目录

什么是DTS？为什么要引入DTS？

DTS即Device Tree Source 设备树源码，Device Tree是一种描述硬件的数据结构，它起源于 OpenFirmware (OF)。 在Linux 2.6中，ARM架构的板极硬件细节过多地被硬编码在arch/arm/plat-xxx和arch/arm/mach-xxx，比如板上的platform设备、 resource、i2c_board_info、spi_board_info以及各种硬件的platform_data，这些板级细节代码对内核来讲只不过是垃圾代码。 而采用Device Tree后，许多硬件的细节可以直接透过它传递给Linux，而不再需要在kernel中进行大量的冗余编码。 每次正式的linux kernel release之后都会有两周的merge window，在这个窗口期间，kernel各个部分的维护者都会提交各自的patch， 将自己测试稳定的代码请求并入kernel main line。每到这个时候，Linus就会比较繁忙，他需要从各个内核维护者的分支上取得最新代码 并merge到自己的kernel source tree中。Tony Lindgren，内核OMAP development tree的维护者，发送了一个邮件给Linus， 请求提交OMAP平台代码修改，并给出了一些细节描述：

1. 简单介绍本次改动
2. 关于如何解决merge conficts。有些git mergetool就可以处理，不能处理的，给出了详细介绍和解决方案

一切都很平常，也给出了足够的信息，然而，正是这个pull request引发了一场针对ARM linux的内核代码的争论。 我相信Linus一定是对ARM相关的代码早就不爽了，ARM的merge工作量较大倒在其次，主要是他认为ARM很多的代码都是垃圾， 代码里面有若干愚蠢的table，而多个人在维护这个table，从而导致了冲突。因此，在处理完OMAP的pull request之后（Linus并非针对OMAP平台， 只是Tony Lindgren撞在枪口上了），他发出了怒吼：

```
Gaah.Guys, this whole ARM thing is a f*cking pain in the ass.
```

之后经过一些讨论，对ARM平台的相关code做出如下相关规范调整，这个也正是引入DTS的原因。

1. ARM的核心代码仍然保存在arch/arm目录下
2. ARM SoC core architecture code保存在arch/arm目录下
3. ARM SOC的周边外设模块的驱动保存在drivers目录下
4. ARM SOC的特定代码在arch/arm/mach-xxx目录下
5. ARM SOC board specific的代码被移除，由DeviceTree机制来负责传递硬件拓扑和硬件资源信息。

本质上，Device Tree改变了原来用hardcode方式将HW 配置信息嵌入到内核代码的方法，改用bootloader传递一个DB的形式。

如果我们认为kernel是一个black box，那么其输入参数应该包括：

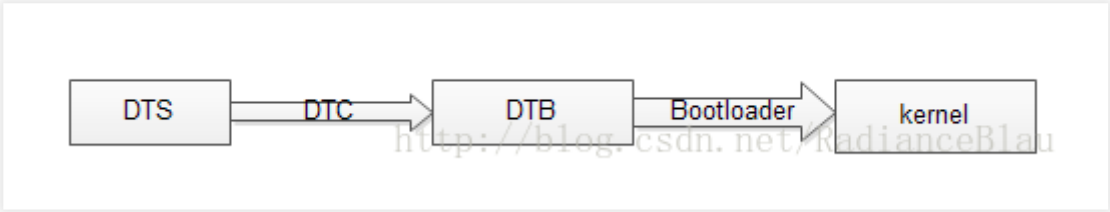
1. 识别platform的信息
2. runtime的配置参数
3. 设备的拓扑结构以及特性

对于嵌入式系统，在系统启动阶段，bootloader会加载内核并将控制权转交给内核，此外，还需要把上述的三个参数信息传递给kernel，以便kernel可以有较大的灵活性。在linux kernel中，Device Tree的设计目标就是如此。

二.DTS基本知识

DTS的加载过程

如果要使用Device Tree，首先用户要了解自己的硬件配置和系统运行参数，并把这些信息组织成Device Tree source file。 通过DTC（Device Tree Compiler），可以将这些适合人类阅读的Device Tree source file变成适合机器处理的 Device Tree binary file（有一个更好听的名字，DTB，device tree blob）。在系统启动的时候，boot program （例如：firmware、bootloader）可以将保存在flash中的DTB copy到内存（当然也可以通过其他方式， 例如可以通过bootloader的交互式命令加载DTB，或者firmware可以探测到device的信息，组织成DTB保存在内存中）， 并把DTB的起始地址传递给client program（例如OS kernel，bootloader或者其他特殊功能的程序）。 对于计算机系统（computer system），一般是firmware->bootloader->OS，对于嵌入式系统，一般是bootloader->OS。



DTS的描述信息

Device Tree由一系列被命名的结点（node）和属性（property）组成，而结点本身可包含子结点。所谓属性， 其实就是成对出现的name和value。在Device Tree中，可描述的信息包括（原先这些信息大多被hard code到kernel中）：

- CPU的数量和类别
- 内存基地址和大小
- 总线和桥
- 外设连接
- 中断控制器和中断使用情况
- GPIO控制器和GPIO使用情况
- Clock控制器和Clock使用情况

它基本上就是画一棵电路板上CPU、总线、设备组成的树，Bootloader会将这棵树传递给内核，然后内核可以识别这棵树， 并根据它展开出Linux内核中的platform_device、

i2c_client、spi_device等设备，而这些设备用到的内存、IRQ等资源， 也被传递给了内核，内核会将这些资源绑定给展开的相应的设备。

是否Device Tree要描述系统中的所有硬件信息？答案是否定的。基本上，那些可以动态探测到的设备是不需要描述的， 例如USB device。不过对于SOC上的usb hostcontroller，它是无法动态识别的，需要在device tree中描述。同样的道理， 在computersystem中，PCI device可以被动态探测到，不需要在device tree中描述，但是PCI bridge如果不能被探测，那么就需要描述之。

.dts文件是一种ASCII 文本格式的Device Tree描述，此文本格式非常人性化，适合人类的阅读习惯。 基本上，在ARM Linux在，一个.dts文件对应一个ARM的machine，一般放置在内核的arch/arm/boot/dts/目录。 由于一个SoC可能对应多个machine（一个SoC可以对应多个产品和电路板），势必这些.dts文件需包含许多共同的部分， Linux内核为了简化，把SoC公用的部分或者多个machine共同的部分一般提炼为.dtsi，类似于C语言的头文件。 其他的machine对应的.dts就include这个.dtsi。譬如，对于RK3288而言， rk3288.dtsi就被rk3288-chrome.dts所引用， rk3288-chrome.dts有如下一行：#include“rk3288.dtsi”，对于rtd1195,在 rtd-119x-nas.dts中就包含了/include/ ”rtd-119x.dtsi” 当然，和C语言的头文件类似，.dtsi也可以include其他的.dtsi，譬如几乎所有的ARM SoC的.dtsi都引用了skeleton.dtsi，即#include”skeleton.dtsi“ 或者 /include/ “skeleton.dtsi”

正常情况下所有的dts文件以及dtsi文件都含有一个根节点”/”,这样include之后就会造成有很多个根节点？按理说 device tree既然是一个树，那么其只能有一个根节点，所有其他的节点都是派生于根节点的child node. 其实Device Tree Compiler会对DTS的node进行合并，最终生成的DTB中只有一个 root node.

device tree的基本单元是node。这些node被组织成树状结构，除了root node，每个node都只有一个parent。 一个device tree文件中只能有一个root node。每个node中包含了若干的property/value来描述该node的一些特性。 每个node用节点名字（node name）标识，节点名字的格式是node-name@unit-address。如果该node没有reg属性（后面会描述这个property）， 那么该节点名字中必须不能包括@和unit-address。unit-address的具体格式是和设备挂在那个bus上相关。例如对于cpu， 其unit-address就是从0开始编址，以此加一。而具体的设备，例如以太网控制器，其unit-address就是寄存器地址。root node的node name是确定的，必须是“/”。 在一个树状结构的device tree中，如何引用一个node呢？要想唯一指定一个node必须使用full path，例如/node-name-1/node-name-2/node-name-N。

DTS的组成结构

```
/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];
        child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
        child-node2 {
        };
    };
    node2 {
```

```
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
        child-node1 {
        };
    };
};
```

上述.dts文件并没有什么真实的用途，但它基本表征了一个Device Tree源文件的结构：

1个root结点”/”；

root结点下面含一系列子结点，本例中为”node1”和 “node2”；

结点”node1”下又含有一系列子结点，本例中为”child-node1”和 “child-node2”；

各结点都有一系列属性。这些属性可能为空，如”an-empty-property”；可能为字符串，如”a-string-property”； 可能为字符串数组，如”a-string-list-property”；可能为Cells（由u32整数组成），如”second-child-property”， 可能为二进制数，如”a-byte-data-property”。

下面以一个最简单的machine为例来看如何写一个.dts文件。假设此machine的配置如下：

1个双核ARM Cortex-A9 32位处理器；

ARM的local bus上的内存映射区域分布了2个串口（分别位于0x101F1000 和0x101F2000）、 GPIO控制器（位于0x101F3000）、SPI控制器（位于0x10115000）、中断控制器（位于0x10140000）和一个external bus桥；

External bus桥上又连接了SMC SMC91111 Ethernet（位于0x10100000）、I2C控制器（位于0x10160000）、64MB NOR Flash（位于0x30000000）；

External bus桥上连接的I2C控制器所对应的I2C总线上又连接了Maxim DS1338实时钟（I2C地址为0x58）

其对应的.dts文件为：

```
/ {
    compatible = "acme,coyotes-revenge";
    #address-cells = <1>;
    #size-cells = <1>;
    interrupt-parent = <&intc>;

    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
            compatible = "arm,cortex-a9";
            reg = <0>;
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
            reg = <1>;
        };
    };

    serial@101f0000 {
        compatible = "arm,pl011";
        reg = <0x101f0000 0x1000 >;
```

```
        interrupts = < 1 0 >;
    };

    serial@101f2000 {
        compatible = "arm,pl011";
        reg = <0x101f2000 0x1000 >;
        interrupts = < 2 0 >;
    };



gpio@101f3000 {
        compatible = "arm,pl061";
        reg = <0x101f3000 0x1000
               0x101f4000 0x0010>;
        interrupts = < 3 0 >;
    };

    intc: interrupt-controller@10140000 {
        compatible = "arm,pl190";
        reg = <0x10140000 0x1000 >;
        interrupt-controller;
        #interrupt-cells = <2>;
    };

    spi@10115000 {
        compatible = "arm,pl022";
        reg = <0x10115000 0x1000 >;
        interrupts = < 4 0 >;
    };



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

        ethernet@0,0 {
            compatible = "smc,smc91c111";
            reg = <0 0 0x1000>;
            interrupts = < 5 2 >;
        };

        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>;
                interrupts = < 7 3 >;
            };
        };

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

```
        };
    };

};
```

上述.dts文件中,root结点"/"的compatible 属性compatible = "acme,coyotes-revenge"; 定义了系统的名称， 它的组织形式为：,。Linux内核透过root结点"/"的compatible 属性即可判断它启动的是什么machine。

在.dts文件的每个设备，都有一个compatible属性，compatible属性用户驱动和设备的绑定。compatible 属性是一个字符串的列表， 列表中的第一个字符串表征了结点代表的确切设备，形式为","，其后的字符串表征可兼容的其他设备。可以说前面的是特指， 后面的则涵盖更广的范围。

如在arch/arm/boot/dts/vexpress-v2m.dtsi中的Flash结点：

```
flash@0,00000000 {
     compatible = "arm,vexpress-flash", "cfi-flash";
     reg = <0 0x00000000 0x04000000>,
     <1 0x00000000 0x04000000>;
     bank-width = <4>;
 };
```

compatible属性的第2个字符串"cfi-flash"明显比第1个字符串"arm,vexpress-flash"涵盖的范围更广。 接下来root结点"/"的cpus子结点下面又包含2个cpu子结点，描述了此machine上的2个CPU，并且二者的compatible 属性为"arm,cortex-a9"。

注意

cpus和cpus的2个cpu子结点的命名，它们遵循的组织形式为：[@]，<>中的内容是必选项，[]中的则为可选项。name是一个ASCII字符串，用于描述结点对应的设备类型，如3com Ethernet适配器对应的结点name宜为ethernet， 而不是3com509。如果一个结点描述的设备有地址，则应该给出@unit-address。多个相同类型设备结点的name可以一样， 只要unit-address不同即可，如本例中含有cpu@0、cpu@1以及serial@101f0000与serial@101f2000这样的同名结点。 设备的unit-address地址也经常在其对应结点的reg属性中给出。 reg的组织形式为reg = <address1 length1 [address2 length2][address3 length3] ... >，其中的每一组addresslength表明了设备使用的一个地址范围。address为1个或多个32位的整型（即cell），而length则为cell的列表或者为空 （若#size-cells = 0）。address和length字段是可变长的，父结点的#address-cells和#size-cells分别决定了子结点的reg属性的address和length字段的长度。

在本例中，root结点的#address-cells = <1>;和#size-cells =<1>;决定了serial、gpio、spi等结点的address和length字段的长度分别为1。 cpus 结点的#address-cells= <1>;和#size-cells =<0>;决定了2个cpu子结点的address为1，而length为空， 于是形成了2个cpu的reg =<0>;和reg =<1>;。external-bus结点的#address-cells= <2>和#size-cells =<1>; 决定了其下的ethernet、i2c、flash的reg字段形如reg = <0 00x1000>;、reg = <1 00x1000>;和reg = <2 00x4000000>;。 其中，address字段长度为0，开始的第一个cell（0、1、2）是对应的片选，第2个cell（0，0，0）是相对该片选的基地址， 第3个cell（0x1000、0x1000、0x4000000）为length。特别要留意的是i2c结点中定义的#address-cells = <1>;和#size-cells =<0>; 又作用到了I2C总线上连接的RTC，它的address字段为0x58，是设备的I2C地址。

root结点的子结点描述的是CPU的视图，因此root子结点的address区域就直接位于CPU的memory区域。但是， 经过总线桥后的address往往需要经过转换才能对应的CPU的memory映射。external-bus的ranges属性定义了经过external-bus桥后的地址范围如何映射到CPU的memory区域。

```
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是地址转换表，其中的每个项目是一个子地址、父地址以及在子地址空间的大小的映射。映射表中的子地址、 父地址分别采用子地址空间的#address-cells和父地址空间的#address-cells大小。对于本例而言，子地址空间的#address-cells为2， 父地址空间的#address-cells值为1，因此0 0 0x10100000 0x10000的前2个cell为external-bus后片选0上偏移0， 第3个cell表示external-bus后片选0上偏移0的地址空间被映射到CPU的0x10100000位置，第4个cell表示映射的大小为0x10000。ranges的后面2个项目的含义可以类推。

Device Tree中还可以中断连接信息，对于中断控制器而言，它提供如下属性：

interrupt-controller – 这个属性为空，中断控制器应该加上此属性表明自己的身份；

#interrupt-cells – 与#address-cells 和 #size-cells相似，它表明连接此中断控制器的设备的interrupts属性的cell大小。在整个Device Tree中，与中断相关的属性还包括：

interrupt-parent - 设备结点透过它来指定它所依附的中断控制器的phandle，当结点没有指定interrupt-parent时，则从父级结点继承。对于本例而言，root结点指定了interrupt-parent= <&intc>;其对应于intc: interrupt-controller@10140000，而root结点的子结点并未指定interrupt-parent，因此它们都继承了intc，即位于0x10140000的中断控制器。

interrupts - 用到了中断的设备结点透过它指定中断号、触发方法等，具体这个属性含有多少个cell，由它依附的中断控制器结点的#interrupt-cells属性决定。而具体每个cell又是什么含义，一般由驱动的实现决定，而且也会在Device Tree的binding文档中说明

譬如，对于ARM GIC中断控制器而言，#interrupt-cells为3，它3个cell的具体含义

kernel/Documentation/devicetree/bindings/arm/gic.txt就有如下文字说明：

```
The 1st cell is the interrupt type; 0 for SPI interrupts, 1 for PPI
interrupts.

The 2nd cell contains the interrupt number for the interrupt type.
SPI interrupts are in the range [0-987].  PPI interrupts are in the
range [0-15].

The 3rd cell is the flags, encoded as follows:
  bits[3:0] trigger type and level flags.
      1 = low-to-high edge triggered
      2 = high-to-low edge triggered
      4 = active high level-sensitive
      8 = active low level-sensitive
  bits[15:8] PPI interrupt cpu mask.  Each bit corresponds to each of
  the 8 possible cpus attached to the GIC.  A bit set to '1' indicated
  the interrupt is wired to that CPU.  Only valid for PPI interrupts.
```

PPI(Private peripheral interrupt) SPI(Shared peripheral interrupt)

一个设备还可能用到多个中断号。对于ARM GIC而言，若某设备使用了SPI的168、169号2个中断，而言都是高电平触发， 则该设备结点的interrupts属性可定义为：interrupts =<0 168 4>, <0 169 4>;

dts引起BSP和driver的变更

没有使用dts之前的BSP和driver

```
static void usbcp_key_release(struct device * dev)
{
    return ;

}

static struct platform_device usbcp_key_dev = {
    .name   = "usbcopy_key",
    .id       = -1,
    .dev     = {
            .release = usbcp_key_release,
        },
};
```

```
static struct platform_driver usbcp_key_driver = {
    .driver = {
        .name   = "usbcopy_key",
        .owner  = THIS_MODULE,
    },
    .probe  = usbcp_key_probe,
    .remove = usbcp_key_remove,
};
```

使用dts之后的driver

```
static const struct of_device_id  usbcp_key_table[] = {
    {.compatible = "Realtek,rtk-gpio-ctl-irq-mux"},
    {},
};


static struct platform_driver usbcp_key_driver = {
    .driver = {
        .name   = "usbcopy_key",
        .of_match_table = usbcp_key_table,
        .owner  = THIS_MODULE,
    },
    .probe  = usbcp_key_probe,
    .remove = usbcp_key_remove,
};
```

```
rtk_gpio_ctl_mlk{
    compatible = "Realtek,rtk-gpio-ctl-irq-mux";
    gpios = <&rtk_iso_gpio 8 0 1>;
};
```


针对上面的dts，注意一下几点：

1. rtk_gpio_ctl_mlk这个是node的名字，自己可以随便定义，当然最好是见名知意，可以通过驱动程序打印当前使用的设备树节点 printk(“now dts node name is %s\n”,pdev->dev.of_node->name);
2. compatible选项是用来和驱动程序中的of_match_table指针所指向的of_device_id结构里的compatible字段匹配的，只有dts里的compatible字段的名字和驱动程序中of_device_id里的compatible字段的名字一样，驱动程序才能进入probe函数
3. 对于gpios这个字段，首先&rtk_iso_gpio指明了这个gpio是连接到的是rtk_iso_gpio，接着那个8是gpio number偏移量，它是以rtk_iso_gpiobase为基准的,紧接着那个0说明

目前配置的gpio number 是设置成输入input,如果是1就是设置成输出output.最后一个字段1是指定这个gpio 默认为高电平，如果是0则是指定这个gpio默认为低电平

4. 如果驱动里面只是利用compatible字段进行匹配进入probe函数，那么gpios 可以不需要，但是如果驱动程序里面是采用设备树相关的方法进行操作获取gpio number,那么gpios这个字段必须使用。 gpios这个字段是由of_get_gpio_flags函数默认指定的name.

获取gpio number的函数如下：

of_get_named_gpio_flags()

of_get_gpio_flags()

注册i2c_board_info，指定IRQ等板级信息。

```
static struct i2c_board_info __initdata afeb9260_i2c_devices[] = {
      {
                I2C_BOARD_INFO("tlv320aic23", 0x1a),
      }, {
                I2C_BOARD_INFO("fm3130", 0x68),
     }, {
                I2C_BOARD_INFO("24c64", 0x50),
     }
 };
```

之类的i2c_board_info代码，目前不再需要出现，现在只需要把tlv320aic23、fm3130、24c64这些设备结点填充作为相应的I2C controller结点的子结点即可，类似于前面的

```
i2c@1,0 {
       compatible = "acme,a1234-i2c-bus";
       …
       rtc@58 {
           compatible = "maxim,ds1338";
           reg = <58>;
           interrupts = < 7 3 >;
       };
   };
```

Device Tree中的I2C client会透过I2C host驱动的probe()函数中调用of_i2c_register_devices(&i2c_dev->adapter);被自动展开。

常见的DTS 函数

Linux内核中目前DTS相关的函数都是以of_前缀开头的，它们的实现位于内核源码的drivers/of下面

```
void __iomem*of_iomap(struct device_node *node, int index)
```

通过设备结点直接进行设备内存区间的 ioremap()，index是内存段的索引。若设备结点的reg属性有多段，可通过index标示要ioremap的是哪一段， 只有1段的情况，index为0。采用Device Tree后，大量的设备驱动通过of_iomap()进行映射，而不再通过传统的ioremap。

```
int of_get_named_gpio_flags(struct device_node *np,const char *propname,
              int index, enum of_gpio_flags *flags)
```

```
static inline int of_get_gpio_flags(structdevice_node *np, int index,
               enum of_gpio_flags *flags)
{
               return of_get_named_gpio_flags(np, "gpios", index,flags);
}
```

从设备树中读取相关GPIO的配置编号和标志,返回值为 gpio number

DTC (device tree compiler)

将.dts编译为.dtb的工具。DTC的源代码位于内核的scripts/dtc目录，在Linux内核使能了Device Tree的情况下， 编译内核的时候主机工具dtc会被编译出来，对应scripts/dtc/Makefile中的“hostprogs-y := dtc”这一hostprogs编译target。 在Linux内核的arch/arm/boot/dts/Makefile中，描述了当某种SoC被选中后，哪些.dtb文件会被编译出来，如与VEXPRESS对应的.dtb包括：

```
dtb-$(CONFIG_ARCH_VEXPRESS) += vexpress-v2p-ca5s.dtb \
         vexpress-v2p-ca9.dtb \
         vexpress-v2p-ca15-tc1.dtb \
         vexpress-v2p-ca15_a7.dtb \
         xenvm-4.2.dtb
```

在Linux下，我们可以单独编译Device Tree文件。当我们在Linux内核下运行make dtbs时，若我们之前选择了ARCH_VEXPRESS， 上述.dtb都会由对应的.dts编译出来。因为arch/arm/Makefile中含有一个dtbs编译target项目