A31 平台 IIC 设备驱动开发说明文档

2013-02-21

版本历史

版本	时间	备注
V1.0	2013-02-21	建立初始版本

目 录

1. 前言 1
1.1. 编写目的 1
1.2. 适用范围 1
1.3. 相关人员 1
2. IIC 模块介绍 2
2.1. 功能介绍 2
2.2. 硬件介绍 2
2.3. 源码结构介绍 3
2.4. 配置介绍 3
3. IIC 体系结构描述 7
4. IIC 常用数据结构描述 8
5. IIC 常用接口描述 10
6. IIC 设备驱动开发 demo 13
7. IIC 常见问题 16

1. 前言

1.1. 编写目的

了解 IIC 在 A31 平台上的开发。

1.2. 适用范围

Allwinner A31 平台。

1.3. 相关人员

A31 平台 IIC 设备驱动开发人员。

2. IIC 模块介绍

2.1. 功能介绍

对 IIC 设备的读写操作给予支持。

2.2. 硬件介绍

1）IIC 总线工作原理

IIC 总线是由数据线 SDA 和时钟 SCL 构成的串行总线，各种被控制器件均并联在这条总线上，每个器件都有一个唯一的地址识别，可以作为总线上的一个发送器件或接收器件(具体由器件的功能决定)。IIC 总线的接口电路结构如图 1 所示。

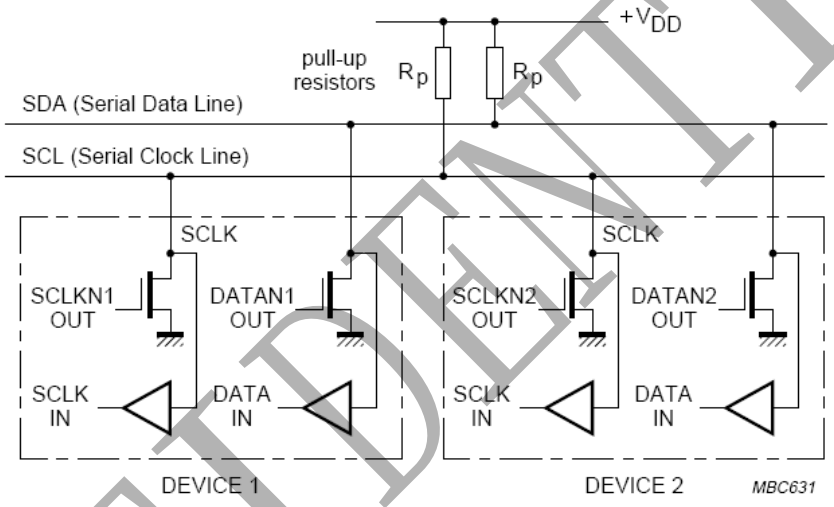

图 1 IIC 总线接口电路结构图

2）IIC 总线的几种信号状态

1. 空闲状态：SDA 和 SCL 都为高电平。
2. 开始条件(S)：SCL 为高电平时，SDA 由高电平向低电平跳变，开始传送数据。
3. 结束条件(P)：SCL 为高电平时，SDA 由低电平向高电平跳变，结束传送数据。
4. 数据有效：在 SCL 的高电平期间，SDA 保持稳定，数据有效。SDA 的改变只能发生在 SCL 的低电平期间。
5. ACK 信号：数据传输的过程中，接收器件每接收一个字节数据要产生一个 ACK 信号，向发送器件发出特定的低电平脉冲，表示已经收到数据。

3）IIC 总线基本操作

IIC 总线必须由主器件（通常为微控制器）控制，主器件产生串行时钟（SCL），同时控制总线的传输方向，并产生开始和停止条件。

数据传输中，首先由主器件产生开始条件，随后是器件的控制字节（前七位是从器件的地址，最后一位为读写位）。接下来是读写操作的数据，以及 ACK 响应信号。数据传输结束时，主器件产生停止条件。具体的过程如图 2 所示。

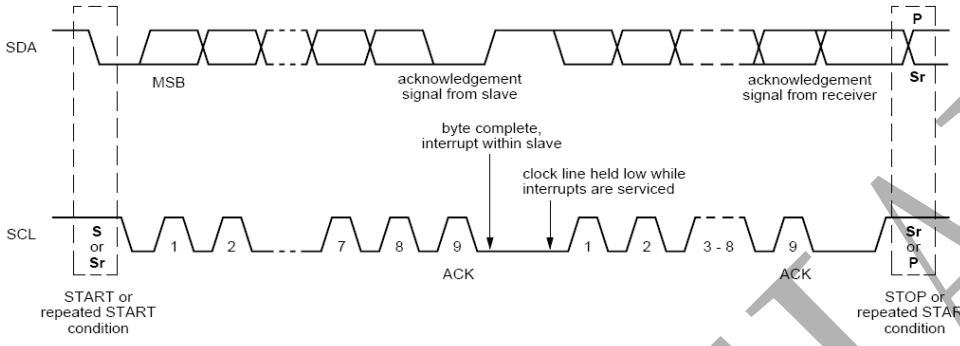

图 2 IIC 总线数据传输图

2.3. 源码结构介绍

在 drivers/i2c/目录下，包含有几个重要文件和目录，如下：

1. 文件 i2c-core.c：IIC 子系统核心功能的实现；
2. 文件 i2c-dev.c：通用的从设备驱动实现；
3. 目录 busses：里面包括基于不同平台实现的 IIC 总线控制器驱动；
4. 目录 algos：里面实现了一些 IIC 总线控制器的 algorithm。

2.4. 配置介绍

1）sys_config.fex 配置说明：

在 sys_config.fex 中有 4 组 IIC 总线可供使用，分别是 twi0、twi1、twi2 和 twi3。配置如下：

```
[twi0_para]
twi_used        = 1

[twi1_para]
twi_used        = 1

[twi2_para]
twi_used        = 1

[twi3_para]
twi_used        = 0
```

其中，若使用哪一组 IIC 总线，将对应的 twi_used 置为 1 即可。

2）menuconfig 配置说明：

对于 IIC 总线控制器的配置，可通过命令 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，如图 3 所示：

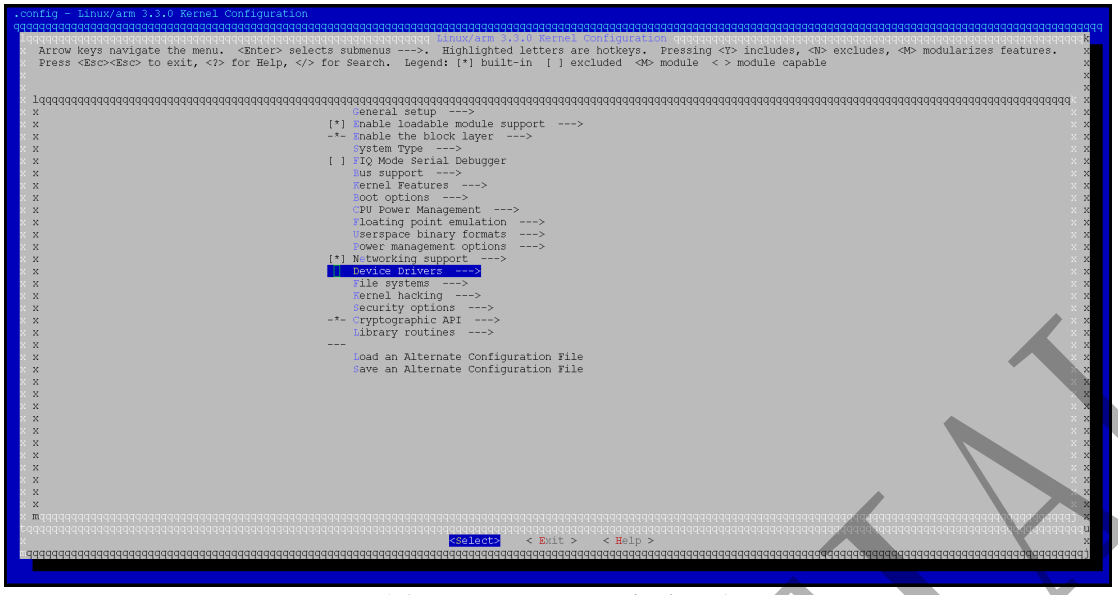

图 3 Device Drivers 选项配置

然后，选择 IIC support 选项，进入下一级配置，如图 4 所示：

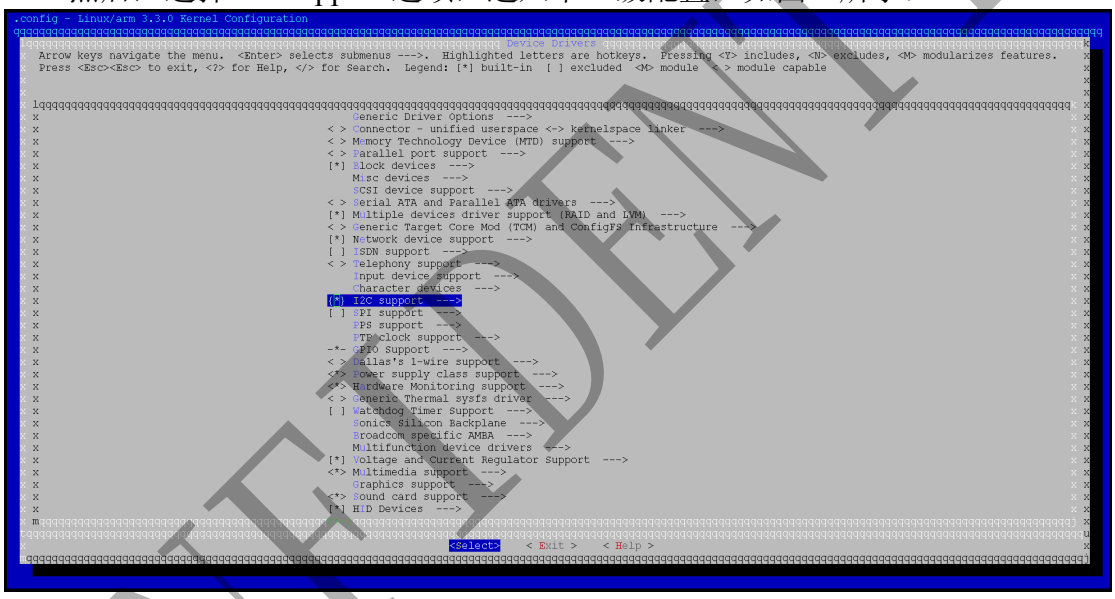

图 4 IIC support 选项配置

接着，选择 IIC HardWare Bus support 选项，进入下一级配置，如图 5 所示：

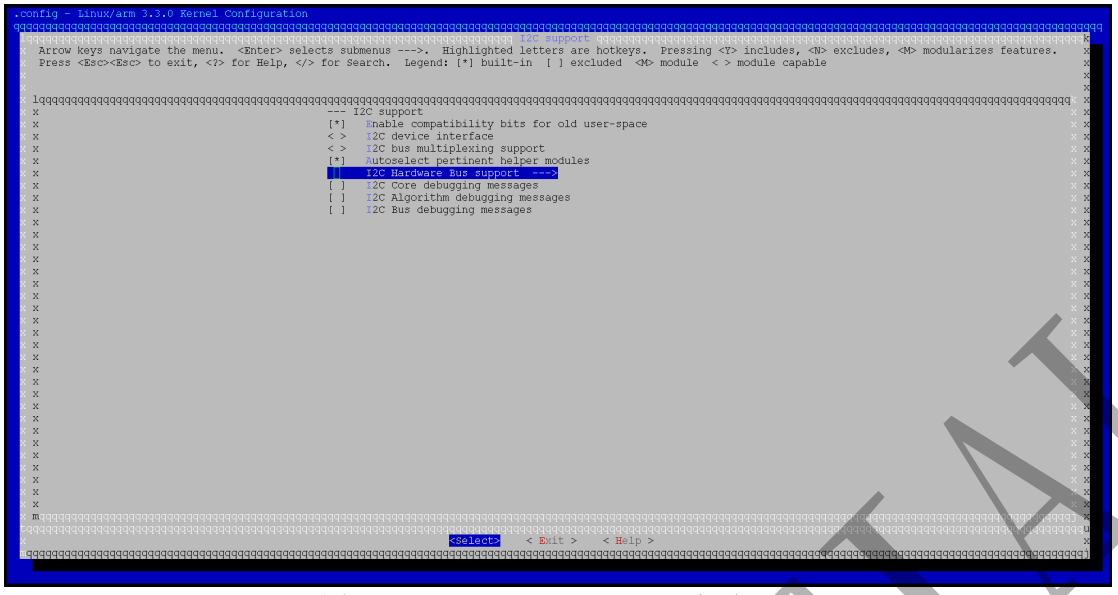

图 5 IIC HardWare Bus support 选项配置

最后，选择 AllWinner Technology SUN6I I2C interface 选项，可选择直接编译进内核，也可以选择编译成模块。如图 6 所示：

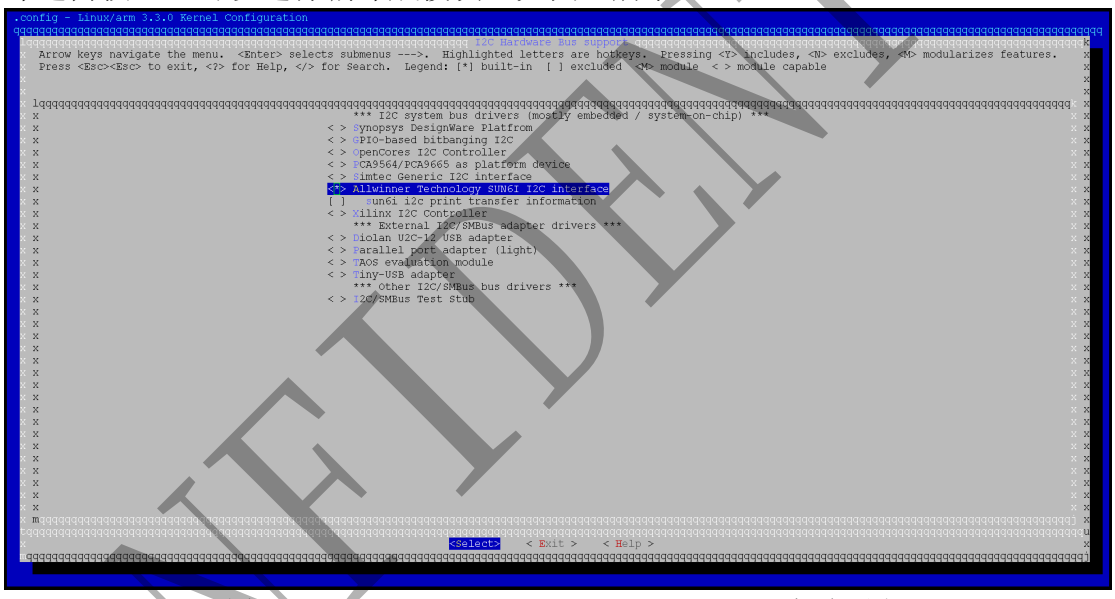

图 6 AllWinner Technology SUN6I IIC interface 选项配置

此外，若需要获取指定 IIC 总线控制器相关的调试打印信息，可选择 sun6i i2c print transfer information 选项，并在 bus num id 选项中指定对应的 IIC 总线控制器编号，可输入 0，1，2 和 3，如图 7 所示：

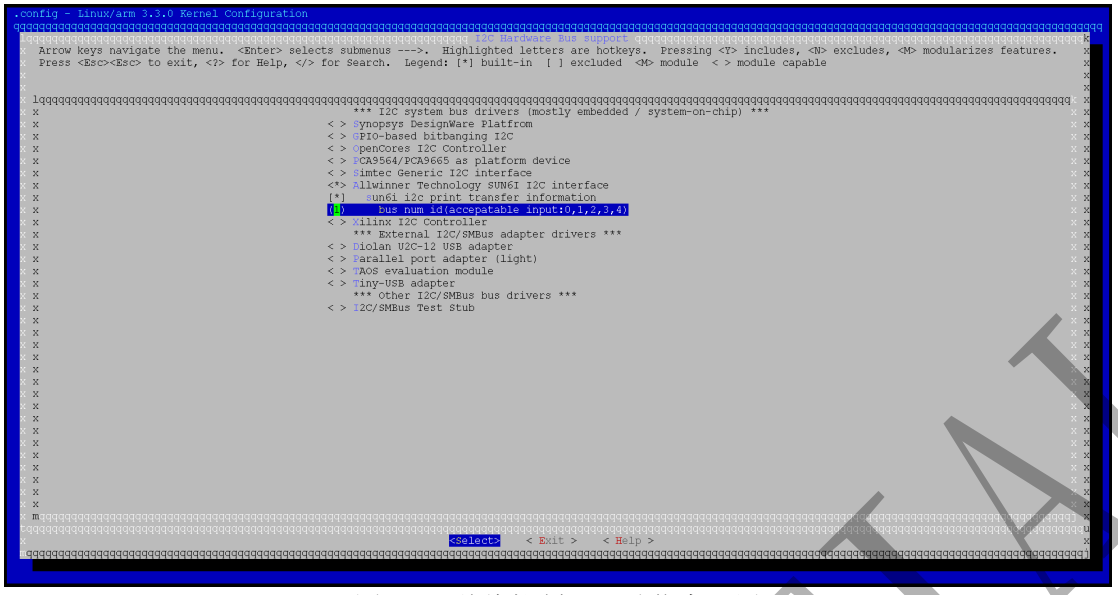

图 7 IIC 总线控制器调试信息配置

3. IIC 体系结构描述

位于 drivers/i2c/busses 目录下的文件 i2c-sun6i.c，是基于 sun6i 平台实现的 IIC 总线控制器驱动。它的职责是为系统中 4 条 IIC 总线实现相应的读写方法，但是控制器驱动本身并不会进行任何的通讯，而是等待设备驱动调用其函数。

图 8 是基于 SUN6I 平台的 IIC 驱动层次架构图，图 8 中有 4 块 IIC adapter，分别对应 SUN6I 平台上的 4 块 IIC 控制器。

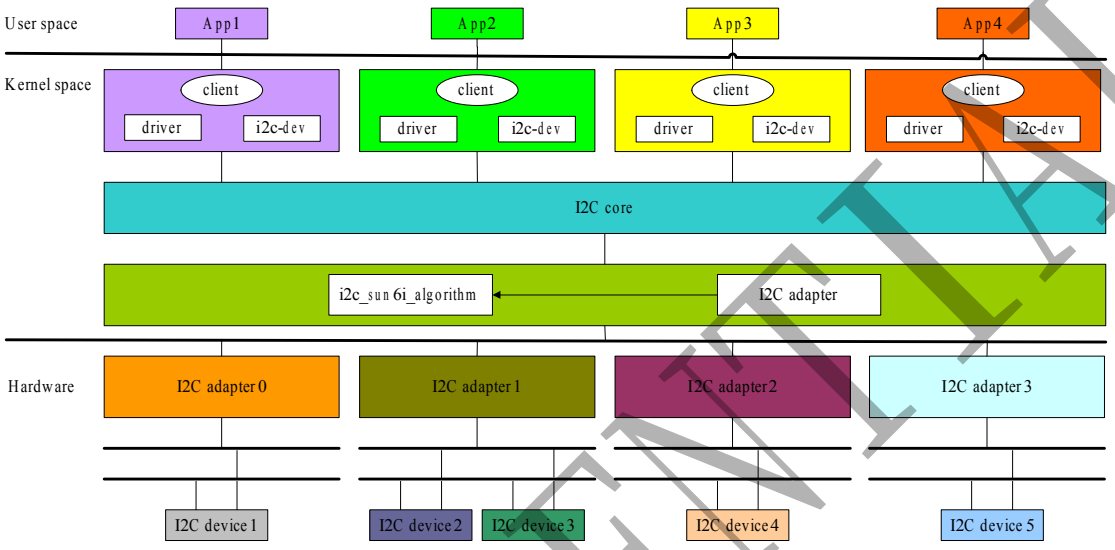

图 8 IIC 驱动层次架构图

系统开机时，IIC 控制器驱动首先被装载，IIC 控制器驱动用于支持 IIC 总线的读写。i2c_sun6i_algorithm 结构体中定义了 IIC 总线通信方法函数 i2c_sun6i_xfer()，该函数实现了对 IIC 总线访问的具体方法，设备驱动通过调用这个函数，实现对 IIC 总线的访问；而在函数 i2c_sun6i_probe()中完成了对 IIC adapter 的初始化。

4. IIC 常用数据结构描述

1）i2c_adapter

```
struct i2c_adapter {
        struct module *owner;                   /*  所属模块  */
        unsigned int id;   /* algorithm 的类型，定义于 i2c-id.h，以 I2C_ALGO_开始  */
        unsigned int class;
        const struct i2c_algorithm *algo;   /*  总线通信方法结构体指针  */
        void *algo_data;                        /* algorithm 数据  */
        struct rt_mutex bus_lock;
        int timeout;                            /*  超时时间，以 jiffies 为单位  */
        int retries;                            /*  重试次数  */
        struct device dev;                      /*  控制器设备  */
        int nr;
        char name[48];                          /*  控制器名称  */
        struct completion dev_released;     /*  用于同步  */
        struct mutex userspace_clients_lock;
        struct list_head userspace_clients;
};
```

i2c_adapter 对应于物理上的一个控制器。一个 IIC 控制器需要 i2c_algorithm 中提供的通信函数来控制控制器上产生特定的访问周期。

2）i2c_algorithm

```
struct i2c_algorithm {
        /* I2C 传输函数指针  */
        int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs, int num);

        /* smbus 传输函数指针  */
        int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr,
                           unsigned short flags, char read_write,
                           u8 command, int size, union i2c_smbus_data *data);

        /*  返回控制器支持的功能  */
        u32 (*functionality) (struct i2c_adapter *);
};
```

i2c_algorithm 中的关键函数 master_xfer()用于产生 IIC 访问周期需要的信号，以 i2c_msg（即 IIC 消息）为单位。

3）i2c_msg

```
struct i2c_msg {
        __u16 addr;             /*  从设备地址  */
        __u16 flags;            /*  消息类型  */
        __u16 len;              /*  消息长度  */
        __u8 *buf;              /*  消息数据  */
};
```

i2c_msg 是 IIC 传输的基本单位，它包含了从设备的具体地址，消息的类型以及要传输的具体数据信息。每个 IIC 消息传输前，都会产生一个开始位，紧接着传送从设备地址，然后开始数据的发送或接收，对最后的消息还需产生一个停止位。

4）i2c_client

```
struct i2c_client {
        unsigned short flags;                   /*  标志  */
        unsigned short addr;                    /*  低 7 位的芯片地址  */
        char name[I2C_NAME_SIZE];               /*  设备名称  */
        struct i2c_adapter *adapter;            /*  依附的 i2c_adapter */
        struct i2c_driver *driver;              /*  依附的 i2c_driver */
        struct device dev;
        int irq;                                /*  设备使用的中断号  */
        struct list_head detected;
};
```

i2c_client 对应于真实的物理设备，每个 IIC 设备都需要一个 i2c_client 来描述。

5）i2c_driver

```
struct i2c_driver {
        unsigned int class;
        int (*attach_adapter)(struct i2c_adapter *);        /*  依附 i2c_adapter 函数指针  */
        int (*detach_adapter)(struct i2c_adapter *);        /*  脱离 i2c_adapter 函数指针  */
        int (*probe)(struct i2c_client *, const struct i2c_device_id *);
        int (*remove)(struct i2c_client *);
        void (*shutdown)(struct i2c_client *);
        int (*suspend)(struct i2c_client *, pm_message_t mesg);
        int (*resume)(struct i2c_client *);
        void (*alert)(struct i2c_client *, unsigned int data);
        int (*command)(struct i2c_client *client, unsigned int cmd, void *arg);
        struct device_driver driver;
        const struct i2c_device_id *id_table;       /*  该驱动所支持的设备 ID 表  */
        int (*detect)(struct i2c_client *, struct i2c_board_info *);        /*  设备探测函数  */
        const unsigned short *address_list;         /*  驱动支持的设备地址  */
        struct list_head clients;                   /*  挂接探测到的支持的设备  */
};
```

i2c_driver 对应一套驱动方法，其主要成员函数是 probe()、remove()、suspend()、resume()等，另外 id_table 是该驱动所支持的 IIC 设备的 ID 表。i2c_driver 与 i2c_client 的关系是一对多，一个 i2c_driver 上可以支持多个同等类型的 i2c_client。

5. IIC 常用接口描述

1）i2c_add_driver

- ***PROTOTYPE***

  static inline int i2c_add_driver(struct i2c_driver *driver);
- ***ARGUMENTS***

  driver the pointer to the i2c device driver;
- ***RETURNS***

  init result;

  = 0 init successful;

  < 0 init failed;
- ***DESCRIPTION***

  Register an i2c device driver to i2c sub-system;

2）i2c_del_driver

- ***PROTOTYPE***

  void i2c_del_driver(struct i2c_driver *driver);
- ***ARGUMENTS***

  driver the pointer to the i2c device driver;
- ***RETURNS***

  None;
- ***DESCRIPTION***

  Unregister an i2c device driver from i2c sub-system;

3）i2c_set_clientdata

- ***PROTOTYPE***

  static inline void i2c_set_clientdata(struct i2c_client *dev, void *data);
- ***ARGUMENTS***

  dev handle to slave device;

  data private data that can be set to i2c client;
- ***RETURNS***

  None;
- ***DESCRIPTION***

  Set private data to i2c client;

4）i2c_get_clientdata

- ***PROTOTYPE***

  static inline void *i2c_get_clientdata(const struct i2c_client *dev);
- ***ARGUMENTS***

  dev handle to slave device;
- ***RETURNS***

  None;
- ***DESCRIPTION***

Get private data from i2c client;

5）i2c_master_send

- ***PROTOTYPE***

  int i2c_master_send(struct i2c_client *client, const char *buf, int count);

- ***ARGUMENTS***

  clinet handle to slave device;

  buf data that will be written to the slave;

  count how many bytes to write, must be less than 64k since msg.len is u16;

- ***RETURNS***

  send result;

  > 0 the number of bytes written;

  < 0 negative errno;

- ***DESCRIPTION***

  Issue a single I2C message in master transmit mode;

6）i2c_master_resv

- ***PROTOTYPE***

  int i2c_master_recv(struct i2c_client *client, char *buf, int count);

- ***ARGUMENTS***

  clinet handle to slave device;

  buf where to store data read from slave;

  count how many bytes to read, must be less than 64k since msg.len is u16;

- ***RETURNS***

  receive result;

  > 0 the number of bytes read;

  < 0 negative errno;

- ***DESCRIPTION***

  Issue a single I2C message in master receive mode;

7）i2c_transfer

- ***PROTOTYPE***

  int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num);

- ***ARGUMENTS***

  adaphandle to I2C bus;

  msgs one or more messages to execute before STOP is issued to terminate the operation; each message begins with a START;

  num Number of messages to be executed;

- ***RETURNS***

  transfer result;

  > 0 number of messages executed;

  < 0 negative errno;

- ***DESCRIPTION***

  Execute a single or combined I2C message; note that there is no requirement that

each message be sent to the same slave address, although that is the most common model.

6. IIC 设备驱动开发 demo

以下代码是一个最简单的 IIC 设备驱动 demo，具体代码如下：

```
#include <linux/i2c.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>

static int i2c_driver_demo_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
	return 0;
}

static int __devexit i2c_driver_demo_remove(struct i2c_client *client)
{
	return 0;
}

static const struct i2c_device_id i2c_driver_demo_id[] = {
	{ "XXXX", 0 },
	{}
};
MODULE_DEVICE_TABLE(i2c, i2c_driver_demo_id);

int i2c_driver_demo_detect(struct i2c_client *client, struct i2c_board_info *info)
{
	struct i2c_adapter *adapter = client->adapter;
	int vendor, device, revision;

	if (!i2c_check_functionality(adapter, I2C_FUNC_SMBUS_BYTE_DATA))
		return -ENODEV;

	/* 方法 1：获取设备特定寄存器的值，该值要能反映出设备的信息，判断设备，例如如下代码段 */
	vendor = i2c_smbus_read_byte_data(client, XXXX_REG_VENDOR);
	if (vendor != XXXX_VENDOR)
		return -ENODEV;

	device = i2c_smbus_read_byte_data(client, XXXX_REG_DEVICE);
	if (device != XXXX_DEVICE)
		return -ENODEV;

	revision = i2c_smbus_read_byte_data(client, XXXX_REG_REVISION);
```

```
        if (revision != XXXX_REVISION)
        return -ENODEV;

        /* 方法 2：获取设备的 CHIP_ID，判断设备，例如如下代码 */
        if (i2c_smbus_read_byte_data(client, XXXX_CHIP_ID_REG) != XXXX_CHIP_ID)
            return -ENODEV;

        const char *type_name = "XXXX";
        strlcpy(info->type, type_name, I2C_NAME_SIZE);
        return 0;
}

/* 0x60 为 I2C 设备地址 */
static const unsigned short normal_i2c[] = {0x60, I2C_CLIENT_END};

static struct i2c_driver i2c_driver_demo = {
        .class = I2C_CLASS_HWMON,
        .probe              = i2c_driver_demo_probe,
        .remove             = __devexit_p(i2c_driver_demo_remove),
        .id_table       = i2c_driver_demo_id,
        .driver       = {
            .name       = "XXXX",
            .owner       = THIS_MODULE,
        },
        .detect             = i2c_driver_demo_detect,
        .address_list       = normal_i2c,
};

static int __init i2c_driver_demo_init(void)
{
        return i2c_add_driver(&i2c_driver_demo);
}

static void __exit i2c_driver_demo_exit(void)
{
        i2c_del_driver(&i2c_driver_demo);
}

module_init(i2c_driver_demo_init);
module_exit(i2c_driver_demo_exit);

MODULE_AUTHOR("anchor");
MODULE_DESCRIPTION("I2C device driver demo");
MODULE_LICENSE("GPL");
```

补充说明：若 IIC 设备驱动不能在 detect 回调函数里访问硬件，可采用如下形式解决，例如：

```
int i2c_driver_demo_detect(struct i2c_client *client, struct i2c_board_info *info)
{
    struct i2c_adapter *adapter = client->adapter;
    if(2 == adapter->nr)
    {
        const char *type_name = "XXXX";
        strlcpy(info->type, type_name, I2C_NAME_SIZE);
        return 0;
    }else
    {
        return -ENODEV;
    }
}
```

在 detect 函数里，需要判断 IIC adapter 与 IIC client 是否进行绑定，若当前 adapter 是该驱动要绑定的 adapter，例如 if 判断中的 2 为从配置信息中读出的 IIC 设备依附的 adapter 编号，则进行相应的 IIC 设备信息注册；否则，不予注册。

7. IIC 常见问题

- 发送 start 失败

当 Master 想要发送数据时会检测 SCK 和 SDA 线的状态，仅当两线均为高时才能正确发出 start 位，否则将失败，发送 start 失败的原因有以下几种类型

1）IIC 总线未上拉

2）IIC 总线上有设备未处于正常工作状态(未上电)，从而将 SCK 或 SDA 线拉低

3）IIC 总线上由于某次传输未正常结束，从而造成的输出某数据线未恢复到高阻状态，从而处于低电平状态

- 无 ACK

无 ACK 经常出现在发送 slave address 时，总线上没有任何一个设备作出地址匹配响应，从而造成 Master 访问总线失败，可能造成无 ACK 的原因如下

1）IIC 总线上无与此 slave address 匹配的设备

2）IIC 总线上有匹配的设备但因其状态未准备好而无 ACK 响应

3）IIC 总线上与之匹配的设备在错误的时钟沿（时钟毛刺造成）已发送 ACK，而 master 在正确的时钟沿未收到 ACK