

EC2X-QuecOpen

SPI 开发指导

LTE 系列

版本: EC2X-QuecOpen_SPI_开发指导_V1.0

日期: 2018-03-09

状态: 临时文件

上海移远通信技术股份有限公司始终以为客户提供最及时、最全面的服务为宗旨。如需任何帮助，请随时联系我司上海总部，联系方式如下：

上海移远通信技术股份有限公司
上海市徐汇区虹梅路 1801 号宏业大厦 7 楼 邮编：200233
电话：+86 21 51086236 邮箱：info@quectel.com

或联系我司当地办事处，详情请登录：

<http://quectel.com/cn/support/sales.htm>

如需技术支持或反馈我司技术文档中的问题，可随时登陆如下网址：

<http://quectel.com/cn/support/technical.htm>

或发送邮件至：support@quectel.com

前言

上海移远通信技术股份有限公司提供该文档内容用以支持其客户的产品设计。客户须按照文档中提供的规范、参数来设计其产品。由于客户操作不当而造成的人身伤害或财产损失，本公司不承担任何责任。在未声明前，上海移远通信技术股份有限公司有权对该文档进行更新。

版权申明

本文档版权属于上海移远通信技术股份有限公司，任何人未经我司允许而复制转载该文档将承担法律责任。

版权所有 ©上海移远通信技术股份有限公司 2018，保留一切权利。

Copyright © Quectel Wireless Solutions Co., Ltd. 2018.

文档历史

修订记录

版本	日期	作者	变更表述
1.0	2018-03-09	高飞虎	初始版本

目录

文档历史	2
目录	3
表格索引	4
图片索引	5
1 引言	6
2 EC20 R2.1-QuecOpen SPI 说明	7
3 硬件电路设计推荐	8
3.1 标准 4 线 SPI 外接 flash 的参考设计	8
3.2 扩展 6 线 SPI 外接 mcu 电路参考设计	8
4 驱动层及设备树软件适配	10
4.1 SPI 管脚使用	10
4.1.1 标准 4 线 spi 管脚使用:	10
4.1.2 扩展 6 线 spi 管脚使用:	11
4.1.2.1 管脚使用	11
4.1.2.2 主从分别发起请求的流程	12
4.2 SPI 软件配置方法	13
4.2.1 SPI 控制器配置说明	13
4.2.2 SPI 设备驱动的使用	14
4.2.2.1 标准 4 线 spi 设备驱动说明	14
4.2.2.2 扩展 6 线 spi 设备驱动说明	15
5 QuecOpen 应用层 API	17
5.1 用户编程说明	17
5.2 SPI API 介绍	17
5.2.1 标准 4 线 spi 操作 API	17
5.2.2 扩展 6 线 spi 应用层操作方式	18
6 SPI 功能测试验证	19
6.1 example 介绍及编译	19
6.1.1 标准 4 线 spi 应用说明	19
6.1.2 扩展 6 线 spi 应用说明	20
6.2 功能测试	20
6.2.1 标准 4 线 spi 应用测试	20
6.2.2 扩展 6 线 spi 应用测试	22
7 SPI 驱动调试方法	23
7.1 一般调试方法	23
7.2 使用 kernel tracer 进行调试	24

表格索引

TABLE 1: 管脚功能复用	10
TABLE 2: 标准 4 线 SPI 管脚使用	10
TABLE 3: 扩展 6 线 SPI 管脚使用	11

图片索引

FIGURE 1: EC20 R2.1-QUECOPEN 模块 SPI 框架	7
FIGURE 2: 标准 4 线 SPI 外接 FLASH 的参考设计	8
FIGURE 3: 扩展 6 线 SPI 外接 MCU 1.8V 直连电路参考设计	9

1 引言

文档从用户开发角度出发，介绍了电路设计，软件驱动层，软件应用层等；可以帮助客户简易而快速的进行开发。

2 EC20 R2.1-QuecOpen SPI 说明

1. 模块默认提供一路 SPI 接口，只支持主模式，默认支持 DMA;
2. 支持的最大时钟频率为 50MHz;
3. 一个 SPI 控制器支持最多四个片选信号(CS);
4. 下图为 EC20 R2.1-QuecOpen 模块内 SPI 框架;

其中 SPI adapter 适配层为 spi 控制器的设备树配置及驱动;

SPI slave 为 QuecOpen 提供的 spi 设备驱动，分为标准 4 线和扩展 6 线两种;

4 线：通常用来连接 spi flash, lcd 等，由模块发起请求;

6 线：通常用来与 mcu 通信，模块、MCU 均可发起请求，相比串口通信也更高速度;

+

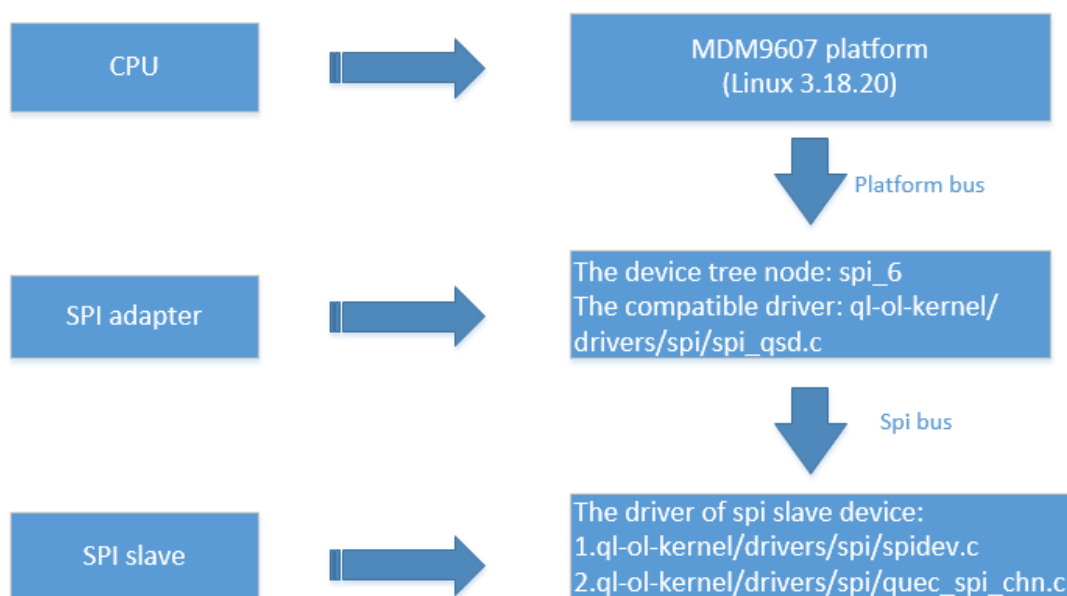


Figure 1: EC20 R2.1-QuecOpen 模块 SPI 框架

3 硬件电路设计推荐

3.1 标准 4 线 SPI 外接 flash 的参考设计

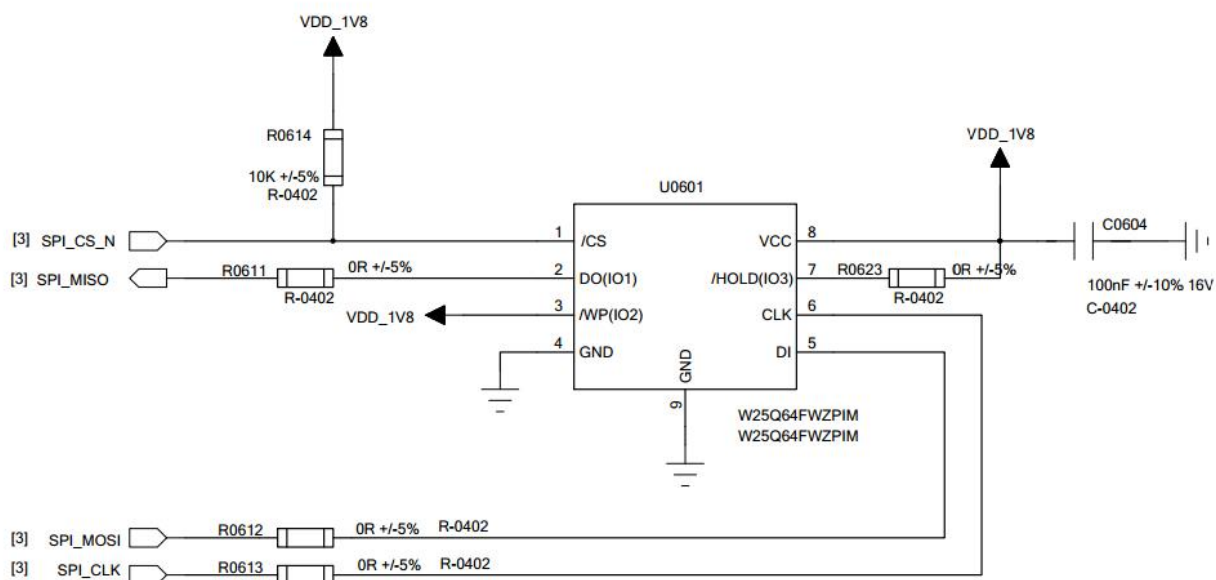


Figure 2: 标准 4 线 SPI 外接 flash 的参考设计

3.2 扩展 6 线 SPI 外接 mcu 电路参考设计

下图为的 SPI 外接 1.8V 的 MCU 电路参考设计：

若 MCU 端为 3.3V，则需要增加电平转换芯片；

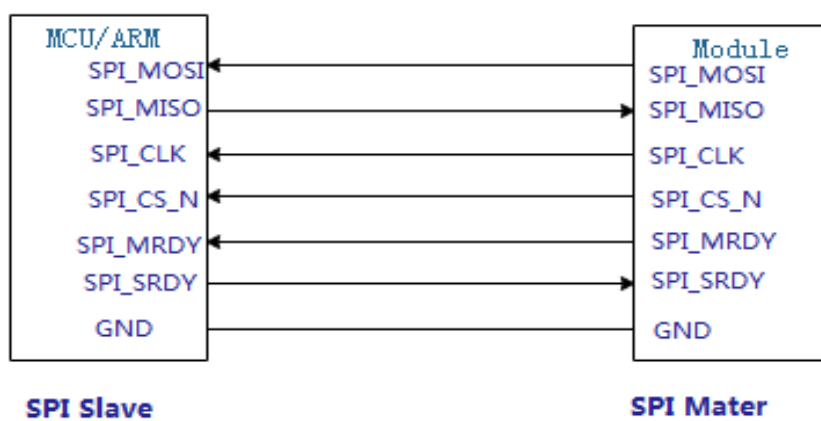


Figure 3: 扩展 6 线 SPI 外接 mcu 1.8v 直连电路参考设计

4 驱动层及设备树软件适配

4.1 SPI 管脚使用

1. 以下表格中，非默认的复用功能需要在软件配置后才有效，请参考对应的功能章节进行软件配置；
2. 具体管脚使用参考《Quectel_EC20 R2.1_QuecOpen_GPIO_Assignment_Speadsheet》。

Table 1: 管脚功能复用

引脚名	引脚号	模式 1 (默认)	模式 2	模式 3	复位状态 ¹⁾	中断唤醒 ²⁾	备注
SPI_CS_N	37	SPI_CS_N_BLS P6	GPIO_22	UART_RTS_BLS P6	B-PD,L	YES	
SPI_MOSI	38	SPI_MOSI_BLS P6	GPIO_20	UART_TXD_BLS P6	B-PD,L	YES	
SPI_MISO	39	SPI_MISO_BLS P6	GPIO_21	UART_RXD_BLS P6	B-PD,L	YES	
SPI_CLK	40	SPI_CLK_BLS P6	GPIO_23	UART_CTS_BLS P6	B-PU,H	NO	BOOT_CONFIG_4

4.1.1 标准 4 线 spi 管脚使用:

Table 2: 标准 4 线 spi 管脚使用

引脚名	引脚号	I/O	描述	备注
SPI_CS_N	37	DO	SPI 片选信号	1.8V 电源域，不用则悬空。
SPI_MOSI	38	DO	SPI 数据输出	1.8V 电源域，不用则悬空。
SPI_MISO	39	DI	SPI 数据输入	1.8V 电源域，不用则悬空。
SPI_CLK	40	DO	SPI 时钟	1.8V 电源域，不用则悬空。

4.1.2 扩展 6 线 spi 管脚使用:

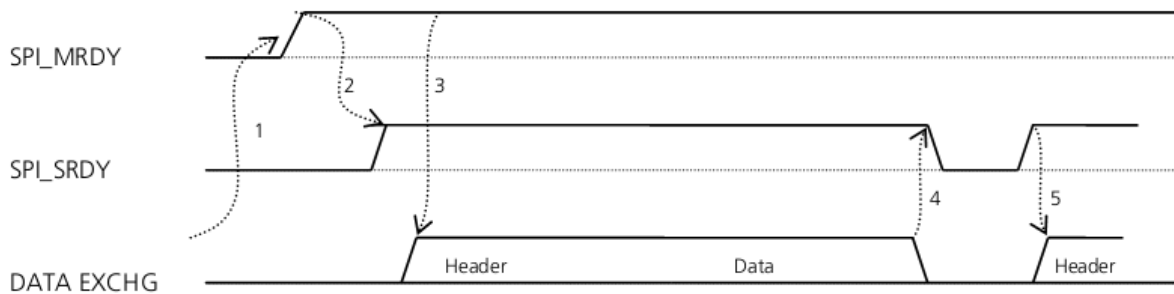
4.1.2.1 管脚使用

Table 3: 扩展 6 线 spi 管脚使用

引脚名	引脚号	I/O	描述	备注
SPI_CS_N	37	DO	SPI 片选信号	1.8V 电源域，不用则悬空。
SPI_MOSI	38	DO	SPI 数据输出	1.8V 电源域，不用则悬空。
SPI_MISO	39	DI	SPI 数据输入	1.8V 电源域，不用则悬空。
SPI_CLK	40	DO	SPI 时钟	1.8V 电源域，不用则悬空。
SPI_MRDY	用户选择	DO	模块输出信号，空闲为低；当模块要输出数据时，驱动自动拉高该 PIN	
SPI_SRDY	用户选择	DI	SPI Slave ready 信号，空闲为低；当 SPI Slave 准备好接收/发送数据时，拉高该 PIN	

4.1.2.2 主从分别发起请求的流程

4G 模块发起请求:



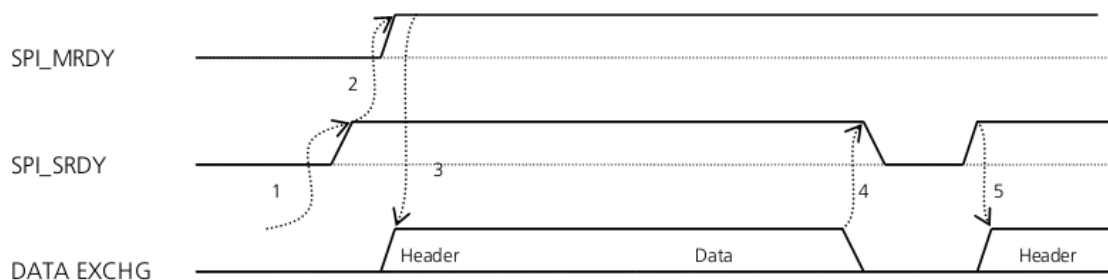
4G 模块流程:

1. 驱动自动拉高 SPI_MRDY 通知 SPI Slave。
2. 判断 SPI_SRDY 是否为高，否则等待 SPI_SRDY 的上升沿中断。
3. 收到 slave 上升沿，开始 SPI 传输。
4. 传输完毕，如果要继续发送数据，则保持 SPI_MRDY 为高，并继续第二步，否则拉低 SPI_MRDY。

SPI Slave 流程:

1. 收到 SPI_MRDY 上升沿中断，表示 4G 模块需要发送数据。
2. 准备好 SPI 传输，并拉高 SPI_SRDY 通知 4G 模块开始 SPI 传输。
3. 等待 SPI 传输结束，并拉低 SPI_SRDY。
4. 如果 SPI_MRDY 为高，再继续第二步。

SPI Slave 发起请求:



SPI Slave 流程:

1. 准备好 SPI 传输，并拉高 SPI_SRDY。
2. 等待 SPI 传输结束，并拉低 SPI_SRDY。
3. 如果继续发送数据，则继续第 1 步。

4G 模块流程:

1. 收到 SPI_SRDY 上升沿中断，表示从机要发送数据。
2. 拉高 SPI_MRDY，并开始 SPI 传输。
3. 等待传输结束，并拉低 SPI_MRDY。

4.2 SPI 软件配置方法

4.2.1 SPI 控制器配置说明

Linux 的 SPI 体系结构分为 3 个组成部分：

SPI 核心：SPI 核心提供了 SPI 总线驱动和设备驱动的注册，注销方法，SPI 通信方法，与具体控制器无关的代码以及探测设备，检测设备地址的上层代码等。

SPI 总线（控制器）驱动：是对 SPI 硬件体系控制器端的实现，控制器由 CPU 控制，也可以直接集成在 CPU 内部。

SPI 设备驱动：即客户的 SPI 从设备驱动，是对 SPI 硬件体系结构中设备端的实现，设备一般挂接在受 CPU 控制的 SPI 控制器上，通过 SPI 控制器与 CPU 交换数据。

以上三部分，用户一般只需要关心和修改 SPI 设备驱动；

SPI 总线驱动：即 SPI 控制器，mdm9607 平台使用的是设备树节点 `spi-qup-v2`；其硬件参数配置，如所兼容的 driver，引脚的选择，寄存器地址，CLK，中断号，以及系统休眠和工作时的管脚配置等 QuecOpen 都已经做好了，用户不需要关心和修改；

```
spi_6: spi@78ba000 {
    compatible = "qcom,spi-qup-v2";
    #address-cells = <1>;
    #size-cells = <0>;
    reg-names = "spi_physical", "spi_bam_physical";
    reg = <0x78ba000 0x600>,
        <0x7884000 0x2b000>;
    interrupt-names = "spi_irq", "spi_bam_irq";
    interrupts = <0 100 0>, <0 238 0>;
    spi-max-frequency = <19200000>;
    pinctrl-names = "spi_default", "spi_sleep";
    pinctrl-0 = <&spi6_default &spi6_cs0_active>;
    pinctrl-1 = <&spi6_sleep &spi6_cs0_sleep>;
    clocks = <&clock_gcc clk_gcc_blspi_ahb_clk>,
        <&clock_gcc clk_gcc_blspi_qup6_spi_apps_clk>;
    clock-names = "iface_clk", "core_clk";
    qcom,infinite-mode = <0>;
    qcom,use-bam;
    qcom,use-pinctrl;
    qcom,ver-reg-exists;
    qcom,bam-consumer-pipe-index = <22>;
    qcom,bam-producer-pipe-index = <23>;
    qcom,master-id = <86>;
}
```

另外，除非用户在 mdm9607 平台上根本不使用 SPI 控制器，那么用户可以用以下方式关闭 SPI 控制器：以下方法至少执行一个

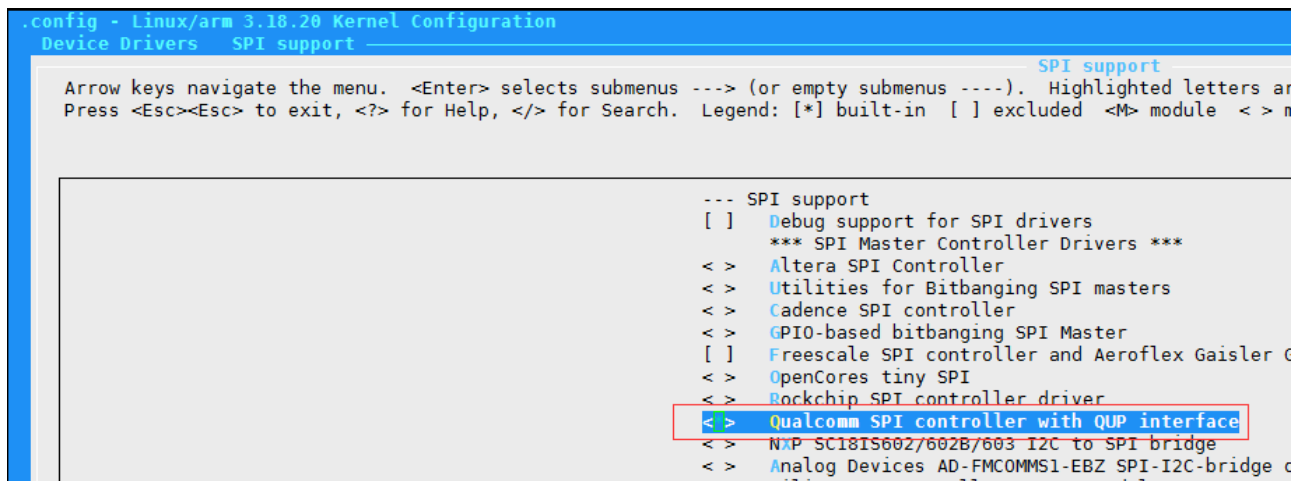
1. 关闭控制器设备节点：

```
--- a/ql-ol-kernel/arch/arm/boot/dts/qcom/mdm9607-mtp.dtsi
+++ b/ql-ol-kernel/arch/arm/boot/dts/qcom/mdm9607-mtp.dtsi
@@ -48,7 +48,7 @@
 //2016-01-19, comment out by jun.wu, remove UART3 && spi_1 from device tree

 &spi_6 {
-    status = "ok";
+    status = "disabled";
 };
```

2. make kernel_menuconfig 去掉 SPI_QUP 内核选项;

```
~/MDM9x07/SDK_FAG0130/ql-ol-sdk$ make kernel_menuconfig
MDM9x07/SDK_FAG0130/ql-ol-sdk/ql-ol-kernel ; make ARCH=arm mdm9607-perf_defcon
g directory `/home/gale/MDM9x07/SDK_FAG0130/ql-ol-sdk/ql-ol-kernel'
g directory `/home/gale/MDM9x07/SDK_FAG0130/ql-ol-sdk/ql-ol-kernel/build'
/basic/fixdep
file
```



```
.config - Linux/arm 3.18.20 Kernel Configuration
Device Drivers  SPI support  SPI support

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are
Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ] excluded <M> module <> n

--- SPI support
[ ] Debug support for SPI drivers
*** SPI Master Controller Drivers ***
<> Altera SPI Controller
<> Utilities for Bitbanging SPI masters
<> Cadence SPI controller
<> GPIO-based bitbanging SPI Master
[ ] Freescale SPI controller and Aeroflex Gaisler c
<> OpenCores tiny SPI
<> Rockchip SPI controller driver
<> Qualcomm SPI controller with QUP interface
<> NXP SC18IS602/602B/603 I2C to SPI bridge
<> Analog Devices AD-FMCOMMS1-EBZ SPI-I2C-bridge c
<> Wilink SPI controller driver module
```

4.2.2 SPI 设备驱动的使用

QuecOpen 模块提供了两种 spi 设备驱动，分为 4 线和 6 线的，默认以 kernel module 方式编译放在 rootfs 的/usr/lib/modules/3.18.20/kernel/drivers/spi/路径下，用户需要 insmod;

4.2.2.1 标准 4 线 spi 设备驱动说明

4 线：驱动位于 ql-ol-kernel/drivers/spi/spidev.c，通常用来连接 spi flash，lcd 等，由模块发起请求，此驱动不使用设备树传参方式，直接 insmod 带入参数更灵活；

内核模块加载时支持的参数：

busnum: spi 控制器编号为 6，由下图配置决定，此参数必须传入，否则 spi 从设备会找不到控制器而导致加载失败；

```
aliases {
    /* smdttty devices */
    smd7 = &smdttty_data1;
    smd8 = &smdttty_data4;
    smd9 = &smdttty_data2;
    /*smd11 = &smdttty_data11;*/ /*modified by max.tang@20
    smd21 = &smdttty_data21;
    smd36 = &smdttty_loopback;
    /* spi device */
    /*spi1 = &spi_1;*/ //2016-01-19, comment out by jun.w
    spi6 = &spi_6;
    i2c2 = &i2c_2;
    i2c4 = &i2c_4; //add zahi.song
    sdhc2 = &sdhc_2; /* SDC2 SD card slot */
};
```

chipselect: 片选支持 0, 1, 2, 3, 此参数必须传入, 否则 spi 设备会注册失败;

spimode: Spi 支持 4 种工作模式, 其值为相位 (CPHA 0x01) 和极性 (CPOL 0x02) 的按位或, 驱动代码默认使用 SPI_MODE_3 模式, 用户可以在 insmod 时修改;

时钟极性 CPOL: 即 SPI 空闲时, 时钟信号 SCLK 的电平 (1:空闲时高电平; 0:空闲时低电平)

时钟相位 CPHA: 即 SPI 在 SCLK 第几个边沿开始采样 (0:第一个边沿开始; 1:第二个边沿开始)

maxspeed: 可选参数, 驱动默认为 9.6Mhz, 实际支持的最大值由 spi 控制器的配置决定, 与理论最大值并不冲突; 支持的可选值 960000, 4800000, 9600000, 16000000, 19200000, 25000000, 50000000

bufsiz: 可选参数, 设定 spi 传输队列中每个 message 的大小, 默认值 4096Bytes, 用户可根据自己传输每次数据量的大小来设定;

加载命令: `insmod /lib/modules/3.18.20/kernel/drivers/spi/spidev.ko busnum=6 chipselect=0 spimode=0 maxspeed=19200000`

加载成功确认:

```
root@mdm9607-perf:~# insmod /lib/modules/3.18.20/kernel/drivers/spi/spidev.ko bu
snum=6 chipselect=0 spimode=0 maxspeed=19200000
root@mdm9607-perf:~# lsmod
spidev 6473 0 - Live 0xbf03a000
shortcut_fe_cm 6612 0 - Live 0xbf035000 (0)
shortcut_fe_ipv6 57017 1 shortcut_fe_cm, Live 0xbf023000 (0)
shortcut_fe 56314 1 shortcut_fe_cm, Live 0xbf011000 (0)
embms_kernel 5481 2 - Live 0xbf00c000 (0)
snd_soc_alc5616 28819 1 - Live 0xbf000000
root@mdm9607-perf:~# ls /dev/spidev6.0
/dev/spidev6.0
```

4.2.2.2 扩展 6 线 spi 设备驱动说明

6 线: 驱动位于 ql-ol-kernel/drivers/spi/quec_chn_spi.c, 通常用来与 mcu 通信, 模块, MCU 均可发起请求, 相比串口通信更高速, 此驱动不使用设备树传参方式, 直接从 insmod 命令行带入客户参数更灵活;

内核模块加载时支持的参数:

busnum: spi 控制器编号为 6, 由下图配置决定; 可选参数, 驱动默认值为 6;


```
aliases {
    /* smdttty devices */
    smd7 = &smdttty_data1;
    smd8 = &smdttty_data4;
    smd9 = &smdttty_data2;
    /*smd11 = &smdttty_data11;*/ /*modified by max.tang@20
    smd21 = &smdttty_data21;
    smd36 = &smdttty_loopback;
    /* spi device */
    /*spi1 = &spi_1;*/ //2016-01-19, comment out by jun.w
    spi6 = &spi_6;
    i2c2 = &i2c_2;
    i2c4 = &i2c_4; //add zahi.song
    sdhc2 = &sdhc_2; /* SDC2 SD card slot */
};
```

chipselect: 片选支持 0,1,2,3, 可选参数, 驱动默认值为 0;

spi_mode: Spi 支持 4 种工作模式, 其值为相位 (CPHA 0x01) 和极性 (CPOL 0x02) 的按位或, 驱动代码默认使用 SPI_MODE_0 模式, 用户可以在 insmod 时修改;

时钟极性 CPOL: 即 SPI 空闲时, 时钟信号 SCLK 的电平 (1:空闲时高电平; 0:空闲时低电平)

时钟相位 CPHA: 即 SPI 在 SCLK 第几个边沿开始采样 (0:第一个边沿开始; 1:第二个边沿开始)

speed_hz: 可选参数, 驱动默认为 9.6Mhz, 实际支持的最大值由 spi 控制器的配置决定, 与理论最大值并不冲突; 支持的可选值 960000, 4800000, 9600000, 16000000, 19200000, 25000000, 50000000

frame_size: 可选参数, 设定 spi 传输队列中每个 message 的大小, 默认值 512Bytes, 用户可根据自己传输每次数据量的大小来设定;

gpiomodemready: 设置 SPI_MRDY 管脚, 驱动代码默认使用 gpio34, 可传参修改;

gpiomcuready: 设置 SPI_SRDY 管脚, 驱动代码默认使用 gpio52, 可传参修改;

加载命令: insmod /lib/modules/3.18.20/kernel/drivers/spi/quec_spi_chn.ko speed_hz=19200000

gpiomodemready=38 gpiomcuready=34

加载成功确认:

```
root@mdm9607-perf:~# lsmod
quec_spi_chn 9069 0 - Live 0xbf03a000
shortcut_fe_cm 6612 0 - Live 0xbf035000 (0)
shortcut_fe_ipv6 57017 1 shortcut_fe_cm, Live 0xbf023000 (0)
shortcut_fe 56314 1 shortcut_fe_cm, Live 0xbf011000 (0)
embms_kernel 5481 2 - Live 0xbf00c000 (0)
snd_soc_alc5616 28819 1 - Live 0xbf000000
root@mdm9607-perf:~# ls /dev/spi6_0 *
/dev/spi6_0_0 /dev/spi6_0_2 /dev/spi6_0_4 /dev/spi6_0_6
/dev/spi6_0_1 /dev/spi6_0_3 /dev/spi6_0_5 /dev/spi6_0_7
root@mdm9607-perf:~#
```

这里提供的 6 线 SPI 驱动虚拟出了 8 个数据通道供使用, 客户 MCU 可与 4G 模块协商各个 channel 的用途;

5 QuecOpen 应用层 API

5.1 用户编程说明

QuecOpen 项目 SDK 中提供了一套完整的用户编程接口；

参考路径：ql-ol-sdk/ql-ol-extsdk/

```
gale@eve-linux02:~/MDM9x07/SDK_FAG0130/ql-ol-sdk/ql-ol-extsdk$ ls
docs example include lib target tools
gale@eve-linux02:~/MDM9x07/SDK_FAG0130/ql-ol-sdk/ql-ol-extsdk$
```

图中的 lib 目录下包含 quectel 提供的 API 接口库；include 目录是所有 API 的头文件；example 目录是提供的按功能划分的 API 使用示例；这里我只介绍关于 SPI 相关的接口以及参考示例；

5.2 SPI API 介绍

按照第 4 节的说明，spi 设备节点已经注册成功，可以直接使用下面 API 进行应用层操作；

5.2.1 标准 4 线 spi 操作 API

在进行标准 4 线 SPI 应用程序的编写需要依赖库 libql_peripheral.a；

头文件：ql_spi.h

spi mode 枚举：spi 支持的工作模式

typedef enum

```
{
    SPIMODE0 = SPI_MODE_0,
    SPIMODE1 = SPI_MODE_1,
    SPIMODE2 = SPI_MODE_2,
    SPIMODE3 = SPI_MODE_3,
}SPI_MODE;
```

Spi clock 枚举：spi 默认支持的时钟大小

typedef enum

```
{
    S_960K = 960000,
    S_4_8M = 4800000,
    S_9_6M = 9600000,
    S_16M = 16000000,
    S_19_2M = 19200000,
    S_25M = 25000000,
    S_50M = 50000000,
}SPI_SPEED;
```

```
int Ql_SPI_Init(char *dev_name,SPI_MODE mode,uint8_t bits, SPI_SPEED speed);
```

函数功能：打开 SPI 设备并配置对应的参数

参数： dev_name: SPI 设备，需要手动加载 spidev.ko
SPI_MODE: SPI 4 种工作模式，SPI_MODE 枚举值
bits: 发送数据字的位数，支持 4,8,16,32
speed: SPI 控制器输出时钟，SPI_SPEED 枚举值

返回值：当前打开的设备文件描述符。

```
int Ql_SPI_Write_Read(int fd,uint8_t* write_buf,uint8_t* read_buf,uint32_t len);
```

函数功能：读写 SPI 数据

参数： fd :spi 设备文件描述符
write_buf:spi 写数据指针
read_buf:spi 读数据指针
len: 读写数据长度

spi 通讯是全双工的，只读可以配置 write_buf 内容为 0 ，只写可以丢弃 read_buf 内容

由于标准 spi 是读写在一个 transfer 里面，所有操作是全双工的。向 read_buf 传递一个 NULL，就是一次只写操作，会丢弃 MISO 线上的数据；同样向 write_buf 传递一个 NULL，就是一次只读操作；

返回值：成功返回 0，否则返回负值；

```
int Ql_SPI_Deinit(int fd);
```

函数功能：关闭 spi 设备

参数： fd: spi 设备文件描述符

参考：ql-ol-extsdk/example/spi/std_spi

5.2.2 扩展 6 线 spi 应用层操作方式

6 线 SPI 驱动同时虚拟出了 8 个数据通道留做备用，客户 MCU 可与 4G 模块协商各个 channel 的用途；

直接使用 open, read, write 来读写 spi 设备，并使用 select 监听设备实现异步通知；

具体参考例子: ql-ol-extsdk/example/spi/six_line

6 SPI 功能测试验证

6.1 example 介绍及编译

6.1.1 标准 4 线 spi 应用说明

ql-ol-extsdk/example/spi/std_spi 示例:

Example 示例以 SPI_MODE_0, 8bits/word, 19.2M speed 初始化设备, 向设备写 1024 个字节, 同时读取 1024 字节回来;

```
#define device    "/dev/spidev6.0"

int main(int argc, char *argv[])
{
    int fd;
    int i;
    uint8_t writebuf[1024];
    uint8_t readbuf[1024];

    fd = Ql_SPI_Init(device,SPIMODE0,8,S_19_2M);

    for(i = 0 ;i < 1024;i++)
        writebuf[i] = i%256;

    Ql_SPI_Write_Read(fd,writebuf,readbuf,1024);

    for (i = 0; i<1024; i++) {
        if (!(i % 32))
            puts("");
        printf("%.2X ", readbuf[i]);
    }
    puts("");
}
```

进入到 ql-ol-sdk/ql-ol-extsdk/example/spi/std_spi 目录, make 生成 example_spi 可执行程序, 可以编译的前提必须是之前进行了交叉编译环境的初始化

source ql-ol-crosstool/ql-ol-crosstool-env-init

```
ql-ol-sdk/ql-ol-extsdk/example/spi/std_spi$ make
a -mfloat-abi=softfp -mcpu=neon -O2 -fexpensive-optimizations
include -I/home/gale/MDM9x07/SDK_FAG0130/ql-ol-sdk/ql-ol-crosstool
-ol-sdk/ql-ol-crosstool/sysroots/armv7a-vfp-neon-oe-linux-gnueabi/
n-oe-linux-gnueabi/usr/include/data -I/home/gale/MDM9x07/SDK_FA
-I/home/gale/MDM9x07/SDK_FAG0130/ql-ol-sdk/ql-ol-crosstool/sysr
dk/ql-ol-crosstool/sysroots/armv7a-vfp-neon-oe-linux-gnueabi/us
dk/ql-ol-crosstool/sysroots/armv7a-vfp-neon-oe-linux-gnueabi/us
oe-linux-gnueabi/usr/include -I/home/gale/MDM9x07/SDK_FAG0130/q
```

6.1.2 扩展 6 线 spi 应用说明

ql-ol-extsdk/example/spi/six_line 示例:

Example 示例主线程向 spi 设备发数据, 子线程同时监听是否可读;

进入到 ql-ol-sdk/ql-ol-extsdk/example/spi/six_line 目录, make 生成

example_six_line_spi 可执行程序, 可以编译的前提必须是之前进行了交叉编译环境的初始化 source ql-ol-crosstool/ql-ol-crosstool-env-init

```
ql-ol-sdk/ql-ol-extsdk/example/spi/six_line$ make
a -mfloat-abi=softfp -mcpu=neon -O2 -fexpensive-optimizations
include -I/home/gale/MDM9x07/SDK_FAG0130/ql-ol-sdk/ql-ol-crossto
-ol-sdk/ql-ol-crosstool/sysroots/armv7a-vfp-neon-oe-linux-gnueabi
n-oe-linux-gnueabi/usr/include/data -I/home/gale/MDM9x07/SDK_F
-I/home/gale/MDM9x07/SDK_FAG0130/ql-ol-sdk/ql-ol-crosstool/sys
dk/ql-ol-crosstool/sysroots/armv7a-vfp-neon-oe-linux-gnueabi/u
dk/ql-ol-crosstool/sysroots/armv7a-vfp-neon-oe-linux-gnueabi/u
oe-linux-gnueabi/usr/include -I/home/gale/MDM9x07/SDK_FAG0130/
le/MDM9x07/SDK_FAG0130/ql-ol-sdk/ql-ol-crosstool/sysroots/armv
ol-crosstool/sysroots/armv7a-vfp-neon-oe-linux-gnueabi/usr/inc
e-linux-gnueabi/usr/include/qmi-framework -L./ -L/home/gale/MD
```

6.2 功能测试

6.2.1 标准 4 线 spi 应用测试

因为未连接 spi slave 设备, 这里我们可以直接短接 GPIO_20, GPIO_21 进行自发自收测试。

1. 加载驱动: insmod /lib/modules/3.18.20/kernel/drivers/spi/spidev.ko busnum=6 chipselect=0 spimode=0 maxspeed=19200000
2. 编译上传 example_spi 到模块
使用 adb push <example_spi 在上位机路径> <模块内部路径, 如/usrdata>

或者

使用串口协议 rz 上传

3. 若使用 OPEN_EVB, 需要用跳线帽连接 J0201 的 spi 排针
GPIO_20 连接 GPIO_21
来连通硬件通路
4. 执行 example_spi, 如下图, 收到的数据与发出去的一致。

```
root@ndm9607-perf:~# ./example_spi
< open(/dev/spidev6.0, 0_RDWR)=8 >
spi mode:0x0
bits per word: 8
max speed      : 19200000 Hz (19200 KHz)

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
root@ndm9607-perf:~#
```


6.2.2 扩展 6 线 spi 应用测试

因为未连接 spi slave 设备，这里我们可以直接短接 GPIO_20 和 GPIO_21，以及 MRDY(gpiomodernready) 和 SRDY(gpiocmuready)进行 4G 模块自发自收测试。

1. 加载驱动模块：insmod /lib/modules/3.18.20/kernel/drivers/spi/quec_spi_chn.ko speed_hz=19200000 gpiomodernready=38 gpiocmuready=34
2. 编译上传 example_six_line_spi 到模块
使用 adb push <example_six_line_spi 在上位机路径> <模块内部路径，如/usrdata>
或者
使用串口协议 rz 上传
3. 若使用 OPEN_EVB，需要用跳线帽连接 J0201 的 spi 排针
GPIO_20 连接 GPIO_21
MRDY 连接 SRDY
来连通硬件通路
4. 执行 example_six_line_spi，如下图，收到的数据与发出去的一致。

```
root@mdm9607-perf:~# ./example_six_line_spi
read 25 bytes hello,I am a six line spi
read 15 bytes test process!
read 25 bytes hello,I am a six line spi
read 15 bytes test process!
read 25 bytes hello,I am a six line spi
read 15 bytes test process!
read 25 bytes hello,I am a six line spi
read 15 bytes test process!
read 25 bytes hello,I am a six line spi
read 15 bytes test process!
```

7 SPI 驱动调试方法

以上内容足以帮助用户让设备正常工作起来，但是总是会遇到一些意外的问题，不管是用户不当操作还是代码上的问题，那么我们需要通过一些调试手段来定位问题。

7.1 一般调试方法

1. QuecOpen 提供的 SDK 中，kernel log 的默认消息级别为 4(KERN_WARNING)，即内核在调用 printk() 时如果未指定消息级别，则默认为 4；
2. 控制台的默认打印级别的 7 (KERN_DEBUG)，即小于 7 的 kernel log 会被内核代码执行到，虽然执行到但此时是存放在内核 log_buffer 里面，当我们使用 dmesg 时才会把 buffer 的 log 输出到标准输出；

那么如果要想打开已经编译进 kernel 的 debug log，直接命令行 dmesg -n 8 修改；
或者在代码中修改默认值：

```
--- a/ql-ol-kernel/include/linux/printk.h
+++ b/ql-ol-kernel/include/linux/printk.h
@@ -40,7 +40,7 @@ static inline const char *printk_skip_level(const char *buffer
#define CONSOLE_LOGLEVEL_SILENT 0 /* Mum's the word */
#define CONSOLE_LOGLEVEL_MIN 1 /* Minimum loglevel we let people use */
#define CONSOLE_LOGLEVEL_QUIET 4 /* Shhh ..., when booted with "quiet" */
-#define CONSOLE_LOGLEVEL_DEFAULT 7 /* anything MORE serious than KERN_DEBUG */
+#define CONSOLE_LOGLEVEL_DEFAULT 8 /* anything MORE serious than KERN_DEBUG */
#define CONSOLE_LOGLEVEL_DEBUG 10 /* issue debug messages */
#define CONSOLE_LOGLEVEL_MOTORMOUTH 15 /* You can't shut this one up */
```

然而在很多驱动模块中，都是会定义自己的 DEBUG 编译宏，如果不打开这个宏，连 printk(KERN_DEBUG) 的代码都不会编译到，下面选中调试选项：

make kernel_menuconfig 选中下面 SPI 调试选项，并编译烧录；

```
--- SPI support
[ ] Debug support for SPI drivers
*** SPI Master Controller Drivers ***
< > Altera SPI Controller
< > Utilities for Bitbanging SPI masters
< > Cadence SPI controller
< > GPIO-based bitbanging SPI Master
[ ] Freescale SPI controller and Aeroflex Gaisler GRLIB SPI controller
< > OpenCores tiny SPI
< > Rockchip SPI controller driver
```

此时会看到消息：


```
[ 98.740632] spichn spi6.0: setup mode 0, 8 bits/w, 19200000 Hz max --> 0
[ 98.740652] spichn spi6.0: setup mode 0, 8 bits/w, 19200000 Hz max --> 0
[ 98.749320] mdm9607-asoc-snd soc:sound: ASoC: CODEC DAI rt5616-aif1 Name: alc5616-codec.2-001b
[ 98.749631] spi_qsd 78ba000.spi: registered child spi6.0
[ 100.040316] spi_qsd 78ba000.spi: pm_runtime: suspending...
[ 111.562611] spi_qsd 78ba000.spi: pm_runtime: resuming...
[ 113.040191] spi_qsd 78ba000.spi: pm_runtime: suspending...
[ 113.563900] spi_qsd 78ba000.spi: pm_runtime: resuming...
[ 115.040328] spi_qsd 78ba000.spi: pm_runtime: suspending...
root@mdm9607-perf:~#
```

7.2 使用 kernel tracer 进行调试

QuecOpen SDK 默认开启了 kernel hacking 中的 kernel debugfs 和 kernel tracer 功能，kernel tracer 功能很强大，常用来调试内核；

```
root@mdm9607-perf:/sys/kernel/debug/tracing# ls
README          instances       trace
available_events options        trace_clock
available_tracers per_cpu        trace_marker
buffer_size_kb  printk_formats trace_options
buffer_total_size_kb saved_cmdlines trace_pipe
current_tracer  saved_cmdlines_size tracing_cpumask
events          saved_tgids    tracing_on
free_buffer     set_event      tracing_thresh
root@mdm9607-perf:/sys/kernel/debug/tracing#
```

1. 打开内核调用栈追踪
echo 1 > options/stacktrace
2. 打开 printk 输出的 log
echo 1 > events/printk/enable
3. 打开 spi event 调试
echo 1 > events/spi/enable
4. 发起一次 spi 访问
5. cat trace 就可以看到内核中 spi 接口的调用状态；

```
#          TASK-PID  CPU#  ||| /   delay
#          | |      |    |||  |   |
#          | |      |    |||  |   |
kworker/u2:4-144 [000] d..2 4148.061315: spi_message_submit: spi6.0 ce033e54
kworker/u2:4-144 [000] d..2 4148.061387: <stack trace>
=> spidev_workq
=> process_one_work
=> worker_thread
=> kthread
=> ret_from_fork
    spi6-126 [000] ...1 4148.061466: spi_master_busy: spi6
    spi6-126 [000] ...1 4148.061485: <stack trace>
=> kthread
=> ret_from_fork
    spi6-126 [000] ...1 4148.062022: spi_message_start: spi6.0 ce033e54
    spi6-126 [000] ...1 4148.062054: <stack trace>
=> kthread
=> ret_from_fork
    spi6-126 [000] ...1 4148.062098: spi_transfer_start: spi6.0 ce033e84 len=512
    spi6-126 [000] ...1 4148.062118: <stack trace>
=> kthread_worker_fn
=> kthread
=> ret_from_fork
    spi6-126 [000] ...1 4148.062548: spi_transfer_stop: spi6.0 ce033e84 len=512
    spi6-126 [000] ...1 4148.062577: <stack trace>
=> kthread_worker_fn
=> kthread
=> ret_from_fork
    spi6-126 [000] ...1 4148.062621: spi_message_done: spi6.0 ce033e54 len=512/512
    spi6-126 [000] ...1 4148.062641: <stack trace>
=> spi_pump_messages
```