(1条消息) 树莓派uboot的串口初始化(uboot驱动结构 1主要流程)_groundhappy的专栏-CSDN博客

可以参看

https://oska874.github.io/%E6%BA%90%E7%A0%81/uboot%E9%A9%B1%E5%8A%A8%E6%A1%86%E6%9E%B6%E6%A6%82%E8%BF%B0.html

http://www.itdadao.com/articles/c15a621074p0.html

的一些解释

在board_f.c的顺序初始化中有serial_init函数。

该函数位于

/drivers/serial/serial-uclass.c当中

```
int serial_init(void)
{
serial_find_console_or_panic();
gd->flags |= GD_FLG_SERIAL_READY;
return 0;
}
```

主要是调用serial_find_console_or_panic随后将gd的串口标记设置为READY状态

进入serial_find_console_or_panic后主要是通过通用的uclass来操作设备。我的树莓派最后是通过执行

```
(!uclass_first_device(UCLASS_SERIAL, &dev) && dev))
gd->cur_serial_dev = dev
```

得到了dev设备，后续操作使用串口都是用的cur_serial_dev。

```
1. int uclass_first_device(enum uclass_id id, struct udevice **devp)
```

```
2.
3.                 ret = uclass_find_first_device(id, &dev);
4.
5.
6. return uclass_get_device_tail(dev, ret, devp);
7.
```

首先通过uclass_find_first_device找到第一个设备。随后再获取设备细节。否则不算成功

```
1. int uclass_find_first_device(enum uclass_id id, struct udevice **devp)
2.
3.
4.
5.
6.
7.
8.         ret = uclass_get(id, &uc);
9.
10.
11.
12. if (list_empty(&uc->dev_head))
13.
14.
15.
16.         *devp = list_first_entry(&uc->dev_head, struct udevice, uclass_node);
17.
18.
19.
```

先看一些结构

1struct driver 驱动对象。作为udevice的一个属性。driver是用来驱动udevice的，比如调用bind来将设备和驱动进行绑定。

ops指向驱动的其他操作。

```
1.
2.
3.
4. const struct udevice_id *of_match;
5. int (*bind)(struct udevice *dev);
6. int (*probe)(struct udevice *dev);
7. int (*remove)(struct udevice *dev);
8. int (*unbind)(struct udevice *dev);
9. int (*ofdata_to_platdata)(struct udevice *dev);
10. int (*child_post_bind)(struct udevice *dev);
11. int (*child_pre_probe)(struct udevice *dev);
12. int (*child_post_remove)(struct udevice *dev);
13. int priv_auto_alloc_size;
14. int platdata_auto_alloc_size;
15. int per_child_auto_alloc_size;
16. int per_child_platdata_auto_alloc_size;
17.
18.
19.
```

2 struct udevice 设备对象。包含struct driver驱动的属性。

```
1.
2. const struct driver *driver;
3.
4.
5.
6.
7.
8.
9.
```

```
10.
11.
12.
13.
14. struct list_head uclass_node;
15. struct list_head child_head;
16. struct list_head sibling_node;
17.
18.
19.
20.
21. struct list_head devres_head;
22.
23.
```

3uclass_driver作为一个uclass的一个属性

```
1.
2.
3.
4. int (*post_bind)(struct udevice *dev);
5. int (*pre_unbind)(struct udevice *dev);
6. int (*pre_probe)(struct udevice *dev);
7. int (*post_probe)(struct udevice *dev);
8. int (*pre_remove)(struct udevice *dev);
9. int (*child_post_bind)(struct udevice *dev);
10. int (*child_pre_probe)(struct udevice *dev);
11. int (*init)(struct uclass *class);
12. int (*destroy)(struct uclass *class);
13. int priv_auto_alloc_size;
```

```
14. int per_device_auto_alloc_size;
15. int per_device_platdata_auto_alloc_size;
16. int per_child_auto_alloc_size;
17. int per_child_platdata_auto_alloc_size;
18.
19.
20.
```

4 uclass 结构包含uclass_driver属性，通过uclass_driver来操作这个uclass。

比如初始化uclass。销毁Uclass。当有一个udevice设备被添加到uclass的链表里面的时候执行pre和post的操作，移除执行等操作

里面有链表包含了许多的udevice

```
1.
2.
3. struct uclass_driver *uc_drv;
4. struct list_head dev_head;
5. struct list_head sibling_node;
6.
```

首先uclass_get找到对应的uclass指针，如果没有找到。那么添加一个。

```
1. int uclass_get(enum uclass_id id, struct uclass **ucp)
2.
3.
4.
5.
6.
7.
8. return uclass_add(id, ucp);
9.
10.
```

```
11.
12.
```

uclass_find通过查找gd->uclass_root对应的uc对象的uc_driv的id是否和key相同来返回这个uc，有点绕。

```
1. struct uclass *uclass_find(enum uclass_id key)
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.         list_for_each_entry(uc, &gd->uclass_root, sibling_node) {
14. if (uc->uc_drv->id == key)
15.
16.
17.
18.
19.
20.
```

至于uclass_add。也并不是凭空添加一个。而是从所有uclass_driver所属段中找到有没有相同id的的 uclass_driver结构，如果有，调用这个uclass_driver结构的init函数执行初始化等等。

```
1. static int uclass_add(enum uclass_id id, struct uclass **ucp)
2.
3. struct uclass_driver *uc_drv;
4.
5.
6.
7.
8.         uc_drv = lists_uclass_lookup(id);
```

```
 9.
10.                 debug("Cannot find uclass for id %d: please add the UCLASS_DRIVER() declaration for this UCLASS_... id\n",
11.
12.
13.
14.
15.
16.
17.
18.
19.         uc = calloc(1, sizeof(*uc));
20.
21.
22. if (uc_drv->priv_auto_alloc_size) {
23.                 uc->priv = calloc(1, uc_drv->priv_auto_alloc_size);
24.
25.
26.
27.
28.
29.
30.         INIT_LIST_HEAD(&uc->sibling_node);
31.         INIT_LIST_HEAD(&uc->dev_head);
32.         list_add(&uc->sibling_node, &DM_UCLASS_ROOT_NON_CONST);
33.
34.
35.
36.
37.
38.
39.
40.
41.
42.
43.
```

```
44. if (uc_drv->priv_auto_alloc_size) {
45.
46.
47.
48.         list_del(&uc->sibling_node);
49.
50.
51.
52.
53.
```

主要是通过 lists_uclass_lookup查询得到uclass_driver对象。调用calloc分配一个uclass结构.并且将其加入到 root_dm的slib过程 list_add(&uc->sibling_node, &DM_UCLASS_ROOT_NON_CONST);

#define DM_UCLASS_ROOT_NON_CONST (((gd_t *)gd)->uclass_root)

后续通过查询gd->uclass_root的表就能找到这个uclass了。通过uclass也可以找到uclass_driver,以及这个uclass下对应的所有设备

那么第一步已经拿到了第一个设备。通过gd->uclass_root表。找到对应的uclass结构。在uclass结构的dev_head当中取得第一个设备。中间的比较过程都是通过比较uclass_driver->id是否匹配得到的。

拿到设备以后要对设备初始化。

```
1. int uclass_get_device_tail(struct udevice *dev, int ret,
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
```

device_probe比较长。假定没有parent之类的东西。主要就是

1如果每个设备需要分配一些内存。那么分配对应的内存

2 执行uc_driver->pre_probe如果有。

（执行dev->ofdata_to_platdata 如果有）

3执行dev->probe函数,如果有。

4执行uc_driver->post_probe,如果有。

这样实现了找到一个设备，打开一个设备的过程

主要具体的udevice设备driver函数有 bind unbind用于和uclass绑定的时候调用 probe remove主要是激活和停用设备。以及ops指针指向设备的具体的功能性函数

具体的uclass 类uclass_driver函数有 pre_unbind post_bind 用在设备bind之后和unbind之前。post_probe pre_remove用在probe之后和remove之前。以及init和destroy针对于uclass自身的初始化

再细看串口的初始化，在serial-uclass.c当中有如下定义

```
UCLASS_DRIVER(serial) = {
.id = UCLASS_SERIAL,
.name = "serial",
.flags = DM_UC_FLAG_SEQ_ALIAS,
.post_probe = serial_post_probe,
.pre_remove = serial_pre_remove,
.per_device_auto_alloc_size = sizeof(struct serial_dev_priv),
};
```

通过UCLASS_DRIVER声明了一个uclass_driver结构

```
1.
2. #define UCLASS_DRIVER(__name)                                           \
3.         ll_entry_declare(struct uclass_driver, __name, uclass)
1. #define ll_entry_declare(_type, _name, _list)                           \
```

```
2.         _type _u_boot_list_2_##_list##_2_##_name __aligned(4)          \
3.
4.                         section(".u_boot_list_2_"#_list"_2_"#_name)))
```

这样在uclass_add的过程中就分配了一个uclass结构，并且加入了链表当中。只要调用了uclass_add就会生成一个设备类

对照前面可以知道。在添加一个串口设备之后会调用serial_post_probe.移除一个串口设备之前则会调用serial_pre_remove。并且为每一个串口设备分配了

struct serial_dev_priv 大小的结构，地址放到 dev->uclass_priv中。

uclass上的每一个设备是如何添加到链表当中的呢

在uboot的init_sequence_f初始化序列当中最开始有initf_dm函数

initf_dm->dm_init_and_scan(true)

{

dm_init();

主要是初始化root driver model

1init_list(gd->uclass_root)

2device_bind_by_name 产生了一个root udevice对象 并且将这个udevice插入对应的uclass链表中，并调用drv的bind函数

3调用device_probe初始化root udevice

dm_scan_platdata();

调用lists_bind_drivers对剩余的其他通过U_BOOT_DEVICE 执行device_bind_by_name（也就是new一个对应于驱动的设备对象）。但是没有对设备进行激活

U_BOOT_DEVICE 定义了所需要的驱动的名称，以及相关的平台数据

```
1.
2.
3.
```

```
4. #if CONFIG_IS_ENABLED(OF_PLATDATA)
5.
6.
7.
```

通过strcmp比较这个name和U_BOOT_DRIVER定义的就能知道是不是相同的驱动。继而产生这个对象。rpi的串口设备就是在这里产生，并且添加的。

}

在rpi.c中定义了

```
1. static const struct pl01x_serial_platdata serial_platdata = {
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12. U_BOOT_DEVICE(bcm2835_serials) = {
13.
14.         .platdata = &serial_platdata,
15.
```

device_bind_by_name 通过 name=serial_pl01x找到了在serial_pl01x.c中定义的 driver

```
1. U_BOOT_DRIVER(serial_pl01x) = {
2.
3.
4.         .of_match = of_match_ptr(pl01x_serial_id),
```

```
5.          .ofdata_to_platdata = of_match_ptr(pl01x_serial_ofdata_to_platdata),
6.          .platdata_auto_alloc_size = sizeof(struct pl01x_serial_platdata),
7.          .probe = pl01x_serial_probe,
8.          .ops    = &pl01x_serial_ops,
9.          .flags = DM_FLAG_PRE_RELOC,
10.         .priv_auto_alloc_size = sizeof(struct pl01x_priv),
11.
```

构建了这样一个串口设备对象。

真正激活这个串口设备是在udevice_first_device中找到device后调用device_probe来激活的。

还有一种是通过of_match来匹配兼容的一些设备。比如这里是.name=serial_pl01x。 这种就一般是通过dtb来。

static const struct udevice_id pl01x_serial_id[] ={
{.compatible = "arm,pl011", .data = TYPE_PL011},
{.compatible = "arm,pl010", .data = TYPE_PL010},
{}
};

（比如fdt里面有一个compatible字符串里面有pl011表明也符合这个驱动。也可以使用这个驱动）这里看设备树的时候再说吧。

加上serial-uclass.c里面的

```
1. UCLASS_DRIVER(serial) = {
2.
3.
4.          .flags          = DM_UC_FLAG_SEQ_ALIAS,
5.          .post_probe     = serial_post_probe,
6.          .pre_remove     = serial_pre_remove,
7.          .per_device_auto_alloc_size = sizeof(struct serial_dev_priv),
```

```
8.
```

以及serial_pl01x.c中具体功能执行函数

```
1. static const struct dm_serial_ops pl01x_serial_ops = {
2.         .putc = pl01x_serial_putc,
3.         .pending = pl01x_serial_pending,
4.         .getc = pl01x_serial_getc,
5.         .setbrg = pl01x_serial_setbrg,
6.
```

这样就能得到串口初始化的流程了

由于没有定义OF_CONTROL.所以

#if CONFIG_IS_ENABLED(OF_CONTROL)
#define of_match_ptr(_ptr) (_ptr)
#else
#define of_match_ptr(_ptr) NULL
#endif /* CONFIG_IS_ENABLED(OF_CONTROL) */

所有的of_match_prt都是NULL，因此不用执行pl01x_serial_ofdata_to_platdata

1driver 没有bind函数。所以device_bind_by_name只是分配了一些结构和变量。

2probe执行 pl01x_serial_probe函数

3post_probe执行serial_post_probe

```
1. static int pl01x_serial_probe(struct udevice *dev)
2.
3. struct pl01x_serial_platdata *plat = dev_get_platdata(dev);
4. struct pl01x_priv *priv = dev_get_priv(dev);
5.
```

```
        priv->regs = (struct pl01x_regs *)plat->base;


return pl01x_generic_serial_init(priv->regs, priv->type);


```

主要是从rpi.c里面的

```
static const struct pl01x_serial_platdata serial_platdata = {


```

获取对应的数据。赋值给priv->regs,主要是寄存器的值。以及串口的类型。这里我们的板子是0X20201000.类型是TYPE_PL011

由于skip_init为true.所以pl01x_generic_serial_init函数是不执行的。

后续执行serial_post_probe

struct dm_serial_ops *ops = serial_get_ops(dev);

转换成dm_serial_ops*结构
执行设置波特率的函数ops->setbrg。（注意。由于rpi的设置里面.skipinit=true，其实不会设置波特率，也就是只能使用默认的配置。不能更改配置，如果需要更改，要修改serial_platdata结构。设置clock等等寄存器参数）。
调用stdio_register_dev将串口设备注册为标准输入输出设备。
也就是将函数包裹一下。
具体看到Uboot的最后等待输入的函数。在cli_simple_loop当中调用cli_readline调用cli_readline_into_buffer
for循环调用getc()获取字符。getc 调用fgetc(stdin)。前面将serial注册成为了stdio，实际的初始化是在console_init_r中执行注册和初始化的。会调用stdio的start等函数。后续stdio就使用serial来操作了。

serial的主要功能get和put也就是简单读写寄存器。。。。设置波特率什么的需要参看具体的操作手册。由于rpi默认设置好了。这里uboot的配置是skipinit。所以就看一下读写过程

```
1. static void pl01x_serial_putc(const char c)
2.
3.
4. while (pl01x_putc(base_regs, '\r') == -EAGAIN);
5.
6. while (pl01x_putc(base_regs, c) == -EAGAIN);
7.
```

```
1. static int pl01x_putc(struct pl01x_regs *regs, char c)
2.
3.
4. if (readl(&regs->fr) & UART_PL01x_FR_TXFF)
5.
6.
7.
8.
9.
10.
11.
12.
13.
```

写的过程是循环写直到写入成功。

#define UART_PL01x_FR_TXFF 0x20

读取fr寄存器 。比较第5位是否为1.如果为1。根据手册。第五位为1表示传输列队满了。因此要等待，等待可写了再写入。

TXFF =transmit FIFO FULL。可写的时候将数据写入dr寄存器。

```
1. static int pl01x_serial_getc(void)
```

```
2.
3.
4. int ch = pl01x_getc(base_regs);
5.
6.
7.
8.
9.
10.
11.
12.
13.
```

```
1. static int pl01x_getc(struct pl01x_regs *regs)
2.
3.
4.
5.
6. if (readl(&regs->fr) & UART_PL01x_FR_RXFE)
7.
8.
9.
10.
11.
12.
13.
14.                 writel(0xFFFFFFFF, &regs->ecr);
15.
16.
17.
18.
19.
```

读的过程是循环读取，直到读取成功。

读取也是先读取fr寄存器。RXFE表示receive transmit fifo empty。也就是判断读取列队是否为空。如果不为空，那么读取这

个数据。

由于dr位的0-7是数据。8-11是错误标记位。如果有错误还要清除这个错误寄存器并且返回-1

整个过程还是比较清楚的。主要是使用了Uboot的驱动模型来执行这个操作。简化的做就非常简单。设置相关的寄存器初始化。后续读取写入就可以了

这种是静态的

还有一种是动态的。

通过

serial_register();

实现串口的注册