LINUX

业界权威机构和专家强力推荐
多年培训、研发经验的总结

设备驱动开发详解

华清远见嵌入式培训中心 宋宝华 编著

Linux

附源代码光盘

人民邮电出版社
POSTS & TELECOM PRESS

第 15 章　Linux 的 I2C 核心、总线与设备驱动

I^2C 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^2C 总线被非常广泛地应用在 EEPROM、实时钟、小型 LCD 等设备与 CPU 的接口中。

Linux 系统定义了 I^2C 驱动体系结构，在 Linux 系统中，I^2C 驱动由 3 部分组成，即 I^2C 核心、I^2C 总线驱动和 I^2C 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^2C 框架。

6.1 节对 Linux I^2C 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。

6.2 节对 Linux I^2C 核心进行分析，讲解 i2c-core.c 文件的功能和主要函数的实现。

6.3 节、6.4 节分别详细介绍 I^2C 总线驱动和 I^2C 设备驱动的编写方法，给出可供参考的设计模板。

6.5 节、6.6 节以 6.3 节和 6.4 节给出的设计模板为基础，讲解 S3C2410 ARM 处理器 I^2C 总线驱动及挂接在其上的 SAA7113H 视频模拟/数字转换芯片设备驱动的编写方法。

15.1 Linux 的 I²C 体系结构

Linux 的 I²C 体系结构分为 3 个组成部分。

（1）I²C 核心。

I²C 核心提供了 I²C 总线驱动和设备驱动的注册、注销方法，I²C 通信方法（即“algorithm”，笔者认为直译为“运算方法”并不合适，为免引起误解，下文将直接使用“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。

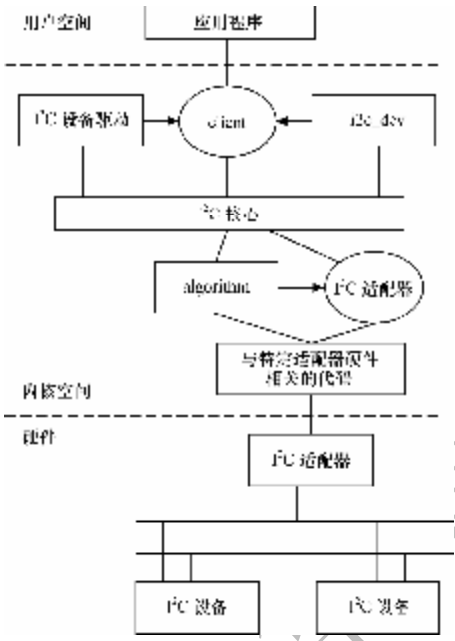

图 15.1　Linux I²C 体系结构

（2）I²C 总线驱动。

I²C 总线驱动是对 I²C 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。

I²C 总线驱动主要包含了 I²C 适配器数据结构 i2c_adapter、I²C 适配器的 algorithm 数据结构 i2c_algorithm 和控制 I²C 适配器产生通信信号的函数。

经由 I²C 总线驱动的代码，我们可以控制 I²C 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）I²C 设备驱动。

I²C 设备驱动是对 I²C 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 I²C 适配器上，通过 I²C 适配器与 CPU 交换数据。

I²C 设备驱动主要包含了数据结构 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 I²C 设备都在 sysfs 文件系统中显示，存于/sys/bus/i2c/目录，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
|   |-- 0-0048 -> ../../../devices/legacy/i2c-0/0-0048
|   |-- 0-0049 -> ../../../devices/legacy/i2c-0/0-0049
|   |-- 0-004a -> ../../../devices/legacy/i2c-0/0-004a
|   |-- 0-004b -> ../../../devices/legacy/i2c-0/0-004b
|   |-- 0-004c -> ../../../devices/legacy/i2c-0/0-004c
```

```
    |   |-- 0-004d -> ../../../devices/legacy/i2c-0/0-004d
    |   |-- 0-004e -> ../../../devices/legacy/i2c-0/0-004e
    |   '-- 0-004f -> ../../../devices/legacy/i2c-0/0-004f
    '-- drivers
       |-- i2c_adapter
       '-- lm75
          |-- 0-0048 -> ../../../../devices/legacy/i2c-0/0-0048
          |-- 0-0049 -> ../../../../devices/legacy/i2c-0/0-0049
          |-- 0-004a -> ../../../../devices/legacy/i2c-0/0-004a
          |-- 0-004b -> ../../../../devices/legacy/i2c-0/0-004b
          |-- 0-004c -> ../../../../devices/legacy/i2c-0/0-004c
          |-- 0-004d -> ../../../../devices/legacy/i2c-0/0-004d
          |-- 0-004e -> ../../../../devices/legacy/i2c-0/0-004e
          '-- 0-004f -> ../../../../devices/legacy/i2c-0/0-004f
```

在 Linux 内核源代码中的 drivers 目录下包含一个 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^2C 核心的功能以及/proc/bus/i2c*接口。

（2）i2c-dev.c。

实现了 I^2C 适配器设备文件的功能，每一个 I^2C 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 并没有针对特定的设备而设计，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^2C 设备的存储空间或寄存器，并控制 I^2C 设备的工作方式。

（3）chips 文件夹。

这个目录中包含了一些特定的 I^2C 设备驱动，如 Dallas 公司的 DS1337 实时钟芯片、EPSON 公司的 RTC8564 实时钟芯片和 I^2C 接口的 EEPROM 驱动等。

（4）busses 文件夹。

这个文件中包含了一些 I^2C 总线的驱动，如 S3C2410 的 I^2C 控制器驱动为 i2c-s3c2410.c。

（5）algos 文件夹。

实现了一些 I^2C 总线适配器的 algorithm。

此外，内核中的 i2c.h 这个头文件对 i2c_driver、i2c_client、i2c_adapter 和 i2c_algorithm 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关键，代码清单 15.1、15.2、15.3、15.4 分别给出了它们的定义。

代码清单 15.1　i2c_adapter 结构体

```
1  struct i2c_adapter {
2   struct module *owner;/*所属模块*/
3   unsigned int id;   /*algorithm 的类型，定义于 i2c-id.h，以 I2C_ALGO_
开始*/
4   unsigned int class;
```

```
    5   struct i2c_algorithm *algo;/*总线通信方法结构体指针    */
    6   void *algo_data;    /* algorithm 数据 */
    7   int (*client_register)(struct i2c_client *);  /*client 注册时调用*/
    8   int (*client_unregister)(struct i2c_client *); /*client 注销时调用
*/
    9   struct semaphore bus_lock;    /*控制并发访问的自旋锁*/
    10  struct semaphore clist_lock;
    11  int timeout;
    12  int retries;    /*重试次数*/
    13  struct device dev;      /* 适配器设备 */
    14  struct class_device class_dev; /* 类设备 */
    15  int nr;
    16  struct list_head clients;  /* client 链表头*/
    17  struct list_head list;
    18  char name[I2C_NAME_SIZE];  /*适配器名称*/
    19  struct completion dev_released;    /*用于同步*/
    20  struct completion class_dev_released;
    21};
```

代码清单 15.2　i2c_algorithm 结构体

```
    1  struct i2c_algorithm {
    2   int (*master_xfer)(struct i2c_adapter *adap,struct i2c_msg *msgs,
    3                   int num);  /*i2c 传输函数指针*/
    4   int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr,   /*smbus
传输函数指针*/
    5                   unsigned short flags, char read_write,
    6                   u8 command, int size, union i2c_smbus_data * data);
    7   int (*slave_send)(struct i2c_adapter *,char*,int);/*当 I2C 适配器为 slave
时，发送函数*/
    8   int (*slave_recv)(struct i2c_adapter *,char*,int); /*当 I2C 适配器为 slave
时，接收函数*/
    9   int (*algo_control)(struct i2c_adapter *, unsigned int, unsigned long);
/*类似 ioctl*/
    10  u32 (*functionality) (struct i2c_adapter *);/*返回适配器支持的功能*/
    11 };
```

上述代码第 4 行对应为 SMBus 传输函数指针，SMBus 大部分基于 I^2C 总线规范，SMBus 不需增加额外引脚。与 I^2C 总线相比，SMBus 增加了一些新的功能特性，在访问时序也有一定的差异。

代码清单 15.3　i2c_driver 结构体

```
    1  struct i2c_driver {
    2   int id;
    3   unsigned int class;
    4   int (*attach_adapter)(struct i2c_adapter *);   /*依附 i2c_adapter
函数指针 */
    5   int (*detach_adapter)(struct i2c_adapter *);   /*脱离 i2c_adapter
函数指针*/
    6   int (*detach_client)(struct i2c_client *);         /*i2c   client
脱离函数指针*/
    7   int (*command)(struct i2c_client *client,unsigned int cmd, void *arg);
/*类似 ioctl*/
```

```
8   struct device_driver driver;        /*设备驱动结构体*/
9   struct list_head list;              /*链表头*/
10 };
```

代码清单 15.4　i2c_client 结构体

```
1  struct i2c_client {
2   unsigned int flags;         /* 标志 */
3   unsigned short addr;            /* 低 7 位为芯片地址 */
4   struct i2c_adapter *adapter;    /*依附的 i2c_adapter*/
5   struct i2c_driver *driver;      /*依附的 i2c_driver */
6   int usage_count;                /* 访问计数  */
7   struct device dev;          /* 设备结构体 */
8   struct list_head list;          /* 链表头 */
9   char name[I2C_NAME_SIZE];       /* 设备名称 */
10  struct completion released;     /* 用于同步 */
11 };
```

下面分析 i2c_driver、i2c_client、i2c_adapter 和 i2c_algorithm 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^2C 适配器需要 i2c_algorithm 中提供的通信函数来控制适配器上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 中包含其使用的 i2c_algorithm 的指针。

i2c_algorithm 中的关键函数 master_xfer()用于产生 I^2C 访问周期需要的信号，以 i2c_msg（即 I^2C 消息）为单位。i2c_msg 结构体也非常关键，代码清单 15.5 给出了它的定义。

代码清单 15.5　i2c_msg 结构体

```
1 struct i2c_msg {
2   __u16 addr;     /* 设备地址*/
3   __u16 flags;        /* 标志 */
4   __u16 len;      /* 消息长度*/
5   __u8 *buf;      /* 消息数据*/
6 };
```

（2）i2c_driver 与 i2c_client。

i2c_driver 对应一套驱动方法，是纯粹的用于辅助作用的数据结构，它不对应于任何的物理实体。i2c_client 对应于真实的物理设备，每个 I^2C 设备都需要一个 i2c_client 来描述。i2c_client 一般被包含在 I^2C 字符设备的私有信息结构体中。

i2c_driver 与 i2c_client 发生关联的时刻在 i2c_driver 的 attach_adapter()函数被运行时。attach_adapter()会探测物理设备，当确定一个 client 存在时，把该 client 使用的 i2c_client 数据结构的 adapter 指针指向对应的 i2c_adapter，driver 指针指向该 i2c_driver，并会调用 i2c_adapter 的 client_register()函数。相反的过程发生在 i2c_driver 的 detach_client()函数被调用的时候。

（3）i2c_adpater 与 i2c_client。

i2c_adpater 与 i2c_client 的关系与 I^2C 硬件体系中适配器和设备的关系一致，即 i2c_client 依附于 i2c_adpater。由于一个适配器上可以连接多个 I^2C 设备，所以一个 i2c_adpater 也可以被多个 i2c_client 依附，i2c_adpater 中包括依附于它的 i2c_client 的链表。

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

从上面的分析可知，虽然 I^2C 硬件体系结构比较简单，但是 I^2C 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，面对复杂的 Linux I^2C 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，可以使我们面对具体问题时迅速地抓住重点。

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。即便上述设备驱动都存在于 Linux 内核中，其基于的平台也可能与我们的电路板不一样。因此，工程师要实现的主要工作如下。

- 提供 I^2C 适配器的硬件驱动，探测、初始化 I^2C 适配器（如申请 I^2C 的 I/O 地址和中断号）、驱动 CPU 控制的 I^2C 适配器从硬件上产生各种信号以及处理 I^2C 中断等。
- 提供 I^2C 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^2C 设备驱动与 i2c_driver 接口，用具体设备 yyy 的 yyy_attach_adapter()函数指针、yyy_detach_client()函数指针和 yyy_command()函数指针的赋值给 i2c_driver 的 attach_ adapter、detach_adapter 和 detach_client 指针。
- 实现 I^2C 设备驱动的文件操作接口，即实现具体设备 yyy 的 yyy_read()、yyy_write()和 yyy_ioctl()函数等。

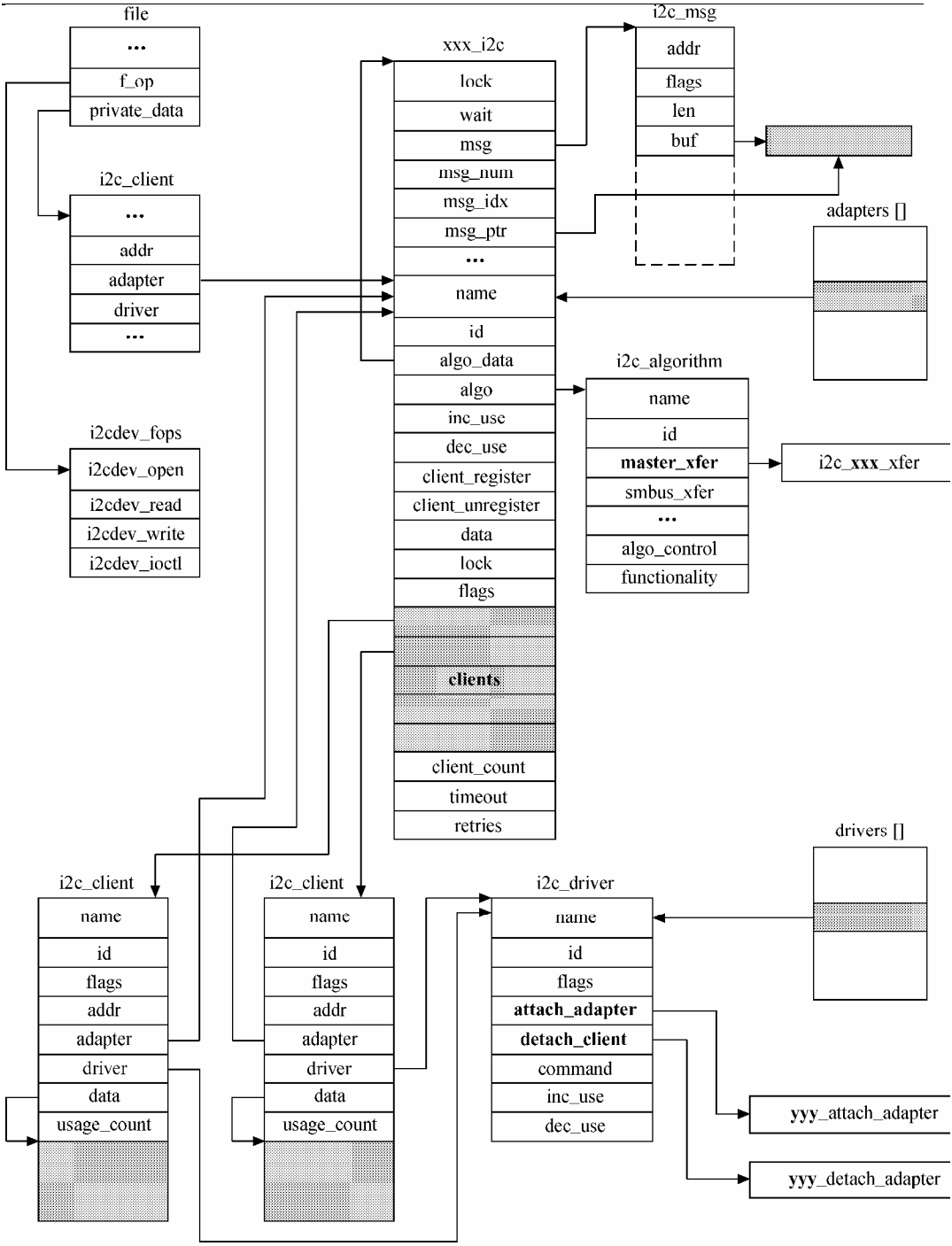

图 15.2　I²C 驱动的各数据结构的关系

上述工作中前两个属于 I²C 总线驱动，后两个属于 I²C 设备驱动，做完这些工作，系统会增加两个内核模块。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 I²C 核心

I²C 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 I²C 总线驱动和设备驱动之间依赖于 I²C 核心作为纽带。I²C 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);
int 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);
int i2c_del_driver(struct i2c_driver *driver);
inline int i2c_add_driver(struct i2c_driver *driver);
```

（3）i2c_client 依附/脱离。

```
int i2c_attach_client(struct i2c_client *client);
int i2c_detach_client(struct i2c_client *client);
```

当一个具体的 client 被侦测到并被关联的时候，设备和 sysfs 文件将被注册。相反地，在 client 被取消关联的时候，sysfs 文件和设备也被注销，如代码清单 15.6 所示。

代码清单 15.6 I²C 核心的 client attach/detach 函数

```
1  int i2c_attach_client(struct i2c_client *client)
2  {
3    ...
4   device_register(&client->dev);
5   device_create_file(&client->dev, &dev_attr_client_name);
6
7   return 0;
8  }
9
10 int i2c_detach_client(struct i2c_client *client)
11 {
12   ...
13  device_remove_file(&client->dev, &dev_attr_client_name);
14  device_unregister(&client->dev);
15   ...
16 }
```

（4）I²C 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^2C 适配器和 I^2C 设备之间的一组消息交互，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 15.7、代码清单 15.8 所示。

代码清单 15.7 I^2C 核心的 i2c_master_send 函数

```
1  int i2c_master_send(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    /*构造一个写消息*/
7   msg.addr = client->addr;
8   msg.flags = client->flags & I2C_M_TEN;
9   msg.len = count;
10  msg.buf = (char *)buf;
11  /*传输消息*/
12  ret = i2c_transfer(adap, &msg, 1);
13
14  return (ret == 1) ? count : ret;
15 }
```

代码清单 15.8 I^2C 核心的 i2c_master_recv 函数

```
1  int i2c_master_recv(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   /*构造一个读消息*/
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  /*传输消息*/
13  ret = i2c_transfer(adap, &msg, 1);
14
15  /* 成功（1 条消息被处理）， 返回读的字节数 */
16  return (ret == 1) ? count : ret;
17 }
```

i2c_transfer()函数本身不具备驱动适配器物理硬件完成消息交互的能力，它只是寻找到 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 15.9 所示。

代码清单 15.9 I²C 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       down(&adap->bus_lock);
7       ret = adap->algo->master_xfer(adap,msgs,num); /* 消息传输 */
8       up(&adap->bus_lock);
9       return ret;
10  } else {
11      dev_dbg(&adap->dev, "I2C level transfers not supported\n");
12      return -ENOSYS;
13  }
14 }
```

（5）I²C 控制命令分配。

下面函数有助于将发给 I²C 适配器设备文件 ioctl 的命令分配给对应适配器的 algorithm 的 algo_control()函数或 i2c_driver 的 command()函数，如下所示：

```
    int i2c_control(struct i2c_client *client, unsigned int cmd, unsigned
long arg);
    void i2c_clients_command(struct i2c_adapter *adap, unsigned int cmd,
void *arg);
```

15.3 Linux I²C 总线驱动

15.3.1 I²C 适配器驱动加载与卸载

I²C 总线驱动模块的加载函数要完成两个工作。

- 初始化 I²C 适配器所使用的硬件资源，如申请 I/O 地址、中断号等。
- 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。

I²C 总线驱动模块的卸载函数要完成的工作与加载函数相反。

- 释放 I²C 适配器所使用的硬件资源，如释放 I/O 地址、中断号等。
- 通过 i2c_del_adapter()删除 i2c_adapter 的数据结构。

代码清单 15.10 所示为 I²C 适配器驱动的模块加载和卸载函数的模板。

代码清单 15.10 I²C 总线驱动的模块加载和卸载函数模板

```
1  static int __init i2c_adapter_xxx_init(void)
2  {
3    xxx_adpater_hw_init();
4    i2c_add_adapter(&xxx_adapter);
5  }
6
7  static void __exit i2c_adapter_xxx_exit(void)
8  {
9    xxx_adpater_hw_free();
10   i2c_del_adapter(&xxx_adapter);
11 }
```

上述代码中 xxx_adpater_hw_init()和 xxx_adpater_hw_free()函数的实现都与具体的 CPU 和 I²C 设备硬件直接相关。

15.3.2 I²C 总线通信方法

我们需要为特定的 I²C 适配器实现其通信方法，主要实现 i2c_algorithm 的 master_xfer()函数和 functionality()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_I2C、I2C_FUNC_10BIT_ADDR 、I2C_FUNC_SMBUS_READ_BYTE 、I2C_FUNC_SMBUS_WRITE_BYTE 等。

master_xfer()函数在 I²C 适配器上完成传递给它的 i2c_msg 数组中的每个 I²C 消息，代码清单 15.11 所示为 xxx 设备的 master_xfer()函数模板。

代码清单 15.11 I²C 总线驱动 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    {
7      i2c_adapter_xxx_start(); /*产生开始位*/
8      /*是读消息*/
9      if (msgs[i]->flags &I2C_M_RD)
10     {
11       i2c_adapter_xxx_setaddr((msg->addr << 1) | 1); /*发送从设备读地
址*/
12       i2c_adapter_xxx_wait_ack(); /*获得从设备的 ack*/
13       i2c_adapter_xxx_readbytes(msgs[i]->buf, msgs[i]->len); /*读取
msgs[i] ->len
14         长的数据到 msgs[i]->buf*/
15     }
16     else
```

```
    17      /*是写消息*/
    18     {
    19       i2c_adapter_xxx_setaddr(msg->addr << 1); /*发送从设备写地址*/
    20       i2c_adapter_xxx_wait_ack(); /*获得从设备的 ack*/
    21       i2c_adapter_xxx_writebytes(msgs[i]->buf, msgs[i]->len); /*读
取 msgs[i] ->len
    22         长的数据到 msgs[i]->buf*/
    23     }
    24   }
    25   i2c_adapter_xxx_stop(); /*产生停止位*/
    26 }
```

上述代码实际上给出了一个 master_xfer()函数处理 I^2C 消息数组的流程，对于数组中的每个消息，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，对最后的消息还需产生一个停止位。如图 15.3 所示为整个 master_xfer()完成的时序。

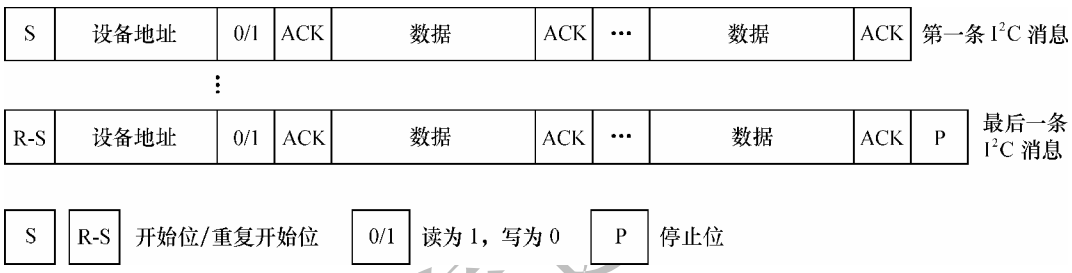

图 15.3 algorithm 中 master_xfer 的时序

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^2C 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() 用于从从设备上接收一串数据，i2c_adapter_xxx_writebytes()用于向从设备写入一串数据，这两个函数的内部也会涉及 I^2C 总线协议中的 ACK 应答。

master_xfer()函数的实现在形式上会很多样，即便是 Linux 内核源代码中已经给出的一些 I^2C 总线驱动的 master_xfer()函数，由于由不同的组织或个人完成，风格上的差别也非常大，不一定能与模板完全对应，如 master_xfer()函数模板给出的消息处理是顺序进行的，而有的驱动以中断方式来完成这个流程（15.5 节的实例即是如此）。不管具体怎么实施，流程的本质都是不变的。因为这个流程不以驱动工程师的意志为转移，最终由 I^2C 总线硬件上的通信协议决定。

多数 I^2C 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^2C 消息数组指针、数组索引及 I^2C 适配器 algorithm 访问控制用的自旋锁、等待队列等，而 master_xfer()函数完成消息数组中消息的处理也可通过对 xxx_i2c 结构体相关成员的访问来控制。代码清单 15.12 所示为 xxx_i2c 结构体的定义，与图 15.2 中的 xxx_i2c 是对应的。

代码清单 15.12　xxx_i2c 结构体模板

```
1  struct xxx_i2c
2  {
3   spinlock_t      lock;
4   wait_queue_head_t   wait;
5   struct i2c_msg  *msg;
6   unsigned int        msg_num;
7   unsigned int        msg_idx;
8   unsigned int        msg_ptr;
9   ...
10  struct i2c_adapter  adap;
11 };
```

15.4 Linux I²C 设备驱动

I²C 设备驱动要使用 i2c_driver 和 i2c_client 数据结构并填充其中的成员函数。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 15.13 所示为已被初始化的 i2c_driver。

代码清单 15.13　已被初始化的 i2c_driver

```
1  static struct i2c_driver yyy_driver =
2  {
3    .driver =
4    {
5      .name = "yyy",
6    } ,
7    .attach_adapter = yyy_attach_adapter,
8    .detach_client =  yyy_detach_client,
9    .command = yyy_command,
10 };
```

15.4.1 Linux I²C 设备驱动的模块加载与卸载

I²C 设备驱动的模块加载函数通用的方法是在 I²C 设备驱动的模块加载函数中完成两件事。

（1）通过 register_chrdev()函数将 I²C 设备注册为一个字符设备。

（2）通过 I²C 核心的 i2c_add_driver()函数添加 i2c_driver。

在模块卸载函数中需要做相反的两件事。

（1）通过 I²C 核心的 i2c_del_driver()函数删除 i2c_driver。

（2）通过 unregister_chrdev()函数注销字符设备。

代码清单 15.14 所示为 I²C 设备驱动的加载与卸载函数模板。

代码清单 15.14 I²C 设备驱动的模块加载与卸载函数模板

```
1  static int __init yyy_init(void)
2  {
3    int res;
4    /*注册字符设备*/
5    res = register_chrdev(YYY_MAJOR, "yyy", &yyy_fops); //老内核接口
6    if (res)
7      goto out;
8    /*添加 i2c_driver*/
9    res = i2c_add_driver(&yyy_driver);
10   if (res)
11     goto out_unreg_class;
12   return 0;
13
14   out_unreg_chrdev: unregister_chrdev(I2C_MAJOR, "i2c");
15   out: printk(KERN_ERR "%s: Driver Initialisation failed\n", __FILE_
_);
16   return res;
17 }
18
19 static void __exit yyy_exit(void)
20 {
21   i2c_del_driver(&i2cdev_driver);
22   unregister_chrdev(YYY_MAJOR, "yyy");
23 }
```

第 5 行代码说明注册“yyy”这个字符设备时，使用的 file_operations 结构体为 yyy_fops，15.4.3 小节将讲解这个结构体中成员函数的实现。

15.4.2 Linux I²C 设备驱动的 i2c_driver 成员函数

i2c_add_driver（&yyy_driver）的执行会引发 i2c_driver 结构体中 yyy_attach_adapter()函数的执行，我们可以在 yyy_attach_adapter()函数里探测物理设备。为了实现探测，yyy_attach_adapter()函数里面也只需简单地调用 I²C 核心的 i2c_probe()函数，如代码清单 15.15 所示。

代码清单 15.15 I²C 设备驱动的 i2c_attach_adapter 函数模板

```
1 static int yyy_attach_adapter(struct i2c_adapter *adapter)
```

```
2 {
3   return i2c_probe(adapter, &addr_data, yyy_detect);
4 }
```

代码第 3 行传递给 i2c_probe()函数的第 1 个参数是 i2c_adapter 指针，第 2 个参数是要探测的地址数据，第 3 个参数是具体的探测函数。要探测的地址实际列表在一个 16 位无符号整型数组中，这个数组以 I2C_CLIENT_END 为最后一个元素。

i2c_probe()函数会引发 yyy_detect()函数的调用，可以在 yyy_detect()函数中初始化 i2c_client，如代码清单 15.16 所示。

代码清单 15.16　I^2C 设备驱动的 detect 函数模板

```
1  static int yyy_detect(struct i2c_adapter *adapter, int address, int
kind)
2  {
3   struct i2c_client *new_client;
4   struct yyy_data *data;
5   int err = 0;
6
7   if (!i2c_check_functionality(adapter, I2C_FUNC_XXX)
8      goto exit;
9
10  if (!(data = kzalloc(sizeof(struct yyy_data), GFP_KERNEL)))
11  {
12     err =  - ENOMEM;
13     goto exit;
14  }
15
16  new_client = &data->client;
17  new_client->addr = address;
18  new_client->adapter = adapter;
19  new_client->driver = &yyy_driver;
20  new_client->flags = 0;
21
22  /* 新的 client 将依附于 adapter */
23  if ((err = i2c_attach_client(new_client)))
24     goto exit_kfree;
25
26  yyy_init_client(new_client);
27  return 0;
28  exit_kfree: kfree(data);
29  exit: return err;
30}
```

代码第 10 行分配私有信息结构体的内存，i2c_client 也被创建。第 16～20 行对新创建的 i2c_client 进行初始化。第 23 行调用内核的 i2c_attach_client()知会 I^2C 核心系统中包含了一个新的 I^2C 设备。第 26 行代码初始化 i2c_client 对应的 I^2C 设备，这个函数是硬件相关的。

如图 15.4 所示为当 I^2C 设备驱动的模块加载函数被调用的时候引发的连锁反应的

流程。

I^2C 设备驱动的卸载函数进行 i2c_del_driver（&yyy_driver）调用后，会引发与 yyy_driver 关联的每个 i2c_client 与之解除关联，即 yyy_detach_client()函数将因此而被调用，代码清单 15.17 所示为函数 yyy_detach_client()的设计。

代码清单 15.17 I^2C 设备驱动的 i2c_detach_client 函数模板

```
1  static int yyy_detach_client(struct i2c_client *client)
2  {
3   int err;
4   struct yyy_data *data = i2c_get_clientdata(client);
5
6   if ((err = i2c_detach_client(client)))
7       return err;
8
9   kfree(data);
10  return 0;
11 }
```

上述函数中第 4 行的 i2c_get_clientdata()函数用于从 yyy_data 私有信息结构中的 i2c_client 的指针获取 yyy_data 的指针。第 6 行调用 I^2C 核心函数 i2c_detach_client()，这个函数会引发 i2c_adapter 的 client_ unregister()函数被调用。第 9 行代码释放 yyy_data 的内存。

图 15.5 所示为当 I^2C 设备驱动的模块卸载函数被调用的时候引发的连锁反应的流程。

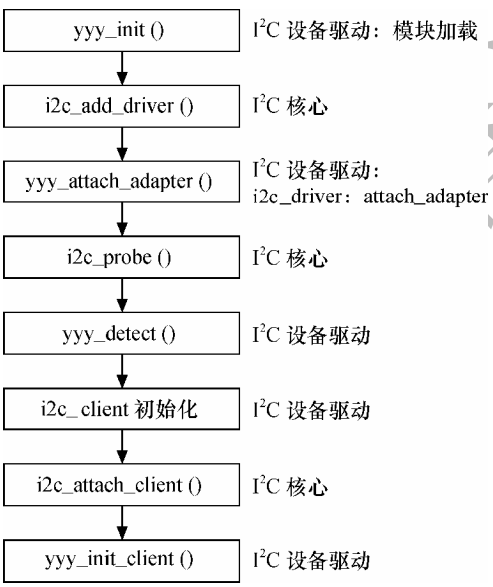

图 15.4 I^2C 设备驱动的模块加载连锁反应

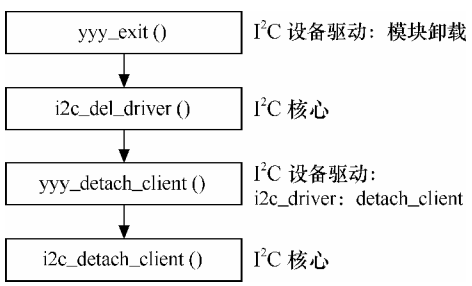

图 15.5 I^2C 设备驱动的模块卸载连锁反应

下面讲解 i2c_driver 中重要函数 yyy_command()的实现，它实现了针对设备的控制命令。具体的控制命令是设备相关的，如对于实时钟而言，命令将是设置时间和获取时间，而对于视频 AD 设备而言，命令会是设置采样方式、选择通道等。

假设 yyy 设备接受两类命令 YYY_CMD1 和 YYY_CMD2，而处理这两个命令的函数分别为 yyy_cmd1()和 yyy_cmd2()，代码清单 15.18 所示为 yyy_command()函数的

设计。

代码清单 15.18 I^2C 设备驱动的 command 函数模板

```
1  static int yyy_command(struct i2c_client *client, unsigned int cmd,
void
2    *arg)
3  {
4    switch (cmd)
5    {
6      case YYY_CMD1:
7        return yyy_cmd1(client, arg);
8      case YYY_CMD2:
9        return yyy_cmd2(client, arg);
10     default:
11       return  - EINVAL;
12   }
13 }
```

具体命令的实现是通过组件 i2c_msg 消息数组，并调用 I^2C 核心的传输、发送和接收函数，由 I^2C 核心的传输、发送和接收函数调用 I^2C 适配器对应的 algorithm 相关函数来完成的。代码清单 15.19 所示为一个 yyy_cmd1()的例子。

代码清单 15.19 I^2C 设备具体命令处理函数模板

```
1  static int yyy_cmd1(struct i2c_client *client, struct rtc_time *dt)
2  {
3   struct i2c_msg msg[2];
4   /*第一条消息是写消息*/
5   msg[0].addr = client->addr;
6   msg[0].flags = 0;
7   msg[0].len = 1;
8   msg[0].buf = &offs;
9   /*第二条消息是读消息*/
10  msg[1].addr = client->addr;
11  msg[1].flags = I2C_M_RD;
12  msg[1].len = sizeof(buf);
13  msg[1].buf = &buf[0];
14
15  i2c_transfer(client->adapter, msg, 2);
16   ...
17 }
```

15.4.3 Linux I²C 设备驱动的文件操作接口

作为一种字符类设备，Linux I²C 设备驱动的文件操作接口与普通的设备驱动是完全一致的，但是在其中要使用 i2c_client、i2c_driver、i2c_adapter 和 i2c_algorithm 结构体和 I²C 核心，并且对设备的读写和控制需要借助体系结构中各组成部分的协同合作。代码清单 15.20 所示为一个 I²C 设备写函数的例子。

代码清单 15.20 I²C 设备文件的接口写函数范例

```
1  static ssize_t yyy_write(struct file *file, char *buf, size_t count,
loff_t off)
2  {
3                  struct     i2c_client     *client     =     (struct
i2c_client*)file->private_data;
4    i2c_msg msg[1];
5    char *tmp;
6    int ret;
7
8    tmp = kmalloc(count, GFP_KERNEL);
9    if (tmp == NULL)
10     return  - ENOMEM;
11   if (copy_from_user(tmp, buf, count))
12   {
13     kfree(tmp);
14     return  - EFAULT;
15   }
16
17   msg[0].addr = client->addr;//地址
18   msg[0].flags = 0;          //0 为写
19   msg[0].len = count;        //要写的字节数
20   msg[0].buf = tmp;          //要写的数据
21   ret = i2c_transfer(client->adapter, msg, 1);  //传输 I²C 消息
22   return (ret == 1) ? count : ret;
23 }
```

上述程序给出的仅仅是一个写函数的例子，具体的写操作与设备密切相关。下面详细讲解 I²C 设备读写过程中数据的流向和函数的调用关系。I²C 设备的写操作经历了如下几个步骤。

（1）从用户空间到字符设备驱动写函数接口，写函数构造 I²C 消息数组。

（2）写函数把构造的 I²C 消息数组传递给 I²C 核心的传输函数 i2c_transfer()。

（3）I²C 核心的传输函数 i2c_transfer()找到对应适配器 algorithm 的通信方法函数 master_xfer()去最终完成 I²C 消息的处理。

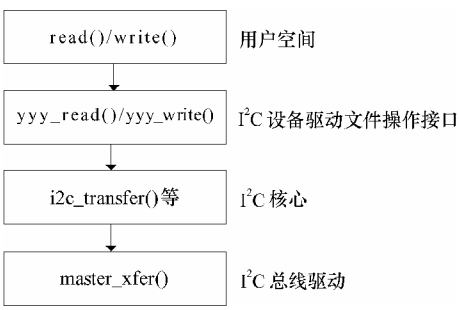

图 15.6 I²C 设备读写完整流程

如图 15.6 所示为从用户空间发起读写操作到 algorithm 进行消息传输的流程。

通常，如果 I²C 设备不是一个输入/输出设备或存储设备，就并不需要给 I²C 设备提供读写函数。许多 I²C 设备只是需要被设置

以某种方式工作，而不是被读写。另外，I^2C 设备驱动的文件操作接口也不是必需的，甚至极少被需要。大多数情况下，我们只需要通过 i2c-dev.c 文件提供的 I^2C 适配器设备文件接口就可完成对 I^2C 设备的读写。

15.4.4 Linux 的 i2c-dev.c 文件分析

i2c-dev.c 文件完全可以被看作一个 I^2C 设备驱动，其结构与 15.4.1～15.4.3 小节的描述是基本一致的，不过，它实现的一个 i2c_client 是虚拟、临时的，随着设备文件的打开而产生，并随设备文件的关闭而撤销，并没有被添加到 i2c_adapter 的 clients 链表中。i2c-dev.c 针对每个 I^2C 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，所以 i2c-dev.c 的主体是“i2c_driver 成员函数 + 字符设备驱动”。

i2c-dev.c 中提供 i2cdev_read()、i2cdev_write()函数来对应用户空间要使用的 read()和 write()文件操作接口，这两个函数分别调用 I^2C 核心的 i2c_master_recv()和 i2c_master_send()函数来构造一条 I^2C 消息并引发适配器 algorithm 通信函数的调用，完成消息的传输，对应于如图 15.7 所示的时序。但是，很遗憾，大多数稍微复杂一点 I^2C 设备的读写流程并不对应于一条消息，往往需要两条甚至更多的消息来进行一次读写周期（即如图 15.8 所示的重复开始位 RepStart 模式），这种情况下，在应用层仍然调用 read()、write()文件 API 来读写 I^2C 设备，将不能正确地读写。许多工程师碰到过类似的问题，往往经过相当长时间的调试都没法解决 I^2C 设备的读写，连错误的原因也无法找到，显然是对 i2cdev_read()和 i2cdev_write()函数的作用有所误解。

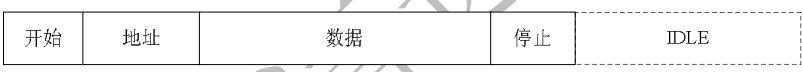

图 15.7　i2cdev_read 和 i2cdev_write 函数对应时序

图 15.8　RepStart 模式

鉴于上述原因，i2c-dev.c 中 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 15.21 所示为 i2cdev_ioctl()函数的框架，其中详细列出了 I2C_RDWR 命令的处理过程。

代码清单 15.21　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       ...   /*设置从设备地址*/
    10  case I2C_TENBIT:
    11      ...
    12  case I2C_PEC:
    13      ...
    14  case I2C_FUNCS:
    15      ...
    16  case I2C_RDWR:
    17    if (copy_from_user(&rdwr_arg,
    18              (struct i2c_rdwr_ioctl_data __user *)arg,
    19              sizeof(rdwr_arg)))
    20      return -EFAULT;
    21    /* 一次传入的消息太多 */
    22    if (rdwr_arg.nmsgs > I2C_RDWR_IOCTL_MAX_MSGS)
    23      return -EINVAL;
    24    /*获得用户空间传入的消息数组
    25    rdwr_pa = (struct i2c_msg *)
    26      kmalloc(rdwr_arg.nmsgs * sizeof(struct i2c_msg),
    27      GFP_KERNEL);
    28    if (rdwr_pa == NULL) return -ENOMEM;
    29    if (copy_from_user(rdwr_pa, rdwr_arg.msgs,
    30              rdwr_arg.nmsgs * sizeof(struct i2c_msg))) {
    31      kfree(rdwr_pa);
    32      return -EFAULT;
    33    }
    34    data_ptrs = kmalloc(rdwr_arg.nmsgs * sizeof(u8 __user *),
GFP_KERNEL);
    35    if (data_ptrs == NULL) {
    36      kfree(rdwr_pa);
    37      return -ENOMEM;
    38    }
    39    res = 0;
    40    for( i=0; i<rdwr_arg.nmsgs; i++ ) {
    41      /* 限制消息的长度 */
    42      if (rdwr_pa[i].len > 8192) {
    43          res = -EINVAL;
    44          break;
```

```
45      }
46      data_ptrs[i] = (u8 __user *)rdwr_pa[i].buf;
47      rdwr_pa[i].buf = kmalloc(rdwr_pa[i].len, GFP_KERNEL);
48      if(rdwr_pa[i].buf == NULL) {
49          res = -ENOMEM;
50          break;
51      }
52      if(copy_from_user(rdwr_pa[i].buf,
53          data_ptrs[i],
54          rdwr_pa[i].len)) {
55              ++i; /* Needs to be kfreed too */
56              res = -EFAULT;
57          break;
58      }
59    }
60    if (res < 0) {
61      int j;
62      for (j = 0; j < i; ++j)
63          kfree(rdwr_pa[j].buf);
64      kfree(data_ptrs);
65      kfree(rdwr_pa);
66      return res;
67    }
68    /*把这些消息交给通信方法去处理*/
69    res = i2c_transfer(client->adapter,
70      rdwr_pa,
71      rdwr_arg.nmsgs);
72    while(i-- > 0) {   /*如果是读消息，把值复制到用户空间*/
73      if( res>=0 && (rdwr_pa[i].flags & I2C_M_RD)) {
74          if(copy_to_user(
75              data_ptrs[i],
76              rdwr_pa[i].buf,
77              rdwr_pa[i].len)) {
78              res = -EFAULT;
79          }
80      }
81      kfree(rdwr_pa[i].buf);
82    }
83    kfree(data_ptrs);
84    kfree(rdwr_pa);
```

```
85    return res;
86  case I2C_SMBUS:
87      ...
88  default:
89      return i2c_control(client,cmd,arg);
90  }
91  return 0;
92 }
```

常用的 IOCTL 包括 I2C_SLAVE（设置从设备地址）、I2C_RETRIES（没有收到设备 ACK 情况下的重试次数，默认为 1）、I2C_TIMEOU（超时）以及 I2C_RDWR。

由第 17～19 行可以看出，应用程序需要通过 i2c_rdwr_ioctl_data 结构体来给内核传递 I^2C 消息，这个结构体定义如代码清单 15.22 所示，i2c_msg 数组指针及消息数量就被包含在 i2c_rdwr_ ioctl_data 结构体中。

代码清单 15.22　i2c_rdwr_ioctl_data 结构体

```
1 struct i2c_rdwr_ioctl_data {
2   struct i2c_msg __user *msgs;   /* I²C 消息指针 */
3   __u32 nmsgs;                   /* I²C 消息数量 */
4 };
```

代码清单 15.23 和代码清单 15.24 所示为直接通过 read()、write()接口和 O_RDWR IOCTL 读写 I^2C 设备的例子。

代码清单 15.23　直接通过 read()/write()读写 I^2C 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
9  #define I2C_RETRIES    0x0701
10 #define I2C_TIMEOUT    0x0702
11 #define I2C_SLAVE      0x0703
12
13 int main(int argc, char **argv)
14 {
15   unsigned int fd;
16   unsigned short mem_addr;
```

```
17  unsigned short size;
18  unsigned short idx;
19  #define BUFF_SIZE   32
20  char buf[BUFF_SIZE];
21  char cswap;
22  union
23  {
24    unsigned short addr;
25    char bytes[2];
26  } tmp;
27
28  if (argc < 3)
29  {
30    printf("Use:\n%s /dev/i2c-x mem_addr size\n", argv[0]);
31    return 0;
32  }
33  sscanf(argv[2], "%d", &mem_addr);
34  sscanf(argv[3], "%d", &size);
35
36  if (size > BUFF_SIZE)
37    size = BUFF_SIZE;
38
39  fd = open(argv[1], O_RDWR);
40
41  if (!fd)
42  {
43    printf("Error on opening the device file\n");
44    return 0;
45  }
46
47  ioctl(fd, I2C_SLAVE, 0x50); /* 设置 E2PROM 地址 */
48  ioctl(fd, I2C_TIMEOUT, 1); /* 设置超时 */
49  ioctl(fd, I2C_RETRIES, 1); /* 设置重试次数 */
50
51  for (idx = 0; idx < size; ++idx, ++mem_addr)
52  {
53    tmp.addr = mem_addr;
54    cswap = tmp.bytes[0];
55    tmp.bytes[0] = tmp.bytes[1];
56    tmp.bytes[1] = cswap;
57    write(fd, &tmp.addr, 2);
58    read(fd, &buf[idx], 1);
```

```
59   }
60   buf[size] = 0;
61   close(fd);
62   printf("Read %d char: %s\n", size, buf);
63   return 0;
64 }
```

代码清单 15.24　通过 O_RDWR IOCTL 读写 I^2C 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
12 #define MAX_I2C_MSG        2
13
14 #define I2C_RETRIES     0x0701
15 #define I2C_TIMEOUT     0x0702
16 #define I2C_RDWR       0x0707
17
18 struct i2c_msg
19 {
20   __u16 addr; /* 从地址 */
21   __u16 flags;
22   #define I2C_M_RD       0x01
23   __u8 *buf; /* 消息数据指针 */
24 };
25 struct i2c_rdwr_ioctl_data
26 {
27   struct i2c_msg *msgs; /* i2c_msg[]指针 */
28   int nmsgs; /* i2c_msg 数量 */
29 };
30
31 int main(int argc, char **argv)
32 {
33   struct i2c_rdwr_ioctl_data work_queue;
34   unsigned int idx;
35   unsigned int fd;
36   unsigned short start_address;
37   int ret;
38
39   if (argc < 4)
40   {
```

```
    41    printf("Usage:\n%s /dev/i2c-x start_addr\n", argv[0]);
    42    return 0;
    43  }
    44
    45  fd = open(argv[1], O_RDWR);
    46
    47  if (!fd)
    48  {
    49    printf("Error on opening the device file\n");
    50    return 0;
    51  }
    52  sscanf(argv[2], "%x", &start_address);
    53  work_queue.nmsgs = MAX_I2C_MSG; /* 消息数量 */
    54
    55     work_queue.msgs  =  (struct  i2c_msg*)malloc(work_queue.nmsgs
*sizeof(struct
    56    i2c_msg));
    57  if (!work_queue.msgs)
    58  {
    59    printf("Memory alloc error\n");
    60    close(fd);
    61    return 0;
    62  }
    63
    64  for (idx = 0; idx < work_queue.nmsgs; ++idx)
    65  {
    66    (work_queue.msgs[idx]).len = 0;
    67    (work_queue.msgs[idx]).addr = start_address + idx;
    68    (work_queue.msgs[idx]).buf = NULL;
    69  }
    70
    71  ioctl(fd, I2C_TIMEOUT, 2); /* 设置超时 */
    72  ioctl(fd, I2C_RETRIES, 1); /* 设置重试次数 */
    73
    74  ret = ioctl(fd, I2C_RDWR, (unsigned long) &work_queue);
    75
    76  if (ret < 0)
    77  {
    78    printf("Error during I2C_RDWR ioctl with error code: %d\n", ret);
    79  }
    80
    81  close(fd);
    82  return ;
    83 }
```

15.5 S3C2410 I²C 总线驱动实例

15.5.1 S3C2410 I²C 控制器硬件描述

S3C2410 处理器内部集成了一个 I²C 控制器，通过 4 个寄存器就可方便地对其进行控制，这 4 个寄存器如下。

- IICCON：I²C 控制寄存器。
- IICSTAT：I²C 状态寄存器。
- IICDS：I²C 收发数据移位寄存器。
- IICADD：I²C 地址寄存器。

S3C2410 处理器内部集成的 I²C 控制器可支持主、从两种模式，我们主要使用其主模式。通过对 IICCON、IICDS 和 IICADD 寄存器的操作，可在 I²C 总线上产生开始位、停止位、数据和地址，而传输的状态则通过 IICSTAT 寄存器获取。

15.5.2 S3C2410 I²C 总线驱动总体分析

S3C2410 的 I²C 总线驱动设计主要要完成以下工作。

- 设计对应于 i2c_adapter_xxx_init()模板的 S3C2410 的模块加载函数和对应于 i2c_ adapter_xxx_exit()函数模板的模块卸载函数。
- 设计对应于 i2c_adapter_xxx_xfer()模板的 S3C2410 适配器的通信方法函数。针对 S3C2410，functionality()函数只需简单地返回 I2C_FUNC_I2C | I2C_FUNC_SMBUS_ EMUL | I2C_FUNC_PROTOCOL_MANGLING 表明其支持的功能。

内核源代码中的/drivers/i2c/busses/i2c-s3c2410.c 文件包含了由 Ben Dooks<ben@simtec.co.uk>编写的 S3C2410 的 I²C 总线驱动，在 Internet 上我们还可以获得其他版本的 S3C2410 I²C 总线驱动源代码，如 Steve Hein<ssh@sgi.com>为三星 SMDK2410 参考板编写的 i2c-s3c2410.c、i2c-algo- s3c2410.c 文件，对比发现，它与 Ben Dooks 版本的驱动实现风格迥异。

这里分析内核中自带的 Ben Dooks 版本驱动，它同时支持 S3C2410 和 S3C2440。图 15.9 给出了 S3C2410 驱动中的主要函数与 15.3 节模板函数的对应关系，由于实现通信方法的方式不一样，模板的一个函数可能对应于 S3C2410 I²C 总线驱动的多个函数。

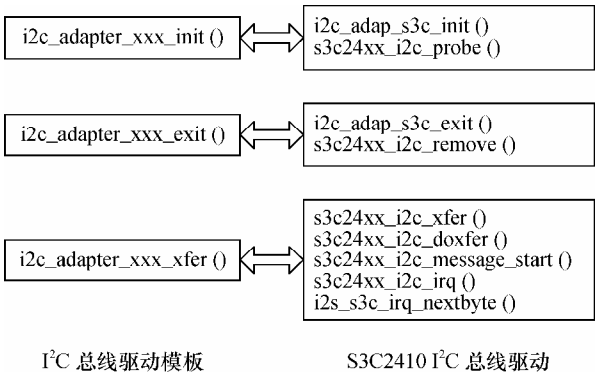

图 15.9　I²C 总线驱动模板与 S3C2410 I²C 总线驱动的映射

15.5.3　S3C2410 I²C 适配器驱动的模块加载与卸载

I²C 适配器驱动被作为一个单独的模块加载进内核，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 15.25 所示。

代码清单 15.25　S3C2410 I²C 总线驱动的模块加载与卸载

```
1  static int __init i2c_adap_s3c_init(void)
2  {
3   int ret;
4
5   ret = platform_driver_register(&s3c2410_i2c_driver);
6   if (ret == 0) {
7       ret = platform_driver_register(&s3c2440_i2c_driver);
8       if (ret)
9           platform_driver_unregister(&s3c2410_i2c_driver);
10  }
11
12  return ret;
13 }
14
15 static void __exit i2c_adap_s3c_exit(void)
16 {
17  platform_driver_unregister(&s3c2410_i2c_driver);
18  platform_driver_unregister(&s3c2440_i2c_driver);
19 }
20 module_init(i2c_adap_s3c_init);
21 module_exit(i2c_adap_s3c_exit);
```

platform_driver 结构体包含了具体适配器的 probe()函数、remove()函数、resume()函数指针等信息，它需要被定义和赋值，如代码清单 15.26 所示。

代码清单 15.26　platform_driver 结构体

```
1 static struct platform_driver s3c2410_i2c_driver = {
2   .probe      = s3c24xx_i2c_probe,
3   .remove     = s3c24xx_i2c_remove,
4   .resume     = s3c24xx_i2c_resume,
5   .driver     = {
6       .owner  = THIS_MODULE,
```

```
    7       .name   = "s3c2410-i2c",
    8   },
    9 };
```

当通过 Linux 内核源代码/drivers/base/platform.c 文件中定义 platform_driver_unregister()函数注册 platform_driver 结构体时，其中 probe 指针指向的 s3c24xx_i2c_probe()函数将被调用，以初始化适配器硬件，如代码清单 15.27 所示。

代码清单 15.27 S3C2410 I²C 总线驱动中的 s3c24xx_i2c_probe 函数

```
    1  static int s3c24xx_i2c_probe(struct platform_device *pdev)
    2  {
    3   struct s3c24xx_i2c *i2c = &s3c24xx_i2c;
    4   struct resource *res;
    5   int ret;
    6
    7   /* 使能 I2C 的时钟 */
    8   i2c->dev = &pdev->dev;
    9   i2c->clk = clk_get(&pdev->dev, "i2c");
   10   if (IS_ERR(i2c->clk)) {
   11       dev_err(&pdev->dev, "cannot get clock\n");
   12       ret = -ENOENT;
   13       goto out;
   14   }
   15   clk_enable(i2c->clk);
   16
   17   /* 映射寄存器 */
   18   res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
   19   if (res == NULL) {
   20       dev_err(&pdev->dev, "cannot find IO resource\n");
   21       ret = -ENOENT;
   22       goto out;
   23   }
   24   i2c->ioarea           =           request_mem_region(res->start,
(res->end-res->start)+1,
   25                   pdev->name);
   26   if (i2c->ioarea == NULL) {
   27       dev_err(&pdev->dev, "cannot request IO\n");
   28       ret = -ENXIO;
   29       goto out;
   30   }
   31
   32   i2c->regs = ioremap(res->start, (res->end-res->start)+1);
   33   if (i2c->regs == NULL) {
   34       dev_err(&pdev->dev, "cannot map IO\n");
   35       ret = -ENXIO;
   36       goto out;
   37   }
   38
   39   /* 设置 I2C 的信息块 */
   40   i2c->adap.algo_data = i2c;
   41   i2c->adap.dev.parent = &pdev->dev;
```

```
42
43  /* 初始化 I2C 控制器 */
44  ret = s3c24xx_i2c_init(i2c);
45  if (ret != 0)
46      goto out;
47
48  /* 申请中断 */
49  res = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
50  if (res == NULL) {
51      ret = -ENOENT;
52      goto out;
53  }
54  ret = request_irq(res->start, s3c24xx_i2c_irq, SA_INTERRUPT,
55            pdev->name, i2c);
56  if (ret != 0) {
57      goto out;
58  }
59  i2c->irq = res;
60
61  ret = i2c_add_adapter(&i2c->adap); /*添加 i2c_adapter*/
62  if (ret < 0) {
63      goto out;
64  }
65
66  platform_set_drvdata(pdev, i2c);
67
68  out:
69  if (ret < 0)
70      s3c24xx_i2c_free(i2c);
71  return ret;
72 }
```

上述代码中的主体工作是使能硬件并申请 I²C 适配器使用的 I/O 地址、中断号等，在这些工作都完成无误后，通过 I²C 核心提供的 i2c_add_adapter()函数添加这个适配器。因为 S3C2410 内部集成 I²C 控制器，可以确定 I²C 适配器一定存在，s3c24xx_i2c_probe()函数虽然命名“探测”，但实际没有也不必进行任何探测工作，之所以这样命名完全是一种设计习惯。

与 s3c24xx_i2c_probe()函数完成相反功能的函数是 s3c24xx_i2c_remove()函数，它在适配器模块卸载函数调用 platform_driver_unregister()函数时所示通过 platform_driver 的 remove 指针方式被调用。xxx_i2c_remove()的设计模板如代码清单 15.28 所示。

代码清单 15.28　S3C2410 I²C 总线驱动中的 s3c24xx_i2c_remove 函数

```
1  static int s3c24xx_i2c_remove(struct platform_device *pdev)
2  {
3   struct s3c24xx_i2c *i2c = platform_get_drvdata(pdev);
```

```
4
5   if (i2c != NULL) {
6       s3c24xx_i2c_free(i2c);
7       platform_set_drvdata(pdev, NULL);
8   }
9
10  return 0;
11 }
```

代码清单 15.27 和代码清单 15.28 中用到的 s3c24xx_i2c 结构体进行适配器所有信息的封装，类似于私有信息结构体，它与代码清单 15.12 所示的 xxx_i2c 结构体模板对应。代码清单 15.29 所示的 s3c24xx_i2c 结构体的定义，以及驱动模块定义的一个 s3c24xx_i2c 结构体全局实例。

代码清单 15.29　s3c24xx_i2c 结构体

```
1  struct s3c24xx_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   enum s3c24xx_i2c_state  state;
9   void __iomem                *regs;
10  struct clk              *clk;
11  struct device               *dev;
12  struct resource         *irq;
13  struct resource         *ioarea;
14  struct i2c_adapter      adap;
15 };
16
17 static struct s3c24xx_i2c s3c24xx_i2c = {
18  .lock   = SPIN_LOCK_UNLOCKED,  /*自旋锁未锁定*/
19  .wait   = __WAIT_QUEUE_HEAD_INITIALIZER(s3c24xx_i2c.wait),/*等待
队列初始化*/
20  .adap   = {
21      .name           = "s3c2410-i2c",
22      .owner          = THIS_MODULE,
23      .algo           = &s3c24xx_i2c_algorithm,
24      .retries            = 2,
25      .class          = I2C_CLASS_HWMON,
```

```
26 },
27 };
```

15.5.4 S3C2410 I²C 总线通信方法

由代码清单 15.29 的第 23 行可以看出，I²C 适配器对应的 i2c_algorithm 结构体实例为 s3c24xx_i2c_algorithm，代码清单 15.30 所示 s3c24xx_i2c_algorithm 的定义。

代码清单 15.30 S3C2410 的 i2c_algorithm 结构体

```
1 static struct i2c_algorithm s3c24xx_i2c_algorithm = {
2   .master_xfer            = s3c24xx_i2c_xfer,
3   .functionality      = s3c24xx_i2c_func,
4 };
```

上述代码第 1 行指定了 S3C2410 I²C 总线通信传输函数 s3c24xx_i2c_xfer()，这个函数非常关键，所有 I²C 总线上对设备的访问最终应该由它来完成，代码清单 15.31 所示为这个重要函数以及其依赖的 s3c24xx_i2c_doxfer() 函数和 s3c24xx_i2c_message_start()函数的源代码。

代码清单 15.31 S3C2410 I²C 总线驱动的 master_xfer 函数

```
1  static int s3c24xx_i2c_xfer(struct i2c_adapter *adap,
2           struct i2c_msg *msgs, int num)
3  {
4   struct s3c24xx_i2c *i2c = (struct s3c24xx_i2c *)adap->algo_data;
5   int retry;
6   int ret;
7
8   for (retry = 0; retry < adap->retries; retry++) {
9       ret = s3c24xx_i2c_doxfer(i2c, msgs, num);
10      if (ret != -EAGAIN)
11          return ret;
12      udelay(100);
13  }
14
15  return -EREMOTEIO;
16 }
17
18  static  int  s3c24xx_i2c_doxfer(struct  s3c24xx_i2c  *i2c,  struct
i2c_msg *msgs, int num)
19 {
20  unsigned long timeout;
21  int ret;
22
23  ret = s3c24xx_i2c_set_master(i2c);
```

```
24  if (ret != 0) {
25      ret = -EAGAIN;
26      goto out;
27  }
28
29  spin_lock_irq(&i2c->lock);
30  i2c->msg    = msgs;
31  i2c->msg_num = num;
32  i2c->msg_ptr = 0;
33  i2c->msg_idx = 0;
34  i2c->state   = STATE_START;
35  s3c24xx_i2c_enable_irq(i2c);
36  s3c24xx_i2c_message_start(i2c, msgs);
37  spin_unlock_irq(&i2c->lock);
38
39  timeout = wait_event_timeout(i2c->wait, i2c->msg_num == 0, HZ * 5);
40
41  ret = i2c->msg_idx;
42
43  if (timeout == 0)
44      dev_dbg(i2c->dev, "timeout\n");
45  else if (ret != num)
46      dev_dbg(i2c->dev, "incomplete xfer (%d)\n", ret);
47
48  msleep(1);/* 确保停止位已经被传递 */
49
50  out:
51  return ret;
52 }
53
54 static void s3c24xx_i2c_message_start(struct s3c24xx_i2c *i2c,
55                   struct i2c_msg *msg)
56 {
57  unsigned int addr = (msg->addr & 0x7f) << 1;
58  unsigned long stat;
59  unsigned long iiccon;
60
61  stat = 0;
```

```
62  stat |=  S3C2410_IICSTAT_TXRXEN;
63
64  if (msg->flags & I2C_M_RD) {
65      stat |= S3C2410_IICSTAT_MASTER_RX;
66      addr |= 1;
67  } else
68      stat |= S3C2410_IICSTAT_MASTER_TX;
69  if (msg->flags & I2C_M_REV_DIR_ADDR)
70      addr ^= 1;
71
72  s3c24xx_i2c_enable_ack(i2c);/*如果要使能 ACK，则使能*/
73
74  iiccon = readl(i2c->regs + S3C2410_IICCON);
75  writel(stat, i2c->regs + S3C2410_IICSTAT);
76  writeb(addr, i2c->regs + S3C2410_IICDS);
77
78  udelay(1);/*在发送新的开始位前延迟 1 位*/
79  writel(iiccon, i2c->regs + S3C2410_IICCON);
80  stat |=  S3C2410_IICSTAT_START;
81  writel(stat, i2c->regs + S3C2410_IICSTAT);
82 }
```

s3c24xx_i2c_xfer()函数调用 s3c24xx_i2c_doxfer()函数传输 I²C 消息，第 8 行的循环意味着最多可以重试 adap->retries 次。

s3c24xx_i2c_doxfer()首先将 S3C2410 的 I²C 适配器设置为 I²C 主设备，其后初始化 s3c24xx_i2c 结构体，使能 I²C 中断，并调用 s3c24xx_i2c_message_start()函数启动 I²C 消息的传输。

s3c24xx_i2c_message_start()函数写 S3C2410 适配器对应的控制寄存器，向 I²C 从设备传递开始位和从设备地址。

上述代码只是启动了 I²C 消息数组的传输周期，并没有完整实现图 15.3 中给出的 algorithm master_xfer 流程。这个流程的完整实现需要借助 I²C 适配器上的中断来步步推进。代码清单 15.32 所示为 S3C2410 I²C 适配器中断处理函数以及其依赖的 i2s_s3c_irq_nextbyte()函数的源代码。

代码清单 15.32　S3C2410 I²C 适配器中断处理函数

```
1  static irqreturn_t s3c24xx_i2c_irq(int irqno, void *dev_id,
2                  struct pt_regs *regs)
3  {
4      struct s3c24xx_i2c *i2c = dev_id;
5      unsigned long status;
6      unsigned long tmp;
7
8      status = readl(i2c->regs + S3C2410_IICSTAT);
9      if (status & S3C2410_IICSTAT_ARBITR) {
10         ...
11     }
12
```

```
    if (i2c->state == STATE_IDLE) {
        tmp = readl(i2c->regs + S3C2410_IICCON);
        tmp &= ~S3C2410_IICCON_IRQPEND;
        writel(tmp, i2c->regs +  S3C2410_IICCON);
        goto out;
    }

    i2s_s3c_irq_nextbyte(i2c, status);/* 把传输工作进一步推进 */

     out:
        return IRQ_HANDLED;
    }

       static  int  i2s_s3c_irq_nextbyte(struct  s3c24xx_i2c  *i2c,
unsigned long iicstat)
    {
    unsigned long tmp;
    unsigned char byte;
    int ret = 0;

    switch (i2c->state) {
    case STATE_IDLE:
        goto out;
        break;
    case STATE_STOP:
        s3c24xx_i2c_disable_irq(i2c);
        goto out_ack;
    case STATE_START:
        /* 我们最近做的一件事是启动一个新 I²C 消息*/
        if (iicstat  & S3C2410_IICSTAT_LASTBIT &&
            !(i2c->msg->flags & I2C_M_IGNORE_NAK)) {
            /* 没有收到 ACK */
            s3c24xx_i2c_stop(i2c, -EREMOTEIO);
            goto out_ack;
        }

        if (i2c->msg->flags & I2C_M_RD)
            i2c->state = STATE_READ;
        else
```

```
51              i2c->state = STATE_WRITE;
52
53          /* 仅一条消息，而且长度为 0（主要用于适配器探测），发送停止位*/
54          if (is_lastmsg(i2c) && i2c->msg->len == 0) {
55              s3c24xx_i2c_stop(i2c, 0);
56              goto out_ack;
57          }
58
59          if (i2c->state == STATE_READ)
60              goto prepare_read;
61          /* 进入写状态 */
62      case STATE_WRITE:
63      retry_write:
64          if (!is_msgend(i2c)) {
65              byte = i2c->msg->buf[i2c->msg_ptr++];
66              writeb(byte, i2c->regs + S3C2410_IICDS);
67
68          } else if (!is_lastmsg(i2c)) {
69              /* 推进到下一条消息 */
70              i2c->msg_ptr = 0;
71              i2c->msg_idx ++;
72              i2c->msg++;
73
74              /* 检查是否要为该消息产生开始位 */
75              if (i2c->msg->flags & I2C_M_NOSTART) {
76                  if (i2c->msg->flags & I2C_M_RD) {
77                      s3c24xx_i2c_stop(i2c, -EINVAL);
78                  }
79                  goto retry_write;
80              } else {
81                  /* 发送新的开始位 */
82                  s3c24xx_i2c_message_start(i2c, i2c->msg);
83                  i2c->state = STATE_START;
84              }
85          } else {
86              s3c24xx_i2c_stop(i2c, 0);/* send stop */
87          }
88          break;
89          case STATE_READ:
90          /* 有一个字节可读，看是否还有消息要处理    */
```

```
    91          if (!(i2c->msg->flags & I2C_M_IGNORE_NAK) &&
    92              !(is_msglast(i2c) && is_lastmsg(i2c))) {
    93
    94              if (iicstat & S3C2410_IICSTAT_LASTBIT) {
    95                  dev_dbg(i2c->dev, "READ: No Ack\n");
    96
    97                  s3c24xx_i2c_stop(i2c, -ECONNREFUSED);
    98                  goto out_ack;
    99              }
    100     }
    101     byte = readb(i2c->regs + S3C2410_IICDS);
    102     i2c->msg->buf[i2c->msg_ptr++] = byte;
    103
    104 prepare_read:
    105     if (is_msglast(i2c)) {/* last byte of buffer */
    106         if (is_lastmsg(i2c))
    107             s3c24xx_i2c_disable_ack(i2c);
    108
    109     } else if (is_msgend(i2c)) {
    110         /* 还有消息要处理吗? */
    111         if (is_lastmsg(i2c)) {
    112             s3c24xx_i2c_stop(i2c, 0);/* last message, send stop and
complete */
    113         } else {
    114             /* 推进到下一条消息 */
    115             i2c->msg_ptr = 0;
    116             i2c->msg_idx++;
    117             i2c->msg++;
    118         }
    119     }
    120     break;
    121 }
    122
    123 /* irq 清除 */
    124     out_ack:
    125 tmp = readl(i2c->regs + S3C2410_IICCON);
    126 tmp &= ~S3C2410_IICCON_IRQPEND;
    127 writel(tmp, i2c->regs + S3C2410_IICCON);
```

```
128  out:
129 return ret;
130 }
```

中断处理函数 s3c24xx_i2c_irq()主要通过调用 i2s_s3c_irq_nextbyte()函数进行传输工作的进一步推进。i2s_s3c_irq_nextbyte()函数通过 switch(i2c->state)语句分成 i2c->state 的不同状态进行处理，在每种状态下，先检查 i2c->state 的状态与硬件寄存器应该处于的状态是否一致，如果不一致，则证明有误，直接返回。当 I²C 处于读状态 STATE_READ 或写状态 STATE_WRITE 时，通过 is_lastmsg()函数判断是否传输的是最后一条 I²C 消息，如果是，则产生停止位，否则通过 i2c->msg_idx++、i2c->msg++推进到下一条消息。

15.6 SAA7113H 视频 AD 芯片的 I²C 设备驱动实例

15.6.1 SAA7113H 视频 AD 芯片硬件描述

如图 15.10 所示，SAA7113H 是飞利浦半导体推出的 9 位视频 AD 芯片，它可以选择 4 路视频输入中的 1 路，并采样为 9 位的数字信号。

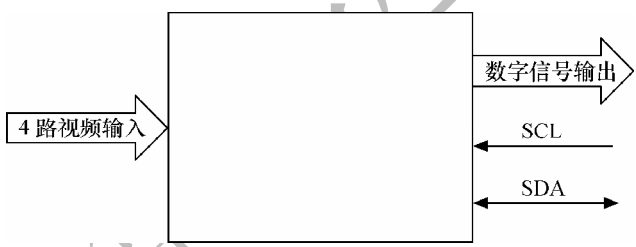

图 15.10　SAA7113H 输入、输出与 I²C 接口

对 SAA7113H 输入通道的选择以及采样方式的设置都需通过其 I²C 接口进行，以 0x4A 地址可读 SAA7113H 寄存器，以 0x4B 可写 SAA7113H 寄存器。SAA7113H 的 I²C 接口连接在 S3C2410 的 I²C 控制器上，作为 S3C2410 I²C 控制器的从设备。SAA7113H 的写时序与图 15.7 给出的写时序是一致的，但是读时序则需通过两条 I²C 消息解决，在第 1 条消息中应该给 SAA7113H 写入寄存器子地址，如图 15.11 所示。

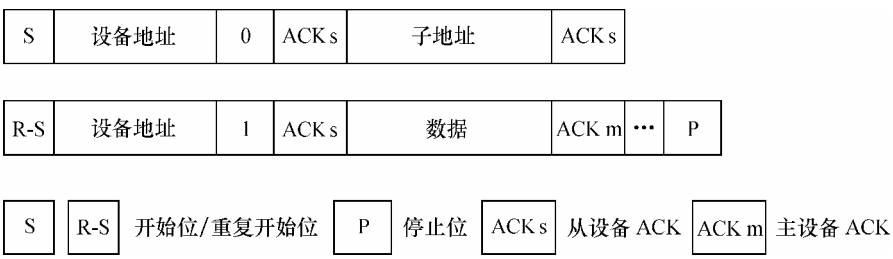

图 15.11　SAA7113H 读时序

15.6.2 SAA7113H 视频 AD 芯片驱动的模块加载与卸载

由于不需要给 SAA7113H 实现文件操作接口，在设备驱动模块的加载和卸载函数中均不再需要注册和注销字符设备的操作，代码清单 15.33 所示为 SAA7113H 设备驱动的模块加载和卸载的源代码。

代码清单 15.33 SAA7113H 设备驱动的模块加载和卸载函数

```
1  static int __init saa711x_init (void)
2  {
3    return i2c_add_driver(&i2c_driver_saa711x); //注册 i2c_driver
4  }
5
6  static void __exit saa711x_exit (void)
7  {
8    i2c_del_driver(&i2c_driver_saa711x); //注销 i2c_driver
9  }
10
11 module_init(saa711x_init);
12 module_exit(saa711x_exit);
```

15.6.3 SAA7113H 设备驱动的 i2c_driver 成员函数

如代码清单 15.34 所示，SAA7113H 设备驱动模块加载和卸载中添加和删除的 i2c_driver_ saa711x 结构体的 attach_adapter 指针被赋值为 saa711x_attach_adapter，detach_client 指针被赋值为 saa711x_detach_client，而 command 指针被赋值为 saa711x_command，代码清单 15.35 所示为这 3 个函数的实现。

代码清单 15.34 SAA7113H 设备驱动 i2c_driver 结构体

```
1 static struct i2c_driver i2c_driver_saa711x = {
2   .driver = {
3   .name = "saa711x",
4  },
5   .id = I2C_DRIVERID_SAA711X,
6   /* 成员函数 */
7   .attach_adapter = saa711x_attach_adapter,
8   .detach_client = saa711x_detach_client,
9   .command = saa711x_command,
10 };
```

代码清单 15.35 SAA7113H 设备驱动 i2c_driver 成员函数

```
1  saa711x_attach_adapter (struct i2c_adapter *adapter)
2  {
3   dprintk(1, KERN_INFO "saa711x.c: starting probe for adapter %s
(0x%x)\n",
4       I2C_NAME(adapter), adapter->id);
```

```
    5
    6   return i2c_probe(adapter, &addr_data, &saa711x_detect_client);
    7    //saa711x_detect_client   会   调   用   i2c_set_clientdata()   、
i2c_attach_client()
    8  }
    9
    10 static int saa711x_detach_client (struct i2c_client *client)
    11 {
    12  struct saa711x *decoder = i2c_get_clientdata(client);
    13  int err;
    14
    15  err = i2c_detach_client(client);
    16  if (err) {
    17      return err;
    18  }
    19
    20  kfree(decoder);
    21  kfree(client);
    22
    23  return 0;
    24 }
    25
    26 static int saa711x_command(struct i2c_client *client, unsigned int
cmd, void
    27   *arg)
    28 {
    29   struct saa711x *decoder = i2c_get_clientdata(client);
    30
    31   switch (cmd)  //处理不同的命令
    32   {
    33     case 0:
    34     case DECODER_INIT:
    35       ...
    36     case DECODER_DUMP:
    37       ...
    38     case DECODER_GET_CAPABILITIES:
    39       ...
    40     case DECODER_GET_STATUS:
    41       ...
    42     case DECODER_SET_VBI_BYPASS:
    43       ...
    44     case DECODER_SET_NORM:
```

```
    45      ...
    46    case DECODER_SET_INPUT:
    47      {
    48        int *iarg = arg;
    49        if (*iarg < 0 ||  *iarg > 9)
    50        {
    51          return  - EINVAL;
    52        }
    53        if (decoder->input !=  *iarg)
    54        {
    55          decoder->input =  *iarg;
    56          saa711x_write(client, 0x02, (decoder->reg[0x02] &0xf0) |
decoder
    57            ->input);
    58          saa711x_write(client, 0x09, (decoder->reg[0x09] &0x7f) |
((decoder
    59            ->input > 3) ? 0x80 : 0));
    60        }
    61      }
    62      break;
    63
    64    case DECODER_SET_OUTPUT:
    65      ...
    66    case DECODER_ENABLE_OUTPUT:
    67      ...
    68    case DECODER_SET_PICTURE:
    69      ...
    70    default:
    71      return  - EINVAL;
    72  }
    73
    74  return 0;
    75 }
```

15.8 总结

Linux I^2C 驱动体系结构有相当的复杂度，它主要由 3 部分组成，即 I^2C 核心、I^2C 总线驱动和 I^2C 设备驱动。I^2C 核心是 I2C 总线驱动和 I^2C 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^2C 中设备与适配器的沟通。I^2C 总线驱动填充 i2c_adapter 和 i2c_algorithm 结构体，I^2C 设备驱动填充 i2c_driver 和 i2c_client 结构体。

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^2C 设备寄存器的能力，使得工程师大多数时候并不需要为具体的 I^2C 设备驱动定义文件操作接口。

最后，工程师在设计 I^2C 设备驱动的程序，并不是一定要遵守 Linux I^2C 驱动的体系结构，完全可以把它当作一个普通的字符设备处理。