全志科技
Allwinner Technology

H6 TWI 接口使用说明书

1.0

2017.07.17

文档履历

版本号	日期	制/修订人	内容描述
1.0	2017.07.17	AWA1199	

目录

1. 概述

1.1 编写目的

全志平台 TWI 硬件通信协议兼容 I2C 通信协议，因此可以使用 Linux 内核中的 I2C 子系统，本文档主要介绍 TWI 设备协议如何使用 Linux 内核 I2C 子系统接口，为 TWI 通信设备驱动开发提供参考。

1.2 适用范围

适用全志 H 系列平台，例如 H5、H6。

1.3 相关人员

TWI 设备驱动,TWI 总线驱动的开发/维护人员

2. 模块介绍

2.1 模块功能介绍

Linux 的 I2C 体系结构 2.1 所示, 图中用分割线分成了三个层次

1. 用户空间, 包括所有的 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 I2C 控制器和 I2C 外设

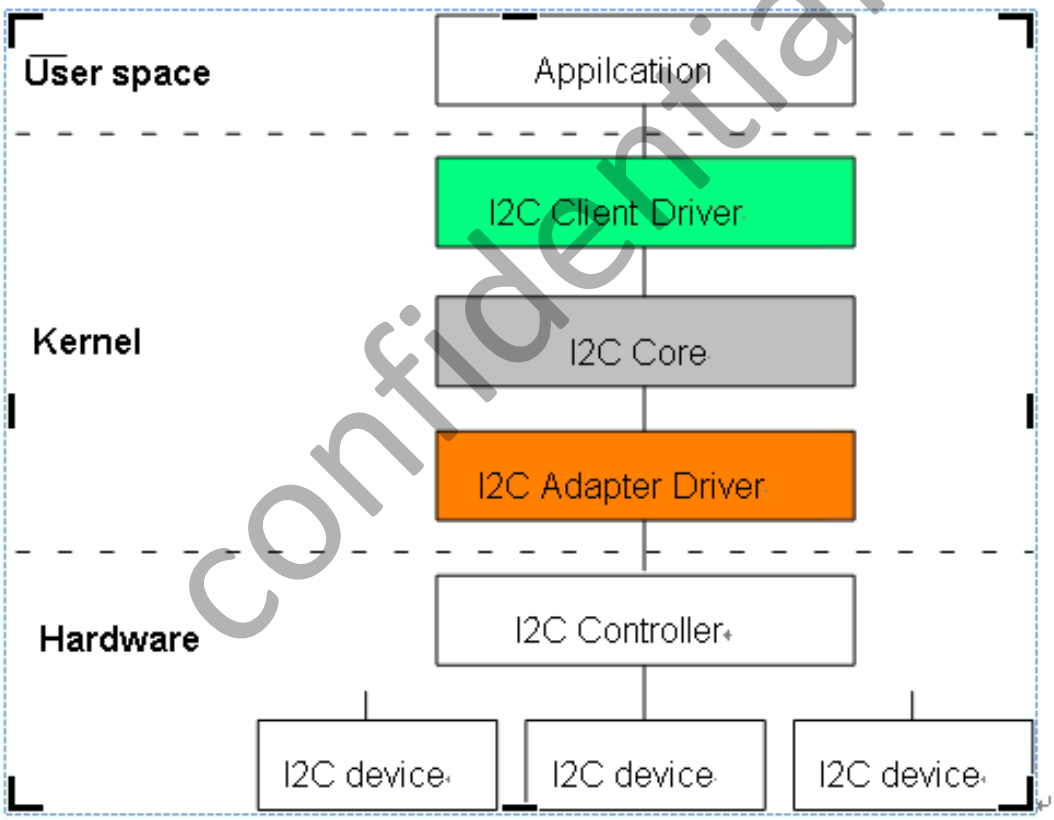

图 1: Linux I2C 体系结构图

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 总线驱动以及 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

I2C 总线驱动主要实现了用于特定 I2C 控制器的总线读写方法, 并注册到 Linux 内核的 I2C 架构, I2C 外设就可以通过 I2C 架构完成设备和总线的适配. 但是总线驱动本身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

I2C Core 的管理正好屏蔽了 I2C 总线驱动的差异, 使得 I2C 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

2.2 相关术语介绍

术语	解释说明
sunxi	指 Allwinner 的一系列 soc 硬件平台
I2C	Inter-Intergrated Circuit 用于 cpu 与外设通信的一种串行总线
TWI	Nornal Two Wire Interface, Sunxi 平台中的 TWI 控制器名称, 兼容 I2C 通信协议
I2C Adapter	I2C Core 将所有 I2C 控制器成为 I2C 适配器, 可以理解成控制器的软件名称
I2C Client	指 I2C 从设备
smbus	System Managerment Bus, 系统管理总线, 基于 I2C 操作原理, 是一个两线接口, 通过它各个设备之间以及设备与系统的其他部分可以互相通信

2.3 模块配置介绍

2.3.1 sys_config.fex 配置说明

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 sys_config.fex 中配置参数相似，如下：

```
[twi0]
twi0_used       = 1
twi0_scl        = port:PH14<2><default><default><default>
twi0_sda        = port:PH15<2><default><default><default>
```

其中，twi0_used 置为 1 表示使能，0 表示不使能；twi0_scl 和 twi0_sda 用于配置相应的 GPIO。

2.3.2 menuconfig 配置说明

在命令行中进入内核根目录，执行 make ARCH=arm64 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：

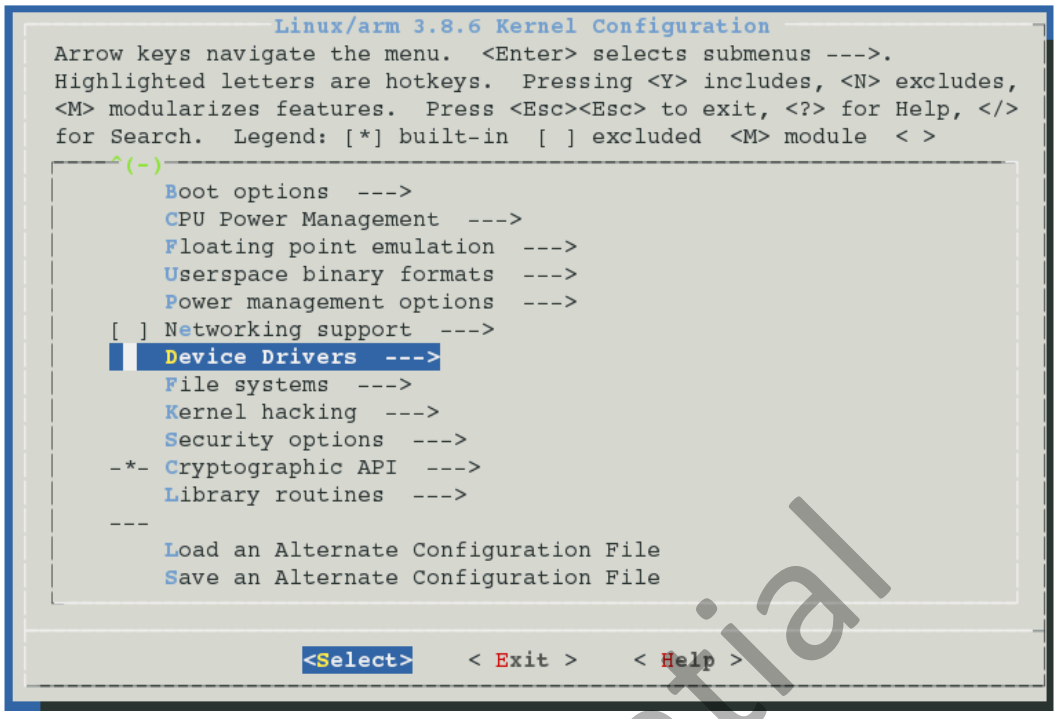

图 2: Device Drivers 选项配置

然后，选择 I2C support 选项，进入下一级配置，如下图所示：

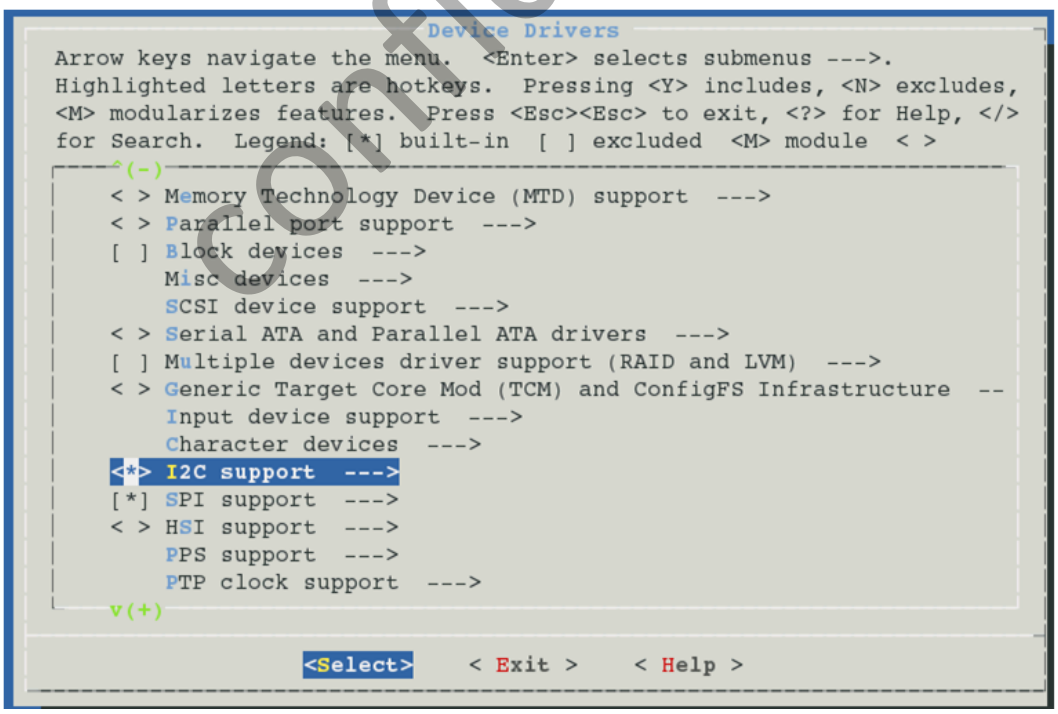

图 3: Device Drivers 选项配置

接着，选择 I2C HardWare Bus support 选项，进入下一级配置，如下图:

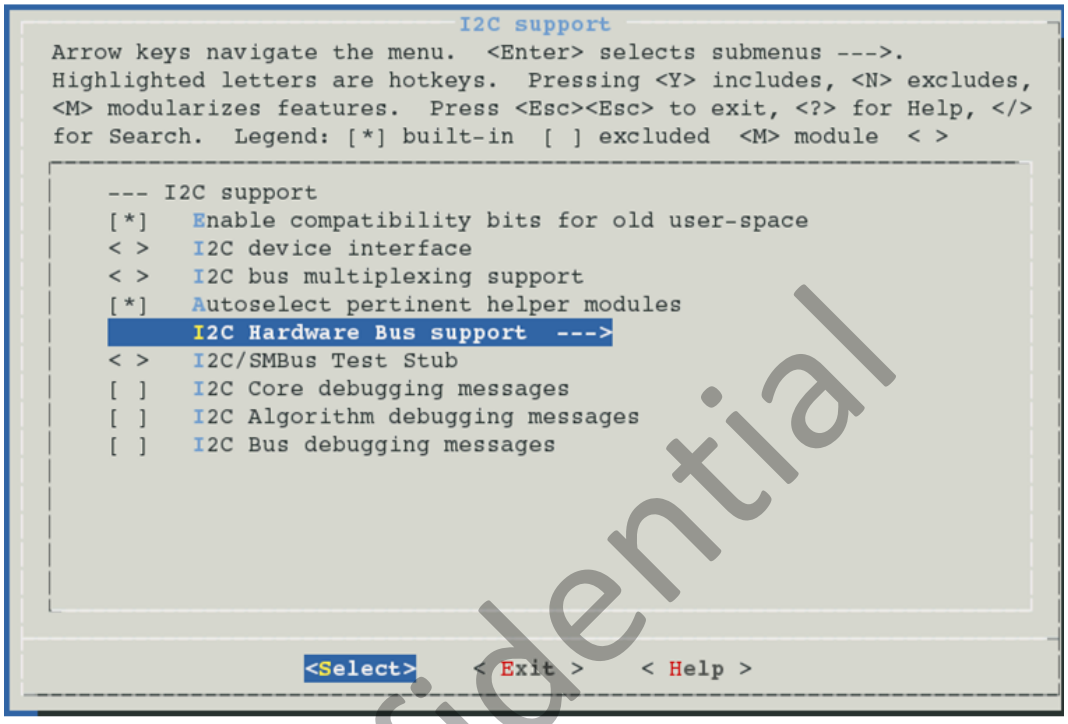

图 4: Device Drivers 选项配置

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:

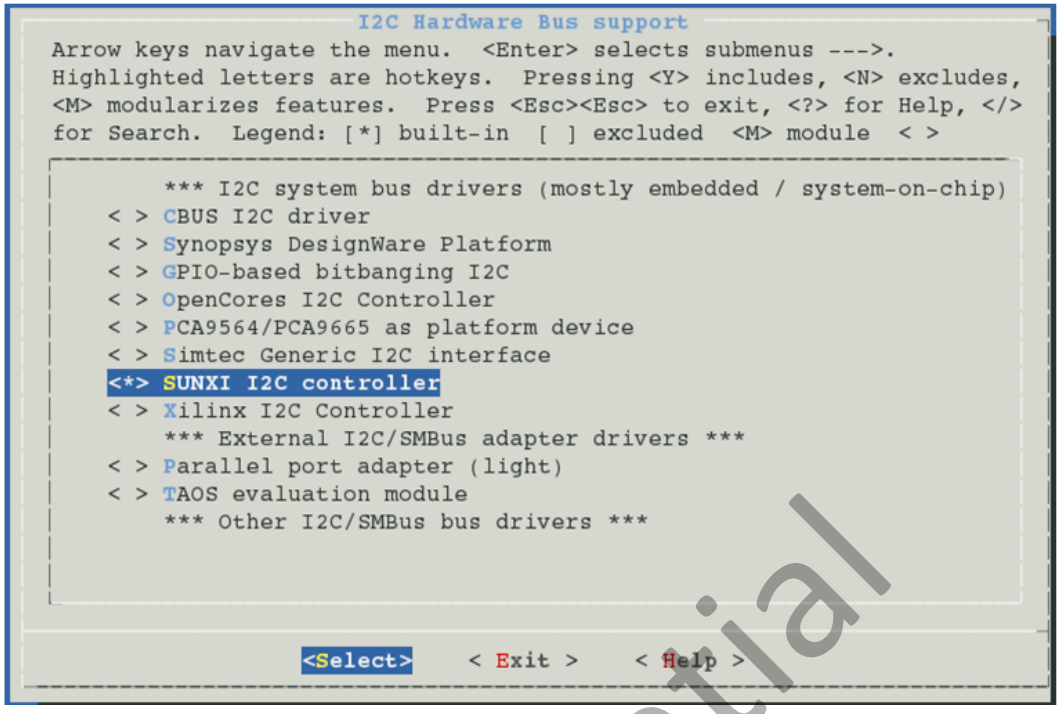

图 5: Device Drivers 选项配置

2.4 源码结构介绍

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 TWI 控制器驱动代码
│     ├──── i2c-sunxi.h // 为 Sunxi 平台的 TWI 控制器驱动定义了一些宏、数据结构
```


3. 接口描述

3.1 设备注册接口

定义在./include/linux/i2c.h

3.1.1 i2c_add_driver()

【函数原型】: #define i2c_add_driver(driver) i2c_register_driver(THIS_MODULE, driver)
int i2c_register_driver(struct module owner, struct i2c_driver driver)
【功能描述】: 注册一个 I2C 设备驱动。从代码可以看带 i2c_add_driver() 是一个宏，由函数 i2c_register_driver() 实现。
【参数说明】: driver，i2c_driver 类型的指针，其中包含了 I2C 设备的名称、probe、detect 等接口信息。
【返回值】: 0，成功；其他值，失败。
其中，结构 i2c_driver 的定义如下:

```
struct i2c_device_id {
    char name[I2C_NAME_SIZE];
    kernel_ulong_t driver_data  /* Data private to the driver */
            __attribute__((aligned(sizeof(kernel_ulong_t))));
};
struct i2c_driver {
    unsigned int class;

    /* Notifies the driver that a new bus has appeared or is about to be
     * removed. You should avoid using this, it will be removed in a
     * near future.
     */
    int (*attach_adapter)(struct i2c_adapter *) __deprecated;
    int (*detach_adapter)(struct i2c_adapter *) __deprecated;
```



```
    /* Standard driver model interfaces */
    int (*probe)(struct i2c_client *, const struct i2c_device_id *);
    int (*remove)(struct i2c_client *);

    /* driver model interfaces that don't relate to enumeration  */
    void (*shutdown)(struct i2c_client *);
    int (*suspend)(struct i2c_client *, pm_message_t mesg);
    int (*resume)(struct i2c_client *);

    /* Alert callback, for example for the SMBus alert protocol.
     * The format and meaning of the data value depends on the protocol.
     * For the SMBus alert protocol, there is a single bit of data passed
     * as the alert response's low bit ("event flag").
     */
    void (*alert)(struct i2c_client *, unsigned int data);

    /* a ioctl like command that can be used to perform specific functions
     * with the device.
     */
    int (*command)(struct i2c_client *client, unsigned int cmd, void *arg);

    struct device_driver driver;
    const struct i2c_device_id *id_table;

    /* Device detection callback for automatic device creation */
    int (*detect)(struct i2c_client *, struct i2c_board_info *);
    const unsigned short *address_list;
    struct list_head clients;
};
```

I2C 设备驱动可能支持多种型号的设备，可以在.id_table 中给出所有支持的设备信息。

3.1.2 i2c_del_driver()

【函数原型】：void i2c_del_driver(struct i2c_driver *driver) 【功能描述】：注销一个 I2C 设备驱动。【参数说明】：driver，i2c_driver 类型的指针，包含有待卸载的 I2C 驱动信息 【返回值】：无 i2c.h 中还给出了快速注册的 I2C 设备驱动的宏：module_i2c_driver()，定义如下：

```
#define module_i2c_driver(__i2c_driver) \
module_driver(__i2c_driver, i2c_add_driver, \
i2c_del_driver)
```

3.1.3 i2c_register_board_info()

【函数原型】：int i2c_register_board_info(int busnum, struct i2c_board_info const *info, unsigned n) 【功能描述】：向某个 I2C 总线注册 I2C 设备信息，I2C 子系统通过此接口保存 I2C 总线和 I2C 设备的适配关系。【参数说明】：busnum，I2C 控制器编号 info，提供 I2C 设备名称、I2C 设备地址信息 n，要注册的 I2C 设备个数 【返回值】：0，成功；其他值，失败注意，注册 I2C 设备信息的方式除了 i2c_register_board_info()，还可以通过 I2C 设备驱动的 detect 接口实现，此接口会在 I2C 子系统注册一个 I2C adapter（即 I2C 控制器）或注册一个 I2C 设备驱动时调用。

3.2 数据传输接口

I2C 设备驱动使用 ``struct i2c_msg" 向 I2C 总线请求读写 I/O。一个 i2c_msg 中包含了一个 I2C 操作，通过调用 i2c_transfer() 接口触发 I2C 总线的数据收发。i2c_transfer() 支持多个 i2c_msg，处理时按串行的顺序依次执行。i2c_msg 的定义也在 i2c.h 中：

```
struct i2c_msg {
    __u16 addr; /* slave address          */
    __u16 flags;
#define I2C_M_TEN       0x0010  /* this is a ten bit chip address */
```



```
#define I2C_M_RD        0x0001  /* read data, from slave to master */
#define I2C_M_NOSTART       0x4000  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_REV_DIR_ADDR 0x2000 /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_IGNORE_NAK    0x1000  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_NO_RD_ACK     0x0800  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_RECV_LEN      0x0400  /* length will be first received byte */
    __u16 len;      /* msg length               */
    __u8 *buf;      /* pointer to msg data          */
};
```

3.2.1 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

3.2.2 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

3.2.3 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

3.2.4 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

3.2.5 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

3.2.6 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

3.2.7 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

3.2.8 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

3.2.9 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

3.2.10 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

3.2.11 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

4. demo

4.1 使用 i2c_register_board_info() 方式注册设备

需要在 I2C 总线驱动和 I2C 设备驱动的初始化之前调用此接口：

```
static struct i2c_board_info eeprom_i2c_board_info[] = {
    {I2C_BOARD_INFO("24c16", 0x50), }
};

void sunxi_i2c_test(void)
{
    int ret = 0;;

  ret = i2c_register_board_info(CONFIG_TWI_CHAN_NUM, eeprom_i2c_board_info, ARRAY_SIZE(eepro
    if (ret < 0) {
        printk("%s()%d - EEPROM init failed!\n", __func__, __LINE__);
    }
    else{
        printk("%s()%d - EEPROM init successed!\n", __func__, __LINE__);
    }
}
```

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 ./drivers/hwmon/bma250.c

此 I2C 设备是一个 Gsensor，使用 detect 方式完成 I2C 设备和 I2C 总线的适配，代码如下，主要目的是完成 info->type 的赋值。

```
static int gsensor_detect(struct i2c_client *client, struct i2c_board_info *info)
{
    struct i2c_adapter *adapter = client->adapter;
    int ret;

    …
    if (twi_id == adapter->nr) {
        for (i2c_num = 0; i2c_num < (sizeof(i2c_address)/sizeof(i2c_address[0]));i2c_num+
+) {
            client->addr = i2c_address[i2c_num];
            pr_info("%s:addr= 0x%x,i2c_num:%d\n",__func__,client->addr,i2c_num);
            ret = i2c_smbus_read_byte_data(client,BMA250_CHIP_ID_REG);
            pr_info("Read ID value is :%d",ret);
            if ((ret &0x00FF) == BMA250_CHIP_ID) {
                pr_info("Bosch Sensortec Device detected!\n" );
                strlcpy(info->type, SENSOR_NAME, I2C_NAME_SIZE);
                return 0;

            } else if((ret &0x00FF) == BMA150_CHIP_ID) {

                pr_info("Bosch Sensortec Device detected!\n" \
                    "BMA150 registered I2C driver!\n");
                strlcpy(info->type, SENSOR_NAME, I2C_NAME_SIZE);
```



```
            return 0;
        } else if((ret &0x00FF) == BMA250E_CHIP_ID) {

            pr_info("Bosch Sensortec Device detected!\n" \
                "BMA250E registered I2C driver!\n");
            strlcpy(info->type, SENSOR_NAME, I2C_NAME_SIZE);
            return 0;
        }
    }

    pr_info("%s:Bosch Sensortec Device not found, \
        maybe the other gsensor equipment! \n",__func__);
    return -ENODEV;
  } else {
    return -ENODEV;
  }
}
```

bma250.c 将 Gsensor 注册成一个 input 设备，定时向上层报告坐标数据，这样应用层就可以采用类似鼠标键盘设备的方式读取到坐标数据。可以看出，input 接口是 Gsensor 这个设备和上层应用的接口，而第 3 章所描述的 I2C 接口是 Gsensor 和 I2C adapter 总线控制器的接口。在参考这个例子上，重点关注 I2C 接口的使用即可。

5. 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.

