7 定时器

本章简单介绍 Linux 内核提供的定时器及其用法。

7.1 Linux 内核中的定时器

内核中定时器的实现依赖于硬件定时器。硬件定时器提供固定频率的时钟源并产生中断，系统使用这个中断来计时。而内核中的定时器功能，则是在这个计时的基础上实现的。

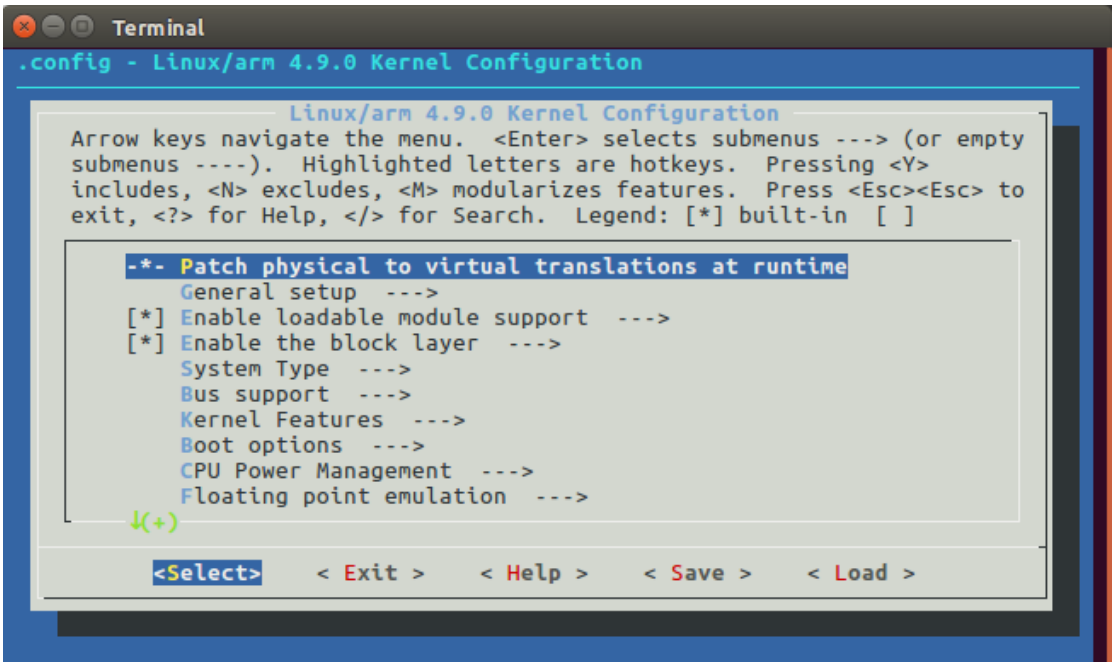
7.1.1 系统频率

产生系统中断的频率就是系统频率，也叫系统节拍即每秒的节拍数。系统节拍是可设置的，在 petalinux 中设置方法为：

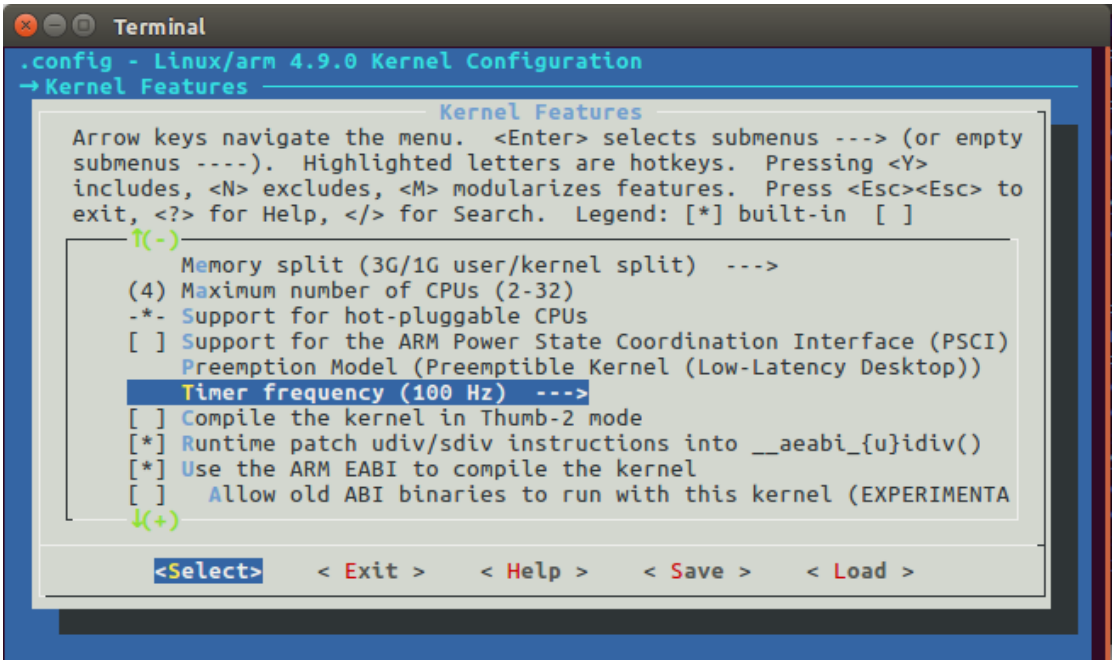
在 petalinux 工程目录中打开终端输入命令：

```
petalinux-config -c kernel
```

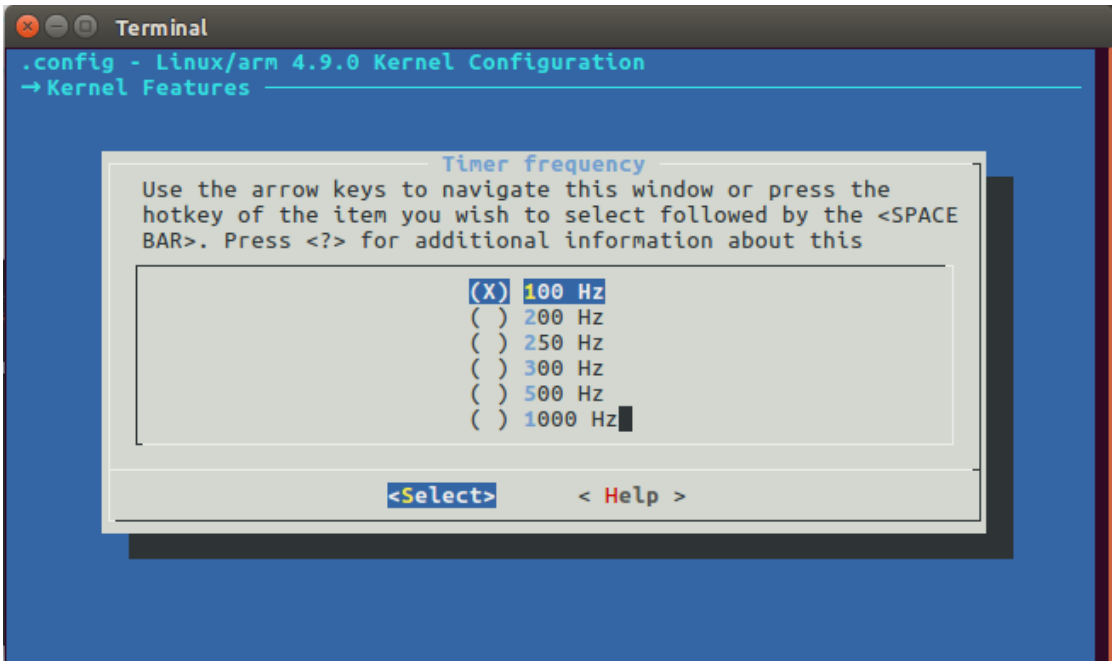
等待一段时间后，会弹出交互界面如下图：



用上下方向键移动选项置”Kernel Features”，按回车进入，如下图：



再按下方向键选择选项”Timer frequency(100 Hz)”按回车(这个选项被藏在很下面，不在一开始的界面里)，就能看到系统频率的选项：



这里默认选择的是 100Hz，每秒 100 个节拍。

系统频率越高，时钟精度也越高，100Hz 的时间进度能达到 10ms，而 1000Hz 能达到 1ms。同时中断产生的频率增加，cpu 的负担也就越大这需要根据实际情况去取舍，我们的实验使用 100Hz 就足够了。

系统时钟定义在 include/asm-generic/param.h 中，为宏定义 HZ，他最终会对应到我们设置的系统时钟值。

7.1.2 节拍数

系统计时的方式就是记录节拍数。系统运行的时间=节拍数/系统频率，单位为秒。

内核中用全局变量 jiffies 来记录节拍书，定义在文件 include/linux/jiffies.h 中：

```
extern u64 __jiffy_data jiffies_64;
extern unsigned long volatile __jiffy_data jiffies;
```

jiffies_64 用于 64 位的系统，jiffies 用于 32 位系统。既然是用来计数的变量，就会有溢出的风险。64 位且不提，32 位的 jiffies 在 100Hz 的系统频率下，500 天左右就会溢出回绕，在 1000Hz 的频率下 50 天左右就会回绕。回绕会带来什么问题，举个例子：

```
1.  unsigned long time_mark = jiffies + HZ * 5;
2.
3.  /* 业务处理 */
4.
5.  if(time_mark > jiffies)
6.  {
7.      /* 未超时 */
8.  }
9.  else
10. {
11.     /* 超时 */
12. }
```

这个例子中，先设定一个时间 time_mark 为当前节拍数再加上 5 秒的节拍，也就是 5 秒后系统节拍 jiffies 会到达的值。这个示例代码的目的是如果业务处理的时间超过 5 秒就执行超时的处理。极端情况下，比如此时 jiffies 为 0xFFFFFDA7，time_mark 则等于 0xFFFFFF9B，如果业务处理时间用了 7 秒，jiffies 会因为超过最大值 0xFFFFFFFF 发生反转，从 0 开始重新计数变成 0x64，这个值远远小于 time_mark，明明超时了，却进入了未超时的条件。

对此，内核的通过把数据强专无符号数强转成有符号数巧妙的规避了回绕的问题，不用传统的比较运算符去比较，而使用下面几个宏来代替：

```
#define time_after(unknown,known) ((long)(known) - (long)(unknown)<0)
#define time_before(unkonwn,known) ((long)(unknown) - (long)(known)<0)
#define time_after_eq(unknown,known) ((long)(unknown) - (long)(known)>=0)
#define time_before_eq(unknown,known) ((long)(known) -(long)(unknown)>=0)
```

以 time_after_eq 为例，参数 unknow 一般指 jiffies，know 通常是我们设定的时间，如上面例子中的 time_mark。after_eq 即大于等于，如果 unknow 大于等于 know，即 jiffies 大于 time_mark，则返回真，对于上面的例程来说就是超时了。

7.1.3 内核定时器及相关函数

Linux 内核中定时器用数据结构 timer_list 来表示。定义在 include/linux/timer.h 中：

```
1.   struct timer_list {
2.     /*
3.      * All fields that change during normal runtime grouped to the
4.      * same cacheline
5.      */
```

```
6.      struct hlist_node   entry;
7.      unsigned long       expires;
8.      void            (*function)(unsigned long);
9.      unsigned long       data;
10.     u32         flags;
11.
12. #ifdef CONFIG_TIMER_STATS
13.     int         start_pid;
14.     void            *start_site;
15.     char            start_comm[16];
16. #endif
17. #ifdef CONFIG_LOCKDEP
18.     struct lockdep_map  lockdep_map;
19. #endif
20. }
```

重要的成员介绍：

expires：设置定时器超时时间点的节拍数，如果我们需要定义一个 10s 的定时器，那 expires 就应该等于 jiffies+10*HZ。为了方便，内核还提供了毫秒、微妙、纳秒和 jiffies 互换的系列函数：

```
/* 毫秒互换 */
int jiffies_to_msecs(const unsigned long j)
long msecs_to_jiffies(const unsigned int m)
/* 微秒互换 */
int jiffies_to_usecs(const unsigned long j)
long usecs_to_jiffies(const unsigned int u)
/* 纳秒互换 */
u64 jiffies_to_nsecs(const unsigned long j)
unsigned long nsecs_to_jiffies(u64 n)
```

function：定时器超回掉函数指针。超时后的需要做的操作就在这个函数中实现。

data：用于设置 function 的输入参数。

定时器相关的函数有：

1. void init_timer(struct timer_list *timer)

初始化定时器，输入参数 timer 即需要初始化的定时器。

2. void add_timer(struct timer_list *timer)

向内核注册定时器，注册后定时器就开始运行，输入参数 timer 即需要注册的定时器。

3. int del_timer(struct timer_list * timer)

删除定时器，不管定时器是否正在运行，正在运行的 timer 会直接停止计时。输入参数 timer 即需要删除的定时器。删除的定时器若正在运行，则返回 1，否则返回 0。

4. int del_timer_sync(struct timer_list *timer)

删除定时器，正在运行的 timer 会等到使用完成后才删除。输入参数 timer 即需要删除的定时器。删除的定时器若正在运行，则返回 1，否则返回 0。

5. int mod_timer(struct timer_list *timer, unsigned long expires)

修改定时器的 expires 即超时时间，如果定时器不在运行，则激活该定时器。输入参数

timer 为需要修改的定时器。expires 即为修改后的超时时间。

7.2 实验

实验目标是，使用定时器定让 led 闪烁，应用程序可以通过输入参数设置 led 闪烁的周期。

7.2.1 原理图

和章节 1.3.1 的内容相同。

7.2.2 设备树

和章节 4.3.2 相同。

7.2.3 驱动程序

使用 petalinux 新建驱动名为”ax-timer-dev”，在 ax- timer -dev 中输入以下代码：

```
1.    #include <linux/module.h>
2.  #include <linux/kernel.h>
3.  #include <linux/fs.h>
4.  #include <linux/init.h>
5.  #include <linux/ide.h>
6.  #include <linux/types.h>
7.  #include <linux/errno.h>
8.  #include <linux/cdev.h>
9.  #include <linux/of.h>
10. #include <linux/of_address.h>
11. #include <linux/of_gpio.h>
12. #include <linux/device.h>
13. #include <linux/delay.h>
14. #include <linux/init.h>
15. #include <linux/gpio.h>
16. #include <linux/semaphore.h>
17. #include <linux/timer.h>
18. #include <asm/uaccess.h>
19. #include <asm/mach/map.h>
20. #include <asm/io.h>
21.
22. /* 设备节点名称 */
23. #define DEVICE_NAME       "timer_led"
```

```
/* 设备号个数 */
#define DEVID_COUNT       1
/* 驱动个数 */
#define DRIVE_COUNT       1
/* 主设备号 */
#define MAJOR_U
/* 次设备号 */
#define MINOR_U           0


/* 把驱动代码中会用到的数据打包进设备结构体 */
struct alinx_char_dev{
    dev_t              devid;             //设备号
    struct cdev        cdev;              //字符设备
    struct class       *class;            //类
    struct device      *device;           //设备
    struct device_node *nd;               //设备树的设备节点
    int                alinx_led_gpio;    //gpio 号
    char               led_status;        //gpio 状态
    unsigned int       time_count;        //定时器时间
    struct timer_list  timer;             //定时器
};
/* 声明设备结构体 */
static struct alinx_char_dev alinx_char = {
    .cdev = {
        .owner = THIS_MODULE,
    },
};

void timer_function(unsigned long data)
{
    /* 反转 led 状态 */
    alinx_char.led_status = !alinx_char.led_status;
    /* 设置 led */
    gpio_set_value(alinx_char.alinx_led_gpio, alinx_char.led_status);
    /* 重新开始计时 */
    mod_timer(&alinx_char.timer, jiffies + msecs_to_jiffies(alinx_char.time_count));
}

/* open 函数实现，对应到 Linux 系统调用函数的 open 函数 */
static int timer_led_open(struct inode *inode_p, struct file *file_p)
{
    printk("gpio_test module open\n");
    return 0;
```

```
68. }
69.
70.
71. /* write 函数实现，对应到 Linux 系统调用函数的 write 函数 */
72. static ssize_t timer_led_write(struct file *file_p, const char __user *buf, size_t len, lof
    f_t *loff_t_p)
73. {
74.     int retvalue;
75.     /* 获取用户数据 */
76.     retvalue = copy_from_user(&alinx_char.time_count, buf, len);
77.     /* 设置好 timer 后先点亮 led */
78.     alinx_char.led_status = 1;
79.     gpio_set_value(alinx_char.alinx_led_gpio, alinx_char.led_status);
80.     /* 开启 timer */
81.     mod_timer(&alinx_char.timer, jiffies + msecs_to_jiffies(alinx_char.time_count));
82.
83.     return 0;
84. }
85.
86. /* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
87. static int timer_led_release(struct inode *inode_p, struct file *file_p)
88. {
89.     printk("gpio_test module release\n");
90.     /* 删除定时器 */
91.     del_timer_sync(&alinx_char.timer);
92.     return 0;
93. }
94.
95. /* 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 */
96. static struct file_operations ax_char_fops = {
97.     .owner   = THIS_MODULE,
98.     .open    = timer_led_open,
99.     .write   = timer_led_write,
100.    .release = timer_led_release,
101. };
102.
103. /* 模块加载时会调用的函数 */
104. static int __init timer_led_init(void)
105. {
106.     /* 用于接受返回值 */
107.     u32 ret = 0;
108.
109.     /* 获取 led 设备节点 */
110.     alinx_char.nd = of_find_node_by_path("/alinxled");
```

```
111.    if(alinx_char.nd == NULL)
112.    {
113.        printk("alinx_char node not find\r\n");
114.        return -EINVAL;
115.    }
116.    else
117.    {
118.        printk("alinx_char node find\r\n");
119.    }
120.
121.    /* 获取节点中 gpio 标号 */
122.    alinx_char.alinx_led_gpio = of_get_named_gpio(alinx_char.nd, "alinxled-gpios", 0);
123.    if(alinx_char.alinx_led_gpio < 0)
124.    {
125.        printk("can not get alinxled-gpios");
126.        return -EINVAL;
127.    }
128.    printk("alinxled-gpio num = %d\r\n", alinx_char.alinx_led_gpio);
129.
130.    /* 申请 gpio 标号对应的引脚 */
131.    ret = gpio_request(alinx_char.alinx_led_gpio, "alinxled");
132.    if(ret != 0)
133.    {
134.        printk("can not request gpio\r\n");
135.    }
136.
137.    /* 把这个 io 设置为输出 */
138.    ret = gpio_direction_output(alinx_char.alinx_led_gpio, 1);
139.    if(ret < 0)
140.    {
141.        printk("can not set gpio\r\n");
142.    }
143.
144.    /* 注册设备号 */
145.    alloc_chrdev_region(&alinx_char.devid, MINOR_U, DEVID_COUNT, DEVICE_NAME);
146.
147.    /* 初始化字符设备结构体 */
148.    cdev_init(&alinx_char.cdev, &ax_char_fops);
149.
150.    /* 注册字符设备 */
151.    cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
152.
153.    /* 创建类 */
154.    alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
```

```
155.     if(IS_ERR(alinx_char.class))
156.     {
157.         return PTR_ERR(alinx_char.class);
158.     }
159.
160.     /* 创建设备节点 */
161.     alinx_char.device = device_create(alinx_char.class, NULL,
162.                                       alinx_char.devid, NULL,
163.                                       DEVICE_NAME);
164.     if (IS_ERR(alinx_char.device))
165.     {
166.         return PTR_ERR(alinx_char.device);
167.     }
168.
169.     /* 设置定时器回掉函数  */
170.     alinx_char.timer.function = timer_function;
171.     /* 初始化定时器 */
172.     init_timer(&alinx_char.timer);
173.     return 0;
174. }
175.
176. /* 卸载模块 */
177. static void __exit timer_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("timer_led_dev_exit_ok\n");
195. }
196.
197. /* 标记加载、卸载函数 */
198. module_init(timer_led_init);
```

```
199. module_exit(timer_led_exit);
200.
201. /* 驱动描述信息 */
202. MODULE_AUTHOR("Alinx");
203. MODULE_ALIAS("gpio_led");
204. MODULE_DESCRIPTION("TIMER LED driver");
205. MODULE_VERSION("v1.0");
206. MODULE_LICENSE("GPL");
```

同样关注加粗的部分，其他都是常规的字符设备注册框架。

42 行定义一个 char 型数来记录 led 的状态。

43 行定义一个 unsigned int 来记录定时器的时间。

44 行定义一个定时器。

53~61 行是定时器的回掉函数，定时器计时结束后就会执行这个函数。在这个函数中进行一次 led 的状态反转，并重新开始计时。

72 行的 open 函数中，根据应用程序输入的值来设置 timer 的计时时间，并开启 timer，使用 mod_timer 函数。

91 行的 release 函数中删除 timer。

170 行在入口函数中，设置定时器的回掉函数为 timer_function()，并初始化定时器。

7.2.4 测试程序

新建 QT 工程名为”timer-test”，新建 main.c，输入下列代码：

```
1.   #include "stdio.h"
2.  #include "unistd.h"
3.  #include "sys/types.h"
4.  #include "sys/stat.h"
5.  #include "fcntl.h"
6.  #include "stdlib.h"
7.  #include "string.h"
8.  #include "linux/ioctl.h"
9.
10. int main(int argc, char *argv[])
11. {
12.     int fd, ret;
13.     char *filename;
14.     unsigned int interval_new, interval_old = 0;
15.
16.     if(argc != 2)
17.     {
18.         printf("Error Usage!\r\n");
19.         return -1;
20.     }
```

```
21.
22.     filename = argv[1];
23.
24.     fd = open(filename, O_RDWR);
25.     if(fd < 0)
26.     {
27.         printf("can not open file %s\r\n", filename);
28.         return -1;
29.     }
30.
31.     while(1)
32.     {
33.         printf("Input interval:");
34.         scanf("%d", &interval_new);
35.
36.         if(interval_new != interval_old)
37.         {
38.             interval_old = interval_new;
39.             ret = write(fd, &interval_new, sizeof(interval_new));
40.             if(ret < 0)
41.             {
42.                 printf("write failed\r\n");
43.             }
44.             else
45.             {
46.                 printf("interval refreshed!\r\n");
47.             }
48.         }
49.         else
50.         {
51.             printf("same interval!");
52.         }
53.     }
54.     close(fd);
55. }
```

内容很简单。

34 行等待用户输入时间间隔。

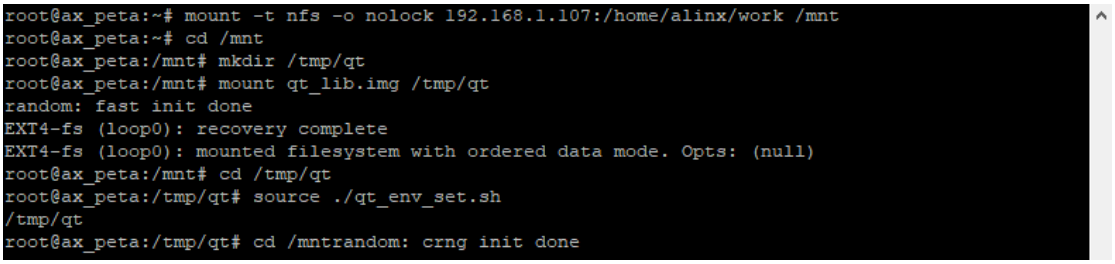
36 行判断如果时间间隔没有变化就忽略。如果有变化则调用 write 函数用新的间隔重新开启 timer。

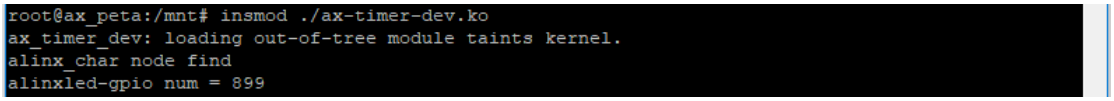
7.3.5 运行测试

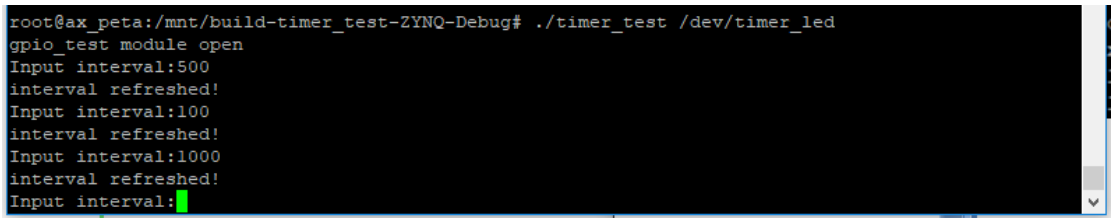
测试步骤如下：

```
mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt //IP 和路径根据实际情况调整

cd /mnt

mkdir /tmp/qt

mount qt_lib.img /tmp/qt

cd /tmp/qt

source ./qt_env_set.sh

cd /mnt

./ax-timer-dev.ko

cd ./build-timer_test-ZYNQ-Debug/ //app 目录可能不一样， 根据实际情况调整

./timer_test /dev/timer_led
```

结果如下图：







运行 app 后，输入不同的时间间隔，观察 led 的变化。