目录

Linux DTS(Device Tree Source)设备树详解之二(dts匹配及发挥作用的流程篇)

一个dts文件确定一个项目，多个项目可以包含同一个dtsi文件。找到该项目对应的dts文件即找到了该设备树的根节点

kernel\arch\arm\boot\dts\qcom\sdm630-mtp.dts

```
/* Copyright (c) 2017, The Linux Foundation. All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 and
 * only version 2 as published by the Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */


/dts-v1/;

#include "sdm630.dtsi"
#include "sdm630-mtp.dtsi"
//#include "sdm660-external-codec.dtsi"
#include "sdm660-internal-codec.dtsi"
#include "synaptics-dsx-i2c.dtsi"


/ {
  model = "Qualcomm Technologies, Inc. SDM 630 PM660 + PM660L MTP";
  compatible = "qcom,sdm630-mtp", "qcom,sdm630", "qcom,mtp";
  qcom,board-id = <8 0>;
  qcom,pmic-id = <0x0001001b 0x0101011a 0x0 0x0>,
      <0x0001001b 0x0201011a 0x0 0x0>;
};
```

```
&tavil_snd {
  qcom,msm-mbhc-moist-cfg = <0>, <0>, <3>;

};
```

当然devicetree的根节点也是需要和板子进行匹配的，这个匹配信息存放在sbl（second boot loader）中， 对应dts文件中描述的board-id（上面代码中的qcom,board-id属性），通过共享内存传递给bootloader， 由bootloader将此board-id匹配dts文件（devicetree的根节点文件），将由dtc编译后的dts文件（dtb文件）加载到内存， 然后在kernel中展开dts树，并且挂载dts树上的所有设备。

（ps：cat /proc/cmdline 查看cmdline）

Dts中相关符号的含义

/ - 根节点

@ - 如果设备有地址，则由此符号指定

& - 引用节点

: - 冒号前的label是为了方便引用给节点起的别名，此label一般使用为&label

, - 属性名称中可以包含逗号。如compatible属性的名字 组成方式为”[manufacturer],[model]”，加入厂商名是为了避免重名。自定义属性名中通常也要有厂商名，并以逗号分隔。

- #并不表示注释。如 #address-cells ，#size-cells 用来决定reg属性的格式。

 - 空属性并不一定表示没有赋值。如 interrupt-controller 一个空属性用来声明这个node接收中断信号数据类型

”” - 引号中的为字符串，字符串数组：”strint1”,”string2”,”string3”

< > - 尖括号中的为32位整形数字，整形数组<12 3 4>

[ ] - 方括号中的为32位十六进制数，十六机制数据[0x11 0x12 0x13] 其中0x可省略

DTS中几个难理解的属性的解释

地址

设备的地址特性根据一下几个属性来控制：

reg #address-cells #size-cells

reg意为region，区域。格式为：

reg = <address1length1 [address2 length2] [address3 length3]>;

父类的address-cells和size-cells决定了子类的相关属性要包含多少个cell，如果子节点有特殊需求的话，可以自己再定义，这样就可以摆脱父节点的控制。

address-cells决定了address1/2/3包含几个cell，size-cells决定了length1/2/3包含了几个cell。本地模块例如：

```
spi@10115000{
        compatible = "arm,pl022";
        reg = <0x10115000 0x1000 >;
};
```

位于0x10115000的SPI设备申请地址空间，起始地址为0x10115000，长度为0x1000，即属于这个SPI设备的地址范围是0x10115000~0x10116000

实际应用中，有另外一种情况，就是通过外部芯片片选激活模块。例如，挂载在外部总线上，需要通过片选线工作的一些模块：

```
external-bus{
    #address-cells = <2>
    #size-cells = <1>;

    ethernet@0,0 {
        compatible = "smc,smc91c111";
        reg = <0 0 0x1000>;
    };

    i2c@1,0 {
        compatible ="acme,a1234-i2c-bus";
        #address-cells = <1>;
        #size-cells = <0>;
        reg = <1 0 0x1000>;
        rtc@58 {
            compatible ="maxim,ds1338";
            reg = <58>;
        };
    };

    flash@2,0 {
        compatible ="samsung,k8f1315ebm", "cfi-flash";
        reg = <2 0 0x4000000>;
    };
};
```

external-bus使用两个cell来描述地址，一个是片选序号，另一个是片选序号上的偏移量。而地址空间长度依然用一个cell来描述。 所以以上的子设备们都需要3个cell来描述地址空间属性—片选、偏移量、地址长度。在上个例子中，有一个例外，就是i2c控制器模块下的rtc模块。 因为I2C设备只是被分配在一个地址上，不需要其他任何空间，所以只需要一个address的cell就可以描述完整，不需要size-cells。

当需要描述的设备不是本地设备时，就需要描述一个从设备地址空间到CPU地址空间的映射关系，这里就需要用到ranges属性。还是以上边的external-bus举例：

```
#address-cells= <1>;
#size-cells= <1>;
...
external-bus{
    #address-cells = <2>
    #size-cells = <1>;
    ranges = <0 0  0x10100000  0x10000     // Chipselect 1,Ethernet
              1 0  0x10160000  0x10000     // Chipselect 2, i2c controller
              2 0  0x30000000  0x1000000>; // Chipselect 3, NOR Flash
};
```

ranges属性为一个地址转换表。表中的每一行都包含了子地址、父地址、在子地址空间内的区域大小。他们的大小（包含的cell） 分别由子节点的address-cells的值、父节点的address-cells的值和子节点的size-cells来决定。以第一行为例：

- 0 0 两个cell，由子节点external-bus的address-cells=<2>决定；
- 0x10100000 一个cell，由父节点的address-cells=<1>决定；
- 0x10000 一个cell，由子节点external-bus的size-cells=<1>决定。

最终第一行说明的意思就是：片选0，偏移0（选中了网卡），被映射到CPU地址空间的0x10100000~0x10110000中，地址长度为0x10000。

中断

描述中断连接需要四个属性：

1. interrupt-controller 一个空属性用来声明这个node接收中断信号；
2. #interrupt-cells 这是中断控制器节点的属性，用来标识这个控制器需要几个单位做中断描述符；
3. interrupt-parent 标识此设备节点属于哪一个中断控制器，如果没有设置这个属性，会自动依附父节点的；
4. interrupts 一个中断标识符列表，表示每一个中断输出信号。

如果有两个，第一个是中断号，第二个是中断类型，如高电平、低电平、边缘触发等触发特性。对于给定的中断控制器，应该仔细阅读相关文档来确定其中断标识该如何解析。一般如下：

二个cell的情况

第一个值： 该中断位于他的中断控制器的索引；

第二个值：触发的type

固定的取值如下：

```
        1 = low-to-high edge triggered          2 = high-to-low edge triggered
        4 = active high level-sensitive          8 = active low level-
sensitive
```

三个cell的情况

第一个值：中断号

第二个值：触发的类型

第三个值：优先级，0级是最高的，7级是最低的；其中0级的中断系统当做 FIQ处理。

其他

除了以上规则外，也可以自己加一些自定义的属性和子节点，但是一定要符合以下的几个规则：

1. 新的设备属性一定要以厂家名字做前缀，这样就可以避免他们会和当前的标准属性存在命名冲突问题；

2．新加的属性具体含义以及子节点必须加以文档描述，这样设备驱动开发者就知道怎么解释这些数据了。描述文档中 必须特别说明compatible的value的意义，应该有什么属性，可以有哪个（些）子节点，以及这代表了什么设备。每个独立的compatible都应该由单独的解释。

新添加的这些要发送到devicetree-discuss\@lists.ozlabs.org邮件列表中进行review，并且检查是否会在将来引发其他的问题。

DTS设备树描述文件中什么代表总线，什么代表设备

一个含有compatible属性的节点就是一个设备。包含一组设备节点的父节点即为总线。

由DTS到device_register的过程

dts描述的设备树是如何通过register_device进行设备挂载的呢？我们来进行一下代码分析

在arch/arm/mach-**/**.c找到DT_MACHINE_START 和 MACHINE_END 宏，如下：

```
DT_MACHINE_START(******_DT, "************* SoC (Flattened DeviceTree)")
    .atag_offset    = 0x100,
    .dt_compat    =******_dt_compat,                // 匹配dts
    .map_io        =******_map_io,                   // 板级地址内存映射，linux mmu
    .init_irq    =irqchip_init,                     // 板级中断初始化.
    .init_time    =******_timer_and_clk_init,       // 板级时钟初始化,如ahb,apb等
    .init_machine   = ******_dt_init,               // 这里是解析dts文件入口.
    .restart    =******_restart,                    // 重启，看门狗寄存器相关可以在这里设置
MACHINE_END
```

其中.dt_compat = **_dt_compat 这个结构体是匹配是哪个dts文件，如：

```
static const char * const ******_dt_compat[] = {
    "******,******-soc",
    NULL
};
```

这个”**,**-soc” 字符串可以在我们的dts的根节点下可以找到.

好了，我们来看看init_machine = **_dt_init 这个回调函数.

arch/arm/mach-xxx/xxx.c ：

void __init xxxxx_dt_init(void)

xxxxx_dt_init(void) -> of_platform_populate(NULL,of_default_bus_match_table, NULL, NULL); of_default_bus_match_table 这个是structof_device_id的全局变量.

```
const struct of_device_id of_default_bus_match_table[] = {
        { .compatible = "simple-bus",},
    #ifdef CONFIG_ARM_AMBA
        { .compatible = "arm,amba-bus",},
    #endif /* CONFIG_ARM_AMBA */
        {} /* Empty terminated list */
```

```
};
```

我们设计dts时，把一些需要指定寄存器基地址的设备放到以compatible = “simple-bus”为匹配项的设备节点下．下面会有介绍为什么．

drivers/of/platform.c :

int of_platform_populate(…)-> of_platform_bus_create(…)

// 在这之前，会有of_get_property(bus,”compatible”, NULL) // 检查是否有compatible，如果没有，返回，继续下一个，也就是说没有compatible，这个设备不会被注册

```
for_each_child_of_node(root, child) {
        printk("[%s %s %d]child->name = %s, child->full_name = %s\n", __FILE__, __func__,__LINE__,
        rc = of_platform_bus_create(child,matches, lookup, parent, true);
        if (rc)
            break;
    }
```

论询dts根节点下的子设备，每个子设备都要of_platform_bus_create(…); 全部完成后，通过 of_node_put(root);释放根节点，因为已经处理完毕;

drivers/of/platform.c : of_platform_bus_create(bus, …)

```
dev = of_platform_device_create_pdata(bus, bus_id,platform_data, parent); // 我们跳到 3-1步去运行
    if (!dev || !of_match_node(matches, bus))  // 就是匹配
                                               // dt_compat    = ******_dt_compat, 也就是 compatible
                                               // 如果匹配成功，以本节点为父节点，继续轮询本节点下的所有子节
        return 0;

    for_each_child_of_node(bus, child) {
        pr_debug("   create child:%s\n", child->full_name);
        rc = of_platform_bus_create(child,matches, lookup, &dev->dev, strict);  // dev->dev以本节点为
        if (rc) {
            of_node_put(child);
            break;
        }
    }
```

drivers/of/platform.c : of_platform_device_create_pdata(…)

```
if (!of_device_is_available(np))   // 查看节点是否有效，如果节点有'status'属性，必须是okay或者是ok，才是有
        return NULL;

    dev = of_device_alloc(np, bus_id, parent);  // alloc设备，设备初始化．返回dev，所有的设备都可认为是p
    if (!dev)
        return NULL;

    #if defined(CONFIG_MICROBLAZE)
        dev->archdata.dma_mask = 0xffffffffUL;
    #endif
```

```
        dev->dev.coherent_dma_mask =DMA_BIT_MASK(32); // dev->dev 是 struct device. 继续初始化
        dev->dev.bus =&platform_bus_type;     //
        dev->dev.platform_data =platform_data;

    printk("[%s %s %d] of_device_add(device register)np->name = %s\n", __FILE__, __func__, __LINE__
    if (of_device_add(dev) != 0){       // 注册device,of_device_add(...) --> device_add(...) // Thi
        platform_device_put(dev);
        return NULL;

    }
```

3-1-1. drivers/of/platform.c : of_device_alloc(…)

1. alloc platform_device *dev
2. 如果有reg和interrupts的相关属性，运行of_address_to_resource 和 of_irq_to_resource_table，加入到dev->resource

```
dev->num_resources = num_reg +num_irq;
     dev->resource = res;
     for (i = 0; i < num_reg; i++, res++) {
         rc = of_address_to_resource(np,i, res);
         /*printk("[%s %s %d] res->name = %s, res->start = 0x%X, res->end =0x%X\n", __FILE__,
         WARN_ON(rc);
     }
     WARN_ON(of_irq_to_resource_table(np, res,num_irq) != num_irq);
```

3. dev->dev.of_node = of_node_get(np);
    - // 这个node属性里有compatible属性，这个属性从dts来，后续driver匹配device时，就是通过这一属性进匹配
    - // 我们可以通过添加下面一句话来查看compatible.
    - // printk(“[%s %s %d]bus->name = %s, of_get_property(…) = %s\n”, __FILE__, __func__,__LINE__, np->name, (char*)of_get_property(np, “compatible”,NULL));
    - // node 再给dev，后续给驱动注册使用.
4. 运行 of_device_make_bus_id 设定device的名字，如: soc.2 或 ac000000.serial 等

3-2. drivers/of/platform.c :

以 compatible = “simple-bus”的节点的子节点都会以这个节点作为父节点在这步注册设备. 至此从dts文件的解析到最终调用of_device_add进行设备注册的过程就比较清晰了

查看挂载上的所有设备

cd /sys/devices/ 查看注册成功的设备 对应devicetree中的设备描述节点^-^