

A64

A64 dev tree&sysconfig 使用
文档/V1.0

文档履历

版本号	日期	制/修订人	制/修订记录
V1.0	2015.03.12	HSR	初始化版本

Confidential

目 录

1. 引言	4
1.1. 编写目的	4
1.2. 适用范围	4
1.3. 相关人员	4
1.4. 术语与缩略语	4
2. 模块介绍	5
2.1. 模块功能介绍	5
2.2. 相关术语介绍	5
3. 如何配置	6
3.1. 配置文件关系	6
3.1.1. 不存在 sys_config.fex 配置情况	6
3.1.2. 存在 sys_config.fex 配置情况	6
3.2. 配置 sys_config.fex	7
3.3. 配置 devicetree	7
4. 接口描述	9
4.1. 常用外部接口	9
4.1.1. irq_of_parse_and_map	9
4.1.2. of_iomap	10
4.1.3. of_property_read_u32	10
4.1.4. of_property_read_string	11
4.1.5. of_property_read_string_index	12
4.1.6. of_find_node_by_name	13
4.1.7. of_find_node_by_type	13
4.1.8. of_find_node_by_path	14
4.1.9. of_get_named_gpio_flags	15
4.2. Sys_config 接口&&dts 接口映射	16
4.2.1. 获取子键内容	17
4.2.2. 获取主键下 GPIO 列表	17
4.2.3. 获取主键数量	17
4.2.4. 获取主键名称	18
4.2.5. 判断主键是否存在	18
5. 接口使用例子	19
5.1. 配置比较	19
5.2. 获取整形属性值	20
5.3. 获取字符型属性值	20
5.4. 获取 gpio 属性值	21
5.5. 获取节点	22
6. 总结	24

Confidential

1. 引言

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 aw1689 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. 术语与缩略语

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

2. 模块介绍

Device Tree 是一种描述硬件的数据结构，可以把嵌入式系统资源抽象成一棵树形结构，可以直观查看系统资源分布；内核可以识别这棵树，并根据它展开出 Linux 内核中的 platform_device 等

2.1. 模块功能介绍

Device Tree 改变了原来用 hardcode 方式将 HW 配置信息嵌入到内核代码的方法，消除了 arch/arm64 下大量的冗余编码。使得各个厂商可以更专注于 driver 开发，开发流程遵从 mainline kernel 的规范。

2.2. 相关术语介绍

FDT: 嵌入式 PowerPC 中，为了适应内核发展&&嵌入式 PowerPC 平台的千变万化，推出了 Standard for Embedded Power Architecture Platform Requirements (ePAPR) 标准，吸收了 Open Firmware 的优点，在 U-boot 引入了扁平设备树 FDT 接口，使用一个单独的 FDT blob 对象将系统硬件信息传递给内核。

DTS: device tree 源文件，包含用户配置信息。

对于 32bit Arm 架构， dts 文件存放在 arch/arm/boot/dts 路径下。

对于 64bit Arm 架构， dts 文件存放在 arch/arm64/boot/dts 路径下。

DTB: DTB 是 DTS 被 DTC 编译后二进制格式的 Device Tree 描述，可由 Linux 内核解析，并为设备驱动提供硬件配置信息。

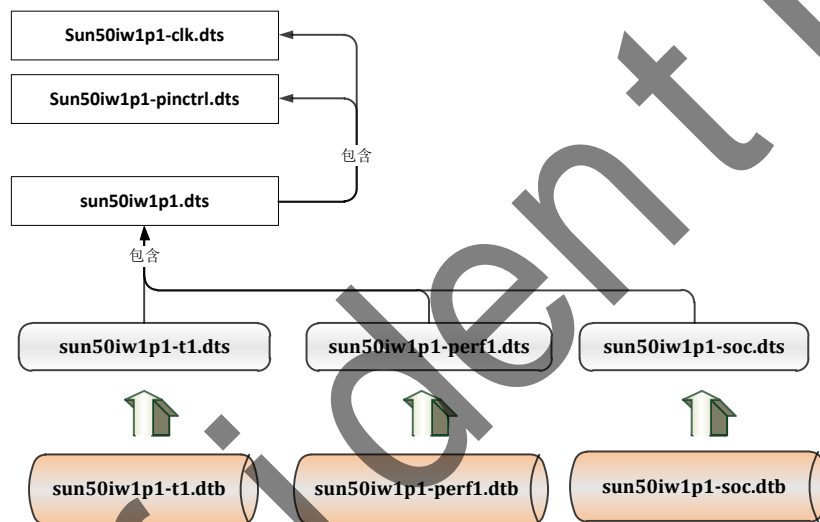
3. 如何配置

3.1. 配置文件关系

3.1.1. 不存在 sys_config.fex 配置情况

当不存在 sys_config.fex 时，一份完整的配置可以包括两个部分：

- 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 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，如图 sun50iw1p1-t1.dtsi。



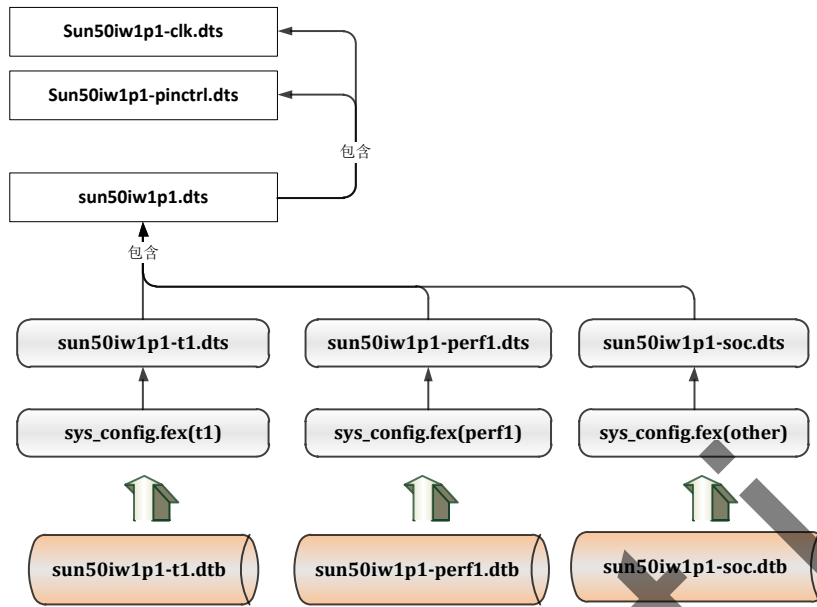
上图显示了三个方案的设备树配置信息，其中：

- 每个方案 dtb 文件，依赖于 sun50iw1p1-\${board}.dts，而 sun50iw1p1-\${board}.dtsi 又包含 sun50iw1p1.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。
- 图示中 sun50iw1p1-soc.dts 文件跟 sun50iw1p1-t1.dts 与 sun50iw1p1-perf1.dts 一样，都属于 board 配置文件。该配置文件定义为一种通用的 board 配置文件，主要为了防止客户移植新的方案时，没有在内核 linux-3.10/arch/arm64/boot/dts/目录下定义客户方案的 board 级配置文件。如果出现这样的情况，内核编译的时候，就会采用 sun50iw1p1-soc.dts，作为该客户方案的 board 级配置文件。

3.1.2. 存在 sys_config.fex 配置情况

当不存在 sys_config.fex 时，一份完整的配置可以包括三个部分：

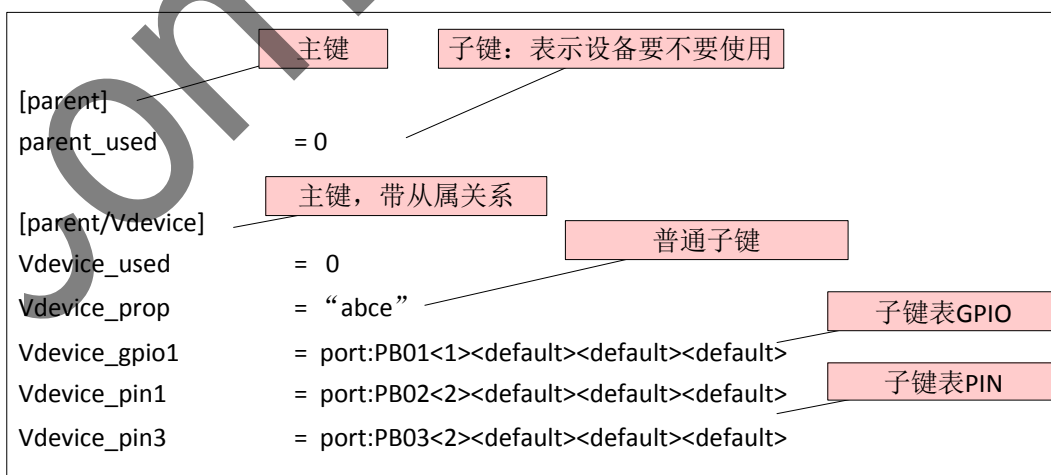
- soc 级配置文件：定义了 SOC 级配置，如设备时钟、中断等资源，如图 sun50iw1p1.dtsi。
- board 级配置文件：定义了板级配置，包含一些板级差异信息，如图 sun50iw1p1-t1.dtsi。
- sys_config.fex 配置文件，为方便客户使用而定义，优先级比 board 级配置、soc 级配置都高。



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，既包含 sys_config.fex 配置信息，同时又依赖于 sun50iw1p1-\${board}.dts，而 sun50iw1p1-\${board}.dtsi 又包含 sun50iw1p1.dtsi，sys_config.fex 配置文件的优先级最高，sys_config.fex 跟 devicetree 文件都存在配置项时，sys_config.fex 的配置项内容会更新到 board 级配置文件或者 soc 级配置文件对应的配置项上去。
- 图示中 sun50iw1p1-soc.dts 文件跟 sun50iw1p1-t1.dts 与 sun50iw1p1-perf1.dts 一样，都属于 board 配置文件。该配置文件定义为一种通用的 board 配置文件，主要为了防止客户移植新的方案时，没有在内核 linux-3.10/arch/arm64/boot/dts/目录下定义客户方案的 board 级配置文件。如果出现这样的情况，内核编译的时候，就会采用 sun50iw1p1-soc.dts，作为该客户方案的 board 级配置文件。

3.2. 配置 sys_config.fex



3.3. 配置 devicetree


```

Vdevice: vdevice@0{
    compatible = "allwinner,sun50i-vdevice";
    device_type = "Vdevice";
    pinctrl-names = "default";
    pinctrl-0 = <&vdevice_pins_a>;
    test_prop = "adb";
    status = "okay";
};

```

Label, 此处名字必须和sys_config.fex主键一致

节点名字

特定属性用来表设备类型, 名字必须和label一致

特定属性用来PIN配置

普通属性

特定属性表设备是否使用

详细命名规范, 参考《sys_config.fex&&dtb 命名规范》。

sonfidential

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

驱动代码片段:

```
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。

➤ **DESCRIPTION**

该函数用于获取节点中的属性值。

➤ **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**

```
//例如获取 string-prop 的属性值, Dts 配置:
/{
    soc@01c20800{
        vdevice: vdevice@0{
            ...
            string_prop = "abcd";
        };
    };
};

例示代码:
test{
    const char *name;
```

```
....  
err = of_property_read_string_index(np, "string_prop", &name);  
if (WARN_ON(err))  
    return;  
}
```

4.1.5. of_property_read_string_index

➤ PROTOTYPE

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

➤ ARGUMENTS

np 想要获取属性值的节点
Proprname 属性名称
Index 用来索引配置在 dts 中属性为 proprname 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";  
        };  
    };  
};
```

例示代码：

```
test{  
    const char *name;  
    ....  
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**

clk 待操作的时钟句柄;
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vdevice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 *type);
```

➤ **ARGUMENTS**

clk 待操作的时钟句柄;
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 vdevice 的节点, dts 配置
/{
    soc@01c20800{
        vdevice: vdevice@0{
            ...
            device_type = "vdevice";
            string_prop = "abcd";
        };
    };
};
```

例示代码片段:

```
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 vdevice 的节点, dts 配置
```

```

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

```

例示代码片段:

```

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
//获取名字为 vdeivce 的节点, dts 配置
/{
    soc@01c20800{
        vdevice: vdevice@0{
            ...

```



```
    device_type = "vdevice";  
    string_prop = "abcd";  
};  
};  
};
```

例示代码片段:

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

4.2. Sys_config 接口&&dts 接口映射

4.2.1. 获取子键内容

Script API:

作用：通过主键名和子键名字，获取子键内容（该接口可以自己识别子键的类型）

原型：

```
script_item_value_type_e script_get_item(char *main_key, char *sub_key, script_item_u *item);
```

Dts API:

说明：dts 标准接口支持通过节点和属性名，获取属性值（用户需要知道属性值得类型）：

作用：获取属性值，使用于属性值为整型数据：

原型：

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

作用：获取属性值，使用于属性值为字符串：

原型：

```
int of_property_read_string(struct device_node *np, const char *propname, const char **out_string)
```

作用：获取 GPIO 信息。

原型：

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

4.2.2. 获取主键下 GPIO 列表

Script API:

作用：获取主键下 GPIO 列表

```
int script_get_pio_list(char *main_key, script_item_u **list);
```

Dts API:

无对应接口。

4.2.3. 获取主键数量

Script API:

作用：获取主键数量

```
unsigned int script_get_main_key_count(void);
```

Dts API:

无对应接口。

4.2.4. 获取主键名称

Script API:

作用：通过主键索引号，获取主键名字

```
char *script_get_main_key_name(unsigned int main_key_index);
```

Dts API:

无对应接口。

4.2.5. 判断主键是否存在

Script API:

作用：判断主键是否存在

```
bool script_is_main_key_exist(char *main_key);
```

Dts API:

说明：dts 标准接口支持四种方式判断节点是否存在：

a. 通过节点名字：

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

b. 通过节点路径：

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

c. 通过节点 phandle 属性：

```
struct device_node *of_find_node_by_phandle(phandle handle)
```

d. 通过节点 device_type 属性：

```
struct device_node *of_find_node_by_type(struct device_node *from, const char *type)
```

5. 接口使用例子

5.1. 配置比较

下表展示了设备 vdevice 在 sys_config.fex 与 dts 中的配置，两种配置形式不一样，但实现的功能是等价的。

```

/*
device config in dts:
/{
    soc@01c20000{
        vdevice@0{
            compatible = "allwinner,sun50i-vdevice";
            device_type= "vdevice";
            vdevice_0=<&pio 1 1 1 1 1 0>;
            vdevice_1=<&pio 1 2 1 1 1 0>;
            vdevice-prop-1=<0x1234>;
            vdevice-prop-3="device-string";
            status = "okay";
        };
    };
};
device config in sys_config.fex
[vdevice]
compatible          = "allwinner,sun50i-vdevice";
vdevice_used        = 1
vdevice_0           = port:PB01<1><1><2><default>
vdevice_1           = port:PB02<1><1><2><default>
vdevice-prop-1      = 0x1234
vdevice-prop-3      = "device-string"
*/

```

说明：GPIO_IN/GPIO_OUT/EINT 采用下边的配置方式,PIN 采用另外配置，参考 pinctrl 使用说明文档。

```

vdevice_0=<&pio 1 1 1 1 1 0>;
|          | | | | | |-----电平
|          | | | | |-----上下拉
|          | | | |-----驱动力
|          | | |-----复用类型，0-GPIOIN 1-GPIOOUT..
|          | |-----pin bank 内偏移.
|          |-----哪个 bank，PA=0, PB=1...以此类推
|          |-----指向哪个 pio，属于 cpus 要用&r_pio
|-----属性名字，相当 sys_config 子键名

```

5.2. 获取整形属性值

通过 script 接口:

```
#include <linux/sys_config.h>
int get_subkey_value_int(void)
{
    script_item_u script_val;
    script_item_value_type_e type;

    type = script_get_item("vdevice", "vdevice-prop-1", &script_val);
    if (SCIRPT_ITEM_VALUE_TYPE_INT != type) {
        return -EINVAL;
    }
    return 0;
}
```

通过 dts 接口:

```
#include <linux/of.h>
int get_subkey_value_int(void)
{
    int ret;
    u32 value;
    struct device_node *node;

    node = of_find_node_by_type(NULL, "vdevice");
    if (!node) {
        return -EINVAL;
    }
    ret = of_property_read_u32(node, "vdevice-prop-1", &value);
    if (ret) {
        return -EINVAL;
    }
    printk("prop-value=%x\n", value);

    return 0;
}
```

5.3. 获取字符型属性值

通过 script 接口:

```
#include <linux/sys_config.h>
```

```
int get_subkey_value_string(void)
{
    script_item_u script_val;
    script_item_value_type_e type;

    type = script_get_item("vdevice", "vdevice-prop-3", &script_val);
    if (SCIRPT_ITEM_VALUE_TYPE_STR!= type) {
        return -EINVAL;
    }
    return 0;
}
```

通过 dts 接口:

```
#include <linux/of.h>
int get_subkey_value_string(void)
{
    int ret;
    const char *string;
    struct device_node *node;

    node = of_find_node_by_type(NULL, "vdevice");
    if(!node){
        return -EINVAL;
    }
    ret = of_property_read_string(node, "vdevice-prop-3", &string);
    if(ret){
        return -EINVAL;
    }
    printk("prop-vlvalue=%s\n", string);

    return 0;
}
```

5.4. 获取 gpio 属性值

通过 script 接口:

```
#include <linux/sys_config.h>
int get_gpio_info(void)
{
    script_item_u script_val;
    script_item_value_type_e type;
```

```
type = script_get_item("vdevice", "vdevice_0", &script_val);
if (SCIRPT_ITEM_VALUE_TYPE_PIO!= type) {
    return -EINVAL;
}
return 0;
}
通过 dts 接口:
#include <linux/sys_config.h>
#include <linux/of.h>
#include <linux/of_gpio.h>
int get_gpio_info(void)
{
    unsigned int gpio;
    struct gpio_config config;
    struct device_node *node;

    node = of_find_node_by_name(NULL,"vdevice");
    if(!node){
        return -EINVAL;
    }
    gpio = of_get_named_gpio_flags(node, "vdevice_0", 0, (enum of_gpio_flags *)&config);
    if (!gpio_is_valid(gpio)) {
        return -EINVAL;
    }
    printk("pin=%d  mul_sel=%d  drive=%d  pull=%d  data=%d gpio=%d\n",
           config.gpio,
           config.mul_sel,
           config.driv_level,
           config.pull,
           config.data,
           gpio);
    return 0;
}
```

5.5. 获取节点

```
通过 scrip 接口:
#include <linux/sys_config.h>
int check_mainkey_exist(void)
{
    int ret;
    ret = script_is_main_key_exist("vdevice");
}
```

```
    if(!ret){
        return -EINVAL;
    }
}

通过 dts 接口:
int check_mainkey_exist(void)
{
    struct device_node *node_1, *node_2;
    /* mode 1*/
    node_1 = of_find_node_by_name(NULL,"vdevice");
    if(!node_1){
        printk("can not find node in dts\n");
        return -EINVAL;
    }
    /*mode 2 */
    node_2 = of_find_node_by_type(NULL,"vdevice");
    if(!node_2){
        return -EINVAL;
    }
    return 0;
}
```


6. 总结

Confidential