引自 100ask

### 内核里操作设备树的常用函数

内核源码中include/linux/目录下有很多of开头的头文件，of表示“open firmware”即开放固件。

#### 11.7.1 内核中设备树相关的头文件介绍

设备树的处理过程是：dtb -> device\_node -> platform\_device。

##### 11.7.1.1 处理DTB

of\_fdt.h // dtb文件的相关操作函数, 我们一般用不到,

// 因为dtb文件在内核中已经被转换为device\_node树(它更易于使用)

##### 11.7.1.2 处理device\_node

of.h // 提供设备树的一般处理函数,

// 比如 of\_property\_read\_u32(读取某个属性的u32值),

// of\_get\_child\_count(获取某个device\_node的子节点数)

of\_address.h // 地址相关的函数,

// 比如 of\_get\_address(获得reg属性中的addr, size值)

// of\_match\_device (从matches数组中取出与当前设备最匹配的一项)

of\_dma.h // 设备树中DMA相关属性的函数

of\_gpio.h // GPIO相关的函数

of\_graph.h // GPU相关驱动中用到的函数, 从设备树中获得GPU信息

of\_iommu.h // 很少用到

of\_irq.h // 中断相关的函数

of\_mdio.h // MDIO (Ethernet PHY) API

of\_net.h // OF helpers for network devices.

of\_pci.h // PCI相关函数

of\_pdt.h // 很少用到

of\_reserved\_mem.h // reserved\_mem的相关函数

##### 11.7.1.3 处理 platform\_device

of\_platform.h // 把device\_node转换为platform\_device时用到的函数,

// 比如of\_device\_alloc(根据device\_node分配设置platform\_device),

// of\_find\_device\_by\_node (根据device\_node查找到platform\_device),

// of\_platform\_bus\_probe (处理device\_node及它的子节点)

of\_device.h // 设备相关的函数, 比如 of\_match\_device

#### 11.7.2 platform\_device相关的函数

of\_platform.h中声明了很多函数，但是作为驱动开发者，我们只使用其中的1、2个。其他的都是给内核自己使用的，内核使用它们来处理设备树，转换得到platform\_device。

##### 11.7.2.1 of\_find\_device\_by\_node

函数原型为：

extern struct platform\_device \*of\_find\_device\_by\_node(struct device\_node \*np);

设备树中的每一个节点，在内核里都有一个device\_node；你可以使用device\_node去找到对应的platform\_device。

##### 11.7.2.2 platform\_get\_resource

这个函数跟设备树没什么关系，但是设备树中的节点被转换为platform\_device后，设备树中的reg属性、interrupts属性也会被转换为“resource”。

这时，你可以使用这个函数取出这些资源。

函数原型为：

/\*\*

\* platform\_get\_resource - get a resource for a device

\* @dev: platform device

\* @type: resource type // 取哪类资源？IORESOURCE\_MEM、IORESOURCE\_REG

\* // IORESOURCE\_IRQ等

\* @num: resource index // 这类资源中的哪一个？

\*/

struct resource \*platform\_get\_resource(struct platform\_device \*dev,

unsigned int type, unsigned int num);

对于设备树节点中的reg属性，它对应IORESOURCE\_MEM类型的资源；

对于设备树节点中的interrupts属性，它对应IORESOURCE\_IRQ类型的资源。

#### 11.7.3 有些节点不会生成platform\_device，怎么访问它们

内核会把dtb文件解析出一系列的device\_node结构体，我们可以直接访问这些device\_node。

内核源码incldue/linux/of.h中声明了device\_node和属性property的操作函数，device\_node和property的结构体定义如下：



##### 11.7.3.1 找到节点

**a. of\_find\_node\_by\_path**

根据路径找到节点，比如“/”就对应根节点，“/memory”对应memory节点。

函数原型：

static inline struct device\_node \*of\_find\_node\_by\_path(const char \*path);

**b. of\_find\_node\_by\_name**

根据名字找到节点，节点如果定义了name属性，那我们可以根据名字找到它。

函数原型：

extern struct device\_node \*of\_find\_node\_by\_name(struct device\_node \*from,

const char \*name);

参数from表示从哪一个节点开始寻找，传入NULL表示从根节点开始寻找。

但是在设备树的官方规范中不建议使用“name”属性，所以这函数也不建议使用。

**c. of\_find\_node\_by\_type**

根据类型找到节点，节点如果定义了device\_type属性，那我们可以根据类型找到它。

函数原型：

extern struct device\_node \*of\_find\_node\_by\_type(struct device\_node \*from,

const char \*type);

参数from表示从哪一个节点开始寻找，传入NULL表示从根节点开始寻找。

但是在设备树的官方规范中不建议使用“device\_type”属性，所以这函数也不建议使用。

**d. of\_find\_compatible\_node**

根据compatible找到节点，节点如果定义了compatible属性，那我们可以根据compatible属性找到它。

函数原型：

extern struct device\_node \*of\_find\_compatible\_node(struct device\_node \*from,

const char \*type, const char \*compat);

参数from表示从哪一个节点开始寻找，传入NULL表示从根节点开始寻找。

参数compat是一个字符串，用来指定compatible属性的值；

参数type是一个字符串，用来指定device\_type属性的值，可以传入NULL。

**e. of\_find\_node\_by\_phandle**

根据phandle找到节点。

dts文件被编译为dtb文件时，每一个节点都有一个数字ID，这些数字ID彼此不同。可以使用数字ID来找到device\_node。这些数字ID就是phandle。

函数原型：

extern struct device\_node \*of\_find\_node\_by\_phandle(phandle handle);

参数from表示从哪一个节点开始寻找，传入NULL表示从根节点开始寻找。

**f. of\_get\_parent**

找到device\_node的父节点。

函数原型：

extern struct device\_node \*of\_get\_parent(const struct device\_node \*node);

参数from表示从哪一个节点开始寻找，传入NULL表示从根节点开始寻找。

**g. of\_get\_next\_parent**

这个函数名比较奇怪，怎么可能有“next parent”？

它实际上也是找到device\_node的父节点，跟of\_get\_parent的返回结果是一样的。

差别在于它多调用下列函数，把node节点的引用计数减少了1。这意味着调用of\_get\_next\_parent之后，你不再需要调用of\_node\_put释放node节点。

of\_node\_put(node);

函数原型：

extern struct device\_node \*of\_get\_next\_parent(struct device\_node \*node);

参数from表示从哪一个节点开始寻找，传入NULL表示从根节点开始寻找。

**h. of\_get\_next\_child**

取出下一个子节点。

函数原型：

extern struct device\_node \*of\_get\_next\_child(const struct device\_node \*node,

struct device\_node \*prev);

参数node表示父节点；

prev表示上一个子节点，设为NULL时表示想找到第1个子节点。

不断调用of\_get\_next\_child时，不断更新pre参数，就可以得到所有的子节点。

**i. of\_get\_next\_available\_child**

取出下一个“可用”的子节点，有些节点的status是“disabled”，那就会跳过这些节点。

函数原型：

struct device\_node \*of\_get\_next\_available\_child(const struct device\_node \*node,

struct device\_node \*prev);

参数node表示父节点；

prev表示上一个子节点，设为NULL时表示想找到第1个子节点。

**j. of\_get\_child\_by\_name**

根据名字取出子节点。

函数原型：

extern struct device\_node \*of\_get\_child\_by\_name(const struct device\_node \*node,

const char \*name);

参数node表示父节点；

name表示子节点的名字。

##### 11.7.3.2 找到属性

内核源码incldue/linux/of.h中声明了device\_node的操作函数，当然也包括属性的操作函数。

**a. of\_find\_property**

找到节点中的属性。

函数原型：

extern struct property \*of\_find\_property(const struct device\_node \*np,

const char \*name,

int \*lenp);

参数np表示节点，我们要在这个节点中找到名为name的属性。

lenp用来保存这个属性的长度，即它的值的长度。

在设备树中，节点大概是这样：

xxx\_node {

xxx\_pp\_name = “hello”;

};

上述节点中，“xxx\_pp\_name”就是属性的名字，值的长度是6。

##### 11.7.3.3 获取属性的值

**a. of\_get\_property**

根据名字找到节点的属性，并且返回它的值。

函数原型：

/\*

\* Find a property with a given name for a given node

\* and return the value.

\*/

const void \*of\_get\_property(const struct device\_node \*np, const char \*name,

int \*lenp)

参数np表示节点，我们要在这个节点中找到名为name的属性，然后返回它的值。

lenp用来保存这个属性的长度，即它的值的长度。

**b. of\_property\_count\_elems\_of\_size**

根据名字找到节点的属性，确定它的值有多少个元素(elem)。

函数原型：

\* of\_property\_count\_elems\_of\_size - Count the number of elements in a property

\*

\* @np: device node from which the property value is to be read.

\* @propname: name of the property to be searched.

\* @elem\_size: size of the individual element

\*

\* Search for a property in a device node and count the number of elements of

\* size elem\_size in it. Returns number of elements on sucess, -EINVAL if the

\* property does not exist or its length does not match a multiple of elem\_size

\* and -ENODATA if the property does not have a value.

\*/

int of\_property\_count\_elems\_of\_size(const struct device\_node \*np,

const char \*propname, int elem\_size)

参数np表示节点，我们要在这个节点中找到名为propname的属性，然后返回下列结果：

return prop->length / elem\_size;

在设备树中，节点大概是这样：

xxx\_node {

xxx\_pp\_name = <0x50000000 1024> <0x60000000 2048>;

};

调用of\_property\_count\_elems\_of\_size(np, “xxx\_pp\_name”, 8)时，返回值是2；

调用of\_property\_count\_elems\_of\_size(np, “xxx\_pp\_name”, 4)时，返回值是4。

**c. 读整数u32/u64**

函数原型为：

static inline int of\_property\_read\_u32(const struct device\_node \*np,

const char \*propname,

u32 \*out\_value);

extern int of\_property\_read\_u64(const struct device\_node \*np,

const char \*propname, u64 \*out\_value);

在设备树中，节点大概是这样：

xxx\_node {

name1 = <0x50000000>;

name2 = <0x50000000 0x60000000>;

};

调用of\_property\_read\_u32 (np, “name1”, &val)时，val将得到值0x50000000；

调用of\_property\_read\_u64 (np, “name2”, &val)时，val将得到值0x0x6000000050000000。

**d. 读某个整数u32/u64**

函数原型为：

extern int of\_property\_read\_u32\_index(const struct device\_node \*np,

const char \*propname,

u32 index, u32 \*out\_value);

在设备树中，节点大概是这样：

xxx\_node {

name2 = <0x50000000 0x60000000>;

};

调用of\_property\_read\_u32 (np, “name2”, 1, &val)时，val将得到值0x0x60000000。

**e. 读数组**

函数原型为：

int of\_property\_read\_variable\_u8\_array(const struct device\_node \*np,

const char \*propname, u8 \*out\_values,

size\_t sz\_min, size\_t sz\_max);

int of\_property\_read\_variable\_u16\_array(const struct device\_node \*np,

const char \*propname, u16 \*out\_values,

size\_t sz\_min, size\_t sz\_max);

int of\_property\_read\_variable\_u32\_array(const struct device\_node \*np,

const char \*propname, u32 \*out\_values,

size\_t sz\_min, size\_t sz\_max);

int of\_property\_read\_variable\_u64\_array(const struct device\_node \*np,

const char \*propname, u64 \*out\_values,

size\_t sz\_min, size\_t sz\_max);

在设备树中，节点大概是这样：

xxx\_node {

name2 = <0x50000012 0x60000034>;

};

上述例子中属性name2的值，长度为8。

调用of\_property\_read\_variable\_u8\_array (np, “name2”, out\_values, 1, 10)时，out\_values中将会保存这8个字节： 0x12,0x00,0x00,0x50,0x34,0x00,0x00,0x60。

调用of\_property\_read\_variable\_u16\_array (np, “name2”, out\_values, 1, 10)时，out\_values中将会保存这4个16位数值： 0x0012, 0x5000,0x0034,0x6000。

总之，这些函数要么能取到全部的数值，要么一个数值都取不到；

如果值的长度在sz\_min和sz\_max之间，就返回全部的数值；否则一个数值都不返回。

**f. 读字符串**

函数原型为：

int of\_property\_read\_string(const struct device\_node \*np, const char \*propname,

const char \*\*out\_string);

返回节点np的属性(名为propname)的值，(\*out\_string)指向这个值，把它当作字符串。