4 pinctrl 和 gpio 子系统

在前面的实验中，我们操作设备都是直接操作寄存器的。虽然上一章使用了设备树，但也只是把寄存器信息放到设备树中，本质上都是一样的。直接操作寄存器来控制硬件资源对于编码的人来说很方便，然而对于阅读代码的人来说想必是很崩溃的。而且在设备树中写入大量的寄存器地址，不仅不易于阅读，在使用时也很容易出错。比如上一章中，仅是控制 led 就用了 4 个寄存器，而且寄存器地址在设备树中的顺序也没有规则，全是自定的，在驱动程序中读取地址也无法对应和确认，最终我们在设置寄存器时就很容易错位。

pinctrl 子系统来可以应对这个问题，同时针对我们实验中用到的 led 也就是 gpio 操作来说还需要结合 gpio 子系统。

4.1 pinctrl 子系统

Linux 内核提供 pinctrl 子系统来统一芯片厂商的 pin 脚管理方法，他会帮我们完成以下工作：

1. 从设备树中获取 pin 脚信息；
2. 设置 pin 脚的复用功能；
3. 配置 pin 脚的电器特性。

使用 pinctrl 子系统，只需要在设备树中配置好相关节点信息，剩余的工作全由 pinctrl 子系统完成，我们只要做甩手掌柜就行了。

4.1.1 zynq 平台 pinctrl 子系统的设备树节点和属性

不同的芯片 pin 脚配置方法有差别，相应的芯片厂家的 pinctrl 子系统实现也会有差别，但我们需要关心的仍然只有设备树的写法。直接来看一段示例：

```
1.  pinctrl0: pinctrl@700 {
2.      compatible = "xlnx,pinctrl-zynq";
3.      reg = <0x700 0x200>;
4.      syscon = <&slcr>;
5.      pinctrl_uart1_default: uart1-default {
6.          mux {
7.              groups = "uart1_10_grp";
8.              function = "uart1";
9.          };
10.         conf {
11.             groups = "uart1_10_grp";
12.             slew-rate = <0>;
13.             io-standard = <1>;
14.         };
15.         conf-rx {
```

```
16.             pins = "MIO49";
17.             bias-high-impedance;
18.         };
19.         conf-tx {
20.             pins = "MIO48";
21.             bias-disable;
22.         };
23.     };
24. };
```

第一行的 pinctrl0: pinctrl@700 节点，就是 ZYNQ 中 pinctrl 子系统的节点。pinctrl0 的父节点为 slcr@f8000000，700 是相对 f8000000 的偏移，也就说 pin 的寄存器起始地址为 0xf8000700。可以在数据手册中找到对应的寄存器说明。

B.28 System Level Control Registers (slcr)

Module Name	System Level Control Registers (slcr)
Base Address	0xF8000000 slcr
Description	System Level Control Registers
Version	1.0
Doc Version	1.3, based on 11/18/2010 SLCR_spec.doc
Vendor Info	Xilinx Zynq slcr

MIO_PIN_00	0x00000700	32	rw	0x00001601	MIO Pin 0 Control
MIO_PIN_01	0x00000704	32	rw	0x00001601	MIO Pin 1 Control
MIO_PIN_02	0x00000708	32	rw	0x00000601	MIO Pin 2 Control
MIO_PIN_03	0x0000070C	32	rw	0x00000601	MIO Pin 3 Control
MIO_PIN_04	0x00000710	32	rw	0x00000601	MIO Pin 4 Control

不过寄存相关的内容不是这里我们需要关系的对象，示例的 2、3、4 行是 pinctrl0 的三个属性，都是 xilinx 设置好的，也不用去关心。重点是第 5 行开始，在 pinctrl0 节点中添加的子设备节点 uart1-default，我们要通过学习这个子设备节点的配置方法，添加我们自己需要的设备节点。这个节点下面没有属性，而是有 4 个子节点：mux、conf、conf-rx、conf-tx。其中 mux 是配置 pin 脚的复用功能，其他三个是配置 pin 脚的电气特性，conf 中设置通用的特性，conf-rx 中设置输入引脚和相关特性，conf-tx 中设置输出引脚和相关特性。

mux 节点中，有两个必要属性：

groups：选择这个复用功能的 pin 脚分组列表。可选的值有：

ethernet0_0_grp, ethernet1_0_grp, mdio0_0_grp, mdio1_0_grp, qspi0_0_grp, qspi1_0_grp, qspi_fbclk, qspi_cs1_grp, spi0_0_grp - spi0_2_grp, spi0_X_ssY (X=0..2, Y=0..2), spi1_0_grp - spi1_3_grp, spi1_X_ssY (X=0..3, Y=0..2), sdio0_0_grp - sdio0_2_grp, sdio1_0_grp - sdio1_3_grp, sdio0_emio_wp, sdio0_emio_cd, sdio1_emio_wp, sdio1_emio_cd, smc0_nor, smc0_nor_cs1_grp, smc0_nor_addr25_grp, smc0_nand, can0_0_grp - can0_10_grp, can1_0_grp - can1_11_grp, uart0_0_grp - uart0_10_grp, uart1_0_grp - uart1_11_grp, i2c0_0_grp - i2c0_10_grp, i2c1_0_grp - i2c1_10_grp, ttc0_0_grp - ttc0_2_grp, ttc1_0_grp - ttc1_2_grp, swdt0_0_grp - swdt0_4_grp, gpio0_0_grp - gpio0_53_grp, usb0_0_grp,

usb1_0_grp。

这些分组包含哪些 pin 脚，可以在 drivers/pinctrl/pinctrl-zynq.c 文件中找到相应的数组，比如示例中的 group"uart1_10_grp"，对应的数组为 static const unsigned int uart1_10_pins[] = {48, 49}，这个 group 包含的 pin 脚即为 pin48 和 pin49。

function：选择 pin 脚分组的复用功能。可选的值有：

ethernet0, ethernet1, mdio0, mdio1, qspi0, qspi1, qspi_fbclk, qspi_cs1, spi0, spi0_ss, spi1, spi1_ss, sdio0, sdio0_pc, sdio0_cd, sdio0_wp, sdio1, sdio1_pc, sdio1_cd, sdio1_wp, smc0_nor, smc0_nor_cs1, smc0_nor_addr25, smc0_nand, can0, can1, uart0, uart1, i2c0, i2c1, ttc0, ttc1, swdt0, gpio0, usb0, usb1。

示例中的设备节点名为 uart1-default，用作 uart1，所以 function 的值是 uart1。

需要什么功能，可以用什么引脚，可以从数据手册中查看，也可以直接在 drivers/pinctrl/pinctrl-zynq.c 文件中查看，很容易看出对应关系。

conf 节点可以分为多组，因为相同复用功能的 pin 组里每个 pin 的电器特性可能会不同，比如示例中的 uart1，有一个 pin 脚用作输出不需要设置偏置，另一个用做输入需要设置成高阻。conf 节点的名字必须要包含"conf"这个关键词，其他的关键词仅用于区分分组含义，比如示例中的 conf-rx 是指这个 conf 节点适用于配置输入 pin 脚的，命名为 conf-in 也是同样的效果，甚至把示例中 rx 和 tx 交换，对使用也没用影响，仅用于含以上的区分。

conf 节点的必要属性为 **pin** 和 **group** 二选一。设置 pin 组时使用属性 group，如示例中第 10 行的 conf 节点，设置单个 pin 或多个不在一个 group 中的 pin 时使用属性 **pin**，如 15 行的 conf-rx 和 19 行的 conf-tx。

group 属性设上面 mux 节点中的一样。

pin：使用这个 conf 的 pin 脚列表。可选的值为：MIO0 - MIO53。

conf 节点还有一些可选属性：

io-standard：设置 io 电平标准，可选值为：

1: LVCMOS18

2: LVCMOS25

3: LVCMOS33

4: HSTL

bias-disable、bias-high-impedance、bias-pull-up：设置 pin 脚偏置，不需要赋值，只需要在对应的 conf 节点中包含需要的偏置属性即可。

slew-rate：设置转换速率，等于 0 时为 slow，等于 1 时为 fast。

low-power-disable、low-power-enable：使能低功耗模式，在对应的 conf 节点中包含需要的偏置属性即可。

还有其他的可选属性不就一一介绍了，可以在内核源码目录 Documentation/devicetree/bindings/pinctrl/pinctrl-binding.txt 文件中，搜索"== Generic pin configuration node content =="词条查看。

4.1.2 pinctrl 子系统的使用

在 pinctrl 节点中添加了设备子节点后，系统还不回去初始化这个设备，我们需要在客户设备(client device)节点中去调用他。举个例子：

```
1.  device {
2.      pinctrl-names = "active", "idle";
```

```
3.      pinctrl-0 = <&state_0_node_a>;
4.      pinctrl-1 = <&state_1_node_a &state_1_node_b>;
5.  };
```

device 是客户设备节点，是调用 pinctrl 的设备。

pinctrl-0 是必选属性，是这个设备对应 pin 的 pinctrl 节点应用列表，如果这个设备对应了多个 pin，那就增加属性 **pinctrl-0~n**，如果一个 pin 对应了多个 pinctrl，就如第 4 行的 pinctrl-1 属性一般附上多个 pinctrl 节点。

pinctrl-names 是下面 pinctrl0~n 的名称列表，命名是自定的但要尽量易读并贴合实际。

4.2 gpio 子系统

pinctrl 子系统中，如果把功能配置成 gpio 的话，就需要用到 gpio 子系统了。gpio 子系统帮我们实现了 gpio 初始化，并提供了一些接口函数给我们操作 gpio。我们要做的同样只是在设备树中配置好信息，然后了解一下接口函数的用法即可。

4.2.1 设备树中的 gpio

在 Documentation/devicetree/bindings/gpio/gpio-zynq.txt 文件中，介绍了 zynq 系列芯片 gpio 在设备数中的描述方式，我们在现有的 zynq 设备数中，也能找到 gpio 的默认配置，如下：

```
1.  gpio0:gpio@e000a000 {
2.      #gpio-cells = <2>;
3.      compatible = "xlnx,zynq-gpio-1.0";
4.      clocks = <&clkc 42>;
5.      gpio-controller;
6.      interrupt-parent = <&intc>;
7.      interrupts = <0 20 4>;
8.      interrupt-controller;
9.      #interrupt-cells = <2>;
10.     reg = <0xe000a000 0x1000>;
11. };
```

这些属性都是必要属性。

interrupt-parent、interrupts、interrupt-controller、#interrupt-cells：这四个属性是中断相关的，等之后讲中断的时候再说。

clocks 属性时时钟设置相关，暂时使用默认配置即可。

#gpio-cells：指代引用这个 gpio 节点时需要两个 cell，他的值必须是 2，即引用节点时需要有两个 cell，第一个 cell 为 gpio 编号，第二个 cell 用于指顶可选参数，第二个 cell 在 zynq 中没有用到填 0 即可。举个例子：alinxled-gpios = <&gpio0 0 0>;，即客户设备的使用的 gpio0_mio0 这个 io。

compatible：兼容新，必须为”xlnx, zynq-gpio-1.0”或”xlnx, zynqmp-gpio-1.0”。

gpio-controller：指定该节点为 gpio 控制器。
reg：这组 gpio 的寄存器首地址和范围。
这里我们不需要修改什么，保持现有配置即可。

4.2.2 gpio 接口函数

gpio 子系统提供接口函数来代替我们直接对寄存器的操作把驱动程序再次分离分层。这节我们介绍一些常用的 gpio 接口函数。

4.2.2.1 of 函数扩展

在介绍 gpio 接口函数之前，先扩展一个 gpio 相关的 of 函数 of_get_named_gpio，用于通过设备树中节点属性如”gpios = <&gpio0 0 0 &gpio0 1 0>;”获取 gpio 编号，函数原型如下：
int of_get_named_gpio(struct device_node *np, const char *propname, int index)
参数说明：
np：设备节点。
propname：属性名称，从这个属性中获取 gpio 编号。
index：需要获取的 gpio 下标。
返回值：gpio 编号；返回负值则获取失败。

4.2.2.2 gpio_request()

在使用 gpio 做其他操作之前，需要先用 gpio_request()函数申请一个 io。至今为止接触到的 Linux 内核提供的接口，都是这种面向对象的思想，先声明一个对象，再操作这个对象。
函数原型：
int gpio_request(unsigned int gpio, const char *label)
参数说明：
gpio：需要申请的 gpio 标号，通过上面介绍的 of_get_named_gpio 函数获取。
label：申请到的 gpio 的标签。
返回值：0 表示成功，其他表示失败。

4.2.2.3 gpio_free()

释放 gpio，与 gpio_request()相对，原型：
void gpio_free(unsigned int gpio)
gpio：需要释放的 gpio 标号。

4.2.2.4 gpio_direction_input()

设置 gpio 为输入，函数原型：

int gpio_direction_input(unsigned gpio)
gpio：需要设置的 gpio 标号。
返回值：0 表示成功，负值表示失败。

4.2.2.5 gpio_direction_output()

设置 gpio 为输出，函数原型：
int gpio_direction_output(unsigned gpio, int value)
gpio：需要设置的 gpio 标号。
value：输出的默认电平。
返回值：0 表示成功，负值表示失败。

4.2.2.6 gpio_get_value()

读取 gpio 的电平，原型：
int __gpio_get_value(unsigned gpio)
gpio：需要读取的 gpio 标号。
返回值：读取到的电平，负值表示失败。

4.2.3.6 gpio_set_value()

设置 gpio 的电平，原型：
void __gpio_set_value(unsigned gpio, int value)
gpio：需要操作的 gpio 标号。
value：需要写入的电平。

4.3 实验

必要的知识都了解了，接下来通过实验来验证。仍然使用 led 来测试。

4.3.1 原理图

和章节 1.3.1 的内容相同。

4.3.2 设备树

打开文件”system-conf.dtsi”，在根节点下添加以下节点内容：

```
1.    amba {
2.       gpio@e000a000 {
```

```
3.          compatible = "xlnx,zynq-gpio-1.0";
4.          #gpio-cells = <0x2>;
5.          clocks = <0x1 0x2a>;
6.          gpio-controller;
7.          interrupt-controller;
8.          #interrupt-cells = <0x2>;
9.          interrupt-parent = <0x4>;
10.         interrupts = <0x0 0x14 0x4>;
11.         reg = <0xe000a000 0x1000>;
12.     };
13.
14.     slcr@f8000000 {
15.         pinctrl@700 {
16.             pinctrl_led_default: led-default {
17.                 mux {
18.                     groups = "gpio0_0_grp";
19.                     function = "gpio0";
20.                 };
21.
22.                 conf {
23.                     pins = "MIO0";
24.                     io-standard = <1>;
25.                     bias-disable;
26.                     slew-rate = <0>;
27.                 };
28.             };
29.         };
30.     };
31. };
32.
33. alinxled {
34.     compatible = "alinx-led";
35.     pinctrl-names = "default";
36.     pinctrl-0 = <&pinctrl_led_default>;
37.     alinxled-gpios = <&gpio0 0 0>;
38. };
```

第 **2~12** 行是 gpio 节点，这个内容其实是 zynq 设备树中已有的，内容 4.2.1 节讲过了。

第 **16~28** 行我们新增的 pin 设备节点，用于配置 led 用到的 pin 脚。led 链接的 io 是 MIO0，所在分组为"gpio0_0_grp"，功能就是 gpio0。因为只有一个引脚，所以只要一个 conf 节点就行了，转换率设置为 slow，电平标准为 1.8，偏置失能，配置目标为”MIO0”。

第 **33** 行则是我们的 led 设备节点，对于 gpio 子系统和 pinctrl 子系统来说，就是客户设备(client device)，通过属性 pinctrl-0 = <&pinctrl_led_default>调用 pin 节点，通过属性 alinxled-gpios = <&gpio0 0 0>来绑定 gpio。这个属性的名字 alinxled-gpios 需要记号，稍后我们

要通过这个名字来获取 gpio 标号。

4.3.3 驱动代码

使用 petalinux 新建名为”ax-pinioled-dev”的驱动程序，并执行 petalinux-config -c rootfs 命令选上新增的驱动程序。

在 ax- pinioled -dev.c 文件中输入下面的代码：

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/ide.h>
#include <linux/types.h>
#include <linux/errno.h>
#include <linux/cdev.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/of_gpio.h>
#include <linux/device.h>
#include <linux/delay.h>
#include <linux/init.h>
#include <linux/gpio.h>
#include <asm/uaccess.h>
#include <asm/mach/map.h>
#include <asm/io.h>

/* 设备节点名称 */
#define DEVICE_NAME       "gpio_leds"
/* 设备号个数 */
#define DEVID_COUNT       1
/* 驱动个数 */
#define DRIVE_COUNT       1
/* 主设备号 */
#define MAJOR1
/* 次设备号 */
#define MINOR1            0
/* LED 点亮时输入的值 */
#define ALINX_LED_ON      1
/* LED 熄灭时输入的值 */
#define ALINX_LED_OFF     0


/* 把驱动代码中会用到的数据打包进设备结构体 */
```

```
37. struct alinx_char_dev{
38.     dev_t              devid;            //设备号
39.     struct cdev        cdev;             //字符设备
40.     struct class       *class;           //类
41.     struct device      *device;          //设备
42.     struct device_node *nd;              //设备树的设备节点
43.     int                alinx_led_gpio;   //gpio 号
44. };
45. /* 声明设备结构体 */
46. static struct alinx_char_dev alinx_char = {
47.     .cdev = {
48.         .owner = THIS_MODULE,
49.     },
50. };
51.
52. /* open 函数实现，对应到 Linux 系统调用函数的 open 函数 */
53. static int gpio_leds_open(struct inode *inode_p, struct file *file_p)
54. {
55.     /* 设置私有数据 */
56.     file_p->private_data = &alinx_char;
57.     printk("gpio_test module open\n");
58.     return 0;
59. }
60.
61.
62. /* write 函数实现，对应到 Linux 系统调用函数的 write 函数 */
63. static ssize_t gpio_leds_write(struct file *file_p, const char __user *buf, size_t len, lof
    f_t *loff_t_p)
64. {
65.     int retvalue;
66.     unsigned char databuf[1];
67.     /* 获取私有数据 */
68.     struct alinx_char_dev *dev = file_p->private_data;
69.     retvalue = copy_from_user(databuf, buf, len);
70.     if(retvalue < 0)
71.     {
72.         printk("alinx led write failed\r\n");
73.     return -EFAULT;
74.     }
75.
76.     if(databuf[0] == ALINX_LED_ON)
77.     {
78.         gpio_set_value(&alinx_char.alinx_led_gpio, !!0);
79.     }
```

```
80.     else if(databuf[0] == ALINX_LED_OFF)
81.     {
82.         gpio_set_value(&alinx_char.alinx_led_gpio, !!1);
83.     }
84.     else
85.     {
86.         printk("gpio_test para err\n");
87.     }
88.
89.     return 0;
90. }
91.
92. /* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
93. static int gpio_leds_release(struct inode *inode_p, struct file *file_p)
94. {
95.     printk("gpio_test module release\n");
96.     return 0;
97. }
98.
99. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关键 */
100. static struct file_operations ax_char_fops = {
101.     .owner   = THIS_MODULE,
102.     .open    = gpio_leds_open,
103.     .write   = gpio_leds_write,
104.     .release = gpio_leds_release,
105. };
106.
107. /* 模块加载时会调用的函数 */
108. static int __init gpio_led_init(void)
109. {
110.     /* 用于接受返回值 */
111.     u32 ret = 0;
112.
113.     /* 获取设备节点 */
114.     alinx_char.nd = of_find_node_by_path("/alinxled");
115.     if(alinx_char.nd == NULL)
116.     {
117.         printk("alinx_char node not find\r\n");
118.         return -EINVAL;
119.     }
120.     else
121.     {
122.         printk("alinx_char node find\r\n");
123.     }
```

```
124.
125.    /* 获取节点中 gpio 标号 */
126.    alinx_char.alinx_led_gpio = of_get_named_gpio(alinx_char.nd, "alinxled-gpios", 0);
127.    if(alinx_char.alinx_led_gpio < 0)
128.    {
129.        printk("can not get alinxled-gpios");
130.        return -EINVAL;
131.    }
132.    printk("alinxled-gpio num = %d\r\n", alinx_char.alinx_led_gpio);
133.
134.    /* 申请 gpio 标号对应的引脚 */
135.    gpio_request(alinx_char.alinx_led_gpio, "alinxled");
136.    if(ret != 0)
137.    {
138.        printk("can not request gpio\r\n");
139.    }
140.
141.    /* 把这个 io 设置为输出 */
142.    ret = gpio_direction_output(alinx_char.alinx_led_gpio, 1);
143.    if(ret < 0)
144.    {
145.        printk("can not set gpio\r\n");
146.    }
147.
148.    /* 注册设备号 */
149.    alloc_chrdev_region(&alinx_char.devid, MINOR1, DEVID_COUNT, DEVICE_NAME);
150.
151.    /* 初始化字符设备结构体 */
152.    cdev_init(&alinx_char.cdev, &ax_char_fops);
153.
154.    /* 注册字符设备 */
155.    cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
156.
157.    /* 创建类 */
158.    alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
159.    if(IS_ERR(alinx_char.class))
160.    {
161.        return PTR_ERR(alinx_char.class);
162.    }
163.
164.    /* 创建设备节点 */
165.    alinx_char.device = device_create(alinx_char.class, NULL,
166.                                      alinx_char.devid, NULL,
167.                                      DEVICE_NAME);
```

```
168.     if (IS_ERR(alinx_char.device))
169.     {
170.         return PTR_ERR(alinx_char.device);
171.     }
172. 
173.     return 0;
174. }
175. 
176. /* 卸载模块 */
177. static void __exit gpio_led_exit(void)
178. {
179.     /* 释放 gpio */
180.     gpio_free(alinx_char.alinx_led_gpio);
181. 
182.     /* 注销字符设备 */
183.     cdev_del(&alinx_char.cdev);
184. 
185.     /* 注销设备号 */
186.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
187. 
188.     /* 删除设备节点 */
189.     device_destroy(alinx_char.class, alinx_char.devid);
190. 
191.     /* 删除类 */
192.     class_destroy(alinx_char.class);
193. 
194.     printk("gpio_led_dev_exit_ok\n");
195. }
196. 
197. /* 标记加载、卸载函数 */
198. module_init(gpio_led_init);
199. module_exit(gpio_led_exit);
200. 
201. /* 驱动描述信息 */
202. MODULE_AUTHOR("Alinx");
203. MODULE_ALIAS("gpio_led");
204. MODULE_DESCRIPTION("PINCTRL AND GPIO LED driver");
205. MODULE_VERSION("v1.0");
206. MODULE_LICENSE("GPL");
```

与上一章驱动代码相比有却别的部分加粗了，主要是在入口函数中用 gpio 子系统获取 gpio 并初始化替换了原先获取寄存器再初始化的操作。

43 行在设备结构体中定义了 gpio 标号便于后续的获取操作。

55 行原先通过寄存器初始化 io 的操作删除了。

56 行在 open 函数中设置私有数据，之后在 write 函数中可以获取使用。

68 行获取私有数据。

79 和 **83** 行使用 gpio_set_value 函数来设置 gpio 的电平，用获取的私有数据应用 gpio 标号，第二个输入参数用!!二值化。

115 行获取/alinxled 节点。

127 行使用 of_get_named_gpio 函数，通过属性 alinxled-gpios 获取 gpio 标号。

136 行使用 gpio_request 函数申请标号对应的引脚。

143 行使用 gpio_direction_output 函数把 gpio 设置为输出。

181 行在卸载模块时，增加 gpio 释放操作。

4.3.4 测试代码

测试 APP 与章节 1.3.4 内容一致，可以直接使用。

4.3.1 运行测试

因为 APP 相同，所以测试方法任然相同，只要能成功点亮 led 就成功了。

```
root@ax_peta:/mnt# insmod ./ax-pinioled-dev.ko
ax_pinioled_dev: loading out-of-tree module taints kernel.
alinx_char node find
alinxled-gpio num = 899
root@ax_peta:/mnt#
```

```
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug# ./axleddev_test /dev/gpio_leds on
gpio_test module open
ps_led1 on
alinx led on
gpio_test module release
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug#
```