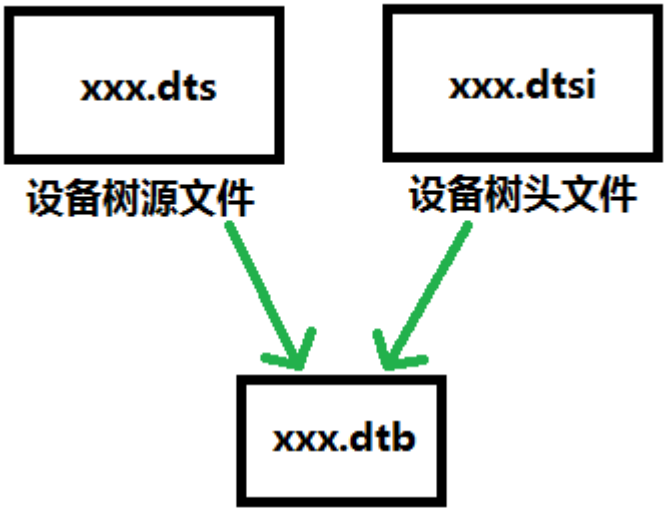
Linux 设备树使用方法

作者:向仔州

设备树文件类型

设备树支持 linux3.x 以上的内核



最后编译的时候是将dts文件编译成dtb二进制文件

这里的dtsi文件就类似C语言的头文件，但是呢！！dtsi文件里面的代码可以被dts文件调用

```
#include "imx6dl.dtsi"
#include "imx6qdl-sabresd.dtsi"

/ {
        model = "Freescale i.MX6 DualLite SABRE Smart Device Board";
        compatible = "fsl,imx6dl-sabresd", "fsl,imx6dl";
};

&battery {
        offset-charger = <1485>;
        offset-discharger = <1464>;
        offset-usb-charger = <1285>;
};

&iomuxc {
        epdc {
                pinctrl_epdc_0: epdcgrp-0 {
                        fsl,pins = <
                                MX6QDL_PAD_EIM_A16__EPDC_DATA00    0x80000000
                                MX6QDL_PAD_EIM_DA10__EPDC_DATA01   0x80000000
                                MX6QDL_PAD_EIM_DA12__EPDC_DATA02   0x80000000
                                MX6QDL_PAD_EIM_DA11__EPDC_DATA03   0x80000000
                                MX6QDL_PAD_EIM_LBA__EPDC_DATA04    0x80000000
                                MX6QDL_PAD_EIM_EB2__EPDC_DATA05    0x80000000
```

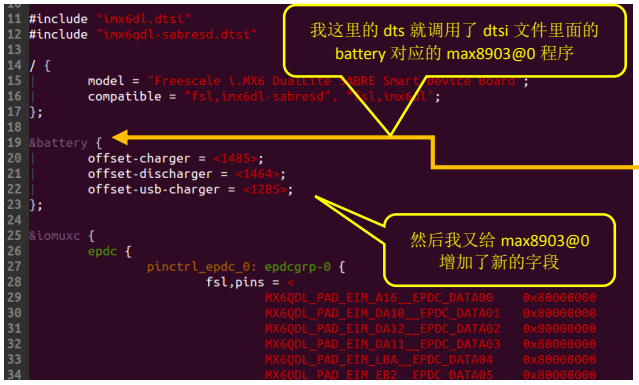
`vim imx6dl-sabresd.dts` 这是打开的 dts 文件

```
#include <dt-bindings/input/input.h>

/ {
        aliases {
                mxcfb0 = &mxcfb1;
                mxcfb1 = &mxcfb2;
                mxcfb2 = &mxcfb3;
                mxcfb3 = &mxcfb4;
        };

        battery: max8903@0 {
                compatible = "fsl,max8903-charger";
                pinctrl-names = "default";
                dok_input = <&gpio2 24 1>;
                uok_input = <&gpio1 27 1>;
                chg_input = <&gpio3 23 1>;
                flt_input = <&gpio5 2 1>;
                fsl,dcm_always_high;
                fsl,dc_valid;
                fsl,usb_valid;
                status = "okay";
        };

/*      leds {
                compatible = "gpio-leds";
```

`vim imx6qdl-sabresd.dtsi` 这是 dtsi 文件

dts 文件才是我们要使用的文件，dtsi 文件里面的代码一般都是用来写一个 CPU 系列通用的程序(比如我这里是 IMX6)，然后根据 CPU 不同的子型号(IMX6 子型号有 imx6Q，imx6dl，imx6s)，决定用哪一个 dts 文件，然后需要用到的通用程序，就不需要单独再在 dts 文件里面编写，直接用&调用 dtsi 文件里面的程序就是了。

我使用的平台是 IMX6dl，所以这里的 dts 文件是 imx6dl-sabresd.dts，我的 dts 文件会去调用一些 dtsi 文件里面的通用程序。



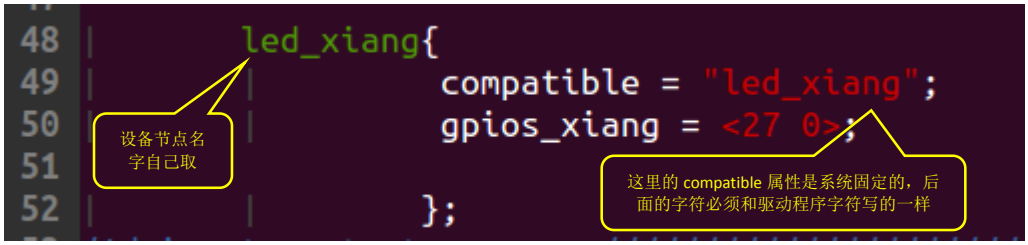
这样的结果就是驱动程序文件一旦匹配了 dts 文件，就会去获取 max8903@0 里面字符变量对应的数据，而且除了本身 8903@0 的数据外，还可以再获取新增加的 offset...里面的数据。

这里就是给个印象，记住就是，后面会将怎么使用设备树。

设备树程序使用实验

```
vim imx6qdl-sabresd.dtsi
```

我在 dtsi 文件里面新增加一个内容，当然你在 dts 增加也可以，看自己喜好。

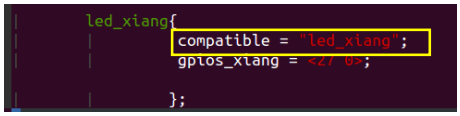


这就是我增加的内容

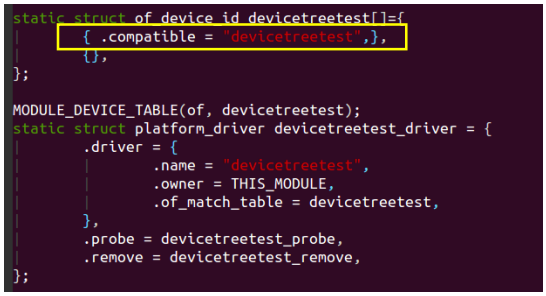
这里的 compatible 就取代了以前老版本 platform_device 的做法，以前是 xxxdevice 匹配 xxxdriver 文件，现在是 dts 里面的设备节点匹配 xxxdriver 文件，但是 dts 只有一个文件，如何能匹配不同的 xxxdriver 文件呢？所以这里的 compatible 里面的字符决定了该设备节点匹配什么驱动文件.

```
root@imx6qdlsolo:/mnt# insmod devicetreetest.ko
enter devicetree_init
```

执行 insmod 之后发现没有执行 probe 函数



设备树 compatible 是 led_xiang



驱动文件里面 compatible 是 devicetreetest

所以要将驱动文件里面的 compatible 字符改成和设备树一样

```
static struct of_device_id devicetreetest[]={
        { .compatible = "led_xiang",},
        {},
};
```

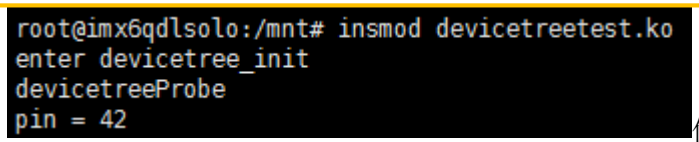
在驱动文件把 compatible 修改后

```
root@imx6qdlsolo:/mnt# insmod devicetreetest.ko
enter devicetree_init
devicetreeProbe
```

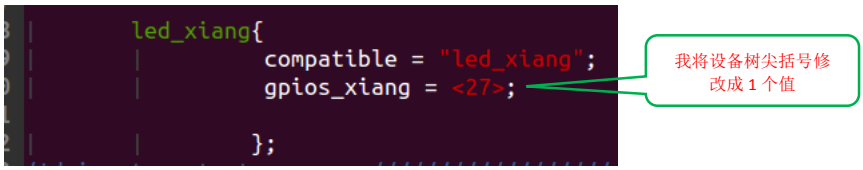
驱动匹配成功执行 probe 函数

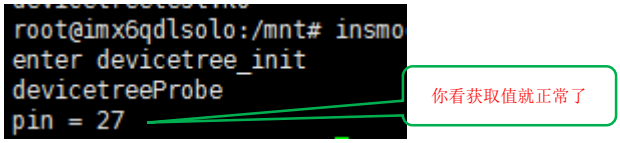
设备树的设备节点和驱动程序匹配成功后，驱动程序就要想办法去获取设备树设备节点里面的数据了

of_property_u32_array(设备节点，属性名称，获取的值，获取几个值) //函数使用方法

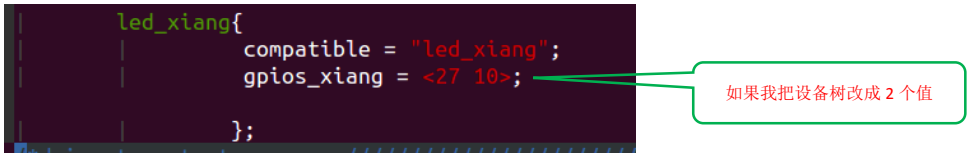


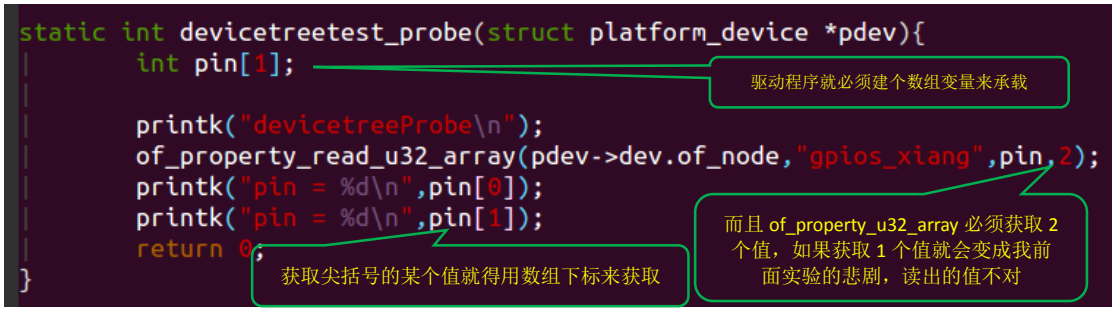
但是为什么执行驱动后得到的 pin 是 42 呢？我要求获取<27 0>尖括号的 27 值啊

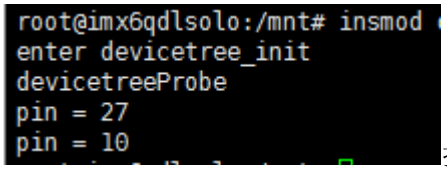




所以使用设备树 of 函数一定要注意 of_property_u32_array，设备树尖括号 1 个值或者多个值解析方式都是不一样的







执行结果正确

这就是设备树的特性，设备树的属性有多少个值，你驱动程序就得匹配多少变量去获取。

设备树格式

每个根/目录下只能有一个 compatible，这个 compatible 是关键字，不能乱写，驱动就是靠 compatible 去找对应的节点

```
/{                          /{
    属性:                       compatible = "xxxx"
    节点:                       cpus{
}                                       }
                                }
```

每个根节点下可以有很多子节点，比如 cpus 就是我取名的一个子节点

```
/{                          /{
    属性:                       compatible = "xxxx"
    节点:                       cpus{
}                                       xzz{
                                        }
                                }
```

子节点下面还可以增加子节点

```
/{                          /{
    属性:                       compatible = "xxxx"
    节点:                       cpus{
}                                       compatible="xzzcpus"
                                            xzz{
                                                compatible="xzznode"
                                            }
                                    }
                            }
```

但是每个子节点只能有一个 compatible，方便驱动程序去寻找

```
/{
    compatible = "xxxx"
    cpus{
            compatible="xzzcpus";
            val = <1 0x3 0x123>;
        }
}
```

除了 compatible 和其它少许的关键字属性名不能变化以外，其余的属性名可以随便取，我这里取了个 val 属性，<>尖括号里面表示多个 32 位的数据值，分号不要忘记写了

```
/{
        compatible = "xxxx"
        cpus{
                compatible="xzzcpus";
                val ="xxxzzzzz abce";
            }
    }
```

属性也可以传字符串

```
/{
        compatible = "xxxx"
        cpus{
                compatible="xzzcpus";
                val = [00 11 22];
            }
    }
```

属性如果用[]中括号，就表示里面存放的是多个 16 进制的字节数据

```
/{
    compatible = "xxxx"
    cpus{
            compatible="xzzcpus";
            val = [0   11 22];
        }
}
```

[]中括号里面必须是两个 16 进制一个字节，如果写一个 16 进制就会出错

```
/{
    compatible = "xxxx"
    cpus{
            compatible="xzzcpus";
            val = [001122];
        }
}
```

[]中括号里面可以不用空格，字节数据可以挨着写，但是驱动程序解析的时候还是一个字节一个字节获取

下面我们用几个 of 函数来试试

```
led_xiang{
        compatible = "led_xiang";
        gpios_xiang = <27 10>;
        addr = <0x11111111 0xffffffff>;
        str = "stringtest";
        array1 = [00 11 22];
        array2 = [001122];

    };
```

这是设备树内容

of_property_read_u32_index(设备节点，属性名称，获取尖括号第几项数据，存入获取的数据)

```
static int devicetreetest_probe(struct platform_device *pdev){

        int pin[1];
        int hex1,hex2;
        const char *buff;

        char array1[10];
        char array2[10];

        printk("devicetreeProbe\n");
        of_property_read_u32_array(pdev->dev.of_node,"gpios_xiang",pin,2);
        printk("pin = %d\n",pin[0]);
        printk("pin = %d\n",pin[1]);


        of_property_read_u32_index(pdev->dev.of_node,"addr",0,&hex1);
        printk("hex1 = %x\n",hex1);

        of_property_read_u32_index(pdev->dev.of_node,"addr",1,&hex2);
        printk("hex2 = %x\n",hex2);


        of_property_read_string(pdev->dev.of_node,"str",&buff);
        printk("---%s---\n",buff);

        of_property_read_u8_array(pdev->dev.of_node,"array1",array1,3);
        printk("%x %x %x\n",array1[0],array1[1],array1[2]);

        of_property_read_u8_array(pdev->dev.of_node,"array2",array2,3);
        printk("%x %x %x\n",array2[0],array2[1],array2[2]);
        return 0;
}
```

addr = <0x11111111 0xffffffff>;
像这种两个 32 位的数据，就要用 index 函数来指定下标一项一项获取，这是获取第 0 项 0x11111111

这是获取第 1 项 0xffffffff

of_property_read_string(设备节点，属性名称，用 const *char 变量来接受字符串)

```
str = "stringtest";
```

像这种字符串的设备树就要用这个函数。

of_property_read_u8_array(设备节点，属性名称，存放字节数组变量，获取中括号第几项数据)

```
array1 = [00 11 22];
array2 = [001122];
```

不管是分开的字节数据，还是连在一起的字节数据，最后 of_property_read_u8_array 获取出来都会将其分开成一个字节一个字节的。

```
root@imx6qdlsolo:/mnt# insmod devicetreetest.ko
enter devicetree_init
devicetreeProbe
pin = 27
pin = 10
hex1 = 11111111
hex2 = ffffffff
---stringtest---
0 11 22
0 11 22
```

```
led_xiang{
        compatible = "led_xiang";
        gpios_xiang = <27 10>;
        addr = <0x11111111 0xffffffff>;
        str = "stringtest";
        array1 = [00 11 22];
        array2 = [001122];

};
```

驱动程序获取结果和设备树定义的数据是一样的，没有问题。

```
led_xiang{
        compatible = "led_xiang";
        gpios_xiang = <20>,<50>;
        str = "stringtest","zifuchuan222";

      };
```

```
static int devicetreetest_probe(struct platform_device *pdev){

    int pin1;
    int pin2;
    const char *buff1;
    const char *buff2;


    printk("devicetreeProbe\n");
    of_property_read_u32_index(pdev->dev.of_node,"gpios_xiang",0,&pin1);
    printk("pin1 = %d\n",pin1);
    of_property_read_u32_index(pdev->dev.of_node,"gpios_xiang",1,&pin2);
    printk("pin2 = %d\n",pin2);


    of_property_read_string_helper(pdev->dev.of_node,"str",&buff1,1,0);
    printk("---%s---\n",buff1);

    of_property_read_string_helper(pdev->dev.of_node,"str",&buff2,1,1);
    printk("---%s---\n",buff2);
    return 0;
}
```

这种两个尖括号也是用 index 去查询

获取几个字符串,这里默认写 1 就是了

获取设备树属性的第几项字符串，主要是看 index 这个值

of_property_read_string_helper(设备节点，属性名称，获取值的变量一定是 const char*变量，获取多少个这里就写 1，设备树第几项字符串)

```
root@imx6qdlsolo:/mnt# insmod d
enter devicetree_init
devicetreeProbe
pin1 = 20
pin2 = 50
---stringtest---
---zifuchuan222---
```

```
led_xiang{
        compatible = "led_xiang";
        gpios_xiang = <20>,<50>;
        str = "stringtest","zifuchuan222";

      };
```

如果两个节点，节点名一样，compatible 名也是一样，会出什么问题

```
led_xiang{
        compatible = "led_xiang";
        gpios_xiang = <20>,<50>;
        str = "stringtest","zifuchuan222";

      };

led_xiang{
        compatible = "led_xiang";
        gpios_xiang = <100>,<200>;
        str = "zzzzz","xxxxxxx";
      };
```

```
enter devicetree_init
devicetreeProbe
pin1 = 100
pin2 = 200
---zzzzz---
---xxxxxxx---
root@imx6qdlsolo:/mnt#
```

如果有两个同名，同 compatible 的设备节点，那么驱动程序会忽略前面同名的设备节点，而去读取最后一个位置的设备节点

```
led_xiang{
        compatible = "led_xiang";
        gpios_xiang = <100>,<200>;
        str = "zzzzz","xxxxxxx";
      };

led_xiang{
        compatible = "led_xiang";
        gpios_xiang = <20>,<50>;
        str = "stringtest","zifuchuan222";

      };
```

我把<20>,<50>放到后面位置，你看驱动就去读取<20>,<50>的设备节点

```
root@imx6qdlsolo:/mnt# insmod
enter devicetree_init
devicetreeProbe
pin1 = 20
pin2 = 50
---stringtest---
---zifuchuan222---
root@imx6qdlsolo:/mnt#
```

像这种两个节点名字一样，compatible 一样的设备树，用@符号将其区分

```
led_xiang@12345{
        compatible = "led_xiang";
        gpios_xiang = <100>,<200>;
        str = "zzzzz","xxxxxxx";
        };

led_xiang@6789{
        compatible = "led_xiang";
        gpios_xiang = <20>,<50>;
        str = "stringtest","zifuchuan222";
      };
```

这样@后面的数字就是区分这个设备节点的序号

这样@后面的数字就是区分这个设备节点的序号

device_node *变量 = of_find_node_by_path(写入设备节点路径) //获取指定路径下的设备节点

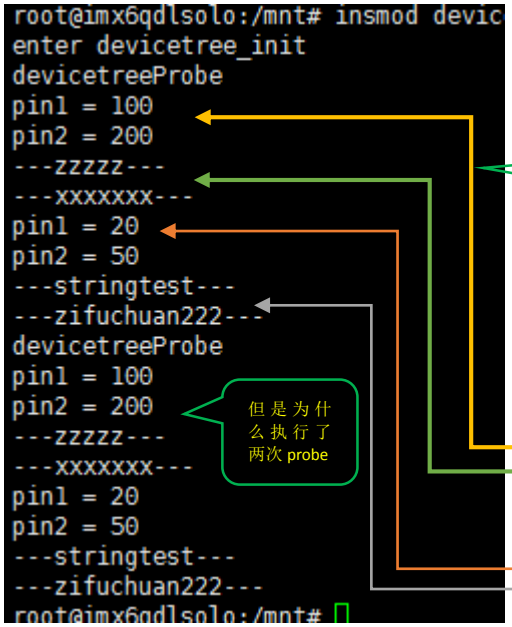
```
static int devicetreetest_probe(struct platform_device *pdev){

        struct device_node *np1=NULL;
        struct device_node *np2=NULL;

        int pin1;
        int pin2;
        const char *buff1;
        const char *buff2;

        printk("devicetreeProbe\n");

        np1=of_find_node_by_path("/led_xiang@12345");
        np2=of_find_node_by_path("/led_xiang@6789");

        of_property_read_u32_index(np1,"gpios_xiang",0,&pin1);
        printk("pin1 = %d\n",pin1);
        of_property_read_u32_index(np1,"gpios_xiang",1,&pin2);
        printk("pin2 = %d\n",pin2);

        of_property_read_string_helper(np1,"str",&buff1,1,0);
        printk("---%s---\n",buff1);
        of_property_read_string_helper(np1,"str",&buff2,1,1);
        printk("---%s---\n",buff2);

        /************************************************/
        of_property_read_u32_index(np2,"gpios_xiang",0,&pin1);
        printk("pin1 = %d\n",pin1);
        of_property_read_u32_index(np2,"gpios_xiang",1,&pin2);
        printk("pin2 = %d\n",pin2);

        of_property_read_string_helper(np2,"str",&buff1,1,0);
        printk("---%s---\n",buff1);
        of_property_read_string_helper(np2,"str",&buff2,1,1);
        printk("---%s---\n",buff2);
        return 0;
}
```

np1 获取序列号为 led_xiang@12345 下面的设备节点，这里有"/"符号，是因为我的这个设备节点是写在设备树/目录下的

np2 也和 np1 一样，只是设备节点不同

打印 led_xiang@12345 节点下面属性的内容

打印 led_xiang@6789 设备节点下面的属性内容




```
led_xiang@12345{
        compatible = "led_xiang";
        gpios_xiang = <100>,<200>;
        str = "zzzzz","xxxxxxx";
        };

led_xiang@6789{
        compatible = "led_xiang";
        gpios_xiang = <20>,<50>;
        str = "stringtest","zifuchuan222";
      };
```

这是因为这里写了两次 compatible，导致驱动识别了两次

```
led_xiang{

        compatible = "led_xiang";

        led_xiang@12345{
                compatible = "xiang";
                gpios_xiang = <100>,<200>;
                str = "zzzzz","xxxxxxx";
         };

        led_xiang@6789{
                compatible = "xiang";
                gpios_xiang = <20>,<50>;
                str = "stringtest","zifuchuan222";
         };
};
```

我们可以给两个@带编号的节点加一个父节点，这样我驱动程序 compatible 去匹配父节点

```
static int devicetreetest_probe(struct platform_device *pdev){

        struct device_node *np1=NULL;
        struct device_node *np2=NULL;

        int pin1;
        int pin2;
        const char *buff1;
        const char *buff2;

        printk("devicetreeProbe\n");

        np1=of_find_node_by_path("/led_xiang/led_xiang@12345");
        np2=of_find_node_by_path("/led_xiang/led_xiang@6789");

        of_property_read_u32_index(np1,"gpios_xiang",0,&pin1);
        printk("pin1 = %d\n",pin1);
        of_property_read_u32_index(np1,"gpios_xiang",1,&pin2);
        printk("pin2 = %d\n",pin2);

        of_property_read_string_helper(np1,"str",&buff1,1,0);
        printk("---%s---\n",buff1);
        of_property_read_string_helper(np1,"str",&buff2,1,1);
        printk("---%s---\n",buff2);

        /*********************************************/
        of_property_read_u32_index(np2,"gpios_xiang",0,&pin1);
        printk("pin1 = %d\n",pin1);
        of_property_read_u32_index(np2,"gpios_xiang",1,&pin2);
        printk("pin2 = %d\n",pin2);

        of_property_read_string_helper(np2,"str",&buff1,1,0);
        printk("---%s---\n",buff1);
        of_property_read_string_helper(np2,"str",&buff2,1,1);
        printk("---%s---\n",buff2);
        return 0;
```

然后用 of_find_node_by_path 寻找绝对路径下的节点

节点之间的引用

```
PIC:xzz@10000000{
        value=<10>
}
```

假如我们在 dtsi 里面定义个节点，给节点取了个别名 PIC

```
&PIC{
        value=<50>//修改了xzz节点里面的value属性值
}
```

在另外一个 dts 文件里面只要包含了该 dtsi 文件，就可以用&节点名，使用该 xzz 节点，而且还可以在当前 dts 文件里修改 dtsi 文件里面 xzz 节点的属性

这样做有什么好处

```
xzz: led_xiang{
        compatible= “led_xiang” ;
        gpios_xiang=<100> ;
        str="zzzzz" ;
};
```

比如说我定义了 gpios_xiang 和 str 属性，然后驱动程序是根据 compatible 去找到这个设备文件，然后获取这些属性

```
&xzz{
        str="ddddd";
} ;
```

现在我想修改一个属性值，这样适合我的客户需求，但是我又不能把原来的其它属性修改到了，这时候你就可以用&获取你要修改节点属性的别名，比如 led_xiang 节点的别名是 xzz，你就获取&xzz，然后修改 str 里面的值，这样 led_xiang 里面的 str 值就被修改了，其它属性还是保持不变

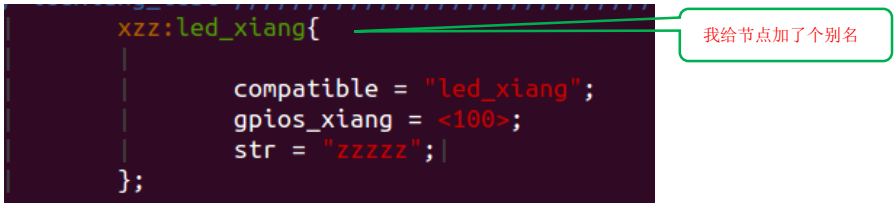
```
        led_xiang{
                compatible = "led_xiang";
                gpios_xiang = <100>;
                str = "zzzzz";
        };
```

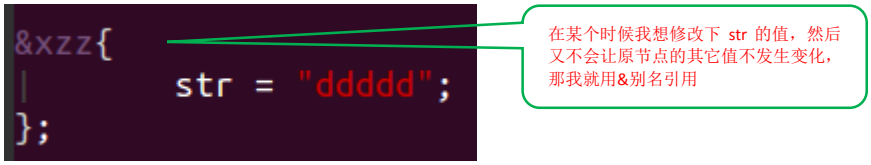
这是先写好的节点属性，和上面的操作方法一样

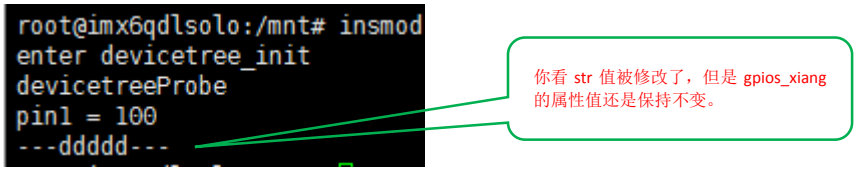
```
static int devicetreetest_probe(struct platform_device *pdev){

        int pin;
        const char *buff;

        printk("devicetreeProbe\n");

        of_property_read_u32_array(pdev->dev.of_node,"gpios_xiang",&pin,1);
        printk("pin1 = %d\n",pin);

        of_property_read_string(pdev->dev.of_node,"str",&buff);
        printk("---%s---\n",buff);

        return 0;
}
```

直接在驱动程序解析没有问题

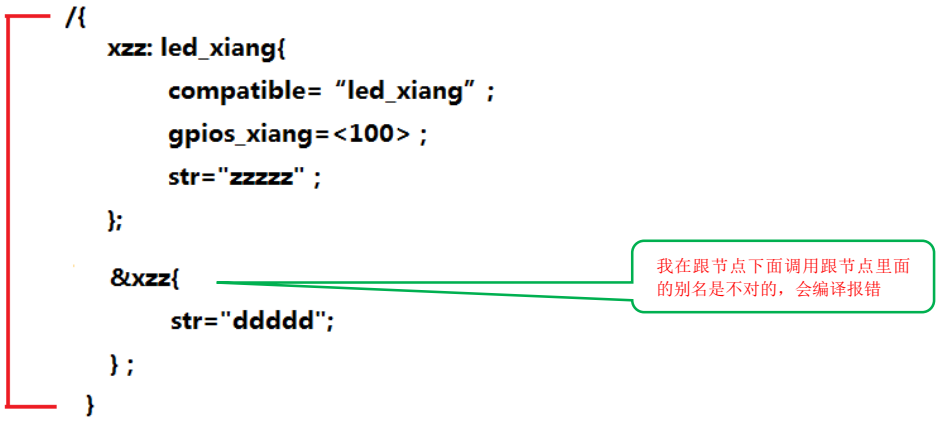
```
root@imx6qdlsolo:/mnt# insmo
enter devicetree_init
devicetreeProbe
pin1 = 100
---zzzzz---
```

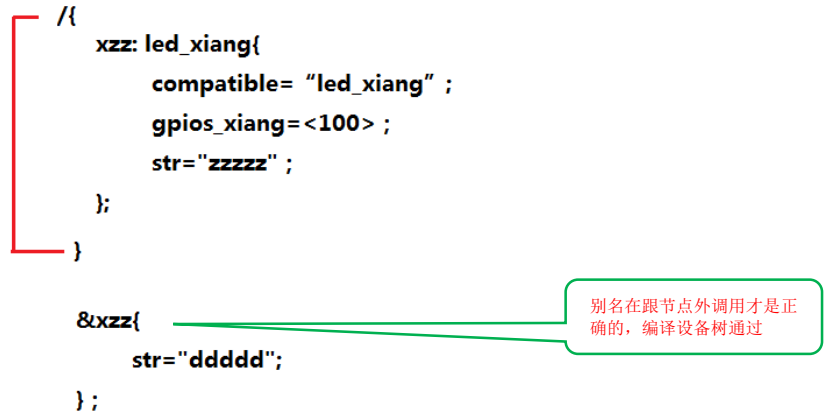




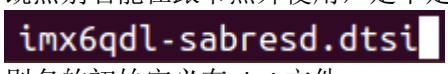


这就是别名引用的好处，下面注意调用别名的语法规则

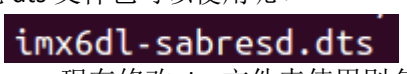




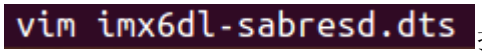
既然别名能在跟节点外使用，是不是其他 dts 文件也可以使用呢？



别名的初始定义在 dtsi 文件



现在修改 dts 文件来使用别名



打开 dts 文件

```
#include "imx6dl.dtsi"
#include "imx6qdl-sabresd.dtsi"

/ {
        model = "Freescale i.MX6 DualLite SABRE Smart Device Board";
        compatible = "fsl,imx6dl-sabresd", "fsl,imx6dl";
};


&xzz{
        str = "ddddd";
};


&battery {
        offset-charger = <1485>;
        offset-discharger = <1464>;
        offset-usb-charger = <1285>;
};
```

只要包含了我写好别名节点的 dtsi 文件

那么当前的 dts 文件就可以调用 dtsi 文件里面的别名节点，从而修改里面的属性

```
root@imx6qdlsolo:/mnt# insmod
enter devicetree_init
devicetreeProbe
pinl = 100
---ddddd---
```

执行没有问题

Pinctrl 用法

先介绍 of_find_property(设备节点,要查找的属性名称，默认填写 NULL)函数使用

```
        xzz:led_xiang{

                compatible = "led_xiang";
                gpios_xiang = <100>;
                str = "zzzzz";
                xzzcontrol;
        };
```

我在设备树增加了一个属性，但是这个属性没有赋任何值，这种格式，驱动可以用来判断这个属性在不在，从而决定执不执行相关代码

```
static int devicetreetest_probe(struct platform_device *pdev){

        int ret;

        printk("devicetreeProbe\n");

        ret = of_find_property(pdev->dev.of_node,"xzzcontrol",NULL);
        printk("ret = %d\n",ret);


        return 0;
}
```

设备树的纯属性信息获取就要用 of_find_property 函数，这里填写要确认哪一个属性是否存在设备树

```
root@imx6qdlsolo:/mnt# insm
enter devicetree_init
devicetreeProbe
ret = -1951055144
```

如果设备树存在驱动要找的属性值，那么 of_find_property 返回负值

```
xzz:led_xiang{

        compatible = "led_xiang";
        gpios_xiang = <100>;
        str = "zzzzz";
};
```

如果 xzzcontrol 属性不存在，of_find_property 返回 0

```
root@imx6qdlsolo:/mnt# insmo
enter devicetree_init
devicetreeProbe
ret = 0
```

一般情况下 of_find_property 函数的返回直接用 if 判断

```
xzz:led_xiang{

        compatible = "led_xiang";
        gpios_xiang = <100>;
        str = "zzzzz";
        xzzcontrol;
};
```

```
static int devicetreetest_probe(struct platform_device *pdev){


        printk("devicetreeProbe\n");

        if(!of_find_property(pdev->dev.of_node,"xzzcontrol",NULL))
        {
                printk("get xzzcontrol failed\n");
        }
        else
        {
                printk("get xzzcontrol success\n");
        }


        return 0;
}
```

一般设备树没有 xzzcontrol 属性，of_find 返回 0，取反之后返回 1 执行 failed

如果设备树有 xzzcontrol 属性，of_find 返回负数，取反之后还是负数，所以执行 else

```
root@imx6qdlsolo:/mnt# insm
enter devicetree_init
devicetreeProbe
get xzzcontrol success
```

这就表示设备树里面有 xzzcontrol 属性

```
xzz:led_xiang{

        compatible = "led_xiang";
        gpios_xiang = <100>;
        str = "zzzzz";
};
```

```
root@imx6qdlsolo:/mnt# insmod d
enter devicetree_init
devicetreeProbe
get xzzcontrol failed
root@imx6qdlsolo:/mnt#
```

这就表示设备树里面没有 xzzcontrol 属性

1.GPIO 初始化

内核最先在 imx6dl.dtsi 只定义了 iomux gpio 初始化驱动程序的名字

```
iomuxc: iomuxc@020e0000 {
    compatible = "fsl,imx6dl-iomuxc";
};
```

然后在 imx6qdl-sarebsd 里面调用了 imx6dl.dtsi 的 iomuxc 节点，从而给 iomuxc 节点增加了新的节点 epdc 节点

```
&iomuxc {
        epdc {
                pinctrl_epdc_0: epdcgrp-0 {
                        fsl,pins = <
                                MX6QDL_PAD_EIM_A16__EPDC_DATA00    0x80000000
                                MX6QDL_PAD_EIM_DA10__EPDC_DATA01   0x80000000
                                MX6QDL_PAD_EIM_DA12__EPDC_DATA02   0x80000000
                                MX6QDL_PAD_EIM_DA11__EPDC_DATA03   0x80000000
                                MX6QDL_PAD_EIM_LBA__EPDC_DATA04    0x80000000
                                MX6QDL_PAD_EIM_EB2__EPDC_DATA05    0x80000000
                                MX6QDL_PAD_EIM_CS0__EPDC_DATA06    0x80000000
                                MX6QDL_PAD_EIM_RW__EPDC_DATA07     0x80000000
                                MX6QDL_PAD_EIM_A21__EPDC_GDCLK     0x80000000
                                MX6QDL_PAD_EIM_A22__EPDC_GDSP      0x80000000
                                MX6QDL_PAD_EIM_A23__EPDC_GDOE      0x80000000
                                MX6QDL_PAD_EIM_A24__EPDC_GDRL      0x80000000
                                MX6QDL_PAD_EIM_D31__EPDC_SDCLK_P   0x80000000
                                MX6QDL_PAD_EIM_D27__EPDC_SDOE      0x80000000
                                MX6QDL_PAD_EIM_DA1__EPDC_SDLE      0x80000000
                                MX6QDL_PAD_EIM_EB1__EPDC_SDSHR     0x80000000
                                MX6QDL_PAD_EIM_DA2__EPDC_BDR0      0x80000000
                                MX6QDL_PAD_EIM_DA4__EPDC_SDCE0     0x80000000
                                MX6QDL_PAD_EIM_DA5__EPDC_SDCE1     0x80000000
                                MX6QDL_PAD_EIM_DA6__EPDC_SDCE2     0x80000000
                        >;
                };
        };
};
```

然后其余地方我也想自定义 GPIO 初始化状态，所以调用 iomuxc 再次增加了 GPIO 管脚节点

```
&iomuxc {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_hog>;

        imx6qdl-sabresd {
                pinctrl_hog: hoggrp {
                        fsl,pins = <
                                MX6QDL_PAD_NANDF_D0__GPIO2_IO00 0x80000000
                                MX6QDL_PAD_NANDF_D1__GPIO2_IO01 0x80000000
                                MX6QDL_PAD_NANDF_D2__GPIO2_IO02 0x80000000
                                MX6QDL_PAD_NANDF_D3__GPIO2_IO03 0x80000000
                                MX6QDL_PAD_GPIO_0__CCM_CLKO1    0x130b0
                                MX6QDL_PAD_NANDF_CLE__GPIO6_IO07 0x80000000
                                MX6QDL_PAD_NANDF_ALE__GPIO6_IO08 0x80000000
                                MX6QDL_PAD_ENET_TXD1__GPIO1_IO29 0x80000000
                                MX6QDL_PAD_EIM_D22__GPIO3_IO22  0x80000000
                                MX6QDL_PAD_ENET_CRS_DV__GPIO1_IO25 0x80000000
                                /*MX6QDL_PAD_EIM_D26__GPIO3_IO26 0x80000000*/
```

到现在，iomuxc 有了子节点 edpc 和子节点 imxpqdl-sabresd

驱动程序如何去计算子节点数量然后去自动配置设备树这些 GPIO 值呢？

我们先了解下 of_get_child_count(主设备节点) 函数使用

```
xzz:led_xiang{

        compatible = "led_xiang";
        gpios_xiang = <100>;
        str = "zzzzz";
};
```

自定义主节点下面没有其他子节点

```
static int devicetreetest_probe(struct platform_device *pdev){

        int ret;

        printk("devicetreeProbe\n");

        ret = of_get_child_count(pdev->dev.of_node);
        printk("node num = %d\n",ret);



        return 0;
}
```

返回值是 int 类型，执行 of_get_child_count 之后没有查找到子节点数量

```
root@imx6qdlsolo:/mnt# insmo
enter devicetree_init
devicetreeProbe
node num = 0
```

```
xzz:led_xiang{

        compatible = "led_xiang";
        gpios_xiang = <100>;
        str = "zzzzz";

        xiang@1{

        };

        xiang@2{

        };
};
```

自定义主节点下，建立两个子节点

```
root@imx6qdlsolo:/mnt# i
enter devicetree_init
devicetreeProbe
node num = 2
```

查找到了子节点数量 2 个

如果我再在子节点下面建立子节点呢？

```
xzz:led_xiang{

        compatible = "led_xiang";
        gpios_xiang = <100>;
        str = "zzzzz";

        xiang@1{

                xzzsubnode@2{

                };
        };

        xiang@2{

        };
};
```

像这样，子节点下添加子节点

编译能正常通过

```
root@imx6qdlsolo:/mnt# i
enter devicetree_init
devicetreeProbe
node num = 2
```

这种子节点下的子节点是不会计算进去的，of_get_child_count 只计算 compatible 下面的子节点

```
xzz:led_xiang{

        compatible = "led_xiang";
        gpios_xiang = <100>;
        str = "zzzzz";

        xiang@1{

        };

        xiang@2{

        };
};
```

dtsi 文件里面有了两个子节点

```
&xzz{
        str = "ddddd";
        xzzsubnode@2{

        };
};
```

调用 xzz 节点里面添加子节点

调用 xzz 节点的 dts 文件又添加了个子节点

```
root@imx6qdlsolo:/mnt#
enter devicetree_init
devicetreeProbe
node num = 3
```

调用节点里面的子节点 of_get_child_count 也能计算进去

```
/{
    nede{
        node@1{
        }
        node@2{
        }
    }
    xzz{
        node@1
        {
        }
    }
}
```

在根节点下可以添加一个子节点，在子节点下可以再添加一个子节点

```
/{
    node{
        node@1{
            node@2{
            }
        }
    xzz{
        }
    }
}
```

如果是在子节点下，最多只能添加一个子节点，如果子节点下添加子节点的子节点就不行

所以设备树最多两个子节点

用 for_each_child_of_node(父节点，要读取的子节点) 宏 循环读取每个子节点

```
static int devicetreetest_probe(struct platform_device *pdev){

        int ret;
        struct device_node *child;
        const char *buff;
        printk("devicetreeProbe\n");

        ret = of_get_child_count(pdev->dev.of_node);
        printk("node num = %d\n",ret);

        for_each_child_of_node(pdev->dev.of_node, child)
        {
                of_property_read_string(child,"str",&buff);
                printk("------%s---------\n",buff);
        }


        return 0;
}
```

for_each_child_of_node 功能是你把父节点传进去 pdev->dev.of_node

```
xzz:led_xiang{

        compatible = "led_xiang";
        gpios_xiang = <100>;
        str = "zzzzz";

        xiang@1{

                str= "xiang@1" ;
        };

        xiang@2{
                str= "xiang@2" ;
        };
};
```

该函数会自动循环寻找父节点下面有多少个子节点，有多少个子节点就循环几次，记住每次循环的子节点你都要记得用 device_node 变量去获取到

```
root@imx6qdlsolo:/mnt# insmod d
enter devicetree_init
devicetreeProbe
node num = 2
------xiang@1---------
------xiang@2---------
```

输出结果和我设备树两个子节点的 str 一致，没有问题

现在回过头来我们再来看 iomuxc 驱动程序如何实现的

```
iomuxc: iomuxc@020e0000 {
    compatible = "fsl,imx6dl-iomuxc";
};
```

```
&iomuxc {
        epdc {
                pinctrl_epdc_0: epdcgrp-0 {
                        fsl,pins = <
                                MX6QDL_PAD_EIM_A16__EPDC_DATA00  0x80000000
                                MX6QDL_PAD_EIM_DA10__EPDC_DATA01  0x80000000
                                MX6QDL_PAD_EIM_DA12__EPDC_DATA02  0x80000000
                                MX6QDL_PAD_EIM_DA11__EPDC_DATA03  0x80000000
                                MX6QDL_PAD_EIM_LBA__EPDC_DATA04  0x80000000
                                MX6QDL_PAD_EIM_EB2__EPDC_DATA05  0x80000000
                                MX6QDL_PAD_EIM_CS0__EPDC_DATA06  0x80000000
                                MX6QDL_PAD_EIM_RW__EPDC_DATA07  0x80000000
                                MX6QDL_PAD_EIM_A21__EPDC_GDCLK  0x80000000
                                MX6QDL_PAD_EIM_A22__EPDC_GDSP  0x80000000
                                MX6QDL_PAD_EIM_A23__EPDC_GDOE  0x80000000
                                MX6QDL_PAD_EIM_A24__EPDC_GDRL  0x80000000
                                MX6QDL_PAD_EIM_D31__EPDC_SDCLK_P  0x80000000
                                MX6QDL_PAD_EIM_D27__EPDC_SDOE  0x80000000
                                MX6QDL_PAD_EIM_DA1__EPDC_SDLE  0x80000000
                                MX6QDL_PAD_EIM_EB1__EPDC_SDSHR  0x80000000
                                MX6QDL_PAD_EIM_DA2__EPDC_BDR0  0x80000000
                                MX6QDL_PAD_EIM_DA4__EPDC_SDCE0  0x80000000
                                MX6QDL_PAD_EIM_DA5__EPDC_SDCE1  0x80000000
                                MX6QDL_PAD_EIM_DA6__EPDC_SDCE2  0x80000000
                        >;
                };
        };
};
&iomuxc {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_hog>;

        imx6qdl-sabresd {
                pinctrl_hog: hoggrp {
                        fsl,pins = <
                                MX6QDL_PAD_NANDF_D0__GPIO2_IO00 0x80000000
                                MX6QDL_PAD_NANDF_D1__GPIO2_IO01 0x80000000
                                MX6QDL_PAD_NANDF_D2__GPIO2_IO02 0x80000000
                                MX6QDL_PAD_NANDF_D3__GPIO2_IO03 0x80000000
                                MX6QDL_PAD_GPIO_0__CCM_CLKO1    0x130b0
                                MX6QDL_PAD_NANDF_CLE__GPIO6_IO07 0x80000000
                                MX6QDL_PAD_NANDF_ALE__GPIO6_IO08 0x80000000
                                MX6QDL_PAD_ENET_TXD1__GPIO1_IO29 0x80000000
                                MX6QDL_PAD_EIM_D22__GPIO3_IO22  0x80000000
                                MX6QDL_PAD_ENET_CRS_DV__GPIO1_IO25 0x80000000
                                /*MX6QDL_PAD_EIM_D26__GPIO3_IO26 0x80000000*/
```

```
iomuxc:iomuxc@020e0000 {
    epdc{
        pinctrl_epdc_0:epdcgrp-0{
            fsl,pins=<.....
                      .....
                      .....>;
        };
    };

    pinctrl-name = "default";
    pinctrl-0 = <&pinctrl_hog>;
    imx6qdl-sabresd{
        pinctrl_hog:hoggrp{
            fsl,pins=<.....
                      .....
                      .....>
        }
    }
}
```

左边设备树展开就是右边这种形式，其实都是写在一个设备节点里面的。

下面看看内核驱动文件是怎么去识别的

```c
static struct of_device_id imx6dl_pinctrl_of_match[] = {
    { .compatible = "fsl,imx6dl-iomuxc", },
    { /* sentinel */ }
};

static int imx6dl_pinctrl_probe(struct platform_device *pdev)
{
    return imx_pinctrl_probe(pdev, &imx6dl_pinctrl_info);
}
```

```c
ret = imx_pinctrl_probe_dt(pdev, info);
```

```c
static int imx_pinctrl_probe_dt(struct platform_device *pdev,
                struct imx_pinctrl_soc_info *info)
{
    struct device_node *np = pdev->dev.of_node;
    struct device_node *child;
    u32 nfuncs = 0;
    u32 i = 0;

    if (!np)
        return -ENODEV;

    nfuncs = of_get_child_count(np);
    if (nfuncs <= 0) {
        dev_err(&pdev->dev, "no functions defined\n");
        return -EINVAL;
    }

    info->nfunctions = nfuncs;
    info->functions = devm_kzalloc(&pdev->dev, nfuncs * sizeof(struct imx_pmx_func),
                    GFP_KERNEL);
    if (!info->functions)
        return -ENOMEM;

    info->ngroups = 0;
    for_each_child_of_node(np, child)
        info->ngroups += of_get_child_count(child);
```

在这里将 iomux 设备节点的数据获取出来，进行一个一个的 GPIO 管脚映射，所以这里的 for_each_child_of_node 函数的功能我上一页已经讲清楚了

```
MX6QDL_PAD_EIM_A16__GPIO2_IO22   0X80000000 /*sw5*/
/*MX6QDL_PAD_EIM_D26__GPIO3_IO26   0x80000000*/ /*uart2 txd*/
/*MX6QDL_PAD_EIM_D27__GPIO3_IO27   0x80000000*/ /*uart2 rxd xiang*/
MX6QDL_PAD_DISP0_DAT5__GPIO4_IO26 0x80000000 /*WLAN*/
MX6QDL_PAD_DISP0_DAT6__GPIO4_IO27 0x80000000 /*WLAN REG_ON*/
MX6QDL_PAD_DISP0_DAT7__GPIO4_IO28 0x80000000 /*WLAN*/
MX6QDL_PAD_DISP0_DAT10__GPIO4_IO31 0x80000000 /*WLAN*/
MX6QDL_PAD_DISP0_DAT11__GPIO5_IO05 0x80000000 /*WLAN*/
MX6QDL_PAD_KEY_ROW3__GPIO4_IO13 0x80000000   /*heat_release or i2c2_dat*/
```

所以我只要调用 iomux 在 fsl,pins 属性里面写入我要初始化的引脚，内核启动后都会自动计算初始化引脚的个数，帮我进行初始化。

中断使用

设备树中断格式

```
mma8451@1c {
        compatible = "fsl,mma8451";
        reg = <0x1c>;
        position = <0>;
        vdd-supply = <&reg_sensor>;
        vddio-supply = <&reg_sensor>;
        interrupt-parent = <&gpio1>;
        interrupts = <18 8>;
        interrupt-route = <1>;
};
```

设备树标准中断写法是，先确定该 IO 口属于哪一个中断控制器，这里指定 gpio1 中断控制器

这个 8 就是该中断是上升沿触发还是下降沿触发还是电平触发还是双边沿触发

然后确定该中断控制器的中断号，这里是 18，也是就是 gpio1_18 引脚作为中断引脚

#adderss-cells，#size-cells 关键字使用

```
/{
        #address-cells = <1>;
        #size-cells = <1>;
        memory{
                reg = <0x30000000 0x4000000 0 4096>
        };
};
```

#address-cells 和 size-cells 一定要写在所有子节点前面

Address-cell=<1>表示下面的 reg 属性<>里面第 1 个数据表示地址

size-cell=<1>表示下面的 reg 属性<>第 2 个数据表示大小

下面总结 reg 属性

```
/{
        #address-cells = <1>;
        #size-cells = <1>;
        memory{
                reg = <0x30000000 0x4000000 0 4096>
        };
};
```

所以 address-cell=1 和 size-cells=1 的缘故，表示 reg 是一个地址加一个大小值为一组数据

另一个地址和一个大小为第 2 组数据

```
/{
    #address-cells = <1>;
    #size-cells = <1>;
    memory{
        reg = <0x30000000 0x4000000 0 4096>
        #address-cells=<1>;
        #size-cells=<0>;
        subnode{
            reg=<0>;
        };
    };
};
```

在子节点下面也可以增加 address-cell 和 size-cells，这种情况下就表示子节点下面的子节点 reg 属性范围

因为#size-cells=<0>，所以 reg 的每项数据都表示地址，不表示大小

```
regulators {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <0>;

        reg_usb_otg_vbus: regulator@0 {
                compatible = "regulator-fixed";
                reg = <0>;
                regulator-name = "usb_otg_vbus";
```

这就是个很明显的案例，看 reg 属性范围。

下面说说中断的设备树节点

```
/{
    xzz : pic#10000000{
        interrupt-controller;
    };

    xzznode{
        compatible="....";
        interrupt-parent=<&xzz>;
    };

};
```

比如我在这个设备里面定义了一个中断控制器，写入关键字 interrupt-controller 就表示这个设备节点是一个中断控制器，用别名 xzz 好让其它设备节点调用

然后我另外一个设备节点的 GPIO 或者外设使用到了 pic#10000000 这个中断控制器，那么我就要用 interrupt-parent 去引用它，所以 &xzz

设备树设置按键中断的方法

```
/{
    button{
        compatible = ".....";
        interrupt-parent=<&gpio6>;    //我按键使用gpio6这组引脚的中断控制器
        interrupts = <8 IRQ_TYPE_EDGE_RISING>; //使用8号中断，上升沿触发
    };
};
```

如果按键在 GPIO6_8 引脚上，那么先设置 GPIO6 的中断控制器

Interrupts = <IO 口号，触发方式>

比如 IMX6 的中断控制器设备树规则

```
gpio6: gpio@020b0000 {
    compatible = "fsl,imx6q-gpio", "fsl,imx35-gpio";
    reg = <0x020b0000 0x4000>;
    interrupts = <0 76 IRQ_TYPE_LEVEL_HIGH>,
                 <0 77 IRQ_TYPE_LEVEL_HIGH>;
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
};
```

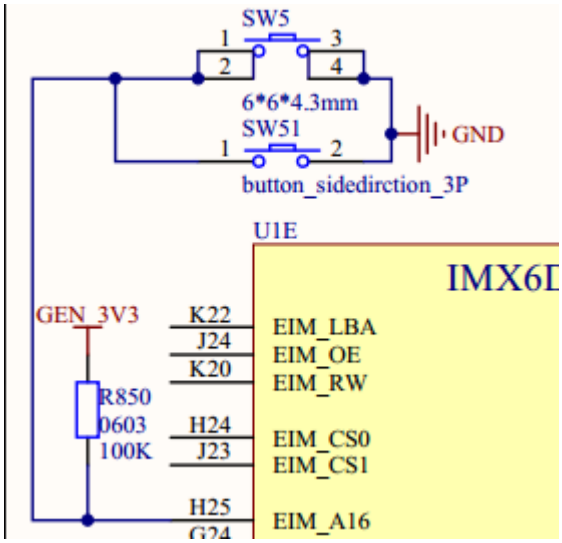
在最初总的 dtsi 文件定义了 gpio6 设备节点引用

而且 gpio6 这组 IO 控制器具备中断功能，所以这里指定了中断控制器字符

Interrupt-cells =2 表示引用 gpio6 的节点使用 interrupts 字符时用两个 32 位数据表示

```
egalax_ts@04 {
        compatible = "eeti,egalax_ts";
        reg = <0x04>;
        interrupt-parent = <&gpio6>;
        interrupts = <8 2>;
        wakeup-gpios = <&gpio6 8 0>;
};
```

用 IMX6 按钮来举例外部中断使用



GPIO2_IO22	EIM_A16 (ALT5)

这里使用的是 GPIO2_22，确定了中断控制器是 GPIO2，中断号是 22

```
led_xiang{

        compatible = "led_xiang";
        interrupt-parent = <&gpio2>;
        interrupts = <22 IRQ_TYPE_EDGE_RISING>;

};
```

设备树节点写入使用的中断控制器 gpio2

按键接在 GPIO2 的 22 引脚上，所以中断号是 22

中断触发方式上升沿

下面介绍驱动层如何获取中断号

```
#include <linux/fs.h>
/*MKDEV转换设备号数据类型的宏定义*/
#include <linux/kdev_t.h>
/*定义字符设备的结构体*/
#include <linux/cdev.h>
/*分配内存空间函数头文件*/
//#include <linux/slab.h>
#include <linux/kernel.h>
#include <linux/uaccess.h>
#include <asm/uaccess.h>
/*包含函数device_create 结构体class等头文件*/
#include <linux/device.h>
#include <linux/gpio.h>
#include <linux/delay.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_gpio.h>
#include <linux/of_platform.h>
#include <asm/irq.h>
#include <linux/irqreturn.h>
#include <linux/interrupt.h>
#include <linux/sched.h>
MODULE_LICENSE("Dual BSD/GPL");
```

定义一大堆头文件

```
static struct of_device_id dt_button[]={
        { .compatible = "led_xiang",},
        {},
};

MODULE_DEVICE_TABLE(of, dt_button);
static struct platform_driver dt_button_driver = {
        .driver = {
                .name = "dt_button",
                .owner = THIS_MODULE,
                .of_match_table = dt_button,
        },
        .probe = dt_button_probe,
        .remove = dt_button_remove,
};

static int __init button_init(void){

        printk("button devicetree_init\n");
        return platform_driver_register(&dt_button_driver);
}

static int __exit button_exit(void){

        printk("button devicetree_exit \n");
        platform_driver_unregister(&dt_button_driver);
        return 0;
}

module_init(button_init);
module_exit(button_exit);
```

这是常规驱动匹配设备树代码，重点代码在下一页

```c
static irqreturn_t buttons_irq(int irq,void* dev_id){

    printk("enter buttons irq process\n");
    printk("irq=%d\n",irq);

    return IRQ_HANDLED;
}

int button1_release(struct inode *inode, struct file *file){
        printk("release button1\n");
        return 0;
}

static int dt_button_probe(struct platform_device *pdev){

        struct resource *res;
        int irqres;

        printk("dt_button_probe\n");

        /*****************button1 add******************/
        res = platform_get_resource(pdev,IORESOURCE_IRQ,0);
        irqres = request_irq(res->start,buttons_irq,0,"SW5",1);

        return 0;
}

static int dt_button_remove(struct platform_device *pdev){

        return 0;
}
```

使用 platform_get_resource 获取设备节点，填入 IORESOURCE_IRQ 获取中断部分内容

这里填入 0，就是获取 interrupts 第 0 个尖括号的中断号

res->start 就是获取的中断号，可以用%d 打印出来，这里就是申请这个中断号

中断处理函数

在设备树里面就设置了中断边沿触发方式，所以这里就填 0 就是了

```
led_xiang{

        compatible = "led_xiang";
        interrupt-parent = <&gpio2>;
        interrupts = <22 IRQ_TYPE_EDGE_RISING>;

};
```

```
root@imx6qdlsolo:/mnt# insmod dt_b
button devicetree_init
dt_button_probe
enter buttons irq process
irq=214
res->start = 214
root@imx6qdlsolo:/mnt# enter butto
irq=214
enter buttons irq process
irq=214
```

这就是这个按键的外部中断号

加载驱动程序之后，按下按钮中断程序已经执行。

如果是两个外部中断怎么做？我们用按键来模拟两个外部中断

```
fs4412-key{
    compatible = "fs4412,key";
    interrupt-parent = <&gpx1>;
    interrupts = <1 2>,<2 2>;
};
```

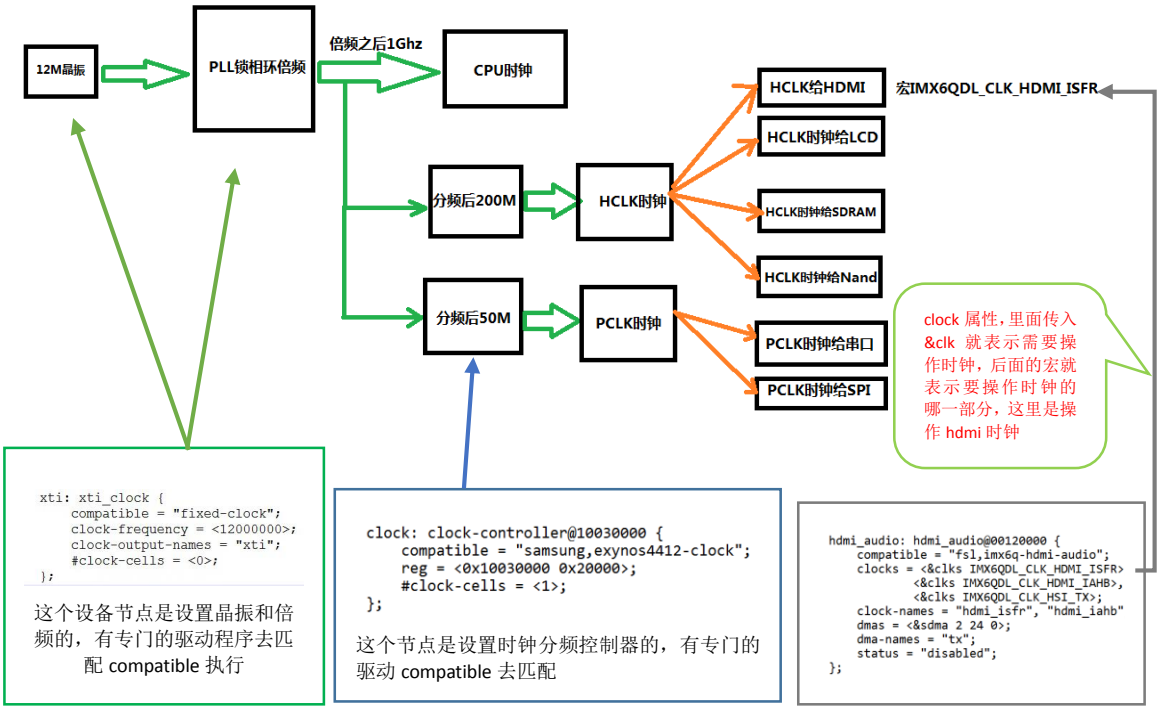
如果是两个外部中断，那么这两个外部中断必须在同一组 GPIO 里面，这样设备树才能用同一个中断控制器去映射，然后 interrupts 写两个尖括号，代表两个中断号

```
    res1 = platform_get_resource(pdev,IORESOURCE_IRQ, 0);
    res2 = platform_get_resource(pdev,IORESOURCE_IRQ, 1);
```

在驱动层就用 0，1 来区别获取设备树的两个中断号

设备树时钟使用方法

下面以 4412 设备树为例来讲解时钟



下面以 IMX6 为例来实际操作时钟案例

```
led_xiang{

        compatible = "led_xiang";
        clocks = <&clks IMX6QDL_CLK_CKO>;

};
```

这里 clock 关键字属性就是要求写入&clk 表示操作时钟，宏 IMX6QDL_CLK_CKO 表示我要操作摄像头时钟

然后驱动程序 compatible 去匹配，获取 clocks 属性里面的数据，这个&clks 在其他 dts 文件已经定义好的引用节点

```
static int devicetreetest_probe(struct platform_device *pdev){

        struct clk *clk;

        printk("devicetreeProbe\n");

        clk = of_clk_get(pdev->dev.of_node, 0);
        clk_prepare_enable(clk);
        printk("disable clk\n");
        clk_disable_unprepare(clk);

        return 0;
}
```

of_clk_get 就是获取设备节点的 clock 属性的值

填 0 就是获取 clocks 第 0 项<>括号里面数据

填入 clk 是将 IMX6QDL_CLK_CKO 摄像头时钟打开

关闭摄像头时钟

```
led_xiang{

        compatible = "led_xiang";
        clocks = <&clks IMX6QDL_CLK_CKO>;

};
```

如果设备树时钟节点有多个时钟怎么操作？

设备节点只有 1 个时钟的操作方法，现在回顾下

```
led_xiang{

        compatible = "led_xiang";
        clocks = <&clks IMX6QDL_CLK_CKO>;
};
```

这里只有一个时钟

of_count_phandle_with_args(设备节点，属性，名称) //返回节点中时钟个数

```
static int devicetreetest_probe(struct platform_device *pdev){

        struct clk *clk;
        int ret;

        printk("devicetreeProbe\n");

        ret = of_count_phandle_with_args(pdev->dev.of_node, "clocks","#clock-cells");
        printk("clock node num = %d\n",ret);

        clk = of_clk_get(pdev->dev.of_node, 0);
        clk_prepare_enable(clk);
        printk("disable clk\n");
        clk_disable_unprepare(clk);

        return 0;
}
```

用 of_count_phandle_with_args 计算设备节点时钟个数

返回 1 表示 1 个时钟

1 个时钟就是编号 0，所以这里获取填入 0，就获取 IMX6QDL_CLK_CKO 时钟

```
root@imx6qdlsolo:/mnt# insmod
enter devicetree_init
devicetreeProbe
clock node num = 1
disable clk
root@imx6qdlsolo:/mnt#
```

现在加入第 2 个时钟

```
led_xiang{

        compatible = "led_xiang";
        clocks = <&clks IMX6QDL_CLK_CKO>,<&clks 201>;
};
```

这里加入了 clks 时钟的 IMX6QDL_CLK_CKO 摄像头时钟和 201 时钟，这里就有两个时钟了

```c
static int devicetreetest_probe(struct platform_device *pdev){

        struct clk *clk;
        int ret;

        printk("devicetreeProbe\n");

        ret = of_count_phandle_with_args(pdev->dev.of_node, "clocks","#clock-cells");
        printk("clock node num = %d\n",ret);

        clk = of_clk_get(pdev->dev.of_node, 0);
        clk_prepare_enable(clk);
        printk("disable clk\n");
        clk_disable_unprepare(clk);

        clk = of_clk_get(pdev->dev.of_node, 1);
        clk_prepare_enable(clk);
        printk("disable 201 clk\n");
        clk_disable_unprepare(clk);

        return 0;
}
```

返回 2，表示获取到了两个时钟

0 表示获取 IMX6QDL_CLK_CKO 摄像头时钟

1 表示获取 201 时钟，如果还有多个时钟就循环用 of_clk_get 0 1 2 3...这样获取下去

```
enter devicetree_init
devicetreeProbe
clock node num = 2
disable clk
disable 201 clk
```

这就是多个时钟的操作方法。

设备树使用的其它函数和零散的方法请参考我的 github
IMX6_devicetree_test/IMX6_devicetree_test/**imx6_devicetree_document**/文档