

spi 接口使用说明书

confidential

1.0
2017.06.10

文档履历

版本号	日期	制/修订人	内容描述
1.0	2017.06.10	AWA1053	

confidential

目录

1. 概述	1
1.1 编写目的	1
1.2 适用范围	1
1.3 相关人员	1
2. 模块介绍	2
2.1 模块功能介绍	2
2.2 相关术语介绍	3
2.3 模块配置介绍	3
2.3.1 sys_config.fex 配置说明	3
2.3.2 menuconfig 配置说明	4
2.4 源码结构介绍	6
3. 接口描述	7
3.1 设备注册接口	7
3.1.1 spi_register_driver()	7
3.1.2 spi_unregister_driver()	8
3.2 数据传输接口	8
3.2.1 spi_message_init()	9
3.2.2 spi_message_add_tail()	9
3.2.3 spi_sync()	10
3.2.4 spi_read()	10

3.2.5 spi_write()	10
4. demo	11
4.1 driversspispi-tle62x0.c	11
4.2 driversmtddevicem25p80.c	14
5. Declaration	15

Confidential

1. 概述

1.1 编写目的

介绍一下 H6 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。

主要包括：

1. 模块介绍；
2. 源码结构；
3. API 接口描述。

1.2 适用范围

适用于 H6 sdk 配套的 Linux 3.10 内核。

1.3 相关人员

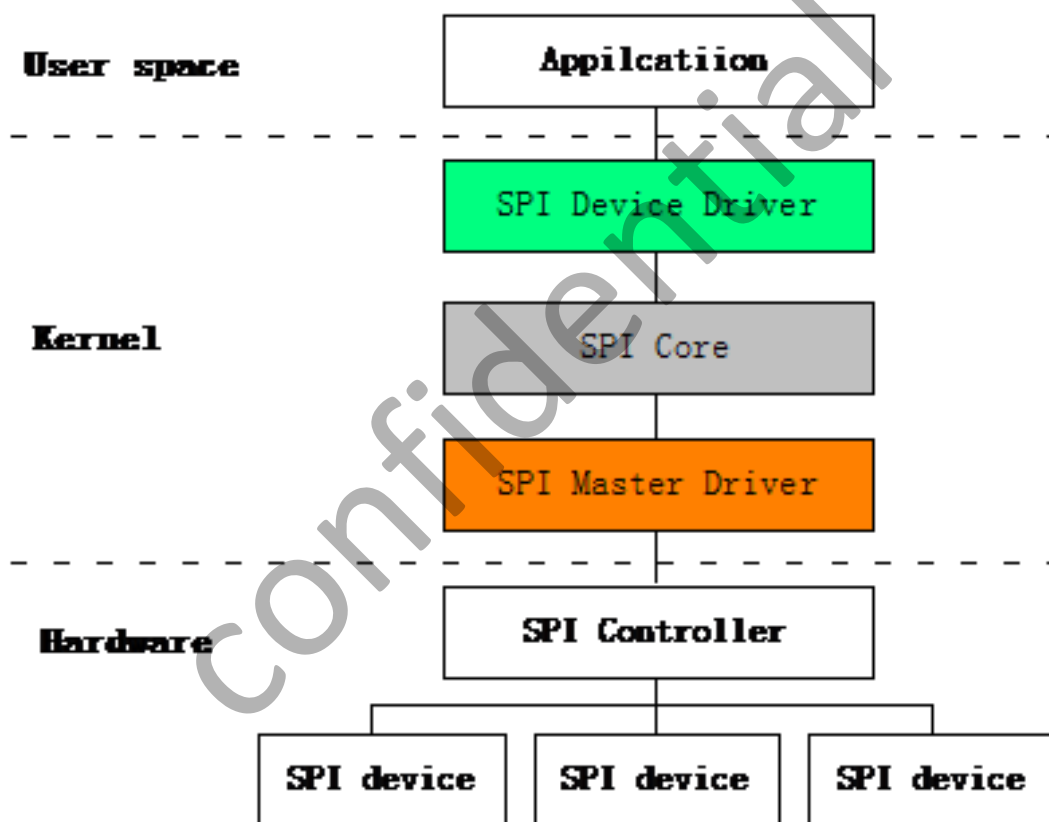
SPI 设备驱动、SPI 总线驱动的开发/ 护人员。

2. 模块介绍

2.1 模块功能介绍

Linux 中 SPI 体系结构如下图所示，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。



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

目前在不同的 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]
spi0_used    = 1
spi0_cs_number = 1
spi0_cs_bitmap = 1
spi0_cs0     = port:PC27<3><1><default><default>
spi0_sclk    = port:PC02<3><default><default><default>
spi0_mosi    = port:PC00<3><default><default><default>
```

```
spi0_miso    = port:PC01<3><default><default><default>
```

其中：

1. spi0_used: 为 1 表示使能，0 表示不使能；
2. spi0_cs_number: SPI 控制器支持的 选数；
3. spi0_cs_bitmap: 由于 SPI 控制器支持多个 CS，这一个参数表示 CS 的掩码；
4. spi0_cs0、spi0_sclk、spi0_mosi 和 spi0_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。

```
[spi0/spi_board0]
compatible      = "st,m25p32"
spi-max-frequency = 33000000
reg             =
```

其中：

1. spi0/spi_board0: 表示挂在总线 spi0 下的设备 spi_board0；
2. compatible: SPI 设备的名字，在做总线适配时会用到；
3. spi-max-frequency: SPI 设备的最大传输速度，单位是 Hz；
4. reg: SPI 设备的 选线，取决于硬件的 CS 连线；
5. 其它可选配置参数：spi-cpha, spi-cpol, spi-cs-high。

2.3.2 menuconfig 配置说明

在命令行中进入内核根目录，执行 `make ARCH=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：

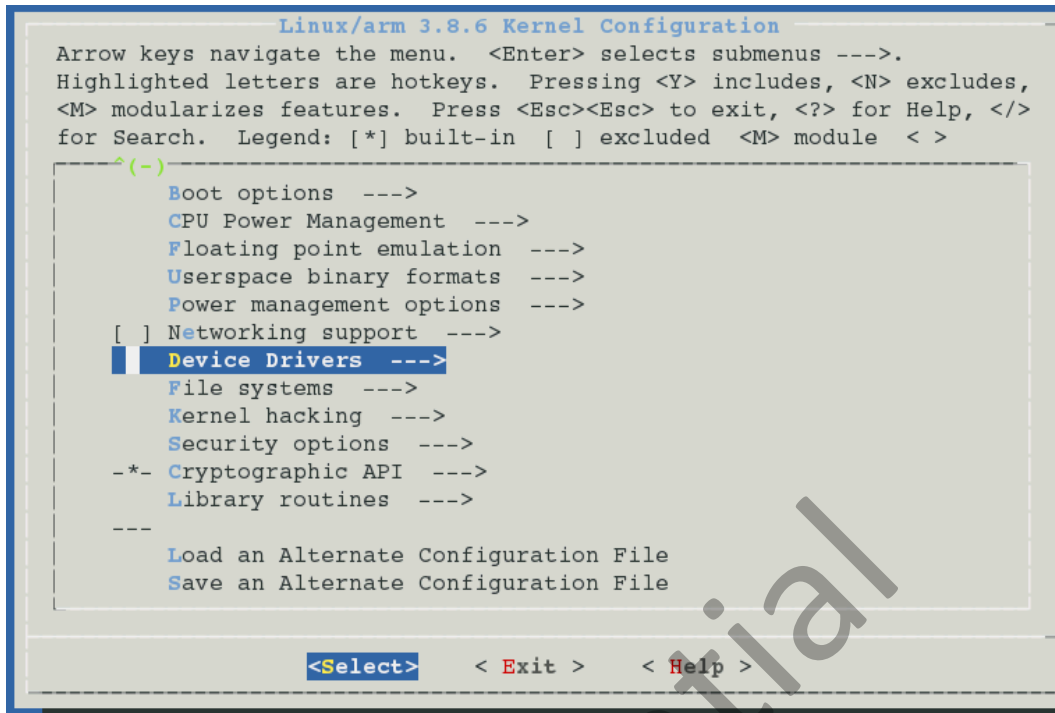


图 2: Device Drivers 配置选项

然后，选择 SPI support 选项，进入下一级配置，如下图所示：

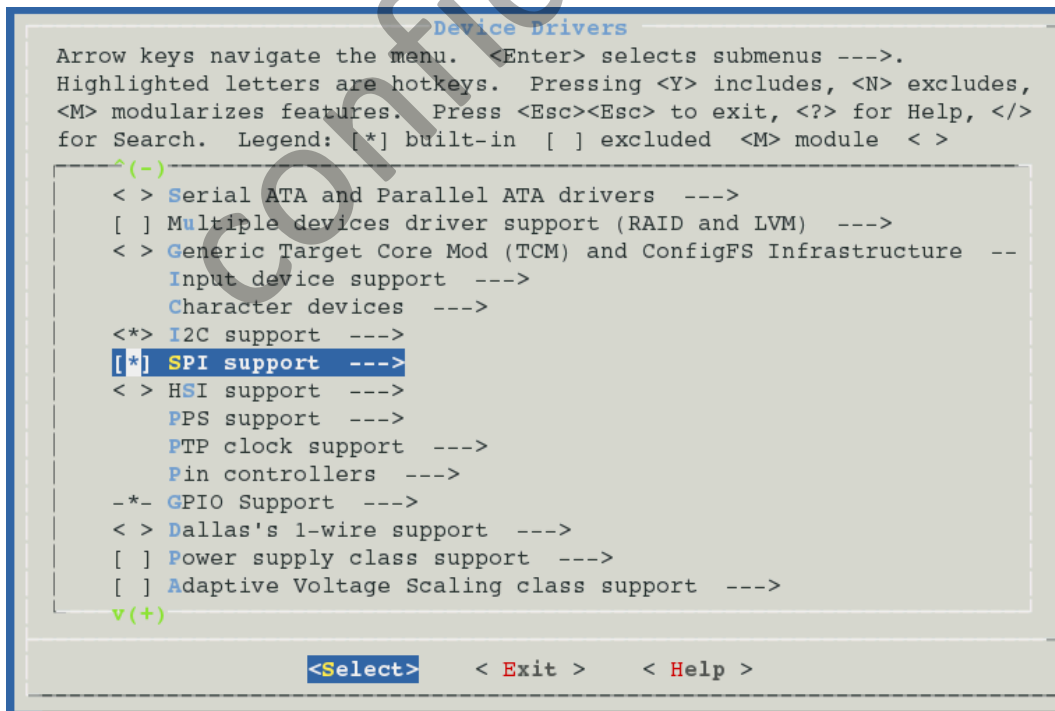


图 3: SPI support 配置选项

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

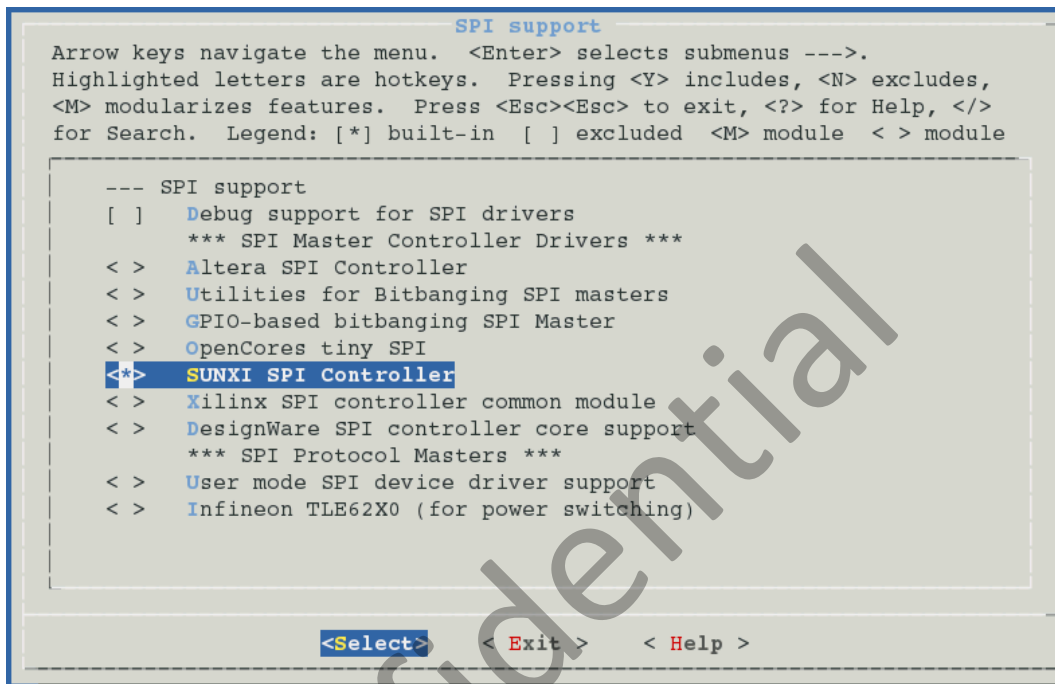


图 4: SUNXI SPI Controller 配置选项

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

定义在 `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 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 了 SPI 设备的名称、probe 等接口信息

【返回值】: 无

spi.h 中还给出了块速注册的 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 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; 小于 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, 用于保存读取到的缓存数据; len, buf 的长度

【返回值】: 0, 成功; 小于 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; 小于 0, 失败

4. demo

4.1 drivers/spi/tle62x0.c

源文件为内核中自带的一个 SPI 设备驱动代码，其中通过 sysfs 方式提供读写方面操作，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",
    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
");
    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
");
        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);
```

4.2 drivers/mtd/device/m25p80.c

现在说的 SPI 设备是一个 NorFlash，需要 MTD 架构的支持，其中实现稍微复杂，和 4.1 一样需要调用 `spi_register_driver()` 注册 SPI 驱动，需要通过 SPI message 实现数据的读写。最大区别在于 `m25p80.c` 另外需要实现 `mtd_info` 的接口，注册 MTD 设备，这样就可以通过 MTD 块设备/字符设备来读写 NorFlash。

5. Declaration

This document is the original work and copyrighted property of Allwinner Technology (“Allwinner”). Reproduction in whole or in part must obtain the written approval of Allwinner and give clear acknowledgement to the copyright owner. The information furnished by Allwinner is believed to be accurate and reliable. Allwinner reserves the right to make changes in circuit design and/or specifications at any time without notice. Allwinner does not assume any responsibility and liability for its use. Nor for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of Allwinner. This datasheet neither states nor implies warranty of any kind, including fitness for any particular application.

Confidential