Linux设备树详解

# 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。

为什么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

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

/dts-v1/;

#include "exynos4412.dtsi" //此设备树依赖于exynos4412.dtsi 文件

#include <dt-bindings/gpio/gpio.h> //gpio引脚配置文件

/ { //根节点 root node

model = "FriendlyARM TINY4412 board based on Exynos4412";

compatible = "friendlyarm,tiny4412", "samsung,exynos4412", "samsung,exynos4";

chosen {

stdout-path = &serial\_0;

};

## 节点语法规范说明

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

/\*\*\*\*\*示例\*\*\*\*\*/

/{

serial@101F0000{

……  
};

gpio@101F3000{

……  
};

interrupt-controller@10140000{

……  
};

spi@10115000{

…….  
};

external-bus{

……

};

i2c@1,0{

…….

rtc@58{

......  
 };

};

};

};

## 节点名及节点路径

/{

…

dm9000{

…  
};

…

};

节点名:dm9000

节点路径:/dm9000

## 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>;

};

…

};

…

};

## 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{

property=value;  
};

};

/\*移植者添加的节点\*/

/{

node{

property2=value;

};

};

/\*\*\*合并后的节点内容\*\*\*/

/{

node{

property2=value;

};

};

## 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”; /{

}; node{

}; property=value;

* status=”okay”;

/\*移植者添加的node节点\*/ };

/{ };

node{

status=”okay”;

};

};

## 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;  
};

};

/\*移植者添加的node节点\*/

&node{

property=value;

status = “okay”;  
}

/\*移植者添加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

## 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

chosen {

bootargs = “root=/dev/nfs rw nfsroot=192.168.1.1 console=ttyS0,115200”;

};

## 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

# 节点描述

头文件： include/of.h

struct device\_node {

const char \*name; //节点名

const char \*type; //设备类型

phandle phandle;

const char \*full\_name; //全路径节点名

struct fwnode\_handle fwnode;

struct property \*properties;

struct property \*deadprops; /\* removed properties \*/

struct device\_node \*parent; ////父节点指针

struct device\_node \*child; //子节点指针

struct device\_node \*sibling;

struct kobject kobj;

unsigned long \_flags;

void \*data;

#if defined(CONFIG\_SPARC)

const char \*path\_component\_name;

unsigned int unique\_id;

struct of\_irq\_controller \*irq\_trans;

#endif

};

## 功能：通过compatible属性查找指定节点

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

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

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

示例：

np = of\_find\_compatible\_node(NULL, NULL, "fsl,imx23-digctl");

digctrl = of\_iomap(np, 0);

linux-3.12.10-ti2013.12.01\arch\arm\boot\dts\imx28.dtsi

digctl: digctl@8001c000 {

compatible = "fsl,imx28-digctl", "fsl,imx23-digctl";

reg = <0x8001c000 0x2000>;

interrupts = <89>;

status = "disabled";

};

## 功能：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)

示例:

static DEFINE\_PCI\_DEVICE\_TABLE(adl\_pci6208\_pci\_table) = {

{ PCI\_VDEVICE(ADLINK, 0x6208), BOARD\_PCI6208 },

{ PCI\_VDEVICE(ADLINK, 0x6216), BOARD\_PCI6216 },

{ 0 }

};

MODULE\_DEVICE\_TABLE(pci, adl\_pci6208\_pci\_table);

## 功能：通过compatible属性查找指定节点

struct device\_node \*of\_find\_matching\_node(struct device\_node \*from,

const struct of\_device\_id \*matches);

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

示例：

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

## 功能：通过路径查找指定节点

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

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

data->current\_node = of\_find\_node\_by\_path("/");

## 功能：通过节点名查找指定节点

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

const char \*name);

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

# 节点属性

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

## 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 = <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>;

};

## 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” ;

...

};

...

};

vcc-supply = <&eth0\_power>;

};

设备树示例

/{

…

mydemo{

compatible = “fs4412,mydemo”;

…

}

}

/\*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)

}

}

## 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-cell = <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上都为一个单元。

### 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 {

compatlibe = “arm,cortex-a9”;

reg = <0>;  
};

cpu@1 {

compatible = “arm,cortex-a9”;

reg = <1>;  
};

};

### 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, p1011”;

reg = <0x101f0000 0x1000>;  
};

gpio@101f3000{  
 compatible = “arm,p1061”;

reg = <0x101f3000 0x1000

0x101f4000 0x0010>;

};

spi@10115000 {  
compatible = “arm,p1022”;

reg = <0x10115000 0x1000 >;

};

…

};

### 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>;   
};

};

### 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 //片选1，ethernet

1 0 0x10160000 0x10000 //片选2，i2c控制器

2 0 0x30000000 0x1000000>; //片选3 NOR FLASH

ehternet@0,0 {

compatible = “smc,smc91c1111”;

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

/{

compatible = “acme,coyotes-revenge”;

#address-cells = <1>;

#size-cells = <1>;

interrupt-parent = <&intc>;

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；

serial@101f0000 {

compatible = “arm,p1011”;

reg = <0x101f0000 0x1000 >;

interrupt = <1 0>;

interrupts 一个中断标识符列表，表示每一个中断输出信号

};

intc: interrupt-controller@10140000

compatible = “arm,p1190”;

reg = <0x10140000 0x1000>;

interrupt-controller;

#interrupt-cells = <2>;

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；  
};

};

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>;

};

pinctl@11000000 {

gpx0:gpx0{

gpio-controller;

#gpio-cells = <2>;S

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 = <&pgx0>;

interrupts = <6 4>;

davicom,no-eeprom;

mac-address = [00 0a 2d a6 55 a2];  
};

…

};

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。

## 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-celslls = <2>;

};

key@11400C24{

compatible = “fs4412,key”;

reg = <0x11400C24 0x4>;

intn-key = <&gpx1 2 2>;  
};

};

# 设备树函数属性

属性描述

头文件： include/of.h

struct property {

char \*name;//属性的名字

int length;//属性值的字节数

void \*value;//属性值的首地址

struct property \*next;

unsigned long \_flags;

unsigned int unique\_id;

struct bin\_attribute attr;

};

## 功能：提取指定属性的值

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

const char \*name, int \*lenp);

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

## 功能：得到属性值中数据的数量

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

const char \*propname, int elem\_size);

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

## 功能：得到属性值中指定标号的32位数据值

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

const char \*propname, u32 index, u32 \*out\_value);

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

## 功能：提取字符串（ 属性值）

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

const char \*propname, const char \*\*out\_string);

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

## 功能：提取默认属性“ #address-cells”的值

int of\_n\_size\_cells(struct device\_node \*np);

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

## 功能：提取默认属性”#size-cells”的值

int of\_n\_addr\_cells(struct device\_node \*np);

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

## 功能：提取I/O口地址

\_\_be32 \*of\_get\_address(struct device\_node \*dev, int index, u64 \*size, unsigned int \*flags);

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

## 功能：从设备树中提取I/O口地址转换成物理地址

u64 of\_translate\_address(struct device\_node \*dev, const \_\_be32 \*in\_addr);

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

## 功能：提取I/O口地址并映射成虚拟地址

void \_\_iomem \*of\_iomap(struct device\_node \*np, int index);

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

## 功能：提取I/O口地址并申请I/O资源及映射成虚拟地址

void \_\_iomem \*of\_io\_request\_and\_map(struct device\_node \*np, int index, const char \*name);

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

## 功能：从设备树中提取资源resource（ I/O地址）

int of\_address\_to\_resource(struct device\_node \*dev, int index, struct resource \*r);

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

# 定义于include/of\_gpio.h

### 功能：从设备树中提取gpio口

int of\_get\_named\_gpio(struct device\_node \*np, const char \*propname, int index);

参数：

const struct device\_node \*np - 设备节点指针

const char \*propname - 属性名

int index - gpio口引脚标号

返回值：

成功：得到GPIO口编号；失败：负数，绝对值是错误码

### 功能：从设备树中提取中断的数量

int of\_irq\_count(struct device\_node \*dev);

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

### 功能：从设备树中提取中断号

int of\_irq\_get(struct device\_node \*dev, int index);

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

### 从设备树中提取中断并映射好

unsigned int irq\_of\_parse\_and\_map(struct device\_node \*dev, int index);

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

### 功能：从设备树中提取MAC地址

void \*of\_get\_mac\_address(struct device\_node \*np);

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

## #ifdef CONFIG\_OF\_GPIO

/\*

\* OF GPIO chip for memory mapped banks

\*/

struct of\_mm\_gpio\_chip {

struct gpio\_chip gc;

void (\*save\_regs)(struct of\_mm\_gpio\_chip \*mm\_gc);

void \_\_iomem \*regs;

};

### 功能：

static inline struct of\_mm\_gpio\_chip \*to\_of\_mm\_gpio\_chip(struct gpio\_chip \*gc)

{

return container\_of(gc, struct of\_mm\_gpio\_chip, gc);

}

### 功能：

extern int of\_get\_named\_gpio\_flags(struct device\_node \*np,

const char \*list\_name, int index, enum of\_gpio\_flags \*flags);

### 功能：

extern unsigned int of\_gpio\_named\_count(struct device\_node \*np,

const char\* propname);

### 功能：

extern int of\_mm\_gpiochip\_add(struct device\_node \*np,

struct of\_mm\_gpio\_chip \*mm\_gc);

### 功能：

extern void of\_gpiochip\_add(struct gpio\_chip \*gc);

### 功能：

extern void of\_gpiochip\_remove(struct gpio\_chip \*gc);

### 功能：

extern struct gpio\_chip \*of\_node\_to\_gpiochip(struct device\_node \*np);

### 功能：

extern int of\_gpio\_simple\_xlate(struct gpio\_chip \*gc,

const struct of\_phandle\_args \*gpiospec,

u32 \*flags);

## #else /\* CONFIG\_OF\_GPIO \*/

/\* Drivers may not strictly depend on the GPIO support, so let them link. \*/

static inline int of\_get\_named\_gpio\_flags(struct device\_node \*np,

const char \*list\_name, int index, enum of\_gpio\_flags \*flags)

{

return -ENOSYS;

}

### 功能：

static inline unsigned int of\_gpio\_named\_count(struct device\_node \*np,

const char\* propname)

{

return 0;

}

### 功能：

static inline int of\_gpio\_simple\_xlate(struct gpio\_chip \*gc,

const struct of\_phandle\_args \*gpiospec,

u32 \*flags)

{

return -ENOSYS;

}

### 功能：

static inline void of\_gpiochip\_add(struct gpio\_chip \*gc) { }

### 功能：

static inline void of\_gpiochip\_remove(struct gpio\_chip \*gc) { }

## #endif /\* CONFIG\_OF\_GPIO \*/

### 功能：

/\*\*

\* of\_gpio\_count - Count GPIOs for a device

\* @np: device node to count GPIOs for

\*

\* The function returns the count of GPIOs specified for a node.

\*

\* Note that the empty GPIO specifiers counts too. For example,

\*

\* gpios = <0

\* &pio1 1 2

\* 0

\* &pio2 3 4>;

\*

\* defines four GPIOs (so this function will return 4), two of which

\* are not specified.

\*/

static inline unsigned int of\_gpio\_count(struct device\_node \*np)

{

return of\_gpio\_named\_count(np, "gpios");

}

### 功能：

/\*\*

\* of\_get\_gpio\_flags() - Get a GPIO number and flags to use with GPIO API

\* @np: device node to get GPIO from

\* @index: index of the GPIO

\* @flags: a flags pointer to fill in

\*

\* Returns GPIO number to use with Linux generic GPIO API, or one of the errno

\* value on the error condition. If @flags is not NULL the function also fills

\* in flags for the GPIO.

\*/

static inline int of\_get\_gpio\_flags(struct device\_node \*np, int index,

enum of\_gpio\_flags \*flags)

{

return of\_get\_named\_gpio\_flags(np, "gpios", index, flags);

}

### 功能：

/\*\*

\* of\_get\_named\_gpio() - Get a GPIO number to use with GPIO API

\* @np: device node to get GPIO from

\* @propname: Name of property containing gpio specifier(s)

\* @index: index of the GPIO

\*

\* Returns GPIO number to use with Linux generic GPIO API, or one of the errno

\* value on the error condition.

\*/

static inline int of\_get\_named\_gpio(struct device\_node \*np,

const char \*propname, int index)

{

return of\_get\_named\_gpio\_flags(np, propname, index, NULL);

}

### 功能：

/\*\*

\* of\_get\_gpio() - Get a GPIO number to use with GPIO API

\* @np: device node to get GPIO from

\* @index: index of the GPIO

\*

\* Returns GPIO number to use with Linux generic GPIO API, or one of the errno

\* value on the error condition.

\*/

static inline int of\_get\_gpio(struct device\_node \*np, int index)

{

return of\_get\_gpio\_flags(np, index, NULL);

}

#endif /\* \_\_LINUX\_OF\_GPIO\_H \*/

# 定义于inlcude/linux/of\_spi.h

## #if defined(CONFIG\_OF\_SPI) || defined(CONFIG\_OF\_SPI\_MODULE)

### 功能

void of\_register\_spi\_devices(struct spi\_master \*master);

### 功能

static inline void of\_register\_spi\_devices(struct spi\_master \*master)

#endif /\* CONFIG\_OF\_SPI \*/

#endif /\* \_\_LINUX\_OF\_SPI \*/

# 定义于include/linux/of\_i2c.h

## #if defined(CONFIG\_OF\_I2C) || defined(CONFIG\_OF\_I2C\_MODULE)

#include <linux/i2c.h>

### 功能

extern void of\_i2c\_register\_devices(struct i2c\_adapter \*adap);

### 功能

/\* must call put\_device() when done with returned i2c\_client device \*/

extern struct i2c\_client \*of\_find\_i2c\_device\_by\_node(struct device\_node \*node);

### 功能

#else

static inline void of\_i2c\_register\_devices(struct i2c\_adapter \*adap)

{

return;

}

#endif /\* CONFIG\_OF\_I2C \*/

#endif /\* \_\_LINUX\_OF\_I2C\_H \*/

# 定义于include/Linux/of\_mtd.h

## #ifdef CONFIG\_OF\_MTD

### 功能

extern const int of\_get\_nand\_ecc\_mode(struct device\_node \*np);

### 功能

int of\_get\_nand\_bus\_width(struct device\_node \*np);

### 功能

bool of\_get\_nand\_on\_flash\_bbt(struct device\_node \*np);

# 定义于include/linux/of\_address.h

## #ifdef CONFIG\_OF\_ADDRESS

### 功能

extern u64 of\_translate\_address(struct device\_node \*np, const \_\_be32 \*addr);

### 功能

extern int of\_address\_to\_resource(struct device\_node \*dev, int index,

struct resource \*r);

### 功能

extern struct device\_node \*of\_find\_matching\_node\_by\_address(

struct device\_node \*from,

const struct of\_device\_id \*matches,

u64 base\_address);

### 功能

extern void \_\_iomem \*of\_iomap(struct device\_node \*device, int index);

### 功能

/\* Extract an address from a device, returns the region size and

\* the address space flags too. The PCI version uses a BAR number

\* instead of an absolute index

\*/

extern const u32 \*of\_get\_address(struct device\_node \*dev, int index,

u64 \*size, unsigned int \*flags);

### 功能

#ifndef pci\_address\_to\_pio

static inline unsigned long pci\_address\_to\_pio(phys\_addr\_t addr) { return -1; }

#define pci\_address\_to\_pio pci\_address\_to\_pio

#endif

## #else /\* CONFIG\_OF\_ADDRESS \*/

### 功能

static inline int of\_address\_to\_resource(struct device\_node \*dev, int index,

struct resource \*r)

{

return -EINVAL;

}

### 功能

static inline struct device\_node \*of\_find\_matching\_node\_by\_address(

struct device\_node \*from,

const struct of\_device\_id \*matches,

u64 base\_address)

{

return NULL;

}

### 功能

static inline void \_\_iomem \*of\_iomap(struct device\_node \*device, int index)

{

return NULL;

}

### 功能

static inline const u32 \*of\_get\_address(struct device\_node \*dev, int index,

u64 \*size, unsigned int \*flags)

{

return NULL;

}

## #endif /\* CONFIG\_OF\_ADDRESS \*/

### 功能

#if defined(CONFIG\_OF\_ADDRESS) && defined(CONFIG\_PCI)

extern const \_\_be32 \*of\_get\_pci\_address(struct device\_node \*dev, int bar\_no,

u64 \*size, unsigned int \*flags);

### 功能

extern int of\_pci\_address\_to\_resource(struct device\_node \*dev, int bar,

struct resource \*r);

### 功能

## #else /\* CONFIG\_OF\_ADDRESS && CONFIG\_PCI \*/

static inline int of\_pci\_address\_to\_resource(struct device\_node \*dev, int bar,

struct resource \*r)

{

return -ENOSYS;

}

### 功能

static inline const \_\_be32 \*of\_get\_pci\_address(struct device\_node \*dev,

int bar\_no, u64 \*size, unsigned int \*flags)

{

return NULL;

}

# 定义于inlcude/linux/of\_fdt.h

Definitions for working with the Flattened Device Tree data format

/\*

\* This is what gets passed to the kernel by prom\_init or kexec

\*

\* The dt struct contains the device tree structure, full pathes and

\* property contents. The dt strings contain a separate block with just

\* the strings for the property names, and is fully page aligned and

\* self contained in a page, so that it can be kept around by the kernel,

\* each property name appears only once in this page (cheap compression)

\*

\* the mem\_rsvmap contains a map of reserved ranges of physical memory,

\* passing it here instead of in the device-tree itself greatly simplifies

\* the job of everybody. It's just a list of u64 pairs (base/size) that

\* ends when size is 0

\*/

struct boot\_param\_header {

\_\_be32 magic; /\* magic word OF\_DT\_HEADER \*/

\_\_be32 totalsize; /\* total size of DT block \*/

\_\_be32 off\_dt\_struct; /\* offset to structure \*/

\_\_be32 off\_dt\_strings; /\* offset to strings \*/

\_\_be32 off\_mem\_rsvmap; /\* offset to memory reserve map \*/

\_\_be32 version; /\* format version \*/

\_\_be32 last\_comp\_version; /\* last compatible version \*/

/\* version 2 fields below \*/

\_\_be32 boot\_cpuid\_phys; /\* Physical CPU id we're booting on \*/

/\* version 3 fields below \*/

\_\_be32 dt\_strings\_size; /\* size of the DT strings block \*/

/\* version 17 fields below \*/

\_\_be32 dt\_struct\_size; /\* size of the DT structure block \*/

};

## #if defined(CONFIG\_OF\_FLATTREE)

struct device\_node;

### 功能

/\* For scanning an arbitrary device-tree at any time \*/

extern char \*of\_fdt\_get\_string(struct boot\_param\_header \*blob, u32 offset);

### 功能

extern void \*of\_fdt\_get\_property(struct boot\_param\_header \*blob,

unsigned long node,

const char \*name,

unsigned long \*size);

### 功能

extern int of\_fdt\_is\_compatible(struct boot\_param\_header \*blob,

unsigned long node,

const char \*compat);

### 功能

extern int of\_fdt\_match(struct boot\_param\_header \*blob, unsigned long node,

const char \*const \*compat);

### 功能

extern void of\_fdt\_unflatten\_tree(unsigned long \*blob,

struct device\_node \*\*mynodes);

### 功能

/\* TBD: Temporary export of fdt globals - remove when code fully merged \*/

extern int \_\_initdata dt\_root\_addr\_cells;

extern int \_\_initdata dt\_root\_size\_cells;

extern struct boot\_param\_header \*initial\_boot\_params;

### 功能

/\* For scanning the flat device-tree at boot time \*/

extern char \*find\_flat\_dt\_string(u32 offset);

extern int of\_scan\_flat\_dt(int (\*it)(unsigned long node, const char \*uname,

int depth, void \*data),

void \*data);

### 功能

extern void \*of\_get\_flat\_dt\_prop(unsigned long node, const char \*name,

unsigned long \*size);

### 功能

extern int of\_flat\_dt\_is\_compatible(unsigned long node, const char \*name);

### 功能

extern int of\_flat\_dt\_match(unsigned long node, const char \*const \*matches);

### 功能

extern unsigned long of\_get\_flat\_dt\_root(void);

### 功能

extern int early\_init\_dt\_scan\_chosen(unsigned long node, const char \*uname,

int depth, void \*data);

### 功能

extern void early\_init\_dt\_check\_for\_initrd(unsigned long node);

### 功能

extern int early\_init\_dt\_scan\_memory(unsigned long node, const char \*uname,

int depth, void \*data);

### 功能

extern void early\_init\_dt\_add\_memory\_arch(u64 base, u64 size);

### 功能

extern void \* early\_init\_dt\_alloc\_memory\_arch(u64 size, u64 align);

### 功能

extern u64 dt\_mem\_next\_cell(int s, \_\_be32 \*\*cellp);

/\*

\* If BLK\_DEV\_INITRD, the fdt early init code will call this function,

\* to be provided by the arch code. start and end are specified as

\* physical addresses.

\*/

## #ifdef CONFIG\_BLK\_DEV\_INITRD

extern void early\_init\_dt\_setup\_initrd\_arch(unsigned long start,

unsigned long end);

#endif

### 功能

/\* Early flat tree scan hooks \*/

extern int early\_init\_dt\_scan\_root(unsigned long node, const char \*uname,

int depth, void \*data);

### 功能

/\* Other Prototypes \*/

extern void unflatten\_device\_tree(void);

extern void early\_init\_devtree(void \*);

## #else /\* CONFIG\_OF\_FLATTREE \*/

static inline void unflatten\_device\_tree(void) {}

#endif /\* CONFIG\_OF\_FLATTREE \*/

# 定义于include/linux/of\_net.h

## #ifdef CONFIG\_OF\_NET

#include <linux/of.h>

### 功能

extern const int of\_get\_phy\_mode(struct device\_node \*np);

### 功能

extern const void \*of\_get\_mac\_address(struct device\_node \*np);

#endif

# 定义于include/linux/of.h

#ifndef \_LINUX\_OF\_H

#define \_LINUX\_OF\_H

/\*

\* Definitions for talking to the Open Firmware PROM on

\* Power Macintosh and other computers.

\*

\* Copyright (C) 1996-2005 Paul Mackerras.

\*

\* Updates for PPC64 by Peter Bergner & David Engebretsen, IBM Corp.

\* Updates for SPARC64 by David S. Miller

\* Derived from PowerPC and Sparc prom.h files by Stephen Rothwell, IBM Corp.

\*

\* This program is free software; you can redistribute it and/or

\* modify it under the terms of the GNU General Public License

\* as published by the Free Software Foundation; either version

\* 2 of the License, or (at your option) any later version.

\*/

typedef u32 phandle;

typedef u32 ihandle;

struct property {

char \*name;

int length;

void \*value;

struct property \*next;

unsigned long \_flags;

unsigned int unique\_id;

};

#if defined(CONFIG\_SPARC)

struct of\_irq\_controller;

#endif

struct device\_node {

const char \*name;

const char \*type;

phandle phandle;

const char \*full\_name;

struct property \*properties;

struct property \*deadprops; /\* removed properties \*/

struct device\_node \*parent;

struct device\_node \*child;

struct device\_node \*sibling;

struct device\_node \*next; /\* next device of same type \*/

struct device\_node \*allnext; /\* next in list of all nodes \*/

struct proc\_dir\_entry \*pde; /\* this node's proc directory \*/

struct kref kref;

unsigned long \_flags;

void \*data;

#if defined(CONFIG\_SPARC)

const char \*path\_component\_name;

unsigned int unique\_id;

struct of\_irq\_controller \*irq\_trans;

#endif

};

#define MAX\_PHANDLE\_ARGS 8

struct of\_phandle\_args {

struct device\_node \*np;

int args\_count;

uint32\_t args[MAX\_PHANDLE\_ARGS];

};

## #ifdef CONFIG\_OF\_DYNAMIC

### 功能

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

### 功能

extern void of\_node\_put(struct device\_node \*node);

## #else /\* CONFIG\_OF\_DYNAMIC \*/

### 功能

/\* Dummy ref counting routines - to be implemented later \*/

static inline struct device\_node \*of\_node\_get(struct device\_node \*node)

{

return node;

}

### 功能

static inline void of\_node\_put(struct device\_node \*node) { }

#endif /\* !CONFIG\_OF\_DYNAMIC \*/

## #ifdef CONFIG\_OF

/\* Pointer for first entry in chain of all nodes. \*/

extern struct device\_node \*of\_allnodes;

extern struct device\_node \*of\_chosen;

extern struct device\_node \*of\_aliases;

extern raw\_spinlock\_t devtree\_lock;

### 功能

static inline bool of\_have\_populated\_dt(void)

{

return of\_allnodes != NULL;

}

### 功能

static inline bool of\_node\_is\_root(const struct device\_node \*node)

{

return node && (node->parent == NULL);

}

### 功能

static inline int of\_node\_check\_flag(struct device\_node \*n, unsigned long flag)

{

return test\_bit(flag, &n->\_flags);

}

### 功能

static inline void of\_node\_set\_flag(struct device\_node \*n, unsigned long flag)

{

set\_bit(flag, &n->\_flags);

}

### 功能

extern struct device\_node \*of\_find\_all\_nodes(struct device\_node \*prev);

### 功能

/\*

\* OF address retrieval & translation

\*/

/\* Helper to read a big number; size is in cells (not bytes) \*/

static inline u64 of\_read\_number(const \_\_be32 \*cell, int size)

{

u64 r = 0;

while (size--)

r = (r << 32) | be32\_to\_cpu(\*(cell++));

return r;

}

/\* Like of\_read\_number, but we want an unsigned long result \*/

static inline unsigned long of\_read\_ulong(const \_\_be32 \*cell, int size)

{

/\* toss away upper bits if unsigned long is smaller than u64 \*/

return of\_read\_number(cell, size);

}

#include <asm/prom.h>

/\* Default #address and #size cells. Allow arch asm/prom.h to override \*/

#if !defined(OF\_ROOT\_NODE\_ADDR\_CELLS\_DEFAULT)

#define OF\_ROOT\_NODE\_ADDR\_CELLS\_DEFAULT 1

#define OF\_ROOT\_NODE\_SIZE\_CELLS\_DEFAULT 1

#endif

/\* Default string compare functions, Allow arch asm/prom.h to override \*/

#if !defined(of\_compat\_cmp)

#define of\_compat\_cmp(s1, s2, l) strcasecmp((s1), (s2))

#define of\_prop\_cmp(s1, s2) strcmp((s1), (s2))

#define of\_node\_cmp(s1, s2) strcasecmp((s1), (s2))

#endif

/\* flag descriptions \*/

#define OF\_DYNAMIC 1 /\* node and properties were allocated via kmalloc \*/

#define OF\_DETACHED 2 /\* node has been detached from the device tree \*/

#define OF\_IS\_DYNAMIC(x) test\_bit(OF\_DYNAMIC, &x->\_flags)

#define OF\_MARK\_DYNAMIC(x) set\_bit(OF\_DYNAMIC, &x->\_flags)

#define OF\_BAD\_ADDR ((u64)-1)

### 功能

static inline const char \*of\_node\_full\_name(const struct device\_node \*np)

{

return np ? np->full\_name : "<no-node>";

}

### 功能

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

const char \*name);

### 功能

#define for\_each\_node\_by\_name(dn, name) \

for (dn = of\_find\_node\_by\_name(NULL, name); dn; \

dn = of\_find\_node\_by\_name(dn, name))

### 功能

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

const char \*type);

### 功能

#define for\_each\_node\_by\_type(dn, type) \

for (dn = of\_find\_node\_by\_type(NULL, type); dn; \

dn = of\_find\_node\_by\_type(dn, type))

### 功能

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

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

### 功能

#define for\_each\_compatible\_node(dn, type, compatible) \

for (dn = of\_find\_compatible\_node(NULL, type, compatible); dn; \

dn = of\_find\_compatible\_node(dn, type, compatible))

### 功能

extern struct device\_node \*of\_find\_matching\_node\_and\_match(

struct device\_node \*from,

const struct of\_device\_id \*matches,

const struct of\_device\_id \*\*match);

### 功能

static inline struct device\_node \*of\_find\_matching\_node(

struct device\_node \*from,

const struct of\_device\_id \*matches)

{

return of\_find\_matching\_node\_and\_match(from, matches, NULL);

}

### 功能

#define for\_each\_matching\_node(dn, matches) \

for (dn = of\_find\_matching\_node(NULL, matches); dn; \

dn = of\_find\_matching\_node(dn, matches))

### 功能

#define for\_each\_matching\_node\_and\_match(dn, matches, match) \

for (dn = of\_find\_matching\_node\_and\_match(NULL, matches, match); \

dn; dn = of\_find\_matching\_node\_and\_match(dn, matches, match))

### 功能

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

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

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

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

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

struct device\_node \*prev);

extern struct device\_node \*of\_get\_next\_available\_child(

const struct device\_node \*node, struct device\_node \*prev);

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

const char \*name);

### 功能

#define for\_each\_child\_of\_node(parent, child) \

for (child = of\_get\_next\_child(parent, NULL); child != NULL; \

child = of\_get\_next\_child(parent, child))

### 功能

#define for\_each\_available\_child\_of\_node(parent, child) \

for (child = of\_get\_next\_available\_child(parent, NULL); child != NULL; \

child = of\_get\_next\_available\_child(parent, child))

### 功能

static inline int of\_get\_child\_count(const struct device\_node \*np)

{

struct device\_node \*child;

int num = 0;

for\_each\_child\_of\_node(np, child)

num++;

return num;

}

### 功能

extern struct device\_node \*of\_find\_node\_with\_property(

struct device\_node \*from, const char \*prop\_name);

#define for\_each\_node\_with\_property(dn, prop\_name) \

for (dn = of\_find\_node\_with\_property(NULL, prop\_name); dn; \

dn = of\_find\_node\_with\_property(dn, prop\_name))

### 功能

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

const char \*name,

int \*lenp);

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

const char \*propname,

u32 index, u32 \*out\_value);

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

const char \*propname, u8 \*out\_values, size\_t sz);

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

const char \*propname, u16 \*out\_values, size\_t sz);

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

const char \*propname,

u32 \*out\_values,

size\_t sz);

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

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

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

const char \*propname,

const char \*\*out\_string);

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

const char \*propname,

int index, const char \*\*output);

extern int of\_property\_match\_string(struct device\_node \*np,

const char \*propname,

const char \*string);

extern int of\_property\_count\_strings(struct device\_node \*np,

const char \*propname);

extern int of\_device\_is\_compatible(const struct device\_node \*device,

const char \*);

extern int of\_device\_is\_available(const struct device\_node \*device);

extern const void \*of\_get\_property(const struct device\_node \*node,

const char \*name,

int \*lenp);

extern struct device\_node \*of\_get\_cpu\_node(int cpu, unsigned int \*thread);

#define for\_each\_property\_of\_node(dn, pp) \

for (pp = dn->properties; pp != NULL; pp = pp->next)

extern int of\_n\_addr\_cells(struct device\_node \*np);

extern int of\_n\_size\_cells(struct device\_node \*np);

extern const struct of\_device\_id \*of\_match\_node(

const struct of\_device\_id \*matches, const struct device\_node \*node);

extern int of\_modalias\_node(struct device\_node \*node, char \*modalias, int len);

extern struct device\_node \*of\_parse\_phandle(const struct device\_node \*np,

const char \*phandle\_name,

int index);

extern int of\_parse\_phandle\_with\_args(const struct device\_node \*np,

const char \*list\_name, const char \*cells\_name, int index,

struct of\_phandle\_args \*out\_args);

extern int of\_parse\_phandle\_with\_fixed\_args(const struct device\_node \*np,

const char \*list\_name, int cells\_count, int index,

struct of\_phandle\_args \*out\_args);

extern int of\_count\_phandle\_with\_args(const struct device\_node \*np,

const char \*list\_name, const char \*cells\_name);

extern void of\_alias\_scan(void \* (\*dt\_alloc)(u64 size, u64 align));

extern int of\_alias\_get\_id(struct device\_node \*np, const char \*stem);

extern int of\_machine\_is\_compatible(const char \*compat);

extern int of\_add\_property(struct device\_node \*np, struct property \*prop);

extern int of\_remove\_property(struct device\_node \*np, struct property \*prop);

extern int of\_update\_property(struct device\_node \*np, struct property \*newprop);

/\* For updating the device tree at runtime \*/

#define OF\_RECONFIG\_ATTACH\_NODE 0x0001

#define OF\_RECONFIG\_DETACH\_NODE 0x0002

#define OF\_RECONFIG\_ADD\_PROPERTY 0x0003

#define OF\_RECONFIG\_REMOVE\_PROPERTY 0x0004

#define OF\_RECONFIG\_UPDATE\_PROPERTY 0x0005

struct of\_prop\_reconfig {

struct device\_node \*dn;

struct property \*prop;

};

extern int of\_reconfig\_notifier\_register(struct notifier\_block \*);

extern int of\_reconfig\_notifier\_unregister(struct notifier\_block \*);

extern int of\_reconfig\_notify(unsigned long, void \*);

extern int of\_attach\_node(struct device\_node \*);

extern int of\_detach\_node(struct device\_node \*);

#define of\_match\_ptr(\_ptr) (\_ptr)

/\*

\* struct property \*prop;

\* const \_\_be32 \*p;

\* u32 u;

\*

\* of\_property\_for\_each\_u32(np, "propname", prop, p, u)

\* printk("U32 value: %x\n", u);

\*/

const \_\_be32 \*of\_prop\_next\_u32(struct property \*prop, const \_\_be32 \*cur,

u32 \*pu);

/\*

\* struct property \*prop;

\* const char \*s;

\*

\* of\_property\_for\_each\_string(np, "propname", prop, s)

\* printk("String value: %s\n", s);

\*/

const char \*of\_prop\_next\_string(struct property \*prop, const char \*cur);

int of\_device\_is\_stdout\_path(struct device\_node \*dn);

## #else /\* CONFIG\_OF \*/

### 功能

static inline const char\* of\_node\_full\_name(struct device\_node \*np)

{

return "<no-node>";

}

### 功能

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

const char \*name)

{

return NULL;

}

### 功能

static inline struct device\_node \*of\_get\_parent(const struct device\_node \*node)

{

return NULL;

}

### 功能

static inline bool of\_have\_populated\_dt(void)

{

return false;

}

### 功能

#define for\_each\_child\_of\_node(parent, child) \

while (0)

static inline struct device\_node \*of\_get\_child\_by\_name(

const struct device\_node \*node,

const char \*name)

{

return NULL;

}

### 功能

static inline int of\_get\_child\_count(const struct device\_node \*np)

{

return 0;

}

### 功能

static inline int of\_device\_is\_compatible(const struct device\_node \*device,

const char \*name)

{

return 0;

}

### 功能

static inline int of\_device\_is\_available(const struct device\_node \*device)

{

return 0;

}

### 功能

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

const char \*name,

int \*lenp)

{

return NULL;

}

### 功能

static inline struct device\_node \*of\_find\_compatible\_node(

struct device\_node \*from,

const char \*type,

const char \*compat)

{

return NULL;

}

### 功能

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

const char \*propname, u32 index, u32 \*out\_value)

{

return -ENOSYS;

}

### 功能

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

const char \*propname, u8 \*out\_values, size\_t sz)

{

return -ENOSYS;

}

### 功能

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

const char \*propname, u16 \*out\_values, size\_t sz)

{

return -ENOSYS;

}

### 功能

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

const char \*propname,

u32 \*out\_values, size\_t sz)

{

return -ENOSYS;

}

### 功能

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

const char \*propname,

const char \*\*out\_string)

{

return -ENOSYS;

}

### 功能

static inline int of\_property\_read\_string\_index(struct device\_node \*np,

const char \*propname, int index,

const char \*\*out\_string)

{

return -ENOSYS;

}

### 功能

static inline int of\_property\_count\_strings(struct device\_node \*np,

const char \*propname)

{

return -ENOSYS;

}

### 功能

static inline const void \*of\_get\_property(const struct device\_node \*node,

const char \*name,

int \*lenp)

{

return NULL;

}

### 功能

static inline struct device\_node \*of\_get\_cpu\_node(int cpu,

unsigned int \*thread)

{

return NULL;

}

### 功能

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

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

{

return -ENOSYS;

}

### 功能

static inline int of\_property\_match\_string(struct device\_node \*np,

const char \*propname,

const char \*string)

{

return -ENOSYS;

}

### 功能

static inline struct device\_node \*of\_parse\_phandle(const struct device\_node \*np,

const char \*phandle\_name,

int index)

{

return NULL;

}

### 功能

static inline int of\_parse\_phandle\_with\_args(struct device\_node \*np,

const char \*list\_name,

const char \*cells\_name,

int index,

struct of\_phandle\_args \*out\_args)

{

return -ENOSYS;

}

### 功能

static inline int of\_parse\_phandle\_with\_fixed\_args(const struct device\_node \*np,

const char \*list\_name, int cells\_count, int index,

struct of\_phandle\_args \*out\_args)

{

return -ENOSYS;

}

### 功能

static inline int of\_count\_phandle\_with\_args(struct device\_node \*np,

const char \*list\_name,

const char \*cells\_name)

{

return -ENOSYS;

}

### 功能

static inline int of\_alias\_get\_id(struct device\_node \*np, const char \*stem)

{

return -ENOSYS;

}

### 功能

static inline int of\_machine\_is\_compatible(const char \*compat)

{

return 0;

}

### 功能

static inline int of\_device\_is\_stdout\_path(struct device\_node \*dn)

{

return 0;

}

### 功能

static inline const \_\_be32 \*of\_prop\_next\_u32(struct property \*prop,

const \_\_be32 \*cur, u32 \*pu)

{

return NULL;

}

### 功能

static inline const char \*of\_prop\_next\_string(struct property \*prop,

const char \*cur)

{

return NULL;

}

#define of\_match\_ptr(\_ptr) NULL

#define of\_match\_node(\_matches, \_node) NULL

#endif /\* CONFIG\_OF \*/

## #ifndef of\_node\_to\_nid

### 功能

static inline int of\_node\_to\_nid(struct device\_node \*np)

{

return numa\_node\_id();

}

#define of\_node\_to\_nid of\_node\_to\_nid

#endif

### 功能

/\*\*

\* of\_property\_read\_bool - Findfrom a property

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

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

\*

\* Search for a property in a device node.

\* Returns true if the property exist false otherwise.

\*/

static inline bool of\_property\_read\_bool(const struct device\_node \*np,

const char \*propname)

{

struct property \*prop = of\_find\_property(np, propname, NULL);

return prop ? true : false;

}

### 功能

static inline int of\_property\_read\_u8(const struct device\_node \*np,

const char \*propname,

u8 \*out\_value)

{

return of\_property\_read\_u8\_array(np, propname, out\_value, 1);

}

### 功能

static inline int of\_property\_read\_u16(const struct device\_node \*np,

const char \*propname,

u16 \*out\_value)

{

return of\_property\_read\_u16\_array(np, propname, out\_value, 1);

}

### 功能

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

const char \*propname,

u32 \*out\_value)

{

return of\_property\_read\_u32\_array(np, propname, out\_value, 1);

}

### 功能

#define of\_property\_for\_each\_u32(np, propname, prop, p, u) \

for (prop = of\_find\_property(np, propname, NULL), \

p = of\_prop\_next\_u32(prop, NULL, &u); \

p; \

p = of\_prop\_next\_u32(prop, p, &u))

#define of\_property\_for\_each\_string(np, propname, prop, s) \

for (prop = of\_find\_property(np, propname, NULL), \

s = of\_prop\_next\_string(prop, NULL); \

s; \

s = of\_prop\_next\_string(prop, s))

#if defined(CONFIG\_PROC\_FS) && defined(CONFIG\_PROC\_DEVICETREE)

extern void proc\_device\_tree\_add\_node(struct device\_node \*, struct proc\_dir\_entry \*);

extern void proc\_device\_tree\_add\_prop(struct proc\_dir\_entry \*pde, struct property \*prop);

extern void proc\_device\_tree\_remove\_prop(struct proc\_dir\_entry \*pde,

struct property \*prop);

extern void proc\_device\_tree\_update\_prop(struct proc\_dir\_entry \*pde,

struct property \*newprop,

struct property \*oldprop);

#endif

#endif /\* \_LINUX\_OF\_H \*/

/\*

\* Procedures for creating, accessing and interpreting the device tree.

\*

\* Paul Mackerras August 1996.

\* Copyright (C) 1996-2005 Paul Mackerras.

\*

\* Adapted for 64bit PowerPC by Dave Engebretsen and Peter Bergner.

\* {engebret|bergner}@us.ibm.com

\*

\* Adapted for sparc and sparc64 by David S. Miller davem@davemloft.net

\*

\* Reconsolidated from arch/x/kernel/prom.c by Stephen Rothwell and

\* Grant Likely.

\*

\* This program is free software; you can redistribute it and/or

\* modify it under the terms of the GNU General Public License

\* as published by the Free Software Foundation; either version

\* 2 of the License, or (at your option) any later version.

\*/

#include <linux/ctype.h>

#include <linux/cpu.h>

#include <linux/module.h>

#include <linux/of.h>

#include <linux/spinlock.h>

#include <linux/slab.h>

#include <linux/proc\_fs.h>

#include "of\_private.h"

LIST\_HEAD(aliases\_lookup);

struct device\_node \*of\_allnodes;

EXPORT\_SYMBOL(of\_allnodes);

struct device\_node \*of\_chosen;

struct device\_node \*of\_aliases;

static struct device\_node \*of\_stdout;

DEFINE\_MUTEX(of\_aliases\_mutex);

/\* use when traversing tree through the allnext, child, sibling,

\* or parent members of struct device\_node.

\*/

DEFINE\_RAW\_SPINLOCK(devtree\_lock);

int of\_n\_addr\_cells(struct device\_node \*np)

{

const \_\_be32 \*ip;

do {

if (np->parent)

np = np->parent;

ip = of\_get\_property(np, "#address-cells", NULL);

if (ip)

return be32\_to\_cpup(ip);

} while (np->parent);

/\* No #address-cells property for the root node \*/

return OF\_ROOT\_NODE\_ADDR\_CELLS\_DEFAULT;

}

EXPORT\_SYMBOL(of\_n\_addr\_cells);

int of\_n\_size\_cells(struct device\_node \*np)

{

const \_\_be32 \*ip;

do {

if (np->parent)

np = np->parent;

ip = of\_get\_property(np, "#size-cells", NULL);

if (ip)

return be32\_to\_cpup(ip);

} while (np->parent);

/\* No #size-cells property for the root node \*/

return OF\_ROOT\_NODE\_SIZE\_CELLS\_DEFAULT;

}

EXPORT\_SYMBOL(of\_n\_size\_cells);

#if defined(CONFIG\_OF\_DYNAMIC)

/\*\*

\* of\_node\_get - Increment refcount of a node

\* @node: Node to inc refcount, NULL is supported to

\* simplify writing of callers

\*

\* Returns node.

\*/

struct device\_node \*of\_node\_get(struct device\_node \*node)

{

if (node)

kref\_get(&node->kref);

return node;

}

EXPORT\_SYMBOL(of\_node\_get);

static inline struct device\_node \*kref\_to\_device\_node(struct kref \*kref)

{

return container\_of(kref, struct device\_node, kref);

}

/\*\*

\* of\_node\_release - release a dynamically allocated node

\* @kref: kref element of the node to be released

\*

\* In of\_node\_put() this function is passed to kref\_put()

\* as the destructor.

\*/

static void of\_node\_release(struct kref \*kref)

{

struct device\_node \*node = kref\_to\_device\_node(kref);

struct property \*prop = node->properties;

/\* We should never be releasing nodes that haven't been detached. \*/

if (!of\_node\_check\_flag(node, OF\_DETACHED)) {

pr\_err("ERROR: Bad of\_node\_put() on %s\n", node->full\_name);

dump\_stack();

kref\_init(&node->kref);

return;

}

if (!of\_node\_check\_flag(node, OF\_DYNAMIC))

return;

while (prop) {

struct property \*next = prop->next;

kfree(prop->name);

kfree(prop->value);

kfree(prop);

prop = next;

if (!prop) {

prop = node->deadprops;

node->deadprops = NULL;

}

}

kfree(node->full\_name);

kfree(node->data);

kfree(node);

}

/\*\*

\* of\_node\_put - Decrement refcount of a node

\* @node: Node to dec refcount, NULL is supported to

\* simplify writing of callers

\*

\*/

void of\_node\_put(struct device\_node \*node)

{

if (node)

kref\_put(&node->kref, of\_node\_release);

}

EXPORT\_SYMBOL(of\_node\_put);

#endif /\* CONFIG\_OF\_DYNAMIC \*/

static struct property \*\_\_of\_find\_property(const struct device\_node \*np,

const char \*name, int \*lenp)

{

struct property \*pp;

if (!np)

return NULL;

for (pp = np->properties; pp; pp = pp->next) {

if (of\_prop\_cmp(pp->name, name) == 0) {

if (lenp)

\*lenp = pp->length;

break;

}

}

return pp;

}

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

const char \*name,

int \*lenp)

{

struct property \*pp;

unsigned long flags;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

pp = \_\_of\_find\_property(np, name, lenp);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return pp;

}

EXPORT\_SYMBOL(of\_find\_property);

/\*\*

\* of\_find\_all\_nodes - Get next node in global list

\* @prev: Previous node or NULL to start iteration

\* of\_node\_put() will be called on it

\*

\* Returns a node pointer with refcount incremented, use

\* of\_node\_put() on it when done.

\*/

struct device\_node \*of\_find\_all\_nodes(struct device\_node \*prev)

{

struct device\_node \*np;

unsigned long flags;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

np = prev ? prev->allnext : of\_allnodes;

for (; np != NULL; np = np->allnext)

if (of\_node\_get(np))

break;

of\_node\_put(prev);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return np;

}

EXPORT\_SYMBOL(of\_find\_all\_nodes);

/\*

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

\* and return the value.

\*/

static const void \*\_\_of\_get\_property(const struct device\_node \*np,

const char \*name, int \*lenp)

{

struct property \*pp = \_\_of\_find\_property(np, name, lenp);

return pp ? pp->value : NULL;

}

/\*

\* 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)

{

struct property \*pp = of\_find\_property(np, name, lenp);

return pp ? pp->value : NULL;

}

EXPORT\_SYMBOL(of\_get\_property);

/\*

\* arch\_match\_cpu\_phys\_id - Match the given logical CPU and physical id

\*

\* @cpu: logical cpu index of a core/thread

\* @phys\_id: physical identifier of a core/thread

\*

\* CPU logical to physical index mapping is architecture specific.

\* However this \_\_weak function provides a default match of physical

\* id to logical cpu index. phys\_id provided here is usually values read

\* from the device tree which must match the hardware internal registers.

\*

\* Returns true if the physical identifier and the logical cpu index

\* correspond to the same core/thread, false otherwise.

\*/

bool \_\_weak arch\_match\_cpu\_phys\_id(int cpu, u64 phys\_id)

{

return (u32)phys\_id == cpu;

}

/\*\*

\* Checks if the given "prop\_name" property holds the physical id of the

\* core/thread corresponding to the logical cpu 'cpu'. If 'thread' is not

\* NULL, local thread number within the core is returned in it.

\*/

static bool \_\_of\_find\_n\_match\_cpu\_property(struct device\_node \*cpun,

const char \*prop\_name, int cpu, unsigned int \*thread)

{

const \_\_be32 \*cell;

int ac, prop\_len, tid;

u64 hwid;

ac = of\_n\_addr\_cells(cpun);

cell = of\_get\_property(cpun, prop\_name, &prop\_len);

if (!cell)

return false;

prop\_len /= sizeof(\*cell);

for (tid = 0; tid < prop\_len; tid++) {

hwid = of\_read\_number(cell, ac);

if (arch\_match\_cpu\_phys\_id(cpu, hwid)) {

if (thread)

\*thread = tid;

return true;

}

cell += ac;

}

return false;

}

/\*\*

\* of\_get\_cpu\_node - Get device node associated with the given logical CPU

\*

\* @cpu: CPU number(logical index) for which device node is required

\* @thread: if not NULL, local thread number within the physical core is

\* returned

\*

\* The main purpose of this function is to retrieve the device node for the

\* given logical CPU index. It should be used to initialize the of\_node in

\* cpu device. Once of\_node in cpu device is populated, all the further

\* references can use that instead.

\*

\* CPU logical to physical index mapping is architecture specific and is built

\* before booting secondary cores. This function uses arch\_match\_cpu\_phys\_id

\* which can be overridden by architecture specific implementation.

\*

\* Returns a node pointer for the logical cpu if found, else NULL.

\*/

struct device\_node \*of\_get\_cpu\_node(int cpu, unsigned int \*thread)

{

struct device\_node \*cpun, \*cpus;

cpus = of\_find\_node\_by\_path("/cpus");

if (!cpus)

return NULL;

for\_each\_child\_of\_node(cpus, cpun) {

if (of\_node\_cmp(cpun->type, "cpu"))

continue;

/\* Check for non-standard "ibm,ppc-interrupt-server#s" property

\* for thread ids on PowerPC. If it doesn't exist fallback to

\* standard "reg" property.

\*/

if (IS\_ENABLED(CONFIG\_PPC) &&

\_\_of\_find\_n\_match\_cpu\_property(cpun,

"ibm,ppc-interrupt-server#s", cpu, thread))

return cpun;

if (\_\_of\_find\_n\_match\_cpu\_property(cpun, "reg", cpu, thread))

return cpun;

}

return NULL;

}

EXPORT\_SYMBOL(of\_get\_cpu\_node);

/\*\* Checks if the given "compat" string matches one of the strings in

\* the device's "compatible" property

\*/

static int \_\_of\_device\_is\_compatible(const struct device\_node \*device,

const char \*compat)

{

const char\* cp;

int cplen, l;

cp = \_\_of\_get\_property(device, "compatible", &cplen);

if (cp == NULL)

return 0;

while (cplen > 0) {

if (of\_compat\_cmp(cp, compat, strlen(compat)) == 0)

return 1;

l = strlen(cp) + 1;

cp += l;

cplen -= l;

}

return 0;

}

/\*\* Checks if the given "compat" string matches one of the strings in

\* the device's "compatible" property

\*/

int of\_device\_is\_compatible(const struct device\_node \*device,

const char \*compat)

{

unsigned long flags;

int res;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

res = \_\_of\_device\_is\_compatible(device, compat);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return res;

}

EXPORT\_SYMBOL(of\_device\_is\_compatible);

/\*\*

\* of\_machine\_is\_compatible - Test root of device tree for a given compatible value

\* @compat: compatible string to look for in root node's compatible property.

\*

\* Returns true if the root node has the given value in its

\* compatible property.

\*/

int of\_machine\_is\_compatible(const char \*compat)

{

struct device\_node \*root;

int rc = 0;

root = of\_find\_node\_by\_path("/");

if (root) {

rc = of\_device\_is\_compatible(root, compat);

of\_node\_put(root);

}

return rc;

}

EXPORT\_SYMBOL(of\_machine\_is\_compatible);

/\*\*

\* \_\_of\_device\_is\_available - check if a device is available for use

\*

\* @device: Node to check for availability, with locks already held

\*

\* Returns 1 if the status property is absent or set to "okay" or "ok",

\* 0 otherwise

\*/

static int \_\_of\_device\_is\_available(const struct device\_node \*device)

{

const char \*status;

int statlen;

status = \_\_of\_get\_property(device, "status", &statlen);

if (status == NULL)

return 1;

if (statlen > 0) {

if (!strcmp(status, "okay") || !strcmp(status, "ok"))

return 1;

}

return 0;

}

/\*\*

\* of\_device\_is\_available - check if a device is available for use

\*

\* @device: Node to check for availability

\*

\* Returns 1 if the status property is absent or set to "okay" or "ok",

\* 0 otherwise

\*/

int of\_device\_is\_available(const struct device\_node \*device)

{

unsigned long flags;

int res;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

res = \_\_of\_device\_is\_available(device);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return res;

}

EXPORT\_SYMBOL(of\_device\_is\_available);

/\*\*

\* of\_get\_parent - Get a node's parent if any

\* @node: Node to get parent

\*

\* Returns a node pointer with refcount incremented, use

\* of\_node\_put() on it when done.

\*/

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

{

struct device\_node \*np;

unsigned long flags;

if (!node)

return NULL;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

np = of\_node\_get(node->parent);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return np;

}

EXPORT\_SYMBOL(of\_get\_parent);

/\*\*

\* of\_get\_next\_parent - Iterate to a node's parent

\* @node: Node to get parent of

\*

\* This is like of\_get\_parent() except that it drops the

\* refcount on the passed node, making it suitable for iterating

\* through a node's parents.

\*

\* Returns a node pointer with refcount incremented, use

\* of\_node\_put() on it when done.

\*/

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

{

struct device\_node \*parent;

unsigned long flags;

if (!node)

return NULL;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

parent = of\_node\_get(node->parent);

of\_node\_put(node);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return parent;

}

EXPORT\_SYMBOL(of\_get\_next\_parent);

/\*\*

\* of\_get\_next\_child - Iterate a node childs

\* @node: parent node

\* @prev: previous child of the parent node, or NULL to get first

\*

\* Returns a node pointer with refcount incremented, use

\* of\_node\_put() on it when done.

\*/

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

struct device\_node \*prev)

{

struct device\_node \*next;

unsigned long flags;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

next = prev ? prev->sibling : node->child;

for (; next; next = next->sibling)

if (of\_node\_get(next))

break;

of\_node\_put(prev);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return next;

}

EXPORT\_SYMBOL(of\_get\_next\_child);

/\*\*

\* of\_get\_next\_available\_child - Find the next available child node

\* @node: parent node

\* @prev: previous child of the parent node, or NULL to get first

\*

\* This function is like of\_get\_next\_child(), except that it

\* automatically skips any disabled nodes (i.e. status = "disabled").

\*/

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

struct device\_node \*prev)

{

struct device\_node \*next;

unsigned long flags;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

next = prev ? prev->sibling : node->child;

for (; next; next = next->sibling) {

if (!\_\_of\_device\_is\_available(next))

continue;

if (of\_node\_get(next))

break;

}

of\_node\_put(prev);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return next;

}

EXPORT\_SYMBOL(of\_get\_next\_available\_child);

/\*\*

\* of\_get\_child\_by\_name - Find the child node by name for a given parent

\* @node: parent node

\* @name: child name to look for.

\*

\* This function looks for child node for given matching name

\*

\* Returns a node pointer if found, with refcount incremented, use

\* of\_node\_put() on it when done.

\* Returns NULL if node is not found.

\*/

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

const char \*name)

{

struct device\_node \*child;

for\_each\_child\_of\_node(node, child)

if (child->name && (of\_node\_cmp(child->name, name) == 0))

break;

return child;

}

EXPORT\_SYMBOL(of\_get\_child\_by\_name);

/\*\*

\* of\_find\_node\_by\_path - Find a node matching a full OF path

\* @path: The full path to match

\*

\* Returns a node pointer with refcount incremented, use

\* of\_node\_put() on it when done.

\*/

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

{

struct device\_node \*np = of\_allnodes;

unsigned long flags;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

for (; np; np = np->allnext) {

if (np->full\_name && (of\_node\_cmp(np->full\_name, path) == 0)

&& of\_node\_get(np))

break;

}

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return np;

}

EXPORT\_SYMBOL(of\_find\_node\_by\_path);

/\*\*

\* of\_find\_node\_by\_name - Find a node by its "name" property

\* @from: The node to start searching from or NULL, the node

\* you pass will not be searched, only the next one

\* will; typically, you pass what the previous call

\* returned. of\_node\_put() will be called on it

\* @name: The name string to match against

\*

\* Returns a node pointer with refcount incremented, use

\* of\_node\_put() on it when done.

\*/

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

const char \*name)

{

struct device\_node \*np;

unsigned long flags;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

np = from ? from->allnext : of\_allnodes;

for (; np; np = np->allnext)

if (np->name && (of\_node\_cmp(np->name, name) == 0)

&& of\_node\_get(np))

break;

of\_node\_put(from);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return np;

}

EXPORT\_SYMBOL(of\_find\_node\_by\_name);

/\*\*

\* of\_find\_node\_by\_type - Find a node by its "device\_type" property

\* @from: The node to start searching from, or NULL to start searching

\* the entire device tree. The node you pass will not be

\* searched, only the next one will; typically, you pass

\* what the previous call returned. of\_node\_put() will be

\* called on from for you.

\* @type: The type string to match against

\*

\* Returns a node pointer with refcount incremented, use

\* of\_node\_put() on it when done.

\*/

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

const char \*type)

{

struct device\_node \*np;

unsigned long flags;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

np = from ? from->allnext : of\_allnodes;

for (; np; np = np->allnext)

if (np->type && (of\_node\_cmp(np->type, type) == 0)

&& of\_node\_get(np))

break;

of\_node\_put(from);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return np;

}

EXPORT\_SYMBOL(of\_find\_node\_by\_type);

/\*\*

\* of\_find\_compatible\_node - Find a node based on type and one of the

\* tokens in its "compatible" property

\* @from: The node to start searching from or NULL, the node

\* you pass will not be searched, only the next one

\* will; typically, you pass what the previous call

\* returned. of\_node\_put() will be called on it

\* @type: The type string to match "device\_type" or NULL to ignore

\* @compatible: The string to match to one of the tokens in the device

\* "compatible" list.

\*

\* Returns a node pointer with refcount incremented, use

\* of\_node\_put() on it when done.

\*/

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

const char \*type, const char \*compatible)

{

struct device\_node \*np;

unsigned long flags;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

np = from ? from->allnext : of\_allnodes;

for (; np; np = np->allnext) {

if (type

&& !(np->type && (of\_node\_cmp(np->type, type) == 0)))

continue;

if (\_\_of\_device\_is\_compatible(np, compatible) &&

of\_node\_get(np))

break;

}

of\_node\_put(from);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return np;

}

EXPORT\_SYMBOL(of\_find\_compatible\_node);

/\*\*

\* of\_find\_node\_with\_property - Find a node which has a property with

\* the given name.

\* @from: The node to start searching from or NULL, the node

\* you pass will not be searched, only the next one

\* will; typically, you pass what the previous call

\* returned. of\_node\_put() will be called on it

\* @prop\_name: The name of the property to look for.

\*

\* Returns a node pointer with refcount incremented, use

\* of\_node\_put() on it when done.

\*/

struct device\_node \*of\_find\_node\_with\_property(struct device\_node \*from,

const char \*prop\_name)

{

struct device\_node \*np;

struct property \*pp;

unsigned long flags;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

np = from ? from->allnext : of\_allnodes;

for (; np; np = np->allnext) {

for (pp = np->properties; pp; pp = pp->next) {

if (of\_prop\_cmp(pp->name, prop\_name) == 0) {

of\_node\_get(np);

goto out;

}

}

}

out:

of\_node\_put(from);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return np;

}

EXPORT\_SYMBOL(of\_find\_node\_with\_property);

static

const struct of\_device\_id \*\_\_of\_match\_node(const struct of\_device\_id \*matches,

const struct device\_node \*node)

{

if (!matches)

return NULL;

while (matches->name[0] || matches->type[0] || matches->compatible[0]) {

int match = 1;

if (matches->name[0])

match &= node->name

&& !strcmp(matches->name, node->name);

if (matches->type[0])

match &= node->type

&& !strcmp(matches->type, node->type);

if (matches->compatible[0])

match &= \_\_of\_device\_is\_compatible(node,

matches->compatible);

if (match)

return matches;

matches++;

}

return NULL;

}

/\*\*

\* of\_match\_node - Tell if an device\_node has a matching of\_match structure

\* @matches: array of of device match structures to search in

\* @node: the of device structure to match against

\*

\* Low level utility function used by device matching.

\*/

const struct of\_device\_id \*of\_match\_node(const struct of\_device\_id \*matches,

const struct device\_node \*node)

{

const struct of\_device\_id \*match;

unsigned long flags;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

match = \_\_of\_match\_node(matches, node);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return match;

}

EXPORT\_SYMBOL(of\_match\_node);

/\*\*

\* of\_find\_matching\_node\_and\_match - Find a node based on an of\_device\_id

\* match table.

\* @from: The node to start searching from or NULL, the node

\* you pass will not be searched, only the next one

\* will; typically, you pass what the previous call

\* returned. of\_node\_put() will be called on it

\* @matches: array of of device match structures to search in

\* @match Updated to point at the matches entry which matched

\*

\* Returns a node pointer with refcount incremented, use

\* of\_node\_put() on it when done.

\*/

struct device\_node \*of\_find\_matching\_node\_and\_match(struct device\_node \*from,

const struct of\_device\_id \*matches,

const struct of\_device\_id \*\*match)

{

struct device\_node \*np;

const struct of\_device\_id \*m;

unsigned long flags;

if (match)

\*match = NULL;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

np = from ? from->allnext : of\_allnodes;

for (; np; np = np->allnext) {

m = \_\_of\_match\_node(matches, np);

if (m && of\_node\_get(np)) {

if (match)

\*match = m;

break;

}

}

of\_node\_put(from);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return np;

}

EXPORT\_SYMBOL(of\_find\_matching\_node\_and\_match);

/\*\*

\* of\_modalias\_node - Lookup appropriate modalias for a device node

\* @node: pointer to a device tree node

\* @modalias: Pointer to buffer that modalias value will be copied into

\* @len: Length of modalias value

\*

\* Based on the value of the compatible property, this routine will attempt

\* to choose an appropriate modalias value for a particular device tree node.

\* It does this by stripping the manufacturer prefix (as delimited by a ',')

\* from the first entry in the compatible list property.

\*

\* This routine returns 0 on success, <0 on failure.

\*/

int of\_modalias\_node(struct device\_node \*node, char \*modalias, int len)

{

const char \*compatible, \*p;

int cplen;

compatible = of\_get\_property(node, "compatible", &cplen);

if (!compatible || strlen(compatible) > cplen)

return -ENODEV;

p = strchr(compatible, ',');

strlcpy(modalias, p ? p + 1 : compatible, len);

return 0;

}

EXPORT\_SYMBOL\_GPL(of\_modalias\_node);

/\*\*

\* of\_find\_node\_by\_phandle - Find a node given a phandle

\* @handle: phandle of the node to find

\*

\* Returns a node pointer with refcount incremented, use

\* of\_node\_put() on it when done.

\*/

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

{

struct device\_node \*np;

unsigned long flags;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

for (np = of\_allnodes; np; np = np->allnext)

if (np->phandle == handle)

break;

of\_node\_get(np);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return np;

}

EXPORT\_SYMBOL(of\_find\_node\_by\_phandle);

/\*\*

\* of\_find\_property\_value\_of\_size

\*

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

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

\* @len: requested length of property value

\*

\* Search for a property in a device node and valid the requested size.

\* Returns the property value on success, -EINVAL if the property does not

\* exist, -ENODATA if property does not have a value, and -EOVERFLOW if the

\* property data isn't large enough.

\*

\*/

static void \*of\_find\_property\_value\_of\_size(const struct device\_node \*np,

const char \*propname, u32 len)

{

struct property \*prop = of\_find\_property(np, propname, NULL);

if (!prop)

return ERR\_PTR(-EINVAL);

if (!prop->value)

return ERR\_PTR(-ENODATA);

if (len > prop->length)

return ERR\_PTR(-EOVERFLOW);

return prop->value;

}

/\*\*

\* of\_property\_read\_u32\_index - Find and read a u32 from a multi-value property.

\*

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

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

\* @index: index of the u32 in the list of values

\* @out\_value: pointer to return value, modified only if no error.

\*

\* Search for a property in a device node and read nth 32-bit value from

\* it. Returns 0 on success, -EINVAL if the property does not exist,

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

\* property data isn't large enough.

\*

\* The out\_value is modified only if a valid u32 value can be decoded.

\*/

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

const char \*propname,

u32 index, u32 \*out\_value)

{

const u32 \*val = of\_find\_property\_value\_of\_size(np, propname,

((index + 1) \* sizeof(\*out\_value)));

if (IS\_ERR(val))

return PTR\_ERR(val);

\*out\_value = be32\_to\_cpup(((\_\_be32 \*)val) + index);

return 0;

}

EXPORT\_SYMBOL\_GPL(of\_property\_read\_u32\_index);

/\*\*

\* of\_property\_read\_u8\_array - Find and read an array of u8 from a property.

\*

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

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

\* @out\_values: pointer to return value, modified only if return value is 0.

\* @sz: number of array elements to read

\*

\* Search for a property in a device node and read 8-bit value(s) from

\* it. Returns 0 on success, -EINVAL if the property does not exist,

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

\* property data isn't large enough.

\*

\* dts entry of array should be like:

\* property = /bits/ 8 <0x50 0x60 0x70>;

\*

\* The out\_values is modified only if a valid u8 value can be decoded.

\*/

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

const char \*propname, u8 \*out\_values, size\_t sz)

{

const u8 \*val = of\_find\_property\_value\_of\_size(np, propname,

(sz \* sizeof(\*out\_values)));

if (IS\_ERR(val))

return PTR\_ERR(val);

while (sz--)

\*out\_values++ = \*val++;

return 0;

}

EXPORT\_SYMBOL\_GPL(of\_property\_read\_u8\_array);

/\*\*

\* of\_property\_read\_u16\_array - Find and read an array of u16 from a property.

\*

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

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

\* @out\_values: pointer to return value, modified only if return value is 0.

\* @sz: number of array elements to read

\*

\* Search for a property in a device node and read 16-bit value(s) from

\* it. Returns 0 on success, -EINVAL if the property does not exist,

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

\* property data isn't large enough.

\*

\* dts entry of array should be like:

\* property = /bits/ 16 <0x5000 0x6000 0x7000>;

\*

\* The out\_values is modified only if a valid u16 value can be decoded.

\*/

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

const char \*propname, u16 \*out\_values, size\_t sz)

{

const \_\_be16 \*val = of\_find\_property\_value\_of\_size(np, propname,

(sz \* sizeof(\*out\_values)));

if (IS\_ERR(val))

return PTR\_ERR(val);

while (sz--)

\*out\_values++ = be16\_to\_cpup(val++);

return 0;

}

EXPORT\_SYMBOL\_GPL(of\_property\_read\_u16\_array);

/\*\*

\* of\_property\_read\_u32\_array - Find and read an array of 32 bit integers

\* from a property.

\*

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

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

\* @out\_values: pointer to return value, modified only if return value is 0.

\* @sz: number of array elements to read

\*

\* Search for a property in a device node and read 32-bit value(s) from

\* it. Returns 0 on success, -EINVAL if the property does not exist,

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

\* property data isn't large enough.

\*

\* The out\_values is modified only if a valid u32 value can be decoded.

\*/

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

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

size\_t sz)

{

const \_\_be32 \*val = of\_find\_property\_value\_of\_size(np, propname,

(sz \* sizeof(\*out\_values)));

if (IS\_ERR(val))

return PTR\_ERR(val);

while (sz--)

\*out\_values++ = be32\_to\_cpup(val++);

return 0;

}

EXPORT\_SYMBOL\_GPL(of\_property\_read\_u32\_array);

/\*\*

\* of\_property\_read\_u64 - Find and read a 64 bit integer from a property

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

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

\* @out\_value: pointer to return value, modified only if return value is 0.

\*

\* Search for a property in a device node and read a 64-bit value from

\* it. Returns 0 on success, -EINVAL if the property does not exist,

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

\* property data isn't large enough.

\*

\* The out\_value is modified only if a valid u64 value can be decoded.

\*/

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

u64 \*out\_value)

{

const \_\_be32 \*val = of\_find\_property\_value\_of\_size(np, propname,

sizeof(\*out\_value));

if (IS\_ERR(val))

return PTR\_ERR(val);

\*out\_value = of\_read\_number(val, 2);

return 0;

}

EXPORT\_SYMBOL\_GPL(of\_property\_read\_u64);

/\*\*

\* of\_property\_read\_string - Find and read a string from a property

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

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

\* @out\_string: pointer to null terminated return string, modified only if

\* return value is 0.

\*

\* Search for a property in a device tree node and retrieve a null

\* terminated string value (pointer to data, not a copy). Returns 0 on

\* success, -EINVAL if the property does not exist, -ENODATA if property

\* does not have a value, and -EILSEQ if the string is not null-terminated

\* within the length of the property data.

\*

\* The out\_string pointer is modified only if a valid string can be decoded.

\*/

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

const char \*\*out\_string)

{

struct property \*prop = of\_find\_property(np, propname, NULL);

if (!prop)

return -EINVAL;

if (!prop->value)

return -ENODATA;

if (strnlen(prop->value, prop->length) >= prop->length)

return -EILSEQ;

\*out\_string = prop->value;

return 0;

}

EXPORT\_SYMBOL\_GPL(of\_property\_read\_string);

/\*\*

\* of\_property\_read\_string\_index - Find and read a string from a multiple

\* strings property.

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

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

\* @index: index of the string in the list of strings

\* @out\_string: pointer to null terminated return string, modified only if

\* return value is 0.

\*

\* Search for a property in a device tree node and retrieve a null

\* terminated string value (pointer to data, not a copy) in the list of strings

\* contained in that property.

\* Returns 0 on success, -EINVAL if the property does not exist, -ENODATA if

\* property does not have a value, and -EILSEQ if the string is not

\* null-terminated within the length of the property data.

\*

\* The out\_string pointer is modified only if a valid string can be decoded.

\*/

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

int index, const char \*\*output)

{

struct property \*prop = of\_find\_property(np, propname, NULL);

int i = 0;

size\_t l = 0, total = 0;

const char \*p;

if (!prop)

return -EINVAL;

if (!prop->value)

return -ENODATA;

if (strnlen(prop->value, prop->length) >= prop->length)

return -EILSEQ;

p = prop->value;

for (i = 0; total < prop->length; total += l, p += l) {

l = strlen(p) + 1;

if (i++ == index) {

\*output = p;

return 0;

}

}

return -ENODATA;

}

EXPORT\_SYMBOL\_GPL(of\_property\_read\_string\_index);

/\*\*

\* of\_property\_match\_string() - Find string in a list and return index

\* @np: pointer to node containing string list property

\* @propname: string list property name

\* @string: pointer to string to search for in string list

\*

\* This function searches a string list property and returns the index

\* of a specific string value.

\*/

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

const char \*string)

{

struct property \*prop = of\_find\_property(np, propname, NULL);

size\_t l;

int i;

const char \*p, \*end;

if (!prop)

return -EINVAL;

if (!prop->value)

return -ENODATA;

p = prop->value;

end = p + prop->length;

for (i = 0; p < end; i++, p += l) {

l = strlen(p) + 1;

if (p + l > end)

return -EILSEQ;

pr\_debug("comparing %s with %s\n", string, p);

if (strcmp(string, p) == 0)

return i; /\* Found it; return index \*/

}

return -ENODATA;

}

EXPORT\_SYMBOL\_GPL(of\_property\_match\_string);

/\*\*

\* of\_property\_count\_strings - Find and return the number of strings from a

\* multiple strings property.

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

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

\*

\* Search for a property in a device tree node and retrieve the number of null

\* terminated string contain in it. Returns the number of strings on

\* success, -EINVAL if the property does not exist, -ENODATA if property

\* does not have a value, and -EILSEQ if the string is not null-terminated

\* within the length of the property data.

\*/

int of\_property\_count\_strings(struct device\_node \*np, const char \*propname)

{

struct property \*prop = of\_find\_property(np, propname, NULL);

int i = 0;

size\_t l = 0, total = 0;

const char \*p;

if (!prop)

return -EINVAL;

if (!prop->value)

return -ENODATA;

if (strnlen(prop->value, prop->length) >= prop->length)

return -EILSEQ;

p = prop->value;

for (i = 0; total < prop->length; total += l, p += l, i++)

l = strlen(p) + 1;

return i;

}

EXPORT\_SYMBOL\_GPL(of\_property\_count\_strings);

static int \_\_of\_parse\_phandle\_with\_args(const struct device\_node \*np,

const char \*list\_name,

const char \*cells\_name,

int cell\_count, int index,

struct of\_phandle\_args \*out\_args)

{

const \_\_be32 \*list, \*list\_end;

int rc = 0, size, cur\_index = 0;

uint32\_t count = 0;

struct device\_node \*node = NULL;

phandle phandle;

/\* Retrieve the phandle list property \*/

list = of\_get\_property(np, list\_name, &size);

if (!list)

return -ENOENT;

list\_end = list + size / sizeof(\*list);

/\* Loop over the phandles until all the requested entry is found \*/

while (list < list\_end) {

rc = -EINVAL;

count = 0;

/\*

\* If phandle is 0, then it is an empty entry with no

\* arguments. Skip forward to the next entry.

\*/

phandle = be32\_to\_cpup(list++);

if (phandle) {

/\*

\* Find the provider node and parse the #\*-cells

\* property to determine the argument length.

\*

\* This is not needed if the cell count is hard-coded

\* (i.e. cells\_name not set, but cell\_count is set),

\* except when we're going to return the found node

\* below.

\*/

if (cells\_name || cur\_index == index) {

node = of\_find\_node\_by\_phandle(phandle);

if (!node) {

pr\_err("%s: could not find phandle\n",

np->full\_name);

goto err;

}

}

if (cells\_name) {

if (of\_property\_read\_u32(node, cells\_name,

&count)) {

pr\_err("%s: could not get %s for %s\n",

np->full\_name, cells\_name,

node->full\_name);

goto err;

}

} else {

count = cell\_count;

}

/\*

\* Make sure that the arguments actually fit in the

\* remaining property data length

\*/

if (list + count > list\_end) {

pr\_err("%s: arguments longer than property\n",

np->full\_name);

goto err;

}

}

/\*

\* All of the error cases above bail out of the loop, so at

\* this point, the parsing is successful. If the requested

\* index matches, then fill the out\_args structure and return,

\* or return -ENOENT for an empty entry.

\*/

rc = -ENOENT;

if (cur\_index == index) {

if (!phandle)

goto err;

if (out\_args) {

int i;

if (WARN\_ON(count > MAX\_PHANDLE\_ARGS))

count = MAX\_PHANDLE\_ARGS;

out\_args->np = node;

out\_args->args\_count = count;

for (i = 0; i < count; i++)

out\_args->args[i] = be32\_to\_cpup(list++);

} else {

of\_node\_put(node);

}

/\* Found it! return success \*/

return 0;

}

of\_node\_put(node);

node = NULL;

list += count;

cur\_index++;

}

/\*

\* Unlock node before returning result; will be one of:

\* -ENOENT : index is for empty phandle

\* -EINVAL : parsing error on data

\* [1..n] : Number of phandle (count mode; when index = -1)

\*/

rc = index < 0 ? cur\_index : -ENOENT;

err:

if (node)

of\_node\_put(node);

return rc;

}

/\*\*

\* of\_parse\_phandle - Resolve a phandle property to a device\_node pointer

\* @np: Pointer to device node holding phandle property

\* @phandle\_name: Name of property holding a phandle value

\* @index: For properties holding a table of phandles, this is the index into

\* the table

\*

\* Returns the device\_node pointer with refcount incremented. Use

\* of\_node\_put() on it when done.

\*/

struct device\_node \*of\_parse\_phandle(const struct device\_node \*np,

const char \*phandle\_name, int index)

{

struct of\_phandle\_args args;

if (index < 0)

return NULL;

if (\_\_of\_parse\_phandle\_with\_args(np, phandle\_name, NULL, 0,

index, &args))

return NULL;

return args.np;

}

EXPORT\_SYMBOL(of\_parse\_phandle);

/\*\*

\* of\_parse\_phandle\_with\_args() - Find a node pointed by phandle in a list

\* @np: pointer to a device tree node containing a list

\* @list\_name: property name that contains a list

\* @cells\_name: property name that specifies phandles' arguments count

\* @index: index of a phandle to parse out

\* @out\_args: optional pointer to output arguments structure (will be filled)

\*

\* This function is useful to parse lists of phandles and their arguments.

\* Returns 0 on success and fills out\_args, on error returns appropriate

\* errno value.

\*

\* Caller is responsible to call of\_node\_put() on the returned out\_args->node

\* pointer.

\*

\* Example:

\*

\* phandle1: node1 {

\* #list-cells = <2>;

\* }

\*

\* phandle2: node2 {

\* #list-cells = <1>;

\* }

\*

\* node3 {

\* list = <&phandle1 1 2 &phandle2 3>;

\* }

\*

\* To get a device\_node of the `node2' node you may call this:

\* of\_parse\_phandle\_with\_args(node3, "list", "#list-cells", 1, &args);

\*/

int of\_parse\_phandle\_with\_args(const struct device\_node \*np, const char \*list\_name,

const char \*cells\_name, int index,

struct of\_phandle\_args \*out\_args)

{

if (index < 0)

return -EINVAL;

return \_\_of\_parse\_phandle\_with\_args(np, list\_name, cells\_name, 0,

index, out\_args);

}

EXPORT\_SYMBOL(of\_parse\_phandle\_with\_args);

/\*\*

\* of\_parse\_phandle\_with\_fixed\_args() - Find a node pointed by phandle in a list

\* @np: pointer to a device tree node containing a list

\* @list\_name: property name that contains a list

\* @cell\_count: number of argument cells following the phandle

\* @index: index of a phandle to parse out

\* @out\_args: optional pointer to output arguments structure (will be filled)

\*

\* This function is useful to parse lists of phandles and their arguments.

\* Returns 0 on success and fills out\_args, on error returns appropriate

\* errno value.

\*

\* Caller is responsible to call of\_node\_put() on the returned out\_args->node

\* pointer.

\*

\* Example:

\*

\* phandle1: node1 {

\* }

\*

\* phandle2: node2 {

\* }

\*

\* node3 {

\* list = <&phandle1 0 2 &phandle2 2 3>;

\* }

\*

\* To get a device\_node of the `node2' node you may call this:

\* of\_parse\_phandle\_with\_fixed\_args(node3, "list", 2, 1, &args);

\*/

int of\_parse\_phandle\_with\_fixed\_args(const struct device\_node \*np,

const char \*list\_name, int cell\_count,

int index, struct of\_phandle\_args \*out\_args)

{

if (index < 0)

return -EINVAL;

return \_\_of\_parse\_phandle\_with\_args(np, list\_name, NULL, cell\_count,

index, out\_args);

}

EXPORT\_SYMBOL(of\_parse\_phandle\_with\_fixed\_args);

/\*\*

\* of\_count\_phandle\_with\_args() - Find the number of phandles references in a property

\* @np: pointer to a device tree node containing a list

\* @list\_name: property name that contains a list

\* @cells\_name: property name that specifies phandles' arguments count

\*

\* Returns the number of phandle + argument tuples within a property. It

\* is a typical pattern to encode a list of phandle and variable

\* arguments into a single property. The number of arguments is encoded

\* by a property in the phandle-target node. For example, a gpios

\* property would contain a list of GPIO specifies consisting of a

\* phandle and 1 or more arguments. The number of arguments are

\* determined by the #gpio-cells property in the node pointed to by the

\* phandle.

\*/

int of\_count\_phandle\_with\_args(const struct device\_node \*np, const char \*list\_name,

const char \*cells\_name)

{

return \_\_of\_parse\_phandle\_with\_args(np, list\_name, cells\_name, 0, -1,

NULL);

}

EXPORT\_SYMBOL(of\_count\_phandle\_with\_args);

#if defined(CONFIG\_OF\_DYNAMIC)

static int of\_property\_notify(int action, struct device\_node \*np,

struct property \*prop)

{

struct of\_prop\_reconfig pr;

pr.dn = np;

pr.prop = prop;

return of\_reconfig\_notify(action, &pr);

}

#else

static int of\_property\_notify(int action, struct device\_node \*np,

struct property \*prop)

{

return 0;

}

#endif

/\*\*

\* of\_add\_property - Add a property to a node

\*/

int of\_add\_property(struct device\_node \*np, struct property \*prop)

{

struct property \*\*next;

unsigned long flags;

int rc;

rc = of\_property\_notify(OF\_RECONFIG\_ADD\_PROPERTY, np, prop);

if (rc)

return rc;

prop->next = NULL;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

next = &np->properties;

while (\*next) {

if (strcmp(prop->name, (\*next)->name) == 0) {

/\* duplicate ! don't insert it \*/

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return -1;

}

next = &(\*next)->next;

}

\*next = prop;

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

#ifdef CONFIG\_PROC\_DEVICETREE

/\* try to add to proc as well if it was initialized \*/

if (np->pde)

proc\_device\_tree\_add\_prop(np->pde, prop);

#endif /\* CONFIG\_PROC\_DEVICETREE \*/

return 0;

}

/\*\*

\* of\_remove\_property - Remove a property from a node.

\*

\* Note that we don't actually remove it, since we have given out

\* who-knows-how-many pointers to the data using get-property.

\* Instead we just move the property to the "dead properties"

\* list, so it won't be found any more.

\*/

int of\_remove\_property(struct device\_node \*np, struct property \*prop)

{

struct property \*\*next;

unsigned long flags;

int found = 0;

int rc;

rc = of\_property\_notify(OF\_RECONFIG\_REMOVE\_PROPERTY, np, prop);

if (rc)

return rc;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

next = &np->properties;

while (\*next) {

if (\*next == prop) {

/\* found the node \*/

\*next = prop->next;

prop->next = np->deadprops;

np->deadprops = prop;

found = 1;

break;

}

next = &(\*next)->next;

}

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

if (!found)

return -ENODEV;

#ifdef CONFIG\_PROC\_DEVICETREE

/\* try to remove the proc node as well \*/

if (np->pde)

proc\_device\_tree\_remove\_prop(np->pde, prop);

#endif /\* CONFIG\_PROC\_DEVICETREE \*/

return 0;

}

/\*

\* of\_update\_property - Update a property in a node, if the property does

\* not exist, add it.

\*

\* Note that we don't actually remove it, since we have given out

\* who-knows-how-many pointers to the data using get-property.

\* Instead we just move the property to the "dead properties" list,

\* and add the new property to the property list

\*/

int of\_update\_property(struct device\_node \*np, struct property \*newprop)

{

struct property \*\*next, \*oldprop;

unsigned long flags;

int rc, found = 0;

rc = of\_property\_notify(OF\_RECONFIG\_UPDATE\_PROPERTY, np, newprop);

if (rc)

return rc;

if (!newprop->name)

return -EINVAL;

oldprop = of\_find\_property(np, newprop->name, NULL);

if (!oldprop)

return of\_add\_property(np, newprop);

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

next = &np->properties;

while (\*next) {

if (\*next == oldprop) {

/\* found the node \*/

newprop->next = oldprop->next;

\*next = newprop;

oldprop->next = np->deadprops;

np->deadprops = oldprop;

found = 1;

break;

}

next = &(\*next)->next;

}

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

if (!found)

return -ENODEV;

#ifdef CONFIG\_PROC\_DEVICETREE

/\* try to add to proc as well if it was initialized \*/

if (np->pde)

proc\_device\_tree\_update\_prop(np->pde, newprop, oldprop);

#endif /\* CONFIG\_PROC\_DEVICETREE \*/

return 0;

}

#if defined(CONFIG\_OF\_DYNAMIC)

/\*

\* Support for dynamic device trees.

\*

\* On some platforms, the device tree can be manipulated at runtime.

\* The routines in this section support adding, removing and changing

\* device tree nodes.

\*/

static BLOCKING\_NOTIFIER\_HEAD(of\_reconfig\_chain);

int of\_reconfig\_notifier\_register(struct notifier\_block \*nb)

{

return blocking\_notifier\_chain\_register(&of\_reconfig\_chain, nb);

}

EXPORT\_SYMBOL\_GPL(of\_reconfig\_notifier\_register);

int of\_reconfig\_notifier\_unregister(struct notifier\_block \*nb)

{

return blocking\_notifier\_chain\_unregister(&of\_reconfig\_chain, nb);

}

EXPORT\_SYMBOL\_GPL(of\_reconfig\_notifier\_unregister);

int of\_reconfig\_notify(unsigned long action, void \*p)

{

int rc;

rc = blocking\_notifier\_call\_chain(&of\_reconfig\_chain, action, p);

return notifier\_to\_errno(rc);

}

#ifdef CONFIG\_PROC\_DEVICETREE

static void of\_add\_proc\_dt\_entry(struct device\_node \*dn)

{

struct proc\_dir\_entry \*ent;

ent = proc\_mkdir(strrchr(dn->full\_name, '/') + 1, dn->parent->pde);

if (ent)

proc\_device\_tree\_add\_node(dn, ent);

}

#else

static void of\_add\_proc\_dt\_entry(struct device\_node \*dn)

{

return;

}

#endif

/\*\*

\* of\_attach\_node - Plug a device node into the tree and global list.

\*/

int of\_attach\_node(struct device\_node \*np)

{

unsigned long flags;

int rc;

rc = of\_reconfig\_notify(OF\_RECONFIG\_ATTACH\_NODE, np);

if (rc)

return rc;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

np->sibling = np->parent->child;

np->allnext = of\_allnodes;

np->parent->child = np;

of\_allnodes = np;

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

of\_add\_proc\_dt\_entry(np);

return 0;

}

#ifdef CONFIG\_PROC\_DEVICETREE

static void of\_remove\_proc\_dt\_entry(struct device\_node \*dn)

{

proc\_remove(dn->pde);

}

#else

static void of\_remove\_proc\_dt\_entry(struct device\_node \*dn)

{

return;

}

#endif

/\*\*

\* of\_detach\_node - "Unplug" a node from the device tree.

\*

\* The caller must hold a reference to the node. The memory associated with

\* the node is not freed until its refcount goes to zero.

\*/

int of\_detach\_node(struct device\_node \*np)

{

struct device\_node \*parent;

unsigned long flags;

int rc = 0;

rc = of\_reconfig\_notify(OF\_RECONFIG\_DETACH\_NODE, np);

if (rc)

return rc;

raw\_spin\_lock\_irqsave(&devtree\_lock, flags);

if (of\_node\_check\_flag(np, OF\_DETACHED)) {

/\* someone already detached it \*/

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return rc;

}

parent = np->parent;

if (!parent) {

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

return rc;

}

if (of\_allnodes == np)

of\_allnodes = np->allnext;

else {

struct device\_node \*prev;

for (prev = of\_allnodes;

prev->allnext != np;

prev = prev->allnext)

;

prev->allnext = np->allnext;

}

if (parent->child == np)

parent->child = np->sibling;

else {

struct device\_node \*prevsib;

for (prevsib = np->parent->child;

prevsib->sibling != np;

prevsib = prevsib->sibling)

;

prevsib->sibling = np->sibling;

}

of\_node\_set\_flag(np, OF\_DETACHED);

raw\_spin\_unlock\_irqrestore(&devtree\_lock, flags);

of\_remove\_proc\_dt\_entry(np);

return rc;

}

#endif /\* defined(CONFIG\_OF\_DYNAMIC) \*/

static void of\_alias\_add(struct alias\_prop \*ap, struct device\_node \*np,

int id, const char \*stem, int stem\_len)

{

ap->np = np;

ap->id = id;

strncpy(ap->stem, stem, stem\_len);

ap->stem[stem\_len] = 0;

list\_add\_tail(&ap->link, &aliases\_lookup);

pr\_debug("adding DT alias:%s: stem=%s id=%i node=%s\n",

ap->alias, ap->stem, ap->id, of\_node\_full\_name(np));

}

/\*\*

\* of\_alias\_scan - Scan all properties of 'aliases' node

\*

\* The function scans all the properties of 'aliases' node and populate

\* the the global lookup table with the properties. It returns the

\* number of alias\_prop found, or error code in error case.

\*

\* @dt\_alloc: An allocator that provides a virtual address to memory

\* for the resulting tree

\*/

void of\_alias\_scan(void \* (\*dt\_alloc)(u64 size, u64 align))

{

struct property \*pp;

of\_chosen = of\_find\_node\_by\_path("/chosen");

if (of\_chosen == NULL)

of\_chosen = of\_find\_node\_by\_path("/chosen@0");

if (of\_chosen) {

const char \*name;

name = of\_get\_property(of\_chosen, "linux,stdout-path", NULL);

if (name)

of\_stdout = of\_find\_node\_by\_path(name);

}

of\_aliases = of\_find\_node\_by\_path("/aliases");

if (!of\_aliases)

return;

for\_each\_property\_of\_node(of\_aliases, pp) {

const char \*start = pp->name;

const char \*end = start + strlen(start);

struct device\_node \*np;

struct alias\_prop \*ap;

int id, len;

/\* Skip those we do not want to proceed \*/

if (!strcmp(pp->name, "name") ||

!strcmp(pp->name, "phandle") ||

!strcmp(pp->name, "linux,phandle"))

continue;

np = of\_find\_node\_by\_path(pp->value);

if (!np)

continue;

/\* walk the alias backwards to extract the id and work out

\* the 'stem' string \*/

while (isdigit(\*(end-1)) && end > start)

end--;

len = end - start;

if (kstrtoint(end, 10, &id) < 0)

continue;

/\* Allocate an alias\_prop with enough space for the stem \*/

ap = dt\_alloc(sizeof(\*ap) + len + 1, 4);

if (!ap)

continue;

memset(ap, 0, sizeof(\*ap) + len + 1);

ap->alias = start;

of\_alias\_add(ap, np, id, start, len);

}

}

/\*\*

\* of\_alias\_get\_id - Get alias id for the given device\_node

\* @np: Pointer to the given device\_node

\* @stem: Alias stem of the given device\_node

\*

\* The function travels the lookup table to get alias id for the given

\* device\_node and alias stem. It returns the alias id if find it.

\*/

int of\_alias\_get\_id(struct device\_node \*np, const char \*stem)

{

struct alias\_prop \*app;

int id = -ENODEV;

mutex\_lock(&of\_aliases\_mutex);

list\_for\_each\_entry(app, &aliases\_lookup, link) {

if (strcmp(app->stem, stem) != 0)

continue;

if (np == app->np) {

id = app->id;

break;

}

}

mutex\_unlock(&of\_aliases\_mutex);

return id;

}

EXPORT\_SYMBOL\_GPL(of\_alias\_get\_id);

const \_\_be32 \*of\_prop\_next\_u32(struct property \*prop, const \_\_be32 \*cur,

u32 \*pu)

{

const void \*curv = cur;

if (!prop)

return NULL;

if (!cur) {

curv = prop->value;

goto out\_val;

}

curv += sizeof(\*cur);

if (curv >= prop->value + prop->length)

return NULL;

out\_val:

\*pu = be32\_to\_cpup(curv);

return curv;

}

EXPORT\_SYMBOL\_GPL(of\_prop\_next\_u32);

const char \*of\_prop\_next\_string(struct property \*prop, const char \*cur)

{

const void \*curv = cur;

if (!prop)

return NULL;

if (!cur)

return prop->value;

curv += strlen(cur) + 1;

if (curv >= prop->value + prop->length)

return NULL;

return curv;

}

EXPORT\_SYMBOL\_GPL(of\_prop\_next\_string);

/\*\*

\* of\_device\_is\_stdout\_path - check if a device node matches the

\* linux,stdout-path property

\*

\* Check if this device node matches the linux,stdout-path property

\* in the chosen node. return true if yes, false otherwise.

\*/

int of\_device\_is\_stdout\_path(struct device\_node \*dn)

{

if (!of\_stdout)

return false;

return of\_stdout == dn;

}

EXPORT\_SYMBOL\_GPL(of\_device\_is\_stdout\_path);