

# 第 15 章

## Linux I<sup>2</sup>C 核心、总线与设备驱动

### 本章导读

I<sup>2</sup>C 总线仅仅使用 SCL、SDA 这两根信号线就实现了设备之间的数据交互，极大地简化了对硬件资源和 PCB 板布线空间的占用。因此，I<sup>2</sup>C 总线非常广泛地应用在 EEPROM、实时钟、小型 LCD 等设备与 CPU 的接口中。

Linux 系统定义了 I<sup>2</sup>C 驱动体系结构。在 Linux 系统中，I<sup>2</sup>C 驱动由 3 部分组成，即 I<sup>2</sup>C 核心、I<sup>2</sup>C 总线驱动和 I<sup>2</sup>C 设备驱动。这 3 部分相互协作，形成了非常通用、可适应性很强的 I<sup>2</sup>C 框架。

15.1 节对 Linux 的 I<sup>2</sup>C 体系结构进行分析，讲解 3 个组成部分各自的功能及相互联系。

15.2 节对 Linux 的 I<sup>2</sup>C 核心进行分析，讲解 i2c-core.c 文件的功能和主要函数的实现。

15.3 节、15.4 节分别详细介绍 I<sup>2</sup>C 适配器驱动和 I<sup>2</sup>C 设备驱动的编写方法，给出可供参考的设计模板。

15.5 节、15.6 节以 15.3 节和 15.4 节给出的设计模板为基础，讲解 Tegra ARM 处理器的 I<sup>2</sup>C 总线驱动，以挂接在 I<sup>2</sup>C 总线上的 AT24XX 系列 EEPROM 为例讲解 I<sup>2</sup>C 设备驱动。

### 15.1 Linux I<sup>2</sup>C 体系结构

Linux 的 I<sup>2</sup>C 体系结构分为 3 个组成部分。

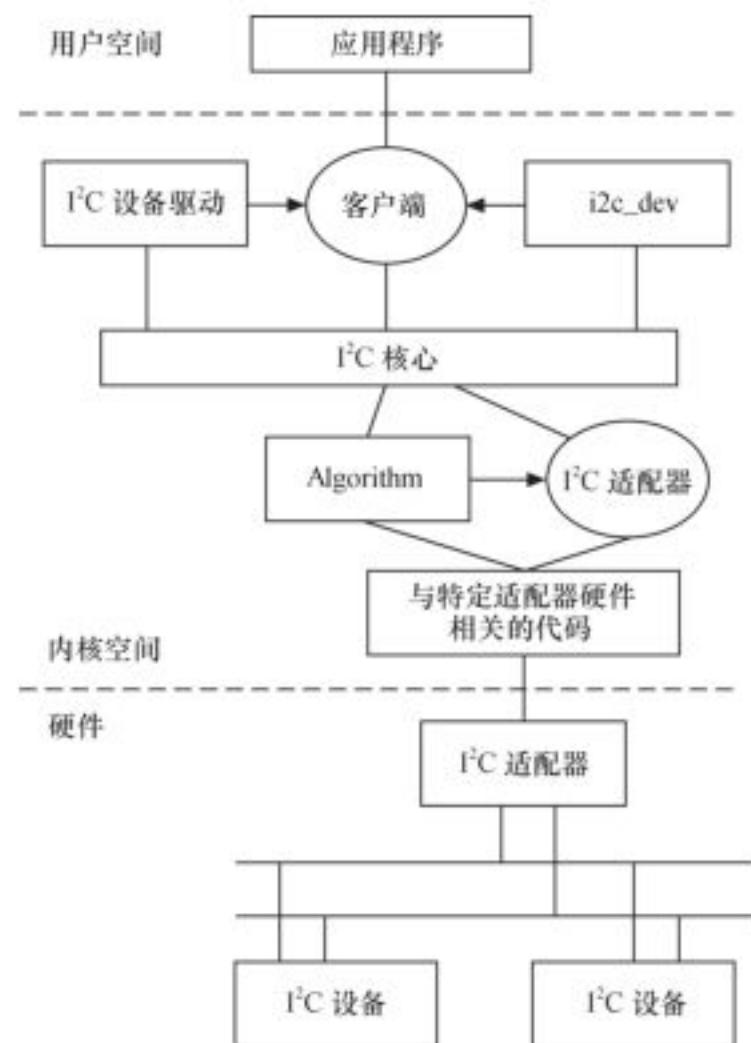
#### (1) I<sup>2</sup>C 核心

I<sup>2</sup>C 核心提供了 I<sup>2</sup>C 总线驱动和设备驱动的注册、注销方法，I<sup>2</sup>C 通信方法（即 Algorithm）上层的与具体适配器无关的代码以及探测设备、检测设备地址的上层代码等，如图 15.1 所示。

#### (2) I<sup>2</sup>C 总线驱动

I<sup>2</sup>C 总线驱动是对 I<sup>2</sup>C 硬件体系结构中适配器端的实现，适配器可由 CPU 控制，甚至可以直接集成在 CPU 内部。

I<sup>2</sup>C 总线驱动主要包含 I<sup>2</sup>C 适配器数据结构 i2c\_adapter、I<sup>2</sup>C 适配器的 Algorithm 数据结构 i2c\_algorithm 和控制 I<sup>2</sup>C 适配器产生通信信号的函数。

图 15.1 Linux 的 I<sup>2</sup>C 体系结构

经由 I<sup>2</sup>C 总线驱动的代码，我们可以控制 I<sup>2</sup>C 适配器以主控方式产生开始位、停止位、读写周期，以及以从设备方式被读写、产生 ACK 等。

### (3) I<sup>2</sup>C 设备驱动

I<sup>2</sup>C 设备驱动（也称为客户驱动）是对 I<sup>2</sup>C 硬件体系结构中设备端的实现，设备一般挂接在受 CPU 控制的 I<sup>2</sup>C 适配器上，通过 I<sup>2</sup>C 适配器与 CPU 交换数据。

I<sup>2</sup>C 设备驱动主要包含数据结构 i2c\_driver 和 i2c\_client，我们需要根据具体设备实现其中的成员函数。

在 Linux 2.6 内核中，所有的 I<sup>2</sup>C 设备都在 sysfs 文件系统中显示，存于 /sys/bus/i2c/ 目录下，以适配器地址和芯片地址的形式列出，例如：

```
$ tree /sys/bus/i2c/
/sys/bus/i2c/
|-- devices
|   |-- i2c0 -> ../../devices/platform/versatile-i2c.0/i2c-0
|   '-- i2c1 -> ../../devices/platform/versatile-i2c.0/i2c-1
`-- drivers
    '-- dummy
```

在Linux内核源代码中的drivers目录下有一个i2c目录，而在i2c目录下又包含如下文件和文件夹。

(1) i2c-core.c

这个文件实现了I<sup>2</sup>C核心的功能以及/proc/bus/i2c\*接口。

(2) i2c-dev.c

实现了I<sup>2</sup>C适配器设备文件的功能，每一个I<sup>2</sup>C适配器都被分配一个设备。通过适配器访问设备时的主设备号都为89，次设备号为0~255。应用程序通过“i2c-%d”(i2c-0, i2c-1,…, i2c-10,… )文件名并使用文件操作接口open()、write()、read()、ioctl()和close()等来访问这个设备。

i2c-dev.c并不是针对特定的设备而设计的，只是提供了通用的read()、write()和ioctl()等接口，应用层可以借用这些接口访问挂接在适配器上的I<sup>2</sup>C设备的存储空间或寄存器，并控制I<sup>2</sup>C设备的工作方式。

(3) busses文件夹

这个文件包含了一些I<sup>2</sup>C主机控制器的驱动，如i2c-tegra.c、i2c-omap.c、i2c-versatile.c、i2c-s3c2410.c等。

(4) algos文件夹

实现了一些I<sup>2</sup>C总线适配器的通信方法。

此外，内核中的i2c.h头文件对i2c\_adapter、i2c\_algorithm、i2c\_driver和i2c\_client这4个数据结构进行了定义。理解这4个结构体的作用十分重要，它们的定义位于include/linux/i2c.h文件中，代码清单15.1、15.2、15.3、15.4分别对它们进行了描述。

代码清单15.1 i2c\_adapter结构体

---

```

1 struct i2c_adapter {
2     struct module *owner;
3     unsigned int class;           /* classes to allow probing for */
4     const struct i2c_algorithm *algo;    /* the algorithm to access the bus */
5     void *algo_data;
6
7     /* data fields that are valid for all devices */
8     struct rt_mutex bus_lock;
9
10    int timeout;                /* in jiffies */
11    int retries;
12    struct device dev;          /* the adapter device */
13
14    int nr;
15    char name[48];
16    struct completion dev_released;
17
18    struct mutex userspace_clients_lock;
19    struct list_head userspace_clients;

```

```

20
21     struct i2c_bus_recovery_info *bus_recovery_info;
22 };

```

代码清单 15.2 i2c\_algorithm 结构体

```

1 struct i2c_algorithm {
2     /* If an adapter algorithm can't do I2C-level access, set master_xfer
3         to NULL. If an adapter algorithm can do SMBus access, set
4         smbus_xfer. If set to NULL, the SMBus protocol is simulated
5         using common I2C messages */
6     /* master_xfer should return the number of messages successfully
7         processed, or a negative value on error */
8     int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs,
9                         int num);
10    int (*smbus_xfer)(struct i2c_adapter *adap, u16 addr,
11                      unsigned short flags, char read_write,
12                      u8 command, int size, union i2c_smbus_data *data);
13
14    /* To determine what the adapter supports */
15    u32 (*functionality)(struct i2c_adapter *);
16 };

```

上述第 8 行代码对应为 I<sup>2</sup>C 传输函数指针，I<sup>2</sup>C 主机驱动的大部分工作也聚集在这里。上述第 10 行代码对应为 SMBus 传输函数指针，SMBus 不需要增加额外引脚，与 I<sup>2</sup>C 总线相比，在访问时序上也有一定的差异。

代码清单 15.3 i2c\_driver 结构体

```

1 struct i2c_driver {
2     unsigned int class;
3
4     /* Notifies the driver that a new bus has appeared. You should avoid
5         * using this, it will be removed in a near future.
6         */
7     int (*attach_adapter)(struct i2c_adapter *) __deprecated;
8
9     /* Standard driver model interfaces */
10    int (*probe)(struct i2c_client *, const struct i2c_device_id *);
11    int (*remove)(struct i2c_client *);
12
13    /* driver model interfaces that don't relate to enumeration */
14    void (*shutdown)(struct i2c_client *);
15    int (*suspend)(struct i2c_client *, pm_message_t mesg);
16    int (*resume)(struct i2c_client *);
17
18    /* Alert callback, for example for the SMBus alert protocol.
19     * The format and meaning of the data value depends on the protocol.

```

```

20      * For the SMBus alert protocol, there is a single bit of data passed
21      * as the alert response's low bit ("event flag").
22      */
23  void (*alert)(struct i2c_client *, unsigned int data);
24
25  /* a ioctl like command that can be used to perform specific functions
26  * with the device.
27  */
28  int (*command)(struct i2c_client *client, unsigned int cmd, void *arg);
29
30  struct device_driver driver;
31  const struct i2c_device_id *id_table;
32
33  /* Device detection callback for automatic device creation */
34  int (*detect)(struct i2c_client *, struct i2c_board_info *);
35  const unsigned short *address_list;
36  struct list_head clients;
37 };
```

---

代码清单 15.4 i2c\_client 结构体

```

1 struct i2c_client {
2     unsigned short flags;           /* div., see below          */
3     unsigned short addr;           /* chip address - NOTE: 7bit */
4                           /* addresses are stored in the */
5                           /* _LOWER_ 7 bits            */
6     char name[I2C_NAME_SIZE];
7     struct i2c_adapter *adapter;   /* the adapter we sit on    */
8     struct device dev;            /* the device structure      */
9     int irq;                     /* irq issued by device      */
10    struct list_head detected;
11};
```

---

下面分析 i2c\_adapter、i2c\_algorithm、i2c\_driver 和 i2c\_client 这 4 个数据结构的作用及其盘根错节的关系。

#### (1) i2c\_adapter 与 i2c\_algorithm

i2c\_adapter 对应于物理上的一个适配器，而 i2c\_algorithm 对应一套通信方法。一个 I<sup>2</sup>C 适配器需要 i2c\_algorithm 提供的通信函数来控制适配器产生特定的访问周期。缺少 i2c\_algorithm 的 i2c\_adapter 什么也做不了，因此 i2c\_adapter 中包含所使用的 i2c\_algorithm 的指针。

i2c\_algorithm 中的关键函数 master\_xfer() 用于产生 I<sup>2</sup>C 访问周期需要的信号，以 i2c\_msg (即 I<sup>2</sup>C 消息) 为单位。i2c\_msg 结构体也是非常重要的，它定义于 include/uapi/linux/i2c.h (在 uapi 目录下，证明用户空间的应用也可能使用这个结构体) 中，代码清单 15.5 给出了它的定义，其中的成员表明了 I<sup>2</sup>C 的传输地址、方向、缓冲区、缓冲区长度等信息。

代码清单 15.5 i2c\_msg 结构体

---

```

1 struct i2c_msg {
2     __u16 addr;                      /* slave address */
3     __u16 flags;
4 #define I2C_M_TEN      0x0010 /* this is a ten bit chip address */
5 #define I2C_M_RD       0x0001 /* read data, from slave to master */
6 #define I2C_M_STOP     0x8000 /* if I2C_FUNC_PROTOCOL_MANGLING */
7 #define I2C_M_NOSTART   0x4000 /* if I2C_FUNC_NOSTART */
8 #define I2C_M_REV_DIR_ADDR 0x2000 /* if I2C_FUNC_PROTOCOL_MANGLING */
9 #define I2C_M_IGNORE_NAK 0x1000 /* if I2C_FUNC_PROTOCOL_MANGLING */
10 #define I2C_M_NO_RD_ACK 0x0800 /* if I2C_FUNC_PROTOCOL_MANGLING */
11 #define I2C_M_RECV_LEN  0x0400 /* length will be first received byte */
12     __u16 len;                      /* msg length */
13     __u8 *buf;                     /* pointer to msg data */
14 };

```

---

### (2) i2c\_driver 与 i2c\_client

i2c\_driver 对应于一套驱动方法，其主要成员函数是 probe()、remove()、suspend()、resume() 等，另外，struct i2c\_device\_id 形式的 id\_table 是该驱动所支持的 I<sup>2</sup>C 设备的 ID 表。i2c\_client 对应于真实的物理设备，每个 I<sup>2</sup>C 设备都需要一个 i2c\_client 来描述。i2c\_driver 与 i2c\_client 的关系是一对多，一个 i2c\_driver 可以支持多个同类型的 i2c\_client。

i2c\_client 的信息通常在 BSP 的板文件中通过 i2c\_board\_info 填充，如下面的代码就定义了一个 I<sup>2</sup>C 设备的 ID 为“ad7142\_joystick”、地址为 0x2C、中断号为 IRQ\_PF5 的 i2c\_client：

```

static struct i2c_board_info __initdata xxx_i2c_board_info[] = {
#if defined(CONFIG_JOYSTICK_AD7142) || defined(CONFIG_JOYSTICK_AD7142_MODULE)
{
    I2C_BOARD_INFO("ad7142_joystick", 0x2C),
    .irq = IRQ_PF5,
},
...
}

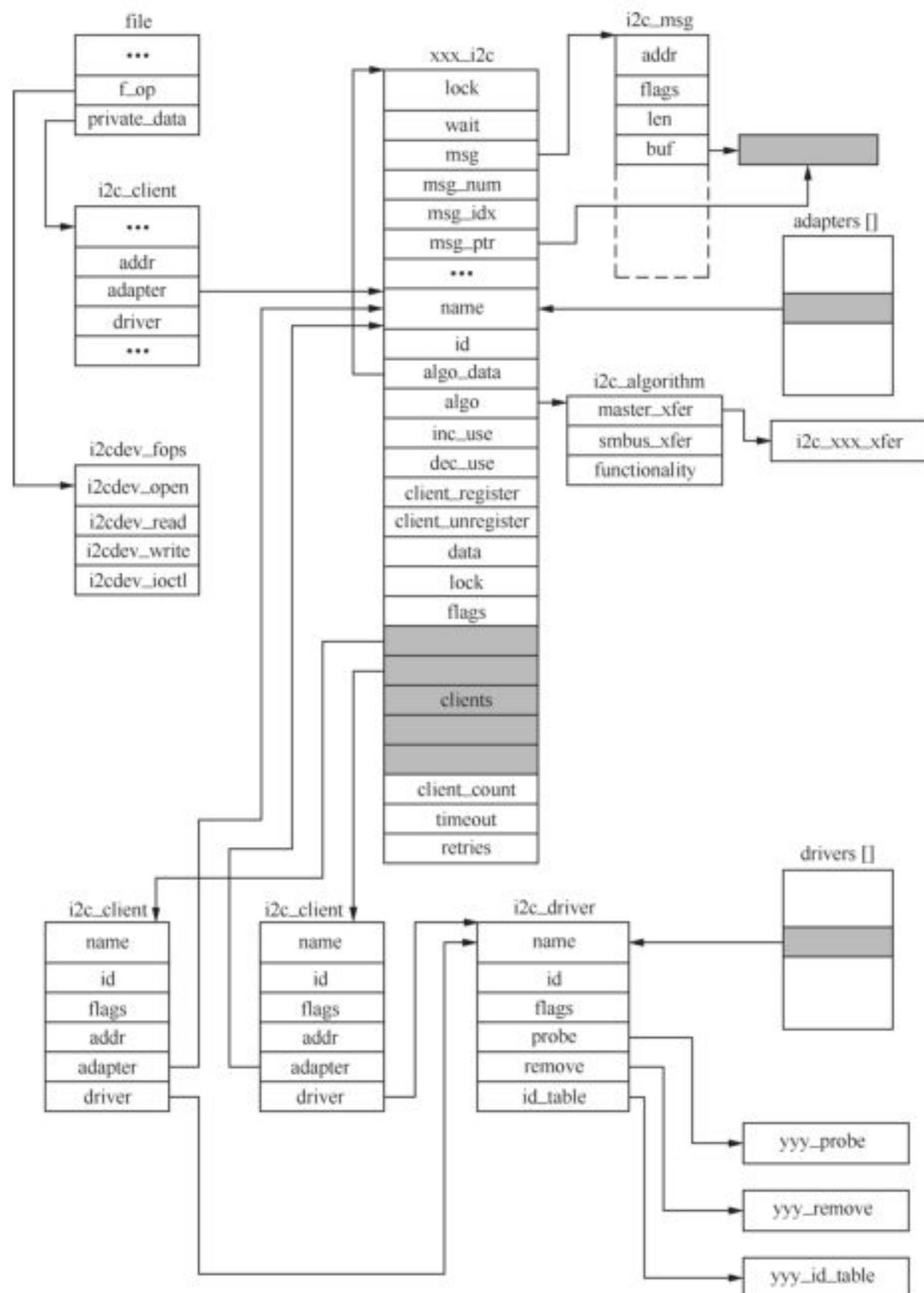
```

在 I<sup>2</sup>C 总线驱动 i2c\_bus\_type 的 match() 函数 i2c\_device\_match() 中，会调用 i2c\_match\_id() 函数匹配在板文件中定义的 ID 和 i2c\_driver 所支持的 ID 表。

### (3) i2c\_adapter 与 i2c\_client

i2c\_adapter 与 i2c\_client 的关系与 I<sup>2</sup>C 硬件体系中适配器和设备的关系一致，即 i2c\_client 依附于 i2c\_adapter。由于一个适配器可以连接多个 I<sup>2</sup>C 设备，所以一个 i2c\_adapter 也可以被多个 i2c\_client 依附，i2c\_adapter 中包括依附于它的 i2c\_client 的链表。

假设 I<sup>2</sup>C 总线适配器 xxx 上有两个使用相同驱动程序的 yyy I<sup>2</sup>C 设备，在打开该 I<sup>2</sup>C 总线的设备节点后，相关数据结构之间的逻辑组织关系将如图 15.2 所示。

图 15.2 I<sup>2</sup>C 驱动的各种数据结构的关系

从上面的分析可知，虽然 I<sup>2</sup>C 硬件体系结构比较简单，但是 I<sup>2</sup>C 体系结构在 Linux 中的实现却相当复杂。当工程师拿到实际的电路板时，面对复杂的 Linux I<sup>2</sup>C 子系统，应该如何

下手写驱动呢？究竟有哪些是需要亲自做的，哪些是内核已经提供的呢？理清这个问题非常有意义，可以使我们在面对具体问题时迅速抓住重点。

一方面，适配器驱动可能是Linux内核本身还不包含的；另一方面，挂接在适配器上的具体设备驱动可能也是Linux内核还不包含的。因此，工程师要实现的主要工作如下。

- 提供I<sup>2</sup>C适配器的硬件驱动，探测、初始化I<sup>2</sup>C适配器（如申请I<sup>2</sup>C的I/O地址和中断号）、驱动CPU控制的I<sup>2</sup>C适配器从硬件上产生各种信号以及处理I<sup>2</sup>C中断等）。
- 提供I<sup>2</sup>C适配器的Algorithm，用具体适配器的xxx\_xfer()函数填充i2c\_algorithm的master\_xfer指针，并把i2c\_algorithm指针赋值给i2c\_adapter的algo指针。
- 实现I<sup>2</sup>C设备驱动中的i2c\_driver接口，用具体设备yyy的yyy\_probe()、yyy\_remove()、yyy\_suspend()、yyy\_resume()函数指针和i2c\_device\_id设备ID表赋值给i2c\_driver的probe、remove、suspend、resume和id\_table指针。
- 实现I<sup>2</sup>C设备所对应类型的具体驱动，i2c\_driver只是实现设备与总线的挂接，而挂接在总线上的设备则千差万别。例如，如果是字符设备，就实现文件操作接口，即实现具体设备yyy的yyy\_read()、yyy\_write()和yyy\_ioctl()函数等；如果是声卡，就实现ALSA驱动。

上述工作中前两个属于I<sup>2</sup>C总线驱动，后两个属于I<sup>2</sup>C设备驱动。15.3~15.4节将详细分析这些工作的实施方法，给出设计模板，而15.5~15.6节将给出两个具体的实例。

## 15.2 Linux I<sup>2</sup>C核心

I<sup>2</sup>C核心（drivers/i2c/i2c-core.c）中提供了一组不依赖于硬件平台的接口函数，这个文件一般不需要被工程师修改，但是理解其中的主要函数非常关键，因为I<sup>2</sup>C总线驱动和设备驱动之间以I<sup>2</sup>C核心作为纽带。I<sup>2</sup>C核心中的主要函数如下。

### (1) 增加 / 删除 i2c\_adapter

```
int i2c_add_adapter(struct i2c_adapter *adap);
void i2c_del_adapter(struct i2c_adapter *adap);
```

### (2) 增加 / 删除 i2c\_driver

```
int i2c_register_driver(struct module *owner, struct i2c_driver *driver);
void i2c_del_driver(struct i2c_driver *driver);
#define i2c_add_driver(driver) \
    i2c_register_driver(THIS_MODULE, driver)
```

### (3) I<sup>2</sup>C传输、发送和接收

```
int i2c_transfer(struct i2c_adapter * adap, struct i2c_msg *msgs, int num);
int i2c_master_send(struct i2c_client *client, const char *buf, int count);
int i2c_master_recv(struct i2c_client *client, char *buf, int count);
```

i2c\_transfer() 函数用于进行 I<sup>2</sup>C 适配器和 I<sup>2</sup>C 设备之间的一组消息交互，其中第 2 个参数是一个指向 i2c\_msg 数组的指针，所以 i2c\_transfer() 一次可以传输多个 i2c\_msg（考虑到很多外设的读写波形比较复杂，比如读寄存器可能要先写，所以需要两个以上的消息）。而对于时序比较简单的外设，i2c\_master\_send() 函数和 i2c\_master\_recv() 函数内部会调用 i2c\_transfer() 函数分别完成一条写消息和一条读消息，如代码清单 15.6、15.7 所示。

代码清单 15.6 I<sup>2</sup>C 核心的 i2c\_master\_send() 函数

---

```

1 int i2c_master_send(const struct i2c_client *client, const char *buf, int count)
2 {
3     int ret;
4     struct i2c_adapter *adap = client->adapter;
5     struct i2c_msg msg;
6
7     msg.addr = client->addr;
8     msg.flags = client->flags & I2C_M_TEN;
9     msg.len = count;
10    msg.buf = (char *)buf;
11
12    ret = i2c_transfer(adap, &msg, 1);
13
14    /*
15     * If everything went ok (i.e. 1 msg transmitted), return #bytes
16     * transmitted, else error code.
17     */
18    return (ret == 1) ? count : ret;
19 }

```

---

代码清单 15.7 I<sup>2</sup>C 核心的 i2c\_master\_recv() 函数

---

```

1 int i2c_master_recv(const struct i2c_client *client, char *buf, int count)
2 {
3     struct i2c_adapter *adap = client->adapter;
4     struct i2c_msg msg;
5     int ret;
6
7     msg.addr = client->addr;
8     msg.flags = client->flags & I2C_M_TEN;
9     msg.flags |= I2C_M_RD;
10    msg.len = count;
11    msg.buf = buf;
12
13    ret = i2c_transfer(adap, &msg, 1);
14
15    /*
16     * If everything went ok (i.e. 1 msg received), return #bytes received,
17     * else error code.
18     */

```

---

---

```

19         return (ret == 1) ? count : ret;
20 }

```

---

i2c\_transfer() 函数本身不具备驱动适配器物理硬件以完成消息交互的能力，它只是寻找与 i2c\_adapter 对应的 i2c\_algorithm，并使用 i2c\_algorithm 的 master\_xfer() 函数真正驱动硬件流程，如代码清单 15.8 所示。

代码清单 15.8 I<sup>2</sup>C 核心的 i2c\_transfer() 函数

---

```

1 int i2c_transfer(struct i2c_adapter * adap, struct i2c_msg *msgs, int num)
2 {
3     int ret;
4
5     if (adap->algo->master_xfer) {
6         ...
7         ret = adap->algo->master_xfer(adap, msgs, num);
8         ...
9         return ret;
10    } else {
11        dev_dbg(&adap->dev, "I2C level transfers not supported\n");
12        return -ENOSYS;
13    }
14 }

```

---

## 15.3 Linux I<sup>2</sup>C 适配器驱动

### 15.3.1 I<sup>2</sup>C 适配器驱动的注册与注销

由于 I<sup>2</sup>C 总线控制器通常是在内存上的，所以它本身也连接在 platform 总线上，要通过 platform\_driver 和 platform\_device 的匹配来执行。因此尽管 I<sup>2</sup>C 适配器给别人提供了总线，它自己也被认为是接在 platform 总线上的一个客户。Linux 的总线、设备和驱动模型实际上是一个树形结构，每个节点虽然可能成为别人的总线控制器，但是自己也被认为是从上一级总线枚举出来的。

通常我们会在与 I<sup>2</sup>C 适配器所对应的 platform\_driver 的 probe() 函数中完成两个工作。

- 初始化 I<sup>2</sup>C 适配器所使用的硬件资源，如申请 I/O 地址、中断号、时钟等。
- 通过 i2c\_add\_adapter() 添加 i2c\_adapter 的数据结构，当然这个 i2c\_adapter 数据结构的成员已经被 xxx 适配器的相应函数指针所初始化。

通常我们会在 platform\_driver 的 remove() 函数中完成与加载函数相反的工作。

- 释放 I<sup>2</sup>C 适配器所使用的硬件资源，如释放 I/O 地址、中断号、时钟等。
- 通过 i2c\_del\_adapter() 删除 i2c\_adapter 的数据结构。

代码清单 15.9 所示为 I<sup>2</sup>C 适配器驱动的注册和注销模板。

代码清单 15.9 I<sup>2</sup>C 适配器驱动的注册和注销模板

```

1 static int xxx_i2c_probe(struct platform_device *pdev)
2 {
3     struct i2c_adapter *adap;
4
5     ...
6     xxx_adpater_hw_init()
7     adap->dev.parent = &pdev->dev;
8     adap->dev.of_node = pdev->dev.of_node;
9
10    rc = i2c_add_adapter(adap);
11    ...
12 }
13
14 static int xxx_i2c_remove(struct platform_device *pdev)
15 {
16     ...
17     xxx_adpater_hw_free()
18     i2c_del_adapter(&dev->adapter);
19
20     return 0;
21 }
22
23 static const struct of_device_id xxx_i2c_of_match[] = {
24     {.compatible = "vendor,xxx-i2c", },
25     {},
26 };
27 MODULE_DEVICE_TABLE(of, xxx_i2c_of_match);
28
29 static struct platform_driver xxx_i2c_driver = {
30     .driver = {
31         .name = "xxx-i2c",
32         .owner = THIS_MODULE,
33         .of_match_table = xxx_i2c_of_match,
34     },
35     .probe = xxx_i2c_probe,
36     .remove = xxx_i2c_remove,
37 };
38 module_platform_driver(xxx_i2c_driver);

```

上述代码中的 xxx\_adpater\_hw\_init() 和 xxx\_adpater\_hw\_free() 函数的实现都与具体的 CPU 和 I<sup>2</sup>C 适配器硬件直接相关。

### 15.3.2 I<sup>2</sup>C 总线的通信方法

我们需要为特定的 I<sup>2</sup>C 适配器实现通信方法，主要是实现 i2c\_algorithm 的 functionality() 函数和 master\_xfer() 函数。

functionality() 函数非常简单，用于返回 algorithm 所支持的通信协议，如 I2C\_FUNC\_

I<sup>2</sup>C、I<sup>2</sup>C\_FUNC\_10BIT\_ADDR、I<sup>2</sup>C\_FUNC\_SMBUS\_READ\_BYTE、I<sup>2</sup>C\_FUNC\_SMBUS\_WRITE\_BYTE 等。

master\_xfer() 函数在 I<sup>2</sup>C 适配器上完成传递给它的 i2c\_msg 数组中的每个 I<sup>2</sup>C 消息，代码清单 15.10 所示为 xxx 设备的 master\_xfer() 函数模板。

代码清单 15.10 master\_xfer() 函数模板

```

1 static int i2c_adapter_xxx_xfer(struct i2c_adapter *adap, struct i2c_msg *msgs,
2     int num)
3 {
4     ...
5     for (i = 0; i < num; i++) {
6         i2c_adapter_xxx_start();                                /* 产生开始位 */
7         /* 是读消息 */
8         if (msgs[i]->flags & I2C_M_RD) {
9             i2c_adapter_xxx_setaddr((msg->addr << 1) | 1);    /* 发送从设备读地址 */
10            i2c_adapter_xxx_wait_ack();                          /* 获得从设备的 ack */
11            i2c_adapter_xxx_readbytes(msgs[i]->buf, msgs[i]->len); /* 读取 msgs[i]->len
12                长的数据到 msgs[i]->buf */
13        } else {                                              /* 是写消息 */
14            i2c_adapter_xxx_setaddr(msg->addr << 1);        /* 发送从设备写地址 */
15            i2c_adapter_xxx_wait_ack();                          /* 获得从设备的 ack */
16            i2c_adapter_xxx_writebytes(msgs[i]->buf, msgs[i]->len); /* 读取 msgs[i]->len
17                长的数据到 msgs[i]->buf */
18        }
19    }
20    i2c_adapter_xxx_stop();                                /* 产生停止位 */
21 }
```

上述代码实际上给出了一个 master\_xfer() 函数处理 I<sup>2</sup>C 消息数组的流程，对于数组中的每个消息，先判断消息类型，若为读消息，则赋从设备地址为 ( msg->addr << 1 ) | 1，否则为 msg->addr << 1。对每个消息产生一个开始位，紧接着传送从设备地址，然后开始数据的发送或接收，且对最后的消息还需产生一个停止位。图 15.3 所示为整个 master\_xfer() 完成的时序。



图 15.3 master\_xfer() 完成的时序

master\_xfer() 函数模板中的 i2c\_adapter\_xxx\_start()、i2c\_adapter\_xxx\_setaddr()、i2c\_adapter\_xxx\_wait\_ack()、i2c\_adapter\_xxx\_readbytes()、i2c\_adapter\_xxx\_writebytes() 和 i2c\_adapter\_xxx\_stop() 函数用于完成适配器的底层硬件操作，与 I<sup>2</sup>C 适配器和 CPU 的具体硬件直接相关，需要由工程师根据芯片的数据手册来实现。

i2c\_adapter\_xxx\_readbytes() 用于从设备上接收一串数据，i2c\_adapter\_xxx\_writebytes() 用于向设备写入一串数据，这两个函数的内部也会涉及 I<sup>2</sup>C 总线协议中的 ACK 应答。

master\_xfer() 函数的实现形式会很多种，多数驱动以中断方式来完成这个流程，比如发起硬件操作请求后，将自己调度出去，因此中间会伴随着睡眠的动作。

多数 I<sup>2</sup>C 总线驱动会定义一个 xxx\_i2c 结构体，作为 i2c\_adapter 的 algo\_data（类似“私有数据”），其中包含 I<sup>2</sup>C 消息数组指针、数组索引及 I<sup>2</sup>C 适配器 Algorithm 访问控制用的自旋锁、等待队列等，而 master\_xfer() 函数在完成 i2c\_msg 数组中消息的处理时，也经常需要访问 xxx\_i2c 结构体的成员以获取寄存器地址、锁等信息。代码清单 15.11 所示为一个典型的 xxx\_i2c 结构体的定义，与图 15.2 中的 xxx\_i2c 是对应的，具体的实现因硬件而异。

代码清单 15.11 xxx\_i2c 结构体模板

---

```

1 struct xxx_i2c {
2     spinlock_t          lock;
3     wait_queue_head_t   wait;
4     struct i2c_msg      *msg;
5     unsigned int         msg_num;
6     unsigned int         msg_idx;
7     unsigned int         msg_ptr;
8     ...
9     struct i2c_adapter  adap;
10 };

```

---

## 15.4 Linux I<sup>2</sup>C 设备驱动

I<sup>2</sup>C 设备驱动要使用 i2c\_driver 和 i2c\_client 数据结构并填充 i2c\_driver 中的成员函数。i2c\_client 一般被包含在设备的私有信息结构体 yyy\_data 中，而 i2c\_driver 则适合被定义为全局变量并初始化，代码清单 15.12 所示为已被初始化的 i2c\_driver。

代码清单 15.12 被初始化的 i2c\_driver

---

```

1 static struct i2c_driver yyy_driver = {
2     .driver = {
3         .name = "yyy",
4     },
5     .probe          = yyy_probe,
6     .remove         = yyy_remove,
7     .id_table      = yyy_id,
8 };

```

---

### 15.4.1 Linux I<sup>2</sup>C 设备驱动的模块加载与卸载

I<sup>2</sup>C 设备驱动的模块加载函数通过 I<sup>2</sup>C 核心的 i2c\_add\_driver() API 函数添加 i2c\_driver 的工作，而模块卸载函数需要做相反的工作：通过 I<sup>2</sup>C 核心的 i2c\_del\_driver() 函数删除 i2c\_driver。代码清单 15.13 所示为 I<sup>2</sup>C 设备驱动的模块加载与卸载函数模板。

代码清单 15.13 I<sup>2</sup>C 外设驱动的模块加载与卸载函数模板

---

```

1 static int __init yyy_init(void)
2 {
3     return i2c_add_driver(&yyy_driver);
4 }
5 module_initcall(yyy_init);
6
7 static void __exit yyy_exit(void)
8 {
9     i2c_del_driver(&yyy_driver);
10}
11 module_exit(yyy_exit);

```

---

### 15.4.2 Linux I<sup>2</sup>C 设备驱动的数据传输

在 I<sup>2</sup>C 设备上读写数据的时序且数据通常通过 i2c\_msg 数组进行组织，最后通过 i2c\_transfer() 函数完成，代码清单 15.14 所示为一个读取指定偏移 offs 的寄存器。

代码清单 15.14 I<sup>2</sup>C 设备驱动的数据传输范例

---

```

1 struct i2c_msg msg[2];
2 /* 第一条消息是写消息 */
3 msg[0].addr = client->addr;
4 msg[0].flags = 0;
5 msg[0].len = 1;
6 msg[0].buf = &offs;
7 /* 第二条消息是读消息 */
8 msg[1].addr = client->addr;
9 msg[1].flags = I2C_M_RD;
10 msg[1].len = sizeof(buf);
11 msg[1].buf = &buf[0];
12
13 i2c_transfer(client->adapter, msg, 2);

```

---

### 15.4.3 Linux 的 i2c-dev.c 文件分析

i2c-dev.c 文件完全可以被看作是一个 I<sup>2</sup>C 设备驱动，不过，它实现的 i2c\_client 是虚拟、临时的，主要是为了便于从用户空间操作 I<sup>2</sup>C 外设。i2c-dev.c 针对每个 I<sup>2</sup>C 适配器生成一个主设备号为 89 的设备文件，实现了 i2c\_driver 的成员函数以及文件操作接口，因此 i2c-dev.c

的主体是“i2c\_driver 成员函数 + 字符设备驱动”。

i2c-dev.c 提供的 i2cdev\_read()、i2cdev\_write() 函数对应于用户空间要使用的 read() 和 write() 文件操作接口，这两个函数分别调用 I<sup>2</sup>C 核心的 i2c\_master\_recv() 和 i2c\_master\_send() 函数来构造一条 I<sup>2</sup>C 消息并引发适配器 Algorithm 通信函数的调用，以完成消息的传输，它们对应于如图 15.4 所示的时序。



图 15.4 i2cdev\_read() 和 i2cdev\_write() 函数对应的时序

但是，很遗憾，大多数稍微复杂一点的 I<sup>2</sup>C 设备的读写流程并不对应于一条消息，往往需要两条甚至多条消息来进行一次读写周期（即如图 15.5 所示的重复开始位的 RepStart 模式），在这种情况下，在应用层仍然调用 read()、write() 文件 API 来读写 I<sup>2</sup>C 设备，将不能正确地读写。



图 15.5 RepStart 模式

鉴于上述原因，i2c-dev.c 中的 i2cdev\_read() 和 i2cdev\_write() 函数不具备太强的通用性，没有太大的实用价值，只能适用于非 RepStart 模式的情况。对于由两条以上消息组成的读写，在用户空间需要组织 i2c\_msg 消息数组并调用 I2C\_RDWR IOCTL 命令。代码清单 15.15 所示为 i2cdev\_ioctl() 函数的框架。

代码清单 15.15 i2c-dev.c 中的 i2cdev\_ioctl() 函数

---

```

1 static int i2cdev_ioctl(struct inode *inode, struct file *file,
2                         unsigned int cmd, unsigned long arg)
3 {
4     struct i2c_client *client = (struct i2c_client *)file->private_data;
5     ...
6     switch (cmd) {
7         case I2C_SLAVE:
8         case I2C_SLAVE_FORCE:
9             ...
10            /* 设置从设备地址 */
11        case I2C_TENBIT:
12        ...
13        case I2C_PEC:
14            ...
15        case I2C_FUNCS:
16        ...
17        case I2C_RDWR:
18            return i2cdev_ioctl_rdrw(client, arg);
19        case I2C_SMBUS:

```

```

19      ...
20  case I2C_RETRY:
21      ...
22  case I2C_TIMEOUT:
23      ...
24  default:
25      return i2c_control(client, cmd, arg);
26 }
27 return 0;
28 }

```

---

常用的 IOCTL 包括 I2C\_SLAVE (设置从设备地址)、I2C\_RETRY (没有收到设备 ACK 情况下的重试次数，默认为 1)、I2C\_TIMEOUT (超时) 以及 I2C\_RDWR。

代码清单 15.16 和代码清单 15.17 所示为直接通过 read()、write() 接口和 O\_RDWR IOCTL 读写 I<sup>2</sup>C 设备的例子。

代码清单 15.16 直接通过 read()/write() 读写 I<sup>2</sup>C 设备

```

1 #include <stdio.h>
2 #include <linux/types.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5 #include <stdlib.h>
6 #include <sys/types.h>
7 #include <sys/ioctl.h>
8 #include <linux/i2c.h>
9 #include <linux/i2c-dev.h>
10
11 int main(int argc, char **argv)
12 {
13     unsigned int fd;
14     unsigned short mem_addr;
15     unsigned short size;
16     unsigned short idx;
17     #define BUFF_SIZE    32
18     char buf[BUFF_SIZE];
19     char cswap;
20     union
21     {
22         unsigned short addr;
23         char bytes[2];
24     } tmp;
25
26     if (argc < 3) {
27         printf("Use:\n%s /dev/i2c-x mem_addr size\n", argv[0]);
28         return 0;
29     }
30     sscanf(argv[2], "%d", &mem_addr);
31     sscanf(argv[3], "%d", &size);

```

```

32
33     if (size > BUFF_SIZE)
34         size = BUFF_SIZE;
35
36     fd = open(argv[1], O_RDWR);
37
38     if (!fd) {
39         printf("Error on opening the device file\n");
40         return 0;
41     }
42
43     ioctl(fd, I2C_SLAVE, 0x50);          /* 设置 EEPROM 地址 */
44     ioctl(fd, I2C_TIMEOUT, 1);           /* 设置超时 */
45     ioctl(fd, I2C_RETRIES, 1);          /* 设置重试次数 */
46
47     for (idx = 0; idx < size; ++idx, ++mem_addr) {
48         tmp.addr = mem_addr;
49         cswap = tmp.bytes[0];
50         tmp.bytes[0] = tmp.bytes[1];
51         tmp.bytes[1] = cswap;
52         write(fd, &tmp.addr, 2);
53         read(fd, &buf[idx], 1);
54     }
55     buf[size] = 0;
56     close(fd);
57     printf("Read %d char: %s\n", size, buf);
58     return 0;
59 }
```

代码清单 15.17 通过 O\_RDWR IOCTL 读写 I<sup>2</sup>C 设备

```

1 #include <stdio.h>
2 #include <linux/types.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5 #include <stdlib.h>
6 #include <sys/types.h>
7 #include <sys/ioctl.h>
8 #include <errno.h>
9 #include <assert.h>
10 #include <string.h>
11 #include <linux/i2c.h>
12 #include <linux/i2c-dev.h>
13
14 int main(int argc, char **argv)
15 {
16     struct i2c_rdwr_ioctl_data work_queue;
17     unsigned int idx;
18     unsigned int fd;
19     unsigned int slave_address, reg_address;
```

```
20 unsigned char val;
21 int i;
22 int ret;
23
24 if (argc < 4) {
25     printf("Usage:\n%s /dev/i2c-x start_addr reg_addr\n", argv[0]);
26     return 0;
27 }
28
29 fd = open(argv[1], O_RDWR);
30
31 if (!fd) {
32     printf("Error on opening the device file\n");
33     return 0;
34 }
35 sscanf(argv[2], "%x", &slave_address);
36 sscanf(argv[3], "%x", &reg_address);
37
38 work_queue.nmsgs = 2; /* 消息数量 */
39 work_queue.msgs = (struct i2c_msg*)malloc(work_queue.nmsgs * sizeof(struct
40             i2c_msg));
41 if (!work_queue.msgs) {
42     printf("Memory alloc error\n");
43     close(fd);
44     return 0;
45 }
46
47 ioctl(fd, I2C_TIMEOUT, 2); /* 设置超时 */
48 ioctl(fd, I2C_RETRIES, 1); /* 设置重试次数 */
49
50 for (i = reg_address; i < reg_address + 16; i++) {
51     val = i;
52     (work_queue.msgs[0]).len = 1;
53     (work_queue.msgs[0]).addr = slave_address;
54     (work_queue.msgs[0]).buf = &val;
55
56     (work_queue.msgs[1]).len = 1;
57     (work_queue.msgs[1]).flags = I2C_M_RD;
58     (work_queue.msgs[1]).addr = slave_address;
59     (work_queue.msgs[1]).buf = &val;
60
61     ret = ioctl(fd, I2C_RDWR, (unsigned long) &work_queue);
62     if (ret < 0)
63         printf("Error during I2C_RDWR ioctl with error code: %d\n", ret);
64 else
65     printf("reg:%02x val:%02x\n", i, val);
66 }
67 close(fd);
68 return ;
69 }
```

使用该工具可指定读取某I<sup>2</sup>C控制器上某I<sup>2</sup>C从设备的某寄存器，如读I<sup>2</sup>C控制器0上的地址为0x18的从设备，从寄存器0x20开始读：

```
# i2c-test /dev/i2c-0 0x18 0x20
reg:20 val:07
reg:21 val:00
reg:22 val:00
reg:23 val:00
reg:24 val:00
reg:25 val:00
reg:26 val:00
reg:27 val:00
reg:28 val:00
reg:29 val:00
reg:2a val:00
reg:2b val:00
reg:2c val:00
reg:2d val:00
reg:2e val:00
reg:2f val:00
```

## 15.5 Tegra I<sup>2</sup>C 总线驱动实例

NVIDIA Tegra I<sup>2</sup>C总线驱动位于drivers/i2c/busses/i2c-tegra.c下，这里我们不具体研究它的硬件细节，只看一下驱动的框架和流程。

I<sup>2</sup>C总线驱动是一个单独的驱动，在模块的加载和卸载函数中，只需注册和注销一个platform\_driver结构体，如代码清单15.18所示。

代码清单 15.18 Tegra I<sup>2</sup>C 总线驱动的模块加载与卸载

---

```
1 /* Match table for of_platform binding */
2 static const struct of_device_id tegra_i2c_of_match[] = {
3     { .compatible = "nvidia,tegral14-i2c", .data = &tegral14_i2c_hw, },
4     { .compatible = "nvidia,tegra30-i2c", .data = &tegra30_i2c_hw, },
5     { .compatible = "nvidia,tegra20-i2c", .data = &tegra20_i2c_hw, },
6     { .compatible = "nvidia,tegra20-i2c-dvc", .data = &tegra20_i2c_hw, },
7     {},
8 };
9 MODULE_DEVICE_TABLE(of, tegra_i2c_of_match);
10
11 static struct platform_driver tegra_i2c_driver = {
12     .probe    = tegra_i2c_probe,
13     .remove   = tegra_i2c_remove,
14     .driver   = {
15         .name    = "tegra-i2c",
16         .owner   = THIS_MODULE,
17         .of_match_table = tegra_i2c_of_match,
18         .pm      = TEGRA_I2C_PM,
```

```

19 },
20 };
21
22 static int __init tegra_i2c_init_driver(void)
23 {
24     return platform_driver_register(&tegra_i2c_driver);
25 }
26
27 static void __exit tegra_i2c_exit_driver(void)
28 {
29     platform_driver_unregister(&tegra_i2c_driver);
30 }
31
32 subsys_initcall(tegra_i2c_init_driver);
33 module_exit(tegra_i2c_exit_driver);

```

---

当在 arch/arm/mach-tegra 下创建一个名字为 tegra-i2c 的同名 platform\_device，或者在 tegra 的设备树中添加了 tegra\_i2c\_of\_match 匹配表兼容的节点后，上述 platform\_driver 中的 probe() 函数会执行。

其中 probe 指针指向的 tegra\_i2c\_probe() 函数将被调用，以初始化适配器硬件、申请适配器要的内存、时钟、中断等资源，最终注册适配器，如代码清单 15.19 所示。

代码清单 15.19 Tegra I<sup>2</sup>C 总线驱动中的 tegra\_i2c\_probe() 函数

```

1 static int tegra_i2c_probe(struct platform_device *pdev)
2 {
3     struct tegra_i2c_dev *i2c_dev;
4     struct resource *res;
5     struct clk *div_clk;
6     struct clk *fast_clk;
7     void __iomem *base;
8     int irq;
9     int ret = 0;
10
11    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
12    base = devm_ioremap_resource(&pdev->dev, res);
13    ...
14    res = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
15    ...
16    irq = res->start;
17
18    div_clk = devm_clk_get(&pdev->dev, "div-clk");
19    ...
20
21    i2c_dev = devm_kzalloc(&pdev->dev, sizeof(*i2c_dev), GFP_KERNEL);
22    ...
23
24    i2c_dev->base = base;
25    i2c_dev->div_clk = div_clk;

```

```

26     i2c_dev->adapter.algo = &tegra_i2c_algo;
27     i2c_dev->irq = irq;
28     i2c_dev->cont_id = pdev->id;
29     i2c_dev->dev = &pdev->dev;
30
31     i2c_dev->rst = devm_reset_control_get(&pdev->dev, "i2c");
32     ...
33
34     ret = of_property_read_u32(i2c_dev->dev->of_node, "clock-frequency",
35                               &i2c_dev->bus_clk_rate);
36     if (ret)
37         i2c_dev->bus_clk_rate = 100000; /* default clock rate */
38
39     i2c_dev->hw = &tegra20_i2c_hw;
40
41     ...
42     init_completion(&i2c_dev->msg_complete);
43
44     ...
45
46     platform_set_drvdata(pdev, i2c_dev);
47
48     ret = tegra_i2c_init(i2c_dev);
49     ...
50
51     ret = devm_request_irq(&pdev->dev, i2c_dev->irq,
52                           tegra_i2c_isr, 0, dev_name(&pdev->dev), i2c_dev);
53     ...
54
55     i2c_set_adapdata(&i2c_dev->adapter, i2c_dev);
56     i2c_dev->adapter.owner = THIS_MODULE;
57     i2c_dev->adapter.class = I2C_CLASS_DEPRECATED;
58     strlcpy(i2c_dev->adapter.name, "Tegra I2C adapter",
59             sizeof(i2c_dev->adapter.name));
60     i2c_dev->adapter.algo = &tegra_i2c_algo;
61     i2c_dev->adapter.dev.parent = &pdev->dev;
62     i2c_dev->adapter.nr = pdev->id;
63     i2c_dev->adapter.dev.of_node = pdev->dev.of_node;
64
65     ret = i2c_add_numbered_adapter(&i2c_dev->adapter);
66     ...
67
68     return 0;
69 }

```

有与 `tegra_i2c_probe()` 函数相反功能的函数是 `tegra_i2c_remove()` 函数，它在适配器模块卸载函数调用 `platform_driver_unregister()` 函数时通过 `platform_driver` 的 `remove` 指针方式被调用。`tegra_i2c_remove()` 的代码如清单 15.20 所示。

**代码清单 15.20 Tegra I<sup>2</sup>C 总线驱动中的 tegra\_i2c\_remove() 函数**


---

```

1 static int tegra_i2c_remove(struct platform_device *pdev)
2 {
3     struct tegra_i2c_dev *i2c_dev = platform_get_drvdata(pdev);
4     i2c_del_adapter(&i2c_dev->adapter);
5     return 0;
6 }

```

---

代码清单 15.19 和代码清单 15.20 中的 tegra\_i2c\_dev 结构体可进行适配器所有信息的封装，类似于私有信息结构体，代码清单 15.21 所示为 tegra\_i2c\_dev 结构体的定义。我们在编程中要时刻牢记 Linux 这个编程习惯，这实际上也是面向对象的一种体现。

**代码清单 15.21 tegra\_i2c\_dev 结构体**


---

```

1 struct tegra_i2c_dev {
2     struct device *dev;
3     const struct tegra_i2c_hw_feature *hw;
4     struct i2c_adapter adapter;
5     struct clk *div_clk;
6     struct clk *fast_clk;
7     struct reset_control *rst;
8     void __iomem *base;
9     int cont_id;
10    int irq;
11    bool irq_disabled;
12    int is_dvc;
13    struct completion msg_complete;
14    int msg_err;
15    u8 *msg_buf;
16    size_t msg_buf_remaining;
17    int msg_read;
18    u32 bus_clk_rate;
19    bool is_suspended;
20 };

```

---

tegra\_i2c\_probe() 函数中的 platform\_set\_drvdata(pdev,i2c\_dev) 和 i2c\_set\_adapdata(&i2c\_dev->adapter,i2c\_dev) 已经把这个结构体的实例依附到了 platform\_device 和 i2c\_adapter 的私有数据上了，在其他地方只要用相应的方法就可以把这个结构体的实例取出来。

由代码清单 15.19 的第 60 行可以看出，与 I<sup>2</sup>C 适配器对应的 i2c\_algorithm 结构体实例为 tegra\_i2c\_algo，代码清单 15.22 给出为 tegra\_i2c\_algo 的定义。

**代码清单 15.22 tegra\_i2c\_algo 结构体**


---

```

1 static const struct i2c_algorithm tegra_i2c_algo = {
2     .master_xfer      = tegra_i2c_xfer,
3     .functionality    = tegra_i2c_func,
4 };

```

---

上述代码第一行指定了 Tegra I<sup>2</sup>C 总线通信传输函数 `tegra_i2c_xfer()`，这个函数非常关键，所有在 I<sup>2</sup>C 总线上对设备的访问最终应该由它来完成，代码清单 15.23 所示为这个重要函数以及其依赖的 `tegra_i2c_xfer_msg()` 函数的源代码。

代码清单 15.23 Tegra I<sup>2</sup>C 总线驱动的 `tegra_i2c_xfer()` 函数

```

1 static int tegra_i2c_xfer_msg(struct tegra_i2c_dev *i2c_dev,
2     struct i2c_msg *msg, enum msg_end_type end_state)
3 {
4     ...
5     i2c_dev->msg_buf = msg->buf;
6     i2c_dev->msg_buf_remaining = msg->len;
7     i2c_dev->msg_err = I2C_ERR_NONE;
8     i2c_dev->msg_read = (msg->flags & I2C_M_RD);
9     reinit_completion(&i2c_dev->msg_complete);
10
11    packet_header = (0 << PACKET_HEADER0_HEADER_SIZE_SHIFT) |
12        PACKET_HEADER0_PROTOCOL_I2C |
13        (i2c_dev->cont_id << PACKET_HEADER0_CONT_ID_SHIFT) |
14        (1 << PACKET_HEADER0_PACKET_ID_SHIFT);
15    i2c_writel(i2c_dev, packet_header, I2C_TX_FIFO);
16
17    packet_header = msg->len - 1;
18    i2c_writel(i2c_dev, packet_header, I2C_TX_FIFO);
19
20    ...
21
22    ret = wait_for_completion_timeout(&i2c_dev->msg_complete, TEGRA_I2C_TIMEOUT);
23    ...
24 }
25
26 static int tegra_i2c_xfer(struct i2c_adapter *adap, struct i2c_msg msgs[],
27     int num)
28 {
29     struct tegra_i2c_dev *i2c_dev = i2c_get_adapdata(adap);
30     int i;
31     int ret = 0;
32
33     ...
34
35     for (i = 0; i < num; i++) {
36         enum msg_end_type end_type = MSG_END_STOP;
37         if (i < (num - 1)) {
38             if (msgs[i + 1].flags & I2C_M_NOSTART)
39                 end_type = MSG_END_CONTINUE;
40             else
41                 end_type = MSG_END_REPEAT_START;
42         }
43         ret = tegra_i2c_xfer_msg(i2c_dev, &msgs[i], end_type);
44         if (ret)

```

---

```

45         break;
46     }
47     tegra_i2c_clock_disable(i2c_dev);
48     return ret ?: i;
49 }

```

---

从代码层面上看，第35行的for循环遍历所有的i2c\_msg，而每个i2c\_msg则由tegra\_i2c\_xfer\_msg()函数处理，它每次发起硬件操作后，实际上需要通过wait\_for\_completion\_timeout()等待传输的完成，因此这里面就会有一个被调度出去的过程。中断到来且I<sup>2</sup>C的包传输结束的时候，就是唤醒这个睡眠进程的时候，如代码清单15.24所示。

代码清单15.24 Tegra I<sup>2</sup>C总线驱动的中断服务程序

---

```

1 static irqreturn_t tegra_i2c_isr(int irq, void *dev_id)
2 {
3     ...
4
5     if (status & I2C_INT_PACKET_XFER_COMPLETE) {
6         BUG_ON(i2c_dev->msg_buf_remaining);
7         complete(&i2c_dev->msg_complete);
8     }
9     return IRQ_HANDLED;
10    ...
11 }

```

---

## 15.6 AT24xx EEPROM的I<sup>2</sup>C设备驱动实例

drivers/misc/eeprom/at24.c文件支持大多数I<sup>2</sup>C接口的EEPROM，正如我们之前所述，一个具体的I<sup>2</sup>C设备驱动由i2c\_driver的形式进行组织，用于将设备挂接于I<sup>2</sup>C总线，组织好了后，再完成设备本身所属类型的驱动。对于EEPROM而言，设备本身的驱动以bin\_attribute二进制sysfs节点形式呈现。代码清单15.25给出了该驱动的框架。

代码清单15.25 AT24xx EEPROM驱动

---

```

1 struct at24_data {
2     struct at24_platform_data chip;
3     ...
4     struct bin_attribute bin;
5     ...
6 };
7
8 static const struct i2c_device_id at24_ids[] = {
9     /* needs 8 addresses as A0-A2 are ignored */
10    { "24c00", AT24_DEVICE_MAGIC(128 / 8, AT24_FLAG_TAKE8ADDR) },
11    /* old variants can't be handled with this generic entry! */
12    { "24c01", AT24_DEVICE_MAGIC(1024 / 8, 0) },

```

---

```
13         { "24c02", AT24_DEVICE_MAGIC(2048 / 8, 0) },
14         ...
15         /* END OF LIST */
16     };
17 MODULE_DEVICE_TABLE(i2c, at24_ids);
18
19 static ssize_t at24_eeprom_read(struct at24_data *at24, char *buf,
20                                unsigned offset, size_t count)
21 {
22     struct i2c_msg msg[2];
23     ...
24     i2c_transfer(client->adapter, msg, 2);
25     ...
26 }
27
28 static ssize_t at24_read(struct at24_data *at24,
29                         char *buf, loff_t off, size_t count)
30 {
31     ...
32
33     status = at24_eeprom_read(at24, buf, off, count);
34     ...
35
36     return retval;
37 }
38
39 static ssize_t at24_bin_read(struct file *filp, struct kobject *kobj,
40                             struct bin_attribute *attr,
41                             char *buf, loff_t off, size_t count)
42 {
43     struct at24_data *at24;
44
45     at24 = dev_get_drvdata(container_of(kobj, struct device, kobj));
46     return at24_read(at24, buf, off, count);
47 }
48
49 ...
50
51 static int at24_probe(struct i2c_client *client, const struct i2c_device_id *id)
52 {
53     ...
54     sysfs_bin_attr_init(&at24->bin);
55     at24->bin.attr.name = "eeprom";
56     at24->bin.attr.mode = chip.flags & AT24_FLAG_IRUGO ? S_IRUGO : S_IRUSR;
57     at24->bin.read = at24_bin_read;
58     at24->bin.size = chip.byte_len;
59
60     ...
61     return err;
```

```

62 }
63
64 static int at24_remove(struct i2c_client *client)
65 {
66     ...
67     sysfs_remove_bin_file(&client->dev.kobj, &at24->bin);
68     ...
69
70     return 0;
71 }
72
73 static struct i2c_driver at24_driver = {
74     .driver = {
75         .name = "at24",
76         .owner = THIS_MODULE,
77     },
78     .probe = at24_probe,
79     .remove = at24_remove,
80     .id_table = at24_ids,
81 };
82
83 static int __init at24_init(void)
84 {
85     ...
86     return i2c_add_driver(&at24_driver);
87 }
88 module_init(at24_init);
89
90 static void __exit at24_exit(void)
91 {
92     i2c_del_driver(&at24_driver);
93 }
94 module_exit(at24_exit);

```

---

drivers/misc/eeprom/at24.c 不依赖于具体的 CPU 和 I<sup>2</sup>C 控制器的硬件特性，因此，如果某一电路板包含该外设，只需要在板级文件中添加对应的 i2c\_board\_info，如：

```

static struct i2c_board_info i2c_devs0[] __initdata = {
    { I2C_BOARD_INFO("24c02", 0x57), },
};

```

在支持设备树的情况下，简单地在.dts 文件中添加一个节点即可：

```

i2c@11000 {
    status = "okay";
    ...
    eeprom@57 {
        compatible = "atmel,24c02";

```

```
    reg = <0x57>;  
}  
};
```

## 15.7 总结

Linux 的 I<sup>2</sup>C 驱动体系结构相当复杂，它主要由 3 部分组成，即 I<sup>2</sup>C 核心、I<sup>2</sup>C 总线驱动和 I<sup>2</sup>C 设备驱动。I<sup>2</sup>C 核心是 I<sup>2</sup>C 总线驱动和 I<sup>2</sup>C 设备驱动的中间枢纽，它以通用的、与平台无关的接口实现了 I<sup>2</sup>C 中设备与适配器的沟通。I<sup>2</sup>C 总线驱动填充 i2c\_adapter 和 i2c\_algorithm 结构体，I<sup>2</sup>C 设备驱动填充 i2c\_driver 结构体并实现其本身所对应设备类型的驱动。

另外，系统中 i2c-dev.c 文件定义的主设备号为 89 的设备可以方便地给应用程序提供读写 I<sup>2</sup>C 设备寄存器的能力，使得工程师在大多数时候并不需要为具体的 I<sup>2</sup>C 设备驱动定义文件操作接口。