链滴

Linux SPI 驱动

作者：zhang-ke-wei

原文链接：https://ld246.com/article/1592578900650

来源网站：链滴

SPI 驱动框架和 I2C 很类似，都分为主机控制器驱动和设备驱动，主机控制器也就是 SOC的 SPI 控制接口。编写好 SPI 控制器驱动以后就可以直接使用，SPI 控制器部分的驱动都是一样，这部分代码由导体厂商提供，重点在于各种类繁多的 SPI 设备驱动。

一、SPI 主机驱动

SPI 主机驱动就是 SOC 的 SPI 控制器驱动，类似 I2C 驱动里面的适配器驱动。 Linux 内核使用 spi_master 表示 SPI 主机驱动， spi_master 是个结构体，定义在 include/linux/spi/spi.h 文件中，内容如下有缩减)：

```
struct spi_master {
    struct device dev;

    struct list_head list;
    ......
    s16 bus_num;

    /* chipselects will be integral to many controllers; some others
    * might use board-specific GPIOs.
    */
    u16 num_chipselect;

    /* some SPI controllers pose alignment requirements on DMAable
    * buffers; let protocol drivers know about these requirements.
    */
    u16 dma_alignment;

    /* spi_device.mode flags understood by this controller driver */
    u16 mode_bits;

    /* bitmask of supported bits_per_word for transfers */
```

```
    u32 bits_per_word_mask;
    ......
    /* limits on transfer speed */
    u32 min_speed_hz;
    u32 max_speed_hz;

    /* other constraints relevant to this driver */
    u16 flags;
    ......
    /* lock and mutex for SPI bus locking */
    spinlock_t bus_lock_spinlock;
    struct mutex bus_lock_mutex;

    /* flag indicating that the SPI bus is locked for exclusive use */
    bool bus_lock_flag;
    ......
    int (*setup)(struct spi_device *spi);

    ......
    int (*transfer)(struct spi_device *spi,
    struct spi_message *mesg);
    ......
    int (*transfer_one_message)(struct spi_master *master,
    struct spi_message *mesg);
    ......
};
```

transfer 函数，和 i2c_algorithm 中的 master_xfer 函数一样，控制器数据传输函数。

transfer_one_message 函数，用于 SPI 数据发送，用于发送一个 spi_message，SPI 的数据会打成 spi_message，然后以队列方式发送出去。

SPI 主机端最终会通过 transfer 函数与 SPI 设备进行通信，因此对于 SPI 主机控制器的驱动编写者而 transfer 函数是需要实现的，不同的 SOC 其 SPI 控制器不同，寄存器都不一样。和 I2C 适配器驱动样， SPI 主机驱动一般都是 SOC 厂商编写。

SPI 主机驱动的核心就是申请 spi_master，然后初始化 spi_master，最后向 Linux 内核注册spi_master。

1、spi_master 申请与释放

spi_alloc_master 函数用于申请 spi_master，函数原型如下：

```
struct spi_master *spi_alloc_master(struct device *dev, unsigned size)
```

函数参数和返回值含义如下：

dev：设备，一般是 platform_device 中的 dev 成员变量。

size： 私有数据大小，可以通过 spi_master_get_devdata 函数获取到这些私有数据。

返回值： 申请到的 spi_master。

spi_master 的释放通过 spi_master_put 函数来完成，当我们删除一个 SPI 主机驱动的时候就需要释放掉前面申请的 spi_master， spi_master_put 函数原型如下：

```
void spi_master_put(struct spi_master *master)
```

函数参数和返回值含义如下：

master：要释放的 spi_master。

返回值： 无。

2、 spi_master 的注册与注销

当 spi_master 初始化完成以后就需要将其注册到 Linux 内核， spi_master 注册函数为 spi_register_master，函数原型如下：

```
int spi_register_master(struct spi_master *master)
```

函数参数和返回值含义如下：

master：要注册的 spi_master。

返回值： 0，成功；负值，失败。

如果要注销 spi_master 可以使用 spi_unregister_master 函数，此函数原型为：

```
void spi_unregister_master(struct spi_master *master)
```

函数参数和返回值含义如下：

master：要注销的 spi_master。

返回值： 无。

如果使用 spi_bitbang_start 注册 spi_master 就要使用 spi_bitbang_stop 来注销掉spi_master。

二、SPI 设备驱动

spi 设备驱动也和 i2c 设备驱动也很类似， Linux 内核使用 spi_driver 结构体来表示 spi 设备驱动，们在编写 SPI 设备驱动的时候需要实现 spi_driver。 spi_driver 结构体定义在include/linux/spi/spi.h 文件中，结构体内容如下：

```
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);
    struct device_driver    driver;
};
```

spi_driver 和 i2c_driver、 platform_driver 基本一样，当 SPI 设备和驱动匹配成功以后 probe 函数会执行。同样的， spi_driver 初始化完成以后需要向 Linux 内核注册， spi_driver 注册函数为spi_register_driver，函数原型如下：

```
int spi_register_driver(struct spi_driver *sdrv)
```

函数参数和返回值含义如下：

sdrv： 要注册的 spi_driver。

返回值： 0，注册成功；赋值，注册失败。

注销 SPI 设备驱动以后也需要注销掉前面注册的 spi_driver，使用 spi_unregister_driver 函数完成 sp_driver的注销，函数原型如下：

```
void spi_unregister_driver(struct spi_driver *sdrv)
```

函数参数和返回值含义如下：

sdrv： 要注销的 spi_driver。

返回值： 无。

exampel

```
/* probe 函数 */
static int xxx_probe(struct spi_device *spi)
{
    /* 具体函数内容 */
    return 0;
}

/* remove 函数 */
static int xxx_remove(struct spi_device *spi)
{
    /* 具体函数内容 */
    return 0;
}

/* 传统匹配方式 ID 列表 */
static const struct spi_device_id xxx_id[] = {
    {"xxx", 0},
    {}
};

/* 设备树匹配列表 */
static const struct of_device_id xxx_of_match[] = {
    { .compatible = "xxx" },
    { /* Sentinel */ }
};

/* SPI 驱动结构体 */
static struct spi_driver xxx_driver = {
    .probe = xxx_probe,
    .remove = xxx_remove,
    .driver = {
        .owner = THIS_MODULE,
        .name = "xxx",
        .of_match_table = xxx_of_match,
    },
    .id_table = xxx_id,
};

/* 驱动入口函数 */
static int __init xxx_init(void)
    return spi_register_driver(&xxx_driver);
}
```

```
/* 驱动出口函数 */
static void __exit xxx_exit(void)
{
    spi_unregister_driver(&xxx_driver);
}

module_init(xxx_init);
module_exit(xxx_exit);
```

三、SPI 设备和驱动匹配过程

SPI 设备和驱动的匹配过程是由 SPI 总线来完成的，这点和 platform、 I2C 等驱动一样， SPI总线为pi_bus_type，定义在 drivers/spi/spi.c 文件中，内容如下：

```
struct bus_type spi_bus_type = {
    .name = "spi",
    .dev_groups = spi_dev_groups,
    .match = spi_match_device,
    .uevent = spi_uevent,
};
```

SPI 设备和驱动的匹配函数为 spi_match_device，函数内容如下：

```
static int spi_match_device(struct device *dev, struct device_driver *drv)
{
    const struct spi_device *spi = to_spi_device(dev);
    const struct spi_driver *sdrv = to_spi_driver(drv);

    /* Attempt an OF style match */
    if (of_driver_match_device(dev, drv))
        return 1;

    /* Then try ACPI */
    if (acpi_driver_match_device(dev, drv))
        return 1;

    if (sdrv->id_table)
        return !!spi_match_id(sdrv->id_table, spi);

    return strcmp(spi->modalias, drv->name) == 0;
}
```

of_driver_match_device 函数用于完成设备树设备和驱动匹配。比较 SPI 设备节点的 compatible 性和 of_device_id 中的 compatible 属性是否相等，如果相当的话就表示 SPI 设备和驱动匹配。

acpi_driver_match_device 函数用于 ACPI 形式的匹配。

spi_match_id 函数用于无设备树的 SPI 设备和驱动匹配过程。比较 SPI设备名字和 spi_device_id 的ame 字段是否相等，相等的话就说明 SPI 设备和驱动匹配。

四、SPI 设备驱动编写流程

（一）SPI 设备信息描述

IO 的 pinctrl 子节点创建与修改、SPI 设备节点的创建与修改

（二）SPI 设备数据收发处理流程

SPI 设备驱动的核心是 spi_driver，这个我们已经在 62.1.2 小节讲过了。当我们向 Linux 内核注册成 spi_driver 以后就可以使用 SPI 核心层提供的 API 函数来对设备进行读写操作了。首先是 spi_transfe 结构体，此结构体用于描述 SPI 传输信息， 结构体内容如下：

```
struct spi_transfer {
    /* it's ok if tx_buf == rx_buf (right?)
     * for MicroWire, one buffer must be null
     * buffers must work with dma_*map_single() calls, unless
     *   spi_message.is_dma_mapped reports a pre-existing mapping
     */
    const void    *tx_buf;
    void        *rx_buf;
    unsigned    len;

    dma_addr_t    tx_dma;
    dma_addr_t    rx_dma;
    struct sg_table tx_sg;
    struct sg_table rx_sg;

    unsigned    cs_change:1;
    unsigned    tx_nbits:3;
    unsigned    rx_nbits:3;
#define    SPI_NBITS_SINGLE    0x01 /* 1bit transfer */
#define    SPI_NBITS_DUAL        0x02 /* 2bits transfer */
#define    SPI_NBITS_QUAD        0x04 /* 4bits transfer */
    u8        bits_per_word;
    u16        delay_usecs;
    u32        speed_hz;

    struct list_head transfer_list;
};
```

spi_transfer 需要组织成 spi_message， spi_message 也是一个结构体，内容如下：

```
struct spi_message {
    struct list_head    transfers;

    struct spi_device    *spi;

    unsigned        is_dma_mapped:1;

    /* REVISIT:  we might want a flag affecting the behavior of the
     * last transfer ... allowing things like "read 16 bit length L"
     * immediately followed by "read L bytes".  Basically imposing
     * a specific message scheduling algorithm.
     *
     * Some controller drivers (message-at-a-time queue processing)
```

```
     * could provide that as their default scheduling algorithm.  But
     * others (with multi-message pipelines) could need a flag to
     * tell them about such special cases.
     */

    /* completion is reported through a callback */
    void            (*complete)(void *context);
    void            *context;
    unsigned        frame_length;
    unsigned        actual_length;
    int             status;

    /* for optional use by whatever driver currently owns the
     * spi_message ...  between calls to spi_async and then later
     * complete(), that's the spi_master controller driver.
     */
    struct list_head    queue;
    void            *state;
};
```

在使用spi_message之前需要对其进行初始化， spi_message初始化函数为spi_message_init，函数型如下：

```
void spi_message_init(struct spi_message *m)
```

函数参数和返回值含义如下：

m： 要初始化的 spi_message。

返回值： 无。

spi_message 初始化完成以后需要将 spi_transfer 添加到 spi_message 队列中，这里我们要用到 spi message_add_tail 函数，此函数原型如下：

```
void spi_message_add_tail(struct spi_transfer *t, struct spi_message *m)
```

函数参数和返回值含义如下：

t： 要添加到队列中的 spi_transfer。

m： spi_transfer 要加入的 spi_message。

返回值： 无。

spi_message 准备好以后既可以进行数据传输了，数据传输分为同步传输和异步传输，同步传输会阻的等待 SPI 数据传输完成，同步传输函数为 spi_sync，函数原型如下：

```
int spi_sync(struct spi_device *spi, struct spi_message *message)
```

函数参数和返回值含义如下：

spi： 要进行数据传输的 spi_device。

message：要传输的 spi_message。

返回值： 无。

异步传输不会阻塞的等到 SPI 数据传输完成，异步传输需要设置 spi_message 中的 complete成员量， complete 是一个回调函数，当 SPI 异步传输完成以后此函数就会被调用。 SPI 异步传输函数为 pi_async，函数原型如下：

```
int spi_async(struct spi_device *spi, struct spi_message *message)
```

函数参数和返回值含义如下：

spi： 要进行数据传输的 spi_device。

message：要传输的 spi_message。

返回值： 无。

在本章实验中，我们采用同步传输方式来完成 SPI 数据的传输工作，也就是 spi_sync 函数。

综上所述， SPI 数据传输步骤如下：

①、申请并初始化 spi_transfer，设置 spi_transfer 的 tx_buf 成员变量， tx_buf 为要发送的数据。后设置 rx_buf 成员变量， rx_buf 保存着接收到的数据。最后设置 len 成员变量，也就是要进行数据信的长度。

②、使用 spi_message_init 函数初始化 spi_message。

③、使用spi_message_add_tail函数将前面设置好的spi_transfer添加到spi_message队列中。

④、使用 spi_sync 函数完成 SPI 数据同步传输。

example

```
/* SPI 多字节发送 */
static int spi_send(struct spi_device *spi, u8 *buf, int len)
{
    int ret;
    struct spi_message m;

    struct spi_transfer t = {
        .tx_buf = buf,
        .len = len,
    };

    spi_message_init(&m); /* 初始化 spi_message */
    spi_message_add_tail(t, &m);/* 将 spi_transfer 添加到 spi_message 队列 */
    ret = spi_sync(spi, &m); /* 同步传输 */

    return ret;
}

/* SPI 多字节接收 */
static int spi_receive(struct spi_device *spi, u8 *buf, int len)
{
    int ret;
    struct spi_message m;

    struct spi_transfer t = {
        .rx_buf = buf,
        .len = len,
    };

    spi_message_init(&m); /* 初始化 spi_message */
    spi_message_add_tail(t, &m);/* 将 spi_transfer 添加到 spi_message 队列 */
```

```
    ret = spi_sync(spi, &m); /* 同步传输 */

    return ret;
}
```