Android 10 device tree

使用说明文档

1.0

2020.2.28

文档履历

版本号	日期	制/修订人	内容描述
1.0	2020.2.28		

目录

1. 前言

1.1 编写目的

介绍 devicetree 配置、设备驱动如何获取 devicetree 配置信息等内容，让用户明确掌握 devicetree 配置与使用方法。

1.2 适用范围

适用于 A64/H64/R18、B100/G102、A20E/V40/T3/T3A/T3L/R40、H5、A63/A63vr、H6、H616、A100、A133 芯片相关平台。

1.3 相关人员

linux 项目组同事, linux 内核和驱动开发人员。

1.4 相关术语

术语 /缩略语	解释说明
DTS	Device Tree Source File, 设备树源码文件
DTB	Device Tree Blob File, 设备树二进制文件
Sys_config.fex	Allwinner 配置文件
FDT	扁平设备树 FDT 接口

2. 模块介绍

Device tree 是一种描述硬件信息的数据结构，它表现为一颗由电路板上 CPU、总线、设备组成的树，Device tree 由一系列被命名的结点 (node) 和属性 (property) 组成，而结点本身可以包含子结点。所谓属性，就是名字对，也就是成对出现的 name 和 value。在 Device Tree 中，可描述以下信息：

- CPU 数量和类别
- 内存基址和大小
- 总线
- 外设
- 中断控制器
- GPIO 控制器
- clock 控制器

Bootloader 会将这棵树传递给内核，内核可以识别这棵树，并根据它展开出 Linux 内核中的 platform_device、i2c_client、spi_device 等设备，而这些设备用到的内存、IRQ、寄存器等资源，也会通过 DTB 传递给内核，内核会将这些资源绑定给展开的相应设备。因此，对 device tree 的理解，可从以下 5 步进行。

- 用于描述硬件设备信息的文本格式，如 dts 或 dtsi
- 认识 DTC 工具
- Bootloader 如何传入 DTB 的
- 内核如何展开文件，获取硬件设备信息的 (对 Device tree 的解析)
- 设备驱动如何使用

2.1 Device Tree source file

.dts 或.dtsi 文件是一种 ASCII 文本格式文件的 Device Tree 描述，在 ARM linux 中，一个.dts 文件对应一个 ARM 的 machine。 由于一个 SoC 可能对应多个 machine，因此.dts 文件可能需要包含许多共同的部分。Linux 内核为了简化，采用了 C 语言包含头文件的方式，将 SoC 公共的部分或者多个 machine 共同的部分提炼为.dtsi，其他的 machine 对应的.dts 就 include 这个.dtsi 即可。设备树是一个包含结点和属性的简单树状结构，属性就是键 -值对 (名字对)，而结点可以同时包含属性和子结点。例如，以下就是一个简单的.dts 文件的简单树

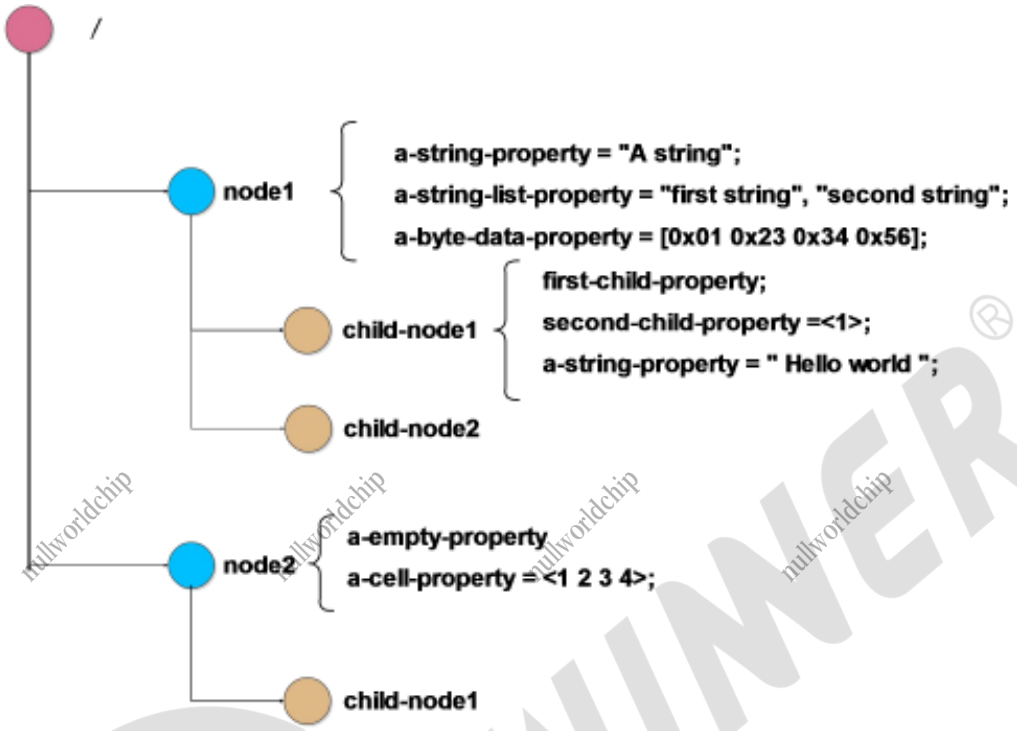

图 1: dts 简单树示例

这棵树显然是没什么用的，因为它并没有描述任何东西，但它确实体现了节点的一些属性：

- 一个单独的根节点：``/"
- 两个子节点：``node1" 和 ``node2"
- 两个 node1 的子节点：``child-node1" 和 ``child-node2"
- 一堆分散在树里的属性

属性是简单的键一值对，它的值可以为空或者包含一个任意字节流。虽然数据类型并没有编码进数据结构，但在设备树源文件中仍有几个基本的数据表示形式。

- 文本字符串（无结束符）可以用双引号表示：a-string-property="hello world"
- 二进制数据用方括号限定
- 不同表示形式的数据可以使用逗号连在一起
- 逗号也可用于创建字符串列表：a-string-list-property="first string","second string"

2.2 Device Tree 结构介绍约定

2.2.1 结点名称 (node names)

规范：device tree 中每个结点的命名必须遵守以下原则：

```
结点名称:
node-name@uint-address
```

原则：

- node-name: 结点名称，小于 31 字符长度的字符串，可以包括下表中的字符。结点的首字符必须是英文字母。通常结点命名应该体现是什么样的设备。

Character	Description
0-9	数字
A-Z	大写字母
a-z	小写字母
,	英文逗号
.	英文句号
_	英文下划线
+	加号
-	减号

- @uint-address: 如果该结点描述的设备有一个地址，则应该加上设备地址 (uint-address)。通常，设备地址就是用来访问该设备的主地址，并且该地址也在结点的 reg 属性中列出。
- 同级结点的命名必须是唯一的，但只要地址不同，多个结点的名称也可以相同，如 (cpu@0 和 cpu@1)。
- 根节点没有 node-name 和 unit-address，它是通过 ``/" 来识别。

实例：

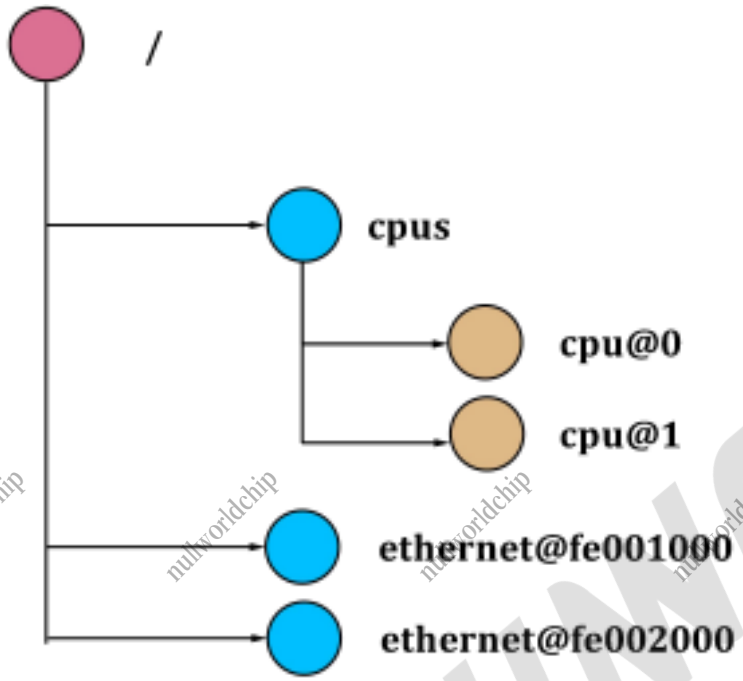

图 2: dts 结点命名规范示例

从例子中，我们可以看到，一个根结点/下有 3 个子结点：结点名称为 cpu 的结点通过地址 0 和 1 来区分，结点名称为 ethernet 的结点，通过地址 fe001000 和 fe002000 来区分。

2.2.2 路径名称

在 device tree 中，唯一识别结点的另一个方法就是指定绝对路径，即给结点指定从根结点到该结点的完整路径。Device tree 中约定了完整路径的表达方式：

```
/node-name-1/node-name-2/.../node-name-N
实例：我们图2 dts结点命名规范示例中cpu@1的完整路径如下
/cpus/cpu@1

注：如果完整路径可以明确表达我们需要确定的结点时，结点之后的地址可以省略
```

2.2.3 属性 (properties)

在 Device Tree 中，结点可以用属性来描述该结点的特征，属性由两部分组成：名称和值。

2.2.3.1 属性名称

属性名称由长度小于 31 的字符串组成。属性名称支持的字符如表 1 所示。属性名称可以分为标准属性名称和非标准属性名称。非标准的属性名称，需要指定一个唯一的前缀，用来识别是哪个公司或机构定义了该属性。例如：

```
fsl,channel-fifo-len 29
ibm,ppc-interrupt-server#s 30
linux,network-index
allwinner,pull = <1>
```

2.2.3.2 属性值

属性值是一个包含属性相关信息的数组，数组可能有 0 个或多个字节。当属性是为了传递真伪信息时，属性值可以为空值。这时，属性值的存在或不存在，就已经足够描述属性的相关信息了。属性值可以采用下表中的形式：

value	Description
	空值
	大端格式的 32bit 整数
	大端格式的 64bit 整数
	字符串，包含结束符
	跟特定属性相关
	一个值，提供引用结点方法
	由一系列值串联在一起

3. 配置方法

3.1 设备树文件关系

- SoC 级配置文件：定义了 SoC 级配置，如设备时钟、中断等资源，不建议修改，如 sun50iw1p1.dtsi。
- board 级配置文件：定义了板级配置，包含一些板级差异信息，推荐差异化修改放在该文件，如 board.dts。
- 设备树文件的配置是该 SoC 所有方案的通用配置，对于 ARM64 cpu 而言，设备树 (以 sun50iw10p1 为例) 的路径为：kernel/linux-4.9/arch/arm64/boot/dts/sunxi/sun50iw10p1-clk.dtsi。

- 设备树文件的配置是该 SoC 所有方案的通用配置，对于 ARM32 cpu 而言，设备树 (以 sun8iw16p1 为例) 的路径为：kernel/linux-4.9/arch/arm/boot/dts/sun8iw16p1-clk.dtsi。

- 板级设备树 (board.dts) 路径：/device/config/chips/a100/configs/b3/board.dts

device tree 的源码结构关系如下：

```
board.dts
  |--------sun50iw10p1.dtsi
            |------sun50iw10p1-pinctrl.dtsi
            |------sun50iw10p1-clk.dtsi
```

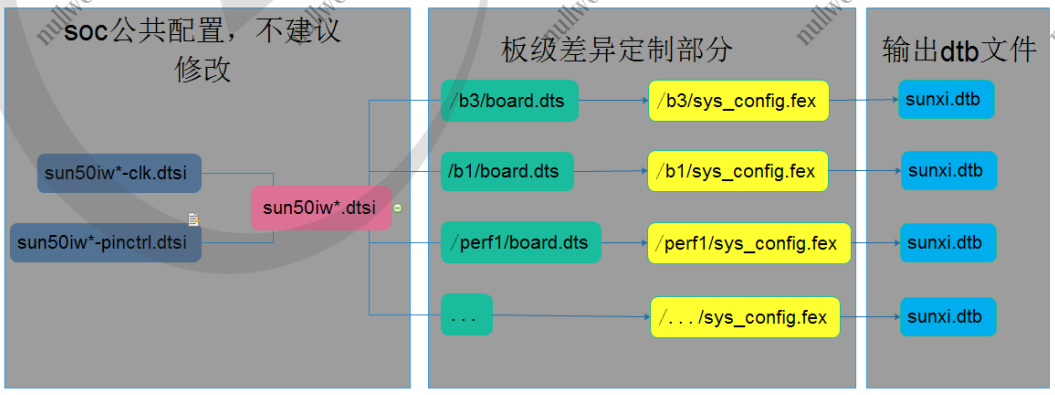

图 3: 设备树关系图

上图显示了三个方案的设备树配置信息，其中:

- 每个方案 dtb 文件，依赖于 board.dts，而 board.dts 又包含 sun50iw*.dtsi，当 board 级配置文件跟 SoC 级配置文件出现相同节点属性时，Board 级配置文件的属性值会去覆盖 SoC 级的相同属性值。

3.1.1 存在 sys_config.fex 配置情况

当存在 sys_config.fex 时，一份完整的配置可以包括三个部分：

- SoC 级配置文件：定义了 SoC 级配置，如设备时钟、中断等资源，如图 sun50iw*.dtsi。
- board 级配置文件：定义了板级配置，包含一些板级差异信息，如图 board.dts。
- sys_config.fex 配置文件，目前只保留 uart0 的配置，其他在 sys_config.fex 的配置只在 boot 阶段使用，内核驱动使用的信息，不建议在 sys_config.fex 文件配置，建议配置在 board.dts。

3.2 配置 sys_config.fex

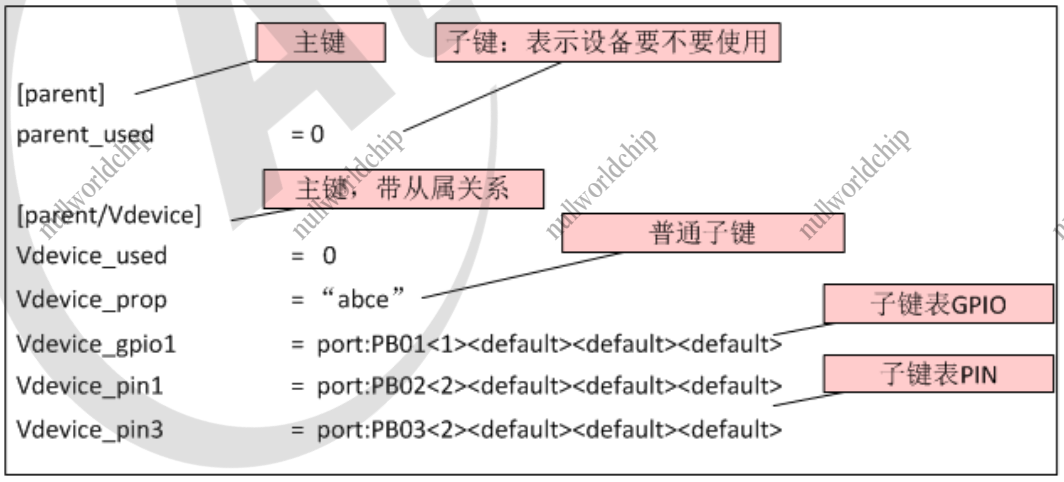

图 4: frame

注：sys_config.fex 后续只保留 boot 阶段使用的信息，内核不再使用该文件，不建议内核驱动相关信息在该文件上配置

3.3 配置 devicetree

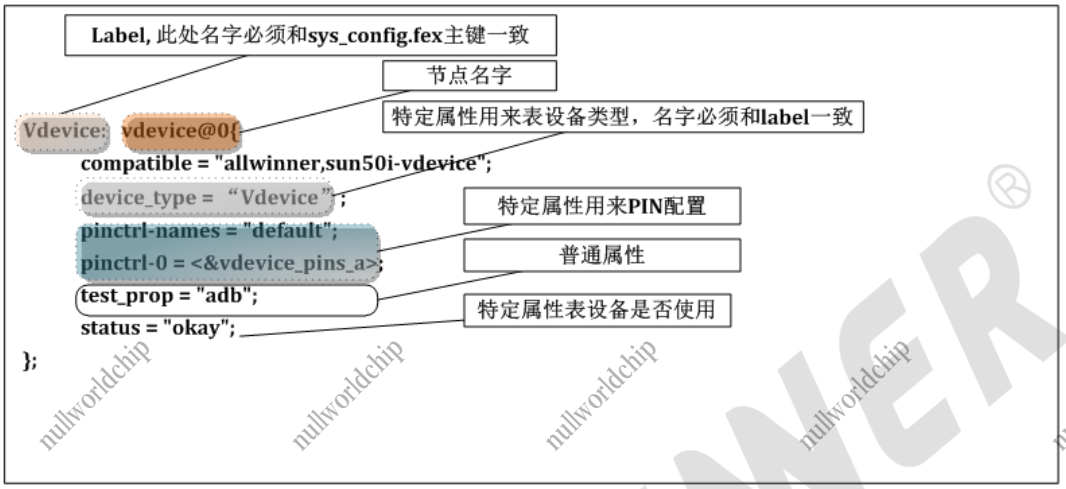

图 5: frame

4. 接口描述

Linux 系统为 device tree 提供了标准的 API 接口。

4.1 常用外部接口

使用内核提供的 device tree 接口，必须引用 Linux 系统提供的 device tree 接口头文件, 包含且不限于以下头文件：

```
#include <linux/of.h>
#include<linux/of_address.h>
#include<linux/of_irq.h>
#include<linux/of_gpio.h>
```

Device tree 常用接口如下介绍。

4.1.1 irq_of_parse_and_map

PROTOTYPE

```
unsigned int irq_of_parse_and_map(struct device_node *dev, int index);
```

ARGUMENTS

- dev 要解析中断号的设备:
- index dts 源文件中节点 interrupt 属性值索引。

RETURNS

- 如果解析成功，返回中断号，否则返回 0。

DEMO

以 timer 节点为例子：

```
Dts配置：
/{
    timer0: timer@1c20c00 {
        ...
        interrupts = <GIC_SPI 18 IRQ_TYPE_EDGE_RISING>;
        ...
    };
};
```

示例代码片段：

```
static void __init sunxi_timer_init(struct device_node *node){
    int irq;
    ....
    irq = irq_of_parse_and_map(node, 0);
    if (irq <= 0)
        panic("Can't parse IRQ");
}
```

4.1.2 of_iomap

PROTOTYPE

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

ARGUMENTS

- np 要映射内存的设备节点；
- index dts 源文件中节点 reg 属性值索引。

RETURNS

- 如果映射成功，返回 IO memory 的虚拟地址，否则返回 NULL。

DEMO

以 timer 节点为例子，dts 配置：

```
/{
    timer0: timer@1c20c00 {
        ...
        reg = <0x0 0x01c20c00 0x0 0x90>;
        ...
    };
};
```

以 timer 为例子，驱动代码片段：

```
static void __init sunxi_timer_init(struct device_node *node){
    ...
    timer_base = of_iomap(node, 0);
}
```

4.1.3 of_property_read_u32

PROTOTYPE

```
static inline int of_property_read_u32(const struct device_node *np,
                const char *propname,u32 *out_value);
```

ARGUMENTS

- np 想要获取属性值的节点:
- propname 属性名称:
- out_value 属性值。

RETURNS

- 如果取值成功，返回 0。

DEMO

以 timer 节点为例子，dts 配置例子：

```
/{
    soc_timer0: timer@1c20c00 {
        clock-frequency = <24000000>;
        timer-prescale = <16>;
    };
};
```

以 timer 节点为例子，驱动中获取 clock-frequency 属性值的例子：

```
int rate=0;
if (of_property_read_u32(node, "clock-frequency", &rate)) {
    pr_err("<%s> must have a clock-frequency property\n",node->name);
    return;
}
```

4.1.4 of_property_read_string

PROTOTYPE

```
static inline int of_property_read_string_index(struct device_node *np,
        const char *propname,const char **output);
```

ARGUMENTS

- np 想要获取属性值的节点:
- propname 属性名称:
- output 用来存放返回字符串。

RETURNS

- 如果取值成功，返回 0。

DESCRIPTION

- 该函数用于获取节点中属性值。(针对属性值为字符串)

DEMO

以 timer 节点为例子，dts 配置例子：

```
/{
    soc_timer0: timer@1c20c00 {
        clock-frequency = <24000000>;
        timer-prescale = <16>;
    };
};
```

以 timer 节点为例子，驱动中获取 clock-frequency 属性值的例子：

```
const char *name = NULL;
if (of_property_read_string(node, "clock-frequency", &name)) {
    pr_err("<%s> must have a clock-frequency property\n",node->name);
    return;
}
```

4.1.5 of_property_read_string_index

PROTOTYPE

```
static inline int of_property_read_string_index(struct device_node *np,
                const char *propname,int index, const char **output);
```

ARGUMENTS

- np 想要获取属性值的节点:
- propname 属性名称:
- index 用来索引配置在 dts 中属性为 propname 的值:
- output 用来存放返回字符串。

RETURNS

- 如果取值成功，返回 0。

DESCRIPTION

- 该函数用于获取节点中属性值。（针对属性值为字符串）

DEMO

例如获取 string-prop 的属性值，Dts 配置：

```
/{
    soc@01c20800{
        vdevice: vdevice@0{
            ...
            string_prop = "abcd";
        };
    };
};
```

例示驱动代码：

```
test{
    const char *name = NULL;
    ....
    err = of_property_read_string_index(np, "string_prop", 0, &name);
    if (WARN_ON(err))
        return;
}
```

4.1.6 of_find_node_by_name

PROTOTYPE

```
extern struct device_node *of_find_node_by_name(struct device_node *from,
                                                const char *name);
```

ARGUMENTS

- from: 从哪个节点开始找起:
- name: 想要查找节点的名字。

RETURNS

- 如果成功，返回节点结构体，失败返回 null。

DESCRIPTION

- 该函数用于获取指定名称的节点。

DEMO

获取名字为 vdeivce 的节点，dts 配置

```
/{
    soc@01c20800{
        vdevice: vdevice@0{
            ...
            string_prop = "abcd";
        };
    };
};
```

示例代码片段：

```
test{
    struct device_node *node;
    ....
    node = of_find_node_by_name(NULL, "vdevice");
    if (!node){
        pr_warn("can not get node.\n");
    };
    of_node_put(node);
}
```

4.1.7 of_find_node_by_type

PROTOTYPE

```
extern struct device_node *of_find_node_by_name(struct device_node *from,
                    const char *name);
```

ARGUMENTS

- from: 从哪个节点开始找起:
- type: 想要查找节点中 device_type 包含的字符串。

RETURNS

- 如果成功，返回节点结构体，失败返回 null。

DESCRIPTION

- 该函数用于获取指定 device_type 的节点。

DEMO

获取名字为 vdeivce 的节点，dts 配置

```
/{
    soc@01c20800{
        vdevice: vdevice@0{
            ...
            device_type = "vdevice";
            string_prop = "abcd";
        };
    };
};
```

示例代码片段：

```
test{
    struct device_node *node;
    ....
    node = of_find_node_by_type(NULL, "vdevice");
    if (!node){
        pr_warn("can not get node.\n");
    };
    of_node_put(node);
}
```

4.1.8 of_find_node_by_path

PROTOTYPE

```
extern struct device_node *of_find_node_by_path(const char *path);
```

ARGUMENTS

- path 通过指定路径查找节点。

RETURNS

- 如果成功，返回节点结构体，失败返回 null。

DESCRIPTION

- 该函数用于获取指定路径的节点。

DEMO

获取名字为 vdeivce 的节点，dts 配置

```
/{
    soc@01c20800{
        vdevice: vdevice@0{
            ...
            device_type = "vdevice";
            string_prop = "abcd";
        };
    };
};
```

示例代码片段：

```
test{
    struct device_node *node;
    ....
    node = of_find_node_by_path("/soc@01c2000/vdevice@0");
    if (!node){
        pr_warn("can not get node.\n");
    };
    of_node_put(node);
}
```

4.1.9 of_get_named_gpio_flags

PROTOTYPE

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

ARGUMENTS

- np 包含所需要查找 GPIO 的节点:

- propname 包含 GPIO 信息的属性:
- index 属性 propname 中属性值的索引:
- flags 用来存放 gpio 的 flags。

RETURNS

- 如果成功，返回 gpio 编号，flags 存放 gpio 配置信息，失败返回 null。

DESCRIPTION

- 该函数用于获取指定名称的 gpio 信息。

DEMO

获取名字为 vdeivce 的节点，dts 配置

```
/{
    soc@01c20800{
        vdevice: vdevice@0{
            ...
            test-gpios=<&pio PA 1 1 1 1 0>;
        };
    };
};
```

例示代码片段：

```
static int gpio_test(struct platform_device *pdev)
{
    struct gpio_config config;
    ....
    node=of_find_node_by_type(NULL, "vdevice");
    if(!node){
        printk(" can not find node\n");
    }
```

```
    ret = of_get_named_gpio_flags(node, "test-gpios", 0,
                        (enum of_gpio_flags *)&config);
    if (!gpio_is_valid(ret)) {
        return -EINVAL;
    }
};
```

5. 其他

5.1 sysfs 设备节点

device tree 会解析 dtb 文件中，并在/sys/devices 目录下会生成对应设备节点，其节点命名规则如下：（1）" 单元地址. 节点名" 节点名的结构是" 单元地址. 节点名"，例如 1c28000.uart、1f01400.prcm。形成这种节点名的设备，在 device tree 里的节点配置具有 reg 属性。

```
uart0: uart@01c28000 {
    compatible = "allwinner,sun50i-uart";
    reg = <0x0 0x01c28000 0x0 0x400>;
    ......
};

prcm {
    compatible = "allwinner,prcm";
    reg = <0x0 0x01f01400 0x0 0x400>;
};
```

（2）" 节点名. 编号" 节点名的结构是``节点名. 编号"，例如 soc.0、usbc0.5。形成这种节点名的设备，在 device tree 里的节点配置没有 reg 属性。

```
soc: soc@01c00000 {
    compatible = "simple-bus";
    ......
};

usbc0:usbc0@0 {
    compatible  = "allwinner,sunxi-otg-manager";
    ......
};
```

编号是按照在 device tree 中的出现顺序从 0 开始编号，每扫描到这样一个节点，编号就增加 1，如 SoC 节点是第 1 个出现的，所以编号是 0，而 usbc0 是第 6 个出现的，所以编号是 5。device tree 之所以这么做，是因为 device tree 中允许配置同名节点，所以需要通过单元地址或者编号来区分这些同名节点。可以参见内核的具体实现代码：

```
arm64_device_init()
    ->of_platform_populate()
        ->of_platform_bus_create()
            ->of_platform_device_create_pdata()
                ->of_device_alloc()
                    ->of_device_make_bus_id()
of_device_make_bus_id()
{
    ......
    reg = of_get_property(node, "reg", NULL);
    if (reg) {
        ......
        dev_set_name(dev, "%llx.%s", (unsigned long long)addr, node->name);
        return;
    }
    magic = atomic_add_return(1, &bus_no_reg_magic);
    dev_set_name(dev, "%s.%d", node->name, magic - 1);
}
```

6. Declaration

This document is the original work and copyrighted property of Allwinner Technology (“Allwinner”). Reproduction in whole or in part must obtain the written approval of Allwinner and give clear acknowledgement to the copyright owner. The information furnished by Allwinner is believed to be accurate and reliable. Allwinner reserves the right to make changes in circuit design and/or specifications at any time without notice. Allwinner does not assume any responsibility and liability for its use. Nor for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of Allwinner. This document neither states nor implies warranty of any kind, including fitness for any particular application.tates nor implies warranty of any kind, including fitness for any particular application.