控制器的序号，从 0 开始编号；
5. chip_select，理论上可以选 0，1，2，3，取决于硬件的 CS 连线；
6. mode，即 spi_board_info 结构中的 mode 成员，其定义同 spi_device 中的 mode：

```
#define   SPI_CPHA    0x01              /* clock phase */
#define   SPI_CPOL    0x02              /* clock polarity */
#define   SPI_MODE_0      (0|0)         /* (original MicroWire) */
#define   SPI_MODE_1      (0|SPI_CPHA)
#define   SPI_MODE_2      (SPI_CPOL|0)
#define   SPI_MODE_3      (SPI_CPOL|SPI_CPHA)
#define   SPI_CS_HIGH     0x04              /* chipselect active high? */
#define   SPI_LSB_FIRST   0x08              /* per-word bits-on-wire */
#define   SPI_3WIRE   0x10              /* SI/SO signals shared */
#define   SPI_LOOP    0x20              /* loopback mode */
#define   SPI_NO_CS   0x40              /* 1 dev/bus, no chipselect */
#define   SPI_READY 0x80                /* slave pulls low to pause */
```

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：

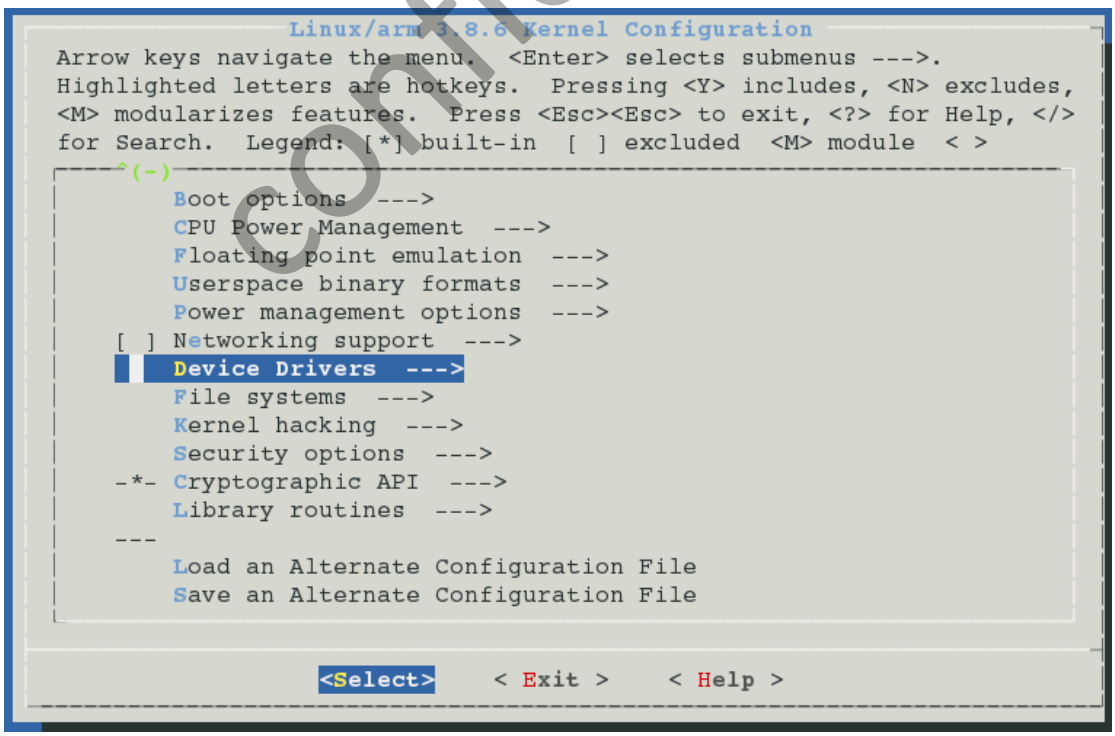

图 2.2 Device Drivers 选项配置

然后，选择 SPI support 选项，进入下一级配置，如下图所示：

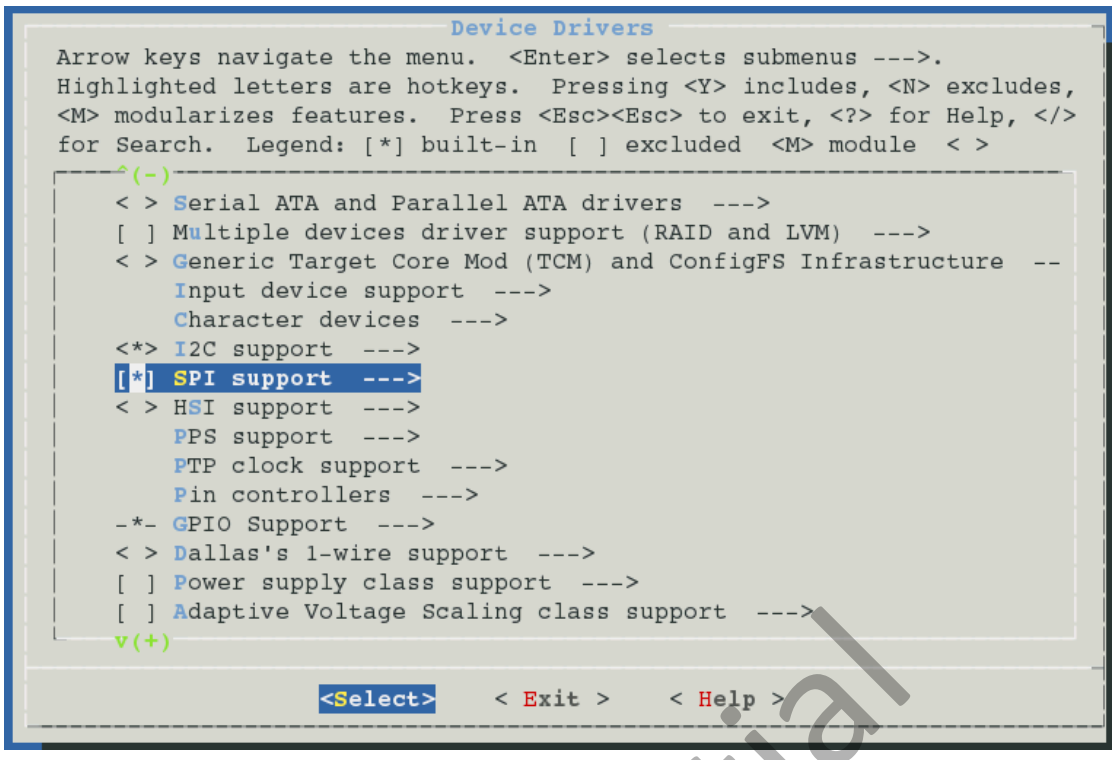

图 2.3 SPI support 选项配置

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：

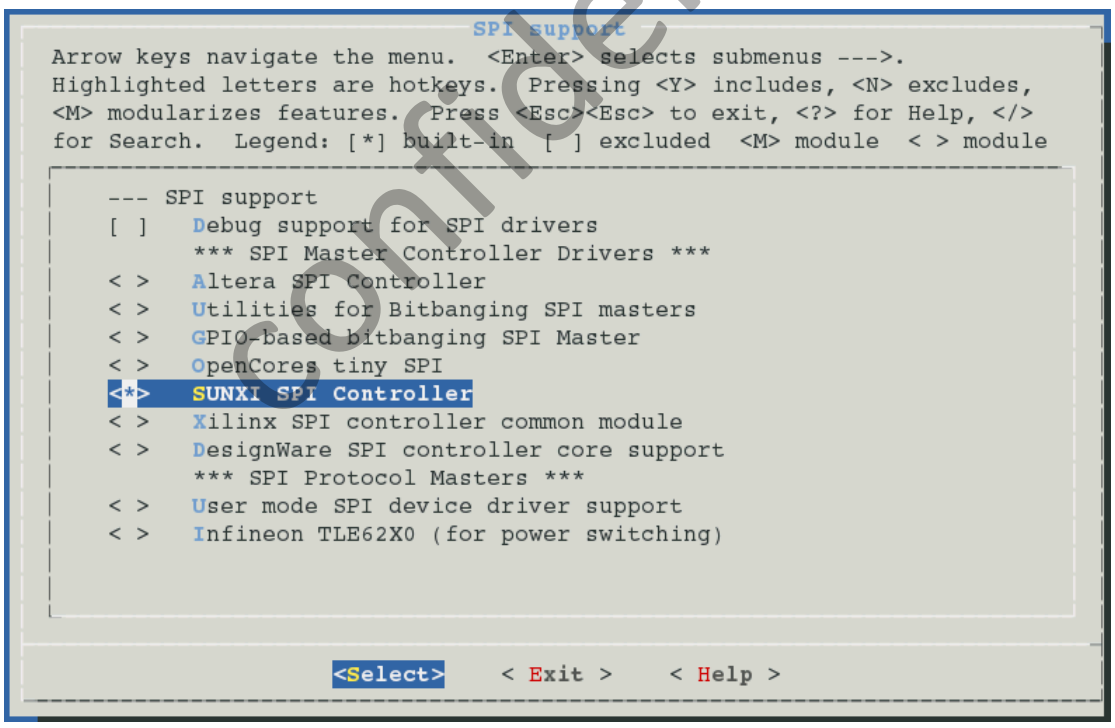

图 2.4 SUNXI SPI Controller 选项配置

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

3.1. 设备注册接口

声明在 include\linux\spi\spi.h。

3.1.1. spi_register_driver()

【函数原型】：int spi_register_driver(struct spi_driver *sdrv)

【功能描述】：注册一个 SPI 设备驱动。

【参数说明】：sdrv，spi_driver 类型的指针，其中包含了 SPI 设备的名称、probe 等接口信息

【返 回 值】：0，成功；其他值，失败。

其中，结构 spi_driver 的定义如下：

```
struct spi_device_id {
	char name[SPI_NAME_SIZE];
	kernel_ulong_t driver_data	/* Data private to the driver */
			__attribute__((aligned(sizeof(kernel_ulong_t))));
};
struct spi_driver {
	const struct spi_device_id *id_table;
	int		(*probe)(struct spi_device *spi);
	int		(*remove)(struct spi_device *spi);
	void		(*shutdown)(struct spi_device *spi);
	int		(*suspend)(struct spi_device *spi, pm_message_t mesg);
	int		(*resume)(struct spi_device *spi);
	struct device_driver	driver;
};
```

SPI 设备驱动可能支持多种型号的设备，可以在.id_table 中给出所有支持的设备信息。

3.1.2. spi_unregister_driver()

【函数原型】：void spi_unregister_driver(struct spi_driver *sdrv)

【功能描述】：注销一个 SPI 设备驱动。

【参数说明】：sdrv，spi_driver 类型的指针，其中包含了 SPI 设备的名称、probe 等接口信息

【返 回 值】：无

spi.h 中还给出了快速注册的 SPI 设备驱动的宏：module_spi_driver()，定义如下：

```
#define module_spi_driver(__spi_driver) \
	module_driver(__spi_driver, spi_register_driver, \
			spi_unregister_driver)
```

3.2. 数据传输接口

SPI 设备驱动使用"struct spi_message"向 SPI 总线请求读写 I/O。

一个 spi_message 其中包含了一个操作序列，每一个操作称作 spi_transfer，这样方便 SPI 总线驱动中串行的执行一个个原子的序列。

spi_message 和 spi_transfer 的定义也在 spi.h 中：

```
struct spi_transfer {
        const void      *tx_buf;
        void            *rx_buf;
        unsigned len;

        dma_addr_t      tx_dma;
        dma_addr_t      rx_dma;

        unsigned cs_change:1;
        u8      bits_per_word;
        u16     delay_usecs;
        u32     speed_hz;

        struct list_head transfer_list;
};

struct spi_message {
        struct list_head    transfers;
        struct spi_device   *spi;
        unsigned        is_dma_mapped:1;
        void            (*complete)(void *context);
        void            *context;
        unsigned        actual_length;
        int             status;
        struct list_head    queue;
        void            *state;
};
```


3.2.1. spi_message_init()

【函数原型】：void spi_message_init(struct spi_message *m)

【功能描述】：初始化一个 SPI message 结构，主要是清零和初始化 transfer 队列。

【参数说明】：m，spi_message 类型的指针

【返 回 值】：无

3.2.2. spi_message_add_tail()

【函数原型】：void spi_message_add_tail(struct spi_transfer *t, struct spi_message *m)
【功能描述】：向 SPI message 中添加一个 transfer。
【参数说明】：t，指向待添加到 SPI transfer 结构
m，spi_message 类型的指针
【返 回 值】：无

3.2.3. spi_sync()

【函数原型】：int spi_sync(struct spi_device *spi, struct spi_message *message)
【功能描述】：启动、并等待 SPI 总线处理完指定的 SPI message。
【参数说明】：spi，指向当前的 SPI 设备
m，spi_message 类型的指针，其中有待处理的 SPI transfer 队列
【返 回 值】：0，成功；<0，失败

3.2.4. spi_read()

【函数原型】：int spi_read(struct spi_device *spi, void *buf, size_t len)
【功能描述】：从 SPI 总线读取一段数据，内部是通过 SPI message 实现。
【参数说明】：spi，指向当前的 SPI 设备
buf，用于保存读取到的数据缓存
len，buf 的长度
【返 回 值】：0，成功；<0，失败

3.2.5. spi_write()

【函数原型】：int spi_write(struct spi_device *spi, const void *buf, size_t len)
【功能描述】：向 SPI 总线写入一段数据，内部也是通过 SPI message 实现。
【参数说明】：spi，指向当前的 SPI 设备
buf，要写入的数据
len，要写入的数据长度
【返 回 值】：0，成功；<0，失败

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