Linux SPI
开发指南

版本号: 1.3
发布日期: 2021.10.18

版本历史

版本号	日期	制/修订人	内容描述
1.0	2020.06.24	AWA1637	建立初版
1.1	2020.11.10	AWA1636	新增 Linux-5.4 内容
1.2	2021.04.10	XAA0193	1. 适用列表中增加了新的 IC 2. 对 5.4 下的 dts 进行了更新 3. 增加了 probe 失败时的排查思路
1.3	2021.04.27	XAA0193	1. 对 spi 的驱动框架进行了补充 2. 根据评审意见进行了修改
1.3	2021.10.18	AWA1669	增加 Slave 测试使用说明

目　录

插　图

1 前言

1.1 文档简介

介绍 SPI 模块的使用方法，方便开发人员使用。

1.2 目标读者

SPI 模块的驱动开发/维护人员。

1.3 适用范围

表 1-1：适用产品列表

内核版本	驱动文件
Linux-4.9	spi-sunxi.c
Linux-5.4	spi-sunxi.c

2 模块介绍

2.1 模块功能介绍

SPI 是一种高速、高效率的串行接口技术。通常由一个主模块和一个或多个从模块组成，主模块选择一个从模块进行同步通信，从而完成数据的交换，被广泛应用于 ADC、LCD 等设备与 MCU 之间。全志的 spi 控制器支持以下功能：

- 全双工同步串行接口。
- 支持 5 种时钟源选择。
- 支持 master 和 slave 两种配置。
- 四个 cs 片选支持。
- 8bit 宽度和 64 字节 fifo 深度。
- cs 和 clk 的极性和相位可配置。
- 支持使用 DMA。
- 支持四种通信模式。
- 批量生产支持最大的 io 速率 100MHz。
- 支持 3 线、4 线 SPI 模式。
- 支持可编程串行行数据帧长：0~32bits。
- 支持 Standard SPI/Dual-Output/Dual-input SPI/Dual i/O SPI/ 和 Quad-Output/Quad-Input SPI。

2.2 相关术语介绍

2.2.1 硬件术语

表 2-1: 硬件术语

术语	解释说明
SPI	Serial Peripheral Interface，同步串行外设接口

2.2.2 软件术语

表 2-2: 软件术语

术语	解释说明
Sunxi	指 Allwinner 的一系列 SOC 硬件平台
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 device tree 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，在设备树中配置参数相似，平台设备树文件的路径为：kernel/内核版本/arch/arm64（32 位平台为 arm）/boot/dts/sunxi/CHIP.dtsi(CHIP 为研发代号，如 sun50iw10p1 等)，对于配置 SPI1 而言，如下：

```
spi1: spi@05011000 {
            #address-cells = <1>;
            #size-cells = <0>;
            compatible = "allwinner,sun50i-spi";//具体的设备，用于驱动和设备的绑定
            device_type = "spi1";//设备节点名称
            reg = <0x0 0x05011000 0x0 0x1000>;//总线寄存器配置
            interrupts = <GIC_SPI 13 IRQ_TYPE_LEVEL_HIGH>;//总线中断号、中断类型
            clocks = <&clk_pll_periph0>, <&clk_spi1>;//设备使用的时钟
            clock-frequency = <100000000>;//控制器的时钟频率
            pinctrl-names = "default", "sleep";//控制器使用的Pin脚名称
            pinctrl-0 = <&spi1_pins_a &spi1_pins_b>;//控制器使用的pin脚配置
            pinctrl-1 = <&spi1_pins_c>;//控制器使用的pin脚配置
            spi1_cs_number = <1>;//控制器cs脚数量
            spi1_cs_bitmap = <1>;/* cs0- 0x1; cs1-0x2, cs0&cs1-0x3. */
            status = "disabled";//控制器是否使能
        };
```

在 Linux-5.4 版本内核中，与 Linux-4.9 内核配置有稍许差异，主要在于 clock 和 dma 的配置上：

```
spi1: spi@4026000 {
            #address-cells = <1>;
            #size-cells = <0>;
            compatible = "allwinner,sun20i-spi";//具体的设备，用于驱动和设备的绑定
            reg = <0x0 0x04026000 0x0 0x1000>;  //设备节点名称
            interrupts-extended = <&plic0 32 IRQ_TYPE_LEVEL_HIGH>;//总线中断号、中断类型
            clocks = <&ccu CLK_PLL_PERIPH0>, <&ccu CLK_SPI1>, <&ccu CLK_BUS_SPI1>;//设备使用
    的时钟
            clock-names = "pll", "mod", "bus";//设备使用的时钟名称
            resets = <&ccu RST_BUS_SPI1>;//设备的reset时钟
            clock-frequency = <100000000>;//控制器的时钟频率
            spi1_cs_number = <1>;//控制器cs脚数量
            spi1_cs_bitmap = <1>;/* cs0- 0x1; cs1-0x2, cs0&cs1-0x3. */
```

```
            dmas = <&dma 23>, <&dma 23>;//控制器使用的dms通道号
            dma-names = "tx", "rx";//控制器使用通道号对应的名字
            status = "disabled";//控制器是否使能
        };
```

为了在 SPI 总线驱动代码中区分每一个 SPI 控制器，需要在 Device Tree 中的 aliases 节点中为每一个 SPI 节点指定别名：

```
aliases {
    soc_spi0 = &spi0;
    soc_spi1 = &spi1;
    ...
};
```

别名形式为字符串 “spi” 加连续编号的数字，在 SPI 总线驱动程序中可以通过 of_alias_get_id() 函数获取对应 SPI 控制器的数字编号，从而区别每一个 SPI 控制器。

其中内核版本为 Linux-4.9 的 spi1_pins_a, spi1_pins_b 的配置文件路径为 kernel/linux-4.9/arch/arm64（32 位平台为 arm）/boot/dts/sunxi/xxx-pinctrl.dtsi，具体配置如下所示：

```
spi1_pins_a: spi1@0 {
  allwinner,pins = "PH4", "PH5", "PH6";
  allwinner,pname = "spi1_sclk", "spi1_mosi",
        "spi1_miso";
  allwinner,function = "spi1";
  allwinner,muxsel = <2>;
  allwinner,drive = <1>;
  allwinner,pull = <0>;
};

spi1_pins_b: spi1@1 {
  allwinner,pins = "PH3";
  allwinner,pname = "spi1_cs0";
  allwinner,function = "spi1";
  allwinner,muxsel = <2>;
  allwinner,drive = <1>;
  allwinner,pull = <1>; // only CS should be pulled up
};

spi1_pins_c: spi1@2 {
  allwinner,pins = "PH3", "PH4", "PH5", "PH6";
  allwinner,function = "io_disabled";
  allwinner,muxsel = <7>;
  allwinner,drive = <1>;
  allwinner,pull = <0>;
};
```

内核版本为 Linux-5.4 的 spi1_pins_a, spi1_pins_b 的具体配置如下所示：

```
spi1_pins_a: spi1@0 {
        pins = "PD11", "PD12", "PD13";
        function = "spi1";
        drive-strength = <10>;
};
```

```
6
7   spi1_pins_b: spi1@1 {
8           pins = "PD10";
9           function = "spi1";
10          drive-strength = <10>;
11          bias-pull-up;   /* only CS should be pulled up */
12  };
13
14  spi1_pins_c: spi1@2 {
15          pins = "PD10", "PD11", "PD12", "PD13";
16          function = "gpio_in";
17  };
```

2.3.2 board.dts 配置说明

board.dts 用于保存每一个板级平台设备差异化的信息的补充（如 demo 板，demo2.0 板，ver1 板等等），里面的配置信息会覆盖上面的 device tree 默认配置信息。

board.dts 的路径为/device/config/chips/{IC}/configs/{BOARD}/board.dts, 其中 SPI1 的具体配置如下：

说明

在 *Linux-5.4* 内核版本中对 *board.dts* 语法做了修改，不再支持同名节点覆盖，使用 “&” 符号引用节点。

```
1   &spi1 {
2       clock-frequency = <100000000>;
3       pinctrl-0 = <&spi1_pins_a &spi1_pins_b>;
4       pinctrl-1 = <&spi1_pins_c>;
5       pinctrl-names = "default", "sleep";
6       spi_slave_mode = <0>;
7       status = "disabled";
8
9       spi_board1@0 {
10          device_type = "spi_board1";
11          compatible = "rohm,dh2228fv";
12          spi-max-frequency = <0x5f5e100>;
13          reg = <0x0>;
14          spi-rx-bus-width = <0x4>;
15          spi-tx-bus-width = <0x4>;
16          status = "disabled";
17      };
18  };
```

注意，如果要使用 spi slave 模式，请把 spi_slave_mode = <0> 修改为：spi_slave_mode = <1>。

spi_board1 还有一些可配置参数，如：

- spi-cpha 和 spi-cpol：配置 spi 的四种传输模式。
- spi-cs-high：配置 cs 引脚有效状态时的电平。

spi1_pins_a, spi1_pins_b 、spi1_pins_c 的具体配置如下所示：

```
spi1_pins_a: spi1@0 {
        pins = "PD11", "PD12", "PD13","PD14", "PD15"; /*clk mosi miso hold wp*/
        function = "spi1";
        drive-strength = <10>;
};

spi1_pins_b: spi1@1 {
    pins = "PD10";
    function = "spi1";
    drive-strength = <10>;
    bias-pull-up;   // only CS should be pulled up
};

spi1_pins_c: spi1@2 {
    allwinner,pins = "PD10", "PD11", "PD12", "PD13","PD14", "PD15";
    allwinner,function = "gpio_in";
    allwinner,muxsel = <0>;
    drive-strength = <10>;
};
```

2.3.3 menuconfig 配置说明

在命令行中进入内核 linux 目录，执行 make ARCH=arm64 menuconfig(32 位系统为 make ARCH=arm menuconfig) 进入配置主界面 (Linux-5.4 内核版本执行：./build.sh menuconfig)，并按以下步骤操作。

选择 Device Drivers 选项进入下一级配置，如下图所示。

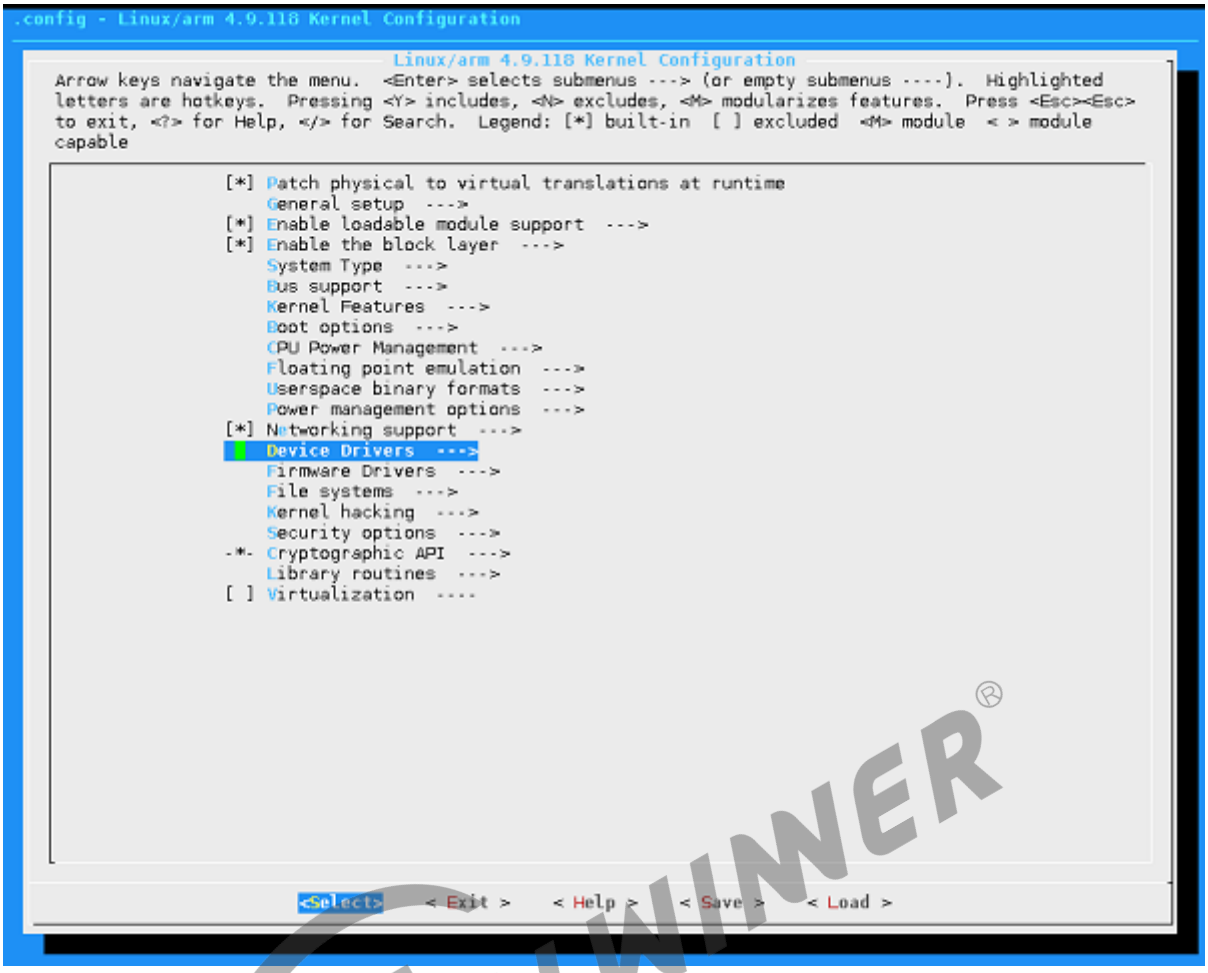

图 2-1: Device Drivers 配置选项

选择 SPI support 选项，进入下一级配置，如下图所示。

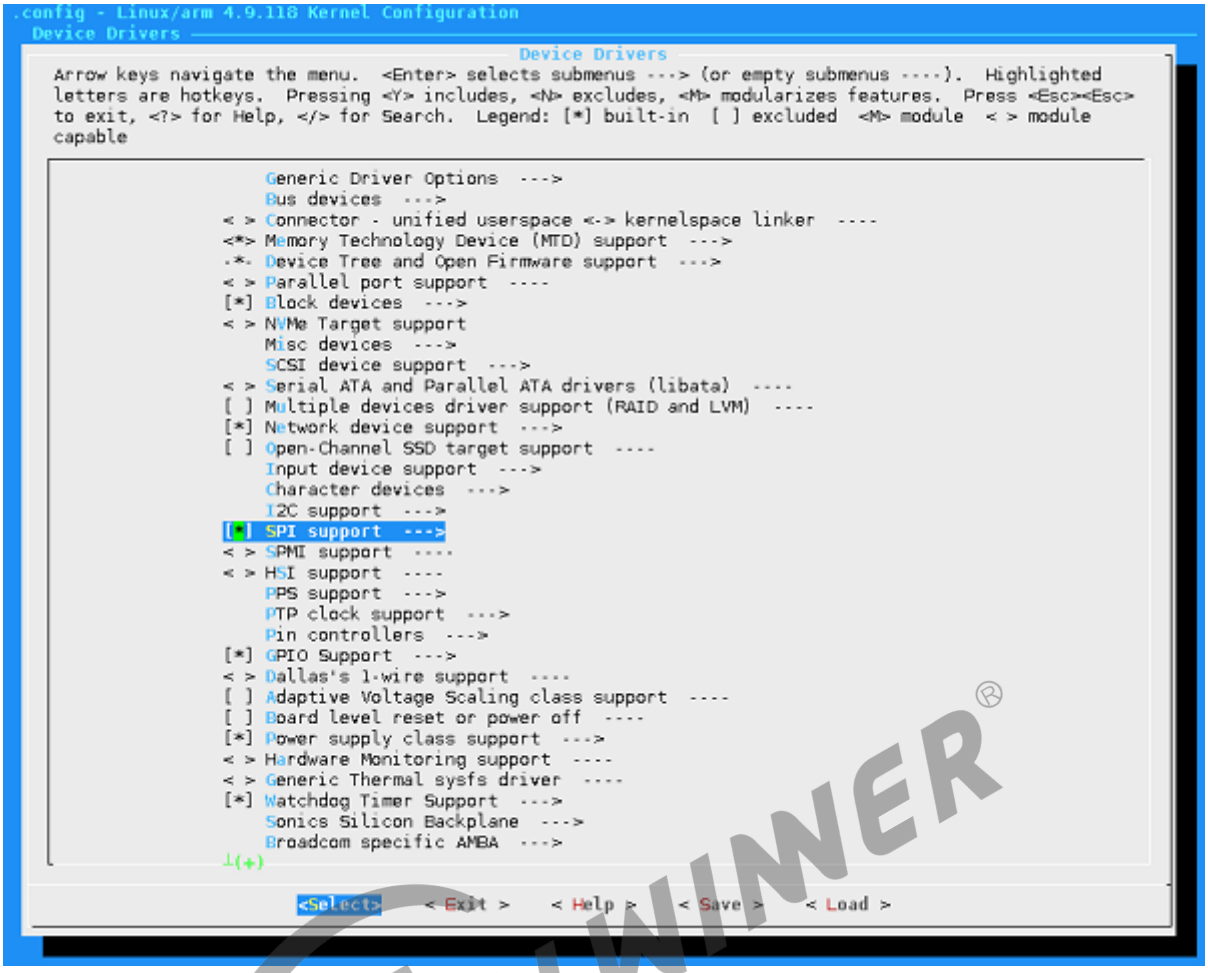

图 2-2: SPI support 配置选项

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。

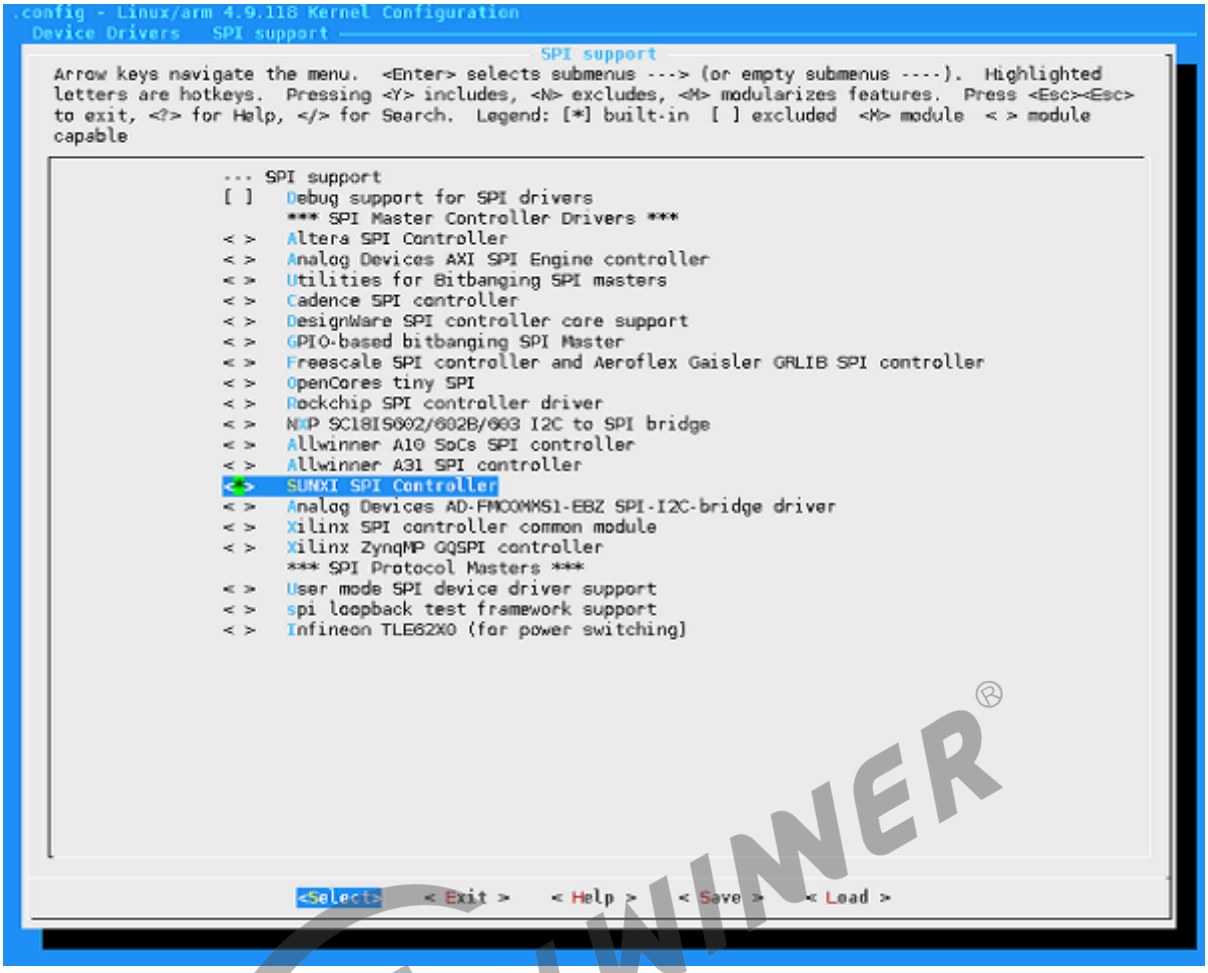

图 2-3: SUNXI SPI Controller 配置选项

如果想要放开 spi 的一些调试打印，可以选上 Debug support for SPI drivers。

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
```

2.5 驱动框架介绍

Linux 中 SPI 体系结构分为三个层次，如下图所示。

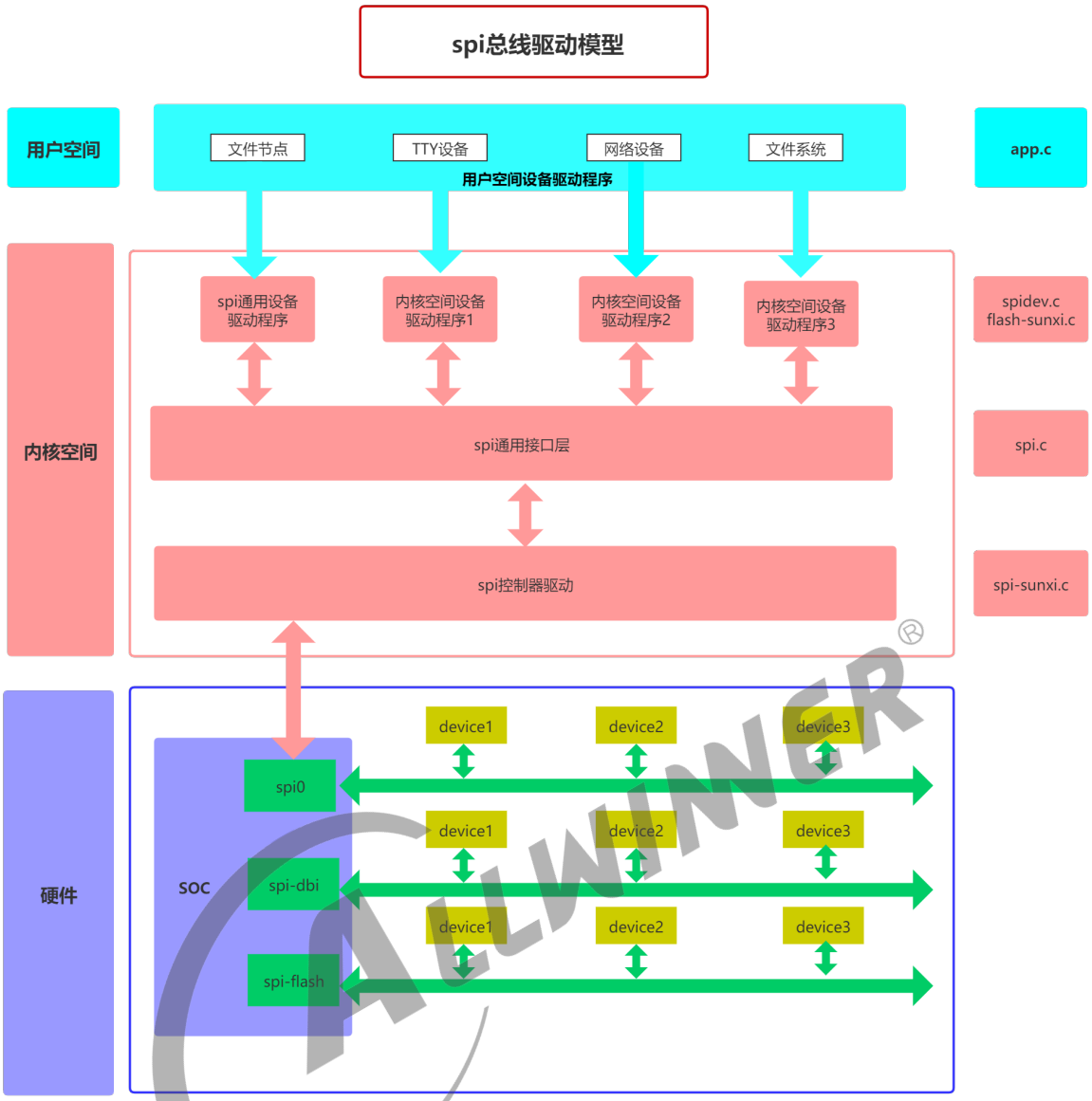

图 2-4: Linux SPI 体系结构图

2.5.1 用户空间

包括所有使用 SPI 设备的应用程序，在这一层用户可以根据自己的实际需求，将 spi 设备进行一些特殊的处理，此时控制器驱动程序并不清楚和关注设备的具体功能，SPI 设备的具体功能是由用户层程序完成的。例如，和 MTD 层交互以便把 SPI 接口的存储设备实现为某个文件系统，和 TTY 子系统交互把 SPI 设备实现为一个 TTY 设备，和网络子系统交互以便把一个 SPI 设备实现为一个网络设备，等等。当然，如果是一个专有的 SPI 设备，我们也可以按设备的协议要求，实现自己的专有协议驱动。同时这部分我们不用关注。

2.5.2 内核空间

内核空间我们同样的会分为一下三部分：

2.5.2.1 SPI 控制器驱动层

考虑到连接在 SPI 控制器上的设备的可变性，在内核没有配备相应的协议驱动程序，对于这种情况，内核为我们准备了通用的 SPI 设备驱动程序，该通用设备驱动程序向用户空间提供了控制 SPI 控制的控制接口，具体的协议控制和数据传输工作交由用户空间根据具体的设备来完成，在这种方式中，只能采用同步的方式和 SPI 设备进行通信，所以通常用于一些数据量较少的简单 SPI 设备。

这一层对应于我们内核中的 spidev.c 这个标准的 spi 设备驱动，或者我司的 spi-nand.c，支持 spi 协议的 nand 驱动等。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。

2.5.2.2 SPI 通用接口封装层

为了简化 SPI 驱动程序的编程工作，同时也为了降低协议驱动程序和控制器驱动程序的耦合程度，内核把控制器驱动和协议驱动的一些通用操作封装成标准的接口，加上一些通用的逻辑处理操作，组成了 SPI 通用接口封装层。这样的好处是，对于控制器驱动程序，只要实现标准的接口回调 API，并把它注册到通用接口层即可，无需直接和协议层驱动程序进行交互。而对于协议层驱动来说，只需通过通用接口层提供的 API 即可完成设备和驱动的注册，并通过通用接口层的 API 完成数据的传输，无需关注 SPI 控制器驱动的实现细节。

这一层对应于驱动中的 spi.c 文件，是内核原生的文件。

2.5.2.3 SPI 控制器驱动层

为了简化 SPI 驱动程序的编程工作，同时也为了降低协议驱动程序和控制器驱动程序的耦合程度，内核把控制器驱动和协议驱动的一些通用操作封装成标准的接口，加上一些通用的逻辑处理操作，组成了 SPI 通用接口封装层。这样的好处是，对于控制器驱动程序，只要实现标准的接口回调 API，并把它注册到通用接口层即可，无需直接和协议层驱动程序进行交互。而对于协议层驱动来说，只需通过通用接口层提供的 API 即可完成设备和驱动的注册，并通过通用接口层的 API 完成数据的传输，无需关注 SPI 控制器驱动的实现细节。

这一层是我们关注的重点，在后文介绍中会详细的展开进行介绍。

2.5.3 硬件

这一层是实际的物理器件，其中包括我们的 spi 控制器以及与控制器相连的各个 spi 子设备，通过 spi 总线能够与 cpu 进行数据的交互。

3 接口描述

3.1 设备注册接口

接口定义在 include/linux/spi/spi.h，主要包含 spi_register_driver 与 spi_unregister_driver 接口，其中给出了快速注册的 SPI 设备驱动的宏 module_spi_driver()，定义如下：

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

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：返回 0 表示成功，返回其他值表示失败。

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

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 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。内核线程使用队列实现了异步传输的功能，对于同一个数据传输的发起者，既然异步方式无需等待数据传输完成即可返回，返回后，该发起者可以立刻又发起一个 message，而这时上一个 message 还没有处理完。对于另外一个不同的发起者来说，也有可能同时发起一次 message 传输请求。

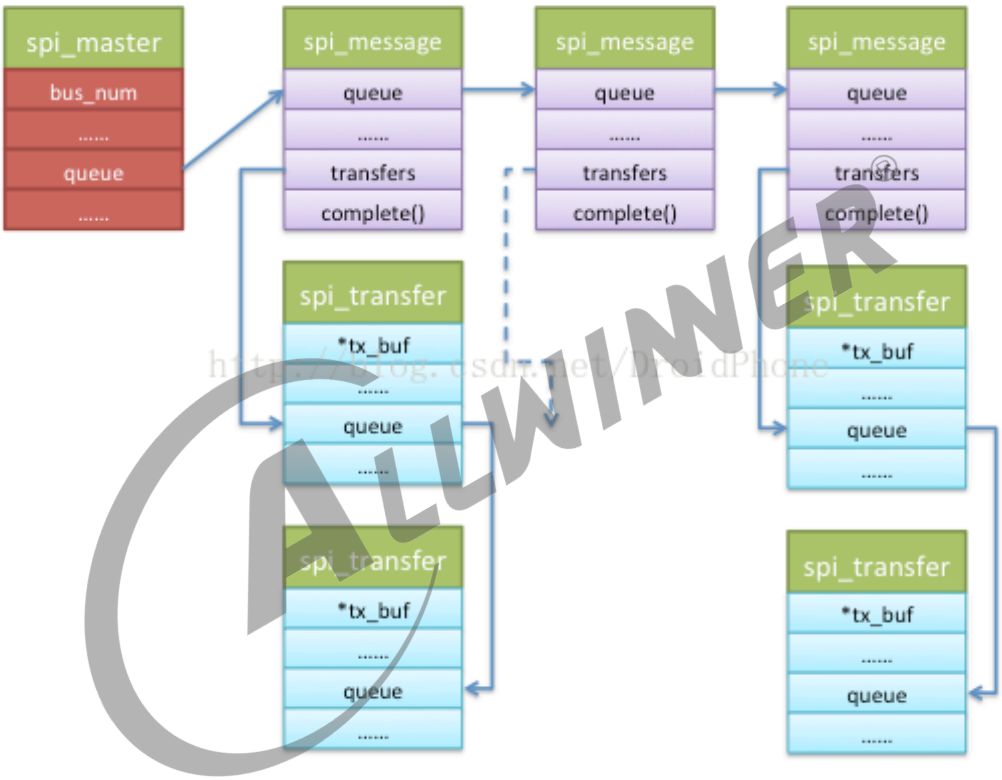

图 3-1: Linux SPI 数据传输流程

```
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。

4 模块使用范例

4.1 内核原生驱动范例

驱动文件在 drivers/spi/spidev.c，此驱动是 Linux 内核自带的一个 spidev 通用驱动。其中调用调用 spi_register_driver() 注册 SPI 驱动，方便使用者实现 SPI message 数据的读写。

```
static int __init spidev_init(void)
{
    int status;

    /* Claim our 256 reserved device numbers.  Then register a class
     * that will key udev/mdev to add/remove /dev nodes.  Last, register
     * the driver which manages those device numbers.
     */
    BUILD_BUG_ON(N_SPI_MINORS > 256);
    status = register_chrdev(SPIDEV_MAJOR, "spi", &spidev_fops);
    if (status < 0)
        return status;

    spidev_class = class_create(THIS_MODULE, "spidev");
    if (IS_ERR(spidev_class)) {
        unregister_chrdev(SPIDEV_MAJOR, spidev_spi_driver.driver.name);
        return PTR_ERR(spidev_class);
    }

    status = spi_register_driver(&spidev_spi_driver);
    if (status < 0) {
        class_destroy(spidev_class);
        unregister_chrdev(SPIDEV_MAJOR, spidev_spi_driver.driver.name);
    }
    return status;
}
module_init(spidev_init);

static void __exit spidev_exit(void)
{
    spi_unregister_driver(&spidev_spi_driver);
    class_destroy(spidev_class);
    unregister_chrdev(SPIDEV_MAJOR, spidev_spi_driver.driver.name);
}
module_exit(spidev_exit);
```

同时需要在对应的 spi 控制器的 dts 下加上 spi 子设备的设备信息描述，具体的配置信息如下所示：

```
&spi1 {
    clock-frequency = <100000000>;
    pinctrl-0 = <&spi1_pins_a &spi1_pins_b>;
```

```
    pinctrl-1 = <&spi1_pins_c>;
    pinctrl-names = "default", "sleep";
    spi_slave_mode = <0>;
    status = "disabled";

    spi_board1@0 {
        device_type = "spi_board1";
        compatible = "rohm,dh2228fv";
        spi-max-frequency = <0x5f5e100>;
        reg = <0x0>;
        spi-rx-bus-width = <0x4>;
        spi-tx-bus-width = <0x4>;
        status = "disabled";
    };
};
```

对于 spi 控制器的描述在这里不再重复的陈述，这里的 spi_board1@0 就是我们虚拟的一个 spi 从设备，

- device_type ： 表示设备的类型;
- compatible ： 驱动匹配信息;
- spi-max-frequency ： 从设备的最大频率;
- reg ： 从设备的寄存器地址;
- spi-rx-bus-width：对从设备进行数据读取时使用的 data 数据线个数;
- spi-tx-bus-width ： 对从设备进行数据写入时使用的 data 数据线个数;
- status ： 从设备的状态;

在 menuconfig（Device Drivers->SPI support）里面配置上 User mode SPI device driver support 选项。

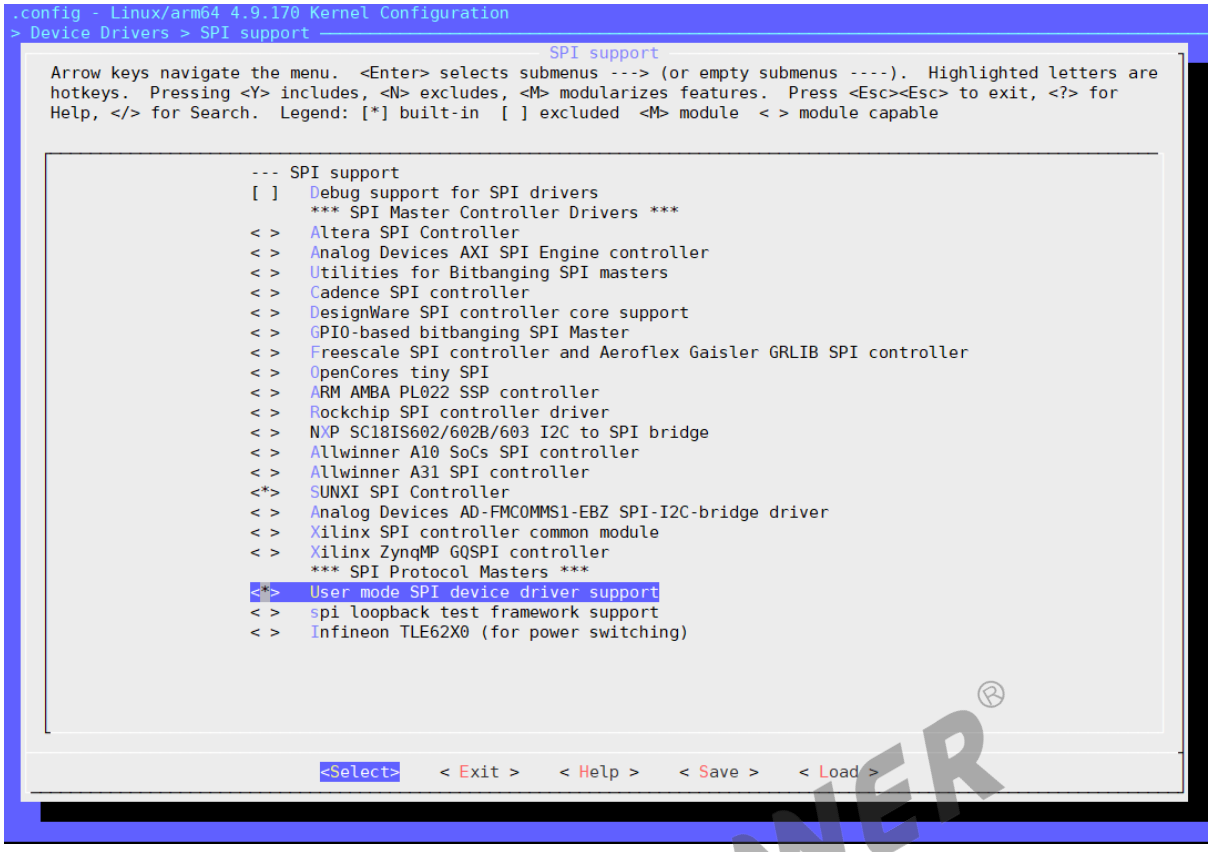

图 4-1: spidev

编译烧录固件之后会在小机文件系统的/dev 目录下发现 spidevX.0(X=0~2) 设备，可以对 spidevX.0 进行读写操作。或者使用 Linux 自带的 spi 工具：在 tina/lichee/linux-5.4/tools 目录下, 运行如下命令:

```
make spi
```

然后在 tina/lichee/linux-5.4/tools/spi/下会有 spidev_test 可执行文件，拷贝到小机根文件系统中，运行如下命令即可进行测试：

```
/spidev_test -D /dev/spidevX.0
```

4.2 Slave 模式驱动范例

需要在 board.dts 中相应的 SPI 节点设备配置 spi_slave_mode = <1>。

4.2.1 Slave 写数据

以 spidev1.0 设备为例，发送 0~9 十个数据：

```c
#define DEVICE_NAME "/dev/spidev1.0"
#define HEAD_LEN 5
#define PKT_MAX_LEN 0x40
#define STATUS_LEN 0x01

#define SUNXI_OP_WRITE  0x01
#define SUNXI_OP_READ   0x03

#define STATUS_WRITABLE 0x02
#define STATUS_READABLE 0x04

#define WRITE_DELAY 200
#define READ_DELAY  100000

void dump_data(unsigned char *buf, unsigned int len)
{
    unsigned int i;
    unsigned char tmp[len*2], cnt = 0;

    for (i = 0; i < len; i++) {
        if (i%0x10== 0)
            cnt += sprintf(tmp + cnt, "0x%08x: ", i);

        cnt += sprintf(tmp + cnt, "%02x ", buf[i]);

        if ( (i%0x10== 0x0f) || (i == (len -1)) ) {
            printf("%s\n", tmp);
            cnt = 0;
        }
    }
}

void batch_rand(char *buf, unsigned int length)
{
    unsigned int i;
    srand(time(0));

    for(i = 0; i < length; i++) {
        *(buf + i) = rand() % 256;
    }
}

int main(int argc, const char *argv[])
{
    unsigned int length = 0, test_len;
    char wbuf_head[HEAD_LEN] = {SUNXI_OP_WRITE, 0x00, 0x00, 0x00, 0x00};
    char rbuf_head[HEAD_LEN] = {SUNXI_OP_READ, 0x00, 0x00, 0x00, 0x00};
    char wbuf[PKT_MAX_LEN], rbuf[PKT_MAX_LEN], i, time;
    int fd, ret;

    test_len = 10;//send 10 numbers
    if (test_len > PKT_MAX_LEN) {
        printf("invalid argument, numbers must less 64B\n");
        return -1;
    }

    wbuf_head[4] = test_len;
    rbuf_head[4] = test_len;

    for (i = 0; i < test_len; i++)
```

```
        wbuf[i] = i;
    printf("wbuf:\n");
    dump_data(wbuf, test_len);

    fd = open(DEVICE_NAME, O_RDWR);
    if (fd <= 0) {
        printf("Fail to to open %s\n", DEVICE_NAME);
        ret = -1;
        return ret;
    }

    {//write
        if (write(fd, wbuf_head,  HEAD_LEN) !=  HEAD_LEN) {
            printf("W Fail to write head\n");
            ret = -1;
            goto err;
        } else
            printf("W write head successful\n");

        usleep(WRITE_DELAY);

        if (write(fd, wbuf,  test_len) !=  test_len) {
            printf("W Fail to write data\n");
            ret = -1;
            goto err;
        } else
            printf("W write data successful\n");

        usleep(READ_DELAY);
    }

err:
    if (fd > 0)
        close(fd);

    return ret;
}
```

4.2.2 Slave 读数据

以 spidev1.0 设备为例，读十个数据：

```
#define DEVICE_NAME "/dev/spidev1.0"
#define HEAD_LEN 5
#define PKT_MAX_LEN 0x40
#define STATUS_LEN 0x01

#define SUNXI_OP_WRITE  0x01
#define SUNXI_OP_READ   0x03

#define STATUS_WRITABLE 0x02
#define STATUS_READABLE 0x04

#define WRITE_DELAY 200
#define READ_DELAY  100000

```

```
15  void dump_data(unsigned char *buf, unsigned int len)
16  {
17      unsigned int i;
18      unsigned char tmp[len*2], cnt = 0;
19
20      for (i = 0; i < len; i++) {
21          if (i%0x10== 0)
22              cnt += sprintf(tmp + cnt, "0x%08x: ", i);
23
24          cnt += sprintf(tmp + cnt, "%02x ", buf[i]);
25
26          if ( (i%0x10== 0x0f) || (i == (len -1)) ) {
27              printf("%s\n", tmp);
28              cnt = 0;
29          }
30      }
31  }
32
33  void batch_rand(char *buf, unsigned int length)
34  {
35      unsigned int i;
36      srand(time(0));
37
38      for(i = 0; i < length; i++) {
39          *(buf + i) = rand() % 256;
40      }
41  }
42
43  int main(int argc, const char *argv[])
44  {
45      unsigned int length = 0, test_len;
46      char wbuf_head[HEAD_LEN] = {SUNXI_OP_WRITE, 0x00, 0x00, 0x00, 0x00};
47      char rbuf_head[HEAD_LEN] = {SUNXI_OP_READ, 0x00, 0x00, 0x00, 0x00};
48      char wbuf[PKT_MAX_LEN], rbuf[PKT_MAX_LEN], i, time;
49      int fd, ret;
50
51      test_len = 10;
52      if (test_len > PKT_MAX_LEN) {
53          printf("inval argument, numbers must less 64B\n");
54          return -1;
55      }
56
57      wbuf_head[4] = test_len;
58      rbuf_head[4] = test_len;
59
60      fd = open(DEVICE_NAME, O_RDWR);
61      if (fd <= 0) {
62          printf("Fail to to open %s\n", DEVICE_NAME);
63          ret = -1;
64          return ret;
65      }
66
67      {//read
68          if (write(fd, rbuf_head,  HEAD_LEN) !=  HEAD_LEN) {
69              printf("R Fail to write head\n");
70              ret = -1;
71              goto err;
72          } else
73              printf("R write head successful\n");
74
```

```
75          usleep(READ_DELAY);
76
77          if (read(fd, rbuf,  test_len) !=  test_len) {
78              printf("R Fail to read data\n");
79              ret = -1;
80              goto err;
81          } else
82              printf("R read data successful\n");
83
84          usleep(READ_DELAY);
85
86      }
87
88      printf("rbuf:\n");
89      dump_data(rbuf, test_len);
90
91  err:
92      if (fd > 0)
93          close(fd);
94
95      return ret;
96  }
```

4.2.3 Slave 使用 & 测试

4.2.3.1 环境搭建

4.2.3.1.1 硬件环境 本此测试使用两块开发板搭建环境，一块做 master，一块做 slave。

将 MASTER 与 SLAVE 的 SPI1 的 CS、CLK 按名字对应连接起来，MASTER 的 MOSI 接 SLAVE 的 MOSI，MASTER 的 MISO 接 SLAVE 的 MISO，将两块开发板共地。

4.2.3.1.2 Menuconfig 打开 menuconfig 的 CONFIG_SPI_SUNXI 与 CONFIG_SPI_SPIDEV，如下图所示。

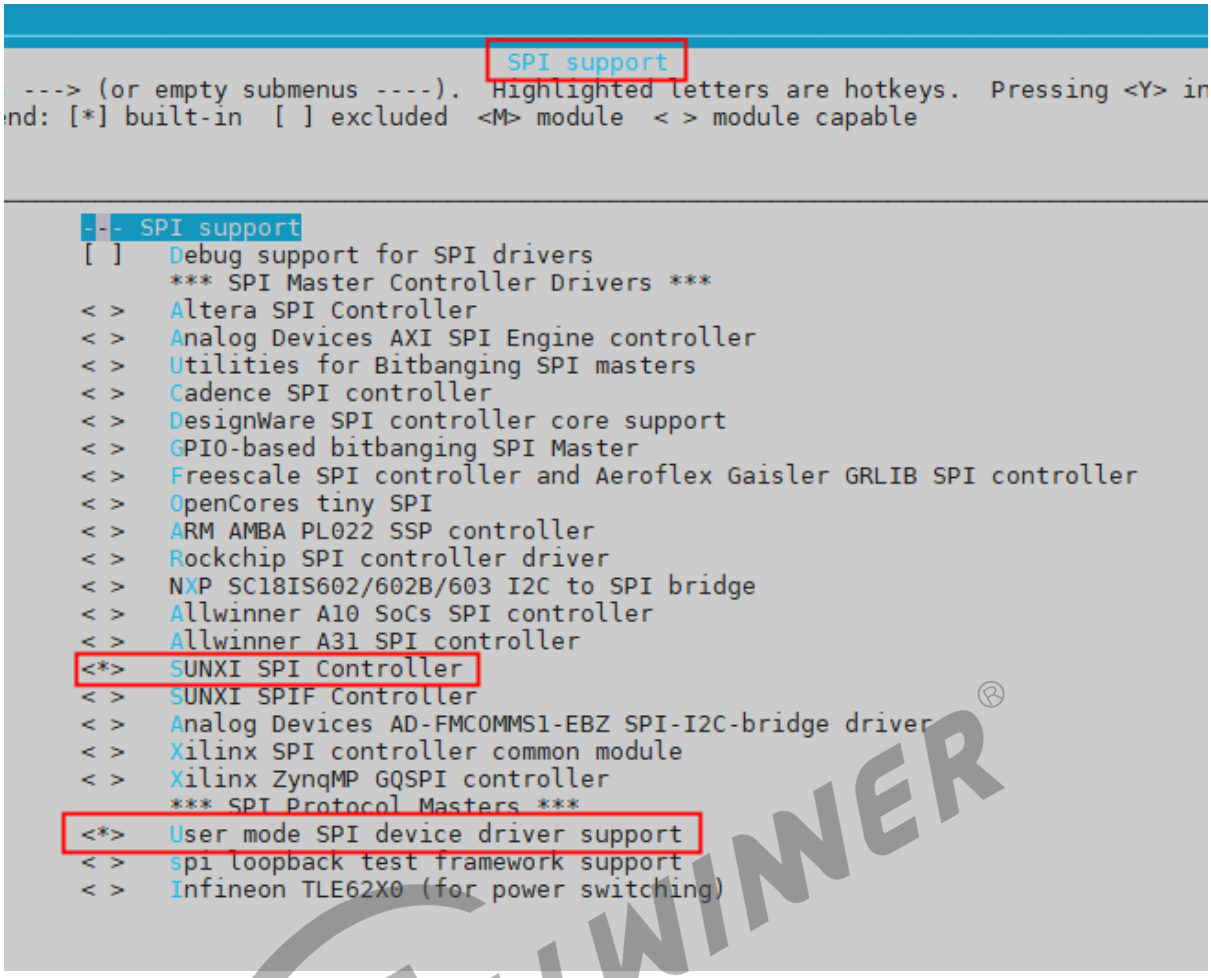

图 4-2: menuconfig

4.2.3.1.3 DTS 设备树路径：device/config/chips/xxx(t507)/configs/xxx(demo2.0)/board.dts，添加以下节点：

```
spi1: spi@05011000 {
            pinctrl-0 = <&spi1_pins_a &spi1_pins_b>;
            pinctrl-1 = <&spi1_pins_c>;
            spi_slave_mode = <0>;
            status = "okay";
            spi_board1 {
                device_type = "spi_board1";
                compatible = "rohm,dh2228fv";
                spi-max-frequency = <30000000>;
                reg = <0x0>;
                spi-rx-bus-width = <0x1>;
                spi-tx-bus-width = <0x1>;
            };
};
```

注：spi_slave_mode = <0> 为 Master 配置；spi_slave_mode = <1>，为 Slave 配置

4.2.3.2 测试

分别设置 Master 和 Salve 的 DTS，并编译出对应固件，烧写固件。

4.2.3.2.1 Slave Slave 端执行下列命令，打开 Slave 的调试打印，这样可以看到读写的数据。

```
echo 4 > /sys/module/spi_sunxi/parameters/debug
```

4.2.3.2.2 Master 交叉编译测试用例中的 slave_test.c，得到测试文件 slave_test。将 slave_test 传入 Master 中。

```
./slave_test <n>
//n 为测试次数，默认不填为1
```

4.2.3.3 测试结果

Maset source data 和 target data 打印数据一致，即表明测试通过。

```
-----------------------------------------
                  n test
-----------------------------------------
W write head successful
W write data successful
source data:
0x00000000: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a
0x00000010: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a
R write head successful
R read data successful
target data:
0x00000000: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a
0x00000010: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a
slave function [PASS]
```

4.2.3.4 自定制说明

用户可以自定制从设备功能，要操作从设备，需要发送 5 个 byte 的操作请求，说明如下：

第 1 个 Byte：操作码

```
SUNXI_OP_WRITE  0x01
SUNXI_OP_READ   0x03
//读写是相对于master
```

第 2~4 个 Byte：地址（2 是高位地址）

第 5 给 Byte：长度（长度要求小于 64Byte）

4.2.3.4.1 操作码添加 现在我们只支持读写操作，用户自行拓展，在drivers/spi/spi-sunxi.c的sunxi_spi_slave_handle_head函数中添加命令对应的操作函数

```
if (head->op_code == SUNXI_OP_WRITE) {
        sunxi_spi_slave_cpu_rx_config(sspi);
    } else if (head->op_code == SUNXI_OP_READ) {
        sunxi_spi_slave_cpu_tx_config(sspi);
    } else {
        dprintk(DEBUG_INFO, "[spi%d] pkt head opcode err\n", sspi->master->bus_num);
        ret = -1;
        goto err1;
}
```

4.2.3.4.2 地址及缓存 第 2~4 个 Byte 的地址是用于指定读写缓存数据，缓存大小宏在drivers/spi/spi-slave-protocol.h中定义，用户自行设置，单位 Byte

```
#define STORAGE_SIZE            128
```

4.2.3.4.3 长度 每次读写数据长度要求小于 64Byte，由于 SPI RX/TX 的 FIFO 缓存大小为 64Byte，为了防止读写时有一端设备没有及时拿走数据导致 buf 溢出，一次传输要求长度小于 64Byte，如果要读写大于 64Byte 数据，可分多次进行传输，地址偏移好就没问题。

5 FAQ

5.1 调试节点

5.1.1 /sys/module/spi_sunxi/parameters/debug

默认情况下 debug 为 1，不打开调试信息。

```
echo 255 > /sys/module/spi_sunxi/parameters/debug
```

即可打开调试信息。

5.1.2 /sys/devices/platform/soc/spi1/info

此节点文件可以打印出当前 SPI1 通道的一些硬件资源信息。

```
cat /sys/devices/platform/soc/spi1/info
```

5.1.3 /sys/devices/platform/soc/spi1/status

此节点文件可以打印出当前 SPI1 通道的一些运行状态信息，包括控制器的各寄存器值。

```
cat /sys/devices/platform/soc/spi1/status
```

5.2 如何在 UBoot 中使用 SPI1

目前 Uboot 中 SPI 已支持使用其他的 SPI 接口，如 SPI1/2 等，但用户仍需自行在设备树中添加对应节点已打开配置。

本例以 R528-evb2 进行说明，在 Uboot 中使用 SPI1 需要添加以下修改，其他板子可以参考借鉴

5.2.1 设备树配置修改

- 在对应的 SoC 级设备树文件中添加 SPI Controller 信息

in `arch/arm/dts/sun8iw20p1-soc-system.dts`

```
spi1: spi1@4026000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "allwinner,sun20i-spi";
    device_type = "spi1";
    reg = <0x0 0x04026000 0x0 0x300>;
};
```

- 在对应的板级设备树文件中添加 SPI 节点和 Pinctrl 信息

in `configs/evb2/uboot-board.dts`

```
&spi1_pins_a {
    allwinner,pins = "PB11", "PB10", "PB9";
    allwinner,pname = "spi1_sclk", "spi1_mosi", "spi1_miso";
    allwinner,function = "spi1";
    allwinner,muxsel = <5>;
    allwinner,drive = <1>;
    allwinner,pull = <0>;
};

&spi1_pins_b {
    allwinner,pins = "PB12", "PB8", "PB0";
    allwinner,pname = "spi1_cs0", "spi1_hold", "spi1_wp";
    allwinner,function = "spi1";
    allwinner,muxsel = <5>;
    allwinner,drive = <1>;
    allwinner,pull = <1>;   // only CS should be pulled up
};

&spi1_pins_c {
    allwinner,pins = "PC2", "PC3", "PC4", "PC5",
            "PC6", "PC7";
    allwinner,function = "gpio_in";
    allwinner,muxsel = <0>;
    allwinner,drive = <1>;
    allwinner,pull = <0>;
};

&spi1 {
    clock-frequency = <100000000>;
    pinctrl-0 = <&spi1_pins_a &spi1_pins_b>;
    pinctrl-1 = <&spi1_pins_c>;
    pinctrl-names = "default", "sleep";
    /*spi-supply = <&reg_dcdc1>;*/
    spi_slave_mode = <0>;
    spi1_cs_number = <1>;
    spi1_cs_bitmap = <1>;
    status = "okay";
```

```
};
```

5.2.2 Uboot 配置

新增了一个 Uboot 命令 (sunxi_sspi) 用于测试 SPI 数据收发。需要在配置文件中打开以下配置信息

in `sun8iw20p1_evb2_defconfig`

```
CONFIG_CMD_SUNXI_SSPI=y
```

5.2.3 测试结果

重新编译 uboot 并烧写后，在板子上短接 SPI1 的 MOSI/MISO 引脚，使用 sunxi_sspi 命令进行数据回环收发测试

- 短接前

```
=> sunxi_sspi 1:0 32 12345678
SPI SEND:
12345678
SPI RECV:
FFFFFFFF
```

- 短接后

```
=> sunxi_sspi 1:0 32 12345678
SPI SEND:
12345678
SPI RECV:
12345678
```

可以看到短接后 SPI1 的收发数据一致，回环测试通过

更多 sunxi_sspi 命令的用法可以使用`h sunxi_sspi`得知

5.3 常见问题

5.3.1 dts 中设置使能不生效

问题现象：在 board.dts 中配置 spi 的 statue 状态为 “okay”，但是启动 Linux 内核却发现 spi 控制器未使能。问题分析：可能状态配置有误，亦或者错误使用其他的控制器例如 spi0。

问题排查步骤：

- 步骤 1：这种问题一般是由于在设备树里，你的设备依赖了别的设备，但是这个设备没能 probe 成功，从而导致你的设备无法 probe。建议对 spi 依赖的 dma 模块进行排查，检查 dma 在 menuconfig 中是否被打开；
- 步骤 2：在 out/目录下搜索.sunxi.dts 并打开：

```
find -name ".sunxi.dts"
```

在文件里找到对应的节点，检查对应的 spi 是否配置成功。

- 步骤 3：在小机 uboot 控制台通过 fdt list spi* 命令查看 dts，是否使能 SPI 成功（status = “okay”），如果还是 disable，则可能 spi 在 uboot 阶段被 disable 掉了（一般 spi0 会保留给 flash 使用，spi0 会在 uboot 阶段关闭掉）。

5.3.2 SPI-Flash 数据传输异常

问题现象：写入与读出数据不一致。

- 步骤 1：进行兼容性排查。以 nor flash 为例，有些物料兼容性不好，会造成读写出错。这个时候可以先确认下次款物料是否在支持列表内。若不在，试着更换物料再做测试。
- 步骤 2：驱动调试。此类问题范围比较大，但是可以从基础调试手段着手跟踪调试。一般思路是打开数据打印，看写入的值是否传到 SPI 总线驱动处理，然后同样的看 SPI 总线驱动刚读出来的数据与前面写的打印数据是否一致，来判断是哪个环节造成读写出错，这个办法可以拓展到其他层次，以确认是文件系统层、MTD 层、SPI 总线驱动层的读或写问题。

著作权声明

商标声明

免责声明