OV5640 驱动架构分析

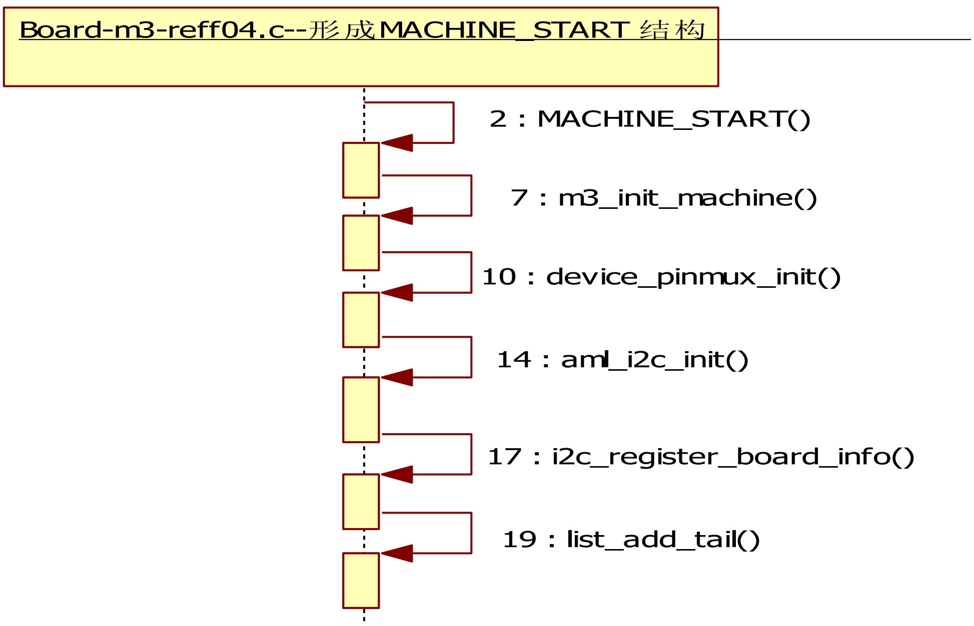
➢ Camera I2C 设备读入到 I2C_BUS_TYPE 下

1：把与 Camera 相关的 I2C 地址放到__i2c_board_list 链表

```
static struct i2c_board_info __initdata aml_i2c_bus_info[]
{
        {
                /*ov5642 i2c address is 0x78*/
            I2C_BOARD_INFO("ov5642_i2c",  0x78 >> 1),
            .platform_data = (void *)&video_ov5640_data,
        },
}
```

具体调用关系如下三图：

图一，在结构体 MACHINE_START 中有 m3_init_machine 指针，当调用 m3_init_machine 后，会顺序执行 知道把 Camera OV5640 相关的 I2C 地址信息放到__i2c_board_list 链表



2： __i2c_board_list 链表被读取到 i2C_BUS_TYPE

从 system.map 中可以看出与 Camera 相关的模块如下：

c003288c t __initcall_i2c_init2
…
c00328b8 t __initcall_aml_i2c_init3
…
c0032c18 t __initcall_i2c_dev_init6
…
c0032c28 t __initcall_videodev_init6
c0032c30 t __initcall_v4l2_i2c_drv_init6
c0032c34 t __initcall_v4l2_i2c_drv_init6

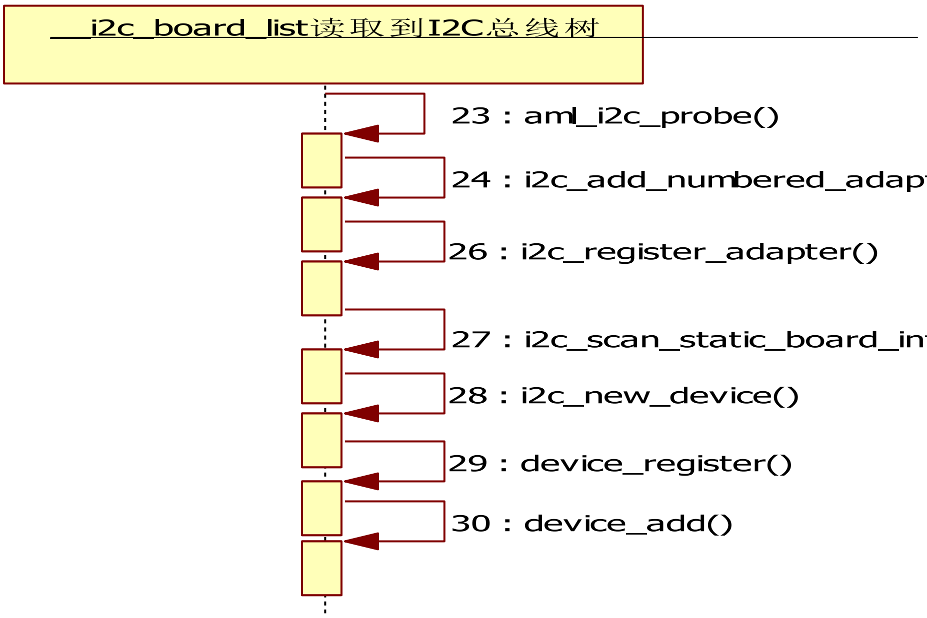
c0032d24 t __initcall_video_init6
c0032d28 t __initcall_video2_init6
主要分两部分 一部分与 I2C 相关，一部分与 Video 相关，一般一些宏定义在 autoconf.h 中得到体现；一般设备的启动次序如下：

1， 先建立和注册总线—如 platform_bus_type， i2c_bus_type
2， 读取设备列表到总线中去，链成链表
3， 驱动模块启动，然后配对到相应的设备 调用 Probe 函数 完成初始化

对于热插拔的设备

1， 设备插入系统，系统调用相应的处理程序 sbin/ueventd 并发布 Net Sock 事件
2， 同时，设备根据总线匹配到相应的驱动 调用驱动的 Probe 函数 完成初始化；

1， **__i2c_board_list 中都是一些 I2C 设备，这些设备在 aml-i2c 驱动 加载时 链接到相应的 i2c 总线树上；具体的流程如下**



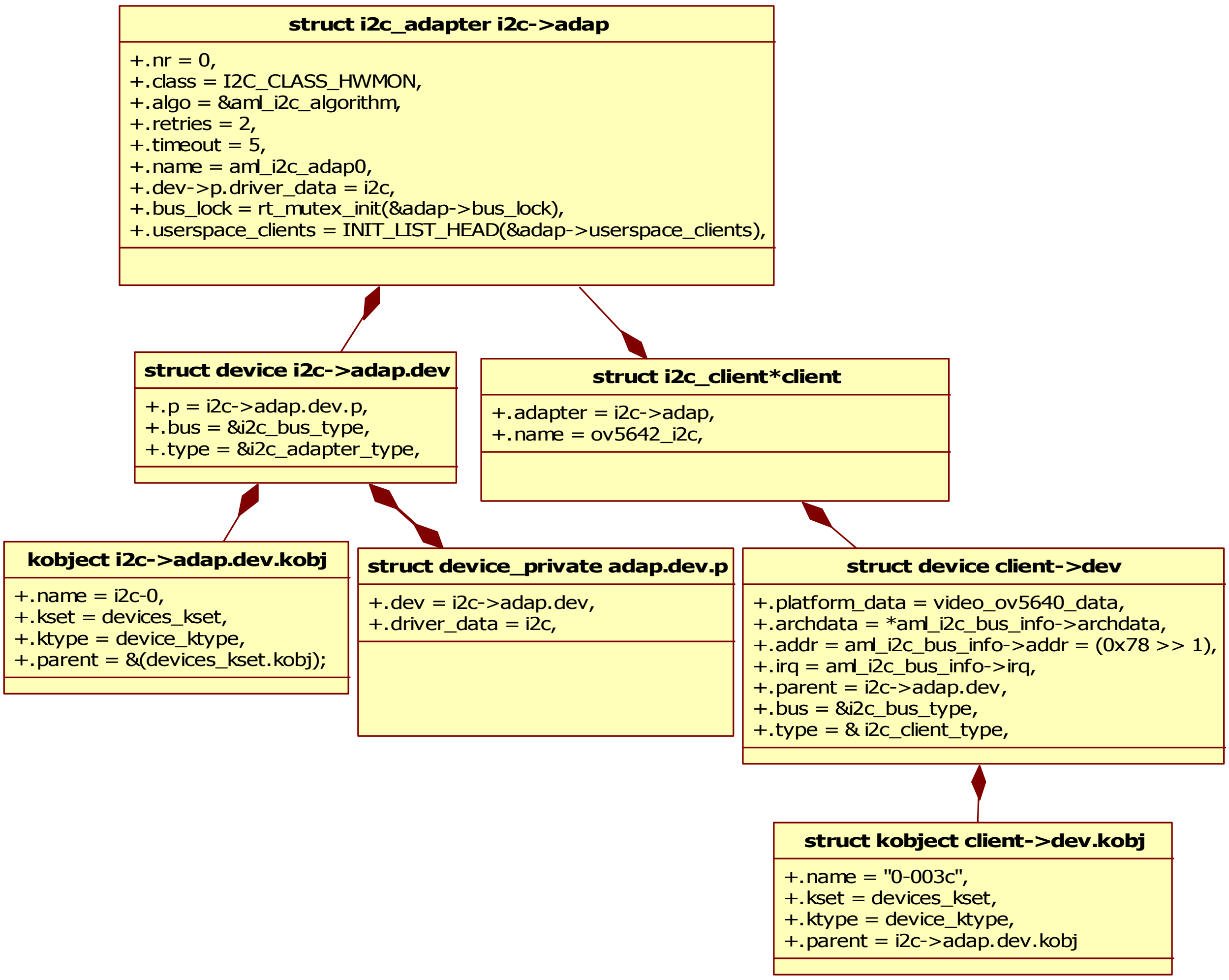
aml_i2c_probe() 会填充数据 adap
i2c_add_numberd_adapter()会把 adap 放到 i2c_adapter_idr 树中

现在重点分析 i2c_register_adapter:

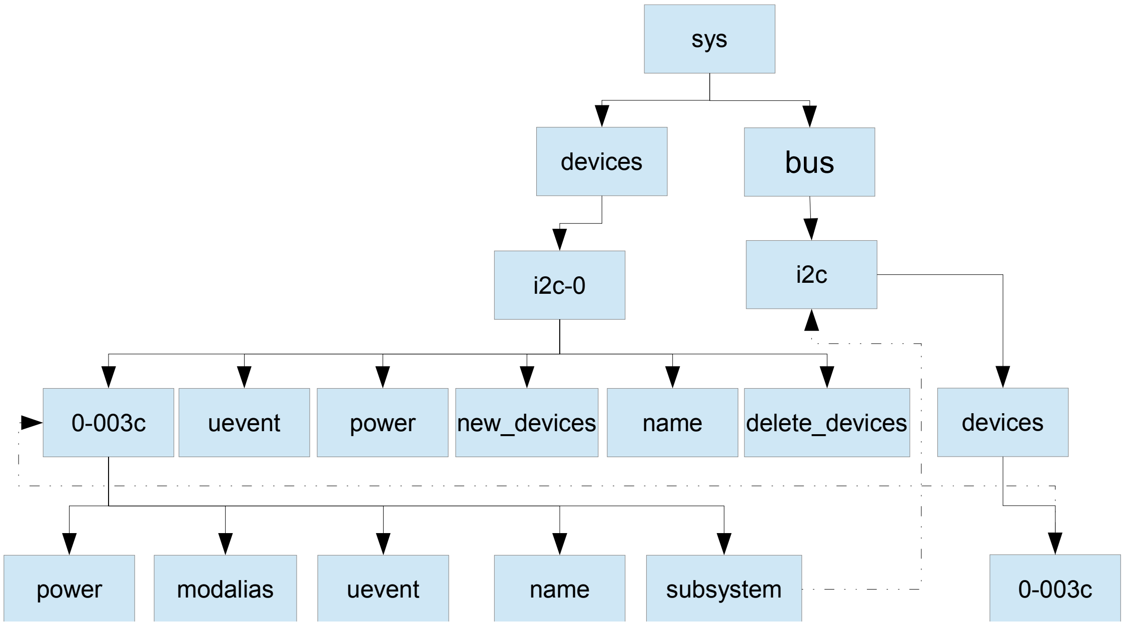
```
static int i2c_register_adapter(struct i2c_adapter *adap)
{
    …
    //注册 i2c-0 设备，创建对应的目录/sys/bus/i2c/devices/i2c-0
    dev_set_name(&adap->dev, "i2c-%d", adap->nr);
    adap->dev.bus = &i2c_bus_type;
    adap->dev.type = &i2c_adapter_type;
    res = device_register(&adap->dev);
    …
    //加载 i2c 设备列表
```

```
    i2c_scan_static_board_info(adap);
    …
    //适配驱动，启动 probe
    dummy = bus_for_each_drv(&i2c_bus_type, NULL, adap,
                 __process_new_adapter);
    …
    return 0;
}
int device_register(struct device *dev)
{
    device_initialize(dev);
    return device_add(dev);
}
```

与 adap 关联的设备注册到 i2c_bus_type 上去



在 Sys 下形成以下树状结构：



device_add 是一个非常关键的函数，它把一个设备连接到总线树，并且创建了 sys 下面给中目录与文件，现在来简单分析 device_add

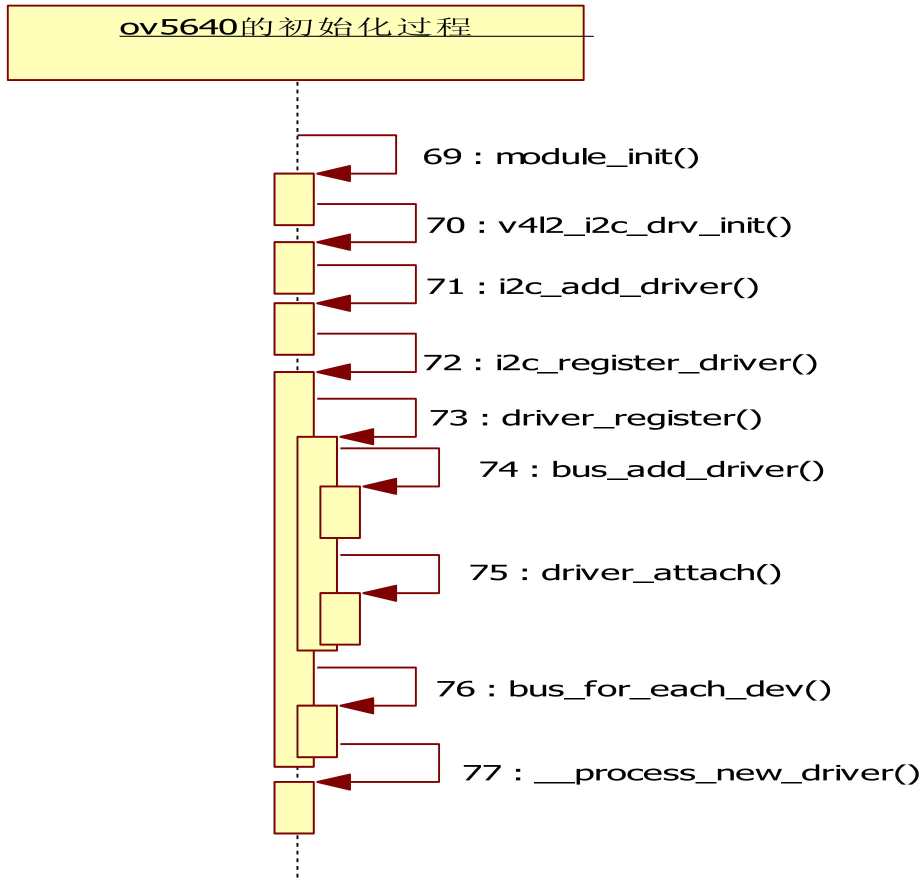
```
int device_add(struct device *dev)
{
    /* we require the name to be set before, and pass NULL */
    //1, kobj_kset_join(kobj); kobj 将会链入 devices_kset
    //2，kobj 在父 kobj 下面创建相应目录
    error = kobject_add(&dev->kobj, dev->kobj.parent, NULL);
    //创建了文件 uevent_attr
    error = device_create_file(dev, &uevent_attr);
    //创建特性
    error = device_add_attrs(dev);
   //klist_add_tail(&dev->p->knode_bus, &bus->p->klist_devices);
   //把 dev 加到总线下的 klist_devices 列表
   error = bus_add_device(dev);
    if (error)
        goto BusError;
    error = dpm_sysfs_add(dev);
    //把设备加入到 list_add_tail(&dev->power.entry, &dpm_list); 电源管理列表
    device_pm_add(dev);

    /* Notify clients of device addition.  This call must come
     * after dpm_sysf_add() and before kobject_uevent().
     */
    //在总线上发出通知
    if (dev->bus)
        blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
                         BUS_NOTIFY_ADD_DEVICE, dev);
```

```
    //发送UEVENT 事件
    kobject_uevent(&dev->kobj, KOBJ_ADD);
    //配对 然后启动 Probe事件，然后加入到设备驱动链表
    bus_probe_device(dev);
    //加入到父kobj链表
    if (parent)
        klist_add_tail(&dev->p->knode_parent,
                   &parent->p->klist_children);
    put_device(dev);
    return error;
}
```

➢ Camera 驱动的初始化---以 OV5640 为例

OV5640 初始化的大致流程如下：

现 在 来 具 体 分 析 一 下 OV5640 的 初 始 化 过 程 ：



1， 首先在 ov5640.c 中并没有 module_init 的驱动入口函数，那么这个驱动入口函数放在哪儿了呢？原来把驱动入口函数放在了 V4L2-i2c-drv.h 里面了 module_init(v4l2_i2c_drv_init);就是 Ov5640 的入口函数

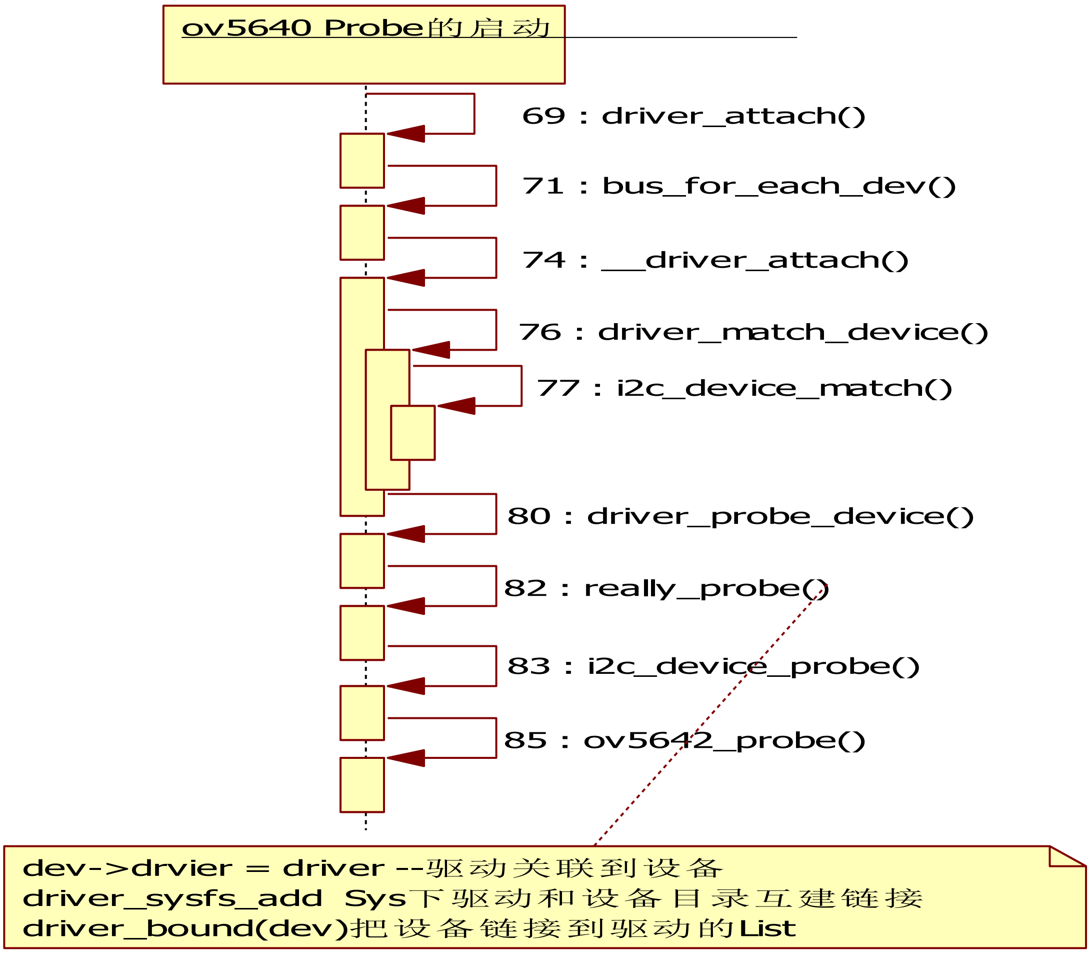
在 v4l2_i2c_drv_init 给 i2c_driver v4l2_i2c_driver 赋值，并且用 i2c_add_driver 注册驱动；而具体 ov5640 与 v4l2_i2c_drv_init 之间是通过 v4l2_i2c_data 关联起来的。其中在 v4l2_i2c_drv.h 中定义了 v4l2_i2c_data 变量，而在 ov5640 中对这个变量进行了赋值。

```
static struct v4l2_i2c_driver_data v4l2_i2c_data = {
    .name = "ov5642",
    .probe = ov5642_probe,
    .remove = ov5642_remove,
    .id_table = ov5642_id,
};
static const struct i2c_device_id ov5642_id[] = {
    { "ov5642_i2c", 0 },
    { }
```

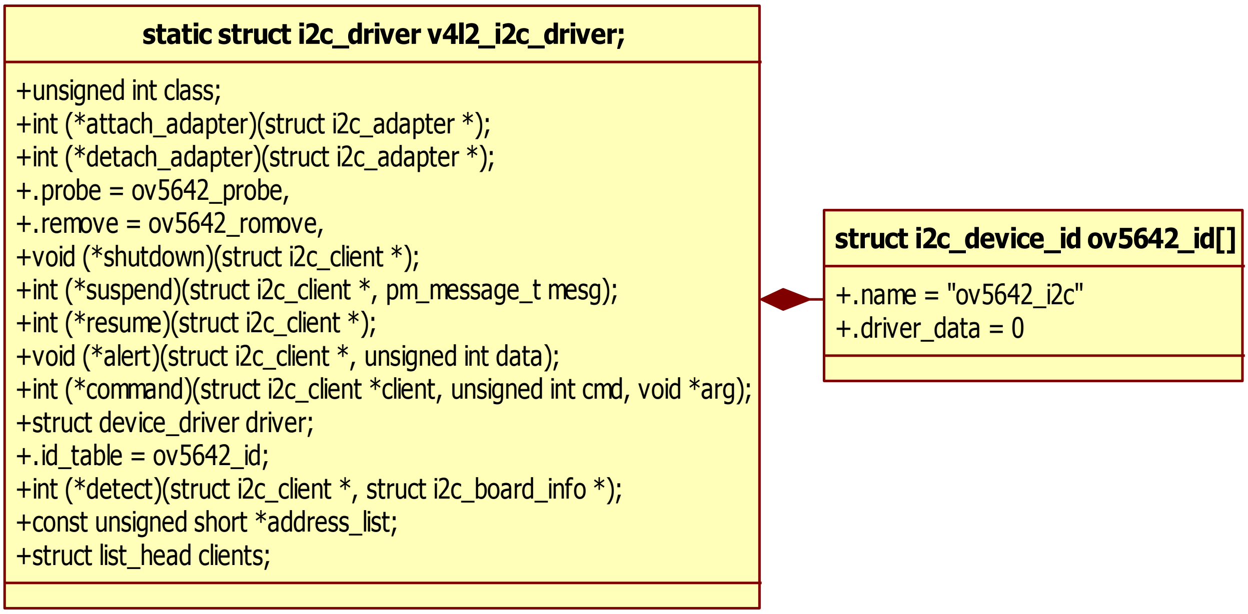
```
    };
```
现在重点分析一下bus_add_driver

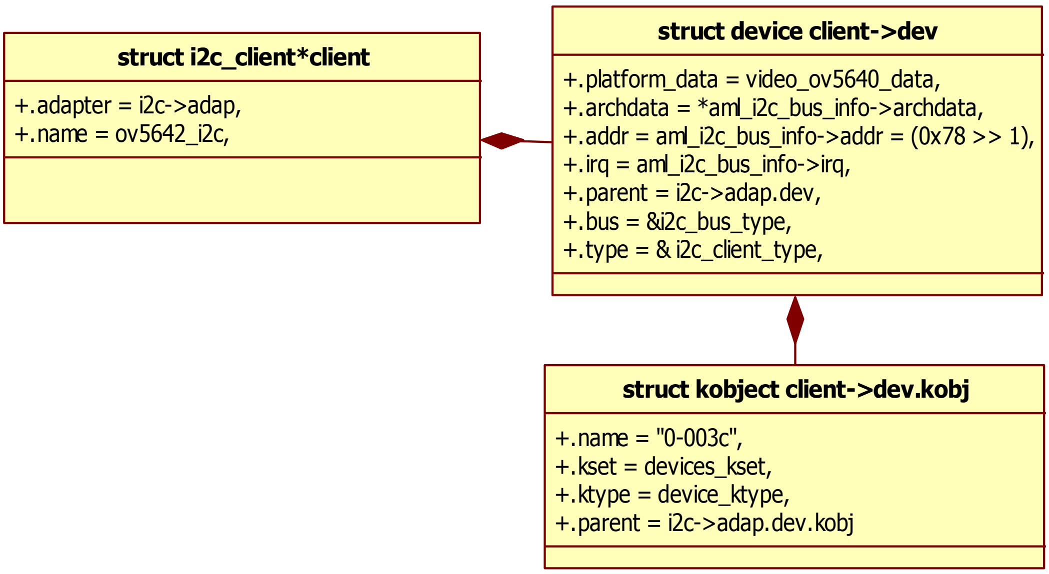
```
int bus_add_driver(struct device_driver *drv)
{
    …
    //在i2c bus的drivers目录下生成ov5642目录，并链入i2c总线链表
    error = kobject_init_and_add(&priv->kobj, &driver_ktype, NULL,
                     "%s", drv->name);
    //这个是配对函数，会启动驱动对用的Probe函数
    driver_attach(drv);
    //
    klist_add_tail(&priv->knode_bus, &bus->p->klist_drivers);
    module_add_driver(drv->owner, drv);

    error = driver_create_file(drv, &driver_attr_uevent);
    if (error) {
        printk(KERN_ERR "%s: uevent attr (%s) failed\n",
            __func__, drv->name);
    }
    error = driver_add_attrs(bus, drv);
    if (error) {
        /* How the hell do we get out of this pickle? Give up */
        printk(KERN_ERR "%s: driver_add_attrs(%s) failed\n",
            __func__, drv->name);
    }

    if (!drv->suppress_bind_attrs) {
        error = add_bind_files(drv);
        if (error) {
            /* Ditto */
            printk(KERN_ERR "%s: add_bind_files(%s) failed\n",
                __func__, drv->name);
        }
    }

    kobject_uevent(&priv->kobj, KOBJ_ADD);
    return 0;
}
//因为dev->type = i2c_client_type 因此不做处理
//注：如dev->type = i2c_adapter_type 则需要做i2c_do_add_adapter
bus_for_each_dev
__process_new_driver
```

//这个是配对函数，会启动驱动对用的Probe函数 对driver_attach(drv)进行具体分析：



关于配对：






```
static int i2c_device_match(struct device *dev, struct device_driver *drv)
{
    …
    return i2c_match_id(driver->id_table, client) != NULL;
    return 0;
}
static const struct i2c_device_id *i2c_match_id(const struct i2c_device_id *id,
                        const struct i2c_client *client)
{
    while (id->name[0]) {
```

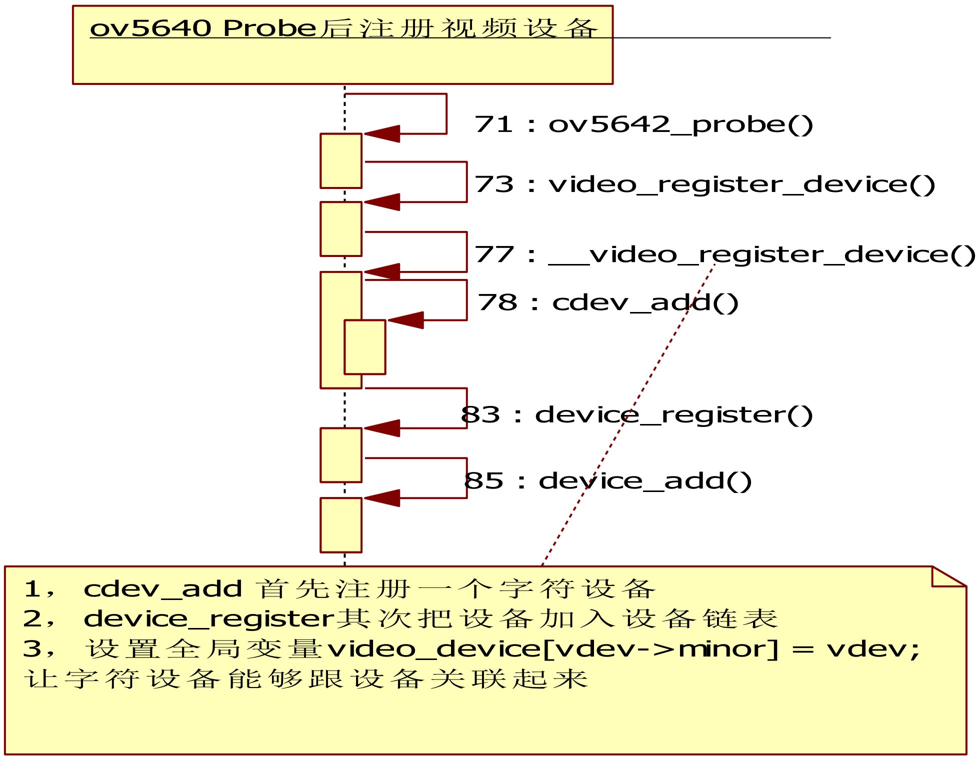
```
        if (strcmp(client->name, id->name) == 0)
            return id;
        id++;
    }
    return NULL;
}
```

由于 client->name ==ov5642_i2c = ov5642-id->name 因此配对成功;

因此执行 dev->bus->probe 因此执行

```
static int i2c_device_probe(struct device *dev)
{
    …
    //也就是执行 ov5642_probe(client, ov5642_id);
    status = driver->probe(client, i2c_match_id(driver->id_table, client));
    …
    return status;
}

static int ov5642_probe(struct i2c_client *client,
            const struct i2c_device_id *id)
{
    //初始化 v4l2_subdev,
    v4l2_i2c_subdev_init(sd, client, &ov5642_ops);
    …
    //初始化视频设备 vdev
    memcpy(t->vdev, &ov5642_template, sizeof(*t->vdev));
    //注册视频设备
    err = video_register_device(t->vdev, VFL_TYPE_GRABBER, video_nr);
    …
}
```

视频设备被注册会生成设备文件:/dev/video0



对应的 Device 目录

/sys/devices/virtual/video4linux/video0

dev ，index，name，power，subsystem，uevent 对应的 cdev 81:0

power, uevent,

// 创建 name, index

device_add_attrs(dev);

//创建 dev

device_create_file(dev, &devt_attr);

//创建 subsystem

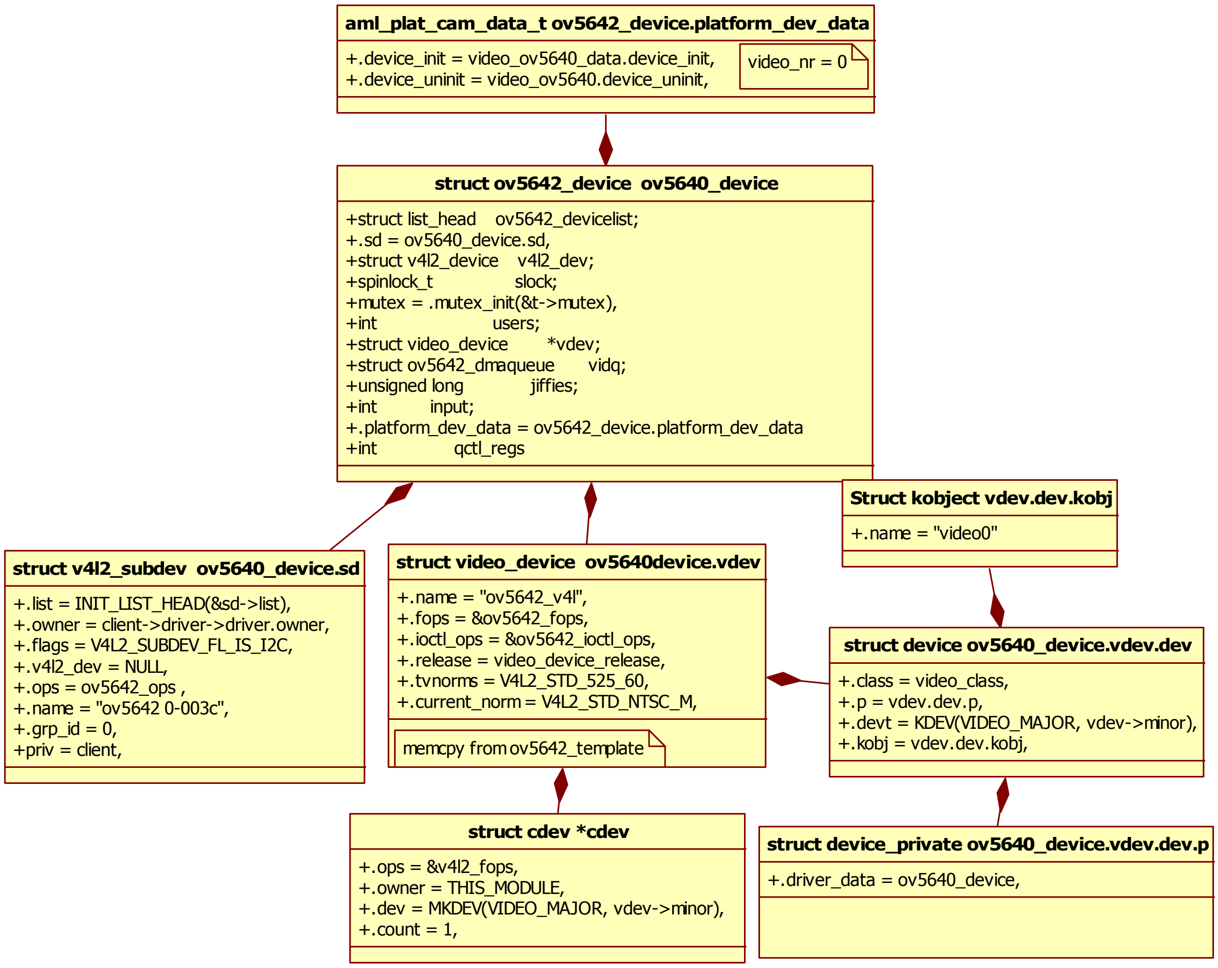
device_add_class_symlinks

对应的设备文件/dev/video0

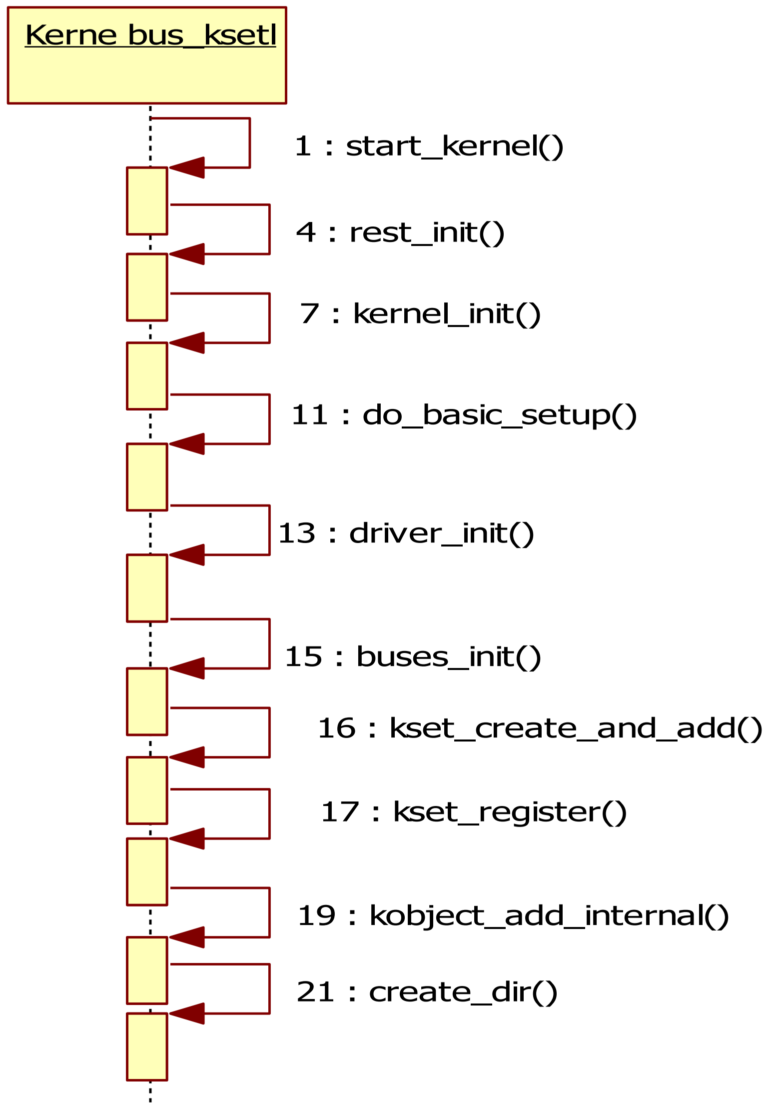
在 class 创建链接/sys/class/video4linux/video0

//创建文件/sys/dev/char/81:0 -> ../../devices/virtual/video4linux/video0

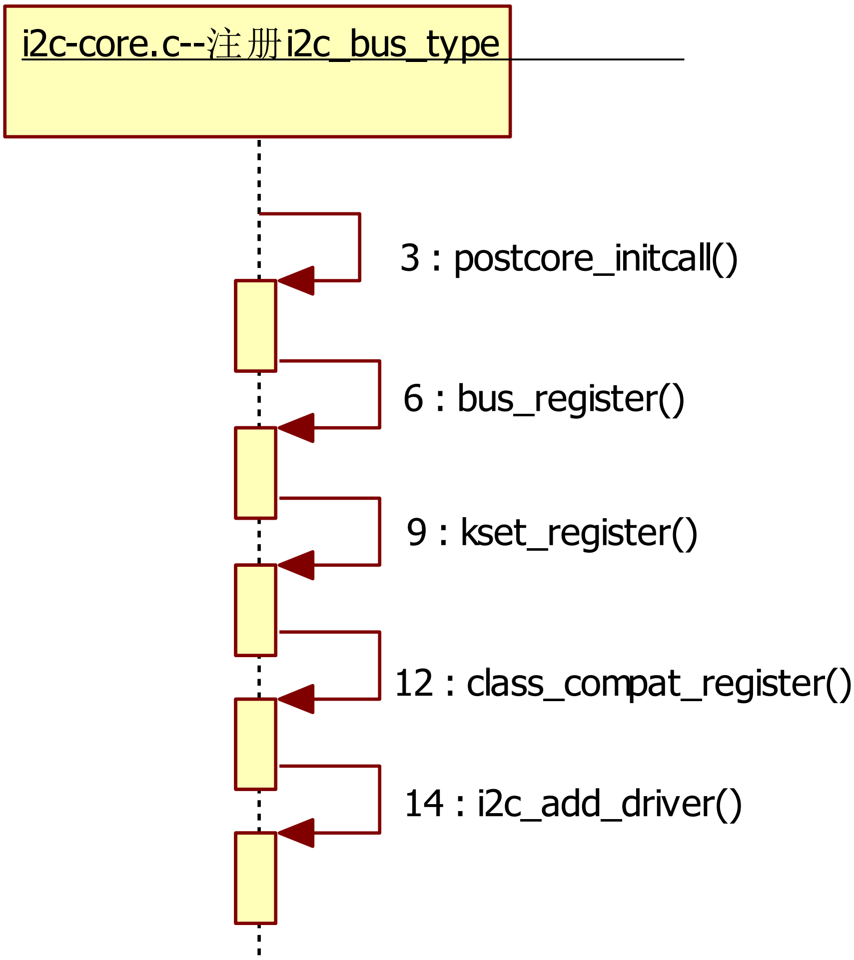
device_create_sys_dev_entry


aml_plat_cam_data_t ov5642_device.platform_dev_data
+.device_init = video_ov5640_data.device_init,
+.device_uninit = video_ov5640.device_uninit,
video_nr = 0
struct ov5642_device ov5640_device
+struct list_head ov5642_devicelist;
+.sd = ov5640_device.sd,
+struct v4l2_device v4l2_dev;
+spinlock_t slock;
+mutex = .mutex_init(&t->mutex),
+int users;
+struct video_device *vdev;
+struct ov5642_dmaqueue vidq;
+unsigned long jiffies;
+int input;
+.platform_dev_data = ov5642_device.platform_dev_data
+int qctl_regs
Struct kobject vdev.dev.kobj
+.name = "video0"
struct v4l2_subdev ov5640_device.sd
+.list = INIT_LIST_HEAD(&sd->list),
+.owner = client->driver->driver.owner,
+.flags = V4L2_SUBDEV_FL_IS_I2C,
+.v4l2_dev = NULL,
+.ops = ov5642_ops ,
+.name = "ov5642 0-003c",
+.grp_id = 0,
+priv = client,
struct video_device ov5640device.vdev
+.name = "ov5642_v4l",
+.fops = &ov5642_fops,
+.ioctl_ops = &ov5642_ioctl_ops,
+.release = video_device_release,
+.tvnorms = V4L2_STD_525_60,
+.current_norm = V4L2_STD_NTSC_M,
memcpy from ov5642_template
struct device ov5640_device.vdev.dev
+.class = video_class,
+.p = vdev.dev.p,
+.devt = KDEV(VIDEO_MAJOR, vdev->minor),
+.kobj = vdev.dev.kobj,
struct cdev *cdev
+.ops = &v4l2_fops,
+.owner = THIS_MODULE,
+.dev = MKDEV(VIDEO_MAJOR, vdev->minor),
+.count = 1,
struct device_private ov5640_device.vdev.dev.p
+.driver_data = ov5640_device,

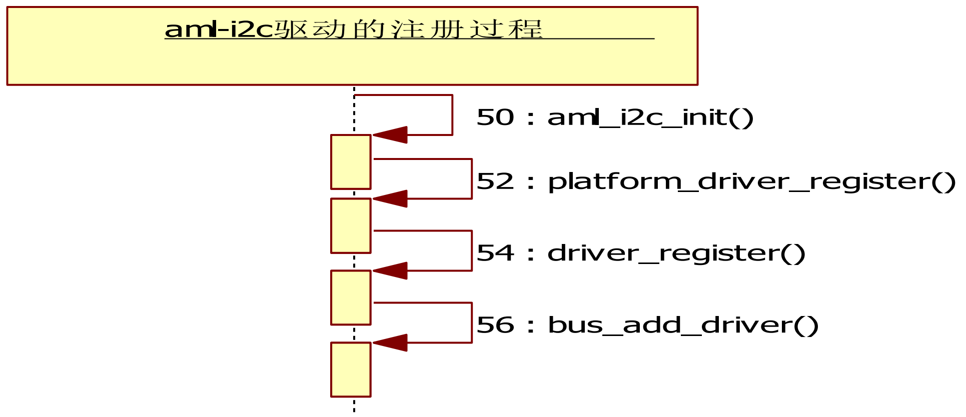
➢ i2c_bus_type 注册的过程



1， 创建了对应的/sys/class/i2c-adapter/ 目录
2， 创建了对应的/sys/bus/i2c/drivers/dummy 目录
m3_init_machine 被调用流程


i2c-core.c--注册i2c_bus_type
3 : postcore_initcall()
6 : bus_register()
9 : kset_register()
12 : class_compat_register()
14 : i2c_add_driver()

➢ platform_driver ---aml-i2c 在 Linux 启动时被挂接在 platform 总线上



平台驱动的结构如下:

```
struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*resume)(struct platform_device *);
    struct device_driver driver;
    const struct platform_device_id *id_table;
};
```

aml-i2c 驱动的结构初始化如下:

```
static struct platform_driver aml_i2c_driver = {
    .probe = aml_i2c_probe,
    .remove = aml_i2c_remove,
    .driver = {
            .name = "aml-i2c",
            .owner = THIS_MODULE,
    },
};
struct device_driver {
    const char      *name;
    struct bus_type     *bus;
    struct module       *owner;
    const char      *mod_name;  /* used for built-in modules */
    bool suppress_bind_attrs;   /* disables bind/unbind via sysfs */
    int (*probe) (struct device *dev);
    int (*remove) (struct device *dev);
    void (*shutdown) (struct device *dev);
    int (*suspend) (struct device *dev, pm_message_t state);
    int (*resume) (struct device *dev);
```

```
    const struct attribute_group **groups;
    const struct dev_pm_ops *pm;
    struct driver_private *p;
};
```

这一系列调用的重点是 bus_add_driver();

```
int bus_add_driver(struct device_driver *drv)
{
    …
    priv->kobj.kset = bus->p->drivers_kset;
    //在/sys/bus/platform/drivers 目录下创建aml-i2c
    error = kobject_init_and_add(&priv->kobj, &driver_ktype, NULL,
                     "%s", drv->name);
    //drv加载时 如找到相应的设备则启动drv->probe
    if (drv->bus->p->drivers_autoprobe) {
        error = driver_attach(drv);
    }
    //把驱动链入总线的klist_drivers
    klist_add_tail(&priv->knode_bus, &bus->p->klist_drivers);
    //aml-i2c驱动没有设置module
    module_add_driver(drv->owner, drv);
    //添加uevent特性文件
    error = driver_create_file(drv, &driver_attr_uevent);
    //platform没有driver_attrs因此不做操作
    error = driver_add_attrs(bus, drv);
    //driver 没有suppress_bind_attrs 因此不做什么
    if (!drv->suppress_bind_attrs) {
        error = add_bind_files(drv);
        if (error) {
            /* Ditto */
            printk(KERN_ERR "%s: add_bind_files(%s) failed\n",
                __func__, drv->name);
        }
    }
    kobject_uevent(&priv->kobj, KOBJ_ADD);
    return 0;
}
```

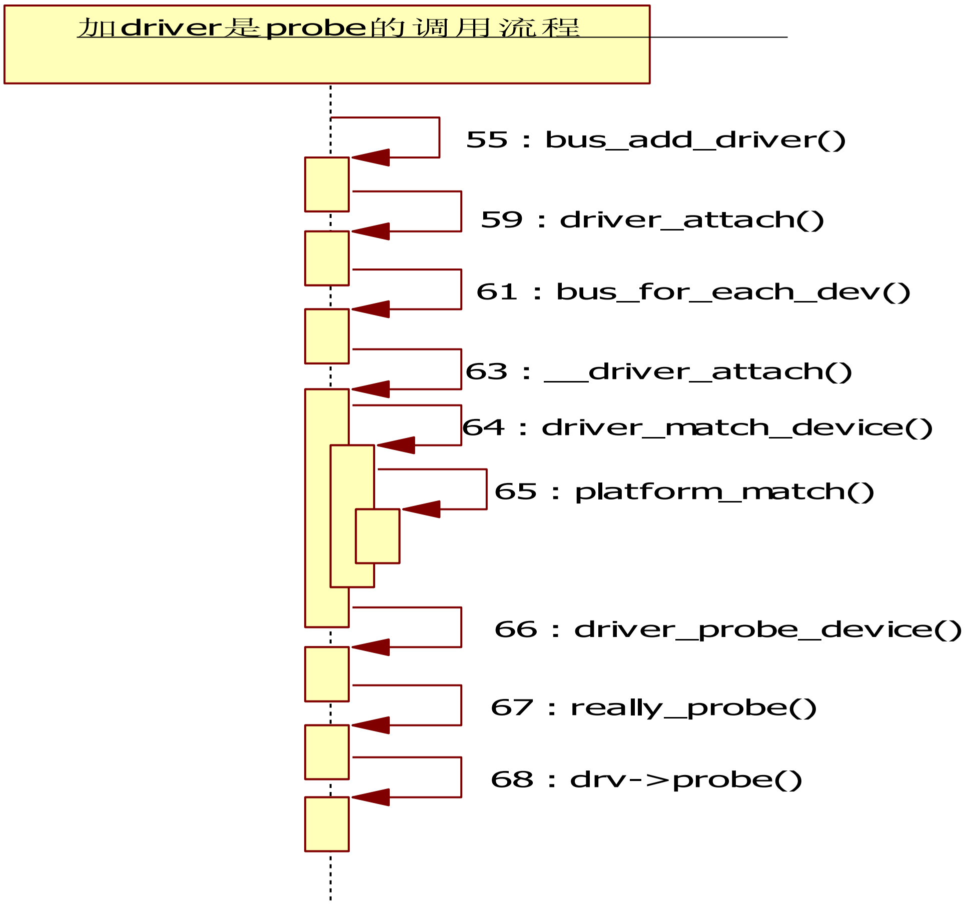
//drv加载时 如找到相应的设备则启动drv->probe流程

probe时

```
dev->driver = drv;
klist_add_tail(&dev->p->knode_driver, &dev->driver->p->klist_devices);
/sys/bus/platform/drivers/aml-i2c/aml-i2c.0/driver->
../../../bus/platform/drivers/aml-i2c
```

/sys/bus/platform/drivers/aml-i2c/aml-i2c.0 -> ../../../../devices/platform/aml-i2c.0

Driver Probe 的过程



注：为了知道 I2C 更具体的内容，则需要对 probe 函数 进行进一步的仔细分析。

```
static int aml_i2c_probe(struct platform_device *pdev)
{
    struct aml_i2c_platform *plat = pdev->dev.platform_data;
    struct resource *res;
    struct aml_i2c *i2c = kzalloc(sizeof(struct aml_i2c), GFP_KERNEL);
    i2c->ops = &aml_i2c_m1_ops;
    /*master a or master b*/
    i2c->master_no = plat->master_no;
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    i2c->master_regs = (struct aml_i2c_reg_master __iomem*)(res->start);
    printk("master_no = %d, resource = %x, maseter_regs=\n", i2c->master_no, res, i2c->master_regs);
```

```
    BUG_ON(!i2c->master_regs);
    BUG_ON(!plat);
    aml_i2c_set_platform_data(i2c, plat);

    /*lock init*/
      mutex_init(&i2c->lock);

      /*setup adapter*/
      i2c->adap.nr = pdev->id==-1? 0: pdev->id;
      i2c->adap.class = I2C_CLASS_HWMON;
      i2c->adap.algo = &aml_i2c_algorithm;
      i2c->adap.retries = 2;
      i2c->adap.timeout = 5;
     //memset(i2c->adap.name, 0 , 48);
      sprintf(i2c->adap.name, ADAPTER_NAME"%d", i2c->adap.nr);
      i2c_set_adapdata(&i2c->adap, i2c);

    ret = i2c_add_numbered_adapter(&i2c->adap);
    if (ret < 0)
    {
            dev_err(&pdev->dev, "Adapter %s registration failed\n",

              i2c->adap.name);
          kzfree(i2c);
          return -1;
    }
    dev_info(&pdev->dev, "add adapter %s(%x)\n", i2c->adap.name, &i2c-
>adap);

      /*need 2 different speed in 1 adapter, add a virtual one*/
      if(plat->master_i2c_speed2){
          i2c->master_i2c_speed2 = plat->master_i2c_speed2;
           /*setup adapter 2*/
           i2c->adap2.nr = i2c->adap.nr+1;
           i2c->adap2.class = I2C_CLASS_HWMON;
           i2c->adap2.algo = &aml_i2c_algorithm_s2;
           i2c->adap2.retries = 2;
           i2c->adap2.timeout = 5;
         //memset(i2c->adap.name, 0 , 48);
                     sprintf(i2c->adap2.name,  ADAPTER_NAME"%d",  i2c-
>adap2.nr);
           i2c_set_adapdata(&i2c->adap2, i2c);
           ret = i2c_add_numbered_adapter(&i2c->adap2);
           if (ret < 0)
```

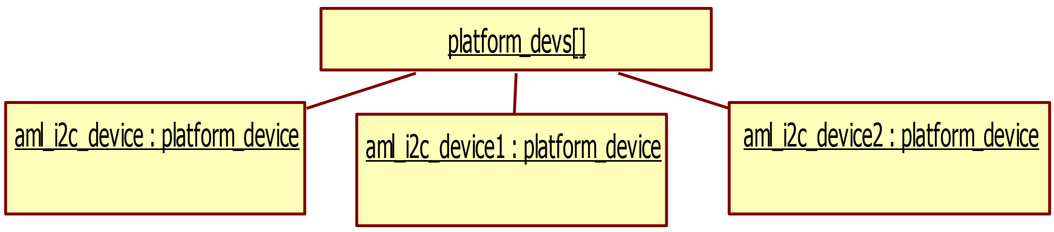
```
            {
                        dev_err(&pdev->dev, "Adapter %s registration
failed\n",
                i2c->adap2.name);
              i2c_del_adapter(&i2c->adap);
                kzfree(i2c);
                return -1;
            }
                      dev_info(&pdev->dev, "add adapter %s\n", i2c-
>adap2.name);
      }
    dev_info(&pdev->dev, "aml i2c bus driver.\n");

      /*setup class*/
    i2c->cls.name = kzalloc(NAME_LEN, GFP_KERNEL);
      if(i2c->adap.nr)
        sprintf(i2c->cls.name, "i2c%d", i2c->adap.nr);
      else
        sprintf(i2c->cls.name, "i2c");
      i2c->cls.class_attrs = i2c_class_attrs;
    ret = class_register(&i2c->cls);
    if(ret)
       printk(" class register i2c_class fail!\n");

    return 0;
}
```

- **platform_device 在 Linux 启动时被挂接在 platform 总线**

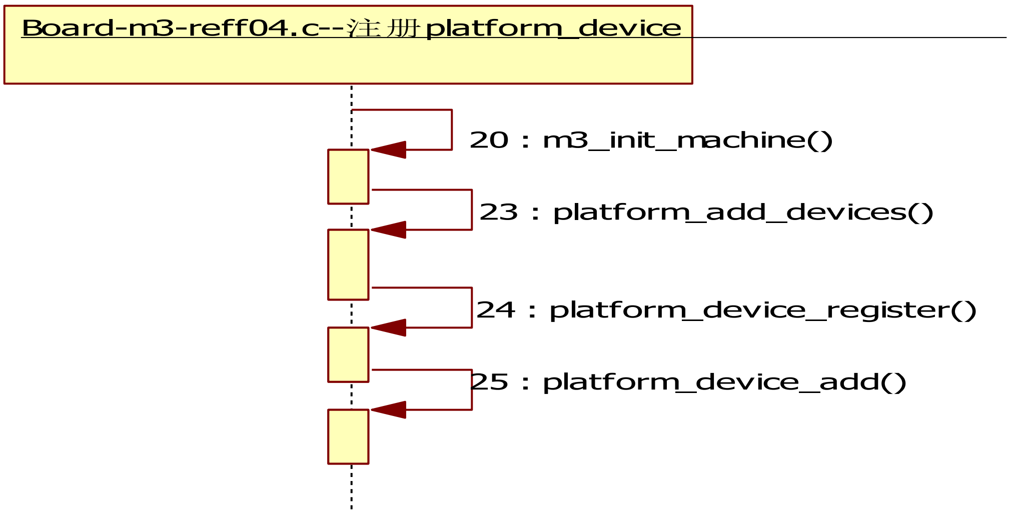
以 aml-i2c.0 aml-i2c.1 aml-i2c.2 为例，board-m3-reff04.c 文件中首先声明了 platform_device 结构变量 aml_i2c_device 如下 ：

```
static struct platform_device aml_i2c_device = {

    .name          = "aml-i2c",

    .id        = 0,

    .num_resources     = ARRAY_SIZE(aml_i2c_resource),

    .resource      = aml_i2c_resource,

    .dev = {

    .platform_data = &aml_i2c_plat,

    },

};
```

另两个结构变量 aml_i2c_device1, aml_i2c_device2 的声明与 aml_i2c_device 类似，这些结构变量组成一个 platform_device 数组 platform_devs[]，如下图



然后 kernel 启动的时候 会调用 m3_init_machine，m3_init_machine 通过调用 platform_add_devices 把 platform_devs 数组中的所有设备注册到总线



aml_i2c_device 被挂接到 platform 总线的具体过程如下：

首先 aml_i2c_device 的成员变量 struct device dev 被初始化

device_initialize(&pdev->dev); 此函数做一些初始化的工作

把 dev->kobj.kset = devices_kset; kobject_init(&dev->kobj, &device_ktype);

Devices_kset 下会挂接所有的设备的 kobj

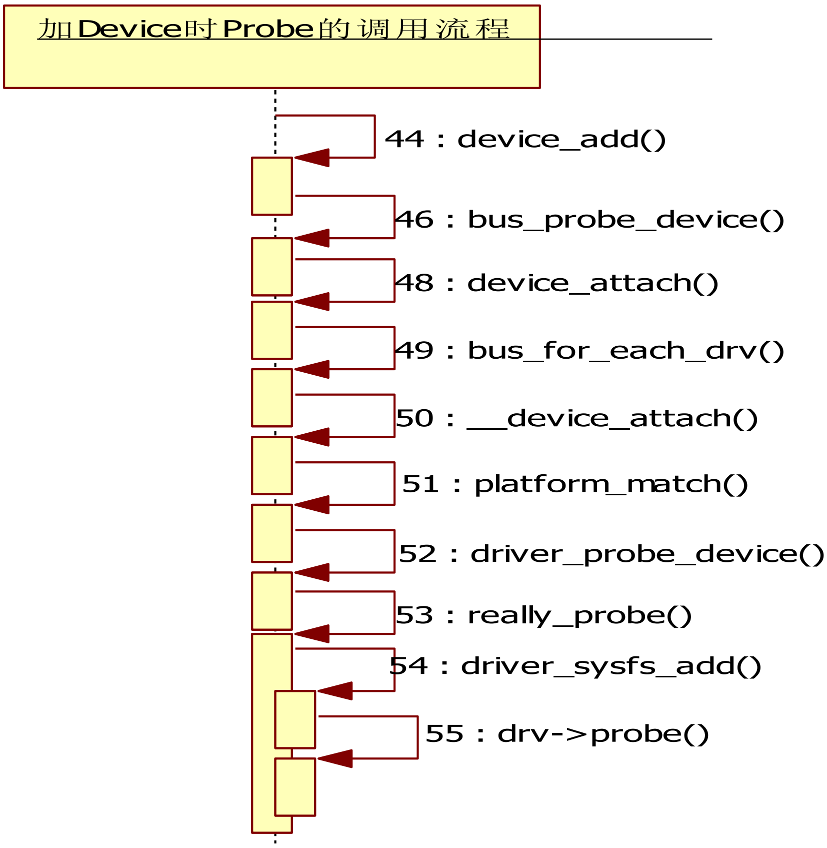
然后是 platform_device_add(pdev);

```
int platform_device_add(struct platform_device *pdev)
{…
    pdev->dev.parent = &platform_bus;
    pdev->dev.bus = &platform_bus_type;
    dev_set_name(&pdev->dev, "%s.%d", pdev->name,  pdev->id);//aml-i2c.0
    device_add(&pdev->dev);
…}
```

其中 device_add(&pdev->dev)做了很多工作，现在对它做一次详细分析

```
int device_add(struct device *dev)
{
…
setup_parent(dev, parent);
// aml_i2c_device.dev.kobj.parent = platform_bus.kobj
error = kobject_add(&dev->kobj, dev->kobj.parent, NULL);
//在 devices/platform 下面创建了一个目录 aml-i2c.0
Error = device_create_file(dev, &uevent_attr);
//在 devices/platform/aml-i2c.0/创建文件 uevent
error = device_add_attrs(dev);
//不增加
```

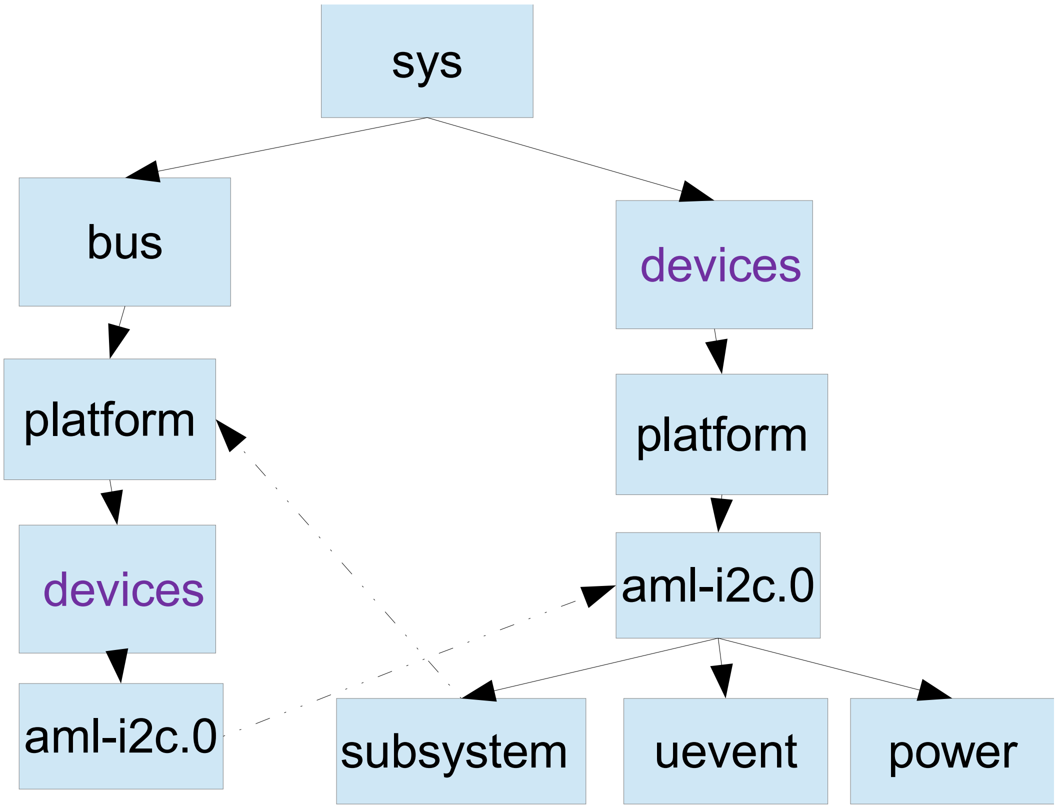
```
error = bus_add_device(dev);
//sys/bus/platform/devices/aml-i2c.0 -> ../../../devices/platform/aml-i2c.0
//sys/devices/platform/aml-i2c.0/subsystem -> ../../../bus/platform
// klist_add_tail(&dev->p->knode_bus, &bus->p->klist_devices);
//把 dev 加到 bus->p->klist_devices

error = dpm_sysfs_add(dev);
//创建 power 目录
device_pm_add(dev);
//把设备加到 dpm_list 以方便电源管理

if (dev->bus)
        blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
                         BUS_NOTIFY_ADD_DEVICE, dev);
//bus_notifier
kobject_uevent(&dev->kobj, KOBJ_ADD);
//发送 uevent 给/sbin/ueventd 以及 kernel 的 uevent_sock

bus_probe_device(dev);
// bus_probe_device - probe drivers for a new device
//probe 一次后 dev->driver = drv;
// klist_add_tail(&dev->p->knode_driver, &dev->driver->p->klist_devices)

if (parent)
        klist_add_tail(&dev->p->knode_parent,
                 &parent->p->klist_children);
…
}
```
// bus_probe_device - probe drivers for a new device
//probe 一次后 dev->driver = drv;
// klist_add_tail(&dev->p->knode_driver, &dev->driver->p->klist_devices)
Probe 函数的调用次序:



/sys/bus/platform/drivers/aml-i2c/aml-i2c.0/driver->
../../../bus/platform/drivers/aml-i2c

/sys/bus/platform/drivers/aml-i2c/aml-i2c.0/driver/aml-i2c.0 ->
../../../../devices/platform/aml-i2c.0

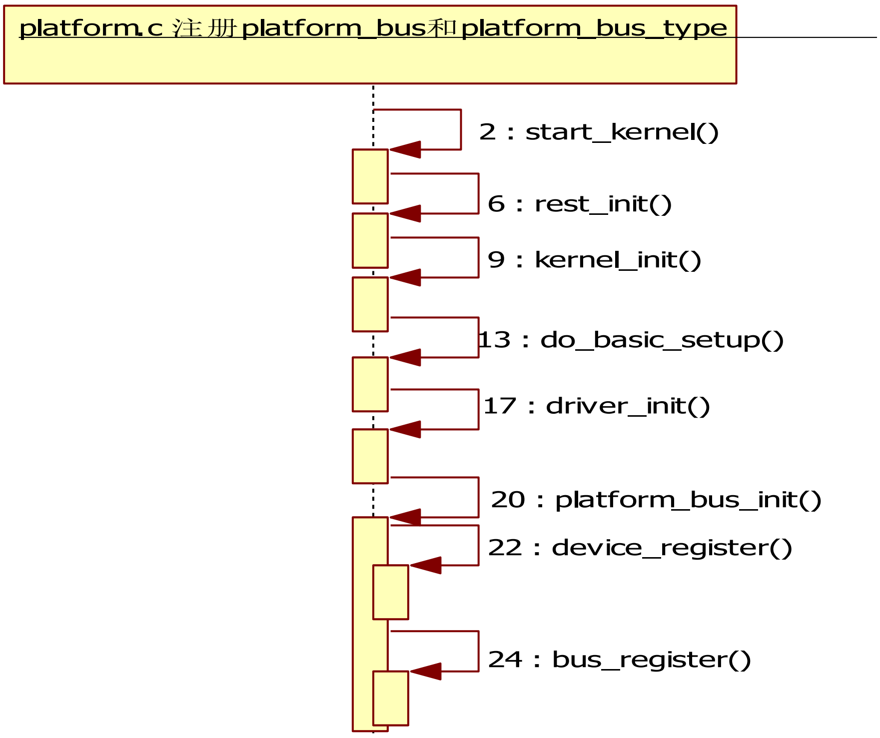
最终一个 aml-i2c.0 设备被挂接后，形成以下数据结构：



注 1 Kernel 启动时，会注册 platform_bus 和 platform_bus_type，**具体参考 platform_bus 和 platform_bus_typed 的注册过程**
注 2 Kernel 启动时，会执行 m3_init_machine()，具体参考 m3_init_machine()被调用流程

➢ platform_bus 以及 platform_bus_type 的注册过程

具体的调用流程如下图所示：



1,用 device_register(&platform_bus); 注册了 platform_bus 设备;

初始化结构

```
struct device platform_bus = {
    .init_name  = "platform",
};
```

主要就是在 devices 目录下创建了 platform 目录，以及在 platform 目录下创建了 uevent 文件;

platform_bus.kobj.name = “platform”;

platform_bus.kobj.ktype = &device_type;

platform_bus.kobj.parent = &(device_kset->kobj)

platform_bus.kobj.kset = device_kset

然后

list_add_tail(&kobj->entry, &kobj->kset->list);

也就是说 platform_bus.kobj.entry 被挂接在 device_kset->list 上

2,而用 bus_register(&platform_bus_type);注册了 platform_bus_type 总线;

```
struct bus_type platform_bus_type = {
    .name       = "platform",
    .dev_attrs  = platform_dev_attrs,
    .match      = platform_match,
    .uevent     = platform_uevent,
```

```
    .pm      = &platform_dev_pm_ops,
};
platform_bus_type.p->subsys.kobj.name = "platform"
platform_bus_type.p->subsys.kobj.kset = bus_kset;
platform_bus_type.p->subsys.kobj.ktype = &bus_ktype;
platform_bus_type.p->drivers_autoprobe = 1;
platform_bus_type.p->subsys.kobj.parent = bus_kset->kobj;
```

1， bus_kset 在 sys 目录下创建了一个目录 bus 这是一个顶层的 kset

2， list_add_tail(&kobj->entry, &kobj->kset->list);

也就是说 platform_bus_type.p->subsys.kobj.entry 被挂接在 bus_kset->list 上

3， 在/sys/bus 目录下创建了文件 platform,

kobject_uevent_env:

env 设置为

SUBSYSTEM="bus"

DEVPATH="/sys/bus/platform"

ACTION="add"

SEQNUM=%llu（记录 uevent 发送数量）

kobj->state_add_uevent_sent = 1;

如果已经配置了网络则利用 uevent_sock 发送消息

netlink_broadcast(uevent_sock, skb, 0, 1, GFP_KERNEL);

uevent_sock 在 Kernel 启动后 被创建

```
postcore_initcall(kobject_uevent_init);
static int __init kobject_uevent_init(void){…
    uevent_sock = netlink_kernel_create(&init_net, NETLINK_KOBJECT_UEVENT,
                      1, NULL, NULL, THIS_MODULE);
…}
```

然后 uevent_helper = /sbin/ueventd

当 uevent_helper 不为空是，直接执行

call_usermodehelper(argv[0], argv, env->envp, UMH_WAIT_EXEC);

也就是运行/sbin/ueventd

关于 bus_attr_uevent 全局变量定义的展开如下：

```
static BUS_ATTR(uevent, S_IWUSR, NULL, bus_uevent_store);
#define BUS_ATTR(_name, _mode, _show, _store)    \
struct bus_attribute bus_attr_##_name = __ATTR(_name, _mode, _show, _store)
#define __ATTR(_name,_mode,_show,_store) { \
    .attr = {.name = __stringify(_name), .mode = _mode },    \
    .show   = _show,                    \
    .store  = _store,                   \
}
struct bus_attribute bus_attr_uevent  =
{.attr = {.name = uevent, .mode = SIWUSR},
 .show = NULL,
```

```
 .store = bus_uevent_store,}
int bus_register(struct bus_type *bus)
{
    …
    retval = bus_create_file(bus, &bus_attr_uevent);//创建一个只写的 uevent 文件
    …
    priv->devices_kset = kset_create_and_add("devices", NULL,
                                 &priv->subsys.kobj);
    //创建 devices 目录在/sys/bus/platform/devices 下

    priv->drivers_kset = kset_create_and_add("drivers", NULL,
                                 &priv->subsys.kobj);
    //创建 drivers 目录在/sys/bus/platform/drivers 下
    retval = add_probe_files(bus);
    //创建特性文件 drivers_autoprobe 和 drivers_probe

    retval = bus_add_attrs(bus);
    //由于 platform_bus_type.bus_attrs 为 NULL，因此不做处理
}
```

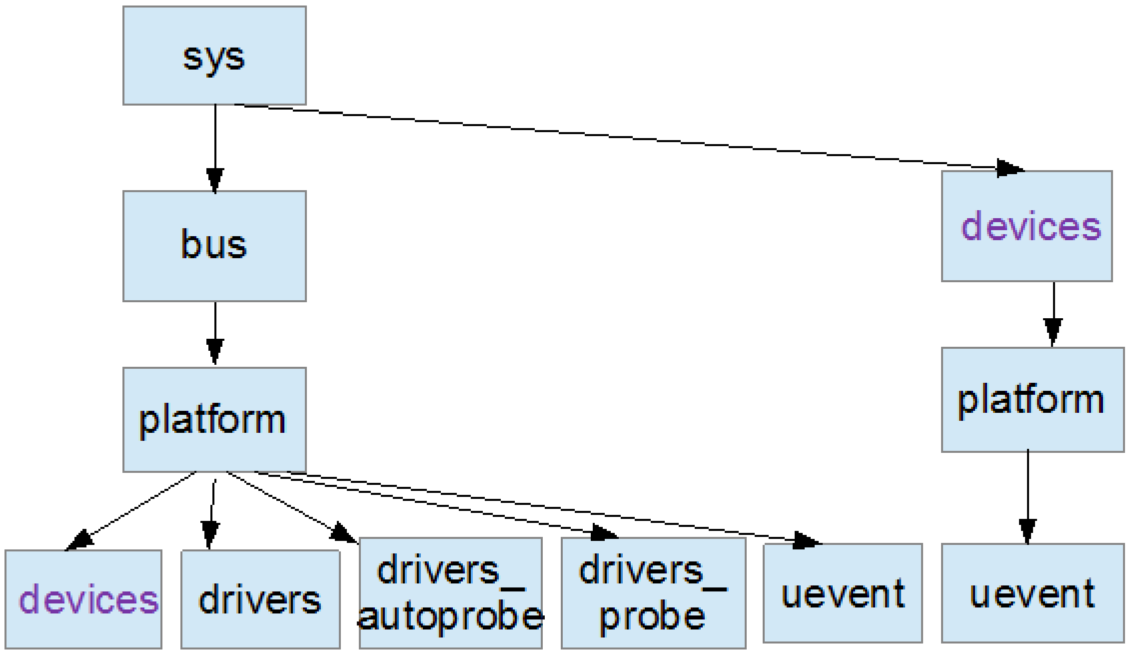
bus_attr_drivers_autoprobe 和 bus_attr_drivers_probe 的展开

```
static BUS_ATTR(drivers_autoprobe, S_IWUSR | S_IRUGO,
        show_drivers_autoprobe, store_drivers_autoprobe);
static BUS_ATTR(drivers_probe, S_IWUSR, NULL, store_drivers_probe);

#define BUS_ATTR(_name, _mode, _show, _store)    \
struct bus_attribute bus_attr_##_name = __ATTR(_name, _mode, _show, _store)
#define __ATTR(_name,_mode,_show,_store) { \
    .attr = {.name = __stringify(_name), .mode = _mode },    \
    .show   = _show,                        \
    .store  = _store,                       \
}
```

到此，bus_register(&platform_bus_type) 已经完成，在/sys/bus/platform 目录下建立了以下目录和文件

1， devices 2,drivers 3,drivers_autoprobe 4,drivers_probe 5, uevent.

其中 drivers_autoprobe, drivers_probe, uevent 为特性文件


sys
bus
platform
devices
drivers
drivers_
autoprobe
drivers_
probe
uevent
devices
platform
uevent

➢ m3_init_machine()被调用流程

1，m3_init_machine 在宏中被使用

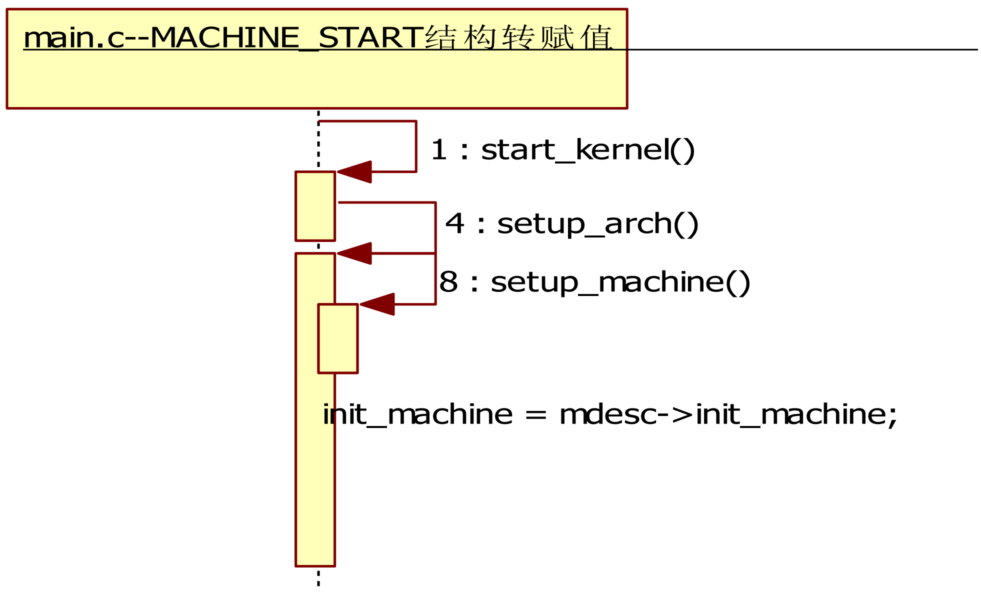
```
MACHINE_START(MESON3_8726M_SKT, "AMLOGIC MESON3 8726M Embel SZ")
    .init_machine    = m3_init_machine,
MACHINE_END
```

其宏定义如下

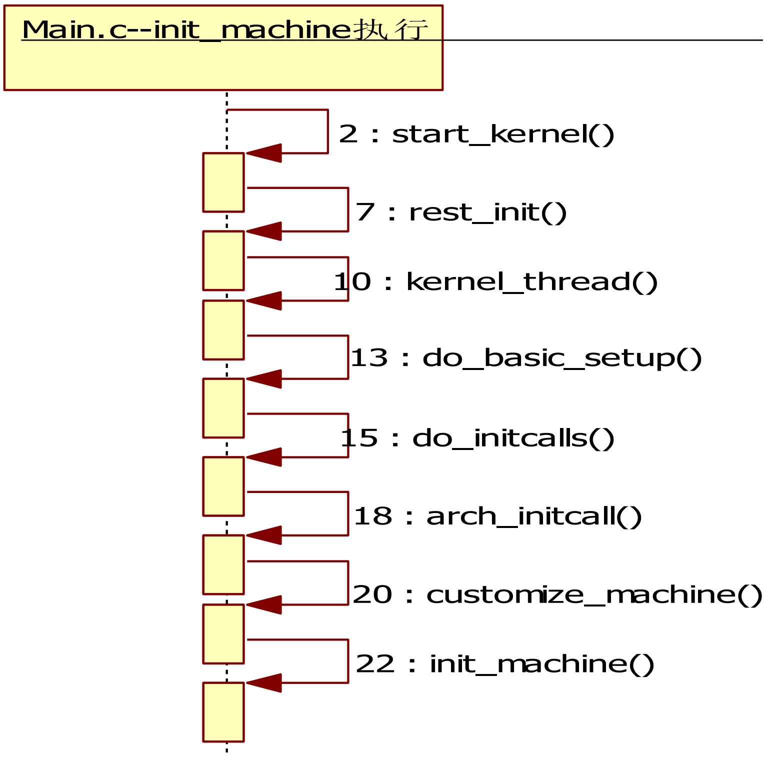
```
#define MACHINE_START(_type,_name)           \
static const struct machine_desc __mach_desc_##_type     \
 __used                          \
 __attribute__((__section__(".arch.info.init"))) = {     \
    .nr      = MACH_TYPE_##_type,          \
    .name        = _name,

#define MACHINE_END                  \
};
```

把宏展开可得到如下结论：m3_init_machinehe 函数指针 赋给了 struct machine_desc 的结构体变量 __mach_desc_MESON3_8726M_SKT 中的成员变量 init_machine

2，读取 machine_start 结构，并把结构的 init_machine 指针赋给全局变量 init_machine



3，执行 init_machine 函数指针，即执行 m3_init_machine()


Main.c--init_machine执行
2 : start_kernel()
7 : rest_init()
10 : kernel_thread()
13 : do_basic_setup()
15 : do_initcalls()
18 : arch_initcall()
20 : customize_machine()
22 : init_machine()

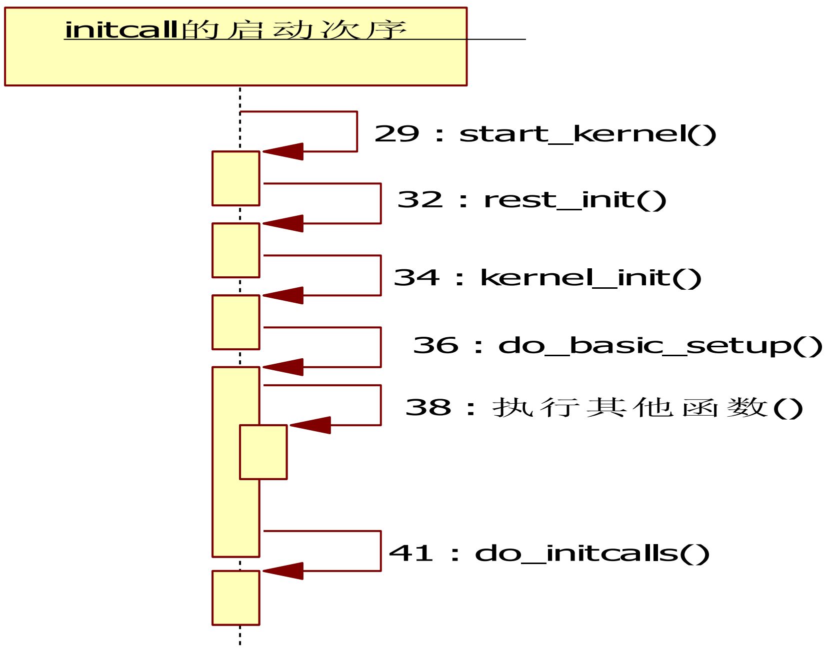
➢ 在 Kernel 启动时 驱动加载次序 module_init

Init.h 中有相关 initcall 的启动次序，在 system.map 中可看出具体的__initcall 指针

```
#define pure_initcall(fn)          __define_initcall("0",fn,0)
#define core_initcall(fn)          __define_initcall("1",fn,1)
#define core_initcall_sync(fn)         __define_initcall("1s",fn,1s)
#define postcore_initcall(fn)          __define_initcall("2",fn,2)
#define postcore_initcall_sync(fn)  __define_initcall("2s",fn,2s)
#define arch_initcall(fn)          __define_initcall("3",fn,3)
#define arch_initcall_sync(fn)         __define_initcall("3s",fn,3s)
#define subsys_initcall(fn)        __define_initcall("4",fn,4)
#define subsys_initcall_sync(fn)       __define_initcall("4s",fn,4s)
#define fs_initcall(fn)            __define_initcall("5",fn,5)
#define fs_initcall_sync(fn)           __define_initcall("5s",fn,5s)
#define rootfs_initcall(fn)        __define_initcall("rootfs",fn,rootfs)
#define device_initcall(fn)        __define_initcall("6",fn,6)
#define device_initcall_sync(fn)       __define_initcall("6s",fn,6s)
#define late_initcall(fn)          __define_initcall("7",fn,7)
#define late_initcall_sync(fn)         __define_initcall("7s",fn,7s)
```

module_init 在的启动序号为 6

```
#define device_initcall(fn)        __define_initcall("6",fn,6)
#define __initcall(fn) device_initcall(fn)
#define module_init(x)  __initcall(x);
```



```
static void __init do_initcalls(void)
{
    initcall_t *fn;
    for (fn = __early_initcall_end; fn < __initcall_end; fn++)
        do_one_initcall(*fn);
    /* Make sure there is no pending stuff from the initcall sequence */
    flush_scheduled_work();
}
```

因此驱动模块在 Kernel 启动过程中的启动次序是非常靠后的

具体的每个驱动的启动次序可以从 system.map 看出：

```
c003288c t __initcall_i2c_init2
c00328b0 t __initcall_video_early_init3
c00328b4 t __initcall_video2_early_init3
c00328b8 t __initcall_aml_i2c_init3
c0032c18 t __initcall_i2c_dev_init6
c0032c28 t __initcall_videodev_init6
c0032c30 t __initcall_v4l2_i2c_drv_init6
c0032c34 t __initcall_v4l2_i2c_drv_init6
c0032d24 t __initcall_video_init6
c0032d28 t __initcall_video2_init6
```