华清远见 FARSIGHT 嵌入式培训专家

嵌入式学院 华清远见旗下品牌

设备树

大纲

- 概述
- 基本语法
- 节点
- 属性

概述

- 设备树（Device Tree）是一种描述硬件的数据结构，在操作系统（OS）引导阶段进行设备初始化的时候，数据结构中的硬件信息被检测并传递给操作系统。最早诞生于Open Firmware，Flattened Device Tree (FDT)格式标准。
- dts文件（Device Tree Source，dts）是以ASCII文本形式描述设备树内容。
- dtb文件是二进制格式，编译工具为：Device Tree Compiler（DTC）。
- 2011年被引入ARM Linux内核。ARM Linux 设备树描述了内核的软/硬件信息。
- 节点（node）和属性（property）
  - 节点用以归类描述一个硬件信息或是软件信息（好比文件系统的目录）
  - 节点内描述了一个或多个属性，属性是键值对（key/value），描述具体的软/硬信息。

概述

- 什么ARM Linux社区会引入设备树呢？
  - 主要是想解决ARM Linux内核代码冗余的问题。
- 学习参考
  - http://www.devicetree.org/Device_Tree_Usage
  - 内核源码目录Documentation\devicetree设备树范例的说明文档
  - 内核源码drivers/of目录下是设备树操作实现源码
  - 内核源码include/linux目录下的of_xxx.h是设备树的头文件

基本语法

```
/{
    node{
        key=value;
        ...
        child_node{
            key=value;
            ...
        };
        ...
    };
    ...
};
```

根节点

根节点的子节点

node节点的子节点

1、属性：键值对
key属性键（以下描述为属性名）
value是该属性的值，值是字节流
键值对以";"结束
一个属性可以没有值

2、节点
节点描述范围"{...};"
child_node是node的子节点
node是child_node的父节点

基本语法

- DTS描述键值对的语法：
  - 1、字符串信息
  - 2、32bits无符号整型数组信息
  - 3、二进制数数组
  - 4、混和形式
  - 5、字符串哈希表
    - Text strings (null terminated) are represented with double quotes:
      - `string-property = "a string"`
    - 'Cells' are 32 bit unsigned integers delimited by angle brackets:
      - `cell-property = <0xbeef 123 0xabcd1234>`
    - binary data is delimited with square brackets:
      - `binary-property = [0x01 0x23 0x45 0x67];`
    - Data of differing representations can be concatenated together using a comma:
      - `mixed-property = "a string", [0x01 0x23 0x45 0x67], <0x12345678>;`
    - Commas are also used to create lists of strings:
      - `string-list = "red fish", "blue fish";`

节点

- 节点名
  - 语法： <name>[@<unit-address>]
- 规范：
  - 名字是ASCII字符串
  - （字母、数字、"-"、等等构成）
  - 最长可以是31个字符
  - 一般的，应该以设备类型命名
  - unit-address一般的是设备地址

```
/ {
    serial@101F0000 {
        ......
    };
    gpio@101F3000 {
        ......
    };
    interrupt-controller@10140000 {
        ......
    };
    spi@10115000 {
        ......
    };
    external-bus {
        ethernet@0,0 {
            ......
        };
        i2c@1,0 {
            ......
            rtc@58 {
                ......
            };
        };
    };
};
```

节点

- 节点名及节点路径
- 范例：

```
/{
        ...
        dm9000{
            ...
        };
        ...
};
```

节点名：dm9000

节点路径：/dm9000

```
/{
        ...
        ethernet@80000000{
            ...
        };
        ...
};
```

节点名：ethernet

节点路径：/ethernet@80000000

节点

- 节点别名（节点引用）
  - 为了解决节点路径名过长的问题，引入了节点别名的概念，可以引用到一个全路径的节点。

```
/{
    aliases {
        demo = &demo0;
    };
    ...
    demo:demo0@80000000{
        ...
    };
    ...
};
```

节点名：demo0
节点路径：/demo0@80000000
节点别名：demo（等价/demo0@80000000）

引用语法范例1：

```
&demo{
    ...
};
```

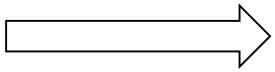
引用语法范例2：

```
/{
    reference-node{
        property=<&demo>;
        ...
    };
    ...
};
```

节点

- 合并节点内容
  - 一般的，一个硬件设备的部分信息不会变化，但是部分信息是可能会变化的，就出现了节点内容合并。即：先编写好节点，仅仅描述部分属性值；使用者后加一部分属性值。在同级路径下，节点名相同的“两个”节点实际是一个节点。

```
/*参考板的已经编写好的 node 节点*/
/{
    node{
        property1=value;
    };
};
/*移植者添加的 node 节点*/
/{
    node{
        property2=value;
    };
};
```

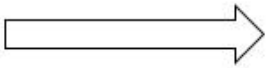


```
/{
    node{
        property1=value;
        property2=value;
    };
};
```

节点

- 替换节点内容
  - 一般的，一个硬件设备的部分属性信息可能会变化，但是设备树里面已经描述了所有的属性值，使用者可以添加已有的属性值，以替换原有的属性值，就出现了节点内容替换。在同级路径下，节点名相同的“两个”节点实际是一个节点。

```
/*参考板的已经编写好的 node 节点*/
/{
    node{
        property=value;
        status = "disabled";
    };
};
/*移植者添加的 node 节点*/
/{
    node{
        status = "okay";
    };
};
```

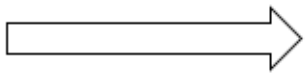


```
/{
    node{
        property=value;
        status = "okay";
    };
};
```

节点

- 引用节点内容
  - 一般的，一个设备可能会使用到别的节点的内容，可以通过节点的别名来引用到其内容。引用的目的可能是合并两个节点的内容、替换部分内容、或是使用部分内容。

```
/*参考板的已经编写好的 node 节点*/
/{
        node:  node@80000000{
                property=value;
                status = "disabled";
        };
};
/*移植者添加的 node 节点*/
&node{
        property=value;
        status = "okay";
};
```



```
/{
        node : node@80000000{
                property=value;
                status = "okay";
        };
};
```

节点

- 引用节点内容

```
/*参考板的已经编写好的 node 节点*/
/{
    node：node@80000000{
        property=value;
    };
};
/*移植者添加的 demo 节点*/
/{
    demo{
        property=<&node>;
    };
};
```

说明：

demo 节点的属性 property 引用了节点的 node 的属性值，一般的，引用的目的是使用 node 节点的部分属性内容。

节点-chosen节点

- chosen节点
  - chosen节点不描述一个真实设备，而是用于firmware传递一些数据给OS，譬如bootloader传递内核启动参数给内核
  - 范例：

```
chosen {
    bootargs = "root=/dev/nfs rw nfsroot=192.168.1.1 console=ttyS0,115200";
};
```

节点

- 查找节点
  - 一般的，涉及设备、总线、驱动的概念，即所谓设备信息和驱动代码分离的驱动框架，如platform、i2c、usb、spi、pci、等等；或是分层驱动框架（MTD设备驱动、framebuffer设备驱动、input设备驱动、...），则设备树中设备节点的会内核初始化时候被查找到，驱动代码将不关心节点的查找。
  - 如果仅仅是接口驱动框架（字符设备驱动、块设备驱动、网络设备驱动），则需要使用内核节点查找函数查找设备树中的设备节点。
- 查找办法
  - 通过节点的compatible属性值查找指定节点
  - 通过节点名查找指定节点
  - 通过节点路径查找指定节点

节点

- 节点描述
  - 头文件：include/of.h
  - struct device_node {
    - const char *name;　　//节点名
    - const char *type;　　//设备类型
    - const char *full_name;　//全路径节点名
    - struct device_node *parent;　　//父节点指针
    - struct device_node *child;　　//子节点指针
    - ...
  - }

节点

```
/*
* 功能：通过compatible属性查找指定节点
* 参数：
*     struct device_node *from - 指向开始路径的节点，如果为NULL，则从根节点开始
*     const char *type - device_type设备类型，可以为NULL
*     const char *compat - 指向节点的compatible属性的值（字符串）的首地址
* 返回值：
*     成功：得到节点的首地址；失败：NULL
*/
struct device_node *of_find_compatible_node(struct device_node *from,
                                            const char *type, const char *compat);
```

节点

设备ID表结构，用于匹配设备节点和驱动

```
struct of_device_id {
        char name[32];          /*设备名*/
        char type[32];          /*设备类型*/
        char compatible[128];   /*用于与设备树compatible属性值匹配的字符串*/
        const void *data;       /*驱动私有数据*/
};
```

//注册支持设备树的设备ID表

include/module.h

MODULE_DEVICE_TABLE(of, ID表首地址)

节点

```
/*
* 功能：通过compatible属性查找指定节点
* 参数：
*      struct device_node *from - 指向开始路径的节点，如果为NULL，则从根节点开始
*      const struct of_device_id *matches - 指向设备ID表，注意ID表必须以NULL结束
*      范例：      const struct of_device_id mydemo_of_match[] = {
                            { .compatible = "fs4412,mydemo", },
                            {}
                    };
* 返回值：
*      成功：得到节点的首地址；失败：NULL
*/
struct device_node *of_find_matching_node(struct device_node *from,
                                          const struct of_device_id *matches);
```

节点

```
/*
* 功能：通过路径查找指定节点
* 参数：
*     const char *path - 带全路径的节点名，也可以是节点的别名
* 返回值：
*     成功：得到节点的首地址；失败：NULL
*/
struct device_node *of_find_node_by_path(const char *path);
```

节点

```
/*
* 功能：通过节点名查找指定节点
* 参数：
*     struct device_node *from - 开始查找节点，如果为NULL，则从根节点开始
*     const char *name- 节点名
* 返回值：
*     成功：得到节点的首地址；失败：NULL
*/
struct device_node *of_find_node_by_name(struct device_node *from,
                                         const char *name);
```

属性

- 有默认意义的属性
  - 1、设备树语法中已经定义好的，具有通用规范意义的属性。
    - 一般的，如果是设备信息和驱动分离框架的设备节点，则能够在内核初始化找到节点时候，自动解析生成相应的设备信息。
    - 常见属性的有：compatible、地址address、中断interrupt
  - ARM Linux内核定义好的，一类设备通用的有默认意义的属性
    - 一般的，不能被内核自动解析生成相应的设备信息，但是内核已经编写了相应的解析提取函数。
    - 常见属性的有：MAC地址、GPIO口、clock、power、regulator、等等

属性

- 驱动自定义属性
  - 针对具体设备，有部分属性很难通用，需要驱动自己定义好，通过内核的属性提取解析函数进行值的获得。

```
ethernet@18000000 {
        compatible = "davicom,dm9000";
        reg = <0x18000000 0x2 0x18000004 0x2>;
        interrupt-parent = <&gpn>;
        interrupts = <7 4>;
        local-mac-address = [00 00 de ad be ef];
        davicom,no-eeprom;
        reset-gpios = <&gpf 12 GPIO_ACTIVE_LOW>;
        vcc-supply = <&eth0_power>;
};
```

属性

- compatible属性
  - 一般的，用于匹配设备节点和设备驱动，规则是驱动设备ID表中的compatible域的值（字符串），和设备树中设备节点中的compatible属性值完全一致，则节点的内容是给驱动的。
  - 设备树中的命名规范如下：
  - /{
  - node{
  - compatible=“厂商名,名称”;
  - ...
  - };
  - ...
  - };

属性-compatible属性

```
/*platform 框架的探测函数*/
static int demo_probe(struct platform_device *devices)
{
    //设备树对应节点的信息已经被内核构造成 struct platform_devic
    ...
}
static const struct of_device_id demo_of_matches[] = {
    { .compatible = "fs4412,mydemo", },
    { }
};
MODULE_DEVICE_TABLE(of, demo_of_matches);
static struct platform_driver demo_drv = {
    .driver = {
        .name = DEMONAME,
        .owner= THIS_MODULE,
        .of_match_table = of_match_ptr(demo_of
    },
```

```
/{
    ......
    mydemo{
        compatible = "fs4412,mydemo
        ......
    };
    ......
```

属性-address

- #address-cells：描述子节点reg属性值的地址表中首地址cell数量
- #size-cells：描述子节点reg属性值的地址表中地址长度cell数量
- reg：描述地址表

```
/{
    parent-node{
        #address-cells = <1>;
        #size-cells = <1>;
        ...
        son-node{
            reg = <addr1 len1 [addr2 len2]   [...]>;
            ...
        };
    };
};
```

说明：

父节点#address-cells 值为 1，#size-cells 值为 1，则子节点中 reg 的值就是一个首地址紧接着一个地址长度为一个单元。

属性-address

- CPU地址描述
  - 每个CPU都分配了唯一的一个ID，描述没有大小的CPU ids

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

属性-address

- 内存映射设备（Memory Mapped Devices）
  - 描述一个设备的内存地址的时候，一般使用1个cell（32bits）描述地址，紧接着1一个cell（32bits）描述地址长度。

```
/ {
    #address-cells = <1>;
    #size-cells = <1>;

    ...

    serial@101f0000 {
        compatible = "arm,pl011";
        reg = <0x101f0000 0x1000 >;
    };
    gpio@101f3000 {
        compatible = "arm,pl061";
        reg = <0x101f3000 0x1000
               0x101f4000 0x0010>;
    };

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

    ...

};
```

属性-address

- 非内存映射设备（Non Memory Mapped Devices）
  - 譬如i2c设备，有一个寻址地址，没有内存地址那样的地址长度和范围，一般使用1个cell（32bits）描述该地址，而没有描述地址长度的cell。

```
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>;
    };
};
```

属性-address

- 地址转换范围Ranges（Address Translation）
  - 有些设备是有片选的，就需要描述片选及片选的偏移量，在说明地址时，还需要说明地址映射范围。

```
/ {
    compatible = "acme,coyotes-revenge";
    #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

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

  - 说明：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。

属性-interrupt

- interrupt-controller 一个空属性用来声明这个node接收中断信号；
- #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；
- 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；
- interrupts 一个中断标识符列表，表示每一个中断输出信号。

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

    serial@101f0000 {
        compatible = "arm,pl011";
        reg = <0x101f0000 0x1000 >;
        interrupts = < 1 0 >;
    };

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

属性-interrupt

- 一般的，如果父节点的#interrupt-cells的值是3，则子节点的interrupts一个cell三个32bits整型值：<中断域 中断 触发方式>
- 实际解析情况，得根据实际使用内核的设备树参加资料来决定。

```
/ {
        gic: interrupt-controller@10490000 {
                compatible = "arm,cortex-a9-gic";
                #interrupt-cells = <3>;
                interrupt-controller;
                cpu-offset = <0x4000>;
                reg = <0x10490000 0x10000>, <0x10480000 0x10000>;
        };
        pinctrl@11000000 {
                gpx0: gpx0 {
                        gpio-controller;
                        #gpio-cells = <2>;
                        interrupt-controller;
                        interrupt-parent = <&gic>;
                        interrupts = <0 16 0>, <0 17 0>, <0 18 0>, <0 19 0>,
                                     <0 20 0>, <0 21 0>, <0 22 0>, <0 23 0>;
                        #interrupt-cells = <2>;
                };
                ...
        };
        ...
};
```

属性-interrupt

- 一般的，如果父节点的#interrupt-cells的值是2，则子节点的interrupts一个cell两个32bits整型值：中断和触发方式。
- 实际解析情况，得根据实际使用内核的设备树参加资料来决定。

```
/ {
        pinctrl@11000000 {
                gpx0: gpx0 {
                        gpio-controller;
                        #gpio-cells = <2>;
                        interrupt-controller;
                        interrupt-parent = <&gic>;
                        interrupts = <0 16 0>, <0 17 0>, <0 18 0>, <0 19 0>,
                                     <0 20 0>, <0 21 0>, <0 22 0>, <0 23 0>;
                        #interrupt-cells = <2>;
                };
                ...
        };
        ethernet@5000000 {
                compatible = "davicom,dm9000";
                reg = <0x5000000 0x2 0x5000004 0x2>;
                interrupt-parent = <&gpx0>;
                interrupts = <6 4>;
                davicom,no-eeprom;
                mac-address = [00 0a 2d a6 55 a2];
        };
        ...
};
```

属性-gpio

- 常用的属性如下：
- gpio-controller：说明该节点描述的是一个gpio控制器
- #gpio-cells：描述gpio使用节点的属性一个cell的内容
  - 属性名=<&引用GPIO节点别名 GPIO标号 工作模式>;

```
/{
    gpx1: gpx1 {
        gpio-controller;
        #gpio-cells = <2>;
    };
    key@11400C24 {
        compatible = "fs4412,key";
        reg = <0x11400C24 0x4>;
        intn-key = <&gpx1 2 2>;
    };
};
```

属性

- 属性描述
- 头文件：include/of.h
- struct property {
- 	char	*name;	//属性的名称
- 	int length;	//属性值的字节数
- 	void	*value;	//属性值的首地址
- 	.....
- };

属性

```
/*
 * 功能：提取指定属性的值
 * 参数：
     const struct device_node *np - 设备节点指针
     const char *name - 属性名称
     int *lenp - 属性值的字节数
 * 返回值：
     成功：属性值的首地址；失败：NULL
 */
struct property *of_find_property(const struct device_node *np,
                                  const char *name, int *lenp);
```

属性

```
/*
 * 功能：得到属性值中数据的数量
 * 参数：
     const struct device_node *np - 设备节点指针
     const char *propname  - 属性名称
     int elem_size - 每个数据的单位（字节数）
 * 返回值：
     成功：属性值的数据个数；失败：负数，绝对值是错误码
 */
int of_property_count_elems_of_size(const struct device_node *np,
                const char *propname, int elem_size);
```

属性

```
/*
 * 功能：得到属性值中指定标号的32位数据值
 * 参数：
     const struct device_node *np - 设备节点指针
     const char *propname  - 属性名称
     u32 index  - 属性值中指定数据的标号
     u32 *out_value - 输出参数，得到指定数据的值
 * 返回值：
     成功：0；失败：负数，绝对值是错误码
 */
int of_property_read_u32_index(const struct device_node *np,
                               const char *propname, u32 index, u32 *out_value);
```

属性

```
/*
 * 功能：提取字符串（属性值）
 * 参数：
     const struct device_node *np - 设备节点指针
     const char *propname  - 属性名称
     const char **out_string - 输出参数，指向字符串（属性值）
 * 返回值：
     成功：0；失败：负数，绝对值是错误码
 */
int of_property_read_string(struct device_node *np,
                            const char *propname, const char **out_string);
```

属性

```
/*
 * 功能：提取默认属性“#address-cells”的值
 * 参数：
        const struct device_node *np - 设备节点指针
 * 返回值：
        成功：地址的数量；失败：负数，绝对值是错误码
 */
int of_n_addr_cells(struct device_node *np);
/*
 * 功能：提取默认属性“#size-cells”的值
 * 参数：
        const struct device_node *np - 设备节点指针
 * 返回值：
        成功：地址长度的数量；失败：负数，绝对值是错误码
 */
int of_n_size_cells(struct device_node *np);
```

属性

* 功能：提取I/O口地址
* 参数：
    const struct device_node *np - 设备节点指针
    int index - 地址的标号
    u64 *size - 输出参数，I/O口地址的长度
    unsigned int *flags - 输出参数，类型（IORESOURCE_IO、IORESOURCE_MEM）
* 返回值：
    成功：I/O口地址的首地址；失败：NULL

```
__be32 *of_get_address(struct device_node *dev, int index, u64 *size, unsigned int *flags);
```

* 功能：从设备树中提取I/O口地址转换成物理地址
* 参数：
    const struct device_node *np - 设备节点指针
    const __be32 *in_addr - 设备树提取的I/O地址
* 返回值：
    成功：物理地址；失败：OF_BAD_ADDR

```
u64 of_translate_address(struct device_node *dev, const __be32 *in_addr);
```

属性

* 功能：提取I/O口地址并映射成虚拟地址
* 参数：
    const struct device_node *np - 设备节点指针
    int index - I/O地址的标号
* 返回值：
    成功：映射好虚拟地址；失败：NULL

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

* 功能：提取I/O口地址并申请I/O资源及映射成虚拟地址
* 参数：
    const struct device_node *np - 设备节点指针
    int index - I/O地址的标号
    const char *name - 设备名，申请I/O地址时使用
* 返回值：
    成功：映射好虚拟地址；失败：NULL

```
void __iomem *of_io_request_and_map(struct device_node *np, int index, const char *name);
```

属性

```
/*
 * 功能：从设备树中提取资源resource（I/O地址）
 * 参数：
        const struct device_node *np - 设备节点指针
        int index - I/O地址资源的标号
        struct resource *r - 输出参数，指向资源resource（I/O地址）
 * 返回值：
        成功：0；失败：负数，绝对值是错误码
 */
int of_address_to_resource(struct device_node *dev, int index, struct resource *r);
```

属性

```
/* include/of_gpio.h
 * 功能：从设备树中提取gpio口
 * 参数：
        const struct device_node *np - 设备节点指针
        const char *propname - 属性名
        int index – 引用gpio位置标号
 * 返回值：
        成功：得到GPIO口功能号；失败：负数，绝对值是错误码
 */
int of_get_named_gpio(struct device_node *np, const char *propname, int index);
```

属性

* 功能：从设备树中提取中断的数量
* 参数：
    const struct device_node *np - 设备节点指针
* 返回值：
    成功：大于等于0，实际中断数量，0则表示没有中断

int of_irq_count(struct device_node *dev);

* 功能：从设备树中提取中断号
* 参数：
    const struct device_node *np - 设备节点指针
    int index - 要提取的中断号的标号
* 返回值：
    成功：中断号；失败：负数，其绝对值是错误码

int of_irq_get(struct device_node *dev, int index);

属性

- /*
  * 从设备树中提取中断并映射好
- * 参数：
- struct device_node *dev - 设备树节点
- int index - 中断编号
- * 返回值：
- 成功：中断号（软）
- 失败负数，绝对值是错误码
- */
- unsigned int irq_of_parse_and_map(struct device_node *dev, int index)

属性

```
/*
 * 功能：从设备树中提取MAC地址
 * 参数：
        const struct device_node *np - 设备节点指针
 * 返回值：
        成功：MAC（6字节）的首地址；失败：NULL
 */
void *of_get_mac_address(struct device_node *np);
```