



Linux TWI 开发指南

版本号: 1.2
发布日期: 2021.04.29

版本历史

版本号	日期	制/修订人	内容描述
1.0	2019.12.31	AWA1637	version 1.0
1.1	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
1.2	2021.04.29	XAA0193	根据评审意见再次进行修改



目 录

1 前言	1
1.1 文档简介	1
1.2 目标读者	1
1.3 适用范围	1
2 模块介绍	2
2.1 模块功能介绍	2
2.2 相关术语介绍	2
2.2.1 硬件术语	2
2.2.2 软件术语	2
2.3 模块配置介绍	3
2.3.1 device tree 默认配置	3
2.3.2 board.dts 板级配置	5
2.3.3 kernel menuconfig 配置	6
2.4 源码模块结构	9
2.5 驱动框架介绍	10
3 模块接口说明	12
3.1 i2c-core 接口	12
3.1.1 i2c_transfer()	12
3.1.2 i2c_master_recv()	12
3.1.3 i2c_master_send()	12
3.1.4 i2c_smbus_read_byte()	13
3.1.5 i2c_smbus_write_byte()	13
3.1.6 i2c_smbus_read_byte_data()	13
3.1.7 i2c_smbus_write_byte_data()	14
3.1.8 i2c_smbus_read_word_data()	14
3.1.9 i2c_smbus_write_word_data()	15
3.1.10 i2c_smbus_read_block_data()	15
3.1.11 i2c_smbus_write_block_data()	15
3.2 i2c 用户态调用接口	16
3.2.1 i2cdev_open()	16
3.2.2 i2cdev_read()	16
3.2.3 i2cdev_write()	17
3.2.4 i2cdev_ioctl()	17
4 模块使用范例	18
4.1 利用 i2c-core 接口读写 TWI 设备	18
4.2 利用用户态接口读写 TWI 设备	21
5 FAQ	22
5.1 调试方法	22

5.1.1	调试工具	22
5.1.1.1	i2c-tools 调试工具	22
5.1.2	调试节点	22
5.1.2.1	/sys/module/i2c_sunxi/parameters/transfer_debug	22
5.1.2.2	/sys/devices/soc.2/1c2ac00.twi.0/info	22
5.1.2.3	/sys/devices/soc.2/1c2ac00.twi/status	23
5.2	常见问题	23
5.2.1	TWI 数据未完全发送	23
5.2.2	TWI 起始信号无法发送	23
5.2.3	TWI 终止信号无法发送	24
5.2.4	TWI 传送超时	25



插 图

2-1 Device Driver	7
2-2 I2C support	7
2-3 I2C device interface	8
2-4 2C HardWare Bus support	8
2-5 SUNXI I2C controller	9
2-6 TWI 模块结构框图	10



1 前言

1.1 文档简介

介绍 Sunxi 平台上 TWI 驱动接口与调试方法，为 TWI 模块开发提供参考。

1.2 目标读者

TWI 模块内核层以及应用层的开发、维护人员。

1.3 适用范围

表 1-1: 适用产品列表

产品名称	内核版本	驱动文件
T509	Linux-4.9	i2c-sunxi.c
MR813	Linux-4.9	i2c-sunxi.c
R818	Linux-4.9	i2c-sunxi.c
A53	Linux-4.9	i2c-sunxi.c
A100	Linux-4.9	i2c-sunxi.c
A100	Linux-5.4	i2c-sunxi.c
A133	Linux-4.9	i2c-sunxi.c
A133	Linux-5.4	i2c-sunxi.c
D1	Linux-5.4	i2c-sunxi.c
R528	Linux-5.4	i2c-sunxi.c

2 模块介绍

2.1 模块功能介绍

全志公司的 twi 总线兼容 i2c 总线协议，是一种简单、双向二线制同步串行总线。它只需要两根线即可在连接于总线上的器件之间传送信息。TWI 控制器支持的标准通信速率为 100Kbps，最高通信速率可以达到 400K bps。。全志的 twi 控制器支持一下功能：

- 支持主机模式和从机模式；
- 主机模式下支持 dma 传输；
- 主机模式下在多个主机的模式下支持总线仲裁；
- 主机模式下支持时钟同步，位和字节等待；
- 从机模式下支持地址检测中断；
- 支持 7bit 从机地址和 10bit 从机地址；
- 支持常规的 i2c 协议模式和自定义传输模式；

sunxi 平台支持多路 TWI，包含 TWI 与 S_TWI，其中 S_TWI0 作为与 PMU 通信使用，该 TWI 不建议复用为其他功能。

2.2 相关术语介绍

2.2.1 硬件术语

表 2-1: 硬件术语

相关术语	解释说明
TWI	Two Wire Interface，全志平台兼容 I2C 标准协议的总线控制器

2.2.2 软件术语

表 2-2: 软件术语

相关术语	解释说明
Sunxi	全志科技使用的 linux 开发平台
I2C_dapter	linux 内核中 I2C 总线适配器的抽象定义.IIC 总线的控制器，在物理上连接若干个 I2C 设备
I2C_algorithm	linux 内核中 I2C 总线通信的抽象定义。描述 I2C 总线适配器与 I2C 设备之间的通信方法
I2C Client	linux 内核中 I2C 设备的抽象定义
I2C Driver	linux 内核中 I2C 设备驱动的抽象定义

2.3 模块配置介绍

在不同的 Sunxi 硬件平台中，TWI 控制器的数目不同；但对于同一块板子上的每一个 TWI 控制器来说，模块配置类似，本小节展示 Sunxi 平台上的 TWI0 控制器配置（其他 TWI 控制器配置类似）。

2.3.1 device tree 默认配置

设备树中存在的是该类芯片所有平台的模块配置，设备树文件的路径为：kernel/linux-4.9/arch/arm64（32 位平台为 arm）/boot/dts/sunxi(32 位系统无这目录)/xxxx.dtsi(CHIP 为研发代号，如 sun50iw10p1 等)，TWI 总线的设备树配置如下所示：

```

1 twi0: twi@0x05002000{
2     #address-cells = <1>;
3     #size-cells = <0>;
4     compatible = "allwinner,sun50i-twi";           //具体的设备，用于驱动和设备的绑定
5     device_type = "twi0";                         //设备节点名称，用于sys_config.fex匹配
6     reg = <0x0 0x05002000 0x0 0x400>;           //TWI0总线寄存器配置
7     interrupts = <GIC_SPI 6 IRQ_TYPE_LEVEL_HIGH>; //TWI0总线中断号、中断类型
8     clocks = <&clk_twi0>;                         //设备使用的时钟
9     clock-frequency = <400000>;                  //TWI0控制器的时钟频率
10    pinctrl-names = "default", "sleep";           //TWI0控制器使用的Pin脚名称，其中default为正常通
信时的引脚配置，sleep为睡眠时的引脚配置
11    pinctrl-0 = <&twi0_pins_a>;                   //TWI0控制器default时使用的pin脚配置
12    pinctrl-1 = <&twi0_pins_b>;                   //TWI0控制器sleep时使用的pin脚配置
13    twi_drv_used = <1>;                           //使用DMA传输数据
14    status = "disabled";                           //TWI0控制器是否使能
15 }
```

在 linux-5.4 中，TWI 的配置与 linux-4.9 内核配置有些不同，区别主要体现在 clock 和 dma 的配置上：

```

1 twi0: twi@0x05002000{
2     #address-cells = <1>;
3     #size-cells = <0>;
4     compatible = "allwinner,sun20i-twi";           //具体的设备，用于驱动和设备的绑定
5     device_type = "twi0";                         //设备节点名称，用于sys_config.fex匹配
```



```

6      reg = <0x0 0x02502000 0x0 0x400>;          //TWI0总线寄存器配置
7      interrupts-extended= <&plic0 25 IRQ_TYPE_LEVEL_HIGH>; //TWI0总线中断号、中断类型
8      clocks = <&ccu CLK_BUS_I2C0>;//twi控制器使用的时钟
9      resets = <&ccu RST_BUS_I2C0>;//twi控制器使用的reset时钟
10     clock-names = "bus";
11     clock-frequency = <400000>; //TWI0控制器的时钟频率
12     dmas = <&dma 43>, <&dma 43>;//TWI0控制器的dma通道号
13     dma-names = "tx", "rx";
14     status = "disabled";//TWI0控制器是否使能
15 };

```

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

```

1  aliases {
2      soc_twi0 = &twi0;
3      soc_twi1 = &twi1;
4      soc_twi2 = &twi2;
5      soc_twi3 = &twi3;
6      ...
7  };

```

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

其中 twi0_pins_a, twi0_pins_b 为 TWI 的引脚配置的配置节点。

linux4.9 中该配置的路径为 arch/arm64 (32 位平台为 arm) /boot/dts/sunxi/xxxx-pinctrl.dtsi(CHIP 为研发代号，如 sun50iw10p1 等)，具体配置如下所示：

```

1  twi0_pins_a: twi0@0 {
2      allwinner,pins = "PD14", "PD15";          //TWI控制器使用的引脚
3      allwinner,pname = "twi0_scl", "twi0_sda"; //TWI控制器的引脚功能说明
4      allwinner,function = "twi0";              //引脚功能描述
5      allwinner,muxsel = <4>;                   //引脚复用功能配置
6      allwinner,drive = <0>;                    //io驱动能力
7      allwinner,pull = <0>;                    //内部电阻状态
8  };
9
10 twi0_pins_b: twi0@1 {
11     allwinner,pins = "PD14", "PD15";
12     allwinner,function = "io_disabled";
13     allwinner,muxsel = <7>;
14     allwinner,drive = <1>;
15     allwinner,pull = <0>;
16 };

```

linux-5.4 中该配置的路径为 arch/arm64 (32 位平台为 arm) /boot/dts/sunxi/xxxx.dtsi(CHIP 为研发代号，如 sun50iw10p1 等)，具体如下所示：

```

1  twi0_pins_a: twi0@0 {
2      pins = "PH0", "PH1";
3      function = "twi0";

```

```

4     drive-strength = <10>;
5 };
6
7 twi0_pins_b: twi0@1 {
8     pins = "PH0", "PH1";
9     function = "gpio_in";
10 };

```

另外 clk_twi0 为时钟的配置。

在 linux-4.9 中，路径为 arch/arm64 (32 位平台为 arm) /boot/dts/sunxi/XXXX-clk.dtsi(CHIP 为研发代号，如 sun50iw10p1 等)，具体配置如下所示：

```

1 clk_twi0: twi0 {
2     #clock-cells = <0>;
3     compatible = "allwinner,periph-clock";
4     clock-output-names = "twi0"; //指定clock名称，用于匹配clock配置
5 };

```

在 linux-5.4 中，无需配置。

2.3.2 board.dts 板级配置

board.dts 用于保存每一个板级平台的设备信息（如 demo 板，perf1 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 longan/device-config/chips/IC/configs/BOARD/board.dts,

在 linux-4.9 中，对应 board.dts 里面 TWI0 的具体配置如下：

```

1 twi0_pins_a: twi0@0 {
2     allwinner,pins = "PA0", "PA1";
3     allwinner,pname = "twi0_scl", "twi0_sda";
4     allwinner,function = "twi0";
5     allwinner,muxsel = <4>;
6     allwinner,drive = <1>;
7     allwinner,pull = <0>;
8 };
9
10 twi0_pins_b: twi0@1 {
11     allwinner,pins = "PA0", "PA1";
12     allwinner,function = "io_disabled";
13     allwinner,muxsel = <7>;
14     allwinner,drive = <1>;
15     allwinner,pull = <0>;
16 };
17
18 twi0: twi0@0x05002000{
19     clock-frequency = <400000>; //i2c时钟频率为400K
20     pinctrl-0 = <&twi0_pins_a>;
21     pinctrl-1 = <&twi0_pins_b>;
22     status = "okay"; //使能TWI0
23 };

```

在 linux-5.4 中，对应 board.dts 里面 TWI0 的具体配置如下：

```
1 &twi0 {
2     clock-frequency = <400000>;
3     pinctrl-0 = <&twi0_pins_a>;
4     pinctrl-1 = <&twi0_pins_b>;
5     pinctrl-names = "default", "sleep";
6     status = "disabled";
7
8     eeprom@50 {
9         compatible = "atmel,24c16";
10        reg = <0x50>;
11        status = "disabled";
12    };
13 };
```

其中，TWI 速率由“clock-frequency”属性配置，最大支持 400K。

对于 TWI 设备，可以把设备节点填充作为 Device Tree 中相应 TWI 控制器的子节点。TWI 控制器驱动的 probe 函数透过 of_i2c_register_devices()，自动展开作为其子节点的 TWI 设备。

对于 twi0 中引用的 pin 口，具体的配置如下：

```
twi0_pins_a: twi0@0 {
    pins = "PB10", "PB11"; /*sck sda*/
    function = "twi0";
    drive-strength = <10>;
};

twi0_pins_b: twi0@1 {
    pins = "PB10", "PB11";
    function = "gpio_in";
};
```

2.3.3 kernel menuconfig 配置

linux-4.9 在命令行进入内核根目录 (/kernel/linux-4.9)，执行 make ARCH=arm64 menuconfig (32 位平台执行：make ARCH=arm menuconfig) 进入配置主界面，并按以下步骤操作：linux-5.4 在根目录中执行 ./build.sh menuconfig

- 1. 选择 Device Drivers 选项进入下一级配置，如下图所示：

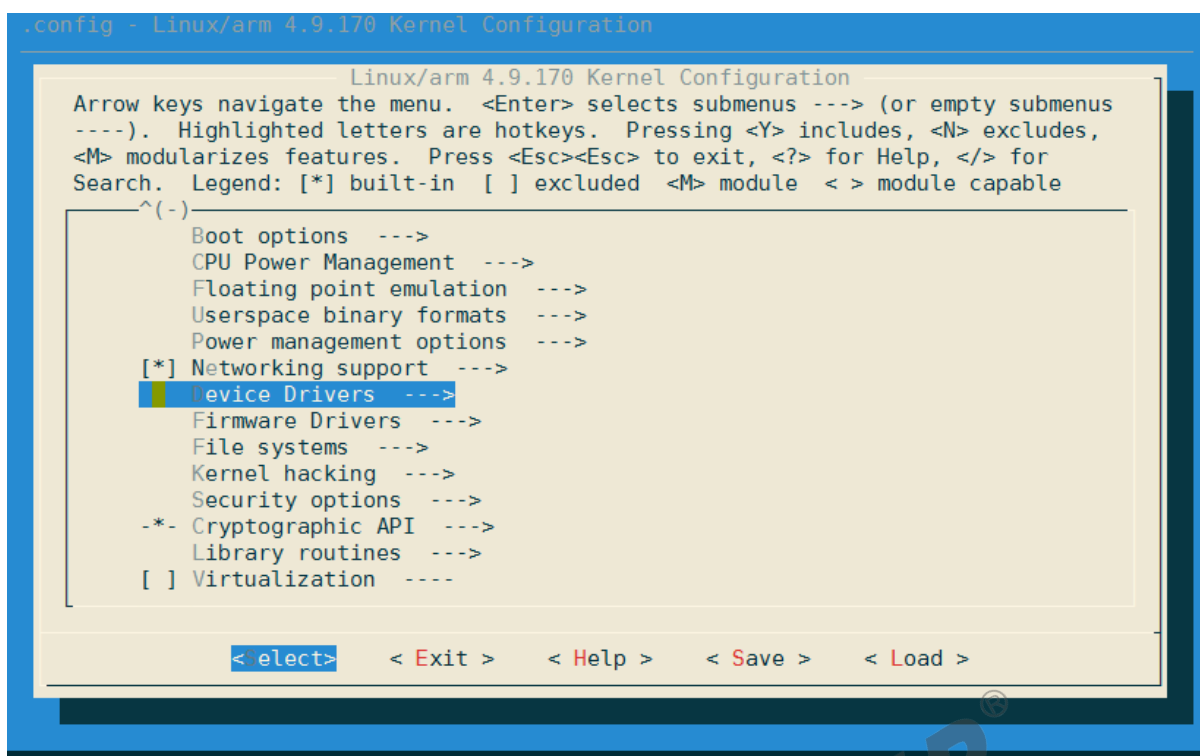


图 2-1: Device Driver

- 2. 选择 I2C support 选项，进入下一级配置，如下图所示：

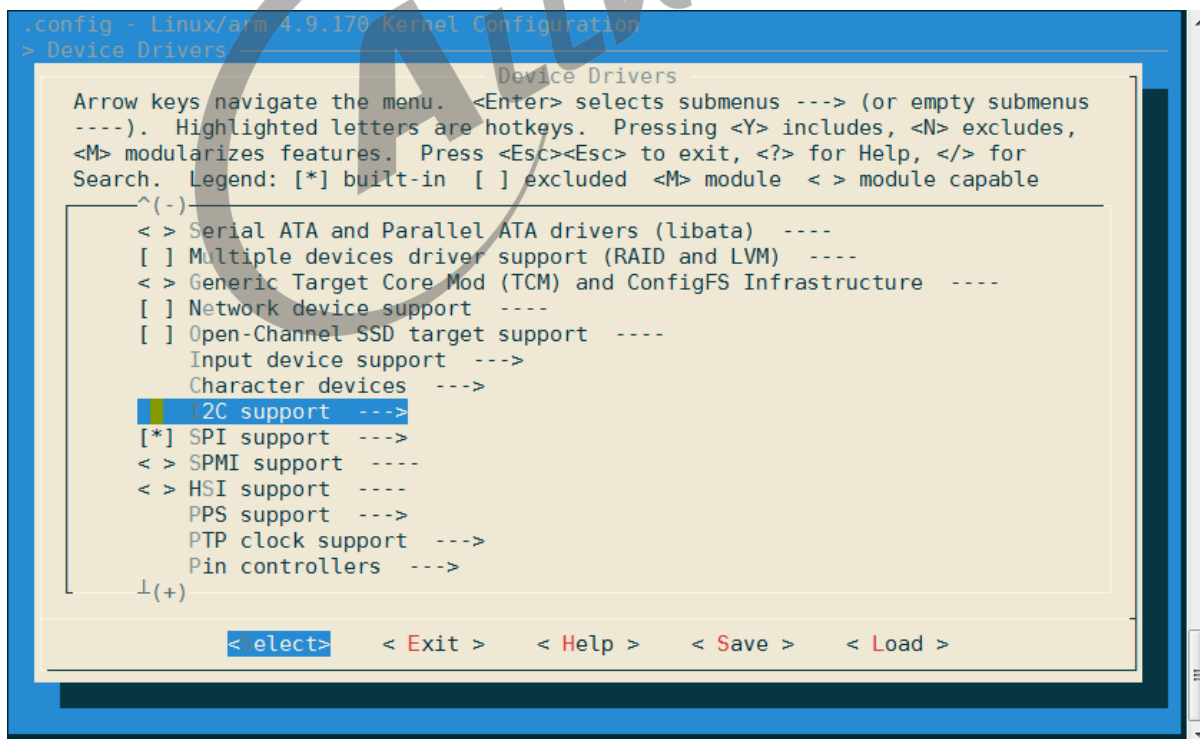


图 2-2: I2C support

- 3. 配置用户 I2C 接口，选择 I2C device interface，如下图所示：

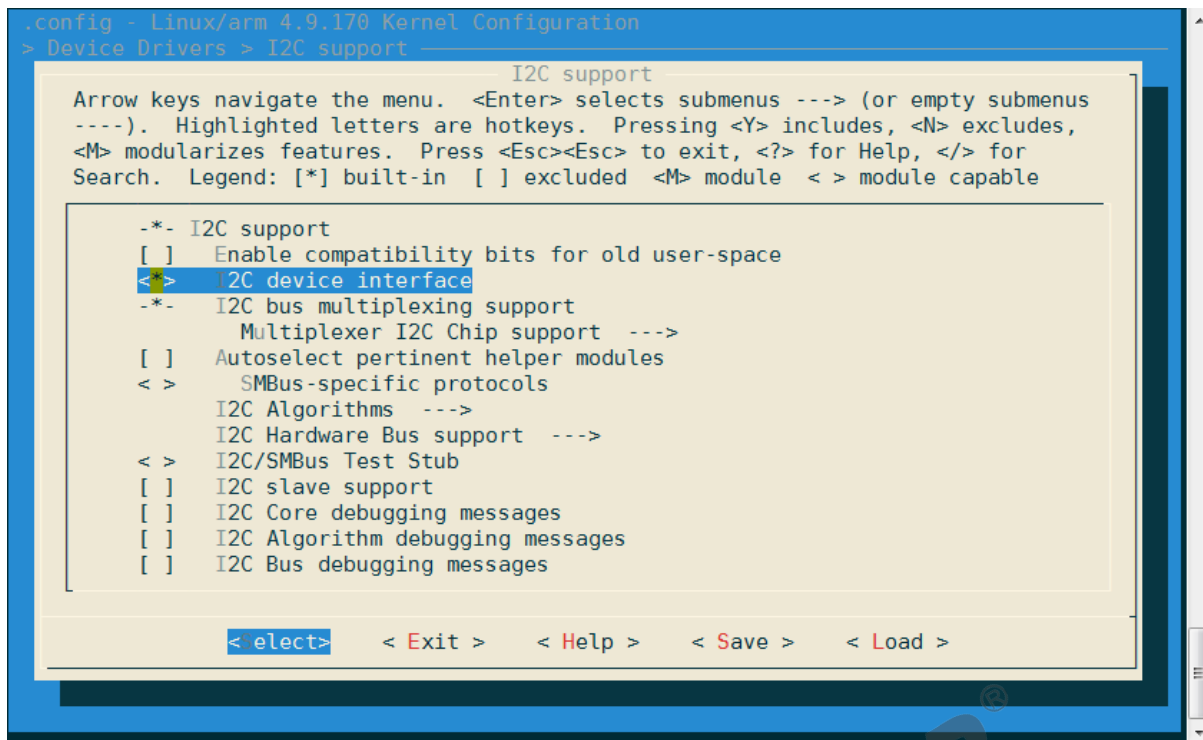


图 2-3: I2C device interface

- 4. 选择 I2C HardWare Bus support 选项，进入下一级配置，如下图所示：

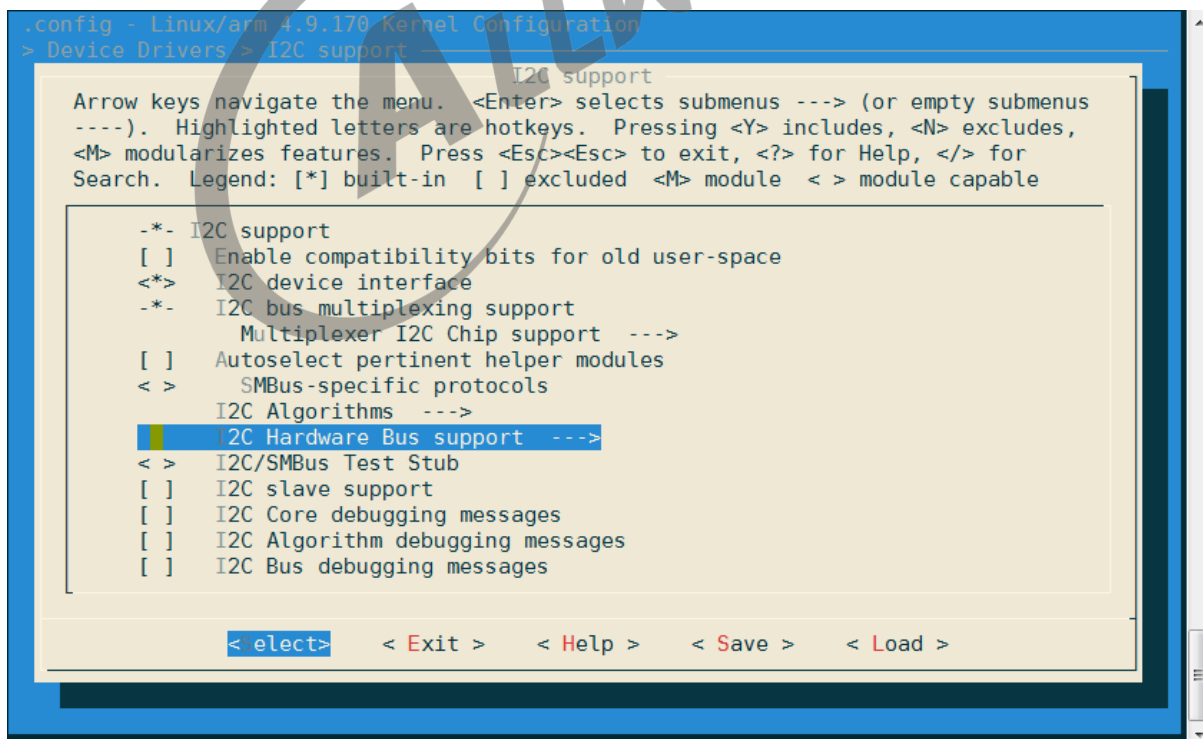


图 2-4: 2C HardWare Bus support

- 5. 选择 SUNXI I2C 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

示：

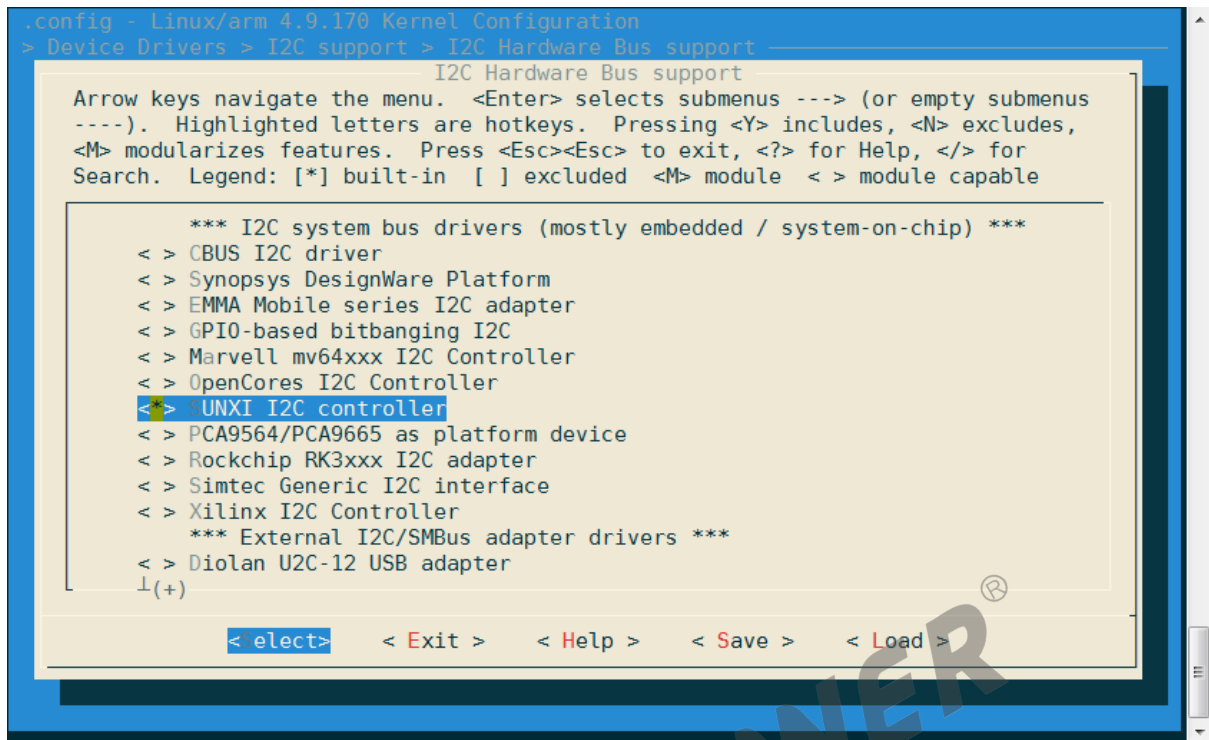


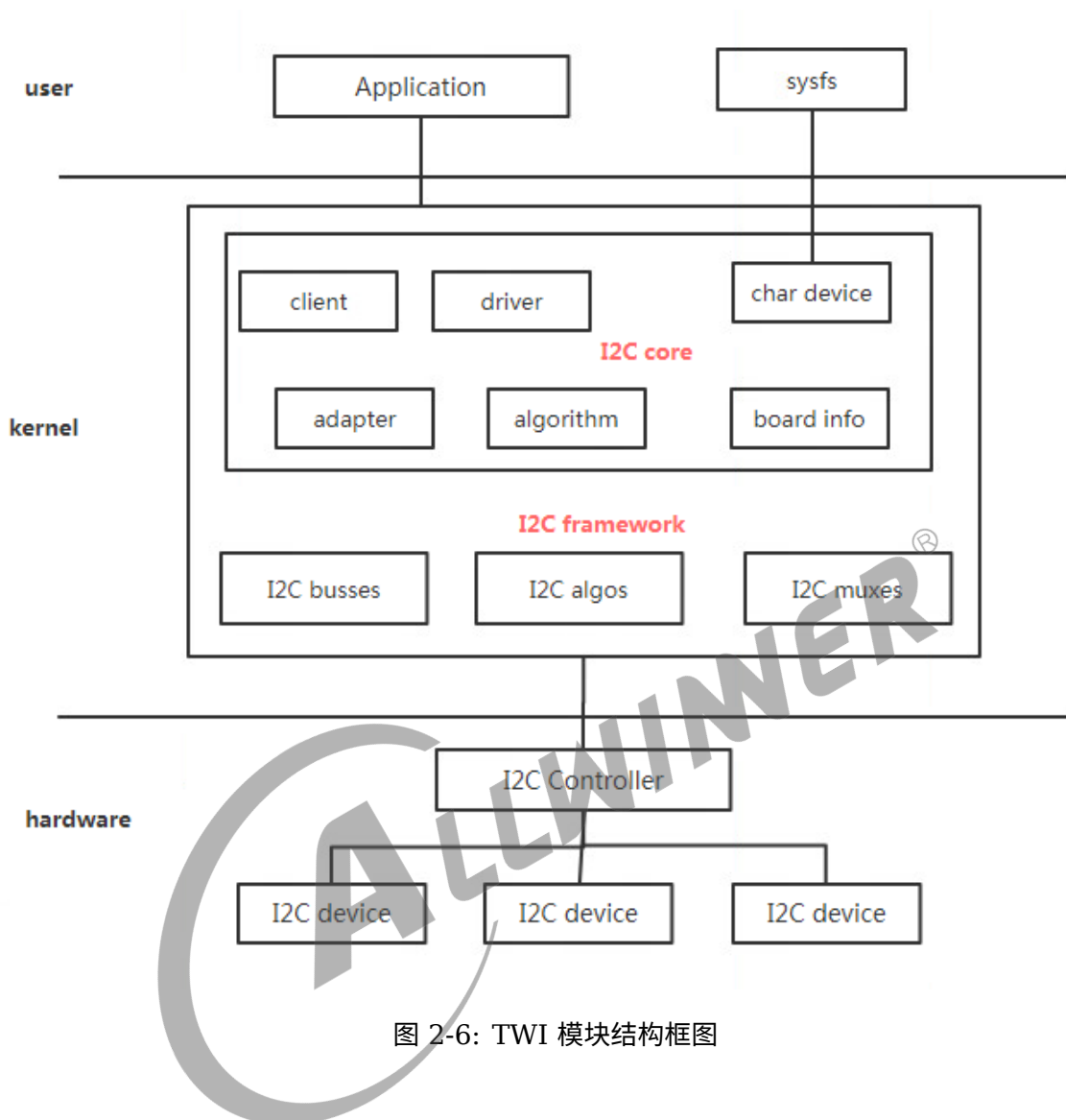
图 2-5: SUNXI I2C controller

2.4 源码模块结构

I2C 总线驱动的源代码位于内核在 drivers/i2c/busses 目录下：

```
kernel/linux-4.9/drivers/i2c/
├── busses
│   ├── i2c-sunxi.c           // Sunxi平台的I2C控制器驱动代码
│   ├── i2c-sunxi.h           // 为Sunxi平台的I2C控制器驱动定义了一些宏、数据结构
│   └── i2c-sunxi-test.c       // Sunxi平台的i2c设备测试代码,5.4下暂未适配
├── i2c-core.c                // I2C子系统核心文件,提供相关的接口函数
└── i2c-dev.c                  // I2C子系统的设备相关文件,用以注册相关的设备文件,方便调试
```

2.5 驱动框架介绍



Linux 中 I2C 体系结构上图所示，图中用分割线分成了三个层次：1. 用户空间，包括所有使用 I2C 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 I2C 控制器和 I2C 外设。

其中，Linux 内核中的 I2C 驱动程序从逻辑上又可以分为 6 个部分：

1. I2C framework 提供一种“访问 I2C slave devices”的方法。由于这些 slave devices 由 I2C controller 控制，因而主要由 I2C controller 驱动实现这一目标。
2. 经过 I2C framework 的抽象，用户可以不用关心 I2C 总线的技术细节，只需要调用系统的接口，就可以与外部设备进行通信。正常情况下，外部设备是位于内核态的其它 driver（如触摸屏，摄像头等等）。I2C framework 也通过字符设备向用户空间提供类似的接口，用户空间程序可以通过该接口访问从设备信息。

3. 在 I2C framework 内部，有 I2C core、I2C busses、I2C algos 和 I2C muxes 四个模块。
4. I2C core 使用 I2C adapter 和 I2C algorithm 两个子模块抽象 I2C controller 的功能，使用 I2C client 和 I2C driver 抽象 I2C slave device 的功能（对应设备模型中的 device 和 device driver）。另外，基于 I2C 协议，通过 smbus 模块实现 SMBus（System Management Bus，系统管理总线）的功能。
5. I2C busses 是各个 I2C controller drivers 的集合，位于 drivers/i2c/busses/目录下，i2c-sunxi-test.c、i2c-sunxi.c、i2c-sunxi.h。
6. I2C algos 包含了一些通用的 I2C algorithm，所谓的 algorithm，是指 I2C 协议的通信方法，用于实现 I2C 的 read/write 指令，一般情况下，都是由硬件实现，不需要特别关注该目录。



3 模块接口说明

3.1 i2c-core 接口

3.1.1 i2c_transfer()

- 函数原型：int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
- 作用：完成 I2C 总线和 I2C 设备之间的一定数目的 I2C message 交互。
- 参数：
 - adap：指向所属的 I2C 总线控制器；
 - msgs：i2c_msg 类型的指针；
 - num：表示一次需要处理几个 I2C msg
- 返回：
 - >0：已经处理的 msg 个数；
 - <0：失败；

3.1.2 i2c_master_recv()

- 函数原型：int i2c_master_recv(const struct i2c_client *client, char *buf, int count)
- 作用：通过封装 i2c_transfer() 完成一次 I2c 接收操作。
- 参数：
 - client：指向当前 I2C 设备的实例；
 - buf：用于保存接收到的数据缓存；
 - count：数据缓存 buf 的长度
- 返回：
 - >0：成功接收的字节数；
 - <0：失败；

3.1.3 i2c_master_send()

- 函数原型：int i2c_master_send(const struct i2c_client *client, const char *buf, int count)

- 作用：通过封装 i2c_transfer() 完成一次 I2c 发送操作。
- 参数：
 - client：指向当前 I2C 从设备的实例；
 - buf：要发送的数据；
 - count：要发送的数据长度
- 返回：
 - >0：成功发送的字节数；
 - <0：失败；

3.1.4 i2c_smbus_read_byte()

- 函数原型：s32 i2c_smbus_read_byte(const struct i2c_client *client)
- 作用：从 I2C 总线读取一个字节。（内部是通过 i2c_transfer() 实现，以下几个接口同。）
- 参数：
 - client：指向当前的 I2C 从设备
- 返回：
 - >0：读取到的数据；
 - <0：失败；

3.1.5 i2c_smbus_write_byte()

- 函数原型：s32 i2c_smbus_write_byte(const struct i2c_client *client, u8 value)
- 作用：从 I2C 总线写入一个字节。
- 参数：
 - client：指向当前的 I2C 从设备；
 - value：要写入的数值
- 返回：
 - 0：成功；
 - <0：失败；

3.1.6 i2c_smbus_read_byte_data()

- 函数原型：s32 i2c_smbus_read_byte_data(const struct i2c_client *client, u8 command)
- 作用：从 I2C 设备指定偏移处读取一个字节。

- 参数：
 - client：指向当前的 I2C 从设备；
 - command：I2C 协议数据的第 0 字节命令码（即偏移值）；
- 返回：
 - >0：读取到的数据；
 - <0：失败；

3.1.7 i2c_smbus_write_byte_data()

- 函数原型：s32 i2c_smbus_write_byte_data(const struct i2c_client *client, u8 command, u8 value)
- 作用：从 I2C 设备指定偏移处写入一个字节。
- 参数：
 - client：指向当前的 I2C 从设备；
 - command：I2C 协议数据的第 0 字节命令码（即偏移值）；
 - value：要写入的数值；
- 返回：
 - 0：成功；
 - <0：失败；

3.1.8 i2c_smbus_read_word_data()

- 函数原型：s32 i2c_smbus_read_word_data(const struct i2c_client *client, u8 command)
- 作用：从 I2C 设备指定偏移处读取一个 word 数据（两个字节，适用于 I2C 设备寄存器是 16 位的情况）。
- 参数：
 - client：指向当前的 I2C 从设备；
 - command：I2C 协议数据的第 0 字节命令码（即偏移值）；
- 返回：
 - >0：读取到的数据；
 - <0：失败；

3.1.9 i2c_smbus_write_word_data()

- 函数原型：s32 i2c_smbus_write_word_data(const struct i2c_client *client, u8 command, u16 value)
- 作用：从 I2C 设备指定偏移处写入一个 word 数据（两个字节）。
- 参数：
 - client：指向当前的 I2C 从设备；
 - command：I2C 协议数据的第 0 字节命令码（即偏移值）；
 - value：要写入的数值
- 返回：
 - 0：成功；
 - <0：失败；

3.1.10 i2c_smbus_read_block_data()

- 函数原型：s32 i2c_smbus_read_block_data(const struct i2c_client *client, u8 command, u8 *values)
- 作用：从 I2C 设备指定偏移处读取一块数据。
- 参数：
 - client：指向当前的 I2C 从设备；
 - command：I2C 协议数据的第 0 字节命令码（即偏移值）；
 - values：用于保存读取到的数据；
- 返回：
 - >0：读取到的数据长度；
 - <0：失败；

3.1.11 i2c_smbus_write_block_data()

- 函数原型：s32 i2c_smbus_write_block_data(const struct i2c_client *client, u8 command, u8 length, const u8 *values)
- 作用：从 I2C 设备指定偏移处写入一块数据（长度最大 32 字节）。
- 参数：
 - client：指向当前的 I2C 从设备；
 - command：I2C 协议数据的第 0 字节命令码（即偏移值）；
 - length：要写入的数据长度；
 - values：要写入的数据；
- 返回：

- 0：成功；
- <0：失败；

3.2 i2c 用户态调用接口

i2c 的操作在内核中是当做字符设备来操作的，可以通过利用文件读写接口（open, write, read, ioctl）等操作内核目录中的/dev/i2c-* 文件来调用相关的接口，i2c 相关的操作定义在 i2c-dev.c 里面，本节将介绍比较重要的几个接口：

3.2.1 i2cdev_open()

- 函数原型：static int i2cdev_open(struct inode *inode, struct file *file)
- 作用：程序（C 语言等）使用 open(file) 时调用的函数。打开一个 i2c 设备，可以像文件读写的方式往 i2c 设备中读写数据
- 参数：
 - inode: inode 节点；
 - file: file 结构体；
- 返回：文件描述符

3.2.2 i2cdev_read()

- 函数原型：static ssize_t i2cdev_read(struct file *file, char __user *buf, size_t count, loff_t *offset)
- 作用：程序（C 语言等）调用 read() 时调用的函数。像往文件里面读数据一样从 i2c 设备中读数据。底层调用 i2c_xfer 传输数据
- 参数：
 - file: file 结构体；
 - buf, 写数据 buf；
 - offset, 文件偏移。
- 返回：
 - 非空：返回读取的字节数；
 - <0：失败；

3.2.3 i2cdev_write()

- 函数原型：static ssize_t i2cdev_write(struct file *file, const char __user *buf, size_t count, loff_t *offset)
- 作用：程序（C 语言等）调用 write() 时调用的函数。像往文件里面写数据一样往 i2c 设备中写数据。底层调用 i2c_xfer 传输数据
- 参数：
 - file: file 结构体;
 - buf: 读数据 buf;
 - offset, 文件偏移。
- 返回：
 - 0: 成功;
 - <0: 失败;

3.2.4 i2cdev_ioctl()

- 函数原型：static long i2cdev_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
- 作用：程序（C 语言等）调用 ioctl() 时调用的函数。像对文件管理 i/o 一样对 i2c 设备管理。该功能比较强大，可以修改 i2c 设备的地址，往 i2 设备里面读写数据，使用 smbus 等等，详细的可以查阅该函数。
- 参数：
 - file: file 结构体;
 - cmd: 指令;
 - arg: 其他参数。
- 返回：
 - 0: 成功;
 - <0: 失败;

4 模块使用范例

4.1 利用 i2c-core 接口读写 TWI 设备

在内核源码中有现成的 i2c 设备驱动实例：tina/lichee/kernel/linux-5.4/drivers/misc/eeeprom/at24.c，这是一个 EEPROM 的 I2C 设备驱动，为了验证 I2C 总线驱动，所以其中通过 sysfs 节点实现读写访问。下面对这个文件的一些关键点进行展示介绍：

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/of_device.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/mutex.h>
#include <linux/mod_devicetable.h>
#include <linux/bitops.h>
#include <linux/jiffies.h>
#include <linux/property.h>
#include <linux/acpi.h>
#include <linux/i2c.h>
#include <linux/nvmem-provider.h>
#include <linux/regmap.h>
#include <linux/pm_runtime.h>
#include <linux/gpio/consumer.h>

#define EEPROM_ATTR(_name) \
{ \
    .attr = { .name = #_name, .mode = 0444 }, \
    .show = _name##_show, \
}

struct i2c_client *this_client;

static const struct i2c_device_id at24_ids[] = {
    { "24c16", 0 },
    { /* END OF LIST */ }
};
MODULE_DEVICE_TABLE(i2c, at24_ids);

static int eeeprom_i2c_rxdata(char *rxdata, int length)
{
    int ret;

    struct i2c_msg msgs[] = {
        {
            .addr = this_client->addr,
            .flags = 0,
            .len = 1,
```

```

        .buf    = &rxdata[0],
    },
    {
        .addr    = this_client->addr,
        .flags   = I2C_M_RD,
        .len     = length,
        .buf     = &rxdata[1],
    },
};

ret = i2c_transfer(this_client->adapter, msgs, 2);
if (ret < 0)
    pr_info("%s i2c read eeprom error: %d\n", __func__, ret);

return ret;
}

static int eeprom_i2c_txdata(char *txdata, int length)
{
    int ret;

    struct i2c_msg msg[] = {
        {
            .addr    = this_client->addr,
            .flags   = 0,
            .len     = length,
            .buf     = txdata,
        },
    };

    ret = i2c_transfer(this_client->adapter, msg, 1);
    if (ret < 0)
        pr_err("%s i2c write eeprom error: %d\n", __func__, ret);

    return 0;
}

static ssize_t read_show(struct kobject *kobj, struct kobj_attribute *attr,
                        char *buf)
{
    int i;
    u8 rxdata[4];
    rxdata[0] = 0x1;
    eeprom_i2c_rxdata(rxdata, 3);

    for(i=0;i<4;i++)
        printk("rxdata[%d]: 0x%x\n", i, rxdata[i]);

    return sprintf(buf, "%s\n", "read end!");
}

static ssize_t write_show(struct kobject *kobj, struct kobj_attribute *attr,
                        char *buf)
{
    int i;
    static u8 txdata[4] = {0x1, 0xAA, 0xBB, 0xCC};

    for(i=0;i<4;i++)
        printk("txdata[%d]: 0x%x\n", i, txdata[i]);
}

```



```
    eeprom_i2c_txdata(txdata,4);

    txdata[1]++;
    txdata[2]++;
    txdata[3]++;

    return sprintf(buf, "%s\n", "write end!");
}

static struct kobj_attribute read  = EEPROM_ATTR(read);
static struct kobj_attribute write = EEPROM_ATTR(write);

static const struct attribute *test_attrs[] = {
    &read.attr,
    &write.attr,
    NULL,
};

static int at24_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
    int err;
    this_client = client;
    printk("1..at24_probe \n");
    err = sysfs_create_files(&client->dev.kobj, test_attrs);
    printk("2..at24_probe \n");
    if(err){
        printk("sysfs_create_files failed\n");
    }
    printk("3..at24_probe \n");
    return 0;
}

static int at24_remove(struct i2c_client *client)
{
    return 0;
}

static struct i2c_driver at24_driver = {
    .driver = {
        .name = "at24",
        .owner = THIS_MODULE,
    },
    .probe = at24_probe,
    .remove = at24_remove,
    .id_table = at24_ids,
};

static int __init at24_init(void)
{
    printk("%s    %d\n", __func__, __LINE__);

    return i2c_add_driver(&at24_driver);
}
module_init(at24_init);

static void __exit at24_exit(void)
{
    printk("%s() %d - \n", __func__, __LINE__);
}
```

```
i2c_del_driver(&at24_driver);  
}  
module_exit(at24_exit);
```

4.2 利用用户态接口读写 TWI 设备

如果配置了 i2c devices interface，可以直接利用文件读写函数来操作 I2C 设备。下面这个程序直接读取 /dev/i2c-* 来读写 i2c 设备：

```
#include <sys/ioctl.h>  
#include <fcntl.h>  
#include <linux/i2c-dev.h>  
#include <linux/i2c.h>  
#define CHIP "/dev/i2c-1"  
#define CHIP_ADDR 0x50  
int main()  
{  
    unsigned char rddata;  
    unsigned char rdaddr[2] = {0, 0}; /* 将要读取的数据在芯片中的偏移量 */  
    unsigned char wrbuf[3] = {0, 0, 0x3c}; /* 要写的数据，头两字节为偏移量 */  
    printf("hello, this is i2c tester\n");  
    int fd = open(CHIP, O_RDWR);  
    if (fd < 0)  
    {  
        printf("open \"CHIP\" failed\n");  
        goto exit;  
    }  
    if (ioctl(fd, I2C_SLAVE_FORCE, CHIP_ADDR) < 0)  
    { /* 设置芯片地址 */  
        printf("ioctl:set slave address failed\n");  
        goto close;  
    }  
    printf("input a char you want to write to E2PROM\n");  
    wrbuf[2] = getchar();  
    printf("write return:%d, write data:%x\n", write(fd, wrbuf, 3), wrbuf[2]);  
    sleep(1);  
    printf("write address return: %d\n", write(fd, rdaddr, 2)); /* 读取之前首先设置读取的偏移量 */  
    printf("read data return:%d\n", read(fd, &rddata, 1));  
    printf("rddata: %c\n", rddata);  
    close(fd);  
    exit:  
    return 0;  
}
```

5 FAQ

5.1 调试方法

5.1.1 调试工具

5.1.1.1 i2c-tools 调试工具

i2c-tools 是一个开源工具，专门用来调试 I2C 设备。可以用 i2c-tools 来获取 i2c 设备的相关信息（默认集成在内核里面），并且读写相关的 i2c 设备的数据。i2c-tools 主要是通过读写/dev/i2c-* 文件获取 I2C 设备，所以需要在 kernel/linux-4.9 的 menuconfig 里面把 I2C 的 device interface 节点打开，具体的 i2c-tools 使用方法如下

```
i2cdetect -l //获取i2c设备信息
i2cdump -y i2c-number i2c-reg //把相关的i2c设备数据dump出来，如i2cdump -y 1 0x50
i2cget -y i2c-number i2c-reg data_rege //读取i2c设备某个地址的数据，如i2cget -y 1 0x50 1
i2cset -y i2c-number i2c-reg data_rege data //往i2c设备某个地址写数据，如i2cset -y 1 0x50 1 1
```

5.1.2 调试节点

5.1.2.1 /sys/module/i2c_sunxi/parameters/transfer_debug

此节点文件的功能是打开某个 TWI 通道通信过程的调试信息。缺省值是-1，不会打印任何通道的通信调试信息。

打开通道 x 通信过程调试信息的方法：

```
echo x > /sys/module/i2c_sunxi/parameters/transfer_debug
```

关闭通信过程调试信息的方法：

```
echo -1 > /sys/module/i2c_sunxi/parameters/transfer_debug
```

5.1.2.2 /sys/devices/soc.2/1c2ac00.twi.0/info

此节点文件可以打印出当前 TWI 通道的一些硬件资源信息。

```
cat /sys/devices/soc.2/1c2ac00.twi.0/info
```

5.1.2.3 /sys/devices/soc.2/1c2ac00.twi/status

此节点文件可以打印出当前 TWI 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/soc.2/1c2ac00.twi/status
```

5.2 常见问题

5.2.1 TWI 数据未完全发送

问题现象：incomplete xfer。具体的 log 如下所示：

```
[ 1658.926643] sunxi_i2c_do_xfer()1936 - [i2c0] incomplete xfer (status: 0x20, dev addr: 0x50)
[ 1658.926643] sunxi_i2c_do_xfer()1936 - [i2c0] incomplete xfer (status: 0x48, dev addr: 0x50)
```

问题分析：此错误表示主控已经发送了数据（status 值为 0x20 时，表示发送了 SLAVE ADDR + WRITE；status 值为 0x48 时，表示发送了 SLAVE ADDR + READ），但是设备没有回 ACK，这表明设备无响应，应该检查是否未接设备、接触不良、设备损坏和上电时序不正确导致的设备未就绪等问题。

问题排查步骤：

- 步骤 1：通过设备树里面的配置信息，核对引脚配置是否正确。每组 TWI 都有好几组引脚配置。
- 步骤 2：更换 TWI 总线下的设备为 at24c16，用 i2ctools 读写 at24c16 看看是否成功，成功则表明总线工作正常；
- 步骤 3：排查设备是否可以正常工作以及设备与 I2C 之间的硬件接口是否完好；
- 步骤 4：详细了解当前需要操作的设备的初始化方法，工作时序，使用方法，排查因初始化设备不正确导致通讯失败；
- 步骤 5：用示波器检查 TWI 引脚输出波形，查看波形是否匹配。

5.2.2 TWI 起始信号无法发送

问题现象：START can't sendout!。具体的 log 如下所示：

```
sunxi_i2c_do_xfer()1865 - [i2c1] START can't sendout!
```

问题分析：此错误表示 TWI 无法发送起始信号，一般跟 TWI 总线的引脚配置以及时钟配置有关。应该检查引脚配置是否正确，时钟配置是否正确，引脚是否存在上拉电阻等等。

问题排查步骤：

- 步骤 1：重新启动内核，通过查看 log，分析 TWI 是否成功初始化，如若存在引脚配置问题，应核对引脚信息是否正确
- 步骤 2：根据原理图，查看 TWI-SCK 和 TWI-SDA 是否经过合适的上拉电阻接到 3.3v 电压；
- 步骤 3：用万用表量 SDA 与 SCL 初始电压，看电压是否在 3.3V 附近（断开此 TWI 控制器所有外设硬件连接与软件通讯进程）；
- 步骤 4：核查引脚配置以及 clk 配置是否进行正确设置；
- 步骤 5：测试 PIN 的功能是否正常，利用寄存器读写的方式，将 PIN 功能直接设为 INPUT 功能（echo [reg] [val] > /sys/class/sunxi_dump/write），然后将 PIN 上拉和接地改变 PIN 状态，读 PIN 的状态（echo [reg,reg] > /sys/class/sunxi_dump/dump;cat dump），看是否匹配。
- 步骤 6：测试 CLK 的功能是否正常，利用寄存器读写的方式，将 TWI 的 CLK gating 等打开，（echo [reg] [val] > /sys/class/sunxi_dump/write），然后读取相应 TWI 的寄存器信息，读 TWI 寄存器的数据（echo [reg] ,[len]> /sys/class/sunxi_dump/dump），查看寄存器数据是否正常。

5.2.3 TWI 终止信号无法发送

问题现象：STOP can't sendout。具体的 log 如下所示：

```
twi_stop()511 - [i2c4] STOP can't sendout!  
sunxi_i2c_core_process()1726 - [i2c4] STOP failed!
```

问题分析：此错误表示 TWI 无法发送终止信号，一般跟 TWI 总线的引脚配置。应该检查引脚配置是否正确，引脚电压是否稳定等等。

问题排查步骤：

- 步骤 1：根据原理图，查看 TWI-SCK 和 TWI-SDA 是否经过合适的上拉电阻接到 3.3v 电压；
- 步骤 2：用万用表量 SDA 与 SCL 初始电压，看电压是否在 3.3V 附近（断开此 TWI 控制器所有外设硬件连接与软件通讯进程）；
- 步骤 3：测试 PIN 的功能是否正常，利用寄存器读写的方式，将 PIN 功能直接设为 INPUT 功能（echo [reg] [val] > /sys/class/sunxi_dump/write），然后将 PIN 上拉和接地改变 PIN 状态，读 PIN 的状态（echo [reg,reg] > /sys/class/sunxi_dump/dump;cat dump），看是否匹配；

- 步骤 4: 查看设备树配置，把其他用到 SCK/SDA 引脚的节点关闭，重新测试 I2C 通信功能。

5.2.4 TWI 传送超时

问题现象：xfer timeout。具体的 log 如下所示：

```
[123.681219] sunxi_i2c_do_xfer()1914 - [i2c3] xfer timeout (dev addr:0x50)
```

问题分析：此错误表示主控已经发送完起始信号，但是在与设备通信的过程中无法正常完成数据发送与接收，导致最终没有发出终止信号来结束 I2C 传输，导致的传输超时问题。应该检查引脚配置是否正常，CLK 配置是否正常，TWI 寄存器数据是否正常，是否有其他设备干扰，中断是否正常等问题。

问题排查步骤：

- 步骤 1: 核实 TWI 控制器配置是否正确；
- 步骤 2: 根据原理图，查看 TWI-SCK 和 TWI-SDA 是否经过合适的上拉电阻接到 3.3v 电压；
- 步骤 3: 用万用表量 SDA 与 SCL 初始电压，看电压是否在 3.3V 附近（断开此 TWI 控制器所有外设硬件连接与软件通讯进程）；
- 步骤 4: 关闭其他 TWI 设备，重新进行烧录测试 TWI 功能是否正常；
- 步骤 4: 测试 PIN 的功能是否正常，利用寄存器读写的方式，将 PIN 功能直接设为 INPUT 功能（echo [reg] [val] > /sys/class/sunxi_dump/write），然后将 PIN 上拉和接地改变 PIN 状态，读 PIN 的状态（echo [reg,reg] > /sys/class/sunxi_dump/dump;cat dump），看是否匹配；
- 步骤 5: 测试 CLK 的功能是否正常，利用寄存器读写的方式，将 TWI 的 CLK gating 等打开，（echo [reg] [val] > /sys/class/sunxi_dump/write），然后读取相应 TWI 的寄存器信息，读 TWI 寄存器的数据（echo [reg] ,[len]> /sys/class/sunxi_dump/dump），查看寄存器数据是否正常；
- 步骤 7: 根据相关的 LOG 跟踪 TWI 代码执行流程，分析报错原因。

著作权声明

版权所有 © 2021 珠海全志科技股份有限公司。保留一切权利。

本文档及内容受著作权法保护，其著作权由珠海全志科技股份有限公司（“全志”）拥有并保留一切权利。

本文档是全志的原创作品和版权财产，未经全志书面许可，任何单位和个人不得擅自摘抄、复制、修改、发表或传播本文档内容的部分或全部，且不得以任何形式传播。

商标声明

、 全志科技、（不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本文档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

免责声明

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本文档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本文档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本文档作为使用指导仅供参考。由于产品版本升级或其他原因，本文档内容有可能修改，如有变更，恕不另行通知。全志尽全力在本文档中提供准确的信息，但并不确保内容完全没有错误，因使用本文档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本文档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本文档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。