Linux I2C设备驱动基本规范

原创 奔跑的码仔 奔跑的码仔 2020-08-04 19:00

不同于单片机驱动开发，即使是简单的I2C设备驱动程序，如果要在Linux上实现同种功能的驱动程序，事情也会变的复杂起来。对于初学者而言，主要的困难就是不知道如何使用Linux现有的驱动框架，去完成驱动程序的开发。I2C设备驱动，相对来说比较简单，但由于Linux大部分设备驱动框架十分的类似，所以，通过对于I2C驱动框架的学习，可以作为继续深入Linux其他设备驱动框架的基础。

学习一项技术的最好方式就是去应用它，所以为了更为高效的学习，本文的最后结合一个i2c设备驱动实例，来分析如何从零实现一个驱动程序。

程序框架

本文主要为了学习如何在Linux上实现一个I2C设备驱动，而对于具体的I2C驱动架构不会详细的展开。一个完整的I2C设备驱动主要包括如下几个部分：

1. 定义i2c_driver数据结构
2. 注册i2c_driver
3. 实现设备访问

具体实现上述几部分时，会用到Linux系统I2C相关的很多数据结构和接口，下面简要介绍一下主要的数据结构和接口。

数据结构

struct i2c_driver

struct i2c_driver代表一个I2C设备驱动实体。其主要的数据成员如下：

```
1  struct i2c_driver {
2      ...
3
4      /* Standard driver model interfaces */
5      int (*probe)(struct i2c_client *client, const struct i2c_device_id *id);
6      int (*remove)(struct i2c_client *client); <-----------(2)
7      ...
8
9      struct device_driver driver;          <-----------(3)
10     const struct i2c_device_id *id_table; <-----------(4)
```

```
11    };
```

- (1) 设备与驱动程序匹配之后，会调用该probe接口完成设备的初始化和注册。
- (2) 卸载设备驱动时，会调用该接口完成设备注销和相关资源的释放。
- (3) Linux设备驱动抽象结构。
- (4) i2c设备id表，驱动程序中根据具体的设备ID来区分不同的设备。以此来达到兼容同种类型，不同型号的设备。

这里还需要进一步说一下struct device_driver结构。

```
1  struct device_driver {
2        const char     *name;              <-----------(1)
3        const struct of_device_id  *of_match_table;     <-----------(2)
4    };
```

- (1) 设备驱动名称，Linux内核未支持DeviceTree之前，设备和驱动程序需要根据name进行匹配。
- (2) 设备和驱动匹配类型表，设备驱动程序需要定义其支持的设备类型，并初始化该ofmatchtable。

struct i2c_device

struct i2c_client代表一个具体的I2C设备实体。其主要数据成员如下：

```
1  struct i2c_client {
2      unsigned short addr;          <-----------(1)
3      char name[I2C_NAME_SIZE];     <-----------(2)
4      struct i2c_adapter *adapter;  <-----------(3)
5    };
```

- (1) 设备芯片通信地址，默认为7bit，保存在addr的低7bit。
- (2) 设备名称。
- (3) I2C设备所依赖的I2C适配器，该适配器用于完成具体的I2C物理信号通信。

struct i2c_device_id

struct i2c_device_id代表一种具体的i2c设备类型，设备与驱动匹配之后，会确定具体的设备类型。其数据成员如下：

```
1  struct i2c_device_id {
2      char name[I2C_NAME_SIZE];                                  <----------
```

```
    kernel_ulong_t driver_data;  /* Data private to the driver */  <----------
};
```

- （1）设备类型名称。
- （2）设备私有数据，数据类型为long，可以指向一个具体的整型，也可以指向一个指针。

struct i2c_adapter

struct i2c_adapter代表具体的I2C控制器，其完成I2C的物理信号通信。对于一个设备驱动，一般不会涉及到该数据结构，待到学习I2C架构时，再进行详细的分析。

主要API

要实现I2C设备驱动主要使用到三个API，其中两个probe和remove，在介绍struct i2cdriver时进行了简单的介绍。另外一个modulei2c_driver，其用于向系统注册一个i2c驱动程序。

probe

设备与驱动程序匹配之后，会调用该probe接口完成设备的初始化和注册。

- 设备初始化

  具体到每个I2C设备芯片，一般都会有一些参数，I2C设备驱动程序会将这些参数封装成结构，然后，在设备初始化阶段完成这些参数的初始化设置。对于设备的初始化配置，一般来源于设备的DTS配置，下面介绍DTS配置时，会具体介绍到。

- 设备注册

  每个I2C设备最终都会绑定到一种具体的Linux设备上，比如，RTC设备，EEROM设备，IIO设备等。设备注册完成的任务就是将该I2C设备通过具体的设备注册接口注册到系统中。这个说的可能比较绕，举个例子，比如，我们编写的这个I2C驱动，用于驱动一个RTC设备，那我们就需要调用devm_rtc_device_register接口进行设备注册，又比如，我们编写一个基于IIO的adc设备驱动，那我们就需要调用iio_device_register接口进行设备注册。

remove

卸载设备驱动时，系统会调用该接口完成设备的注销和相关资源的释放。

module_i2c_drvier

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

module_i2c_driver用于将I2C设备驱动注册到系统，其中，i2c_add_driver和i2c_del_register用于完成驱动的添加和删除。

通信API

I2C设备驱动与设备进行通信时，有两种方式可供选择：（1）基于i2c_msg方式；(2)基于SMbus方式。

i2c_msg

i2cmsg可以作为I2C传输的一个单元进行使用，通过将通信数据封装到i2cmsg中，之后再通过i2c_transfer完成驱动程序与设备的I2C通信。

struct i2c_msg的定义如下：

```
struct i2c_msg {
  __u16 addr;  /* slave address      */
  __u16 flags;
  #define I2C_M_RD    0x0001  /* read data, from slave to master */
            /* I2C_M_RD is guaranteed to be 0x0001! */
  #define I2C_M_TEN    0x0010  /* this is a ten bit chip address */
  #define I2C_M_DMA_SAFE    0x0200  /* the buffer of this message is DMA safe
            /* makes only sense in kernelspace */
            /* userspace buffers are copied anyway */
  #define I2C_M_RECV_LEN    0x0400  /* length will be first received byte */
  #define I2C_M_NO_RD_ACK    0x0800  /* if I2C_FUNC_PROTOCOL_MANGLING */
  #define I2C_M_IGNORE_NAK  0x1000  /* if I2C_FUNC_PROTOCOL_MANGLING */
  #define I2C_M_REV_DIR_ADDR  0x2000  /* if I2C_FUNC_PROTOCOL_MANGLING */
  #define I2C_M_NOSTART    0x4000  /* if I2C_FUNC_NOSTART */
  #define I2C_M_STOP    0x8000  /* if I2C_FUNC_PROTOCOL_MANGLING */
    __u16 len;    /* msg length        */
    __u8 *buf;    /* pointer to msg data      */
  };
```

下面展示了一个读取寄存reg1上数据的示例。

```
struct i2c_msg msg[2];

  msg[0].addr = client->addr;
  msg[0].flags = 0;//写
  msg[0].len = 1;
  msg[0].buf = &reg1;

  msg[1].addr = client->addr;
  msg[1].flags = I2C_M_RD;//读
  msg[1].len   = sizeof(buf);
  msg[1].buf   = &buf[0];

  i2c_transfer(client->adapeter, msg, 2);
```

上面定义了两个msg，第一个msg定义了将要读取的设备寄存器，第二个msg用于读取该寄存器中的数据。

SMbus

SMbus是Intel基于I2C推出的一种通用的通信协议(System Management Bus)，其可以认为是I2C的通信子集，其定义了一套I2C主-从设备之间通信的时序。SMbus与I2C的关系，可以类比与网络通信中的HTTP和TCP的关系，I2C提供的基本的通信规则，其上可以跑的是裸数据，而SMbus规定了数据的格式。Linux系统的I2C通信架构提供了关于SMbus的支持，在支持SMbus的适配器和I2C设备之间可以使用SMbus协议进行通信。具体的SMbus协议可以参考Linux内核文档。

SMbus提供丰富的通信接口，用于传输单字节、双字节、字节数据数组等数据单元。

1. 单字节数据传输：

```
extern s32 i2c_smbus_read_byte(const struct i2c_client *client);
    extern s32 i2c_smbus_write_byte(const struct i2c_client *client, u8 value)
    extern s32 i2c_smbus_read_byte_data(const struct i2c_client *client,
                u8 command);
    extern s32 i2c_smbus_write_byte_data(const struct i2c_client *client,
                 u8 command, u8 value);
```

2. 双字节数据传输：

```
extern s32 i2c_smbus_read_word_data(const struct i2c_client *client,
                u8 command);
    extern s32 i2c_smbus_write_word_data(const struct i2c_client *client,
                u8 command, u16 value);
```

3. 字节数组数据传输：

```
extern s32 i2c_smbus_read_block_data(const struct i2c_client *client,
             u8 command, u8 *values);
extern s32 i2c_smbus_write_block_data(const struct i2c_client *client,
                  u8 command, u8 length, const u8 *values);
    /* Returns the number of read bytes */
extern s32 i2c_smbus_read_i2c_block_data(const struct i2c_client *client,
               u8 command, u8 length, u8 *values);
extern s32 i2c_smbus_write_i2c_block_data(const struct i2c_client *client,
                u8 command, u8 length,
                const u8 *values);
```

几点注意事项：

1. command代表具体的设备寄存器。
2. Linux推荐尽可能的使用SMbus协议与设备进行I2C通信。
3. 在使用SMbus进行通信之前，需要检查当前的适配器是否支持需要的SMbus操作，比如：

```
if (!i2c_check_functionality(adapter, I2C_FUNC_SMBUS_BYTE_DATA
       | I2C_FUNC_SMBUS_I2C_BLOCK)) {
       dev_err(&adapter->dev, "doesn't support required functionality\n");
       return -EIO;
    }
```

上面这段代码用于检查当前的adapter是否支持：I2C_FUNC_SMBUS_BYTE_DATA和I2C_FUNC_SMBUS_I2C_BLOCK这两项操作，如果不支持，返回EIO错误。

设备访问

用户空间访问I2C设备的方式各不相同，具体需要依据I2C设备的类型而定，比如，RTC设备通过rtc_class_ops 接口访问，ads1015通过sensor_device_attribute访问。

每个I2C适配器在/dev目录下都有一个对应的设备文件，我们通过这个设备文件直接访问挂接在该适配器之下的设备。

DTS配置

设备驱动程序编写完成之后，具体设备需要在DTS中定义其挂接的适配器，并配置设备的通信地址等信息，下面就是一个典型的I2C设备配置信息。

```
&i2c1 {
            clock-frequency = <100000>;
            pinctrl-names = "default";
            pinctrl-0 = <&pinctrl_i2c1>;
            status = "okay";

        rx8010:rtc@32 {
                compatible = "epson,rx8010";
                status = "okay";
                reg = <0x32>;
        };
    };
```

- rx8010设备挂接在i2c1适配器下。
- rx8010的通信地址为0x32。
- rx8010的驱动程序兼容字段为“epson,rx8010”，对应到上面所讲的ofmatchtable中的设备兼容性。
- clock-frequency表示I2C通信时钟为100KHz

实例

rx8010为EPSON公司的一款RTC芯片，其使用I2C进行通信，其驱动源码可以参考这里。下面简要分析一下这个驱动程序。

定义 struct i2c_driver

首先，其作为一个I2C设备，必须实现struct i2c_driver结构。

```
static const struct i2c_device_id rx8010_id[] = {
    { "rx8010", 0 },
    { }
```

```
};
MODULE_DEVICE_TABLE(i2c, rx8010_id);

static const struct of_device_id rx8010_of_match[] = {
  { .compatible = "epson,rx8010" },
  { }
};
MODULE_DEVICE_TABLE(of, rx8010_of_match);


static struct i2c_driver rx8010_driver = {
  .driver = {
    .name = "rtc-rx8010",
    .of_match_table = of_match_ptr(rx8010_of_match),
  },
  .probe    = rx8010_probe,
  .id_table  = rx8010_id,
};
```

之后通过module_i2c_driver(rx8010_driver)注册驱动程序。

实现probe接口

实现prome接口完成设备的初始化和反初始化操作。

```
static int rx8010_probe(struct i2c_client *client,
        const struct i2c_device_id *id)
{
  struct i2c_adapter *adapter = client->adapter;
  struct rx8010_data *rx8010;
  int err = 0;

  if (!i2c_check_functionality(adapter, I2C_FUNC_SMBUS_BYTE_DATA
    | I2C_FUNC_SMBUS_I2C_BLOCK)) {
    dev_err(&adapter->dev, "doesn't support required functionality\n");
    return -EIO;
  }

```

```
    rx8010 = devm_kzalloc(&client->dev, sizeof(struct rx8010_data),        <-
      | I2C_FUNC_SMBUS_I2C_BLOCK)) {
            GFP_KERNEL);
    if (!rx8010)
      return -ENOMEM;

    rx8010->client = client;
    i2c_set_clientdata(client, rx8010);

    err = rx8010_init_client(client);
    if (err)
      return err;

    .... ...

    rx8010->rtc = devm_rtc_device_register(&client->dev, client->name,    <--
      &rx8010_rtc_ops, THIS_MODULE);

    if (IS_ERR(rx8010->rtc)) {
      dev_err(&client->dev, "unable to register the class device\n");
      return PTR_ERR(rx8010->rtc);
    }

    return 0;
  }
```

- (1)可以看到rx8010使用的是SMbus通信协议，这里进行了适配器功能检测。
- (2)分配设备私有数据，并进行初始化。
- (3)rx8010为RTC设备，所以最终调用devm_rtc_device_register，将设备注册成为RTC设备。

设备通信

rx8010gettime和rx8010settime实现RTC时间的读取和设置操作，具体实现中，与设备通信时使用到的就是SMbus协议。比如，读取RTC时间的操作

```
err = i2c_smbus_read_i2c_block_data(rx8010->client, RX8010_SEC,
              7, date);
    if (err != 7)
      return err < 0 ? err : -EIO;
```

通过i2c_smbus_read_i2c_blockdata连续读取了开始寄存器RX8010SEC的7个字节，并将其存储到date数组中。