[网友问答5]i2c的设备树和驱动是如何匹配以及何时调用probe的?

原创 土豆居士 一口Linux 2020-12-01 09:00

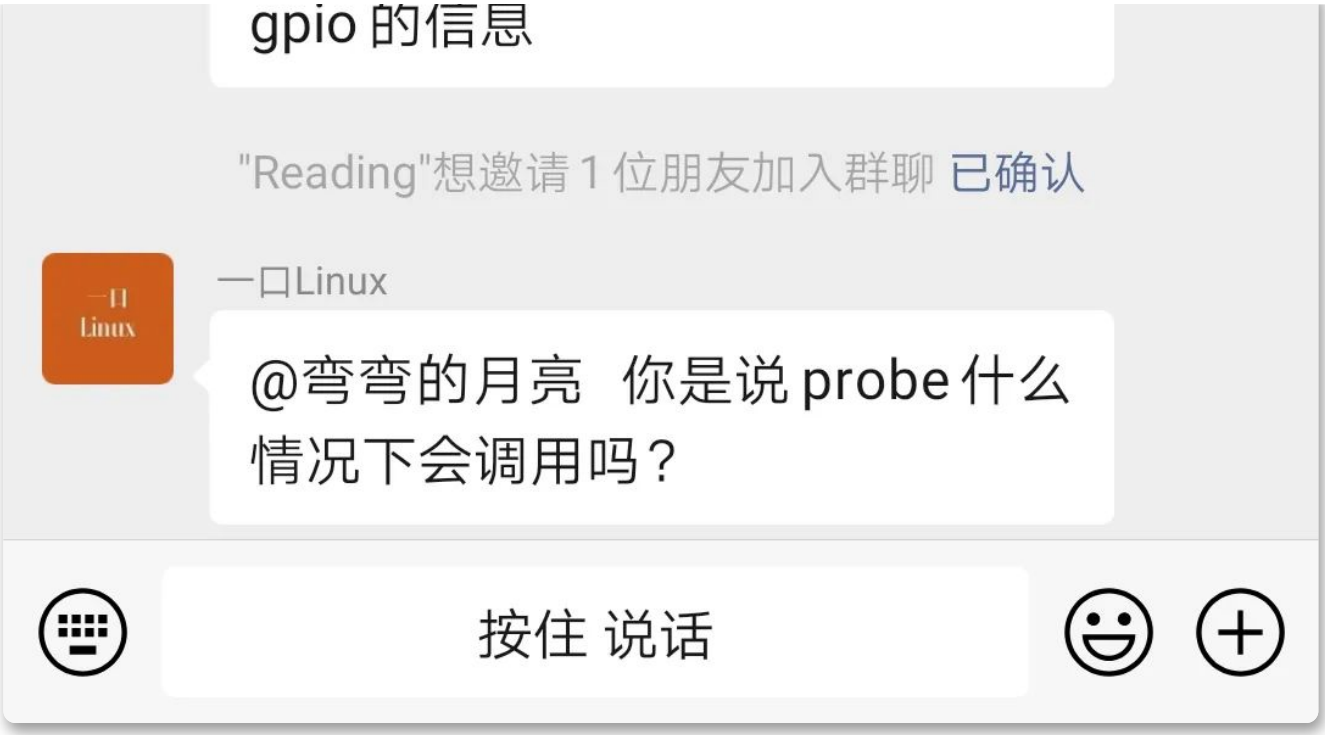
收录于合集

#粉丝问答 29 #Linux驱动 52 #i2c 2 #所有原创 194 #arm 22

一、粉丝提问

i2c的设备树和驱动是如何匹配以及何时调用probe的？粉丝手里的I2C外设是ov5640，一个摄像头。粉丝提问，一口君必须安排。


5:06
一口Linux技术讨论10群(288)
CikY.
这不是已经读 dts 的配置信息了么
弯弯的月亮
是可以读到设备树里边的信息，我搞不清楚是怎么匹配的
CikY.
那你可以去看一下 of_get_named_gpio() of_property_read_u32()函数的是实现
CikY.
核心是 of_node，你在 probe 的时候，这个 dts 的 node 父节点信息就已经放进去了
弯弯的月亮
是的，我的疑问是驱动匹配成功才会执行
弯弯的月亮
才会有你说的获取 rst 和 pwn 两个



二、问题分析

设备树信息如下：

```
ov5640: ov5640@3c {
 compatible = "ovti,ov5640";
 reg = <0x3c>;
 pinctrl-names = "default";
 pinctrl-0 = <&pinctrl_csi1
                          &csi_pwn_rst>;
 clocks = <&clks IMX6UL_CLK_CSI>;
 clock-names = "csi_mclk";
 pwn-gpios = <&gpio1 4 1>;
 rst-gpios = <&gpio1 2 0>;
 csi_id = <0>;
 mclk = <24000000>;
 mclk_source = <0>;
 status = "okay";
 port {
  ov5640_ep: endpoint {
   remote-endpoint = <&csi1_ep>;
  };
 };
};
```

驱动最重要的结构体如下：

```
00599: static const struct i2c_device_id ov5640_id[] = {
00600:     {"ov5640", 0},
00601:     {},
00602: };
00603: static const struct of_device_id of_ov5640_id[] = {
00604:     {.compatible="ovti,ov5640"},
00605:     {},
00606: };
00607:
00608: MODULE_DEVICE_TABLE(i2c, ov5640_id);
00609:
00610: static struct i2c_driver ov5640_i2c_driver = {
00611:     .driver = {
00612:         .owner = THIS_MODULE,
00613:         .name  = "ov5640",
00614:         .of_match_table = of_match_ptr(of_ov5640_id),
00615:     },
00616:     .probe  = ov5640_probe,
00617:     .remove = ov5640_remove,
00618:     .id_table = ov5640_id,
00619: };
```

ov5640_i2c_driver

要搞懂这个问题，我们需要有一些基础知识：

1.内核如何维护i2c总线

Linux内核维护很多总线，platform、usb、i2c、spi、pci等等，这个总线的架构在内核中都支持的很完善，内核通过以下结构体来维护总线：

```
struct bus_type {
 const char  *name;
 const char  *dev_name;
 struct device  *dev_root;
 struct device_attribute *dev_attrs; /* use dev_groups instead */
 const struct attribute_group **bus_groups;
 const struct attribute_group **dev_groups;
 const struct attribute_group **drv_groups;

 int (*match)(struct device *dev, struct device_driver *drv);
 int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
 int (*probe)(struct device *dev);
 int (*remove)(struct device *dev);
 void (*shutdown)(struct device *dev);
```

```
 int (*online)(struct device *dev);
 int (*offline)(struct device *dev);

 int (*suspend)(struct device *dev, pm_message_t state);
 int (*resume)(struct device *dev);

 const struct dev_pm_ops *pm;

 struct iommu_ops *iommu_ops;

 struct subsys_private *p;
 struct lock_class_key lock_key;
};
```

i2c对应总线结构体变量为i2c_bus_type，定义如下：

```
drivers/i2c/I2c-core.c
```

```
struct bus_type i2c_bus_type = {
 .name  = "i2c",
 .match  = i2c_device_match,
 .probe  = i2c_device_probe,
 .remove  = i2c_device_remove,
 .shutdown = i2c_device_shutdown,
 .pm  = &i2c_device_pm_ops,
};
```

其中：

1. i2c_device_match(),匹配总线维护的驱动链表和设备信息链表，如果其中名字完全相同，则返回true，否则false；
2. i2c_device_probe(),当我们注册一个i2c_drive或者i2c_client结构体时，会从对应的链表中查找节点，并通过i2c_device_match函数比较，如果匹配成功，则调用i2c_drive中定义的probe函数，即ov5640的ov5640_probe()函数；
3. remove：如果卸载i2c_drive或者i2c_client结构体，会调用该函数卸载对应的资源；
4. shutdown、pm是电源管理的接口，在此不讨论。

该结构体变量在函数i2c_init()中初始化：

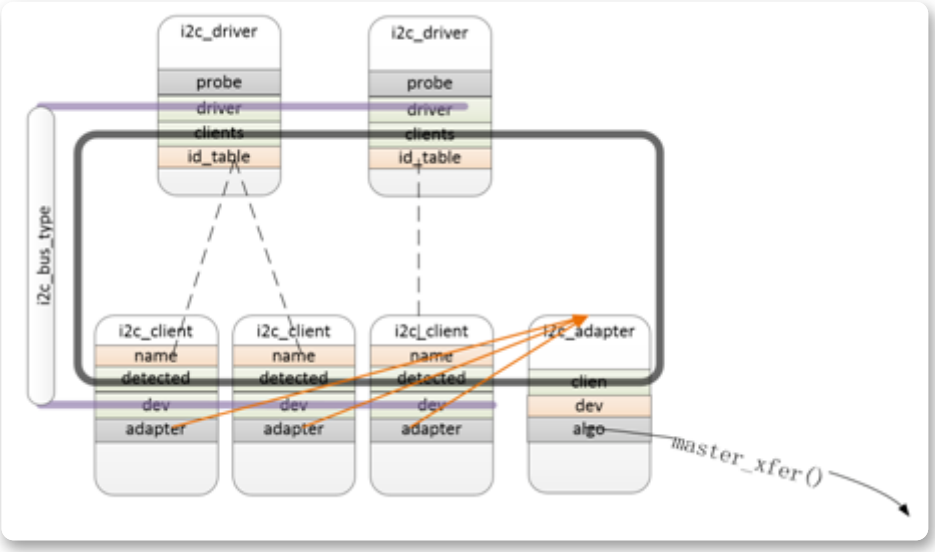
```
static int __init i2c_init(void)
```

```
{
 ……
 retval = bus_register(&i2c_bus_type);
 ……
}
```

i2c架构是通用架构，可支持多种不同的i2c控制器驱动。

2. i2c架构如何如何管理硬件信息和驱动?

不论哪一种总线，一定会维护两个链表，一个是驱动链表，一个是硬件信息链表。链表如下：



i2c总线的两个节点信息如下：

「struct i2c_driver」

```
struct i2c_driver {
 unsigned int class;

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

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

 /* driver model interfaces that don't relate to enumeration  */
```

```
    void (*shutdown)(struct i2c_client *);
    int (*suspend)(struct i2c_client *, pm_message_t mesg);
    int (*resume)(struct i2c_client *);

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

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

    struct device_driver driver;
    const struct i2c_device_id *id_table;

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

1. 当总线匹配驱动和硬件信息成功后就会调用其中的probe()函数；
2. struct device_driver driver，内核中注册的驱动模块，必须包含该类型的结构体成员。

「struct i2c_client」

成员	含义
unsigned short flags	从设备地址长度
unsigned short addr	从设备地址
char name[I2C_NAME_SIZE]	从设备地址名称
struct i2c_adapter *adapter	从设备地址对应的控制器驱动地址
struct device dev	注册到内核的每一个设备模块都需要先定义一个该结构体变量，对应struct device_driver driver
int irq	从设备地址往往会有一根中断线连接到SOC的中断控制器

成员	含义
struct list_head detected	链表

3. i2c_driver和i2c_client

1) i2c_driver如何注册

i2c_driver结构需要我们自己定义，然后通过函数i2c_register_driver()注册，将该结构体变量注册到i2c_driver链表，同时从i2c_client链表中查找是否有匹配的节点:

设备树情况下，会比较i2c_drive->driver->of_match_table->compatible和i2c_client->name，对应例子中的of_ov5640_id:

非设备树比较i2c_drive->id_table->name和i2c_client->name，对应例子中的ov5640_id:

```
static const struct i2c_device_id ov5640_id[] = {
	{"ov5640", 0},
	{},
};
```

代码中并没有直接调用函数i2c_register_driver()注册，而是使用了下面的这个宏:

```
01894: module_i2c_driver(ov5640_i2c_driver);
```

该宏定义如下:

include/linux/I2c.h

```
00553: #define module_i2c_driver(__i2c_driver) \
00554: 	module_driver(__i2c_driver, i2c_add_driver, \
00555: 			i2c_del_driver)
```

```
01203: #define module_driver(__driver, __register, __unregister, ...) \
01204: static int __init __driver##_init(void) \
01205: { \
01206:     return __register(&(__driver) , ##__VA_ARGS__); \
01207: } \
01208: module_init(__driver##_init); \
01209: static void __exit __driver##_exit(void) \
01210: { \
01211:     __unregister(&(__driver) , ##__VA_ARGS__); \
01212: } \
01213: module_exit(__driver##_exit);
```

该宏其实自动帮我生成了insmod和rmmod会用到宏module_init和module_exit，以及注册和注销i2c_driver结构体的代码。

如果看不明白宏，可以编写测试文件：test.c

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

#define module_driver(__driver, __register, __unregister, ...) \
static int __init __driver##_init(void) \
{ \
 return __register(&(__driver) , ##__VA_ARGS__); \
} \
module_init(__driver##_init); \
static void __exit __driver##_exit(void) \
{ \
 __unregister(&(__driver) , ##__VA_ARGS__); \
} \
module_exit(__driver##_exit);

module_i2c_drive(ov5640_i2c_driver);
```

预编译:

```
gcc -E test.c
```

得到宏替换后的结果:

```
static int __init ov5640_i2c_driver_init(void)
{
  return i2c_add_driver(&(ov5640_i2c_driver));
}
module_init(ov5640_i2c_driver_init);
static void __exit ov5640_i2c_driver_exit(void)
{
 i2c_del_driver(&(ov5640_i2c_driver));
}
module_exit(ov5640_i2c_driver_exit);;
```

内核中有大量的高效简洁的宏定义，Linux内核就是个宝库，里面有大量的优秀的代码，想提高自己的编程能力，就一定要多看代码，代码读百遍，其义自见。

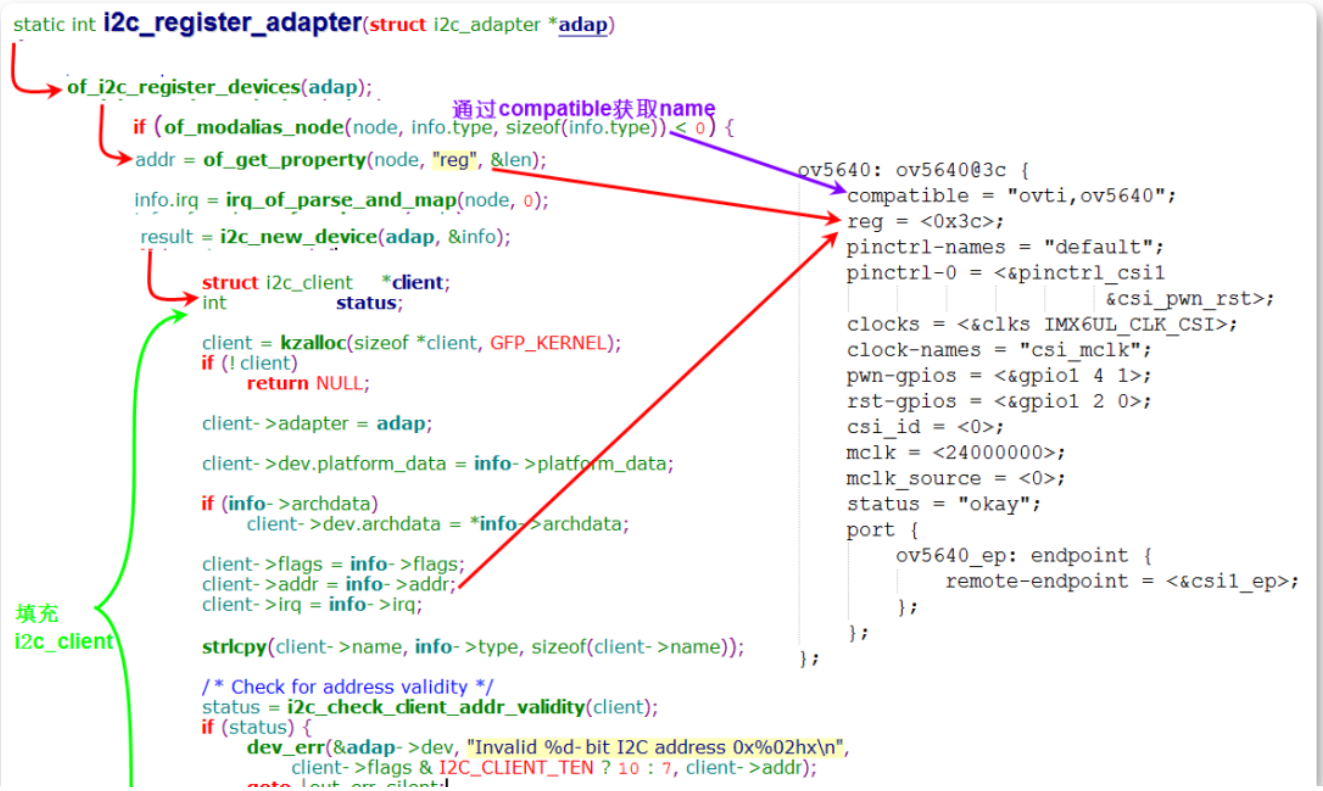
一口君认为，如果Linux代码都看不太明白，就不要自称精通C语言，充其量是会用C语言。

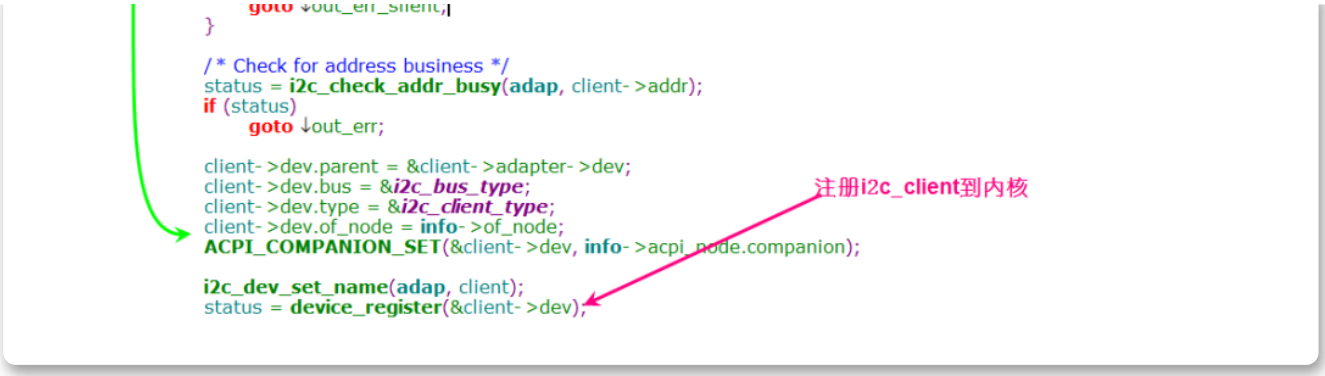
2）i2c_client如何生成(只讨论有设备树的情况)

在有设备树的情况下，i2c_client的生成是要在控制器驱动adapter注册情况下从设备树中枚举出来的。

i2c控制器有很多种，不同的厂家都会设计自己特有的i2c控制器，但是不论哪一个控制器，最终都会调用 i2c_register_adapter()注册控制器驱动。

i2c_client生成流程如下：





i2c_client

三、 i2c的设备树和驱动是如何匹配以及何时调用probe?

1. i2c的设备树和驱动是如何match，何时调用probe?

从第二章第3节可知，驱动程序中 module_i2c_drive()这个宏其实最终是调用i2c_add_driver(&(ov5640_i2c_driver));注册ov5640_i2c_driver结构体；当我们insmod加载驱动模块文件时，会调用i2c_add_driver()。

该函数定义如下：

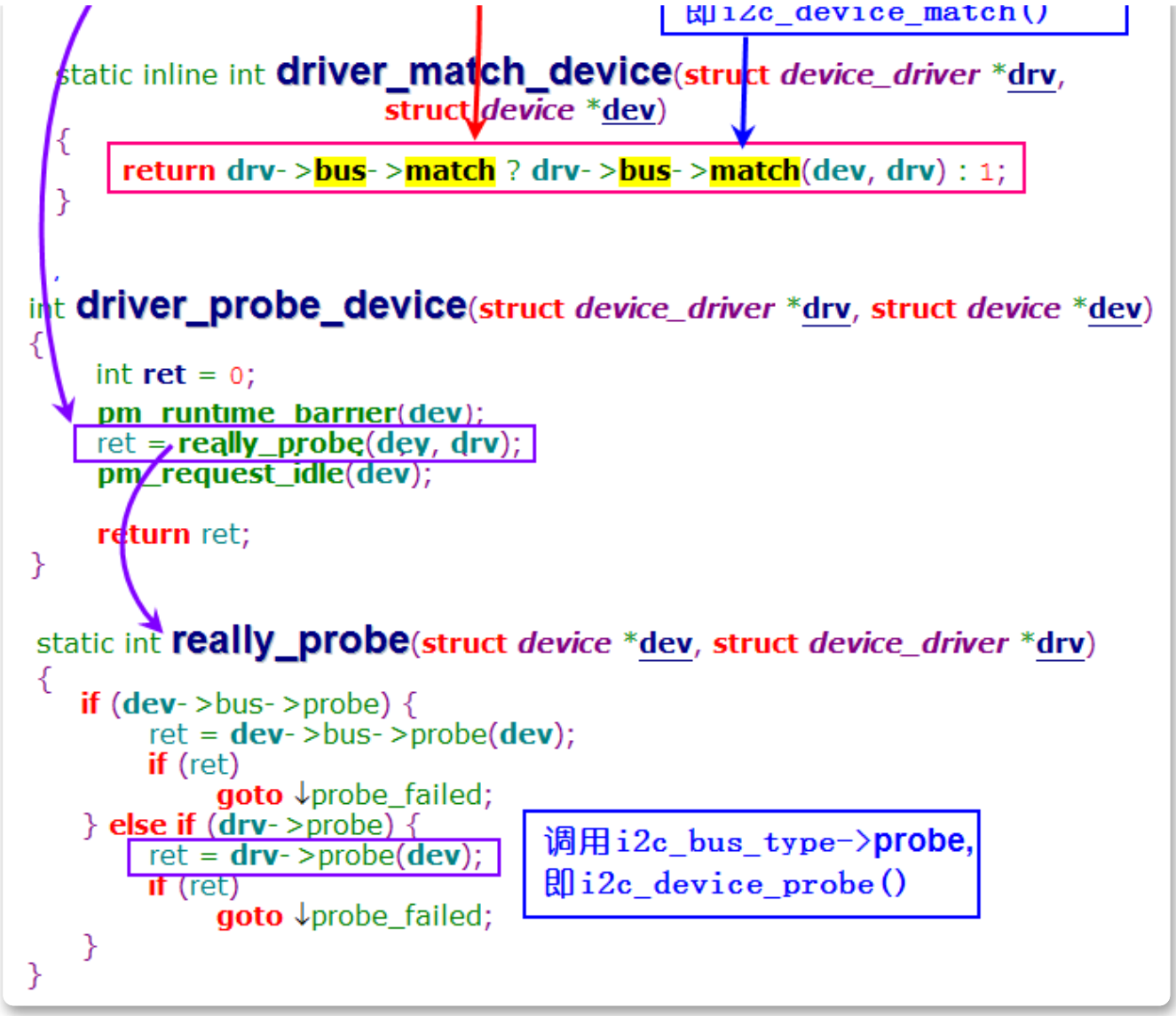
```
#define i2c_add_driver(driver) \
 i2c_register_driver(THIS_MODULE, driver)
```

下面我们来追踪i2c_register_driver()这个函数：

```
int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
{
    res = driver_register(&driver->driver);
    if (res)
        return res;

    i2c_for_each_dev(driver, __process_new_driver);
}


 int driver_register(struct device_driver *drv)
 {
      int ret;
      struct device_driver *other;

      ret = bus_add_driver(drv);

      return ret;
 } ? end driver_register ?


 int bus_add_driver(struct device_driver *drv)
 {
   error = driver_attach(drv);
 }

 int driver_attach(struct device_driver *drv)
 {
      return bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);
 }

int bus_for_each_dev(struct bus_type *bus, struct device *start,
              void *data, int (*fn)(struct device *, void *))
{
      struct klist_iter i;
      struct device *dev;
      int error = 0;

      klist_iter_init_node(&bus->p->klist_devices, &i,
                           (start ? &start->p->knode_bus : NULL));
      while ((dev = next_device(&i)) && !error)
           error = fn(dev, data);
      klist_iter_exit(&i);
      return error;
}

static int __driver_attach(struct device *dev, void *data)
{
    if (!driver_match_device(drv, dev))
        return 0;

    if (!dev->driver)
        driver_probe_device(drv, dev);
}
```

如果匹配不成功，则返回错误，不会调用probe函数

调用i2c_bus_type->match,



其中drv->bus就是我们之前所说的i2c_bus_type，上图中，分别调用了.match、.probe:

```
struct bus_type i2c_bus_type = {
 .name  = "i2c",
 .match  = i2c_device_match,
 .probe  = i2c_device_probe,
 .remove  = i2c_device_remove,
 .shutdown = i2c_device_shutdown,
 .pm  = &i2c_device_pm_ops,
};
```

下面我们来追一追这两个函数

2. i2c_device_match()

```
static int i2c_device_match(struct device *dev, struct device_driver *drv)
{
    struct i2c_client    *client = i2c_verify_client(dev);
    struct i2c_driver    *driver;
```


```
    if (! client)
        return 0;

    /* Attempt an OF style match */
    if (of_driver_match_device(dev, drv))
        return 1;

    /* Then ACPI style match */
    if (acpi_driver_match_device(dev, drv))
        return 1;

    driver = to_i2c_driver(drv);
    /* match on an id table if there is one */
    if (driver->id_table)
        return i2c_match_id(driver->id_table, client) != NULL;

    return 0;
} ? end i2c_device_match ?
```

```
static const struct i2c_device_id ov5640_id[] = {
    {"ov5640", 0},
    {},
};
```

```
static struct i2c_driver ov5640_i2c_driver = {
    .driver = {
        .owner = THIS_MODULE,
        .name  = "ov5640",
        .of_match_table = of_match_ptr(of_ov5640_id)
    },
    .probe  = ov5640_probe,
    .remove = ov5640_remove,
    .id_table = ov5640_id,
};
```

```
static inline int of_driver_match_device(struct device *dev,
                 const struct device_driver *drv)
{
    return of_match_device(drv->of_match_table, dev) != NULL;
}

const struct of_device_id *of_match_device(const struct of_device_id *matches,
                 const struct device *dev)
{
    if ((! matches) || (! dev->of_node))
        return NULL;
    return of_match_node(matches, dev->of_node);
}

const struct of_device_id *__of_match_node(const struct of_device_id *matches,
                 const struct device_node *node)
{
    const struct of_device_id *best_match = NULL;
    int score, best_score = 0;

    if (! matches)
        return NULL;

    for (; matches->name[0] || matches->type[0] || matches->compatible[0]; matches++) {
        score = __of_device_is_compatible(node, matches->compatible,
                     matches->type, matches->name);
        if (score > best_score) {
            best_match = matches;
            best_score = score;
        }
    }

    return best_match;
} ? end __of_match_node ?

static int __of_device_is_compatible(const struct device_node *device,
                 const char *compat, const char *type, const char *name)
{
    struct property *prop;
    const char *cp;
    int index = 0, score = 0;

    /* Compatible match has highest priority */
    if (compat && compat[0]) {
        prop = __of_find_property(device, "compatible", NULL);
        for (cp = of_prop_next_string(prop, NULL); cp;
             cp = of_prop_next_string(prop, cp), index++) {
            if (of_compat_cmp(cp, compat, strlen(compat)) == 0) {
                score = INT_MAX/2 - (index << 2);
                break;
            }
        }
        if (! score)
            return 0;
    }
}

#define of_compat_cmp(s1, s2, l)    strcasecmp((s1), (s2))

int strcasecmp(const char *s1, const char *s2)
{
    int c1, c2;

    do {
        c1 = tolower(*s1++);
        c2 = tolower(*s2++);
```

```
    } while (c1 == c2 && c1 != 0);
    return c1 - c2;
}
```

i2c_device_match

3. i2c_device_probe

如下图所示，通过driver->probe()调用到我们定义的 `struct i2c_driver ov5640_i2c_driver` 结构体变量中的ov5640_probe()函数：

```
static int i2c_device_probe(struct device *dev)
{
    struct i2c_client    *client = i2c_verify_client(dev);
    struct i2c_driver    *driver;
    int status;

    if (!client)
        return 0;

    driver = to_i2c_driver(dev->driver);
    if (!driver->probe || !driver->id_table)
        return -ENODEV;

    if (!device_can_wakeup(&client->dev))
        device_init_wakeup(&client->dev,
                   client->flags & I2C_CLIENT_WAKE);
    dev_dbg(dev, "probe\n");

    acpi_dev_pm_attach(&client->dev, true);
    status = driver->probe(client, i2c_match_id(driver->id_table, client));
    if (status)
        acpi_dev_pm_detach(&client->dev, true);

    return status;
} ? end i2c_device_probe ?
```

i2c_device_probe

【注意】 内核代码中大量使用到 `driver = to_i2c_driver(dev->driver);` 通过通用的结构体变量成员 `struct device_driver *driver` 来查找自己注册的xx_driver地址。

• END •

其他网友提问汇总

1. 两个线程，两个互斥锁，怎么形成一个死循环？

2. 一个端口号可以同时被两个进程绑定吗?

3. 一个多线程的简单例子让你看清线程调度的随机性

4. 粉丝提问|c语言：如何定义一个和库函数名一样的函数，并在函数中调用该库函数

推荐阅读

【1】嵌入式工程师到底要不要学习ARM汇编指令？必读

【2】Modbus协议概念最详细介绍必读

【3】嵌入式工程师到底要不要学习ARM汇编指令？

【4】如何用C语言操作sqlite3，一文搞懂

【5】4. 从0开始学ARM-ARM汇编指令其实很简单

进群，请加一口君个人微信，带你嵌入式入门进阶。

点击"**阅读原文**"查看更多分享，欢迎**点分享、收藏、点赞、在看。**

收录于合集 #粉丝问答 29

上一篇 · [粉丝问答6]子进程进程的父进程关系

阅读原文

喜欢此内容的人还喜欢

570个常用的Linux命令，1349页Linux命令速查手册（附PDF）

一口Linux