

WK 系列 SPI 拓展 4 串口驱动移植参考文档 V2.4

修订记录

版本号	修订时间	作者	修订内容
V2.4	20220625	胥勋伟	创建版本

目录

修订记录	2
目 录	错误!未定义书签。
1 概述	5
1.1 WK 系列串口扩展芯片简介	5
2 SPI 拓展串口驱动简介	5
2.1 硬件连接示意图	5
2.2 LINUX 串口驱动基本框架简介	6
3 WK SPI 拓展 UART 驱动简介	7
3.1 开平台简介	7
3.1.1 硬件平台	7
3.1.2 软件平台简介	7
3.2 驱动介绍	7
3.2.1 串口驱动信息描述和数据结构	7
3.2.2 串口驱动的底层基本操作	9
3.2.3 驱动驱动架构与应用层（用户空间）之间的分析	14
4 驱动的移植	19
4.1 配置 DTS 节点	19
4.2 驱动修改	20
4.2.1 晶振频率值修改	20
4.2.2 调试接口	21
4.2.3 功能接口	21
4.2.4 芯片型号修改	21
4.3 驱动的编译	22
4.3.1 编译前的准备工作	22

4.3.2 编译驱动	23
4.4 驱动调试	26
4.4.1 加载驱动	26
4.4.2 测试	27
5 特别申明.....	28

1 概述

本文档主要适用于 SPI 扩展 UART 驱动移植的参考。本文档基于 V2.4 版本的驱动来进行说明的，其它版本的驱动也可以参考。

1.1 WK 系列串口扩展芯片简介

目前 WK 系列能实现 SPI 扩展 UART 的芯片包括 WK2124、WK2204、WK2168、WK2132。目前 WK2124、WK2204、WK2168 能实现 SPI 扩展 4 路 UART，WK2132 能实现扩展 2 路 UART。目前这几款芯片使用的都是相同的 linux 驱动。

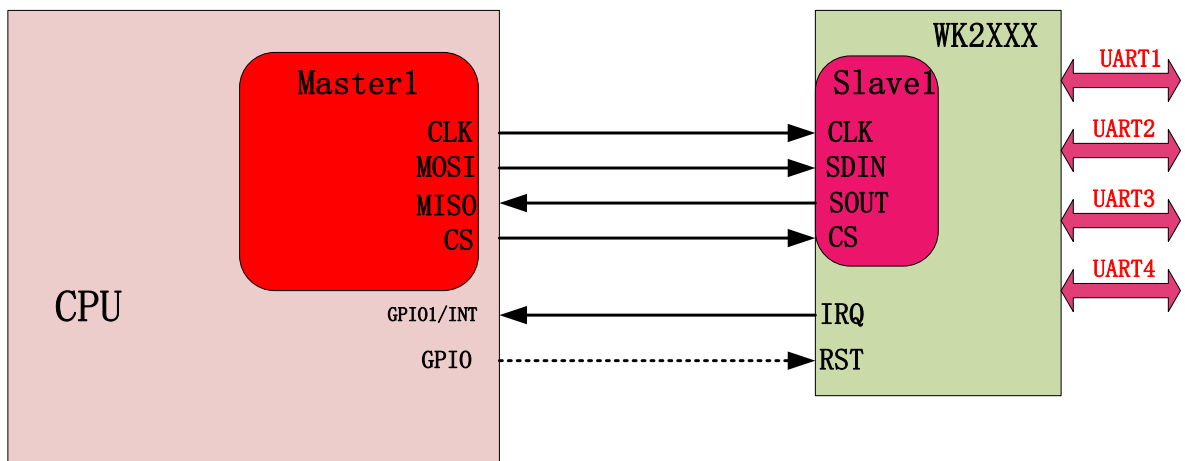
WK 系列扩展的子通道的 UART 具备如下功能特点：

每个子通道 UART 的波特率、字长、校验格式可以独立设置，最高可以提供 2Mbps 的通信速率。

每个子通道具备收/发独立的 256 级 FIFO，FIFO 的中断可按用户需求进行编程触发点且具备超时中断功能。

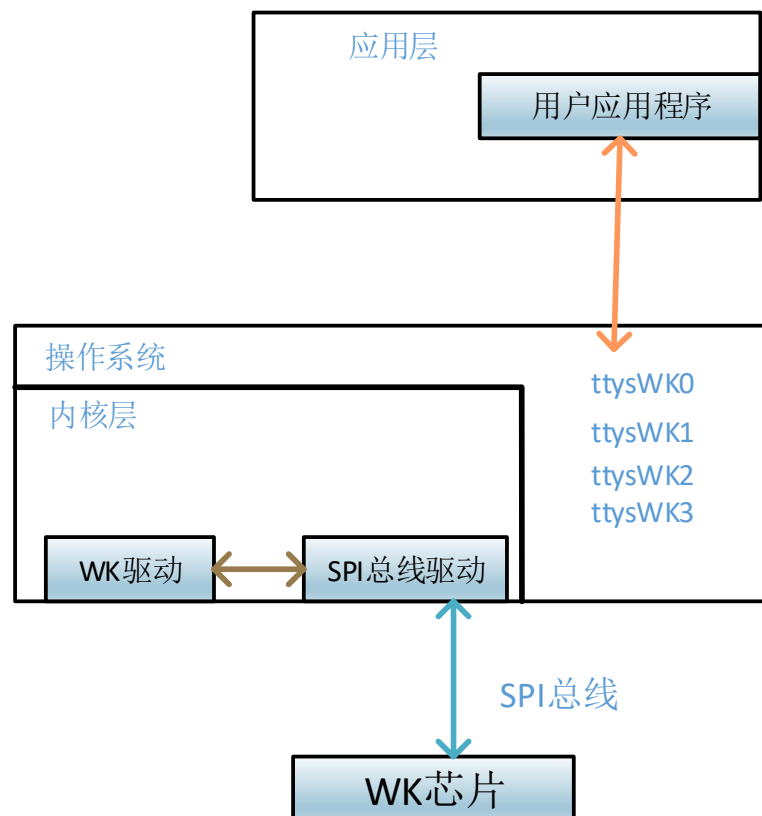
2 SPI 拓展串口驱动简介

2.1 硬件连接示意图



1. WK 芯片作 SPI 从设备和 CPU 端的主 SPI 需要连接的信号有 CS 信号（此信号不能一直拉低，需要用主 SPI 的 CS 信号控制）、CLK 信号、MOSI 信号、MISO 信号，具体连接方式如上图。
2. IRQ 信号为 WK 芯片的中断输出信号，需要连接到 CPU 具有外部中断功能的 GPIO 上。IRQ 引脚外部需要加上拉电阻。
3. RST 作为复位引脚，在 SPI 拓展 4 串口的时候，可以不用连接到 CPU.直接使用阻容复位电路。

2.2 linux 串口驱动基本框架简介



1. WK 驱动工作在 linux 内核层，向上提供 4 个串口设备节点供应用层用户调用。也就是说 WK 驱动注册成功以后，在/dev/ 目录下会生成 ttysWK0、ttysWK1、ttysWK2、ttysWK3 共 4 个串口设备节点，应用层就可以按照操作普通串口节点的方式操作。

2. WK 驱动需要和 WK 芯片进行数据交互，数据交互是通过 SPI 总线进行的，所以 WK 驱动会调用 SPI 总线驱动接口进行数据收发。

3 WK SPI 拓展 UART 驱动简介

3.1 开平台简介

3.1.1 硬件平台

本驱动在开发的时候使用了 Firefly-RK3399 这款开发板。该开发板接口丰富，开发驱动很方便。

3.1.2 软件平台简介

该驱动是在 Ubuntu16.04 系统上开发。内核版本 4.4.具体见下图：

```
root@firefly:/# uname -a
Linux firefly 4.4.179 #4 SMP Wed Mar 9 14:21:44 CST 2022 aarch64 aarch64 aarch64 GNU/Linux
root@firefly:/#
```

3.2 驱动介绍

驱动源文件 wk2xxx_spi.c

下面对驱动作一些基本介绍。

3.2.1 串口驱动信息描述和数据结构

3.2.1.1 串口驱动描述

鉴于芯片的相关特性和驱动编写的需要，定义了结构体 wk2xxx_port 用于对 WK 的 SPI 转串口驱动进行描述。程序清单如下所示。

```
struct wk2xxx_port
{
    const struct wk2xxx_devtype *devtype;
    struct uart_driver          uart;
    struct spi_device           *spi_wk;
```

```

struct workqueue_struct *workqueue;
struct work_struct work;
unsigned char          buf[256];
struct kthread_worker  kworker;
struct task_struct     *kworker_task;
struct kthread_work    irq_work;
int irq_gpio_num;      /*中断 IO 的 GPIO 编号*/
int rst_gpio_num;      /*复位引脚的 GPIO 编号*/
int irq_gpio;          /*中断编号*/
int minor;             /* minor number */
int tx_empty;
struct wk2xxx_one      p[NR_PORTS];
};

```

3.2.1.2 串口端口描述

定义一个结构体 wk2xxx_one 来描述 WK2xxx 芯片的串口端口进行描述，实际上是对 uart_port 的进一步封装，增加了两个内核队列和芯片子串口一些寄存器。程序如下：

```

struct wk2xxx_one
{
    struct uart_port port; // [NR_PORTS];
    struct kthread_work start_tx_work;
    struct kthread_work stop_rx_work;
    uint8_t line;
    uint8_t new_lcr_reg;
    uint8_t new_fwcr_reg;
    uint8_t new_scr_reg;
    /*baud register*/
    uint8_t new_baud1_reg;
    uint8_t new_baud0_reg;
    uint8_t new_pres_reg;
};

```


3.2.2 串口驱动的底层基本操作

3.2.2.1 读全局寄存器描述

该函数调用 SPI 接口，实现读 WK2XXX 芯片的全局寄存器，全局寄存器通常包括 GENA、GRST、GIER、GIFR、GMUT、GPDIR、GPDAT 等

```
static int wk2xxx_read_global_reg(struct spi_device *spi, uint8_t reg, uint8_t *
dat)
{
    struct spi_message msg;
    uint8_t buf_wdat[2];
    uint8_t buf_rdat[2];
    int status;
    struct spi_transfer index_xfer = {
        .len          = 2,
        .speed_hz     = wk2xxx_spi_speed,
    };
    mutex_lock(&wk2xxxs_reg_lock);
    status = 0;
    spi_message_init(&msg);
    buf_wdat[0] = 0x40|reg;
    buf_wdat[1] = 0x00;
    buf_rdat[0] = 0x00;
    buf_rdat[1] = 0x00;
    index_xfer.tx_buf = buf_wdat;
    index_xfer.rx_buf =(void *) buf_rdat;
    spi_message_add_tail(&index_xfer, &msg);
    status = spi_sync(spi, &msg);
    mutex_unlock(&wk2xxxs_reg_lock);
    if(status){
        return status;
    }
    *dat = buf_rdat[1];
    return 0;
}
```

3.2.2.2 写全局寄存器

该函数调用 SPI 接口，实现对 WK2XXX 芯片的全局寄存器写操作，全局寄存器通

常包括 GENA、GRST、GIER、GIFR、GMUT、GPDIR、GPDAT 等

```
/*
 * This function write wk2xxx of Global register:
 */
static int wk2xxx_write_global_reg(struct spi_device *spi,uint8_t reg,uint8_t
dat)
{
    struct spi_message msg;
    uint8_t buf_reg[2];
    int status;
    struct spi_transfer index_xfer = {
        .len          = 2,
        .speed_hz     = wk2xxx_spi_speed,
    };
    mutex_lock(&wk2xxxs_reg_lock);
    spi_message_init(&msg);
    /* register index */
    buf_reg[0] = 0x00|reg;
    buf_reg[1] = dat;
    index_xfer.tx_buf = buf_reg;
    spi_message_add_tail(&index_xfer, &msg);
    status = spi_sync(spi, &msg);
    mutex_unlock(&wk2xxxs_reg_lock);
    return status;
}
```

3.2.2.3 读子串口寄存器函数描述

该函数调用 SPI 接口实现读子串口的相关寄存器。Uint8_t port 这个参数表示对应的子串口编号。

```
/*
 * This function read wk2xxx of slave register:
 */
static int wk2xxx_read_slave_reg(struct spi_device *spi,uint8_t port,uint8_t r
eg,uint8_t *dat)
{
    struct spi_message msg;
    uint8_t buf_wdat[2];
    uint8_t buf_rdat[2];
    int status;
    struct spi_transfer index_xfer = {
```

```

        .len          = 2,
        .speed_hz     = wk2xxx_spi_speed,
    };
    mutex_lock(&wk2xxxs_reg_lock);
    status = 0;
    spi_message_init(&msg);
    buf_wdat[0] = 0x40|(((port-1)<<4)|reg);
    buf_wdat[1] = 0x00;
    buf_rdat[0] = 0x00;
    buf_rdat[1] = 0x00;
    index_xfer.tx_buf = buf_wdat;
    index_xfer.rx_buf =(void *) buf_rdat;
    spi_message_add_tail(&index_xfer, &msg);
    status = spi_sync(spi, &msg);
    mutex_unlock(&wk2xxxs_reg_lock);
    if(status){
        return status;
    }
    *dat = buf_rdat[1];
    return 0;
}

```

3.2.2.4 写子串口寄存器函数描述

该函数调用 SPI 接口，实现写子串口的寄存器。

```

/*
 * This function write wk2xxx of Slave register:
 */
static int wk2xxx_write_slave_reg(struct spi_device *spi,uint8_t port,uint8_t
reg,uint8_t dat)
{
    struct spi_message msg;
    uint8_t buf_reg[2];
    int status;
    struct spi_transfer index_xfer = {
        .len          = 2,
        .speed_hz     = wk2xxx_spi_speed,
    };
    mutex_lock(&wk2xxxs_reg_lock);
    spi_message_init(&msg);
    /* register index */

```

```

buf_reg[0] = ((port-1)<<4)|reg;
buf_reg[1] = dat;
index_xfer.tx_buf = buf_reg;
spi_message_add_tail(&index_xfer, &msg);
status = spi_sync(spi, &msg);
mutex_unlock(&wk2xxxs_reg_lock);
return status;
}

```

3.2.2.5 读 FIFO 函数描述

该函数通过调用 SPI 接口实现读子串口的 FIFO（也就是子串口的接收缓存区）。

具体函数如下：

```

static int wk2xxx_read_fifo(struct spi_device *spi,uint8_t port,uint8_t fifolen,
uint8_t *dat)
{
    struct spi_message msg;
    int status,i;
    uint8_t recive_fifo_data[MAX_RFCOUNT_SIZE+1]={0};
    uint8_t transmit_fifo_data[MAX_RFCOUNT_SIZE+1]={0};
    struct spi_transfer index_xfer = {
        .len          = fifolen+1,
        .speed_hz     = wk2xxx_spi_speed,
    };
    if(!(fifolen>0)){
        printk(KERN_ERR "%s,fifolen error!!\n", __func__);
        return 1;
    }
    mutex_lock(&wk2xxxs_reg_lock);
    spi_message_init(&msg);
    /* register index */
    transmit_fifo_data[0] = ((port-1)<<4)|0xc0;
    index_xfer.tx_buf = transmit_fifo_data;
    index_xfer.rx_buf =(void *) recive_fifo_data;
    spi_message_add_tail(&index_xfer, &msg);
    status = spi_sync(spi, &msg);
    for(i=0;i<fifolen;i++)
        *(dat+i)=recive_fifo_data[i+1];
    mutex_unlock(&wk2xxxs_reg_lock);
    return status;
}

```

```
}
```

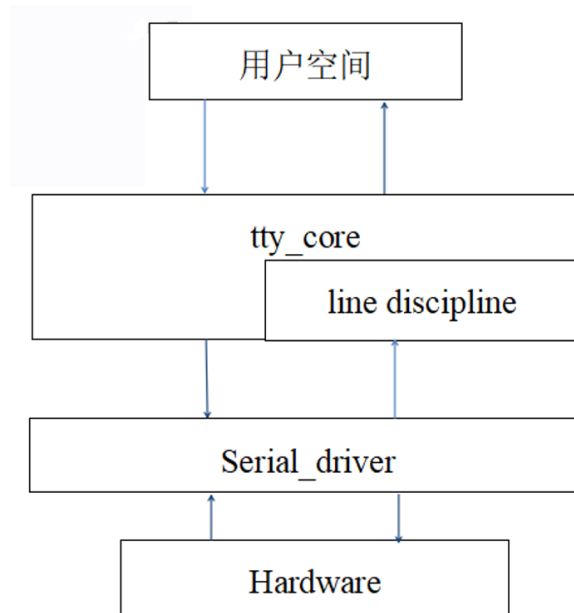
3.2.2.6 写 FIFO 函数描述

该函数通过调用 SPI 接口实现写子串口 FIFO（也就是子串口发送缓存区）。具体函数如下：

```
static int wk2xxx_write_fifo(struct spi_device *spi, uint8_t port, uint8_t fifolen, uint8_t *dat)
{
    struct spi_message msg;
    int status, i;
    uint8_t receive_fifo_data[MAX_RFCOUNT_SIZE+1]={0};
    uint8_t transmit_fifo_data[MAX_RFCOUNT_SIZE+1]={0};
    struct spi_transfer index_xfer = {
        .len          = fifolen+1,
        .speed_hz     = wk2xxx_spi_speed,
    };
    if(!(fifolen>0)){
        printk(KERN_ERR "%s, fifolen error, fifolen:%d!!\n", __func__, fifolen);
        return 1;
    }
    mutex_lock(&wk2xxxs_reg_lock);
    spi_message_init(&msg);
    /* register index */
    transmit_fifo_data[0] = ((port-1)<<4)|0x80;
    for(i=0; i<fifolen; i++){
        transmit_fifo_data[i+1]=*(dat+i);
    }
    index_xfer.tx_buf = transmit_fifo_data;
    index_xfer.rx_buf =(void *) receive_fifo_data;
    spi_message_add_tail(&index_xfer, &msg);
    status = spi_sync(spi, &msg);
    mutex_unlock(&wk2xxxs_reg_lock);
    return status;
}
```

3.2.3 驱动驱动架构与应用层（用户空间）之间的分析

本驱动遵循标准的 tty 驱动的架构。tty 架构如下图所示：



一般来说 tty 架构可以分成两层：一层是下层我们的串口驱动层，直接操作 WK2XXX 芯片，同时向上提供一组标准的接口，这组接口通过结构体 `struct uart_ops` 来实现，该结构体涵盖了驱动对串口的所有操作。还有一层是上层 tty 层，包括 `tty_core`、`line_discipline`。他们各自实现实现一个 ops 结构，用户空间通过 tty 注册的字符设备节点来访问驱动。

Wk2xxx 驱动的 `struct uart_ops` 结构体如下：

```
static struct uart_ops wk2xxx_pops = {
    tx_empty:      wk2xxx_tx_empty,
    set_mctrl:     wk2xxx_set_mctrl,
    get_mctrl:     wk2xxx_get_mctrl,
    stop_tx:       wk2xxx_stop_tx,
    start_tx:      wk2xxx_start_tx,
    stop_rx:       wk2xxx_stop_rx,
    enable_ms:     wk2xxx_enable_ms,
    break_ctl:     wk2xxx_break_ctl,
    startup:       wk2xxx_startup,
    shutdown:      wk2xxx_shutdown,
    set_termios:   wk2xxx_termios,
    type:          wk2xxx_type,
```

```

release_port:    wk2xxx_release_port,
request_port:    wk2xxx_request_port,
config_port:     wk2xxx_config_port,
verify_port:     wk2xxx_verify_port,

};

```

3.2.3.1 驱动注册

在驱动编译完成以后。驱动加载成功以后，驱动会向系统注册 4 个串口设备节点，我们可以在/dev/ 目录下找到 ttysWK0、ttysWK1、ttysWK2、ttysWK3 这 4 个节点。

用户空间可以通过这个 4 个节点访问 4 个不同的串口。通常用户空间通过

3.2.3.2 用户空间 Open()\close()串口设备节点

用户空间通常通过 open()\close()函数打开或者关闭设备节点。

1、如下用户空间打开串口设备节点：

```

141 int OpenDev(char *Dev)
142 {
143     int fd;
144     fd = open( Dev, O_RDWR | O_NOCTTY | O_NDELAY );
145     if (-1 == fd){
146         perror("Can't Open Serial Port");
147         return -1;
148     }
149     if(fcntl(fd, F_SETFL, 0) < 0)
150         printf("fail\n");
151     else
152         printf("success\n");
153     return fd;
154 }
155

```

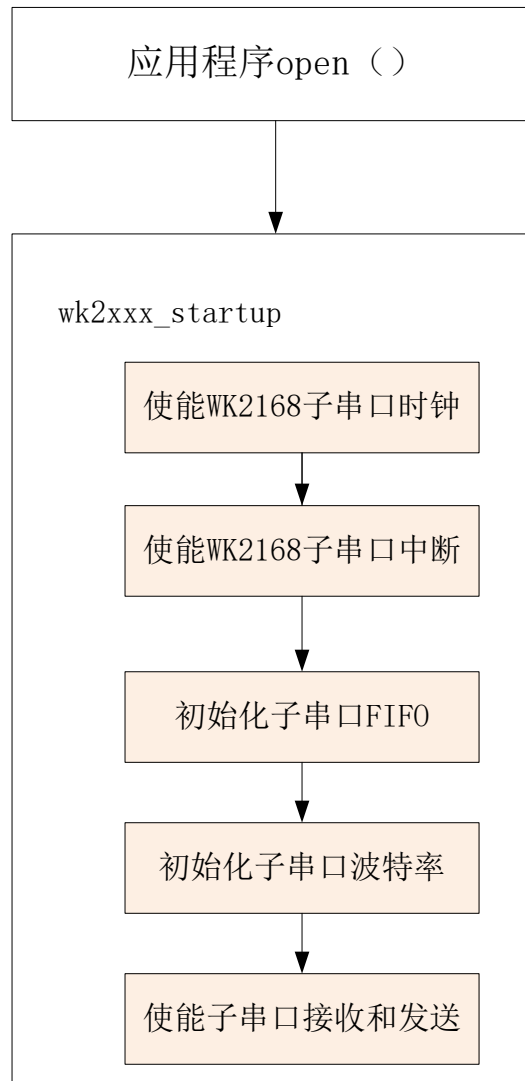
注意：Dev 为设备节点指针（设备节点的路径如下）

```
char *tty_name[4] = {"/dev/ttysWK0", "/dev/ttysWK1", "/dev/ttysWK2", "/dev/ttysWK3"};
```

当用户空间打开串口的时候，驱动层会调用如下函数：

```
static int wk2xxx_startup(struct uart_port *port)
```

该函数主要是来初始化 wk2xxx 芯片当前子串口的寄存器和设置子串口的初始波特率（115200）。示意图入下：



2、如用户空间关闭设备节点

用户关闭设备节点如下：`close(fd);`

注意：`fd` 是 `open` 串口时获得的。

当用户空间调用 `close()` 串口的时候。驱动主要是调用

`static void wk2xxx_shutdown(struct uart_port *port)` 函数来实现关闭串口。

该函数主要实现关闭子串口的时钟、中断等操作。

3.2.3.3 设置子串口波特率和数据格式

应用层设置波特率和数据格式有专

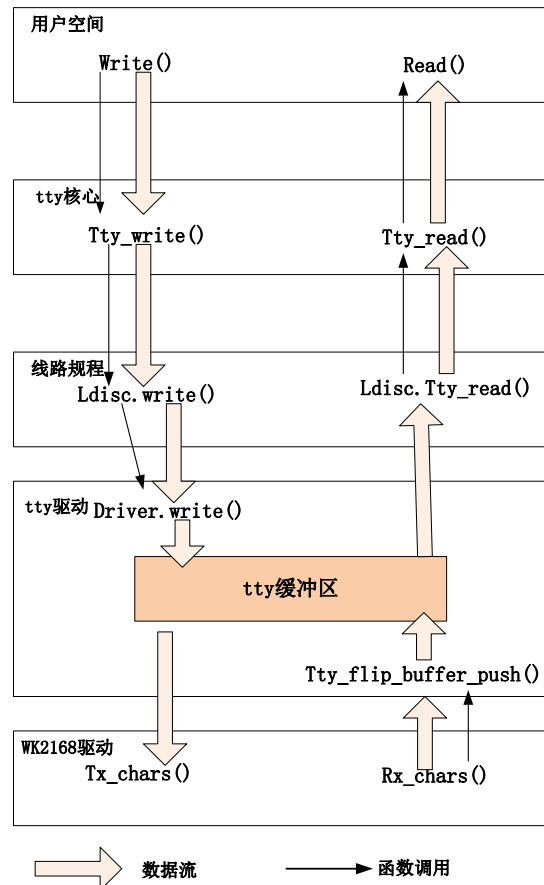
驱动层设置波特率是通过如下的函数来实现的：

```
static void wk2xxx_termios( struct uart_port *port, struct ktermios *termios,s
truct ktermios *old)
```


3.2.3.4 通过串口读写数

应用层通过 `write()` /`read()`函数来实现子串口的收发。那么驱动层是怎么来实现的：

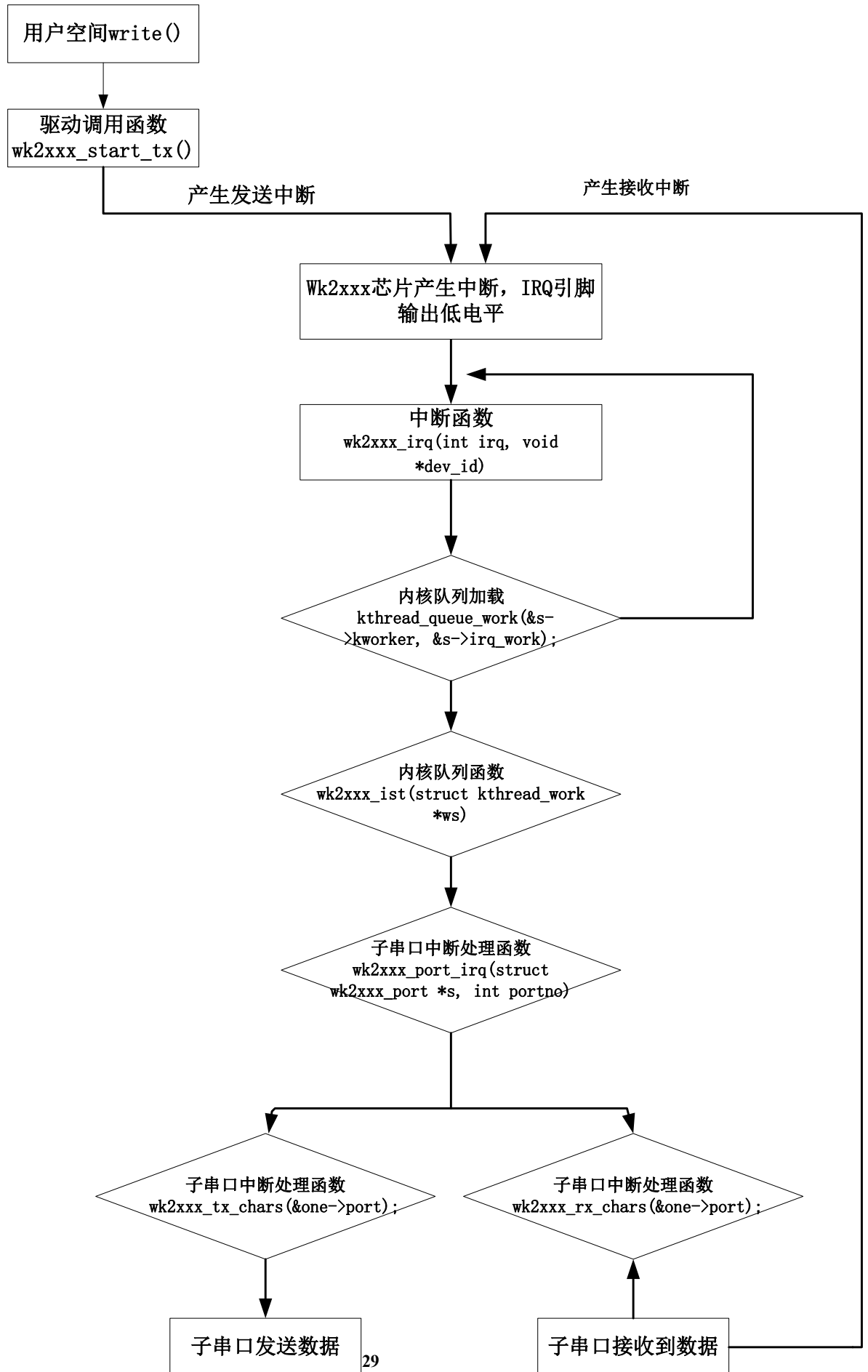
1、用户空间和驱动层之间的数据是怎么交互的。



用户空间和驱动层之间在数据传递上并不是直接传递的。当 `write()`写数据时，用户空间仅仅是把数据传递给 `tty` 缓冲区，然后驱动程序收到发送数据的指令，然后按照一定的流程去发送数据；当接收数据的时候，驱动层首先把接收的数据放入 `tty` 缓冲区，用户空间 `read()`去读数据，那么就能从 `tty` 缓冲区读出子串口接收的数据。

2、驱动层接收和发送数据的实现

驱动层接收和发送数据都依赖于中断。具体的示意图如下：



发送数据：用户空间需要发送数据，首先调用 `write()`，并把需要发送的数据传递到 `tty` 缓存区。驱动层调用 `wk2xxx_start_tx()` 告诉驱动有数据需要发送，WK2xxx 芯片产生中断，中断函数通过 `wk2xxx_tx_chars()` 函数把 `tty` 缓存区的数据取出来，并把数据写入 `wk2xxx` 芯片的发送 `fifo`，芯片再自动发送发送 `fifo` 中的数据。

接收数据：当 WK2xxx 芯片接收的数据都是暂时存在子串口的接收 `fifo`，当接收 `fifo` 中数据个数到达设置的接收中断触点，芯片产生接收中断，中断函数通过 `wk2xxx_rx_chars()` 函数，从接收 `fifo` 中读出接收的数据，然后传递给 `tty` 缓存区。那么用户空间就可以通过 `read()` 函数读到接收的数据。

4 驱动的移植

驱动的移植一般过程就是修改内核端的 DTS 配置，然后编译驱动，加载驱动，最后就是测试。

4.1 配置 DTS 节点

在 DTS 文件当中添加 SPI 驱动节点描述。如下图所示：

```
&spi1 {
    status = "okay";
    max-freq = <48000000>;
    wk2xxx_spi: wk2xxx_spi@00{
        status = "okay";
        compatible = "wkmic,wk2xxx_spi";
        reg = <0x00>;
        spi-max-frequency = <10000000>;
        reset_gpio = <&gpio4 29| GPIO_ACTIVE_HIGH>; //GPIO4_D5
        irq_gpio = <&gpio0 12 IRQ_TYPE_LEVEL_LOW>; //GPIO0_B4
    };
};
```

本驱动使用的是 SPI1，

1、status：如果要启用 SPI，那么设置为 `okay`，如不启用，设置为 `disable`

- 2、wk2xxx_spi@00:由于硬件使用的是 SPI1 的 cs0 引脚, 所以设置为 00.如果使用 cs1, 则设置为 01
- 3、compatible:这里的属性必须与驱动中的结构体: of_device_id 中的成员 compatible 保持一致。这个是 SPI 驱动匹配的关键。
- 4、reg: 此处与 wk2xxx_spi@00:保持一致。此处设置为: 00
- 5、spi-max-frequency: 此处设置 spi 使用的最高频率。wk2xxx 芯片 spi 最高支持 100000000。
- 6、reset_gpio:该选项在 SPI 驱动当中不是必须的。该 gpio 和 WK2xxx 芯片的复位引脚相连, 用于控制芯片的复位。根据实际使用的 gpio 去修改。
- 7、irq_gpio: 该 gpio 和 wk2xxx 芯片的 IRQ 引脚相连, 用于接收 wk2xxx 芯片传递来的中断信号。估计具体使用的 GPIO 去修改。
- 8、SPI 的工作模式设置, 默认工作在 0 模式, 所以在 dts 中没有单独设置。

4.2 驱动修改

驱动当中有些差异配置, 是需要根据具体的硬件使用情况去修改的。

4.2.1 晶振频率值修改

如下中 WK_CRASTAL_CLK 是芯片外部的实际晶振值, 目前我们在测试中使用的是 24Mhz 的晶振, 所以晶振值是 24000000.如果用的是 12Mhz 晶振就修改为 12000000.

```
84 #define NR_PORTS 4
85 //
86 #define SERIAL_WK2XXX_MAJOR 207
87 #define CALLOUT_WK2XXX_MAJOR 208
88 #define MINOR_START 5
89 //wk2xxx hardware configuration
90 #define wk2xxx_spi_speed 100000000
91 #define WK_CRASTAL_CLK (24000000)
92 #define WK2XXX_ISR_PASS_LIMIT 2
93 #define PORT_WK2XXX 1
94 /*****/
95
```

4.2.2 调试接口

开启如下的宏，可以在驱动运行的时候增加打印信息。方便调试

```

9  /*****The debug control *****/
0  #define _DEBUG_WK_FUNCTION
1  //#define _DEBUG_WK_RX
2  //#define _DEBUG_WK_TX
3  //#define _DEBUG_WK_IRQ
4  //#define _DEBUG_WK_VALUE
5  //#define _DEBUG_WK_TEST
6

```

4.2.3 功能接口

通常下面的这些功能，按照默认设置就可以，除非有相应需求才开启

```

6
7  /*****Functional control interface*****/
8  #define WK_FIFO_FUNCTION
9  //#define WK_FlowControl_FUNCTION
0  //#define WK_WORK_KTHREAD
1  //#define WK_RS485_FUNCTION
2  //#define WK_RSTGPIO_FUNCTION

```

上面的宏定义是一些功能接口：

```

#define WK_FIFO_FUNCTION    //用读写 fifo 的方式读写 uart 数据，默认开启
//#define WK_FlowControl_FUNCTION    //硬件流控功能开启，默认不开启
//#define WK_WORK_KTHREAD
//#define WK_RS485_FUNCTION    //RS485 自动收发功能开启，根据实际需求开启，
//#define WK_RSTGPIO_FUNCTION    //复位引脚开启。如果硬件设计了复位引脚控制，可以
开启

```

4.2.4 芯片型号修改

WK2XXX 系列芯片在主接口相同的情况下，驱动是可以兼容的，但是还有还是存在一些差异，比如扩展子串口的数量，我们可以通过修改芯片类型结构体去实现，入下图红色框中的

```

52
53 static int wk2xxx_probe(struct spi_device *spi)
54 {
55     const struct sched_param sched_param = { .sched_priority = MAX_RT_PRIO / 2 };
56     //const struct sched_param sched_param = { .sched_priority = 100 / 2 };
57     uint8_t i;
58     int ret, irq;
59     uint8_t dat[1];
60     static struct wk2xxx_port *s;
61     #ifdef _DEBUG_WK_FUNCTION
62     printk(KERN_ALERT "%s!--in--\n", __func__);
63     #endif
64
65     /* Setup SPI bus */
66     spi->bits_per_word = 8;
67     /* only supports mode 0 on WK2124 */
68     spi->mode = spi->mode ? : SPI_MODE_0;
69     spi->max_speed_hz = spi->max_speed_hz ? : 10000000;
70     ret = spi_setup(spi);
71     if (ret)
72     {
73         return ret;
74     }
75     /* Alloc port structure */
76     s = devm_kzalloc(&spi->dev, sizeof(*s) + sizeof(struct wk2xxx_one) * NR_PORTS, GFP_KERNEL);
77     if (!s) {
78         printk(KERN_ALERT "wk2xxx_probe(devm_kzalloc) fail.\n");
79         return -ENOMEM;
80     }
81     s->spi_wk = spi;
82     s->devtype = &wk2124_devtype;
83     dev_set_drvdata(&spi->dev, s);
84     #ifdef WK_RSTGPIO_FUNCTION
85     wk2xxx_spi_extgpio_parse_dt(&spi->dev, &s->rst_gpio_num);
86

```

根据芯片型号修改

可以按照下面的方式去修改

```

s->devtype=&wk2124_devtype;表示芯片是 wk2124.
s->devtype=&wk2168_devtype;表示芯片是 wk2168.
s->devtype=&wk2204_devtype;表示芯片是 wk2204.
s->devtype=&wk2132_devtype;表示芯片是 wk2132.
s->devtype=&wk2212_devtype;表示芯片是 wk2212.

```

4.3 驱动的编译

驱动可以和内核编译到一起，也可以单独编译成模块加载。我们就按照编译成模块的方式分享一遍驱动的编译过程。

4.3.1 编译前的准备工作

编译驱动以前需要搭建好交叉编译环境。其次就是要准备好编译工具（编译器），最后就是先要编译好开发平台内核。以上这些网上都有详细的资料，这里再在介绍。

其次准备驱动源文件和 makefile 文件。

驱动配套的 Makefile 文件如下，请参考

```
ARCH:= arm64
```

```

MVT00L_PREFIX = /home/xxw/firefly_pro/Firefly_Linux_SDK_v1.0/prebuilts/gcc/linux-x86/aarch64/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu-
CROSS_COMPILE= $(MVT00L_PREFIX)
KDIR := /home/xxw/firefly_pro/Firefly_Linux_SDK_v1.0/aio3399-kernel
TARGET          =wk2xxx_spi
EXEC = $(TARGET)
obj-m :=$(TARGET).o
PWD :=$(shell pwd)
all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
    rm -rf *.o *~core.depend.*.cmd *.ko *.mod.c .tmp_versions $(TARGET)

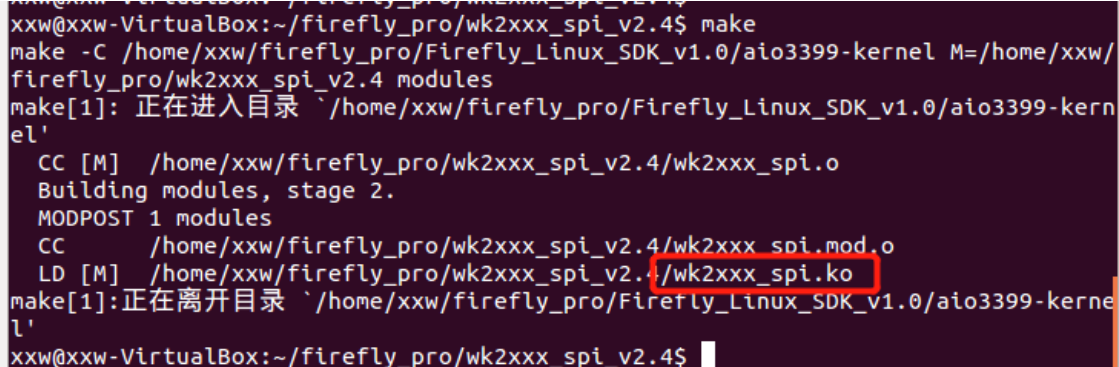
```

注意下面通常是需要修改的：编译器路径和内核文件路径

MVT00L_PREFIX：指向编译器的路径
KDIR：内核文件路径

4.3.2 编译驱动

正常情况下，把驱动和 Makefile 文件放交叉编译环境中，执行 make 指令，就会成功编译出对应的驱动模块文件 wk2xxx_spi.ko 文件。



```

xxw@xxw-VirtualBox:~/firefly_pro/wk2xxx_spi_v2.4$ make
make -C /home/xxw/firefly_pro/Firefly_Linux_SDK_v1.0/aio3399-kernel M=/home/xxw/firefly_pro/wk2xxx_spi_v2.4 modules
make[1]: 正在进入目录 `/home/xxw/firefly_pro/Firefly_Linux_SDK_v1.0/aio3399-kernel'
  CC [M] /home/xxw/firefly_pro/wk2xxx_spi_v2.4/wk2xxx_spi.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/xxw/firefly_pro/wk2xxx_spi_v2.4/wk2xxx_spi.mod.o
  LD [M]  /home/xxw/firefly_pro/wk2xxx_spi_v2.4/wk2xxx_spi.ko
make[1]:正在离开目录 `/home/xxw/firefly_pro/Firefly_Linux_SDK_v1.0/aio3399-kernel'
xxw@xxw-VirtualBox:~/firefly_pro/wk2xxx_spi_v2.4$

```

但是编译过程当中通常都会遇见一些问题，这些问题主要是和平台差异相关的问题。下面就把我们遇见过的一些问题分享一下。

4.3.2.1 Kthread_work 相关定义问题

在头文件 `#include <linux/kthread.h>` 下，定义了相关的函数和结构体。入下图

```

extern void __init_kthread_worker(struct kthread_worker *worker,
                                const char *name, struct lock_class_key *key);

#define init_kthread_worker(worker) \
do { \
    static struct lock_class_key __key; \
    __init_kthread_worker((worker), ("#worker")->lock, &__key) \
} while (0)

#define init_kthread_work(work, fn) \
do { \
    memset((work), 0, sizeof(struct kthread_work)); \
    INIT_LIST_HEAD(&(work)->node); \
    (work)->func = (fn); \
} while (0)

int kthread_worker_fn(void *worker_ptr);

bool queue_kthread_work(struct kthread_worker *worker,
                       struct kthread_work *work);
void flush_kthread_work(struct kthread_work *work);
void flush_kthread_worker(struct kthread_worker *worker);

```

在实际调试中，我们发现不同平台下，对于这些函数定义存在差异。所以我们在头文件中定义了宏定义

```
#define WK_WORK_KTHREAD
```

通过条件编译的方式来切换。

如果在编译的时候，出现如下的一些编译错误，就可以通过宏定义

#define WK_WORK_KTHREAD 来调整。

```

xxw@xxw-VirtualBox:~/firefly_pro/wk2xxx_spi_v2.4$ make
make -C /home/xxw/firefly_pro/Firefly_Linux_SDK_v1.0/aio3399-kernel M=/home/xxw/
firefly_pro/wk2xxx_spi_v2.4 modules
make[1]: 正在进入目录 `/home/xxw/firefly_pro/Firefly_Linux_SDK_v1.0/aio3399-kern
el'
CC [M] /home/xxw/firefly_pro/wk2xxx_spi_v2.4/wk2xxx_spi.o
/home/xxw/firefly_pro/wk2xxx_spi_v2.4/wk2xxx_spi.c: In function 'wk2xxx_irq':
/home/xxw/firefly_pro/wk2xxx_spi_v2.4/wk2xxx_spi.c:789:9: error: implicit declar
ation of function 'kthread_queue_work' [-Werror=implicit-function-declaration]
    ret=kthread_queue_work(&s->kworker, &s->irq_work);
    ^~~~~~
/home/xxw/firefly_pro/wk2xxx_spi_v2.4/wk2xxx_spi.c: In function 'wk2xxx_shutdown
':
/home/xxw/firefly_pro/wk2xxx_spi_v2.4/wk2xxx_spi.c:1149:9: error: implicit decla
ration of function 'kthread_flush_work' [-Werror=implicit-function-declaration]
    kthread_flush_work(&one->start_tx_work);
    ^~~~~~
/home/xxw/firefly_pro/wk2xxx_spi_v2.4/wk2xxx_spi.c: In function 'wk2xxx_probe':
/home/xxw/firefly_pro/wk2xxx_spi_v2.4/wk2xxx_spi.c:1629:5: error: implicit decla
ration of function 'kthread_init_worker' [-Werror=implicit-function-declaration]
    kthread_init_worker(&(s->kworker));
    ^~~~~~
/home/xxw/firefly_pro/wk2xxx_spi_v2.4/wk2xxx_spi.c:1630:2: error: implicit decla
ration of function 'kthread_init_work' [-Werror=implicit-function-declaration]
    kthread_init_work(&s->irq_work, wk2xxx_list);
    ^~~~~~
/home/xxw/firefly_pro/wk2xxx_spi_v2.4/wk2xxx_spi.c: In function 'wk2xxx_remove':
/home/xxw/firefly_pro/wk2xxx_spi_v2.4/wk2xxx_spi.c:1722:5: error: implicit decla
ration of function 'kthread_flush_worker' [-Werror=implicit-function-declaration]
    kthread_flush_worker(&s->kworker);
    ^~~~~~
cc1: some warnings being treated as errors
make[2]: *** [/home/xxw/firefly_pro/wk2xxx_spi_v2.4/wk2xxx_spi.o] 错误 1
make[1]: *** [_module_/home/xxw/firefly_pro/wk2xxx_spi_v2.4] 错误 2
make[1]: 正在离开目录 `/home/xxw/firefly_pro/Firefly_Linux_SDK_v1.0/aio3399-kernel'
make: *** [all] 错误 2
xxw@xxw-VirtualBox:~/firefly_pro/wk2xxx_spi_v2.4$

```


4.3.2.2 Port.flags 赋值问题

struct uart_port

在函数 `static int wk2xxx_probe(struct spi_device *spi)` 中给 struct `uart_port` 初始化的时候。可能会出现如下代码编译不过的情况。

```
s->p[i].port.iotype = SERIAL_IO_PORT;
s->p[i].port.flags  = ASYNC_BOOT_AUTOCONF;
```

如果出现 `port.flags` 这个标志位编译不过，那么可以如下操作：

1、替换。如下图

```
s->p[i].port.iotype = SERIAL_IO_PORT;
s->p[i].port.flags  = ASYNC_BOOT_AUTOCONF;
//s->p[i].port.iotype = UPIO_PORT;
//s->p[i].port.flags  = UPF_FIXED_TYPE | UPF_LOW_LATENCY;
```

用红色框中的标志代替上面的。

或者用 `UPF_BOOT_AUTOCONF` 替换 `ASYNC_BOOT_AUTOCONF`,如下所示

```
s->p[i].port.flags  = UPF_BOOT_AUTOCONF;
//s->p[i].port.flags  = ASYNC_BOOT_AUTOCONF;
```

3、如果方法 1 不行，那么只有参考平台串口驱动中该标志位的赋值。

说明：struct `uart_port` 该结构体定义在 `include/linux/serial_core.h`

4.3.2.3 MAX_RT_PRIO 无定义问题

```
const struct sched_param sched_param = { .sched_priority = MAX_RT_PRIO / 2 };
```

`MAX_RT_PRIO` 这个参数有可能定义的头文件找不到，导致编译不过，可以直接用 100 代替该参数。如下图所示：

```
static int wk2xxx_probe(struct spi_device *spi)
{
    const struct sched_param sched_param = { .sched_priority = MAX_RT_PRIO / 2 };
    //const struct sched_param sched_param = { .sched_priority = 100 / 2 };
    uint8_t i;
    int ret, irq;
    uint8_t dat[1];
    static struct wk2xxx_port *s;
    #ifdef _DEBUG_WK_FUNCTION
        printk(KERN_ALERT "%s!--in--\n", __func__);
    #endif
}
```

4.4 驱动调试

在驱动编译完成以后，会生成 wk2xxx_spi.ko 文件。

4.4.1 加载驱动

把 wk2xxx_spi.ko 文件 push 到开发板。执行如下命令加载驱动模块

insmod wk2xx_spi.ko 如下：

```
wk2xxx_spi: loading out-of-tree module to kernel.
wk2xxx_init: SPI driver for spi to Uart chip WK2XXX, etc.
wk2xxx_init: V2.4.2 On 2022.05.05

wk2xxx_probe(0x30)  GENA = 0x30
wk2xxx_probe(0x35)  GENA = 0x35
wk2xxx_probe(0x3f)  GENA = 0x3F
wk2xxx_irq_gpio: 146, irq: 220
wk2xxx_serial_init.
spi0.0: ttysWK0 at I/O 0x1 (irq = 0, base_baud = 691200) is a wk2xxx
uart_add_one_port: 1. status= 0x0
spi0.0: ttysWK1 at I/O 0x2 (irq = 0, base_baud = 691200) is a wk2xxx
uart_add_one_port: 2. status= 0x0
spi0.0: ttysWK2 at I/O 0x3 (irq = 0, base_baud = 691200) is a wk2xxx
uart_add_one_port: 3. status= 0x0
spi0.0: ttysWK3 at I/O 0x4 (irq = 0, base_baud = 691200) is a wk2xxx
uart_add_one_port: 4. status= 0x0
devm_request_irq success. ret=0.
```

驱动加载成功，会在/dev/目录下出现对应的串口节点，如图：

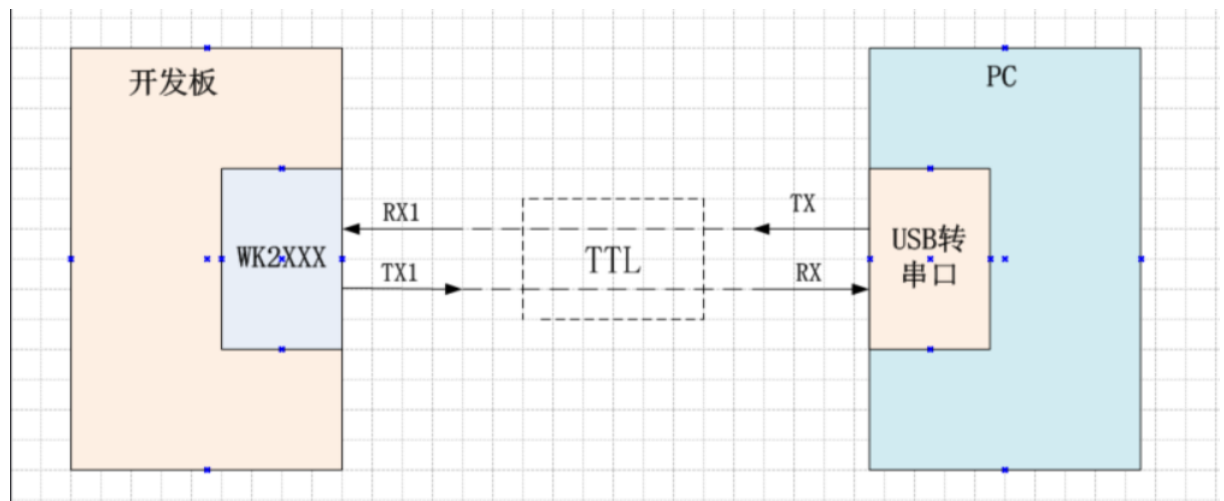
```
crw----- 1 root root 207, 5 Feb 20 20:25 /dev/ttysWK0
crw----- 1 root root 207, 6 Feb 20 20:25 /dev/ttysWK1
crw----- 1 root root 207, 7 Feb 20 20:25 /dev/ttysWK2
crw----- 1 root root 207, 8 Feb 20 20:25 /dev/ttysWK3
```

ttysWK* 这些节点就可以当做标准串口设备节点编程使用。

4.4.2 测试

我们下面介绍一些简单的测试方法.简单的测试数据发送和数据接收。

我们在 linux 开发板端采用命令的方式操作串口设备节点。然后用 USB 转串口工具，一端连接 WK2XXX 芯片的 UART1,一端连接到 PC 端，PC 端用串口助手接收和发送数据。硬件连接示意图如下



1、串口发送字符串

在开发板端使用命令 `echo "123456abcdefg">/dev/ttyWK1`

该命令的默认波特率是 9600。演示结果如下。

2、串口接收字符和发送字符

首先是 cat 设备节点，如下图。Cat 设备节点入下图。

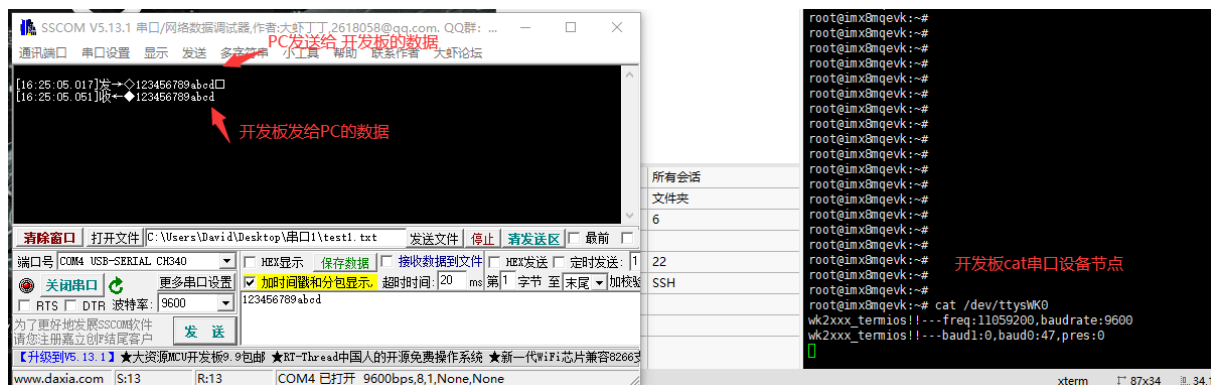
```

root@imx8mqevk:~# cat /dev/ttyWK0
wk2xxx_termios!!--freq:11059200,baudrate:9600
wk2xxx_termios!!--baud1:0,baud0:47,pres:0

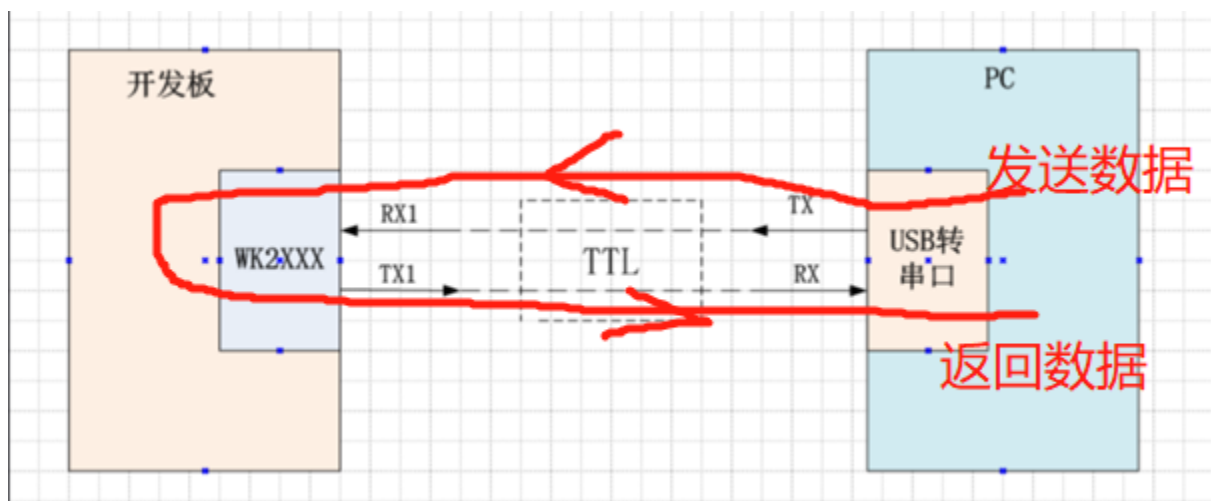
```

cat 设备节点有如下功能。首先是会打开对应的串口设备节点（波特率 9600），准备接收字符数据。其次是把接收的字符数据然后通过串口设备节点发送出来，这一点是需要注意的。所以 cat 命令不光有数据的接收，也有数据的发送。

演示操作截图如下：



如上图所示：该测试过程的数据流向与过程如下：PC 上的 USB 串口通过 TX 发送出数据----->开发板上的 WK2xxx 接收到数据，并传给系统-----》开发板系统把收到的数据，然后通过 WK2xxx 芯片发送出来-----》PC 接收到返回的数据。示意图如下：



5 特别申明

- 成都为开微电子有限公司（简称为开微电子）保有在不事先通知而修改这份文档的权利。在使用本产品前请联系为开微电子获取该新产品说明的最新版本。

- 为开微电子认为提供的信息是准确可信的。尽管这样，为开微电子对文档中可能出现的错误不承担任何责任。
- 对于使用该器件引起的专利纠纷及第三方侵权为开微电子不承担任何责任。
- 为开微电子的产品不建议应用于生命相关的设备和系统，在使用该器件中因为设备或系统运转失灵而导致的损失为开微电子不承担任何责任。
- 为开微电子对本手册拥有版权等知识产权，受法律保护。未经为开微电子许可，任何单位及个人不得以任何方式或理由对本手册进行使用、复制、修改、抄录、传播等。