

petalinux

1. create project

```
source /xilinx/settings.sh

petalinux-create -t project --template zynq -n xxx_name
petalinux-config --get-hw-description .
```

2. create modules

相当于创建linux模块驱动

```
petalinux-create -t modules -n xxx_module --enable
```

3. create app

创建一个linux平台下的app可执行程序

```
petalinux-create -t apps -n xxx_app --enable
```

然后进入recipes-apps/xxx_app目录，修改files/xxx_app.c文件，默认是一个hello world!打印程序。

4. build

```
petalinux-build -c rootfs
petalinux-build -c xxx_module
petalinux-build -c xxx_app -x do_install
```

```
petalinux-build
```

5. app单独编译&调试

```
petalinux-build -c xxx_app -x do_compile # 对指定app进行编译, 编译后生成的app可执行文件在以下位置
ls build/tmp/work/cortexa9hf-neon-xilinx-linux-gnueabi/myapp3/1.0-r0/myapp3

# 然后将此文件通过scp的方式copy到zynq设备, 修改权限chmod 777 ./myapp3 执行程
```

```
序./myapp3
```

```
# 这样的方式不用每次app修改后都烧写到flash 通过scp的方式加快app开发调试速度。
```

6. package to BOOT.BIN

```
petalinux-package --boot --fsbl zynq_fsbl.elf --u-boot --kernel --fpga
system.bit --force
```

```
# --boot 打包成BOOT.BIN文件
```

```
# -- 输入fsdb文件
```

```
# --u-boot 输入默认u-boot文件，一般是指u-boot.elf
```

```
# --kernel 输入petalinux内核，默认是指image.ub
```

```
# --fpga 指定FPGA bits文件。
```

```
#一般是建议先不用--kernel选项生成BOOT.BIN文件，然后将BOOT.BIN文件和image.ub一起copy到SD卡，从SD卡启动看看程序效果，程序稳定后可以再考虑弄成为flash启动，这样调试速度会快一些。
```

7. 一个简单的驱动开发例程——GPIO流水灯(Petalinux部分)

<https://blog.csdn.net/u013029731/article/details/85042431/>

8. 实现app开机启动

参考ug1144, Ch.7: Customizing the Rootfs

参考：https://blog.csdn.net/qq_37775990/article/details/126951572

实现开机启动

本章节内容参考UG1144

1) 创建myapp-init应用

```
cd <plnx-proj-root>
```

```
petalinux-create -t apps --template install -n myapp-init --enable
```

2) 修改myapp-init.bb配置文件

配置文件的位置在：

```
project-spec/meta-user/recipes-apps/myapp-init/myapp-init.bb
```

修改文件内容为：

```
#
```

```
# This file is the myapp-init recipe.
```

```
#
```

```
SUMMARY = "Simple myapp-init application"
```

```
SECTION = "PETALINUX/apps"
```

```
LICENSE = "MIT"
```

```
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/"
```

```
MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"
SRC_URI = "file://myapp-init \
"
S = "${WORKDIR}"
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
inherit update-rc.d
INITSCRIPT_NAME = "myapp-init"
INITSCRIPT_PARAMS = "start 99 S ."
do_install() {
    install -d ${D}${sysconfdir}/init.d
    install -m 0755 ${S}/myapp-init ${D}${sysconfdir}/init.d/myapp-init
}
FILES_${PN} += "${sysconfdir}/*"
```

□3) 修改myapp-init脚本文件内容
脚本文件的位置在：

```
project-spec/meta-user/recipes-apps/
myapp-init/files/myapp-init
```

本文修改的内容为加载xilinx-axidma.ko module和在后台启动程序xxx_app

```
#!/bin/sh
```

```
cd /lib/modules/5.4.0-xilinx-v2020.1/extra
insmod xxx_module.ko
```

```
cd /usr/bin
./xxx_app &
```

注意，最好在此加上&，让其后台执行，不然系统会把当前命令执行完之后再进行下面的任务，
比如下面的任务是配置网络，这对于需要网络的开机启动尤其重要

完成后进行petalinux-build□使用新生成的镜像，下次就可以开机自启动了。

开机启动执行log example:

```
Hello PetaLinux World, startup test....
blink: loading out-of-tree module taints kernel.
<1>Hello module world.
<1>Module parameters were (0xdeadbeef) and "default"
blink_init: Registers mapped to mmio = 0xf09d0000
Registration is a success the major device number is 245.
If you want to talk to the device driver,
create a device file by following command.

mknod /dev/blink_Dev c 245 0

The device file name is important, because
the ioctl program assumes that's the file you'll use
#####
    Blink LED Application device_open(c7e4c6c0)
```

```
#####
*****
start LED sparkle...
*****
app led on
KERNEL PRINT : set_blink_ctrl

app led off
KERNEL PRINT : reset_blink_ctrl

app led on
KERNEL PRINT : set_blink_ctrl

app led off
KERNEL PRINT : reset_blink_ctrl

device_release(ef1d9370,c7e4c6c0)
#####
INIT: Entering runlevel: 5ation
Configuring network interfaces... IPv6: ADDRCONF(NETDEV_UP): eth0: link is
not ready
udhcpd (v1.24.1) started
Sending discover...
Sending discover...
macb e000b000.ethernet eth0: link up (1000/Full)
IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
Sending discover...
Sending select for 192.168.0.166...
Lease of 192.168.0.166 obtained, lease time 268435455
/etc/udhcpd.d/50default: Adding DNS 192.168.0.1
done.
Starting Dropbear SSH server: Generating key, this may take a while...
Public key portion is:
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQCh7/mYSvw3pyvt0K//+5A2N2sIGUoo7ZXjChqDaBD/iV8M
DrVQiGiyyXmxkUlcrgKVeWzP
NQ0Q4i58cvtTIIJmeEQI3rM9WD7V+o/HcDRr8TlyIbK0S0KHCcHpglKwm54e01evpmX2tt4cdXfx
dnaRhc0NjPMJq4cAutFu0y085QGwsVQ1
0FJYrdmt4Rc5T0BZdF3LtuXaV0F7mJ7aavI7vpsme1JIvzA0kGMSXH+HqY2wG4Ak6D07WdH78AaQ
sI86vDLA1WLaPP4oCMjiLjeFKIuAbnhl
d+HuJtwvj5fx4GZcRyQ5VrwVE7anQmAu40lw/09dwQQTAJ0LX1u6Ui0V
root@petalinux_boot_from_flash
Fingerprint: md5 61:94:fc:ff:b1:94:85:d7:fa:19:26:a9:a4:92:df:7f
dropbear.
hwclock: can't open '/dev/misc/rtc': No such file or directory
Starting syslogd/klogd: done
Starting tcf-agent: OK

PetaLinux 2018.3 petalinux_boot_from_flash /dev/ttyPS0

petalinux_boot_from_flash login: random: crng init done
```

```
PetaLinux 2018.3 petalinux_boot_from_flash /dev/ttyPS0  
petalinux_boot_from_flash login:
```

从上面的执行log看出来，很明显获取IP地址是在开机启动执行程序完成之后才开始的，所以一定要注意这一点。

9. zynq设备上运行指定驱动的app程序

在zynq fpga运行过程。

```
root@petalinux_boot_from_flash:/lib/modules/4.14.0-xilinx-v2018.3/extra# pwd  
/lib/modules/4.14.0-xilinx-v2018.3/extra  
root@petalinux_boot_from_flash:/lib/modules/4.14.0-xilinx-v2018.3/extra# ls  
blink.ko  
  
modprobe blink.ko      # 加载驱动  
mknod /dev/blink_Dev c 245 0 # 标 dev, 这个信息根据modprobe的提示信息输入  
ls /dev/blink_Dev     # 已经有dev  
blinkapp              # 运用app程序
```

10. linux 加载module

```
insmod blink.ko  
# 如果有同名文件module已经加载，可以先lsmod查看  
  
lsmod # 列出已经加载的module  
  
rmmod blink.ko # 删除已加载的module，可以在后面两次加载新的module.
```

11. module驱动代码实例

11.1 一个简单的device module 驱动代码

主要描述用户数据和内核数据之间如何互通，因为不能直接访问，需要通过copy_to_user & copy_from_user函数来完成。

```
#include <linux/uaccess.h> // copy_to_user copy_from_user  
  
#include <asm/uaccess.h> /* for get_user and put_user */ //  
raw_copy_to_user, raw_copy_from_user
```

```
/*
 * This function is called whenever a process which has already opened the
 * device file attempts to read from it.
 */
static ssize_t device_read( struct file *file, /* see include/linux/fs.h
 */
                           char __user * buffer, /* buffer to be filled
 with data */
                           size_t length, /* length of the buffer */
                           loff_t * offset)
{
    int ret;

    char k_buffer[20] = {0}; // kernel core buffer
    memcpy(k_buffer, "core_to_user_msg02", length);
    ret = copy_to_user(buffer, k_buffer, length);

    return SUCCESS;
}
/*
 * This function is called when somebody tries to
 * write into our device file.
 */
static ssize_t device_write(struct file *file,
                            const char __user * buffer,
                            size_t length,
                            loff_t * offset)
{
    // user function
    int ret;
    char k_buffer[20] = {0}; // kernel core buffer
    ret = copy_from_user(k_buffer, buffer, length);

    printk("kernel::: blink write: %s\n", k_buffer);
    // end

    return SUCCESS;
}
```

11.2 gpio驱动led举例

驱动部分：

```
#include <linux/init.h> //定义了module_init
#include <linux/module.h> //最基本的头文件, 其中定义了MODULE_LICENSE这一类宏
#include <linux/fs.h> //file_operations结构体定义在该头文件中
#include <linux/device.h> //class[]device结构体的定义位置
#include <linux/kernel.h> //printk头文件

#include <linux/uaccess.h> //copy_from_user头文件
#include <asm/io.h> //ioremap头文件

#include <linux/ioport.h>
#include <linux/of.h>
#include <linux/delay.h>

//定义主设备号
static int major_num;
// 定义设备文件名
#define DEVICE_NAME "leds"

//定义class[]device结构体
#define CLASS_NAME "mygpio"
static struct class* gpio_class;
static struct device* gpio_device;

#define LEDS_BASE_ADDR (0x41200000) //GPIO的Base addr[]用于映射

static unsigned *leds;

static int leds_drv_open(struct inode *Inode, struct file *File)
{
    *leds = 0x0;
    return 0;
}

static ssize_t leds_drv_read(struct file *file, char __user *buf, size_t
count, loff_t *ppos)
{
    return 0;
}

static ssize_t leds_drv_write(struct file *file, const char __user *buf,
size_t count, loff_t *ppos)
{
    unsigned int ret = 0;
    unsigned int tmp_val;

    ret = copy_from_user(&tmp_val, buf, count);

    *leds = tmp_val & 0xf;

    return ret;
}
```

```
}

// 描述与设备文件触发的事件对应的回调函数指针
static struct file_operations dev_fops =
{
    .owner = THIS_MODULE,
    .open = leds_drv_open,
    .read = leds_drv_read,
    .write = leds_drv_write,
};

// 初始化Linux驱动
static int __init leds_drv_init(void)
{
    int ret;

    leds = ioremap(LEDS_BASE_ADDR, 0x100);

    //获取主设备号
    major_num = register_chrdev(0, DEVICE_NAME, &dev_fops);

    //创建设备类
    gpio_class = class_create(THIS_MODULE, CLASS_NAME);

    if(IS_ERR(gpio_class))
    {
        unregister_chrdev(major_num, DEVICE_NAME);
        printk(KERN_ALERT "Failed to register device class\n");
        return PTR_ERR(gpio_class);
    }
    //注册设备
    gpio_device = device_create(gpio_class, NULL, MKDEV(major_num, 0), NULL,
    DEVICE_NAME);

    if(IS_ERR(gpio_device))
    {
        class_destroy(gpio_class);
        unregister_chrdev(major_num, DEVICE_NAME);
        printk(KERN_ALERT "Failed to create the device\n");
        return PTR_ERR(gpio_device);
    }
    printk(KERN_INFO "LED_GPIO_init: device created correctly\n");

    return 0;
}

// 卸载Linux驱动
static void __exit leds_drv_exit(void)
{
    iounmap(leds);
}
```

```
// 删除设备文件,后创建的先卸载
device_destroy(gpio_class, MKDEV(major_num, 0));
class_destroy(gpio_class);

unregister_chrdev(major_num, DEVICE_NAME); //释放设备号

// 输出日志信息
printk("LED_GPIO_exit success!\n");
}

// 注册初始化Linux驱动函数
module_init(leds_drv_init);
// 注册卸载Linux驱动函数
module_exit(leds_drv_exit);

MODULE_LICENSE("GPL");
```

应用程序app部分：

```
#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char** argv)
{
    int fd;

    fd = open("/dev/leds", O_RDWR);

    if(fd < 0)
    {
        printf("fd = %d open failed!\n", fd);
    }

    unsigned int leds = 0;

    while(1)
    {
        write(fd, &leds, 4);

        leds++;
        leds %= 0xf;
        sleep(1);
    }

    close(fd);
}
```

```
    return 0;  
}
```