SUNXI TWI

模块使用说明

1.0

2019.08.30

文档履历

版本号	日期	制/修订人	内容描述
1.0	2019.08.30		初始发布版本

目录

1. 概述

1.1 编写目的

介绍 Linux 内核中 I2C 子系统的接口及使用方法。

1.2 适用范围

适用于 sunxi 硬件平台。

1.3 相关人员

I2C 总线驱动、I2C 设备驱动的使用人员等。

2. 模块介绍

2.1 模块功能

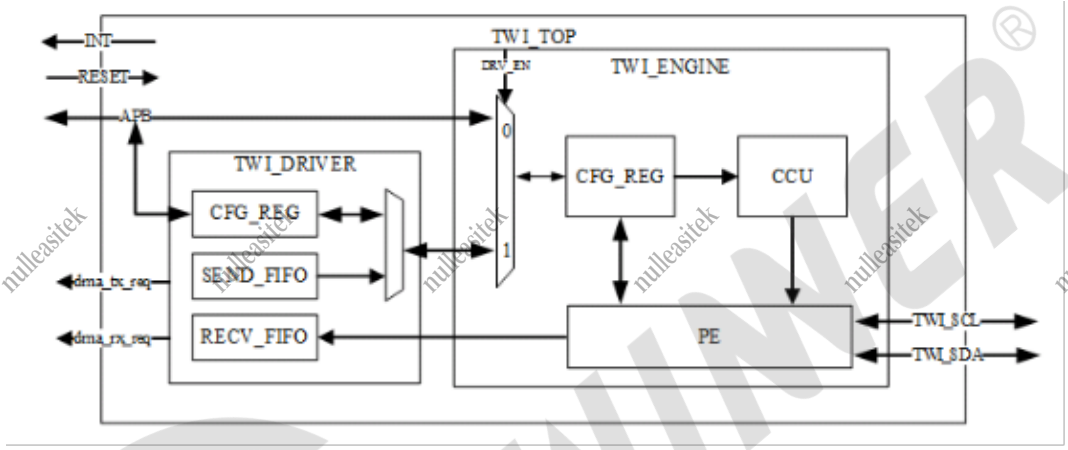

图 1: TWI 控制器

TWI 控制器的框图如上所示。该控制器支持上的 TWI 通路标准通信速率为 100Kbps，最高通信速率可以达到 400K bps。CPUX 的 TWI 时钟源来自 APB2，CPUS 的 R-TWI 时钟源来自 APBS。当 TWI 工作在 master 的情况下，TWI 驱动支持包传输和 DMA 运输方式。更多细节请查阅相关的 Spec 文档。

2.2 系统结构框图

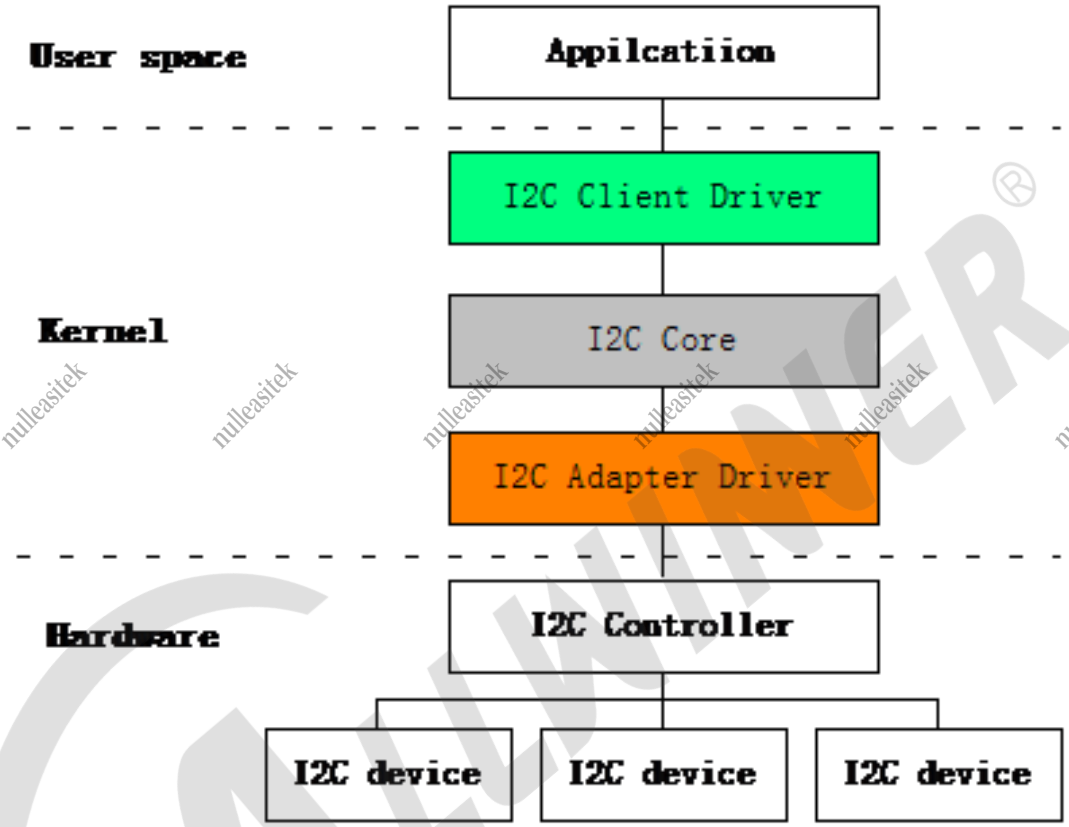

图 2: TWI 模块结构框图

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 3 个部分：

1. I2C 核心（I2C Core）：实现对 I2C 总线驱动及 I2C 设备驱动的管理；
2. I2C 总线驱动（I2C adapter driver）：针对不同类型的 I2C 控制器，实现对 I2C 总线访问的具体方法；

3. I2C 设备驱动（I2C client driver）：针对特定的 I2C 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。

需要了解更多关于 Linux 的 I2C 体系的知识，可以查阅 I2C 子系统基础知识

2.3 相关术语介绍

术语	解释说明
I2C	Inter-Integrated Circuit，用于 CPU 与外设通信的一种串行总线。
TWI	Normal Two Wire Interface，Sunxi 平台中的 I2C 控制器名称。
I2C_dapter	I2C 总线适配器。IIC 总线的控制器，在物理上连接若干个 IIC 设备。
I2C_algorithm	I2C 总线驱动程序。描述 I2C 总线适配器与 I2C 设备之间的通信方法
I2C Client	I2C 从设备。
I2C Driver	I2C 设备驱动。

2.4 模块配置

在命令行进入内核根目录，执行 make ARCH=arm menuconfig 进入配置主界面，并按以下步骤操作：

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

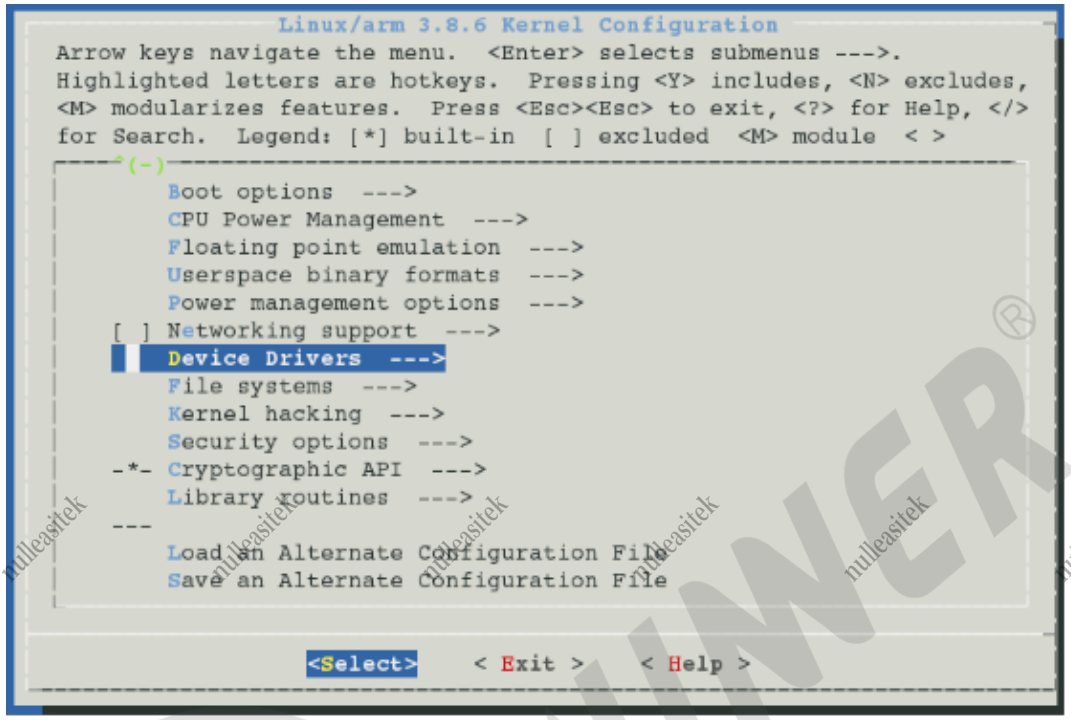

图 3: Device Driver

- 然后，选择 I2C support 选项，进入下一级配置，如下图所示：

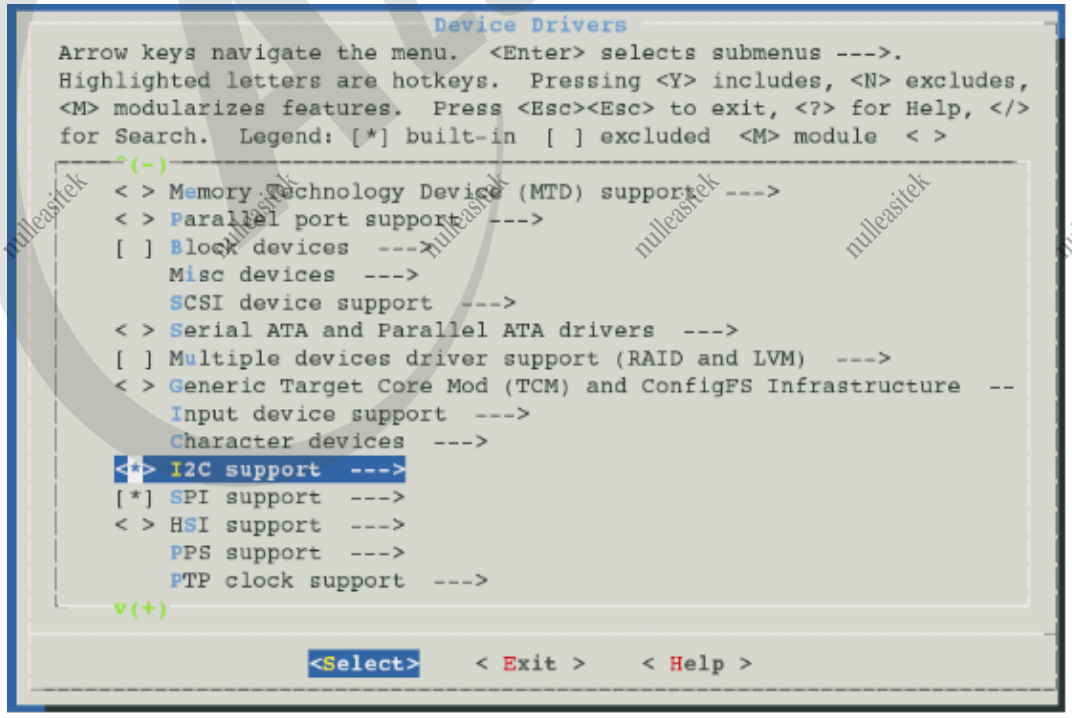

图 4: I2C support

- 随后，选择 I2C device interface，如下图所示：

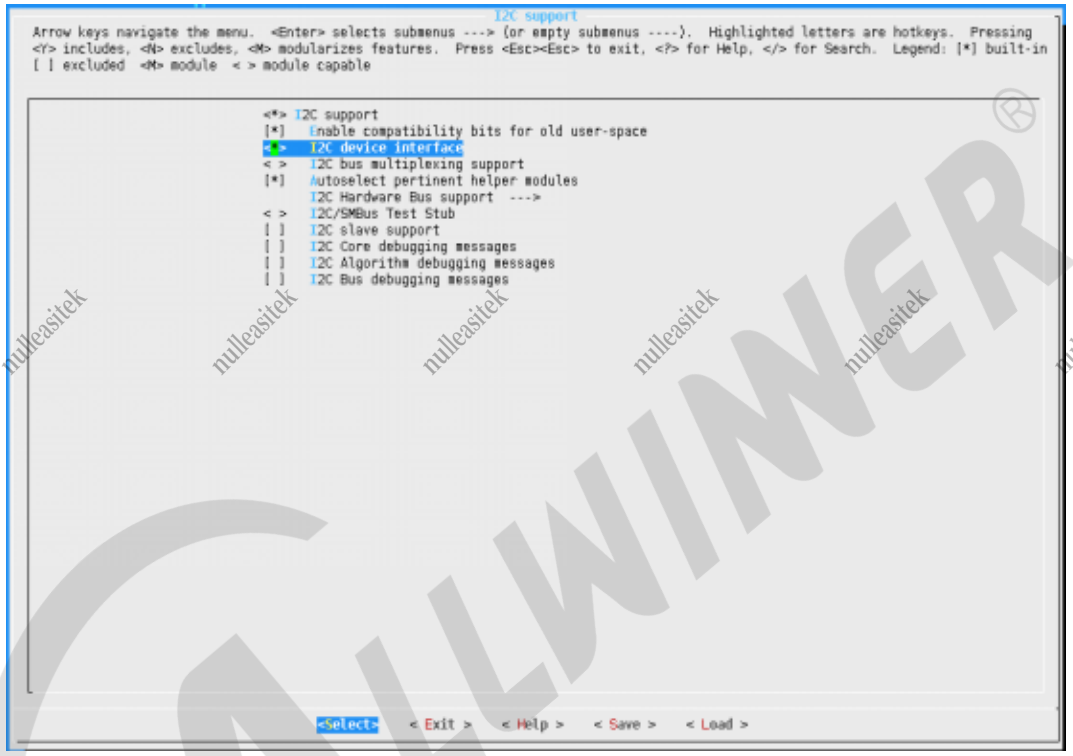

图 5: I2C device interface

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

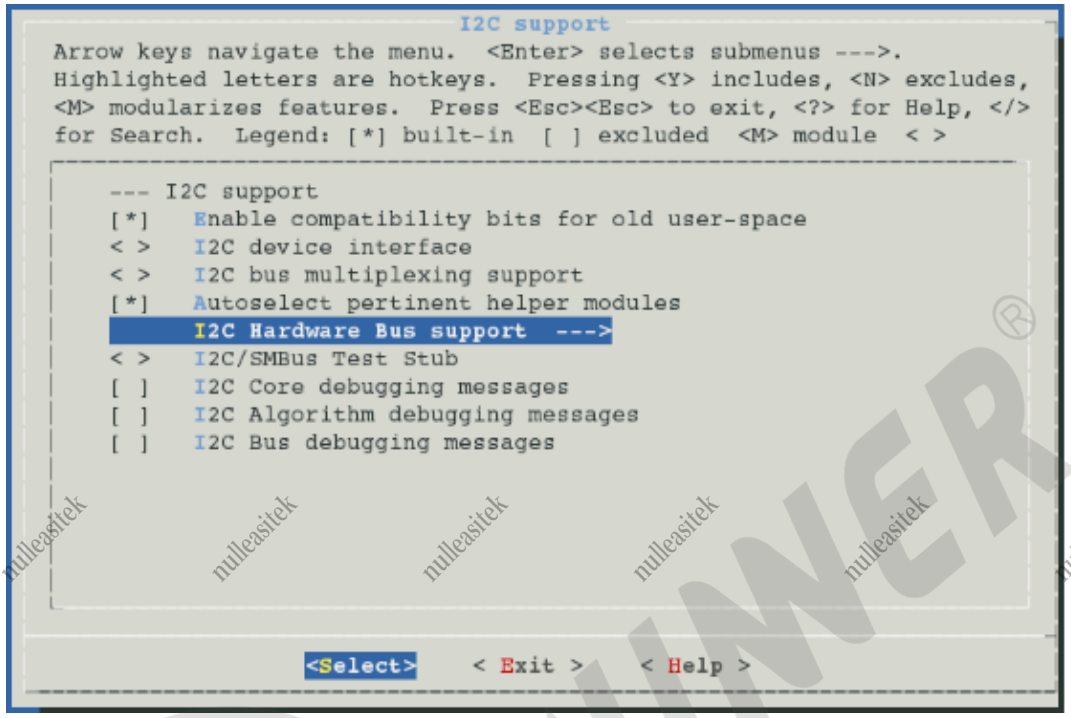

图 6: 2C HardWare Bus support

- 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：

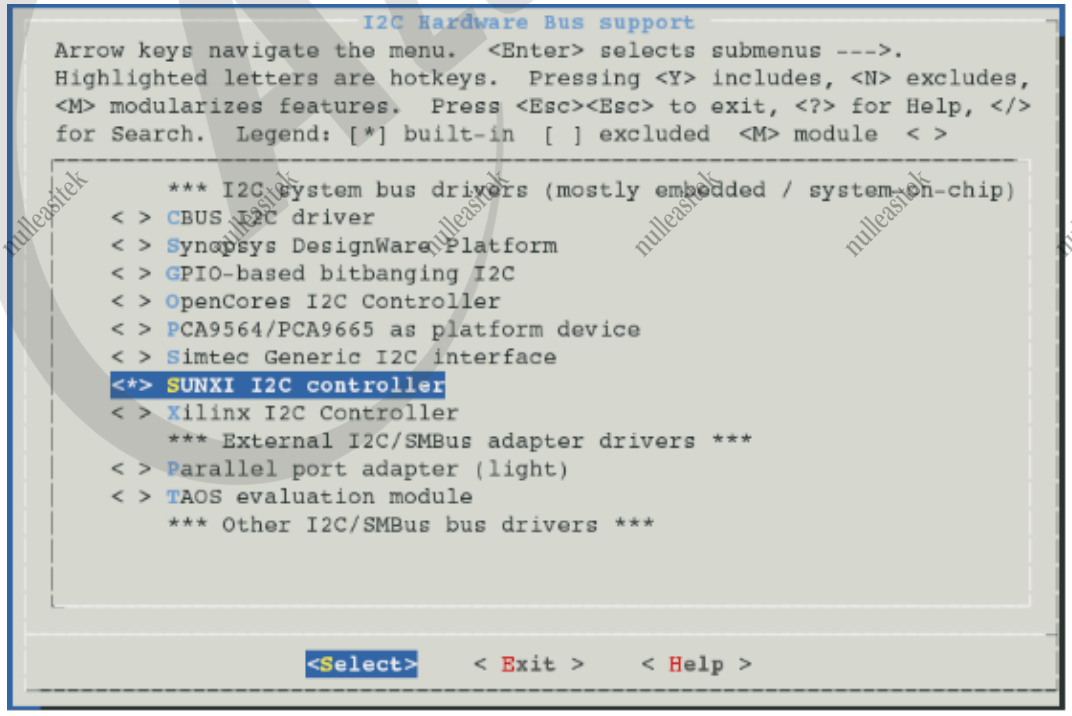

图 7: SUNXI I2C controller

- 如果当前的配置是给 FPGA 使用，因为 FPGA 板上只有一个 TWI 控制器，还需要多做一项配置，配置 I2C 外设要使用哪个 TWI 通道。如下图所示：

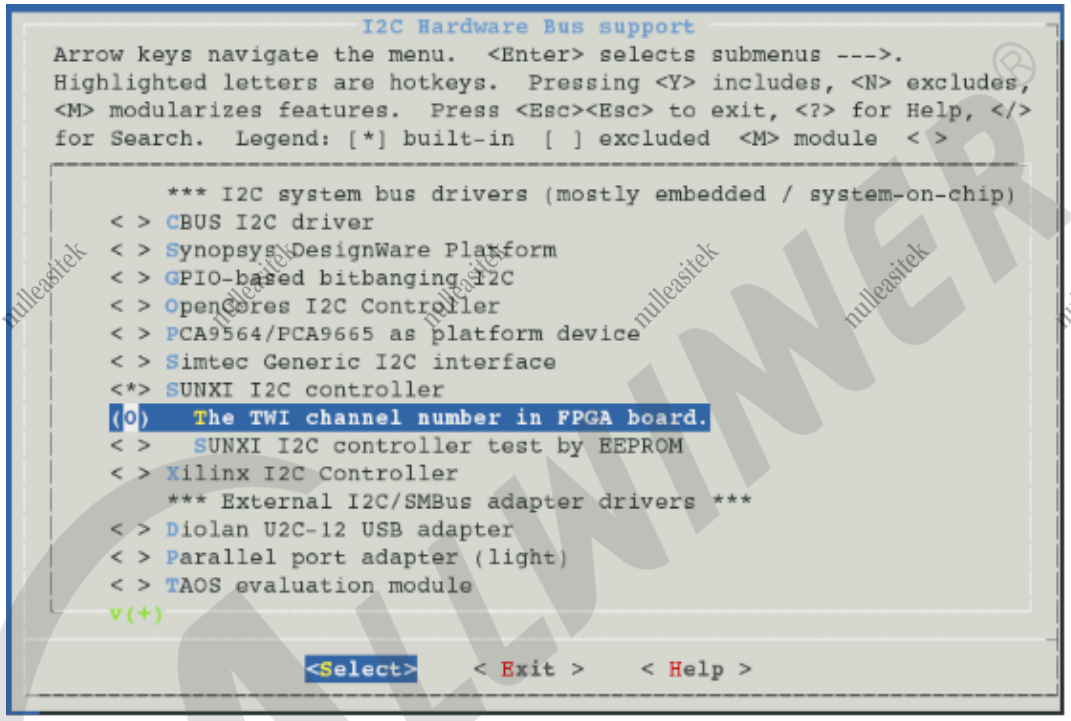

图 8: STWI 通道

2.5 设备树配置

在不同的 Sunxi 硬件平台中，TWI 控制器的数目也不同，但对于每一个 TWI 控制器来说，在 Device Tree 中配置参数相似，如下：

```
twi0: twi@0x01c2ac00{
    compatible = "allwinner,sun50i-twi"; //具体的设备，用于驱动和设备的绑定
    device_type = "twi0"; //设备节点名称，用于sys_config.fex匹配
    reg = <0x0 0x01c2ac00 0x0 0x400>; //设备使用的地址
    interrupts = <GIC_SPI 6 IRQ_TYPE_LEVEL_HIGH>; //设备使用的中断
```

```
        clocks = <&clk_twi0>; //设备使用的时钟
        clock-frequency = <400000>; //TWI控制器的时钟频率
        pinctrl-names = "default"; //设备使用的pin脚名称
        pinctrl-0 = <&twi0_pins_a>; //设备使用的pin脚配置
        twi_drv_used = <1>; //使用DMA传输数据
        status = "okay"; //设备是否使用
    };
```

其中 twi0_pins_a 的配置如下所示

```
    twi0_pins_a: twi0@0 {
        allwinner,pins = "PI3", "PI4"; //使用的引脚
        allwinner,pname = "twi0_scl", "twi0_sda"; //相关的引脚用途
        allwinner,function = "twi0"; //引脚功能
        allwinner,muxsel = <5>; //引脚配置
        allwinner,drive = <1>;
        allwinner,pull = <0>;
    };
```

为了在 I2C 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：

```
    aliases {
        soc_twi0 = &twi0;
        soc_twi1 = &twi1;
        soc_twi2 = &twi2;
        soc_twi3 = &twi3;
        ...
    };
```

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。

对于 I2C 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 I2C 设备。

```
    twi0: twi@0x01c2ac00{
        #address-cells = <1>;
        #size-cells = <0>;
        ...
```

```
        eeprom@50 {
            compatible = "atmel,24c16"; //具体的设备名称，用于驱动和设备的绑定
            reg = <0x50>; //设备使用的寄存器地址
        };
    };
```

2.6 源码模块结构

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

```
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
├── 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
```

3. 接口设计

3.1 内部接口

3.1.1 i2c_transfer()

- 函数原型：int i2c_transfer(struct i2c_adapter adap, struct i2c_msg msgs, int num)
- 功能描述：完成 I2C 总线和 I2C 设备之间的一定数目的 I2C message 交互。
- 参数说明：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
- 返回值：>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)
- 功能描述：通过封装 i2c_transfer() 完成一次 I2c 接收操作。
- 参数说明：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
- 返回值：>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)
- 功能描述：通过封装 i2c_transfer() 完成一次 I2c 发送操作。
- 参数说明：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
- 返回值：>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)
- 功能描述：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。）
- 参数说明：client，指向当前的 I2C 从设备
- 返回值：>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)
- 功能描述：从 I2C 总线写入一个字节。
- 参数说明：client，指向当前的 I2C 从设备；value，要写入的数值
- 返回值：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)
- 功能描述：从 I2C 设备指定偏移处读取一个字节。
- 参数说明：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）
- 返回值：>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)
- 功能描述：从 I2C 设备指定偏移处写入一个字节。
- 参数说明：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
- 返回值：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)
- 功能描述：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）。
- 参数说明：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）
- 返回值：>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)
- 功能描述：从 I2C 设备指定偏移处写入一个 word 数据（两个字节）。
- 参数说明：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
- 返回值：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)
- 功能描述：从 I2C 设备指定偏移处读取一块数据。
- 参数说明：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
- 返回值：>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)
- 功能描述：从 I2C 设备指定偏移处写入一块数据（长度最大 32 字节）。

- 参数说明：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
- 返回值：0，成功；<0，失败

3.2 外部接口

i2c 的操作在内核中是当做字符设备来操作的，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：

3.3 i2cdev_open()

- 函数原型：static int i2cdev_open(struct inode *inode, struct file *file)
- 功能描述：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
- 参数说明：inode：inode 节点；file：file 结构体
- 返回值：文件描述符

3.3.1 i2cdev_read()

- 函数原型：static ssize_t i2cdev_read(struct file *file, char __user *buf, size_t count,loff_t *offset)
- 功能描述：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
- 参数说明：file，file 结构体；buf，写数据 buf；offset, 文件偏移。
- 返回值：成功返回读取的字节数，失败返回负数

3.3.2 i2cdev_write()

- 函数原型：static ssize_t i2cdev_write(struct file *file, const char __user *buf,size_t count, loff_t *offset)
- 功能描述：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

- 参数说明：file，file 结构体；buf，读数据 buf；offset, 文件偏移。
- 返回值：成功返回 0，失败返回负数

3.3.3 i2cdev_ioctl()

- 函数原型：static long i2cdev_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
- 功能描述：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。
- 参数说明：file，file 结构体，cmd，指令，arg，其他参数。
- 返回值：成功返回 0，失败返回负数

3.3.4 i2c-tools 调试接口

可以用 i2c-tools 来获取 i2c 设备的相关信息，以及读写相关的 i2c 设备的数据。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
```

更多关于 i2c-tools 的用法，请查看 i2c-tools 的使用说明

3.4 sysfs 调试接口

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
```

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
```

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
```

4. Demo

4.1 利用内部接口读写 I2C 设备

下面是一个 EEPROM 的 I2C 设备驱动，该设备只为了验证 I2C 总线驱动，所以其中通过 sysfs 节点实现读写访问。

```
#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 eeprom_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 利用外部接口读写 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("oictl: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. 总结

本文档介绍了 Sunxi I2C 设计总线的使用方法。

6. Declaration

This document is the original work and copyrighted property of Allwinner Technology (“Allwinner”). Reproduction in whole or in part must obtain the written approval of Allwinner and give clear acknowledgement to the copyright owner. The information furnished by Allwinner is believed to be accurate and reliable. Allwinner reserves the right to make changes in circuit design and/or specifications at any time without notice. Allwinner does not assume any responsibility and liability for its use. Nor for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of Allwinner. This datasheet neither states nor implies warranty of any kind, including fitness for any particular application.tates nor implies warranty of any kind, including fitness for any particular application.