第四十章 Linux I2C 驱动实验

对于 I2C 我相信大家都很熟悉，基本上做过单片机开发的朋友都接触过，在电子产品硬件设计当中，I2C 是一种很常见的同步、串行、低速、近距离通信接口，用于连接各种 IC、传感器等器件，它们都会提供 I2C 接口与 SoC 主控相连，比如陀螺仪、加速度计、触摸屏等，其最大优势在于可以在总线上扩展多个外围设备的支持。

Linux 内核开发者为了让驱动开发工程师在内核中方便的添加自己的 I2C 设备驱动程序，更容易的在 linux 下驱动自己的 I2C 接口硬件，进而引入了 I2C 总线框架。与 Linux 下的 platform 虚拟总线不同的是，I2C 是实际的物理总线，所以 I2C 总线框架也是 Linux 下总线、设备、驱动模型的产物。

本章我们来学习一下如何在 Linux 下的 I2C 总线框架，以及如何使用 I2C 总线框架编写一个 I2C 接口的外设驱动程序；本章重点是学习 Linux 下的 I2C 总线框架。

40.1 I2C & AP3216C 简介

40.1.1 I2C 简介

I2C 是很常见的一种总线协议，I2C 是 NXP 公司设计的，I2C 使用两条线在主控制器和从机之间进行数据通信。一条是 SCL(串行时钟线)，另外一条是 SDA(串行数据线)，这两条数据线需要接上拉电阻，总线空闲的时候 SCL 和 SDA 处于高电平。I2C 总线标准模式下速度可以达到 100Kb/S，快速模式下可以达到 400Kb/S。I2C 总线工作是按照一定的协议来运行的，接下来就看一下 I2C 协议。

I2C 是支持多从机的，也就是一个 I2C 控制器下可以挂多个 I2C 从设备，这些不同的 I2C 从设备有不同的器件地址，这样 I2C 主控制器就可以通过 I2C 设备的器件地址访问指定的 I2C 设备了，一个 I2C 总线连接多个 I2C 设备如图 40.1.1.1 所示：

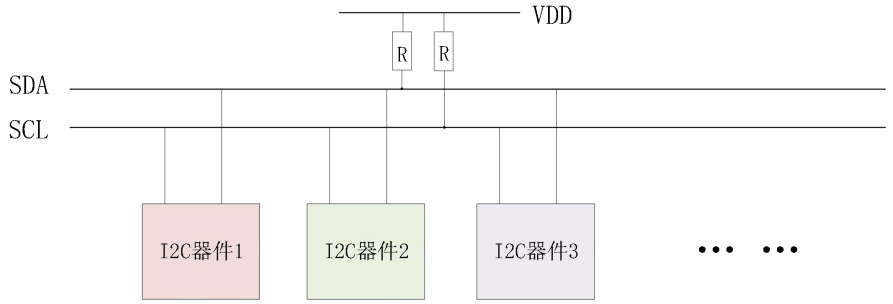

图 40.1.1.1 I2C 多个设备连接结构图

图 40.1.1.1 中 SDA 和 SCL 这两根线必须要接一个上拉电阻，一般是 4.7K。其余的 I2C 从器件都挂接到 SDA 和 SCL 这两根线上，这样就可以通过 SDA 和 SCL 这两根线来访问多个 I2C 设备。

接下来看一下 I2C 协议有关的术语：

1、起始位

顾名思义，也就是 I2C 通信起始标志，通过这个起始位就可以告诉 I2C 从机，“我”要开始进行 I2C 通信了。在 SCL 为高电平的时候，SDA 出现下降沿就表示为起始位，如图 40.1.1.2 所示：

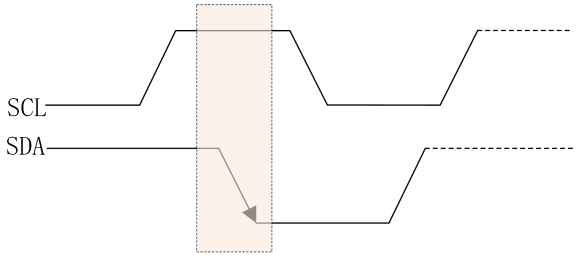

图 40.1.1.2 I2C 通信起始位

2、停止位

停止位就是停止 I2C 通信的标志位，和起始位的功能相反。在 SCL 位高电平的时候，SDA 出现上升沿就表示为停止位，如图 40.1.1.3 所示：

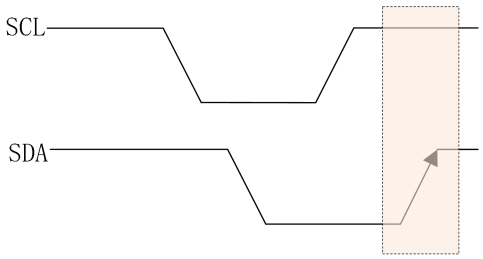

图 40.1.1.3 I2C 通信停止位

3、数据传输

I2C 总线在数据传输的时候要保证在 SCL 高电平期间，SDA 上的数据稳定，因此 SDA 上的数据变化只能在 SCL 低电平期间发生，如图 40.1.1.4 所示：

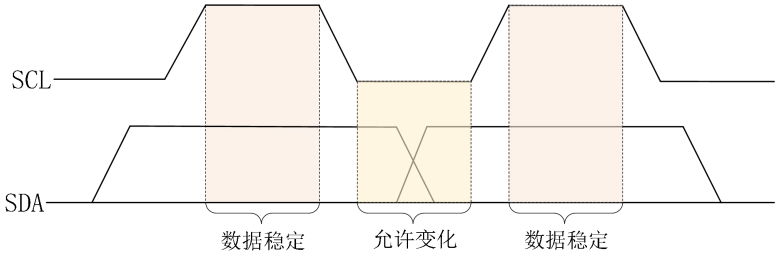

图 40.1.1.4 I2C 数据传输

4、应答信号

当 I2C 主机发送完 8 位数据以后会将 SDA 设置为输入状态，等待 I2C 从机应答，也就是等待 I2C 从机告诉主机它接收到数据了。应答信号是由从机发出的，主机需要提供应答信号所需的时钟，主机发送完 8 位数据以后紧跟着的一个时钟信号就是给应答信号使用的。从机通过将 SDA 拉低来表示发出应答信号，表示通信成功，否则表示通信失败。

5、I2C 写时序

主机通过 I2C 总线与从机之间进行通信不外乎两个操作：写和读，I2C 总线单字节写时序如图 40.1.1.5 所示：

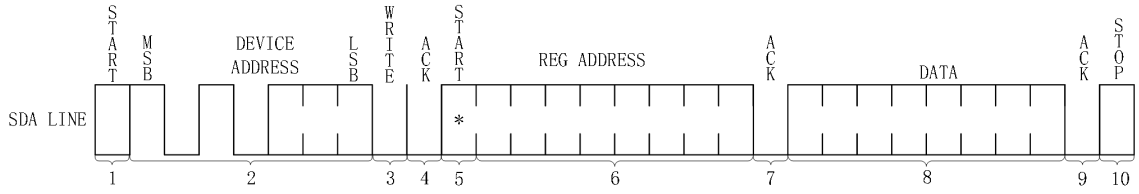

图 40.1.1.5 I2C 写时序

图 40.1.1.5 就是 I2C 写时序，我们来看一下写时序的具体步骤：

1)、开始信号。

2)、发送 I2C 设备地址，每个 I2C 器件都有一个设备地址，通过发送具体的设备地址来决定访问哪个 I2C 器件。这是一个 8 位的数据，其中高 7 位是设备地址，最后 1 位是读写位，为 1 的话表示这是一个读操作，为 0 的话表示这是一个写操作。

3)、 I2C 器件地址后面跟着一个读写位，为 0 表示写操作，为 1 表示读操作。

4)、从机发送的 ACK 应答信号。

5)、重新发送开始信号。

6)、发送要写写入数据的寄存器地址。

7)、从机发送的 ACK 应答信号。

8)、发送要写入寄存器的数据。

9)、从机发送的 ACK 应答信号。

10)、停止信号。

6、I2C 读时序

I2C 总线单字节读时序如图 40.1.1.6 所示：

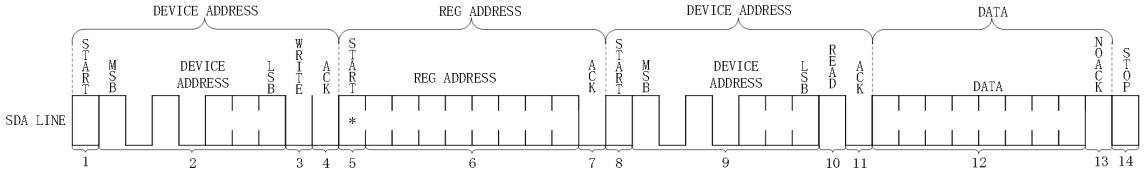

图 40.1.1.6 I2C 单字节读时序

I2C 单字节读时序比写时序要复杂一点，读时序分为 4 大步，第一步是发送设备地址，第二步是发送要读取的寄存器地址，第三步重新发送设备地址，最后一步就是 I2C 从器件输出要读取的寄存器值，我们具体来看一下这步。

1)、主机发送起始信号。

2)、主机发送要读取的 I2C 从设备地址。

3)、读写控制位，因为是向 I2C 从设备发送数据，因此是写信号。

4)、从机发送的 ACK 应答信号。

5)、重新发送 START 信号。

6)、主机发送要读取的寄存器地址。

7)、从机发送的 ACK 应答信号。

8)、重新发送 START 信号。

9)、重新发送要读取的 I2C 从设备地址。

10)、读写控制位，这里是读信号，表示接下来是从 I2C 从设备里面读取数据。

11)、从机发送的 ACK 应答信号。

12)、从 I2C 器件里面读取到的数据。

13)、主机发出 NO ACK 信号，表示读取完成，不需要从机再发送 ACK 信号了。

14)、主机发出 STOP 信号，停止 I2C 通信。

7、I2C 多字节读写时序

有时候我们需要读写多个字节，多字节读写时序和单字节的基本一致，只是在读写数据的时候可以连续发送多个自己的数据，其他的控制时序都是和单字节一样的。

40.1.2 STM32MP1 I2C 简介

STM32MP157D 有 6 个 I2C 接口，其中 I2C4 和 I2C6 可以在 A7 安全模式或者 A7 非安全模式下使用，M4 无法使用，STM32MP157 的 I2C 部分特性如下：

①、兼容 I2C 总线规范第 03 版。

②、支持从模式和主模式，支持多主模式功能。

③、支持标准模式 (Sm)、快速模式 (Fm) 和超快速模式 (Fm+)，其中，标准模式 100kHz，快速模式 400 kHz，超快速模式可以到 1 MHz。

④、7 位和 10 位寻址模式。

⑤、多个 7 位从地址，所有 7 位地址应答模式。

⑥、软件复位。

⑦、带 DMA 功能的 1 字节缓冲。

⑧、广播呼叫。

关于 STM32M157 IIC 更多详细的介绍，请参考《STM32MP157 参考手册》相关章节。

40.1.3 AP3216C 简介

STM32MP1 开发板上通过 I2C5 连接了一个三合一环境传感器：AP3216C，AP3216C 是由敦南科技推出的一款传感器，其支持环境光强度(ALS)、接近距离(PS)和红外线强度(IR)这三个环境参数检测。该芯片可以通过 IIC 接口与主控制相连，并且支持中断，AP3216C 的特点如下：

①、I2C 接口，快速模式下波特率可以到 400Kbit/S

②、多种工作模式选择：ALS、PS+IR、ALS+PS+IR、PD 等等。

③、内建温度补偿电路。

④、宽工作温度范围(-30°C～+80°C)。

⑤、超小封装，4.1mm x 2.4mm x 1.35mm

⑥、环境光传感器具有 16 位分辨率。

⑦、接近传感器和红外传感器具有 10 位分辨率。

AP3216C 常被用于手机、平板、导航设备等，其内置的接近传感器可以用于检测是否有物体接近，比如手机上用来检测耳朵是否接触听筒，如果检测到的话就表示正在打电话，手机就会关闭手机屏幕以省电。也可以使用环境光传感器检测光照强度，可以实现自动背光亮度调节。

AP3216C 结构如图 40.1.3.1 所示：

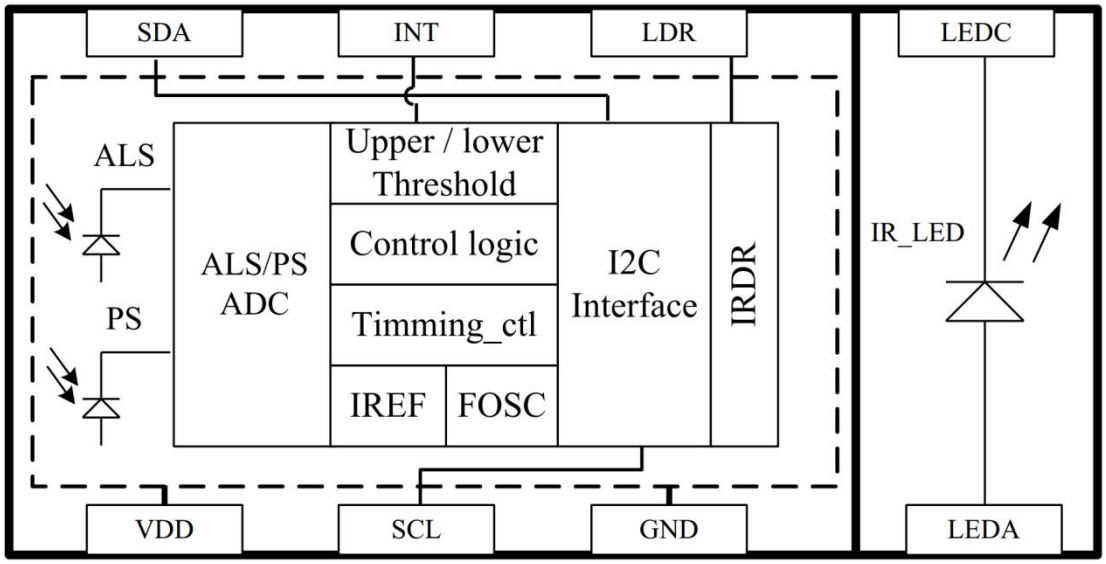

图 40.1.3.1 AP3216C 结构图

AP3216 的设备地址为 0X1E，同几乎所有的 I2C 从器件一样，AP3216C 内部也有一些寄存器，通过这些寄存器我们可以配置 AP3216C 的工作模式，并且读取相应的数据。AP3216C 我们用的寄存器如表 40.1.3.1 所示：

寄存器地址	位	寄存器功能	描述
0X00	2:0	系统模式	000：掉电模式(默认)。 001：使能 ALS。 010：使能 PS+IR。 011：使能 ALS+PS+IR。 100：软复位。 101：ALS 单次模式。 110：PS+IR 单次模式。

			111：ALS+PS+IR 单次模式。
0X0A	7	IR 低位数据	0：IR&PS 数据有效，1:无效
	1:0		IR 最低 2 位数据。
0X0B	7:0	IR 高位数据	IR 高 8 位数据。
0X0C	7:0	ALS 低位数据	ALS 低 8 位数据。
0X0D	7:0	ALS 高位数据	ALS 高 8 位数据。
0X0E	7	PS 低位数据	0，物体在远离；1，物体在接近。
	6		0，IR&PS 数据有效；1，IR&PS 数据无效
	3:0		PS 最低 4 位数据。
0X0F	7	PS 高位数据	0，物体在远离；1，物体在接近。
	6		0，IR&PS 数据有效；1，IR&PS 数据无效
	5:0		PS 最低 6 位数据。

表 40.1.3.1 本章使用的 AP3216C 寄存器表

在表 40.1.3.1 中，0X00 这个寄存器是模式控制寄存器，用来设置 AP3216C 的工作模式，一般开始先将其设置为 0X04，也就是先软件复位一次 AP3216C。接下来根据实际使用情况选择合适的工作模式，比如设置为 0X03，也就是开启 ALS+PS+IR。0X0A~0X0F 这 6 个寄存器就是数据寄存器，保存着 ALS、PS 和 IR 这三个传感器获取到的数据值。如果同时打开 ALS、PS 和 IR 的读取间隔最少要 112.5ms，因为 AP3216C 完成一次转换需要 112.5ms。关于 AP3216C 的介绍就到这里，如果要想详细的研究此芯片的话，请大家自行查阅其数据手册。

40.2 Linux I2C 总线框架简介

使用裸机的方式编写一个 I2C 器件的驱动程序，我们一般需要实现两部分：

①、I2C 主机驱动。

②、I2C 设备驱动。

I2C 主机驱动也就是 SoC 的 I2C 控制器对应的驱动程序，I2C 设备驱动其实就是挂在 I2C 总线下的具体设备对应的驱动程序，例如 eeprom、触摸屏 IC、传感器 IC 等；对于主机驱动来说，一旦编写完成就不需要再做修改，其他的 I2C 设备直接调用主机驱动提供的 API 函数完成读写操作即可。这个正好符合 Linux 的驱动分离与分层的思想，因此 Linux 内核也将
I2C 驱动分为两部分。

Linux 内核开发者为了让驱动开发工程师在内核中方便的添加自己的 I2C 设备驱动程序，方便大家更容易的在 linux 下驱动自己的 I2C 接口硬件，进而引入了 I2C 总线框架，我们一般也叫作 I2C 子系统，Linux 下 I2C 子系统总体框架如下所示：

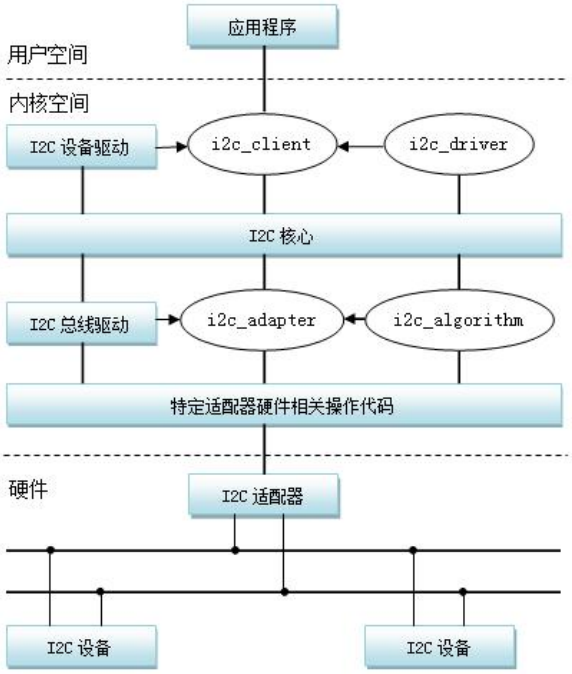

图 40.2.3.1 I2C 子系统框架图

从图 40.2.3.1 可以知道，I2C 子系统分为三大组成部分：

1、I2C 核心(I2C-core)

I2C 核心提供了 I2C 总线驱动（适配器）和设备驱动的注册、注销方法，I2C 通信方法(algorithm)与具体硬件无关的代码，以及探测设备地址的上层代码等；

2、I2C 总线驱动(I2C adapter)

I2C 总线驱动是 I2C 适配器的软件实现，提供 I2C 适配器与从设备间完成数据通信的能力。I2C 总线驱动由 i2c_adapter 和 i2c_algorithm 来描述。I2C 适配器是 SoC 中内置 i2c 控制器的软件抽象，可以理解为他所代表的是一个 I2C 主机；

3、I2C 设备驱动(I2C client driver)

包括两部分：设备的注册和驱动的注册。

I2C 子系统帮助内核统一管理 I2C 设备，让驱动开发工程师在内核中可以更加容易地添加自己的 I2C 设备驱动程序。

40.2.1 I2C 总线驱动

首先来看一下 I2C 总线，在讲 platform 的时候就说过，platform 是虚拟出来的一条总线，目的是为了实现总线、设备、驱动框架。对于 I2C 而言，不需要虚拟出一条总线，直接使用 I2C 总线即可。I2C 总线驱动重点是 I2C 适配器(也就是 SoC 的 I2C 接口控制器)驱动，这里要用到两个重要的数据结构：i2c_adapter 和 i2c_algorithm，I2C 子系统将 SoC 的 I2C 适配器(控制器)抽象成一个 i2c_adapter 结构体，i2c_adapter 结构体定义在 include/linux/i2c.h 文件中，结构体内容如下：

示例代码 40.2.1 i2c_adapter 结构体

```
685 struct i2c_adapter {
686     struct module *owner;
687     unsigned int class;
688     const struct i2c_algorithm *algo;
```

```
689     void *algo_data;
690
691     /* data fields that are valid for all devices   */
692     const struct i2c_lock_operations *lock_ops;
693     struct rt_mutex bus_lock;
694     struct rt_mutex mux_lock;
695
696     int timeout;                /* in jiffies */
697     int retries;
698     struct device dev;          /* the adapter device */
699     unsigned long locked_flags; /* owned by the I2C core */
700 #define I2C_ALF_IS_SUSPENDED          0
701 #define I2C_ALF_SUSPEND_REPORTED      1
702
703     int nr;
704     char name[48];
705     struct completion dev_released;
706
707     struct mutex userspace_clients_lock;
708     struct list_head userspace_clients;
709
710     struct i2c_bus_recovery_info *bus_recovery_info;
711     const struct i2c_adapter_quirks *quirks;
712
713     struct irq_domain *host_notify_domain;
714 };
```

第 688 行，i2c_algorithm 类型的指针变量 algo，对于一个 I2C 适配器，肯定要对外提供读写 API 函数，设备驱动程序可以使用这些 API 函数来完成读写操作。i2c_algorithm 就是 I2C 适配器与 IIC 设备进行通信的方法。

i2c_algorithm 结构体定义在 include/linux/i2c.h 文件中，内容如下：

示例代码 40.2.1.2 i2c_algorithm 结构体

```
526 struct i2c_algorithm {
527     /*
528      * If an adapter algorithm can't do I2C-level access, set
529      * master_xfer to NULL. If an adapter algorithm can do SMBus
530      * access, set smbus_xfer. If set to NULL, the SMBus protocol is
531      * simulated using common I2C messages.
532      *
533      * master_xfer should return the number of messages successfully
534      * processed, or a negative value on error
535      */
536     int (*master_xfer)(struct i2c_adapter *adap,
                           struct i2c_msg *msgs,
537                int num);
```

```
538     int (*master_xfer_atomic)(struct i2c_adapter *adap,
539                               struct i2c_msg *msgs, int num);
540     int (*smbus_xfer)(struct i2c_adapter *adap, u16 addr,
541             unsigned short flags, char read_write,
542             u8 command, int size, union i2c_smbus_data *data);
543     int (*smbus_xfer_atomic)(struct i2c_adapter *adap, u16 addr,
544                unsigned short flags, char read_write,
545                u8 command, int size, union i2c_smbus_data *data);
546
547     /* To determine what the adapter supports */
548     u32 (*functionality)(struct i2c_adapter *adap);
549
550 #if IS_ENABLED(CONFIG_I2C_SLAVE)
551     int (*reg_slave)(struct i2c_client *client);
552     int (*unreg_slave)(struct i2c_client *client);
553 #endif
554 };
```

第 536 行，master_xfer 就是 I2C 适配器的传输函数，可以通过此函数来完成与 IIC 设备之间的通信。

第 540 行，smbus_xfer 就是 SMBUS 总线的传输函数。smbus 协议是从 I2C 协议的基础上发展而来的，他们之间有很大的相似度，SMBus 与 I2C 总线之间在时序特性上存在一些差别，应用于移动 PC 和桌面 PC 系统中的低速率通讯。

综上所述，I2C 总线驱动，或者说 I2C 适配器驱动的主要工作就是初始化 i2c_adapter 结构体变量，然后设置 i2c_algorithm 中的 master_xfer 函数。完成以后通过 i2c_add_numbered_adapter 或 i2c_add_adapter 这两个函数向 I2C 子系统注册设置好的 i2c_adapter，这两个函数的原型如下：

```
int i2c_add_adapter(struct i2c_adapter *adapter)
int i2c_add_numbered_adapter(struct i2c_adapter *adap)
```

这两个函数的区别在于 i2c_add_adapter 会动态分配一个总线编号，而 i2c_add_numbered_adapter 函数则指定一个静态的总线编号。函数参数和返回值含义如下：

adapter 或 adap：要添加到 Linux 内核中的 i2c_adapter，也就是 I2C 适配器。

返回值：0，成功；负值，失败。

如果要删除 I2C 适配器的话使用 i2c_del_adapter 函数即可，函数原型如下：

```
void i2c_del_adapter(struct i2c_adapter * adap)
```

函数参数和返回值含义如下：

adap：要删除的 I2C 适配器。

返回值：无。

关于 I2C 的总线(控制器或适配器)驱动就讲解到这里，一般 SoC 的 I2C 总线驱动都是由半导体厂商编写的，比如 STM32MP1 的 I2C 适配器驱动 ST 官方已经编写好了，这个不需要用户去编写。因此 I2C 总线驱动对我们这些 SoC 使用者来说是被屏蔽掉的，我们只要专注于 I2C 设备驱动即可，除非你是在半导体公司上班，工作内容就是写 I2C 适配器驱动。

40.2.2 I2C 总线设备

I2C 设备驱动重点关注两个数据结构：i2c_client 和 i2c_driver，根据总线、设备和驱动

模型，I2C 总线上一小节已经讲了。还剩下设备和驱动，i2c_client 用于描述 I2C 总线下的设备，i2c_driver 则用于描述 I2C 总线下的设备驱动，类似于 platform 总线下的 platform_device 和 platform_driver。

1、i2c_client 结构体

i2c_client 结构体定义在 include/linux/i2c.h 文件中，内容如下：

示例代码 40.2.2.1 i2c_client 结构体

```
313    struct i2c_client {
314        unsigned short flags;           /* div., see below      */
......
328        struct i2c_adapter *adapter;    /* the adapter we sit on */
329        struct device dev;              /* the device structure  */
330        int init_irq;           /* irq set at initialization    */
331        int irq;                /* irq issued by device         */
332        struct list_head detected;
333    #if IS_ENABLED(CONFIG_I2C_SLAVE)
334        i2c_slave_cb_t slave_cb;    /* callback for slave mode   */
335    #endif
336    };
```

一个 I2C 设备对应一个 i2c_client 结构体变量，系统每检测到一个 I2C 从设备就会给这个设备分配一个 i2c_client。

2、i2c_driver 结构体

i2c_driver 类似 platform_driver，是我们编写 I2C 设备驱动重点要处理的内容，i2c_driver 结构体定义在 include/linux/i2c.h 文件中，内容如下：

示例代码 40.2.2.2 i2c_driver 结构体

```
253    struct i2c_driver {
254        unsigned int class;
255
256        /* Standard driver model interfaces */
257        int (*probe)(struct i2c_client *client,
                        const struct i2c_device_id *id);
258        int (*remove)(struct i2c_client *client);
259
260        /* New driver model interface to aid the seamless removal of
261         * the current probe()'s, more commonly unused than used
262         second parameter.*/
263        int (*probe_new)(struct i2c_client *client);
264
265      /* driver model interfaces that don't relate to enumeration  */
266        void (*shutdown)(struct i2c_client *client);
267
268        /* Alert callback, for example for the SMBus alert protocol.
269         * The format and meaning of the data value depends on the
270         * protocol. For the SMBus alert protocol, there is a single
```

```
271         * bit of data passed as the alert response's low bit ("event
272         * flag"). For the SMBus Host Notify protocol, the data
273         * corresponds to the 16-bit payload data reported by the
274         slave device acting as master.*/
275        void (*alert)(struct i2c_client *client,
                         enum i2c_alert_protocol protocol,
276                       unsigned int data);
277
278        /* a ioctl like command that can be used to perform specific
279         * functions with the device.
280         */
281        int (*command)(struct i2c_client *client, unsigned int cmd,
                         void *arg);
282
283        struct device_driver driver;
284        const struct i2c_device_id *id_table;
285
286        /* Device detection callback for automatic device creation */
287        int (*detect)(struct i2c_client *client,
                         struct i2c_board_info *info);
288        const unsigned short *address_list;
289        struct list_head clients;
290
291        bool disable_i2c_core_irq_mapping;
292    };
```

第 257 行，当 I2C 设备和驱动匹配成功以后 probe 函数就会执行，和 platform 驱动一样。

第 283 行，device_driver 驱动结构体，如果使用设备树的话，需要设置 device_driver 的 of_match_table 成员变量，也就是驱动的兼容(compatible)属性。

第 284 行，id_table 是传统的、未使用设备树的设备匹配 ID 表。

对于我们 I2C 设备驱动编写人来说，重点工作就是构建 i2c_driver，构建完成以后需要向 I2C 子系统注册这个 i2c_driver。i2c_driver 注册函数为 int i2c_register_driver，此函数原型如下：

```
int i2c_register_driver(struct module      *owner,
                        struct i2c_driver  *driver)
```

函数参数和返回值含义如下：

owner：一般为 THIS_MODULE。

driver：要注册的 i2c_driver。

返回值：0，成功；负值，失败。

另外 i2c_add_driver 也常常用于注册 i2c_driver，i2c_add_driver 是一个宏，定义如下：

示例代码 40.2.2.3 i2c_add_driver 宏

```
844    #define i2c_add_driver(driver) \
845        i2c_register_driver(THIS_MODULE, driver)
```

i2c_add_driver 就是对 i2c_register_driver 做了一个简单的封装，只有一个参数，就是要注册的 i2c_driver。

注销 I2C 设备驱动的时候需要将前面注册的 i2c_driver 从 I2C 子系统中注销掉，需要用到 i2c_del_driver 函数，此函数原型如下：

```
void i2c_del_driver(struct i2c_driver    *driver)
```

函数参数和返回值含义如下：

driver：要注销的 i2c_driver。

返回值：无。

i2c_driver 的注册示例代码如下：

示例代码 40.2.2.4 i2c_driver 注册流程

```
1  /* i2c 驱动的 probe 函数 */
2  static int xxx_probe(struct i2c_client *client,
                        const struct i2c_device_id *id)
3  {
4      /* 函数具体程序 */
5      return 0;
6  }
7
8  /* i2c 驱动的 remove 函数 */
9  static int ap3216c_remove(struct i2c_client *client)
10 {
11     /* 函数具体程序 */
12     return 0;
13 }
14
15 /* 传统匹配方式 ID 列表 */
16 static const struct i2c_device_id xxx_id[] = {
17     {"xxx", 0},
18     {}
19 };
20
21 /* 设备树匹配列表 */
22 static const struct of_device_id xxx_of_match[] = {
23     { .compatible = "xxx" },
24     { /* Sentinel */ }
25 };
26
27 /* i2c 驱动结构体 */
28 static struct i2c_driver xxx_driver = {
29     .probe = xxx_probe,
30     .remove = xxx_remove,
31     .driver = {
32         .owner = THIS_MODULE,
33         .name = "xxx",
34         .of_match_table = xxx_of_match,
35         },
```

```
36          .id_table = xxx_id,
37      };
38
39  /* 驱动入口函数 */
40  static int __init xxx_init(void)
41  {
42      int ret = 0;
43
44      ret = i2c_add_driver(&xxx_driver);
45      return ret;
46  }
47
48  /* 驱动出口函数 */
49  static void __exit xxx_exit(void)
50  {
51      i2c_del_driver(&xxx_driver);
52  }
53
54  module_init(xxx_init);
55  module_exit(xxx_exit);
```

第 16~19 行，i2c_device_id，无设备树的时候匹配 ID 表。

第 22~25 行，of_device_id，设备树所使用的匹配表。

第 28~37 行，i2c_driver，当 I2C 设备和 I2C 驱动匹配成功以后 probe 函数就会执行，这些和 platform 驱动一样，probe 函数里面基本就是标准的字符设备驱动那一套了。

40.2.3 I2C 设备和驱动匹配过程

I2C 设备和驱动的匹配过程是由 I2C 子系统核心层来完成的，drivers/i2c/i2c-core-base.c 就是 I2C 的核心部分，I2C 核心提供了一些与具体硬件无关的 API 函数，比如前面讲过的：

1、i2c_adapter 注册/注销函数

```
int i2c_add_adapter(struct i2c_adapter *adapter)
int i2c_add_numbered_adapter(struct i2c_adapter *adap)
void i2c_del_adapter(struct i2c_adapter * adap)
```

2、i2c_driver 注册/注销函数

```
int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
int i2c_add_driver (struct i2c_driver *driver)
void i2c_del_driver(struct i2c_driver *driver)
```

设备和驱动的匹配过程也是由核心层完成的，I2C 总线的数据结构为 i2c_bus_type，定义在 drivers/i2c/i2c-core-base.c 文件，i2c_bus_type 内容如下：

示例代码 40.2.3.1 i2c_bus_type 结构体

```
492 struct bus_type i2c_bus_type = {
493     .name       = "i2c",
494     .match      = i2c_device_match,
495     .probe      = i2c_device_probe,
```

```
496     .remove    = i2c_device_remove,
497     .shutdown  = i2c_device_shutdown,
498 };
```

.match 就是 I2C 总线的设备和驱动匹配函数，在这里就是 i2c_device_match 这个函数，此函数内容如下：

示例代码 40.2.3.2 i2c_device_match 函数

```
93  static int i2c_device_match(struct device *dev,
                                struct device_driver *drv)
94  {
95      struct i2c_client   *client = i2c_verify_client(dev);
96      struct i2c_driver   *driver;
97
98
99      /* Attempt an OF style match */
100     if (i2c_of_match_device(drv->of_match_table, client))
101         return 1;
102
103     /* Then ACPI style match */
104     if (acpi_driver_match_device(dev, drv))
105         return 1;
106
107     driver = to_i2c_driver(drv);
108
109     /* Finally an I2C match */
110     if (i2c_match_id(driver->id_table, client))
111         return 1;
112
113     return 0;
114 }
```

第 100 行，i2c_of_match_device 函数用于完成设备树中定义的设备与驱动匹配过程。比较 I2C 设备节点的 compatible 属性和 of_device_id 中的 compatible 属性是否相等，如果相当的话就表示 I2C 设备和驱动匹配。

第 104 行，acpi_driver_match_device 函数用于 ACPI 形式的匹配。

第 110 行，i2c_match_id 函数用于传统的、无设备树的 I2C 设备和驱动匹配过程。比较 I2C 设备名字和 i2c_device_id 的 name 字段是否相等，相等的话就说明 I2C 设备和驱动匹配成功。

40.3 STM32MP1 I2C 适配器驱动分析

上一小节我们讲解了 Linux 下的 I2C 子系统，重点分为 I2C 适配器驱动和 I2C 设备驱动，其中 I2C 适配器驱动就是 SoC 的 I2C 控制器驱动。I2C 设备驱动是需要用户根据不同的 I2C 从设备去编写，而 I2C 适配器驱动一般都是 SoC 厂商去编写的，比如 ST 就已经提供了 STM3MP21 的 I2C 适配器驱动程序。在内核源码 arch/arm/boot/dts/stm32mp151.dtsi 设备树文件中找到 STM32MP1 的 I2C 控制器节点，节点内容如下所示：

示例代码 40.3.1 I2C1 控制器节点

```
590     i2c1: i2c@40012000 {
591         compatible = "st,stm32mp15-i2c";
592         reg = <0x40012000 0x400>;
593         interrupt-names = "event", "error";
594         interrupts-extended = <&exti 21 IRQ_TYPE_LEVEL_HIGH>,
595                               <&intc GIC_SPI 32 IRQ_TYPE_LEVEL_HIGH>;
596         clocks = <&rcc I2C1_K>;
597         resets = <&rcc I2C1_R>;
598         #address-cells = <1>;
599         #size-cells = <0>;
600         dmas = <&dmamux1 33 0x400 0x80000001>,
601                <&dmamux1 34 0x400 0x80000001>;
602         dma-names = "rx", "tx";
603         power-domains = <&pd_core>;
604         st,syscfg-fmp = <&syscfg 0x4 0x1>;
605         wakeup-source;
606         status = "disabled";
607     };
```

重点关注 i2c1 节点的 compatible 属性值，因为通过 compatible 属性值可以在 Linux 源码里面找到对应的驱动文件。这里 i2c1 节点的 compatible 属性值“st,stm32mp15-i2c”，在 Linux 源码中搜索这个字符串即可找到对应的驱动文件。STM32MP1 的 I2C 适配器驱动驱动文件为 drivers/i2c/busses/i2c-stm32f7.c，在此文件中有如下内容：

示例代码 40.3.2 i2c-stm32f7.c 文件代码段

```
2520    static const struct of_device_id stm32f7_i2c_match[] = {
2521     { .compatible = "st,stm32f7-i2c", .data = &stm32f7_setup},
2522     { .compatible = "st,stm32mp15-i2c", .data = &stm32mp15_setup},
2523     {},
2524    };
2525    MODULE_DEVICE_TABLE(of, stm32f7_i2c_match);
2526
2527    static struct platform_driver stm32f7_i2c_driver = {
2528        .driver = {
2529            .name = "stm32f7-i2c",
2530            .of_match_table = stm32f7_i2c_match,
2531            .pm = &stm32f7_i2c_pm_ops,
2532        },
2533        .probe = stm32f7_i2c_probe,
2534        .remove = stm32f7_i2c_remove,
2535    };
2536
2537    module_platform_driver(stm32f7_i2c_driver);
```

从示例代码 40.3.2 可以看出，STM32MP1 的 I2C 适配器驱动是个标准的 platform 驱动，由此可以看出，虽然 I2C 总线为别的设备提供了一种总线驱动框架，但是 I2C 适配器却是 platform 驱动。就像你的部门老大是你的领导，你是他的下属，但是放到整个公司，你的部

门老大却也是老板的下属。

第 2529 行，“st,stm32mp15-i2c”属性值，设备树中 i2c1 节点的 compatible 属性值就是与此匹配上的。因此 i2c-stm32f7.c 文件就是 STM32MP1 的 I2C 适配器驱动文件。

第 2533 行，当设备和驱动匹配成功以后 stm32f7_i2c_probe 函数就会执行，stm32f7_i2c_probe 函数就会完成 I2C 适配器初始化工作。

stm32f7_i2c_probe 函数内容如下所示(有省略)：

示例代码 40.3.3 stm32f7_i2c_probe 函数代码段

```
2106  static int stm32f7_i2c_probe(struct platform_device *pdev)
2107  {
2108      struct stm32f7_i2c_dev *i2c_dev;
2109      const struct stm32f7_i2c_setup *setup;
2110      struct resource *res;
2111      u32 rise_time, fall_time;
2112      struct i2c_adapter *adap;
2113      struct reset_control *rst;
2114      dma_addr_t phy_addr;
2115      int irq_error, ret;
2116
2117      i2c_dev = devm_kzalloc(&pdev->dev, sizeof(*i2c_dev),
                               GFP_KERNEL);
2118      if (!i2c_dev)
2119          return -ENOMEM;
2120
2121      res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
2122      i2c_dev->base = devm_ioremap_resource(&pdev->dev, res);
2123      if (IS_ERR(i2c_dev->base))
2124          return PTR_ERR(i2c_dev->base);
2125      phy_addr = (dma_addr_t)res->start;
2126
2127      i2c_dev->irq_event = platform_get_irq(pdev, 0);
2128      if (i2c_dev->irq_event <= 0) {
2129          if (i2c_dev->irq_event != -EPROBE_DEFER)
2130              dev_err(&pdev->dev, "Failed to get IRQ event: %d\n",
2131                      i2c_dev->irq_event);
2132          return i2c_dev->irq_event ? : -ENOENT;
2133      }
2134
2135      irq_error = platform_get_irq(pdev, 1);
2136      if (irq_error <= 0) {
2137          if (irq_error != -EPROBE_DEFER)
2138              dev_err(&pdev->dev, "Failed to get IRQ error: %d\n",
2139                      irq_error);
2140          return irq_error ? : -ENOENT;
2141      }
```

```
......
2159        ret = device_property_read_u32(&pdev->dev,
"clock-frequency",
2160                                       &i2c_dev->bus_rate);
2161        if (ret)
2162            i2c_dev->bus_rate = I2C_STD_RATE;
2163
2164        if (i2c_dev->bus_rate > I2C_FASTPLUS_RATE) {
2165            dev_err(&pdev->dev, "Invalid bus speed (%i>%i)\n",
2166                i2c_dev->bus_rate, I2C_FASTPLUS_RATE);
2167            return -EINVAL;
2168        }
......
2183
2184        ret = devm_request_threaded_irq(&pdev->dev,
                                            i2c_dev->irq_event,
2185                    stm32f7_i2c_isr_event,
2186                    stm32f7_i2c_isr_event_thread,
2187                    IRQF_ONESHOT,
2188                    pdev->name, i2c_dev);
2189        if (ret) {
2190            dev_err(&pdev->dev, "Failed to request irq event %i\n",
2191                i2c_dev->irq_event);
2192            goto clk_free;
2193        }
2194
2195        ret = devm_request_irq(&pdev->dev, irq_error,
                                   stm32f7_i2c_isr_error, 0,
2196                    pdev->name, i2c_dev);
2197        if (ret) {
2198            dev_err(&pdev->dev, "Failed to request irq error %i\n",
2199                irq_error);
2200            goto clk_free;
2201        }

2226        if (i2c_dev->bus_rate > I2C_FAST_RATE) {
2227            ret = stm32f7_i2c_setup_fm_plus_bits(pdev, i2c_dev);
2228            if (ret)
2229                goto clk_free;
2230        }
2231
2232        adap = &i2c_dev->adap;
2233        i2c_set_adapdata(adap, i2c_dev);
2234        snprintf(adap->name, sizeof(adap->name), "STM32F7 I2C(%pa)",
```

```
2235            &res->start);
2236        adap->owner = THIS_MODULE;
2237        adap->timeout = 2 * HZ;
2238        adap->retries = 3;
2239        adap->algo = &stm32f7_i2c_algo;
2240        adap->dev.parent = &pdev->dev;
2241        adap->dev.of_node = pdev->dev.of_node;
2242
2243        init_completion(&i2c_dev->complete);
2244
2245        /* Init DMA config if supported */
2246        i2c_dev->dma = stm32_i2c_dma_request(i2c_dev->dev, phy_addr,
2247                                             STM32F7_I2C_TXDR,
2248                                             STM32F7_I2C_RXDR);
2249        if (PTR_ERR(i2c_dev->dma) == -ENODEV)
2250            i2c_dev->dma = NULL;
2251        else if (IS_ERR(i2c_dev->dma)) {
2252            ret = PTR_ERR(i2c_dev->dma);
2253            goto fmp_clear;
2254        }
......
2276        stm32f7_i2c_hw_config(i2c_dev);
2277
2278        ret = i2c_add_adapter(adap);
2279        if (ret)
2280            goto pm_disable;
......
2307        return 0;
......
2340    }
```

第 2117 行，ST 使用 stm32f7_i2c_dev 结构体来表示 STM32MP1 系列 SOC 的 I2C 控制器，这里使用 devm_kzalloc 函数来申请内存。

第 2121~2122 行，调用 platform_get_resource 函数从设备树中获取 I2C1 控制器寄存器物理基地址，也就是 0x40012000。获取到寄存器基地址以后使用 devm_ioremap_resource 函数对其进行内存映射，得到可以在 Linux 中使用的虚拟地址。

第 2127 行和第 2135 行，调用 platform_get_irq 函数获取中断号。

第 2159~2160 行，设置 I2C 频率默认为 I2C_STD_RATE=100KHz，如果设备树节点设置了“clock-frequency”属性的话 I2C 频率就使用 clock-frequency 属性值。

第 2184~2196 行，注册 I2C 控制器的两个中断。

第 2232~2241 行，stm32f7_i2c_dev 结构体有个 adap 的成员变量，adap 就是 i2c_adapter，这里初始化 i2c_adapter。第 2239 行设置 i2c_adapter 的 algo 成员变量为 stm32f7_i2c_algo，也就是设置 i2c_algorithm。

第 2246 行，申请 DMA，看来 STM32MP1 的 I2C 适配器驱动是可以采用 DMA 方式。

第 2276 行，调用 stm32f7_i2c_hw_config 函数初始化 I2C1 控制器的相关硬件寄存器。

第 2278 行，调用 i2c_add_adapter 函数向 Linux 内核注册 i2c_adapter。

stm32f7_i2c_probe 函数主要的工作就是一下两点：

①、初始化 i2c_adapter，设置 i2c_algorithm 为 stm32f7_i2c_algo，最后向 Linux 内核注册 i2c_adapter。

②、初始化 I2C1 控制器的相关寄存器。stm32f7_i2c_algo 包含 I2C1 适配器与 I2C 设备的通信函数 master_xfer，stm32f7_i2c_algo 结构体定义如下：

示例代码 40.3.4 stm32f7_i2c_algo 结构体

```
2098    static const struct i2c_algorithm stm32f7_i2c_algo = {
2099        .master_xfer = stm32f7_i2c_xfer,
2100        .smbus_xfer = stm32f7_i2c_smbus_xfer,
2101        .functionality = stm32f7_i2c_func,
2102        .reg_slave = stm32f7_i2c_reg_slave,
2103        .unreg_slave = stm32f7_i2c_unreg_slave,
2104    };
```

我们先来看一下. functionality，functionality 用于返回此 I2C 适配器支持什么样的通信协议，在这里 functionality 就是 stm32f7_i2c_func 函数，stm32f7_i2c_func 函数内容如下：

示例代码 40.3.5stm32f7_i2c_func 函数

```
2088    static u32 stm32f7_i2c_func(struct i2c_adapter *adap)
2089    {
2090        return I2C_FUNC_I2C | I2C_FUNC_10BIT_ADDR | I2C_FUNC_SLAVE |
2091            I2C_FUNC_SMBUS_QUICK | I2C_FUNC_SMBUS_BYTE |
......
2094            I2C_FUNC_SMBUS_PROC_CALL | I2C_FUNC_SMBUS_PEC |
2095            I2C_FUNC_SMBUS_I2C_BLOCK | I2C_FUNC_SMBUS_HOST_NOTIFY;
2096    }
```

重点来看一下 stm32f7_i2c_xfer 函数，因为最终就是通过此函数来完成与 I2C 设备通信的，此函数内容如下：

示例代码 40.3.6stm32f7_i2c_xfer 函数

```
1657    static int stm32f7_i2c_xfer(struct i2c_adapter *i2c_adap,
1658                    struct i2c_msg msgs[], int num)
1659    {
1660        struct stm32f7_i2c_dev *i2c_dev =
i2c_get_adapdata(i2c_adap);
1661        struct stm32f7_i2c_msg *f7_msg = &i2c_dev->f7_msg;
1662        struct stm32_i2c_dma *dma = i2c_dev->dma;
1663        unsigned long time_left;
1664        int ret;
1665
1666        i2c_dev->msg = msgs;
1667        i2c_dev->msg_num = num;
1668        i2c_dev->msg_id = 0;
1669        f7_msg->smbus = false;
1670
1671        ret = pm_runtime_get_sync(i2c_dev->dev);
```

```
1672        if (ret < 0)
1673            return ret;
1674
1675        ret = stm32f7_i2c_wait_free_bus(i2c_dev);
1676        if (ret)
1677            goto pm_free;
1678
1679        stm32f7_i2c_xfer_msg(i2c_dev, msgs);
1680
1681        time_left = wait_for_completion_timeout(&i2c_dev->complete,
1682                                i2c_dev->adap.timeout);
1683        ret = f7_msg->result;
1684
1685        if (!time_left) {
1686            dev_dbg(i2c_dev->dev, "Access to slave 0x%x timed out\n",
1687                i2c_dev->msg->addr);
1688            if (i2c_dev->use_dma)
1689                dmaengine_terminate_all(dma->chan_using);
1690            ret = -ETIMEDOUT;
1691        }
1692
1693    pm_free:
1694        pm_runtime_mark_last_busy(i2c_dev->dev);
1695        pm_runtime_put_autosuspend(i2c_dev->dev);
1696
1697        return (ret < 0) ? ret : num;
1698    }
```

第 1675 行，调用 stm32f7_i2c_wait_free_bus 函数等待 I2C 总线空闲，也就是读取 I2C 控制的 ISR 寄存器的 bit15(BUSY)位，此位用来标记 I2C 控制器是否忙。

第 1679 行，调用 stm32f7_i2c_xfer_msg 函数发送数据，此函数也是操作 I2C 控制器硬件寄存器的，具体内容这里就不分析了，大家感兴趣可以自己阅读代码。

40.4 I2C 设备驱动编写流程

I2C 适配器驱动 SOC 厂商已经替我们编写好了，我们需要做的就是编写具体的设备驱动，本小节我们就来学习一下 I2C 设备驱动的详细编写流程。

40.4.1 I2C 设备信息描述

1、未使用设备树的时候

首先肯定要描述 I2C 设备节点信息，先来看一下没有使用设备树的时候是如何在 BSP 里面描述 I2C 设备信息的，在未使用设备树的时候需要在 BSP 里面使用 i2c_board_info 结构体来描述一个具体的 I2C 设备。i2c_board_info 结构体如下：

示例代码 40.4.1.1 i2c_board_info 结构体

```
406 struct i2c_board_info {
```

```
407     char        type[I2C_NAME_SIZE];
408     unsigned short  flags;
409     unsigned short  addr;
410     const char  *dev_name;
411     void        *platform_data;
412     struct device_node *of_node;
413     struct fwnode_handle *fwnode;
414     const struct property_entry *properties;
415     const struct resource *resources;
416     unsigned int    num_resources;
417     int     irq;
418 };
```

type 和 addr 这两个成员变量是必须要设置的，一个是 I2C 设备的名字，一个是 I2C 设备的器件地址。举个例子，打开 arch/arm/mach-imx/mach-armadillo5x0.c 文件，此文件中有关于 s35390a 这个 I2C 器件对应的设备描述信息：

示例代码 40.4.1.2 s35390a 的 I2C 设备信息

```
246 static struct i2c_board_info armadillo5x0_i2c_rtc = {
247     I2C_BOARD_INFO("s35390a", 0x30),
248 };
```

示例代码 40.4.1.2 中使用 I2C_BOARD_INFO 来完成 armadillo5x0_i2c_rtc 的初始化工作，I2C_BOARD_INFO 是一个宏，定义如下：

示例代码 40.4.1.3 I2C_BOARD_INFO 宏

```
433 #define I2C_BOARD_INFO(dev_type, dev_addr) \
434     .type = dev_type, .addr = (dev_addr)
```

可以看出，I2C_BOARD_INFO 宏其实就是设置 i2c_board_info 的 type 和 addr 这两个成员变量，因此示例代码 40.4.1.2 的主要工作就是设置 I2C 设备名字为 s35390a，器件地址为 0X30。

大家可以在 Linux 源码里面全局搜索 i2c_board_info，会找到大量以 i2c_board_info 定义的 I2C 设备信息，这些就是未使用设备树的时候 I2C 设备的描述方式，当采用了设备树以后就不会再使用 i2c_board_info 来描述 I2C 设备了。

2、使用设备树的时候

使用设备树的时候 I2C 设备信息通过创建相应的节点就行了，比如在我们的 STM32MP1 的开发板上有一个 I2C 器件 AP3216C，这是三合一的环境传感器，并且该器件挂在 STM32MP1 I2C5 总线接口上，因此必须在 i2c5 节点下创建一个子节点来描述 AP3216C 设备，节点示例如下所示：

示例代码 40.4.1.4 i2c 从设备节点示例

```
1   &i2c5 {
2       pinctrl-names = "default", "sleep";
3       pinctrl-0 = <&i2c5_pins_a>;
4       pinctrl-1 = <&i2c5_pins_sleep_a>;
5       status = "okay";
6
7       ap3216c@1e {
8           compatible = "alientek,ap3216c";
```

```
9           reg = <0x1e>;
10      };
11  };
```

第 2~4 行，设置了 i2c5 的 pinmux 的配置。

第 7~10 行，向 i2c5 添加 ap3216c 子节点，第 7 行“ap3216c@1e”是子节点名字，“@”后面的“1e”就是 ap3216c 的 I2C 器件地址。第 8 行设置 compatible 属性值为“alientek,ap3216c”。第 9 行的 reg 属性也是设置 ap3216c 的器件地址的，因此值为 0x1e。I2C 设备节点的创建重点是 compatible 属性和 reg 属性的设置，一个用于匹配驱动，一个用于设置器件地址。

40.4.2 I2C 设备数据收发处理流程

在 40.2.2 小节已经说过了，I2C 设备驱动首先要做的就是初始化 i2c_driver 并向 Linux 内核注册。当设备和驱动匹配以后 i2c_driver 里面的 probe 函数就会执行，probe 函数里面所做的就是字符设备驱动那一套了。一般需要在 probe 函数里面初始化 I2C 设备，要初始化 I2C 设备就必须能够对 I2C 设备寄存器进行读写操作，这里就要用到 i2c_transfer 函数了。i2c_transfer 函数最终会调用 I2C 适配器中 i2c_algorithm 里面的 master_xfer 函数，对于 STM32MP1 而言就是 stm32f7_i2c_xfer 这个函数。i2c_transfer 函数原型如下：

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

函数参数和返回值含义如下：

adap：所使用的 I2C 适配器，i2c_client 会保存其对应的 i2c_adapter。

msgs：I2C 要发送的一个或多个消息。

num：消息数量，也就是 msgs 的数量。

返回值：负值，失败，其他非负值，发送的 msgs 数量。

我们重点来看一下 msgs 这个参数，这是一个 i2c_msg 类型的指针参数，I2C 进行数据收发说白了就是消息的传递，Linux 内核使用 i2c_msg 结构体来描述一个消息。i2c_msg 结构体定义在 include/uapi/linux/i2c.h 文件中，结构体内容如下：

示例代码 40.4.2.1 i2c_msg 结构体

```
69 struct i2c_msg {
70     __u16 addr;                  /* 从机地址          */
71     __u16 flags;                 /* 标志              */
72     #define I2C_M_TEN            0x0010
73     #define I2C_M_RD             0x0001
74     #define I2C_M_STOP           0x8000
75     #define I2C_M_NOSTART        0x4000
76     #define I2C_M_REV_DIR_ADDR   0x2000
77     #define I2C_M_IGNORE_NAK     0x1000
78     #define I2C_M_NO_RD_ACK      0x0800
79     #define I2C_M_RECV_LEN       0x0400
80     __u16 len;                   /* 消息(本 msg)长度 */
81     __u8 *buf;                   /* 消息数据          */
82 };
```

使用 i2c_transfer 函数发送数据之前要先构建好 i2c_msg，使用 i2c_transfer 进行 I2C 数据收发的示例代码如下：

示例代码 40.4.2.2 I2C 设备多寄存器数据读写

```
1  /* 设备结构体 */
2  struct xxx_dev {
3      ......
4      void *private_data; /* 私有数据，一般会设置为 i2c_client */
5  };
6
7  /*
8   * @description    : 读取 I2C 设备多个寄存器数据
9   * @param - dev    : I2C 设备
10  * @param - reg    : 要读取的寄存器首地址
11  * @param - val    : 读取到的数据
12  * @param - len    : 要读取的数据长度
13  * @return         : 操作结果
14  */
15 static int xxx_read_regs(struct xxx_dev *dev, u8 reg, void *val,
                            int len)
16 {
17     int ret;
18     struct i2c_msg msg[2];
19     struct i2c_client *client = (struct i2c_client *)
                                   dev->private_data;
20
21     /* msg[0]，第一条写消息，发送要读取的寄存器首地址 */
22     msg[0].addr = client->addr;         /* I2C 器件地址     */
23     msg[0].flags = 0;                   /* 标记为发送数据   */
24     msg[0].buf = &reg;                  /* 读取的首地址     */
25     msg[0].len = 1;                     /* reg 长度         */
26
27     /* msg[1]，第二条读消息，读取寄存器数据 */
28     msg[1].addr = client->addr;         /* I2C 器件地址     */
29     msg[1].flags = I2C_M_RD;            /* 标记为读取数据   */
30     msg[1].buf = val;                   /* 读取数据缓冲区   */
31     msg[1].len = len;                   /* 要读取的数据长度 */
32
33     ret = i2c_transfer(client->adapter, msg, 2);
34     if(ret == 2) {
35         ret = 0;
36     } else {
37         ret = -EREMOTEIO;
38     }
39     return ret;
```

```
40 }
41
42 /*
43  * @description    : 向 I2C 设备多个寄存器写入数据
44  * @param - dev    : 要写入的设备结构体
45  * @param - reg    : 要写入的寄存器首地址
46  * @param - val    : 要写入的数据缓冲区
47  * @param - len    : 要写入的数据长度
48  * @return         : 操作结果
49  */
50 static s32 xxx_write_regs(struct xxx_dev *dev, u8 reg, u8 *buf,
                              u8 len)
51 {
52     u8 b[256];
53     struct i2c_msg msg;
54     struct i2c_client *client = (struct i2c_client *)
                                    dev->private_data;
55
56     b[0] = reg;                 /* 寄存器首地址                    */
57     memcpy(&b[1],buf,len);      /* 将要发送的数据拷贝到数组 b 里面  */
58
59     msg.addr = client->addr;    /* I2C 器件地址                    */
60     msg.flags = 0;              /* 标记为写数据                    */
61
62     msg.buf = b;                /* 要发送的数据缓冲区              */
63     msg.len = len + 1;          /* 要发送的数据长度                */
64
65     return i2c_transfer(client->adapter, &msg, 1);
66 }
```

第 2~5 行，设备结构体，在设备结构体里面添加一个执行 void 的指针成员变量 private_data，此成员变量用于保存设备的私有数据。在 I2C 设备驱动中我们一般将其指向 I2C 设备对应的 i2c_client。

第 15~40 行，xxx_read_regs 函数用于读取 I2C 设备多个寄存器数据。第 18 行定义了一个 i2c_msg 数组，2 个数组元素，因为 I2C 读取数据的时候要先发送要读取的寄存器地址，然后再读取数据，所以需要准备两个 i2c_msg。一个用于发送寄存器地址，一个用于读取寄存器值。对于 msg[0]，将 flags 设置为 0，表示写数据。msg[0]的 addr 是 I2C 设备的器件地址，msg[0]的 buf 成员变量就是要读取的寄存器地址。对于 msg[1]，将 flags 设置为 I2C_M_RD，表示读取数据。msg[1]的 buf 成员变量用于保存读取到的数据，len 成员变量就是要读取的数据长度。调用 i2c_transfer 函数完成 I2C 数据读操作。

第 50~66 行，xxx_write_regs 函数用于向 I2C 设备多个寄存器写数据，I2C 写操作要比读操作简单一点，因此一个 i2c_msg 即可。数组 b 用于存放寄存器首地址和要发送的数据，第 59 行设置 msg 的 addr 为 I2C 器件地址。第 60 行设置 msg 的 flags 为 0，也就是写数据。第 62 行设置要发送的数据，也就是数组 b。第 63 行设置 msg 的 len 为 len+1，因为要加上一个字节的寄存器地址。最后通过 i2c_transfer 函数完成向 I2C 设备的写操作。

另外还有两个 API 函数分别用于 I2C 数据的收发操作，这两个函数最终都会调用 i2c_transfer。首先来看一下 I2C 数据发送函数 i2c_master_send，函数原型如下：

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

函数参数和返回值含义如下：

client：I2C 设备对应的 i2c_client。

buf：要发送的数据。

count：要发送的数据字节数，要小于 64KB，以为 i2c_msg 的 len 成员变量是一个 u16(无符号 16 位)类型的数据。

返回值：负值，失败，其他非负值，发送的字节数。

I2C 数据接收函数为 i2c_master_recv，函数原型如下：

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

函数参数和返回值含义如下：

client：I2C 设备对应的 i2c_client。

buf：要接收的数据。

count：要接收的数据字节数，要小于 64KB，以为 i2c_msg 的 len 成员变量是一个 u16(无符号 16 位)类型的数据。

返回值：负值，失败，其他非负值，发送的字节数。

关于 Linux 下 I2C 设备驱动的编写流程就讲解到这里，重点就是 i2c_msg 的构建和 i2c_transfer 函数的调用，接下来我们就编写 AP3216C 这个 I2C 设备的 Linux 驱动。

40.5 硬件原理图分析

AP3612C 原理图如图 40.5.1 所示：

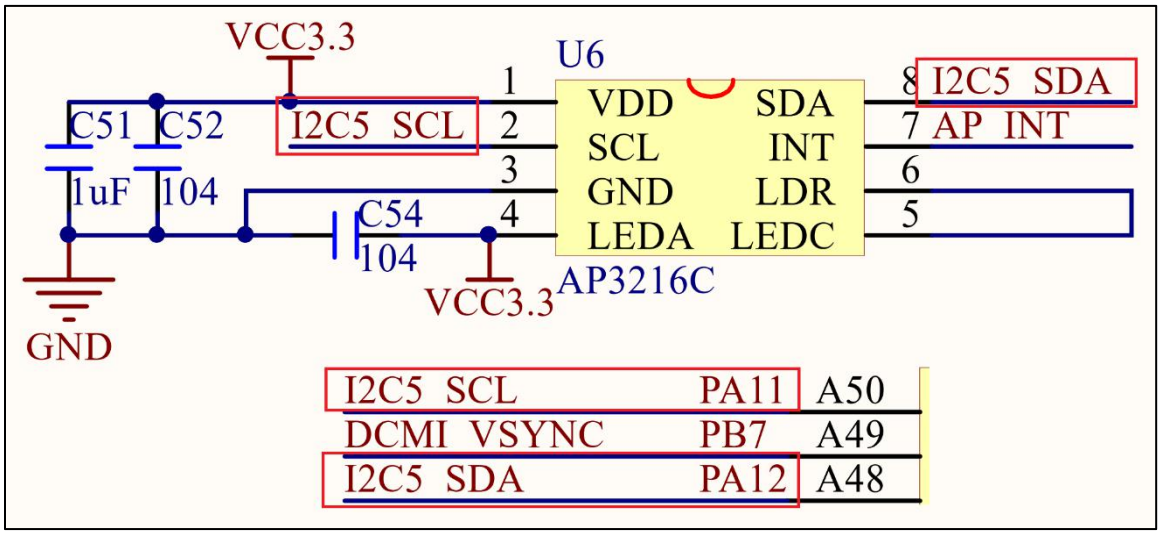

图 40.5.1 AP3216C 原理图

从图 40.5.1 可以看出 AP3216C 使用的是 I2C5，其中 I2C5_SCL 使用的是 PA11 这个 IO，I2C_SDA 使用的是 PA12 这个 IO。AP3216C 还有个中断引脚，这里我们没有用到中断功能。

40.6 实验程序编写

本实验对应的例程路径为：开发板光盘→1、程序源码→2、Linux 驱动例程→21_iic。

40.6.1 修改设备树

1、IO 修改或添加

AP3216C 用到了 I2C5 接口。因为 I2C5 所使用的 IO 分别为 PA11 和 PA12，所以我们要根据数据手册设置 I2C5 的 pinmux 的配置。如果要用到 AP3216C 的中断功能的话还需要初始化 AP_INT 对应的 PE4 这个 IO，本章实验我们不使用中断功能。因此只需要设置 PA11 和 PA12 这个两个 IO 复用为 AF4 功能，ST 其实已经将这个两个 IO 设置好了，打开 stm32mp15-pinctrl.dtsi，然后找到如下内容：

示例代码 40.6.1.1 I2C5 的 pinmux 配置

```
i2c5_pins_a: i2c5-0 {
    pins {
        pinmux = <STM32_PINMUX('A', 11, AF4)>, /* I2C5_SCL   */
             <STM32_PINMUX('A', 12, AF4)>;      /* I2C5_SDA   */
        bias-disable;
        drive-open-drain;
        slew-rate = <0>;
    };
};

i2c5_pins_sleep_a: i2c5-1 {
    pins {
        pinmux = <STM32_PINMUX('A', 11, ANALOG)>, /* I2C5_SCL */
             <STM32_PINMUX('A', 12, ANALOG)>; /* I2C5_SDA */

    };
};
```

示例代码 40.6.1.1 中，定义了 I2C5 接口的两个 pinmux 配置分别为：i2c5_pins_a 和 i2c5_pins_sleep_a。第一个默认的状态下使用，第二个是在 sleep 状态下使用。

2、在 i2c5 节点追加 ap3216c 子节点

接着我们打开 stm32mp157d-atk.dts 文件，通过节点内容追加的方式，向 i2c5 节点中添加“ap3216c@1e”子节点，节点如下所示：

示例代码 40.6.1.2 向 i2c5 追加 ap3216c 子节点

```
&i2c5 {
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&i2c5_pins_a>;
    pinctrl-1 = <&i2c5_pins_sleep_a>;
    status = "okay";

    ap3216c@1e {
        compatible = "alientek,ap3216c";
```

```
9           reg = <0x1e>;
10      };
11  };
```

第 2~4 行，给 I2C5 节点设置了 pinmux 配置。

第 7 行，ap3216c 子节点，@后面的“1e”是 ap3216c 的器件地址。

第 8 行，设置 compatible 值为“alientek,ap3216c”。

第 9 行，reg 属性也是设置 ap3216c 器件地址的，因此 reg 设置为 0x1e。

设备树修改完成以后使用“make dtbs”重新编译一下，然后使用新的设备树启动 Linux 内核。/sys/bus/i2c/devices 目录下存放着所有 I2C 设备，如果设备树修改正确的话，会在/sys/bus/i2c/devices 目录下看到一个名为“0-001e”的子目录，如图 40.6.1.1 所示：

```
[root@ATK-stm32mp1]:~$ cd /sys/bus/i2c/devices/
[root@ATK-stm32mp1]:/sys/bus/i2c/devices$ ls
0-001e  i2c-0
[root@ATK-stm32mp1]:/sys/bus/i2c/devices$
```

图 40.6.1.1 当前系统 I2C 设备

图 40.6.1.1 中的“0-001e”就是 ap3216c 的设备目录，“1e”就是 ap3216c 器件地址。进入 0-001e 目录，可以看到“name”文件，name 文件保存着此设备名字，在这里就是“ap3216c”，如图 40.6.1.2 所示：

```
[root@ATK-stm32mp1]:/sys/bus/i2c/devices$ cat 0-001e/name
ap3216c
[root@ATK-stm32mp1]:/sys/bus/i2c/devices$
```

图 40.6.1.2 ap3216c 器件名字

40.6.2 AP3216C 驱动编写

新建名为“21_iic”的文件夹，然后在 21_iic 文件夹里面创建 vscode 工程，工作区命名为“iic”。工程创建好以后新建 ap3216c.c 和 ap3216creg.h 这两个文件，ap3216c.c 为 AP3216C 的驱动代码，ap3216creg.h 是 AP3216C 寄存器头文件。先在 ap3216creg.h 中定义好 AP3216C 的寄存器，输入如下内容，

示例代码 40.6.2.1 ap3216creg.h 文件代码段

```
1   #ifndef AP3216C_H
2   #define AP3216C_H
3   /***************************************************************
4   Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
5   文件名     : ap3216creg.h
6   作者       : 正点原子 Linux 团队
7   版本       : V1.0
8   描述       : AP3216C 寄存器地址描述头文件
9   其他       : 无
10  论坛       : www.openedv.com
11  日志       : 初版 V1.0 2021/03/19 正点原子 Linux 团队创建
12  ***************************************************************/
13
14  #define AP3216C_ADDR        0X1E        /* AP3216C 器件地址     */
15
16  /* AP3316C 寄存器 */
```

```
17  #define AP3216C_SYSTEMCONG  0x00    /* 配置寄存器          */
18  #define AP3216C_INTSTATUS   0X01    /* 中断状态寄存器      */
19  #define AP3216C_INTCLEAR    0X02    /* 中断清除寄存器      */
20  #define AP3216C_IRDATALOW   0x0A    /* IR 数据低字节       */
21  #define AP3216C_IRDATAHIGH  0x0B    /* IR 数据高字节       */
22  #define AP3216C_ALSDATALOW  0x0C    /* ALS 数据低字节      */
23  #define AP3216C_ALSDATAHIGH 0X0D    /* ALS 数据高字节      */
24  #define AP3216C_PSDATALOW   0X0E    /* PS 数据低字节       */
25  #define AP3216C_PSDATAHIGH  0X0F    /* PS 数据高字节       */
26
27  #endif
```

ap3216creg.h 没什么好讲的，就是一些寄存器宏定义。然后在 ap3216c.c 输入如下内容：

示例代码 40.6.2.2 ap3216.c 文件代码段

```
1  /***************************************************************
2  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
3  文件名    : ap3216c.c
4  作者      : 正点原子 Linux 团队
5  版本      : V1.0
6  描述      : AP3216C 驱动程序
7  其他      : 无
8  论坛      : www.openedv.com
9  日志      : 初版 V1.0 2021/03/19 正点原子 Linux 团队创建
10 ***************************************************************/
11 #include <linux/types.h>
12 #include <linux/kernel.h>
13 #include <linux/delay.h>
14 #include <linux/ide.h>
15 #include <linux/init.h>
16 #include <linux/module.h>
17 #include <linux/errno.h>
18 #include <linux/gpio.h>
19 #include <linux/cdev.h>
20 #include <linux/device.h>
21 #include <linux/of_gpio.h>
22 #include <linux/semaphore.h>
23 #include <linux/timer.h>
24 #include <linux/i2c.h>
25 #include <asm/mach/map.h>
26 #include <asm/uaccess.h>
27 #include <asm/io.h>
28 #include "ap3216creg.h"
29
30 #define AP3216C_CNT 1
31 #define AP3216C_NAME    "ap3216c"
```

```
32
33  struct ap3216c_dev {
34      struct i2c_client *client;  /* i2c 设备        */
35      dev_t devid;                /* 设备号          */
36      struct cdev cdev;           /* cdev            */
37      struct class *class;        /* 类              */
38      struct device *device;      /* 设备            */
39      struct device_node  *nd;    /* 设备节点        */
40      unsigned short ir, als, ps; /* 三个光传感器数据  */
41  };
42
43  /*
44   * @description   :   从 ap3216c 读取多个寄存器数据
45   * @param - dev   :   ap3216c 设备
46   * @param - reg   :   要读取的寄存器首地址
47   * @param - val   :   读取到的数据
48   * @param - len   :   要读取的数据长度
49   * @return        :   操作结果
50   */
51  static int ap3216c_read_regs(struct ap3216c_dev *dev, u8 reg,
                                 void *val, int len)
52  {
53      int ret;
54      struct i2c_msg msg[2];
55      struct i2c_client *client = (struct i2c_client *)dev->client;
56
57      /* msg[0]为发送要读取的首地址 */
58      msg[0].addr = client->addr;         /* ap3216c 地址     */
59      msg[0].flags = 0;                   /* 标记为发送数据    */
60      msg[0].buf = &reg;                  /* 读取的首地址      */
61      msg[0].len = 1;                     /* reg 长度         */
62
63      /* msg[1]读取数据 */
64      msg[1].addr = client->addr;         /* ap3216c 地址     */
65      msg[1].flags = I2C_M_RD;            /* 标记为读取数据    */
66      msg[1].buf = val;                   /* 读取数据缓冲区    */
67      msg[1].len = len;                   /* 要读取的数据长度  */
68
69      ret = i2c_transfer(client->adapter, msg, 2);
70      if(ret == 2) {
71          ret = 0;
72      } else {
73          printk("i2c rd failed=%d reg=%06x len=%d\n",ret, reg, len);
74          ret = -EREMOTEIO;
```

```
75      }
76      return ret;
77  }
78
79  /*
80   * @description: 向 ap3216c 多个寄存器写入数据
81   * @param - dev: ap3216c 设备
82   * @param - reg: 要写入的寄存器首地址
83   * @param - val: 要写入的数据缓冲区
84   * @param - len: 要写入的数据长度
85   * @return     : 操作结果
86   */
87  static s32 ap3216c_write_regs(struct ap3216c_dev *dev, u8 reg,
                                  u8 *buf, u8 len)
88  {
89      u8 b[256];
90      struct i2c_msg msg;
91      struct i2c_client *client = (struct i2c_client *)dev->client;
92
93      b[0] = reg;                     /* 寄存器首地址                    */
94      memcpy(&b[1],buf,len);          /* 将要写入的数据拷贝到数组 b 里面  */
95
96      msg.addr = client->addr;        /* ap3216c 地址                    */
97      msg.flags = 0;                  /* 标记为写数据                    */
98
99      msg.buf = b;                    /* 要写入的数据缓冲区              */
100     msg.len = len + 1;              /* 要写入的数据长度                */
101
102     return i2c_transfer(client->adapter, &msg, 1);
103 }
104
105 /*
106  * @description: 读取 ap3216c 指定寄存器值，读取一个寄存器
107  * @param - dev: ap3216c 设备
108  * @param - reg: 要读取的寄存器
109  * @return     : 读取到的寄存器值
110  */
111 static unsigned char ap3216c_read_reg(struct ap3216c_dev *dev,
                                          u8 reg)
112 {
113     u8 data = 0;
114
115     ap3216c_read_regs(dev, reg, &data, 1);
116     return data;
```

```
117 }
118
119 /*
120  * @description: 向 ap3216c 指定寄存器写入指定的值，写一个寄存器
121  * @param - dev: ap3216c 设备
122  * @param - reg: 要写的寄存器
123  * @param - data:要写入的值
124  * @return   :    无
125  */
126 static void ap3216c_write_reg(struct ap3216c_dev *dev, u8 reg,
                                  u8 data)
127 {
128     u8 buf = 0;
129     buf = data;
130     ap3216c_write_regs(dev, reg, &buf, 1);
131 }
132
133 /*
134  * @description   : 读取 AP3216C 的数据，包括 ALS,PS 和 IR, 注意！如果同时
135  *                 :打开 ALS,IR+PS 两次数据读取的时间间隔要大于 112.5ms
136  * @param - ir    : ir 数据
137  * @param - ps    : ps 数据
138  * @param - ps    : als 数据
139  * @return        : 无。
140  */
141 void ap3216c_readdata(struct ap3216c_dev *dev)
142 {
143     unsigned char i =0;
144     unsigned char buf[6];
145
146     /* 循环读取所有传感器数据 */
147     for(i = 0; i < 6; i++) {
148         buf[i] = ap3216c_read_reg(dev, AP3216C_IRDATALOW + i);
149     }
150
151     if(buf[0] & 0X80)     /* IR_OF 位为 1,则数据无效   */
152         dev->ir = 0;
153     else                  /* 读取 IR 传感器的数据       */
154         dev->ir = ((unsigned short)buf[1] << 2) | (buf[0] & 0X03);
155
156     dev->als = ((unsigned short)buf[3] << 8) | buf[2];
157
158     if(buf[4] & 0x40)     /* IR_OF 位为 1,则数据无效   */
159         dev->ps = 0;
```

```
160     else                    /* 读取 PS 传感器的数据      */
161         dev->ps = ((unsigned short)(buf[5] & 0X3F) << 4) | (buf[4] &
0X0F);
162 }
163
164 /*
165  * @description   : 打开设备
166  * @param - inode : 传递给驱动的 inode
167  * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
168  *                  一般在 open 的时候将 private_data 指向设备结构体。
169  * @return        : 0 成功;其他 失败
170  */
171 static int ap3216c_open(struct inode *inode, struct file *filp)
172 {
173     /* 从 file 结构体获取 cdev 指针，再根据 cdev 获取 ap3216c_dev 首地址 */
174     struct cdev *cdev = filp->f_path.dentry->d_inode->i_cdev;
175     struct ap3216c_dev *ap3216cdev = container_of(cdev,
                              struct ap3216c_dev, cdev);
176
177     /* 初始化 AP3216C */
178     ap3216c_write_reg(ap3216cdev, AP3216C_SYSTEMCONG, 0x04);
179     mdelay(50);
180     ap3216c_write_reg(ap3216cdev, AP3216C_SYSTEMCONG, 0X03);
181     return 0;
182 }
183
184 /*
185  * @description   : 从设备读取数据
186  * @param - filp  : 要打开的设备文件(文件描述符)
187  * @param - buf   : 返回给用户空间的数据缓冲区
188  * @param - cnt   : 要读取的数据长度
189  * @param - offt  : 相对于文件首地址的偏移
190  * @return        : 读取的字节数，如果为负值，表示读取失败
191  */
192 static ssize_t ap3216c_read(struct file *filp, char __user *buf,
                              size_t cnt, loff_t *off)
193 {
194     short data[3];
195     long err = 0;
196
197     struct cdev *cdev = filp->f_path.dentry->d_inode->i_cdev;
198     struct ap3216c_dev *dev = container_of(cdev, struct ap3216c_dev,
                                           cdev);
199
```

```
200     ap3216c_readdata(dev);
201
202     data[0] = dev->ir;
203     data[1] = dev->als;
204     data[2] = dev->ps;
205     err = copy_to_user(buf, data, sizeof(data));
206     return 0;
207 }
208
209 /*
210  * @description   : 关闭/释放设备
211  * @param - filp  : 要关闭的设备文件(文件描述符)
212  * @return        : 0 成功;其他 失败
213  */
214 static int ap3216c_release(struct inode *inode, struct file *filp)
215 {
216     return 0;
217 }
218
219 /* AP3216C 操作函数 */
220 static const struct file_operations ap3216c_ops = {
221     .owner = THIS_MODULE,
222     .open = ap3216c_open,
223     .read = ap3216c_read,
224     .release = ap3216c_release,
225 };
226
227  /*
228   * @description  : i2c 驱动的 probe 函数，当驱动与
229   *                 设备匹配以后此函数就会执行
230   * @param - client  : i2c 设备
231   * @param - id      : i2c 设备 ID
232   * @return          : 0，成功;其他负值,失败
233   */
234 static int ap3216c_probe(struct i2c_client *client,
                             const struct i2c_device_id *id)
235 {
236     int ret;
237     struct ap3216c_dev *ap3216cdev;
238
239
240     ap3216cdev = devm_kzalloc(&client->dev, sizeof(*ap3216cdev),
                                  GFP_KERNEL);
241     if(!ap3216cdev)
```

```
242         return -ENOMEM;
243 
244     /* 注册字符设备驱动 */
245     /* 1、创建设备号 */
246     ret = alloc_chrdev_region(&ap3216cdev->devid, 0, AP3216C_CNT,
                                  AP3216C_NAME);
247     if(ret < 0) {
248         pr_err("%s Couldn't alloc_chrdev_region, ret=%d\r\n",
                    AP3216C_NAME, ret);
249         return -ENOMEM;
250     }
251 
252     /* 2、初始化 cdev */
253     ap3216cdev->cdev.owner = THIS_MODULE;
254     cdev_init(&ap3216cdev->cdev, &ap3216c_ops);
255 
256     /* 3、添加一个 cdev */
257     ret = cdev_add(&ap3216cdev->cdev, ap3216cdev->devid,
                  AP3216C_CNT);
258     if(ret < 0) {
259         goto del_unregister;
260     }
261 
262     /* 4、创建类 */
263     ap3216cdev->class = class_create(THIS_MODULE, AP3216C_NAME);
264     if (IS_ERR(ap3216cdev->class)) {
265         goto del_cdev;
266     }
267 
268     /* 5、创建设备 */
269     ap3216cdev->device = device_create(ap3216cdev->class, NULL,
                             ap3216cdev->devid, NULL, AP3216C_NAME);
270     if (IS_ERR(ap3216cdev->device)) {
271         goto destroy_class;
272     }
273     ap3216cdev->client = client;
274     /* 保存 ap3216cdev 结构体 */
275     i2c_set_clientdata(client,ap3216cdev);
276 
277     return 0;
278 destroy_class:
279     device_destroy(ap3216cdev->class, ap3216cdev->devid);
280 del_cdev:
281     cdev_del(&ap3216cdev->cdev);
```

```
282 del_unregister:
283     unregister_chrdev_region(ap3216cdev->devid, AP3216C_CNT);
284     return -EIO;
285 }
286 
287 /*
288  * @description  : i2c 驱动的 remove 函数，移除 i2c 驱动的时候此函数会执行
289  * @param - client   : i2c 设备
290  * @return           : 0，成功;其他负值,失败
291  */
292 static int ap3216c_remove(struct i2c_client *client)
293 {
294     struct ap3216c_dev *ap3216cdev = i2c_get_clientdata(client);
295     /* 注销字符设备驱动 */
296     /* 1、删除 cdev */
297     cdev_del(&ap3216cdev->cdev);
298     /* 2、注销设备号 */
299     unregister_chrdev_region(ap3216cdev->devid, AP3216C_CNT);
300     /* 3、注销设备 */
301     device_destroy(ap3216cdev->class, ap3216cdev->devid);
302     /* 4、注销类 */
303     class_destroy(ap3216cdev->class);
304     return 0;
305 }
306 
307 /* 传统匹配方式 ID 列表 */
308 static const struct i2c_device_id ap3216c_id[] = {
309     {"alientek,ap3216c", 0},
310     {}
311 };
312 
313 /* 设备树匹配列表 */
314 static const struct of_device_id ap3216c_of_match[] = {
315     { .compatible = "alientek,ap3216c" },
316     { /* Sentinel */ }
317 };
318 
319 /* i2c 驱动结构体 */
320 static struct i2c_driver ap3216c_driver = {
321     .probe = ap3216c_probe,
322     .remove = ap3216c_remove,
323     .driver = {
324             .owner = THIS_MODULE,
325             .name = "ap3216c",
```

```
326             .of_match_table = ap3216c_of_match,
327         },
328     .id_table = ap3216c_id,
329 };
330
331 /*
332  * @description   : 驱动入口函数
333  * @param         : 无
334  * @return        : 无
335  */
336 static int __init ap3216c_init(void)
337 {
338     int ret = 0;
339
340     ret = i2c_add_driver(&ap3216c_driver);
341     return ret;
342 }
343
344 /*
345  * @description   : 驱动出口函数
346  * @param         : 无
347  * @return        : 无
348  */
349 static void __exit ap3216c_exit(void)
350 {
351     i2c_del_driver(&ap3216c_driver);
352 }
353
354 /* module_i2c_driver(ap3216c_driver) */
355
356 module_init(ap3216c_init);
357 module_exit(ap3216c_exit);
358 MODULE_LICENSE("GPL");
359 MODULE_AUTHOR("ALIENTEK");
360 MODULE_INFO(intree, "Y");
```

在示例代码 40.6.2.2 里，没有定义一个全局变量，那是因为 linux 内核不推荐使用全局变量，要使用内存的就用 devm_kzalloc 之类的函数去申请空间。

第 33~41 行，自定义一个 ap3216c_dev 结构体。第 34 行的 client 成员变量用来存储从设备树提供的 i2c_client 结构体。第 40 行的 ir、als 和 ps 分别存储 AP3216C 的 IR、ALS 和 PS 数据。

第 51~77 行，ap3216c_read_regs 函数实现多字节读取，但是 AP3216C 好像不支持连续多字节读取，此函数在测试其他 I2C 设备的时候可以实现多给字节连续读取，但是在 AP3216C 上不能连续读取多个字节，不过读取一个字节没有问题的。

第 87~103 行，ap3216c_write_regs 函数实现连续多字节写操作。

第 111~117 行，ap3216c_read_reg 函数用于读取 AP3216C 的指定寄存器数据，用于一个寄存器的数据读取。

第 126~131 行，ap3216c_write_reg 函数用于向 AP3216C 的指定寄存器写入数据，用于一个寄存器的数据写操作。

第 141~162 行，读取 AP3216C 的 PS、ALS 和 IR 等传感器原始数据值。

第 171~225 行，标准的字符设备驱动框架。ap3216c_dev 结构体里有一个 cdev 的变量成员，第 174 行就是获取 ap3216c_dev 里的 cdev 这个变量的地址，在第 175 行使用 container_of 宏获取 ap3216c_dev 的首地址。

第 234~285 行，ap3216c_probe 函数，当 I2C 设备和驱动匹配成功以后此函数就会执行，和 platform 驱动框架一样。此函数前面都是标准的字符设备注册代码，第 275 行，调用 i2c_set_clientdata 函数将 ap3216cdev 变量的地址绑定到 client，进行绑定之后，可以通过 i2c_get_clientdata 来获取 ap3216cdev 变量指针。

第 292~305 行，ap3216c_remove 函数，当 I2C 驱动模块卸载时会执行此函数。第 294 行通过调用 i2c_get_clientdata 函数来得到 ap3216cdev 变量的地址，后面执行的一系列卸载、注销操作都是前面讲到过的标准字符设备。

第 308~311 行，ap3216c_id 匹配表，i2c_device_id 类型。用于传统的设备和驱动匹配，也就是没有使用设备树的时候。

第 314~317 行，ap3216c_of_match 匹配表，of_device_id 类型，用于设备树设备和驱动匹配。这里只写了一个 compatible 属性，值为“alientek,ap3216c”。

第 320~329 行，ap3216c_driver 结构体变量，i2c_driver 类型。

第 336~342 行，驱动入口函数 ap3216c_init，此函数通过调用 i2c_add_driver 来向 Linux 内核注册 i2c_driver，也就是 ap3216c_driver。

第 349~352 行，驱动出口函数 ap3216c_exit，此函数通过调用 i2c_del_driver 来注销掉前面注册的 ap3216c_driver。

40.6.3 编写测试 APP

新建 ap3216cApp.c 文件，然后在里面输入如下所示内容：

示例代码 40.6.3.1 测试 APP

```
12  #include "stdio.h"
13  #include "unistd.h"
14  #include "sys/types.h"
15  #include "sys/stat.h"
16  #include "sys/ioctl.h"
17  #include "fcntl.h"
18  #include "stdlib.h"
19  #include "string.h"
20  #include <poll.h>
21  #include <sys/select.h>
22  #include <sys/time.h>
23  #include <signal.h>
24  #include <fcntl.h>
25  /*
26   * @description   : main 主程序
27   * @param - argc  : argv 数组元素个数
```

```
28  * @param - argv  : 具体参数
29  * @return        : 0 成功;其他 失败
30  */
31 int main(int argc, char *argv[])
32 {
33     int fd;
34     char *filename;
35     unsigned short databuf[3];
36     unsigned short ir, als, ps;
37     int ret = 0;
38 
39     if (argc != 2) {
40         printf("Error Usage!\r\n");
41         return -1;
42     }
43 
44     filename = argv[1];
45     fd = open(filename, O_RDWR);
46     if(fd < 0) {
47         printf("can't open file %s\r\n", filename);
48         return -1;
49     }
50 
51     while (1) {
52         ret = read(fd, databuf, sizeof(databuf));
53         if(ret == 0) {           /* 数据读取成功 */
54             ir =  databuf[0];    /* ir 传感器数据 */
55             als = databuf[1];    /* als 传感器数据 */
56             ps =  databuf[2];    /* ps 传感器数据 */
57             printf("ir = %d, als = %d, ps = %d\r\n", ir, als, ps);
58         }
59         usleep(200000);   /*100ms */
60     }
61     close(fd);            /* 关闭文件 */
62     return 0;
63 }
```

ap3216cApp.c 文件内容很简单，就是在 while 循环中不断的读取 AP3216C 的设备文件，从而得到 ir、als 和 ps 这三个数据值，然后将其输出到终端上。

40.7 运行测试

40.7.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m

变量的值改为“ap3216c.o”，Makefile 内容如下所示：

示例代码 40.7.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/my_linux/linux-5.4.31
......
4  obj-m := ap3216c.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为“ap3216c.o”。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“ap3216c.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译 ap3216cApp.c 这个测试程序：

```
arm-none-linux-gnueabihf-gcc ap3216cApp.c -o ap3216cApp
```

编译成功以后就会生成 ap3216cApp 这个应用程序。

40.7.2 运行测试

将上一小节编译出来 ap3216c.ko 和 ap3216cApp 这两个文件拷贝到 rootfs/lib/modules/5.4.31 目录中，重启开发板，进入到目录 lib/modules/5.4.31 中。输入如下命令加载 ap3216c.ko 这个驱动模块。

```
depmod                  //第一次加载驱动的时候需要运行此命令
modprobe ap3216c        //加载驱动模块
```

当驱动模块加载成功以后使用 ap3216cApp 来测试，输入如下命令：

```
./ap3216cApp /dev/ap3216c
```

测试 APP 会不断的从 AP3216C 中读取数据，然后输出到终端上，如图 40.7.2.1 所示：

```
[root@ATK-stm32mp1]:/lib/modules/5.4.31$ ./ap3216cApp  /dev/ap3216c
ir = 0, als = 0, ps = 0
ir = 0, als = 636, ps = 6
ir = 0, als = 635, ps = 0
ir = 0, als = 633, ps = 3
ir = 1, als = 633, ps = 5
ir = 4, als = 638, ps = 0
ir = 0, als = 640, ps = 0
```

图 40.7.2.1 获取到的 AP3216C 数据

大家可以用手电筒照一下 AP3216C，或者手指靠近 AP3216C 来观察传感器数据有没有变化。