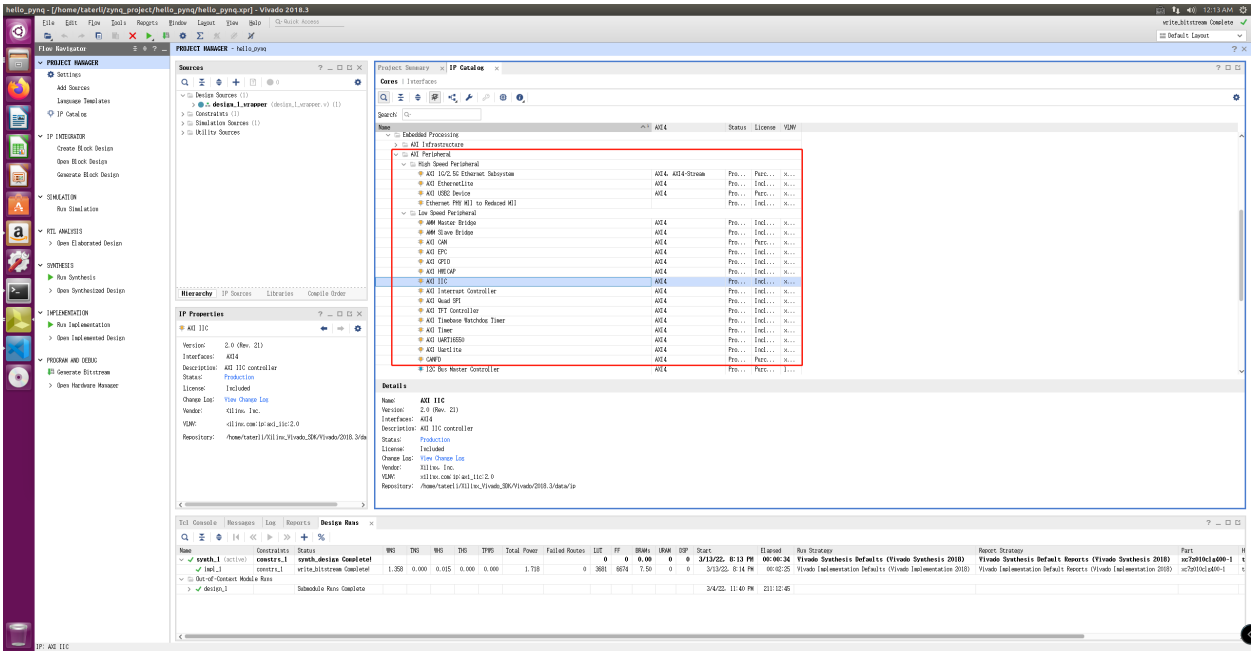
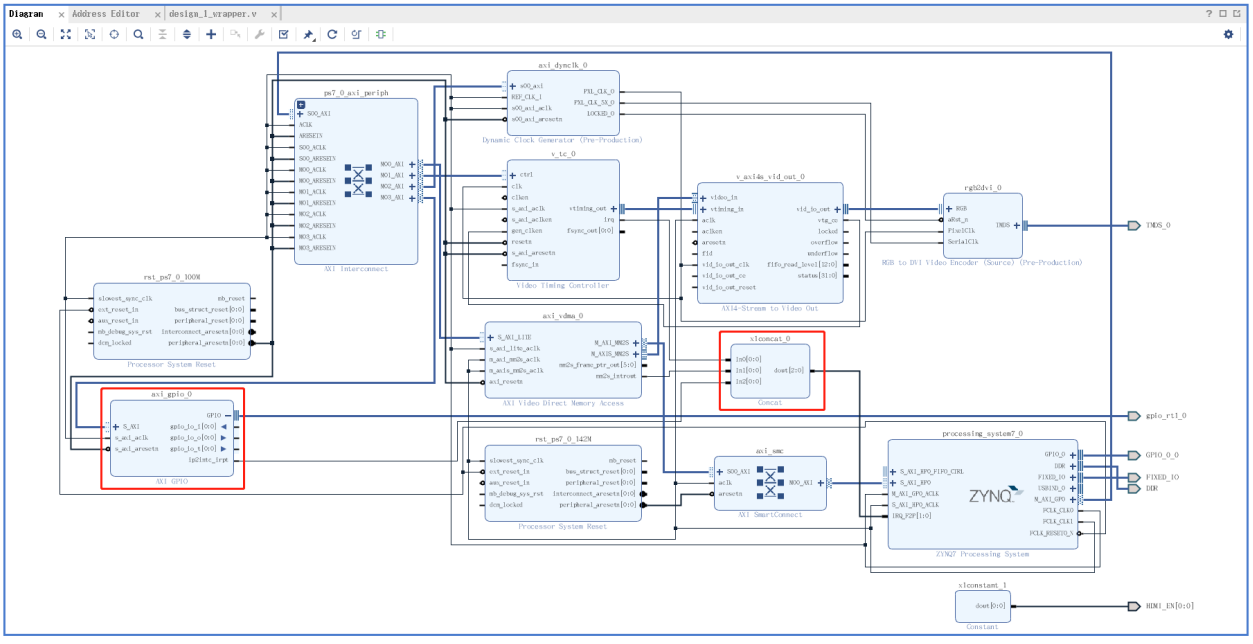
# [L27]Linux AXI 外设学习

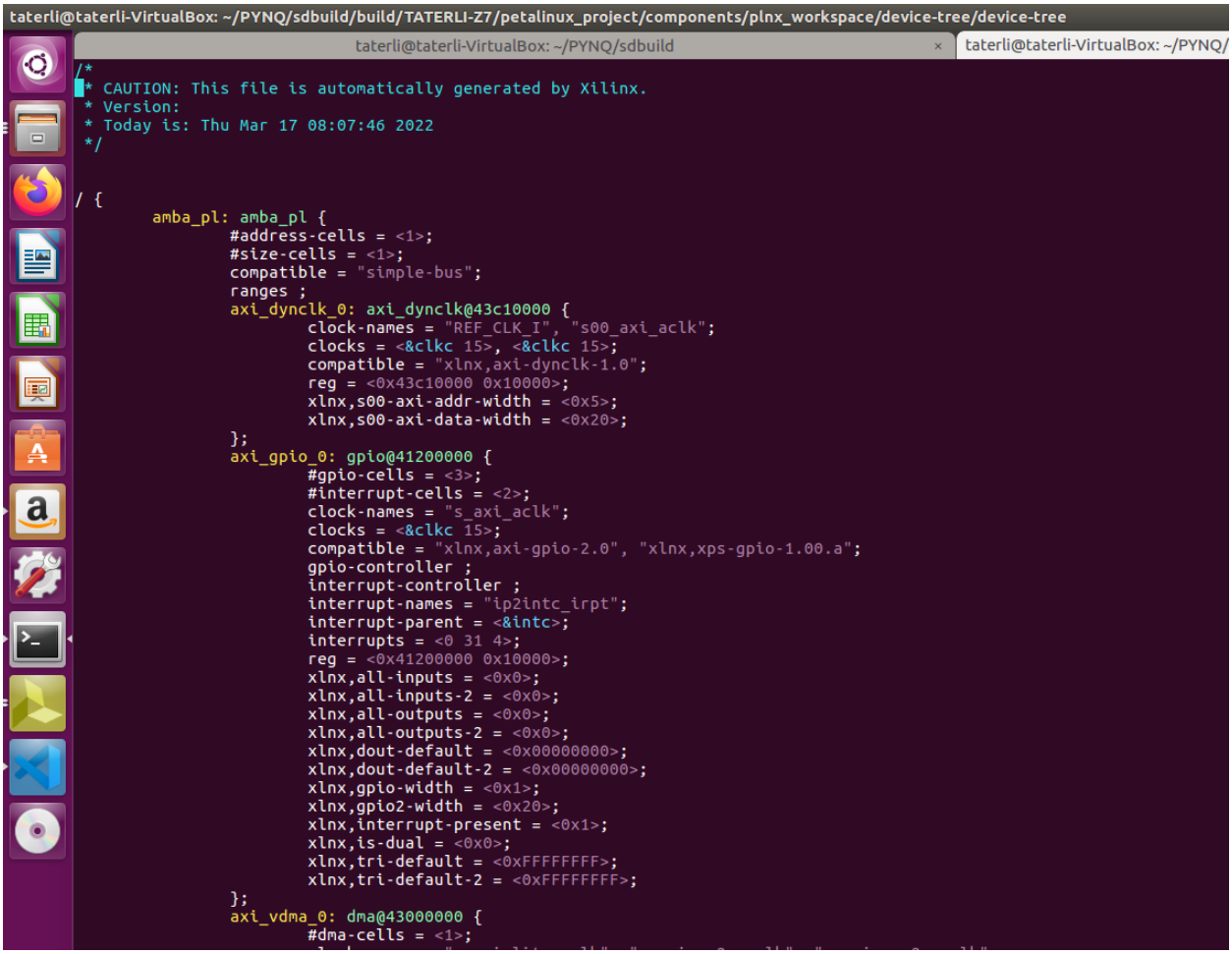
像ZYNQ这样的系统,除了硬核之外,还有软核,而像我的开发板上,PS一侧基本没什么外设,包括SPI,I2C等等都没有,我们还可以用Xilinx或者其他人提供的IP.



我先拖一个AXI GPIO出来,1个GPIO并且允许中断,然后修改xlconcat_0,让他可以输入3个中断,再把AXI GPIO也接入到中断管理.



然后按顺序绑定引脚,生成bitstream,更新BOOT等各种操作,一般来说,在验证他是否正常之前,最好先在SDK中试试,由于这个基础知识在一开始已经说过了,因此不再重复,最后通过hdf所生成的pcw.dtsi文件,会显示出新增的IP信息,文件自然是在PetaLinux工程目录内,不记得大致搜索一下就有.




AXI GPIO对应的内核绑定文档在Documentation/devicetree/bindings/gpio/gpio-xilinx.txt有提供,上面怎么配置明显是自动生成的,看下面怎么做.

```
Example to demonstrate how reset-gpios property is used in drivers:

driver: driver@80000000 {
  compatible = "xlnx,driver";
  reset-gpios = <&gpio 0 0 GPIO_ACTIVE_LOW>; /* gpio phandle, gpio pin-number, channel offset, flag state */
  reg = <0x0 0x80000000 0x0 0x10000>;
};
```

从AXI GPIO创建,然后配置dts测试,比如我这么测试,另外AXI GPIO比较残疾,只能支持上升沿.

```
sysled {
    compatible = "gpio-leds";
```

```
    heartbeat {
        label = "heartbeat";
        gpios = <&gpio0 54 0>;
        linux,default-trigger = "heartbeat";
    };

    cpu0_led {
        label = "cpu0_led";
        gpios = <&axi_gpio_0 0 0 GPIO_ACTIVE_HIGH>;
        linux,default-trigger = "cpu0";
    };
};

usrled {
    compatible = "taterli,led";
    status = "okay";
    default-state = "on";

    led-gpio = <&gpio0 55 GPIO_ACTIVE_HIGH>;
};

usrkey {
    compatible = "taterli,key";
    status = "okay";
    default-state = "on";

    key-gpio = <&axi_gpio_0 1 0 GPIO_ACTIVE_LOW>;
    interrupt-parent = <&axi_gpio_0>;
    interrupts = <1 IRQ_TYPE_EDGE_RISING>;
};
```

Xilinx还提供了很多标准外设,有低速有高速,也有付费才能用的,比如说我I2C的IP就是免费的,如果拖到设计里面,最后会生成如下的dts描述.

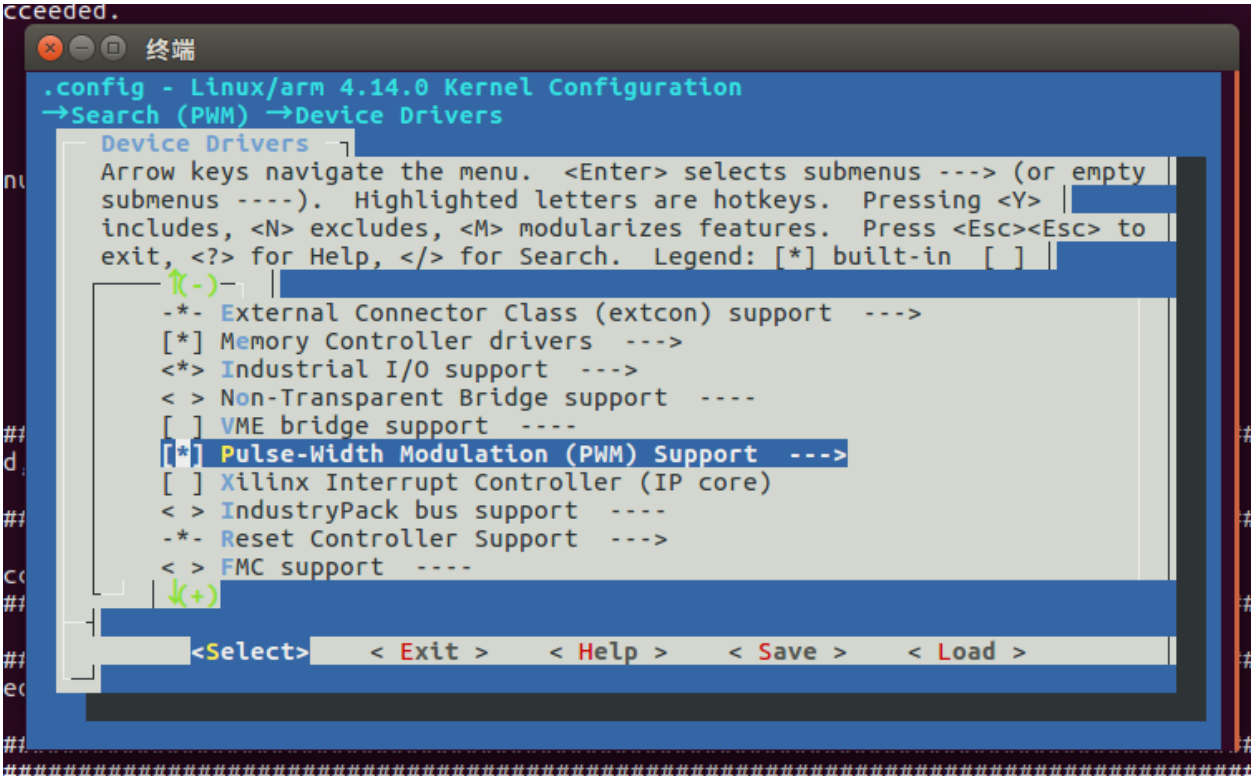
```
axi_iic_0: i2c@41600000 {
      #address-cells = <1>;
      #size-cells = <0>;
      clock-names = "s_axi_aclk";
      clocks = <&clkc 15>;
      compatible = "xlnx,axi-iic-2.0", "xlnx,xps-iic-2.00.a";
      interrupt-names = "iic2intc_irpt";
      interrupt-parent = <&intc>;
      interrupts = <0 32 4>;
      reg = <0x41600000 0x10000>;
};
```

他的使用就和普通的I2C外设基本是一样的,具体功能就看文档了,不过既然是一个半定制系统,那么我们还可以开发自己的外设,比如我们很早就做出来的PWM IP,这里也可以拖进来,生成如下的dts.

```
taterli_pwm_v1_0_0: taterli_pwm_v1_0@43c20000 {
      clock-names = "s00_axi_aclk";
      clocks = <&clkc 15>;
      compatible = "xlnx,taterli-pwm-v1-0-1.0";
      reg = <0x43c20000 0x10000>;
      xlnx,s00-axi-addr-width = <0x4>;
```

```
        xlnx,s00-axi-data-width = <0x20>;
};
```

打开内核的PWM支持,然后试着写一个最简单的驱动,主要是Platform注册,然后实现PWM需要的几个OP.




放驱动代码之前,我们重点关注一下两个结构体:

```
/**
 * struct pwm_chip - abstract a PWM controller
 * @dev: device providing the PWMs
 * @list: list node for internal use
 * @ops: callbacks for this PWM controller
 * @base: number of first PWM controlled by this chip
 * @npwm: number of PWMs controlled by this chip
 * @pwms: array of PWM devices allocated by the framework
 * @of_xlate: request a PWM device given a device tree PWM specifier
 * @of_pwm_n_cells: number of cells expected in the device tree PWM specifier
 */
struct pwm_chip {
  struct device *dev;
  struct list_head list;
  const struct pwm_ops *ops;
  int base;
  unsigned int npwm;

  struct pwm_device *pwms;

  struct pwm_device * (*of_xlate)(struct pwm_chip *pc,
          const struct of_phandle_args *args);
```

```
  unsigned int of_pwm_n_cells;
};

/**
 * struct pwm_ops - PWM controller operations
 * @request: optional hook for requesting a PWM
 * @free: optional hook for freeing a PWM
 * @config: configure duty cycles and period length for this PWM
 * @set_polarity: configure the polarity of this PWM
 * @capture: capture and report PWM signal
 * @enable: enable PWM output toggling
 * @disable: disable PWM output toggling
 * @apply: atomically apply a new PWM config. The state argument
 *     should be adjusted with the real hardware config (if the
 *     approximate the period or duty_cycle value, state should
 *     reflect it)
 * @get_state: get the current PWM state. This function is only
 *         called once per PWM device when the PWM chip is
 *         registered.
 * @dbg_show: optional routine to show contents in debugfs
 * @owner: helps prevent removal of modules exporting active PWMs
 */
struct pwm_ops {
  int (*request)(struct pwm_chip *chip, struct pwm_device *pwm);
  void (*free)(struct pwm_chip *chip, struct pwm_device *pwm);
  int (*config)(struct pwm_chip *chip, struct pwm_device *pwm,
          int duty_ns, int period_ns);
  int (*set_polarity)(struct pwm_chip *chip, struct pwm_device *pwm,
          enum pwm_polarity polarity);
  int (*capture)(struct pwm_chip *chip, struct pwm_device *pwm,
           struct pwm_capture *result, unsigned long timeout);
  int (*enable)(struct pwm_chip *chip, struct pwm_device *pwm);
  void (*disable)(struct pwm_chip *chip, struct pwm_device *pwm);
  int (*apply)(struct pwm_chip *chip, struct pwm_device *pwm,
         struct pwm_state *state);
  void (*get_state)(struct pwm_chip *chip, struct pwm_device *pwm,
        struct pwm_state *state);
#ifdef CONFIG_DEBUG_FS
  void (*dbg_show)(struct pwm_chip *chip, struct seq_file *s);
#endif
  struct module *owner;
};
```

只要实现上面两个结构体就可以,我这里只实现了必要的函数,毕竟我的PWM模块也太简单了点.

```
/*************************************************************
 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
 文件名    : pwm-dglnt.c
 作者      : 邓涛
 版本      : V1.0
 描述      : Digilent AXI_PWM驱动程序
 其他      : 无
 论坛      : www.openedv.com
 日志      : 初版V1.0 2020/7/9 邓涛创建
 *************************************************************/

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/pwm.h>
```

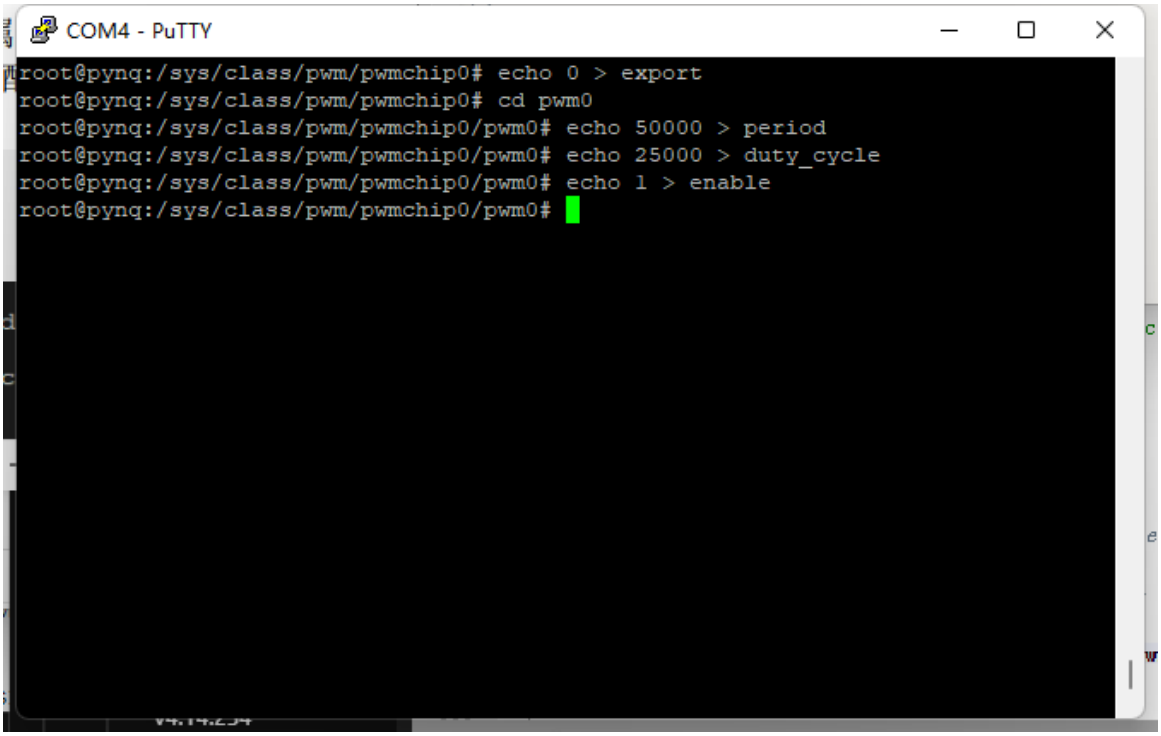
```
#include <linux/io.h>
#include <linux/clk.h>

/* 寄存器定义 */
#define PWM_BASE 0
#define PWM_CCR 4
#define PWM_EN 8
#define PWM_ID 12

struct tl_pwm_dev
{
    struct device *dev;         /* 通用platform驱动要的东西 */
    struct pwm_chip chip;       /* 用户pwm driver的 */
    void __iomem *reg;          /* 寄存器映射 */
    unsigned int period_min_ns; /* PWM 最小周期 */
};

#define S_TO_NS 1000000000U /* 秒换算成纳秒的量级单位 */

/* 这里是主要配置函数,传入的是周期和占空比的时间,所有要依赖初始化时候计算的周期. */
static int tl_pwm_config(struct pwm_chip *chip, struct pwm_device *pwm,
                         int duty_ns, int period_ns)
{
    int duty, period;
    struct tl_pwm_dev *tl_pwm = container_of(chip, struct tl_pwm_dev, chip);

    if (tl_pwm->period_min_ns > period_ns)
        period_ns = tl_pwm->period_min_ns;

    period = period_ns / tl_pwm->period_min_ns;
    duty = duty_ns / tl_pwm->period_min_ns;

    printk(KERN_INFO "tl_pwm period=%d duty=%d\n", period, duty);

    writel(period, tl_pwm->reg + PWM_BASE);
    writel(duty, tl_pwm->reg + PWM_CCR);
    return 0;
}

/* 使能PWM */
static int tl_pwm_enable(struct pwm_chip *chip, struct pwm_device *pwm)
{
    struct tl_pwm_dev *tl_pwm = container_of(chip, struct tl_pwm_dev, chip);
    writel(1, tl_pwm->reg + PWM_EN);
    return 0;
}

/* 禁止PWM */
static void tl_pwm_disable(struct pwm_chip *chip, struct pwm_device *pwm)
{
    struct tl_pwm_dev *tl_pwm = container_of(chip, struct tl_pwm_dev, chip);
    writel(0, tl_pwm->reg + PWM_EN);
}

static const struct pwm_ops tl_pwm_ops = {
    .config = tl_pwm_config,
    .enable = tl_pwm_enable,
    .disable = tl_pwm_disable,
    .owner = THIS_MODULE,
};

static int tl_pwm_probe(struct platform_device *pdev)
{
```

```
    struct tl_pwm_dev *tl_pwm;
    struct clk *clk;
    unsigned long clk_rate;
    struct resource *res;
    int ret;
    int id;

    /* 实例化一个tl_pwm_dev对象 */
    tl_pwm = devm_kzalloc(&pdev->dev, sizeof(*tl_pwm), GFP_KERNEL);
    if (!tl_pwm)
        return -ENOMEM;

    tl_pwm->dev = &pdev->dev;

    /* 获取寄存器位置和长度信息. */
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    tl_pwm->reg = devm_ioremap_resource(&pdev->dev, res);
    if (IS_ERR(tl_pwm->reg))
        return PTR_ERR(tl_pwm->reg);

    /* 判断外设ID */
    id = readl(tl_pwm->reg + PWM_ID);
    printk(KERN_INFO "tl_pwd id = 0x%08x\n", id);

    /* 获取时钟为了获取到这个模块的TIM BASE频率. */
    clk = devm_clk_get(&pdev->dev, "s00_axi_aclk");
    if (IS_ERR(clk))
    {
        dev_err(&pdev->dev, "failed to get pwm clock\n");
        return PTR_ERR(clk);
    }

    clk_rate = clk_get_rate(clk);

    /* 计算PWM的最小周期 */
    tl_pwm->period_min_ns = S_TO_NS / clk_rate;
    printk(KERN_INFO "tl_pwm clk=%ld  period_min_ns=%d\n",
           clk_rate, tl_pwm->period_min_ns);

    /* 注册PWM设备 */
    tl_pwm->chip.dev = &pdev->dev;
    tl_pwm->chip.ops = &tl_pwm_ops;
    tl_pwm->chip.base = 0;
    tl_pwm->chip.npwm = 1;

    ret = pwmchip_add(&tl_pwm->chip);
    if (ret < 0)
    {
        dev_err(&pdev->dev, "pwmchip_add failed: %d\n", ret);
        return ret;
    }

    platform_set_drvdata(pdev, tl_pwm);
    return 0;
}

static int tl_pwm_remove(struct platform_device *pdev)
{
    struct tl_pwm_dev *tl_pwm = platform_get_drvdata(pdev);

    /* 禁止PWM输出 */
    writel(0, tl_pwm->reg + PWM_EN);
```

```
    /* 卸载PWM设备 */
    return pwmchip_remove(&tl_pwm->chip);
}

static const struct of_device_id tl_pwm_of_match[] = {
    {
        .compatible = "xlnx,taterli-pwm-v1-0-1.0",
    },
    {},
};
MODULE_DEVICE_TABLE(of, tl_pwm_of_match);

static struct platform_driver tl_pwm_driver = {
    .driver = {
        .name = "taterli-pwm-v1-0-1.0",
        .of_match_table = tl_pwm_of_match,
    },
    .probe = tl_pwm_probe,
    .remove = tl_pwm_remove,
};

module_platform_driver(tl_pwm_driver);

MODULE_AUTHOR("Deng Tao <773904075@qq.com>");
MODULE_DESCRIPTION("Simple Driver for Digilent AXI_PWM IP Core");
MODULE_LICENSE("GPL v2");
```

然后就可以在用户空间开心操作了.

```
root@pynq:/sys/class/pwm/pwmchip0# echo 0 > export
root@pynq:/sys/class/pwm/pwmchip0# cd pwm0
root@pynq:/sys/class/pwm/pwmchip0/pwm0# echo 50000 > period
root@pynq:/sys/class/pwm/pwmchip0/pwm0# echo 25000 > duty_cycle
root@pynq:/sys/class/pwm/pwmchip0/pwm0# echo 1 > enable
root@pynq:/sys/class/pwm/pwmchip0/pwm0#
```




虽然例子很简单,但是再复杂的外设其实也是同样的方法做出来的,大家可以从这里先暂停下来,尝试实现一些手边的东西的驱动.