裸机开发比较底层，是直接操作寄存器的，即使提供了库函数，也只是将操作寄存器进行了封装。而可以运行Linux系统的芯片一般都外设复杂寄存器繁多，裸机开发就很困难。

Linux下的驱动开发主要是调用Linux内核提供的统一api函数，只要提供必要参数就可直接实现所需要的操作，但要按照驱动框架，统一了外设，远离了寄存器。

* 裸机开发：操作寄存器控制设备
* 驱动开发：调用Linux内核的api函数实现控制设备
* 应用开发：开关读写四个函数就直接控制设备（已被视为文件）

dts设备树文件描述了板子的设备信息，会与Linux内核匹配，使内核得到可操控的设备。

* 内核：内核让用户不需要关注处理器的差异性，不同的芯片上可以进行同样的指令和操作。
* 驱动：驱动让用户不需要关注外设的差异性，不同的设备都可按照文件进行开关读写操作。

Linux驱动设备分为三大类：字符设备、块设备和网络设备（一个设备不一定只属于其中一种）。对设备驱动的编写需要编写file\_operations结构体里包含的函数（如open，close，write等函数），将设备包装为文件。

应用程序和驱动的关系：

驱动是负责获取外设数据、控制外设的，然后将数据交给应用程序。而数据的具体应用和功能是应用程序决定的，应用程序只当外设为文件，不需要具体的了解。

单片机开发中，设备的驱动和应用是在一个工程里的，是混在一起的。而Linux下，驱动和应用是完全分开的，内核和驱动运行在内核空间中，应用程序运行在用户空间中。应用程序通过直接或间接调用系统调用来使用内核资源，比如，应用程序调用C库提供的printf函数，而该函数最终调用的write函数为系统调用，使内核完成写操作（系统调用过程就是应用发生软中断并提供一个系统调用号，内核接收中断并根据系统调用号执行内核里的对应函数）。

注意：驱动最终是给应用调用的，要考虑到应用开发的便利性（这也就是要编写开关读写函数的原因）。驱动是分驱动框架的，要按照驱动框架来编写。

编写驱动的重要事项：由于驱动程序是依赖于内核源码的，所以测试驱动用的Linux系统的zImage和dtb文件要是编译驱动时包含的内核源码编译生成的，不然有可能出错（也就是驱动编译包含的内核源码和生成Linux系统的内核源码要是一套源码）。

**虚拟字符设备驱动实验（Ubuntu：1\_virtualchrdev文件夹）**

字符设备驱动编写就是file\_operations结构体里的成员函数的编写，它就是字符设备驱动的框架。

Linux驱动程序可以选择编译到内核里，也就是包含在zImage里，内核启动后可以自行运行驱动程序；也可以选择编译为模块（.ko文件），需要的时候加载模块，不需要时卸载模块即可。通过module\_init(xxx\_init)函数来注册模块加载函数后，通过insmod命令就会调用xxx\_init函数；通过module\_exit(xxx\_exit)函数来注册模块卸载函数后，通过rmmod命令就会调用xxx\_exit函数。

modprobe命令也可以加载模块，并且会根据依赖关系自动加载该模块依赖的但没有被加载的模块，更加智能，modprobe -r为卸载模块命令（modprobe命令默认会去/lib/modules/系统版本号/文件夹下找模块，所以要创建此目录，比如mkdir /lib/modules/4.1.15/）。每当modprobe命令加载一个新的模块之前，需要modules.dep文件里有此模块的信息，要通过depmod命令直接写入信息到modules.dep，如果没有该命令就只能重新配置BusyBox。lsmod命令可查看当前加载着的模块。

内核提供了注册和注销字符设备的函数（register\_chrdev和unregister\_chrdev），一般在驱动模块入口函数注册设备，在出口函数注销设备。Linux中每个设备都有个32位的设备号，设备号由高12位主设备号（表明某个具体驱动）和低20位次设备号（表明此驱动下的某个具体设备）组成，通过cat /proc/devices可以查看当前设备的设备号。而register\_chrdev和unregister\_chrdev函数的缺点是注册设备时会直接占用一整个主设备号，哪怕就只有一个设备（更像是注册了一种设备驱动，不看设备数量）。

通过 man <函数名> 命令可以直接查看这个函数的参数、返回值、头文件等各种用法信息。

实验步骤：

1、编写虚拟字符驱动模块代码

首先，编写驱动模块的基本代码，包括加载模块函数和卸载模块函数，将这两个函数注册到模块init和exit信息中，就可以实现加载和卸载时的调用，同时注册模块其他信息，比如license、author等。用modprobe加载，lsmod查看，rmmod卸载。

其次，在加载模块函数中注册一个虚拟字符设备，给出参数主设备号、设备名称和操作函数集。在卸载模块函数中通过主设备号和设备名称注销虚拟字符设备。用cat /proc/devices可以查看已注册的设备。

最后，编写自己的open、release、read、write四个函数赋值给操作集中对应的成员，owner赋值为THIS\_MODULE表示操作集属于此模块。注意，read函数中驱动给应用传递数据时不能直接赋值，要使用copy\_to\_user函数，同样write函数中应用给驱动传递数据时也要使用copy\_from\_user函数，不能直接赋值的目的是阻止应用通过指针直接访问到内核数据。

驱动框架就已经搭建好了，应用程序可通过文件基本操作函数调用到对应注册设备的操作集里的函数，最后调用到自己定义的四个驱动函数。

2、编写测试的应用程序代码

编写main函数，通过argv给设备文件名赋值，执行打开open()，读取read()，写入write()，关闭close()测试。对应函数的使用方法和头文件需求，通过man <函数名>可以查看。注意，应用程序printf和内核驱动printk两个打印函数可能同时写的冲突问题。

3、编写Makefile文件

通过指定工作目录和模块源码目录，按照内核编译规则，依赖内核源码编译驱动模块得到.ko文件，同时交叉编译得到应用程序的可执行文件。

4、应用程序测试驱动

将驱动模块和应用程序文件发送到/lib/modules/4.1.15/目录下，modprobe virtualchrdev.ko加载驱动模块，mknod /dev/virtualchrdev c 200 0创建设备节点（c表示为字符设备，200为主设备号，0为次设备号，注册设备不会自动生成设备节点）。/dev/下有了自定义的设备后，./virtualchrdev\_app /dev/virtualchrdev进行测试，完全成功（后面那个是输入给应用的参数，为设备文件名）。

**寄存器操作LED灯驱动实验（Ubuntu：2\_led文件夹）**

Linux不能直接对寄存器物理地址进行读写操作，因为Linux会使能MMU，MMU通过内存映射将虚拟地址映射到物理地址上，而应用只能访问到虚拟地址。

因此Linux要访问寄存器，就需要得到物理地址所对应映射的虚拟地址，io.h中有两个函数，ioremap函数可以将物理地址转为虚拟地址，它会将所需要的物理地址范围映射到虚拟地址上并给出虚拟地址，iounmap函数会将需要释放的虚拟地址释放掉，即解除与物理地址的映射关系。对虚拟地址内存的读操作为readb、readw、readl函数，写操作为writeb、writew、writel函数。

实验步骤：

1、搭建模块和驱动框架。

2、初始化外设时钟、电气属性和IO设置。先宏定义LED所用到的寄存器物理地址，在加载函数中通过ioremap得到这些寄存器映射的虚拟地址，然后进行寄存器配置灯。在卸载函数中通过iounmap释放掉映射的虚拟地址。

3、在文件写函数中，通过已经得到的虚拟地址可以非常方便的修改寄存器，控制灯的量灭。为了不影响其他引脚，要使用读-改-写的方式。

4、编写应用程序测试。打开文件，根据输入参数写入0或1，然后关闭文件。

**新字符设备驱动实验（Ubuntu：3\_newchrdev文件夹）**

前两个实验的字符设备驱动有一缺点，注册驱动设备的register\_chrdev函数会占用一整个主设备号，浪费很多次设备号资源，而且需要自己查看并指定一个没用过的主设备号。

取而代之的是，使用alloc\_chrdev\_region函数可以向系统申请注册一个没有使用的设备号，使用register\_chrdev\_region函数可以申请注册指定主设备号和次设备号，使用unregister\_chrdev\_region函数可以注销前两个函数注册的设备号。编写驱动时要考虑完善，给定主设备号和没给定主设备号的情况都要考虑到。

新字符驱动使用cdev结构体来表示一个字符设备，里面包含了设备号和设备文件操作集合，注册完设备号后，就要初始化cdev结构体来注册设备的操作集。使用cdev\_init函数初始化cdev，使用cdev\_add函数添加到Linux内核中。至此才完成了字符设备的注册。卸载模块时，首先要cdev\_del函数注销字符设备，才能接着注销设备号。

旧的是一个函数接收主设备号、设备名称和操作集就完成注册。而新的先需要主设备号、次设备号和设备名称完成设备号的注册，再用file\_operations操作集初始化cdev结构体，最后将cdev和设备号一同添加到内核来完成字符设备驱动的注册。（其实旧的函数内部实现也是新的函数封装组成的，因此比较简洁）

先前实验中设备文件节点是通过mknod手动创建的，2.6版本内核开始引入udev机制提供热插拔管理，可在加载驱动时自动创建/dev/下的设备文件。BusyBox提供了udev的简化版本mdev，由它实现设备文件节点的自动创建和删除。自动创建时，使用class\_create函数会创建一个类保存在/sysfs下，接着使用device\_create函数时mdev会在/sysfs下查找对应的类并生成设备节点。自动删除时，先device\_destroy删除设备节点，再class\_destroy删除类结构体。

文件私有数据：在open函数中将设备信息结构体数据赋值给filp->private\_data，在read、write函数中通过filp->private\_data访问设备信息，保证隐私。当多个相同设备需要驱动时，open函数需要kmalloc函数申请开辟一个设备信息结构体内存并将地址赋值给private\_data，这样保证了不同设备使用的是自己的设备信息结构体，互不冲突，当然release函数中需要kfree函数释放申请的内存（如果直接使用自定义的全局变量，那多设备只能共用一个信息可能出现冲突）。

goto错误处理：每步申请资源都有可能出现问题，所以需要判断返回值，返回值若是错误的就要进行相应的错误处理（将先前申请的资源释放再退出），如果都写在一起会使得申请资源代码变得冗长。所以出现错误就通过goto命令进行label跳转，label写在正常代码的return后面（保证正常工作时不会运行到），错误处理按照申请资源步骤反向处理（第n步出错要释放前n-1步申请的资源，第n-1步出错要释放前n-2步申请的资源，相当于第n步错误处理要释放第n-1步申请的资源并执行第n-1步错误处理）最后return 错误码。将正常申请资源代码和错误代码分开，可读性高。

实验步骤：

1、编写模块框架。自定义了一个设备信息结构体，里面包含了cdev、设备号、主设备号、次设备号、设备名称和设备数量。将所有注册设备驱动需要的信息都封装进了一个newchrdev\_t结构体中。

2、加载函数中，第一步初始化LED（与上一实验相同），第二步根据major是否为0，两种方法注册LED字符设备驱动的设备号，第三步将定义好的file\_operations结构体初始化给cdev结构体，第四步将cdev和设备号一同添加到内核中，完成字符设备驱动注册。第五步，创建一个class结构体，该结构体在创建设备节点时mdev需要查找他，接着创建设备节点，完成了设备节点的自动创建。在成功return后面写入错误处理代码，以申请资源相反顺序释放资源后return并贴上label，在申请错误时通过goto跳转到对应错误处理标签。

3、卸载函数中，第一步，删除设备节点和class结构体，第二步注销字符设备驱动，第三步注销设备号，最后关闭LED并释放虚拟地址映射（与上一实验相同）。

4、文件操作函数规定不能直接访问设备信息结构体数据，在open函数中通过输入参数file指针里的private\_data指针成员指向设备信息结构体地址，在read、write等函数中只能通过输入参数file指针里的private\_data指针成员间接去访问设备信息结构体。

5、测试的应用程序与上一实验一样。

**Linux设备树**

DTS文件采用树形结构描述板级设备（如CPU数量、内存基地址、IIC接口所接设备等等）。在没有设备树文件的Linux版本中或者单片机中，都是通过在某.c文件中定义各种设备信息的结构体来描述设备的。但由于ARM芯片很多，每种芯片的开发板也很多，导致描述板级设备的.c文件太多了，它们都会被编译进内核导致臃肿，因此引出了设备树来单独保存板级设备信息，将它从内核代码层次中分离出来（只有一个dts文件会编译进内核代码）。

dts是设备树源码文件，DTC工具相当于gcc编译器，可以将dts编译成dtb文件。在Linux源码顶层路径下，make dtbs编译所有dts文件，make xxxx.dtb编译指定的dtb文件。

dts语法是ASCII文本，适宜阅读。dts可以引入头文件，其中有个头文件为.dtsi，是一个芯片在不同的平台或设备上共有信息提取出的集合（一般是cpu内部的外设）。dts从'/'开始表示根，通过{}将设备排列出树结构，同一大括号内的设备为兄弟节点，子节点设备在父节点设备大括号内部，同时大括号内部有此设备的属性信息。在设备名字前加上&符号表示追加，表明设备是在原定义基础上追加大括号中的设备信息。支持C语言风格的注释。dts文件最开头的/dts-v1/;表示dts文件版本为版本1。接下来是引用dtsi和其他头文件。后面的/ {...};段就是设备树根节点（与dtsi里的根节点是取并集的），里面是子节点段（xxx {...};）和根节点的属性（xxx = "aaaaa";）。剩下的在根节点段外的&xxx {...};都是对已有设备属性的追加，相同属性重新赋值可以覆盖原值。节点名字完整要求是node-name@unit-address，unit-address一般是外设寄存器起始地址，也可能是通信设备地址，可以在前面添加标签（label: node-name@unit-address）以方便通过标签来访问节点，也可以省去地址（node-name）。

Linux系统启动后可以在根文件系统/proc/device-tree/里看到设备树的节点信息，属性变成了文件，cat命令可查看属性值，节点变成了文件夹，形成树形结构。自己添加设备节点就在dts文件根节点段里最后面添加（理论上应该根据结构放到对应外设里，但放在根节点里比较方便查找修改）。

特殊节点：

* aliases节点主要功能是给节点取别名，方便访问，但是一般需要访问的节点都会加上label，然后通过&label访问。被取了别名并使能的设备会以 /dev/别名 这样形式的设备节点出现。
* chosen节点不是一个真实的设备，主要是用于uboot给Linux内核传递参数，uboot会将bootargs环境变量添加到chosen节点作为一个属性。

标准属性：

* compatible属性，描述兼容性且每个节点必须有，值格式为 "厂商,驱动1名字","厂商,驱动2名字",... 表明了设备的厂商和支持哪些驱动（系统会通过此属性查找对应设备的驱动文件来驱动设备）。
* model属性，描述设备模块信息。
* status属性，描述设备状态，'okay'表示已使能可操作，'disabled'表示未使能不可操作。
* reg属性，描述寄存器地址，值格式为 <寄存器起始地址 地址长度>。
* #address-cells和#size-cells属性，描述其子节点的reg属性值中起始地址和地址长度的值分别要占几个32位（4字节）空间来保存（reg里是连续空间中保存了2个值，因此需要知道前多少字节保存的是地址，后多少字节保存的是长度）。
* ranges属性，描述子节点和父节点reg地址的地址映射关系，为空表示没有地址映射。
* name属性，描述节点名字（新版已经弃用了）。
* device\_type属性，描述节点设备类型（新版已经弃用了）。

设备下的compatible属性是用来找对应驱动的，而根节点下的compatible属性是用于查找内核是否兼容此平台。内核编译时会生成MACHINE结构体，里面变量有机器ID和compatible列表。没有设备树的时候，通过查找保存着的机器ID列表中有无MACHINE结构体的机器ID，若有则表明内核可兼容此平台；有设备树后，通过查找MACHINE结构体的compatible列表中有无该平台dts文件中根节点的compatible属性值，若有则表明内核可兼容此平台。

Documentation/devicetree/bindings/中有各种外设基于不同类型芯片的绑定信息文档，文档提供了配置设备树的属性和示例来帮助用户参考设置设备树。

Linux内核的OF操作函数：

驱动中使用OF函数获取设备树的属性内容。OF函数指的是一大批of\_xxxxx为名字的设备树操作函数。

* 驱动获取设备树节点首先要找到节点，of\_find\_node\_by\_name函数按名字查找，of\_find\_node\_by\_type函数按类型查找，of\_find\_compatible\_node函数按兼容性查找，of\_find\_node\_by\_path函数按路径查找，找到则返回节点结构体。
* of\_get\_parent函数和of\_get\_next\_child函数分别为查找父节点和下一个子节点。
* of\_find\_property函数查找返回某一个节点的某一个属性的结构体。of\_property\_count\_elems\_of\_size函数获取到某节点某属性值的元素个数（如reg=<1 2 3 4>就是reg属性有四个元素，每个元素默认大小是u32的）。of\_property\_read\_u32\_index函数读取某节点某属性值对应下标的u32元素值。of\_property\_read\_string\_index函数读取某节点某属性值对应下标的字符串元素值。
* of\_property\_read\_u8\_array函数、of\_property\_read\_u16\_array函数、of\_property\_read\_u32\_array函数、of\_property\_read\_u64\_array函数分别是一次性获取某节点某属性以u8、u16、u32、u64为类型的数组数据值。
* of\_property\_read\_u8函数、of\_property\_read\_u16函数、of\_property\_read\_u32函数、of\_property\_read\_u64函数分别是获取某节点某属性以u8、u16、u32、u64为类型的数据值。
* of\_property\_read\_string函数获取某节点某属性的字符串值。
* of\_n\_addr\_cells函数获取#address-cells属性值。of\_n\_size\_cells函数获取#size-cells属性值。
* of\_iomap函数将某个节点的reg属性数组中第index对数据（起始地址和地址长度）进行地址映射并返回虚拟地址指针。

**设备树led驱动实验（Ubuntu：5\_dtsled文件夹）**

先前的led驱动控制寄存器是通过物理地址内存映射到虚拟地址进行操作的，其中物理地址是宏定义定死的。而此次使用OF函数从设备树获取led的属性值，通过属性值来操作led。

实验步骤：

1、设备树添加节点。在根节点下最后添加一个自己的设备节点，添加compatible、status和reg属性，其中reg属性保存的是led操作需要的寄存器物理地址和地址长度数组。make dtbs编译生成新的dtb文件。

2、编写模块和驱动框架。所有均和上个实验一样，最主要的不同是，物理地址不是宏定义给出，先通过OF函数根据路径得到led的设备树节点，分别获取节点的compatible、status和reg属性，将reg属性中的值就是物理地址和地址长度，将它们进行ioremap映射为虚拟地址。（of\_iomap函数也可以一次性实现读取reg值并映射为虚拟地址）

3、编写测试的应用程序。代码和上个实验的测试应用程序相同。

4、测试成功

**pinctrl和gpio子系统驱动实验（Ubuntu：6\_gpioled文件夹）**

目前为止，前面的实验不断地更新和丰富驱动框架，但其实对于led的控制还是处于裸机阶段，因为依旧是直接控制寄存器来操控led的。Linux内核提供了pinctrl子系统和gpio子系统来处理针对GPIO引脚的操作。

pinctrl子系统可以帮助用户设置引脚的复用和电气属性，只需要在设备树中设置好相关引脚的属性，pinctrl子系统会自动将其初始化。dtsi文件中已经定义好了IOMUX\_SNVS控制器、IOMUX控制器和IOMUX\_GPR控制器的起始地址，它的头文件中也宏定义好了所有的引脚复用（里面的地址是基于控制器基地址的偏移地址），所以只要在复用控制器节点下直接追加自己设备需要的引脚复用和电气属性就行。在对应复用控制器节点下追加创建自己设备的pinctrl子节点，将设备所用引脚放到此子节点下。

gpio子系统可以帮助用户进行引脚的gpio操作，gpio子系统只负责操作那些复用为gpio的引脚。dtsi文件中已经定义好了GPIO控制器设备树节点。创建自己的设备节点，通过给pinctrl-0，pinctrl-1，pinctrl-2...这些属性赋值设备树节点标签，来告诉gpio子系统用哪个设备树节点初始化引脚（pinctrl-names可以定义设备的多种状态，pinctrl-names = "default"; pinctrl-0 = <&pinctrl\_usbdhc1>;就是表明第一种状态即默认状态的初始化是使用标签为pinctrl\_usbdhc1设备树节点里的属性初始化）。xxxx-gpios = <&gpio1 2 GPIO\_ACTIVE\_LOW>;表明自定义的xxx引脚是GPIO01IO02且低电平有效，驱动就可以对此属性用of\_get\_named\_gpio函数得到这个引脚，从而去控制输入输出。gpio子系统提供了一系列API函数，gpio\_request函数来请求GPIO引脚，gpio\_free函数来释放GPIO引脚，gpio\_direction\_input函数设置为输入，gpio\_direction\_output函数设置为输出，gpio\_get\_value函数读取电平，gpio\_set\_value函数设置电平。

实验步骤：

1、创建设备树节点。在根节点下创建gpioled节点，设置pinctrl-names为"default"，pinctrl-0为<&pinctrl\_gpioled>，led-gpio为<&gpio1 3 GPIO\_ACTIVE\_LOW>。表明使用默认的pinctrl子系统引脚初始化方法，初始化引脚和属性保存在标签为pinctrl\_gpioled的设备树节点中，自定义的IO为GPIO01IO03。在iomux节点下，仿照其他节点定义pinctrl\_gpioled的复用信息和电气属性。make dtbs编译生成新的dtb文件。

2、编写模块和驱动框架。和以前实验一样，只是引脚的初始化pinctrl已经完成，只需要得到引脚编号就可以操作。先通过路径得到pinctrl\_gpioled节点，利用of\_get\_named\_gpio函数让gpio子系统提供led-gpio这个自定义属性的IO编号，最后申请此编号。其实，用此编号就能操控IO，但申请是为了占用此IO资源让其他进程无法使用，或者发现其他进程已经占用了此IO资源。官方的dts文件里默认使用到了此引脚，所以需要将其他地方对此引脚的使用全部注释掉，才能成功申请到IO资源。（这样一对比，直接控制寄存器来控制IO过于流氓了，非常有可能影响到其他进程的正常工作。。。）

3、编写测试代码。和上个实验一样。

4、测试成功。’

**蜂鸣器驱动实验（Ubuntu：7\_beep文件夹）**

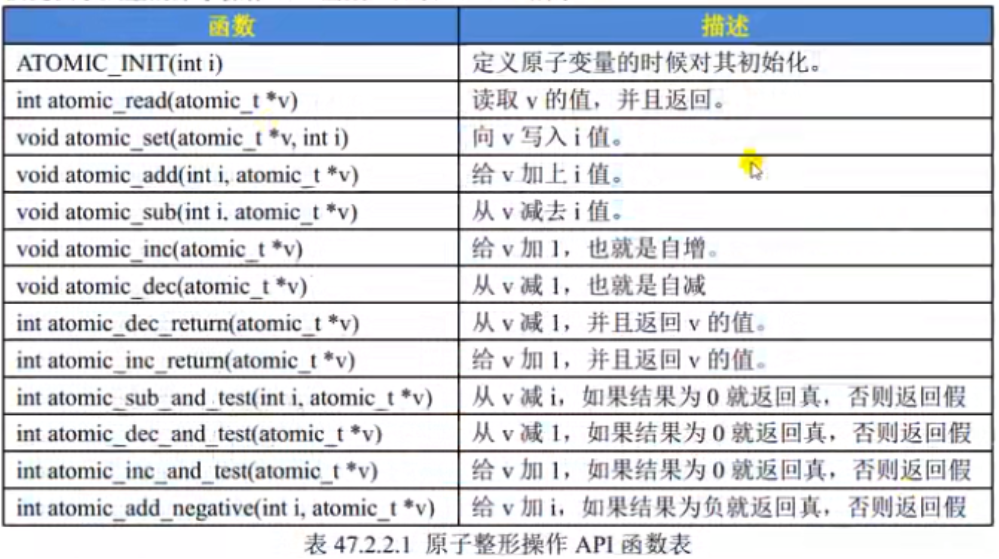
所有实验原理上个实验已经讲了，只是换了个引脚。唯一要注意的是蜂鸣器的引脚SNVS\_TAMPER1在复用设置时，此引脚在MX6UL和MX6ULL当中地址是不同的，所以要选择MX6ULL的MX6ULL\_PAD\_SNVS\_TAMPER1\_\_GPIO5\_IO01。

实验步骤：

和上一个实验全部一样，还是用pinctrl子系统在设备树加载时就进行复用设置，驱动通过gpio子系统提供的API函数控制引脚，只是把gpioled改成了beep。

**并发与竞争实验（Ubuntu：8\_atomic文件夹、9\_spinlock文件夹、10\_semaphore文件夹）**

一开始编写驱动时就应该考虑到并发和竞争，否则代码调试改错难度很大。Linux系统由于自己的多任务、中断、抢占和多核会引发竞争问题。需要保护的是共享数据，防止共享数据被并发访问，要访问临界区需要原子操作。原子整型变量类型为atomic\_t，64位芯片的原子整型变量类型为atomic64\_t，这种变量可以用来记录资源当前被多少进程共享，限制共享数量。

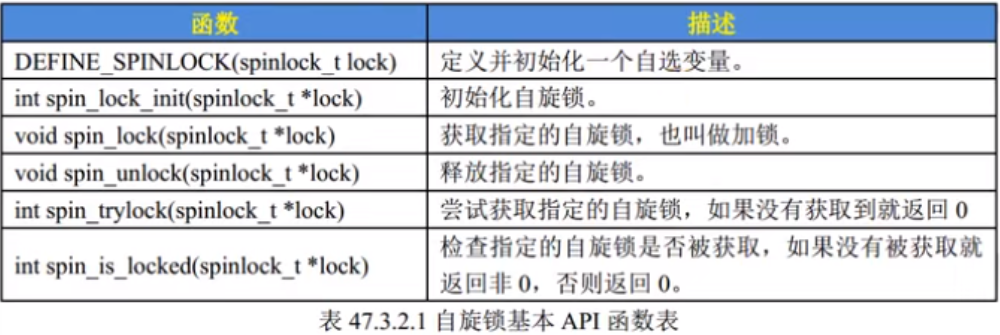


原子位操作则不需要变量，是直接通过指向内存的指针对内存进行位操作。



对于更加复杂的临界区操作时，比如保证一个结构体变量操作的原子性，这时候需要使用锁机制。当一个线程访问某共享资源时需要先获取相应的锁，锁只能被一个线程持有，线程不释放持有的锁则其他线程无法获取锁。

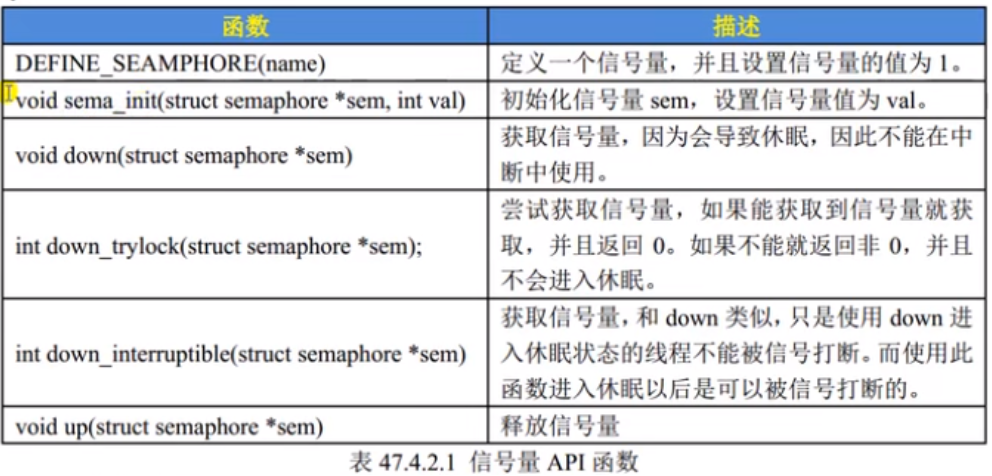
自旋锁spinlock\_t的意思就是需要获取锁的线程不会休眠或者做其他事而是一直循环等待锁的获取（原地打转），缺点是等待锁的自旋状态浪费CPU时间，适合短时期的多核或者单核多线程的轻量级加锁。自旋锁会自动禁止抢占，使用自旋锁要注意死锁现象的发生，不要在持有锁期间进入休眠或调用阻塞。中断是会打断进程的，如果中断也要获取和进程相同锁，那就要在进程获取锁时禁止本地中断（本地指的是本CPU）。



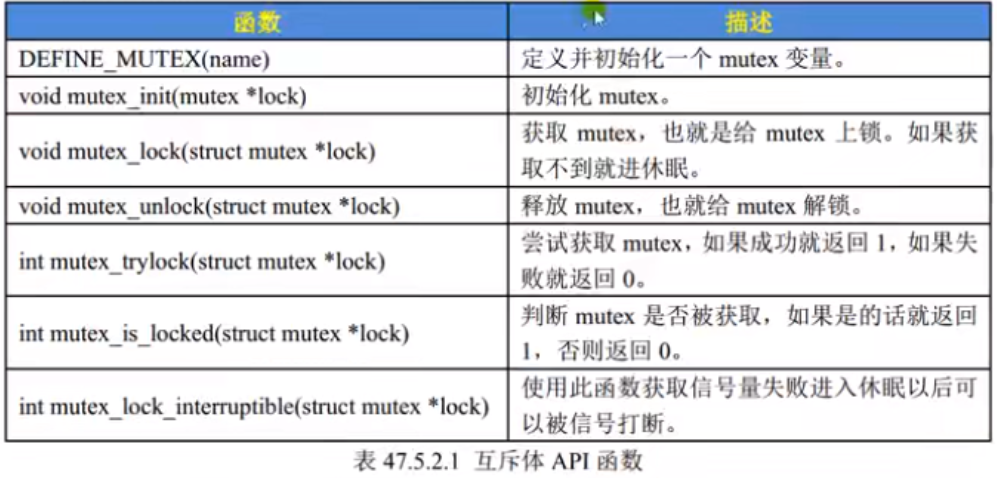
为了驱动的可移植性，编写驱动都将芯片当做多核来处理。



信号量semaphore相比自旋锁，可以让等待资源的线程进入休眠状态，减少CPU时间的浪费。适用于占用资源比较久的场景，当等待资源时间大于休眠切换线程的时间开销时建议使用信号量，否则建议使用自旋锁。因为信号量不会禁止抢占，所以可以在信号量的临界区休眠或阻塞。中断不能使用信号量只能用自旋锁，因为中断不能休眠。持有自旋锁期间不能获取信号量，因为持有锁期间不能休眠。计数型信号量同一时间只允许一定数量的线程共享资源，计数值为1时就是互斥访问。每有一个线程获取信号量，信号量值降一，释放信号量，值升1，信号量大于0表示还可以获取，等于0表示无法获取。



Linux还定义了互斥体mutex，专门处理互斥访问。功能特点和二值信号量相同，但机制比二值信号量更加专业。



实验步骤：

1、原子操作实验（8\_atomic）。在gpioled实验基础上，驱动定义一个原子整型变量，初始化为1表明同一时间只有一个线程可以占用led资源。第一种方式，在open函数中通过atomic\_read函数判断原子变量是否小于等于0，是则返回错误，否则atomic\_dec函数让原子变量减一表示占用资源。第二种方式，在open函数中通过atomic\_dec\_and\_test函数判断原子变量减一后是否为0，是则已经占用了资源继续其他操作，否则atomic\_inc函数加回来并返回错误。两种方法，close函数中都要atomic\_inc函数加一，表示释放资源。

应用程序的代码主要在关闭文件前加入延时25s，测试时通过连续多次运行同一个此应用程序，表明同一时间只有一个应用程序可以成功打开文件并操作led。

2、自旋锁操作（9\_spinlock）。在gpioled实验基础上，驱动定义一个设备状态变量和自旋锁，设备状态初始化为0表示未被占用，初始化自旋锁保护设备状态变量。在open函数中通过spin\_lock\_irqsave函数持有自旋锁并保存中断状态，可以独占设备状态变量，判断设备状态，为0则变量加一表示此线程占用并使用spin\_lock\_irqrestore函数释放自旋锁并恢复中断状态，为1表示已经被占用直接spin\_lock\_irqrestore函数释放自旋锁并恢复中断状态。在close函数中使用spin\_lock\_irqsave函数持有自旋锁并保存中断状态，将设备状态变量减一，再spin\_lock\_irqrestore函数释放自旋锁并恢复中断状态。

应用程序的代码主要在关闭文件前加入延时25s，测试时通过连续多次运行同一个此应用程序，表明同一时间只有一个应用程序可以成功打开文件并操作led。

3、信号量和互斥体操作（10\_semaphore）。在gpioled实验基础上，驱动定义一个信号量或者互斥体，初始化信号量值为1或者初始化互斥体，表明资源只能互斥访问。在open函数中通过信号量的down函数或者互斥体的mutex\_lock函数获取资源，如果资源已经被占用会进入休眠等待。在close函数中使用信号量的up函数或者互斥体的mutex\_unlock函数释放对资源的占用。

应用程序的代码主要在关闭文件前加入延时25s，测试时通过连续两次运行同一个此应用程序，表明同一时间只有一个应用程序可以成功打开文件并操作led。第二次运行的程序会进入休眠等待状态，直到第一次运行的程序完成操作释放资源后，第二次运行的程序才开始继续运行。

**内核定时器实验（Ubuntu：13\_timer文件夹）**

Cortex-M内核使用systick作为系统定时器。定时器有硬件定时器（硬件外设实现）和软件定时器（指令周期实现），对硬件定时器的控制也是相当于驱动字符设备。

Linux内核有着很多周期性管理的程序，需要有一个定时器一定时间产生一个定时中断来通知内核，此定时中断的频率被叫做节拍率HZ（不是内核主频，Linux内核图形化配置界面可以设置节拍率）。高节拍率可以提高时间精度，但同时更频繁的中断会给系统带来更高的负担，所以要选择合适的值，本实验采用默认的100Hz。

Linux内核使用全局变量jiffies来记录启动以来的系统节拍数，64位系统为jiffies\_64，32位系统为jiffies是jiffies\_64的低32位。本实验就利用节拍数变化来实现基础定时。系统提供了time\_after函数和time\_before函数来判断jiffies有没有超过设定时间（函数内部处理了jiffies溢出绕回），jiffies\_to\_msecs函数和jiffies\_to\_usecs函数将jiffies值转为ms、us，msecs\_to\_jiffies函数和usecs\_to\_jiffies函数将ms、us转为jiffies值。

内核定时器采用系统时钟实现，当到了设定时间（设定的是jiffies时间点，而不是定时周期）就会自动执行设定好的定时处理函数，属于软件定时器，与硬件定时器不同的是，它不是通过中断进入处理函数，并且只会执行一次（要周期定时需要在处理函数中重新开启定时器）。系统提供了init\_timer、add\_timer、del\_timer、del\_timer\_sync、mod\_timer函数来操作内核定时器。

内核也有延时函数：ndelay函数、udelay函数和mdelay函数。

和开关读写函数相同，应用程序调用ioctl函数时，也会对应调用驱动中file\_operations结构体中的unlocked\_ioctl函数或compat\_ioctl函数。64位系统中调用32位的应用程序会调用compat\_ioctl函数，32位系统中调用32位的应用程序会调用unlocked\_ioctl函数。自己设备的ioctl的命令可以自定义，但是需要符合Linux的规则，规定32位命令被分为四个字段（8位幻数type、8位序数nr、2位数据传输方向、14位数据大小size），构建命令函数：没有参数的\_IO函数、从驱动读数据的\_IOR函数、向驱动写数据的\_IOW函数、双向数据传输的\_IOWR函数。ioctl函数通过switch语句来处理不同的命令。

实验步骤：

1、编写模块和驱动框架。这次实验在gpioled实验基础上，模块和驱动框架相同。

2、在open函数中初始化timer\_list类型变量timer，设置timer的定时处理函数（自定义的函数，输入参数为timer里的data变量，主要功能是翻转led电平并开启下一个定时器），设置定时时间为当前jiffies加上period毫秒转jiffies值，也就是定时priod毫秒，设置timer的data变量为设备信息结构体指针（方便定时函数里访问设备的信息），最后add\_timer开启定时器。在close函数中用del\_timer删除定时器。

3、复写unlocked\_ioctl函数为led\_ioctl函数，通过第二个参数输入的命令和第三个参数输入的命令参数的指针，通过switch语句进行不同的处理。命令通过按照Linux命令规则自行宏定义实现。

4、编写应用测试代码。打开文件后，不断循环通过scanf函数读取输入的命令编号和命令参数，根据不同命令编号给ioctl函数使用不同的命令和命令参数指针（命令也需要宏定义在应用代码中）。

5、实验完全成功，可以通过输入命令控制led是否闪烁和闪烁速度。

**Linux中断实验（Ubuntu：14\_irq文件夹）**

Linux内核提供了完善的中断框架（半导体厂商负责实现自己芯片的中断框架），编写驱动者只需要申请中断并且注册中断处理函数即可。

Linux内核中通过request\_irq函数申请中断并且使能中断，此函数可能会导致睡眠（要知道需要使用中断的中断号）。通过free\_irq函数删除中断并禁止中断。中断处理函数形式irqreturn\_t (\*irq\_handler\_t)(int, void\*)，返回的是一个枚举类型，共有三种返回值。使能某个中断enable\_irq函数和禁止某个中断disable\_irq函数、disable\_irq\_nosync函数。使能处理器全局中断local\_irq\_enable函数和禁止处理器全局中断local\_irq\_disable函数（可能会影响其他程序的中断状态）。local\_irq\_save函数保存当前中断状态并禁止全局中断，local\_irq\_restore函数读取恢复先前中断状态。

中断服务函数属于中断处理的上半部，执行时间越短越好。而将中断处理中比较耗时的处理代码提出来，就是中断处理的下半部。建议只将那些不希望被其他中断打断的，对时间敏感的，与硬件有关的处理代码放在上半部。

* Linux提供了结构体softirq\_action软中断来处理下半部。系统定义了10个软中断，CPU共享它们的软中断处理函数，但CPU只会在自己触发的中断时执行对应的软中断，通过open\_softirq函数注册软中断处理函数，通过raise\_softirq函数触发软中断（编译时必须静态注册软中断）。软中断执行延迟于中断执行，但还是优先于线程执行。但不建议使用。
* tasklet是利用软中断实现的另一种下半部机制，定义tasklet\_struct结构体，通过tasklet\_init函数进行初始化。在上半部调用tasklet\_schedule函数就能在合适的时间运行下半部的代码。
* 工作队列是另一种下半部机制。工作队列会将下半部代码交给一个内核线程执行，由于处于进程上下文中，所以允许中途的睡眠和调度（不允许睡眠和调度的代码不能使用工作队列）。work\_struct结构体表示一个工作，workqueue\_struct结构体表示工作队列，Linux内核使用工作者线程（worker thread）来处理工作队列中的各个工作。使用时，只需定义一个work\_struct工作，通过INIT\_WORK函数来初始化，在上半部调用schedule\_work函数就能将下半部的代码排入工作者线程的工作队列中。

设备树中的中断信息节点。Linux内核通过读取设备树的中断属性信息来配置中断。dtsi文件中有芯片中断控制器的设备树节点，interrupt-controller属性表明此节点为中断控制器，#interrupt-cells=<3>指定其子节点interrupts属性值占3个空间（中断种类、中断号、触发类型）。其他设备树节点可通过interrupt-parent=<&intc>来指定自己的中断父节点，接受父节点#interrupt-cells的规则，可以interrupt-controller也表明自己为中断控制器，且为自己的子节点定#interrupt-cells规则。在使用中，将自定义设备需要中断的引脚所属gpio作为自己的中断父节点（如：interrupt-parent=<&gpio5>），interrupts属性设置两个值，分别是引脚编号和触发方式（如：interrupts=<0 8>，IO00引脚的低电平触发；1为上升沿，2为下降沿，4为高电平，8为低电平）。驱动中，通过irq\_of\_parse\_and\_map函数获取某设备树节点的interrupts属性值对应中断号，仅针对GPIO的也可使用gpio编号通过gpio\_to\_irq函数得到中断号。

实验步骤：

1、修改设备树。在按键设备树节点中添加interrupt-parent属性和interrupts属性。

2、编写模块和驱动框架代码。代码框架保持不变的基础上，初始化原子变量按键值和按键释放标志，初始化20ms的消抖计时器，初始化用于中断下半部处理的tasklet和work。通过gpio\_to\_irq函数或者更为普适的irq\_of\_parse\_and\_map函数得到按键引脚对应的中断号，申请中断号并使能中断。按键按下和释放都会触发设置好的中断函数，中断函数里设置20ms消抖定时，并且调度tasklet和work来处理中断下半部。定时到达时，进入定时器处理函数，根据电平状况，将按键值写入到原子变量中。read函数里将原子变量保存的按键值发送给应用程序。（在真正的驱动编写中，tasklet和work只需要用一个，将所有中断处理的下半部写到它们对应的函数中即可。有一点要注意，work对应的处理函数参数为work结构体的指针，不能直接访问到设备信息结构体，但由于work是设备信息结构体的成员，可以通过container\_of函数根据成员地址求出包含它的设备信息结构体的首地址）。在卸载模块函数中，添加上free\_irq函数删除并禁止按键中断，del\_timer\_sync函数删除定时器。

3、编写测试应用程序。和以前的差不多，打开文件后，循环读取文件，如果得到的按键值有效就打印出来，最后关闭文件。

4、实验测试完全成功。这次实验比较复杂，在使用中断的同时，还用到了以前的定时器和原子变量。

**阻塞和非阻塞IO实验（Ubuntu：15\_blockio文件夹、16\_nonblockio文件夹）**

在获取不到设备资源的情况下，阻塞式IO会将对应线程挂起直到资源可以获取再唤醒；非阻塞式IO对应线程不会挂起，要么一直轮询等待资源可获取，要么直接超时放弃返回错误信息。

应用程序打开文件，open("xxxxx", O\_RDWR)默认是阻塞方式打开，所以读写都是阻塞式IO；而open("xxxxx", O\_RDWR|O\_NONBLOCK)是非阻塞方式打开，读写都是非阻塞式IO。

系统使用等待队列来实现阻塞进程的唤醒工作。驱动加载时会定义等待队列头wait\_queue\_head\_t结构体，并使用init\_waitqueue\_head函数初始化等待队列头。每个进程访问驱动时都会通过DECLARE\_WAITQUEUE函数定义初始化一个wait\_queue\_t结构体的等待队列项，当资源不可用时会将进程对应的等待队列项添加到等待队列头中再设置休眠状态并开启任务调度，当设备可用时唤醒设置运行状态再从等待队列头中移除等待队列项（等待队列的功能相当于记录哪些进程阻塞休眠了）。add\_wait\_queue函数添加等待队列项，remove\_wait\_queue函数移除等待队列项。当设备可用时，触发中断，在中断处理中使用wake\_up函数或者wake\_up\_interruptible函数来唤醒休眠进程（前者函数将等待队列头中全部进程唤醒，后者函数唤醒等待队列头中状态为TASK\_INTERRUPTIBLE的进程）。除了直接唤醒等待队列项外，还可以设置唤醒条件，让进程休眠等待某个事件，事件发生后才能被唤醒（使用wait\_event\_xxx函数使进程在事件未发生前一直休眠，事件发生后才可通过wake\_up唤醒）。

应用程序通过select、poll和epoll函数实现轮询机制，检测输入的文件描述符集合是否有可以读或写或异常的文件，通过返回值判断哪些文件可以开始操作，以此进行非阻塞的方式访问那些设备。应用程序使用的三个轮询函数都对应驱动里file\_operations中的一个poll操作函数。poll函数中会调用poll\_wait函数，就是将等待队列头添加到应用程序传进来的poll\_table中，当轮询没有可用设备时，会让进程休眠直到轮询超时或者有事件发生再次检查设备可用性，这需要用到等待队列头。

实验步骤：

1、阻塞式IO实验。在14\_irq实验的基础上添加上阻塞式IO代码。定义一个等待队列头，并在驱动加载函数中初始化它。在read函数中，如果没有可以获取的按键值，有两种方法实现阻塞。第一种通过wait\_event\_interruptible函数休眠等待按键值有效，有效后，通过按键中断引发的wake\_up函数实现唤醒并返回按键值。第二种定义并初始化一个该进程的等待队列项，将等待队列项加入等待队列头中国，设置进程状态为TASK\_INTERRUPTIBLE并开启调度进入休眠，当按键中断函数判断按键值有效后引发wake\_up函数，唤醒该进程设置进程状态为TASK\_RUNNING，将等待队列项移出等待队列头并并返回按键值。测试应用代码不需要修改。

2、非阻塞IO实验。在原先实验基础上，在read函数中判断filp->f\_flags的O\_NONBLOCK位是否为1，是则表明为非阻塞访问，如果没有可获取的按键值直接返回错误，如果有则返回按键值。重写file\_operations的poll函数，将等待队列头加入输入参数poll\_table中，判断有无按键值，有则返回POLLIN|POLLRDNORM；无则返回0。在应用测试代码中，打开文件函数标志位加入O\_NONBLOCK，表示非阻塞方式打开。读取时，先利用select函数或者poll函数，轮询判断需要访问的文件描述符是否可以读取，可以则读取数据，否则继续轮询。

3、实验测试结果完全正确。不使用阻塞或者非阻塞机制的IO操作将占用99%的CPU，而使用阻塞或者非阻塞机制的IO操作后CPU占用几乎为0%。

**异步通知实验（Ubuntu：17\_async文件夹）**

先前实验是应用程序通过不断循环来读取按键值，但这样CPU的使用率比较高，为了降低使用率采用了阻塞IO或者非阻塞IO的方法。但这些方法还都是应用程序主动去读取按键值。信号，相当于软件层次的“中断”，让系统可以向应用程序发送通知，应用程序接收信号再被动处理信号。通过信号通知应用程序去操作的方法就是异步通知。

异步通知（设备通知自己访问）、阻塞（自己等待访问设备）和非阻塞（自己查询访问设备），它们是不同场景的不同解决办法，没有优劣之分。系统一共定义了32种信号，比较常用的是终止进程信号SIGKILL，暂停进程执行信号SIGSTOP，允许输入输出信号SIGIO，终端中断信号（Ctrl+C）SIGINT。除了SIGKILL和SIGSTOP信号外，其余信号都可以被忽略（对信号不做处理）。

应用程序中通过signal函数可以向指定信号设置信号处理函数。通过fcntl函数控制文件描述符（即控制设备驱动），设置当前进程接收文件描述符的信号，并获取当前文件描述符的状态并修改开启异步通知，此步会自动调用file\_operations里的fasync成员函数将驱动的异步通知绑定到此文件描述符上（驱动异步通知结构体-->文件描述符-->应用程序）。

驱动程序中需要通过fasync\_struct结构体指针来操作异步通知，实现file\_operations里的fasync成员函数，将异步通知结构体指针绑定到应用程序，利用kill\_fasync函数向应用发送信号。关闭文件时，也需要fasync成员函数将异步通知结构体指针与应用程序解除绑定关系。

实验步骤：

1、编写驱动。在原有的14\_irq实验基础上，在设备信息结构体中添加fasync\_struct结构体指针，不需要初始化。重写file\_operations里的fasync成员函数，在函数中通过调用fasync\_helper函数控制fasync\_struct异步通知结构体指针与文件描述符的绑定和解绑。在close函数中，调用fasync成员函数（fd=-1，on=0）实现异步通知结构体指针与文件描述符的解绑。在计时器处理函数中，判断是否有可获取按键，有则通过kill\_fasync函数发送SIGIO信号。

2、编写应用。在原有的14\_irq实验基础上，打开文件后，通过signal函数设置SIGIO的信号处理函数，处理函数主要负责读取按键值并打印。再通过fcntl函数设置当前进程号接收文件描述符的信号，并修改文件描述符状态开启文件描述符的异步通知功能，此步会自动调用驱动中file\_operations里的fasync成员函数，实现驱动的异步通知与文件描述符的绑定。这样，驱动的异步通知绑定了文件描述符，文件描述符又绑定了此进程，所以应用程序可以接受到驱动发来的信号。

3、测试完全正确。实验小点：打印时如果不加\n不会刷新输出缓存，就会看不到显示，可用fflush(stdout)刷新缓存区。

**platform驱动实验（Ubuntu：18\_platform文件夹、19\_dtsplatform文件夹）**

以前实验的驱动比较简单，就是实现一些IO操作，但当驱动开始变得复杂时，就要考虑到驱动的可重用性和跨平台性能（在不同主控芯片上都可以使用）。

驱动的分离，将控制器驱动和设备驱动分离（比如，每个平台提供一个IIC控制器驱动，MPU6050提供一个驱动程序，不同的平台就可以通过自己的IIC控制器驱动用同一个MPU6050驱动程序去控制MPU6050），大大减少了内核驱动程序的代码量。Linux系统核心层提供了统一的API控制器接口（不同的平台使用自己不同的代码实现了同一控制器的API函数），使用此API接口编写的设备驱动就可以实现跨平台功能。这样，主机控制器驱动和设备驱动就被Linux系统的API接口分离分隔开了。主机驱动一般由半导体厂商提供，所以需要用户写的就是使用Linux系统API接口的设备驱动。

驱动的分层，核心层提供统一的接口，与连接的硬件完全无关；设备层使用核心层提供的接口控制连接的设备。

根据驱动的分离与分层，驱动代码产生了 总线-驱动-设备 的驱动框架。总线部分由Linux内核提供，属于核心层，不需要用户编写，用户需要编写驱动和设备。驱动指的是具体的设备驱动，设备指的是设备的属性。

总线数据类型为bus\_type，注册总线函数bus\_register，注销总线函数bus\_unregister，总线的主要工作是完成总线下的设备和驱动之间的匹配。Linux系统下/sys/bus/下有各种类型的总线目录（i2c、spi、usb等），每个总线目录下有devices/和drivers/两个目录，里面有各种设备和设备的驱动，总线将它们进行匹配。

驱动数据类型为device\_driver，注册设备驱动函数driver\_register，注销设备驱动函数driver\_unregister。向总线注册设备驱动时，会自动对该总线下的所有设备进行匹配检查，如果有匹配成功的，驱动结构体device\_driver里的probe函数就会执行。

设备数据类型为device，注册设备函数为device\_register，注销设备函数device\_unregister。同样，向总线注册设备时，会自动对该总线下的所有设备驱动进行匹配检查，如果有匹配成功的，对应驱动结构体device\_driver里的probe函数就会执行。

IIC、SPI、USB等总线是物理上实际存在的具体总线，但有些外设不能归类进具体的总线，比如定时器、RTC、LCD等，因此，Linux内核创造了一个虚拟的总线：platform总线。platform驱动类型为platform\_driver结构体（内部包含device\_driver结构体成员变量），里面有of\_match\_table、id\_table和driver的name变量用于设备和驱动的匹配，注册使用platform\_driver\_register函数，注销使用platform\_driver\_unregister函数。platform设备类型为platform\_device结构体（内部包含device结构体成员变量），用name成员变量或者设备树的compatible属性来进行设备和驱动的匹配，注册使用platform\_device\_register函数，注销使用platform\_device\_unregister函数（有设备树时就不需要用platform\_device\_register函数来注册设备，而是修改设备树节点）。不同总线的配对match函数是不一样的，platform总线的match函数是先进行设备树形式的匹配，再进行id\_table的匹配，最后进行两个结构体name的匹配。

实验步骤：

1、无设备树platform实验

首先，编写device模块。除了正常的模块框架外，在模块加载函数中注册一个platform总线上的设备（本实验是led设备），需要初始化这个platform\_device结构体。给name赋值用于匹配同名的驱动（无设备树时这是最简单的匹配方式）；给id赋值（不需要id就给-1值）；给dev结构体中的release赋函数用于放置当设备移除时需要的操作；给resource赋resource结构体指针其指向了资源数组首地址（resource结构体也需要单独初始化，本实验主要用它保存led的寄存器地址和长度）；给num\_resources赋值用于表明资源数组的元素个数（ARRAY\_SIZE函数可以直接通过数组首地址计算数组元素个数）。在模块卸载函数中注销这个platform总线上的设备（本实验是led设备）。

其次，编写driver模块。除了正常的模块框架外，在模块加载函数中注册一个platform总线上的驱动（本实验是led驱动），需要初始化这个platform\_driver结构体。给name赋值要和设备的name一样用于匹配；给probe赋驱动设备配对成功函数（驱动和设备匹配成功时执行，里面操作就是IO映射与初始化、注册字符设备号和字符设备驱动、创建设备节点，和以前的字符设备驱动实验一样）；给remove赋驱动设备取消配对函数（无论是注销了设备还是驱动都会执行，操作就是删除设备节点、注销字符设备驱动和字符设备号、关闭IO和IO映射）。在模块卸载函数中注销这个platform总线上的驱动（本实验是led驱动）。probe函数输入参数为指向此匹配设备的platform\_device结构体指针，所以可以得到设备的资源数组，通过platform\_get\_resource函数可以得到指定platform\_device结构体的指定类型资源的指定第几个的资源，通过resource\_size函数可以计算resource结构体中MEM类型资源start到end的地址长度。

最后，应用测试就是以前的点灯控制代码，测试完全成功。

2、有设备树platform实验

由于有设备树，所以不需要编写device模块。编写driver模块，除了正常的模块框架外，在模块加载函数中注册一个platform总线上的驱动（本实验是led驱动），需要初始化这个platform\_driver结构体。给name赋值（无设备树的设备才会用name匹配，这里不会用到）；给probe赋驱动设备配对成功函数（驱动和设备匹配成功时执行，里面操作就是获取设备树节点、获取并申请gpio编号、注册字符设备号和字符设备驱动、创建设备节点，和以前的字符设备驱动实验一样）；给remove赋驱动设备取消配对函数（无论是注销了设备还是驱动都会执行，操作就是删除设备节点、注销字符设备驱动和字符设备号、注销gpio编号）。由于probe函数是匹配成功执行的，输入参数是指向此匹配设备的platform\_device结构体指针，所以可以直接访问此结构体里的dev里的成员of\_node来得到设备树节点，继而得到匹配设备的gpio编号（其实，platform\_device结构体里面有大量的设备信息，并且platform也提供了大量API函数方便用户获取设备各种信息）。

最后，应用测试就是以前的点灯控制代码，测试完全成功。

3、总结

此次实验和以前的有/无设备树的字符设备驱动实验是很相似的，唯一的不同是多了设备匹配这个环节。以前的实验在**模块加载环节**就开始初始化字符设备和字符设备驱动，在**模块卸载环节**释放申请的资源；而这个实验将初始化字符设备和字符设备驱动放到了**匹配成功环节**的probe函数中，将释放申请的资源放到了**匹配取消环节**的remove函数中，而**模块加载环节**只进行platform总线上驱动的初始化和注册，**模块卸载环节**只进行platform总线上驱动的注销。

从代码上看起来只是套了一层platform驱动的皮，以前：模块加载—>创建字符驱动；现在：模块加载—>注册platform驱动、platform驱动匹配设备—>创建字符驱动。从效果上看起来，以前的实验是没有确认设备是否存在，并且驱动和设备的匹配是人为指定的；现在的实验确认了设备的存在（不存在的话就不会匹配了），并且驱动和设备的匹配是Linux系统按照自己的规则自动匹配。实现了驱动程序对自己可驱动设备的自动匹配和驱动。

**Linux自带LED驱动实验（Ubuntu：）**

内核自带的驱动都是通过内核的图形化配置界面上选择使能或者不使能来控制的，在配置界面选中所需要的内核驱动，就会将此驱动编译进内核。

选中了led与gpio的连接后，根目录下.config文件里会出现CONFIG\_LEDS\_GPIO=y，源码驱动文件夹下的Makefile文件里obj-$(CONFIG\_LEDS\_GPIO) += leds-gpio.o也就会变成obj-y += leds-gpio.o，这样就把需要的驱动添加到内核编译目标obj-y中了（leds-gpio.o的对应源码就是leds-gpio.c。如果不选中就会添加到obj-目标里。同理，编译成模块的目标是obj-m）。

内核自带的LED驱动源码也是使用platform\_driver，通过一个宏定义的module\_platform\_driver函数实现注册，此函数就是集合了module\_init函数、module\_exit函数和他们对应的加载卸载函数（加载卸载函数里面只含有注册和注销platform\_driver）。驱动匹配的name是"leds-gpio"，compatible值是"gpio-leds"。这样一看，驱动编译成模块还是编译进内核，代码区别不大。

实验步骤：

1、make menuconfig打开图形化配置界面，通过图形化界面将驱动编译进内核。

2、通过绑定文档在设备树中添加能被Linux自带LED驱动匹配的设备树节点（没有设备树的话就要通过platform\_device\_register来注册设备）。绑定文档可以指导设备树信息该如何编写。linuxleds母节点里compatible = "gpio-leds"设置兼容值。led0子节点里label = "redled"设置灯的标签，default-state = "on"设置默认状态为开，linux,default-trigger = "heartbeat"设置默认触发模式为双闪心跳灯。其他还有些pinctrl、gpios等属性。make dtbs进行编译。（linuxleds是被leds-gpio驱动控制的设备，它的每个子节点代表一个灯的信息）

3、开启系统，进入/sys/devices/platform/linuxleds/leds/redled/目录下，里面就是这个灯的控制信息。cat brightness可以查看当前灯亮灭，cat trigger可以查看当前灯的触发功能。在无触发功能下echo 1 > brightness可以亮灯，echo 0 > brightness可以灭灯。echo none > trigger设置无触发功能，echo heartbeat > trigger设置触发功能为双闪心跳灯。实现了应用层通过写入数据就可以控制灯的各种功能。

**Linux杂项MISC驱动实验（Ubuntu：20\_miscbeep文件夹）**

MISC驱动也叫杂项驱动，主要负责驱动板子上那些无法分类的外设，其本身就是最简单的字符设备驱动。MISC驱动通常嵌套在platform总线下的（MISC是设备无法分类，platform是总线无法分类或不实际存在）。所有MISC设备的主设备号都为10，随着Linux字符设备越来越多，主设备号变得越来越紧张，使用MISC设备驱动就可以缓解此问题。通过miscdevice结构体来注册一个MISC设备，注册MISC设备时会自动创建cdev和/dev下的设备节点，用户只需要设置minor次设备号（255表示由系统动态分配次设备号）、name设备名称和fops操作集。MISC设备注册函数misc\_register，MISC设备注销函数misc\_deregister。

实验步骤：

1、编写设备树节点。

2、编写模块和platform驱动框架，platform驱动需要匹配设备树节点的兼容值。在platform驱动的probe函数中获取匹配设备的设备树节点，得到设备树节点的gpio编号并申请gpio编号，设置gpio为输出默认高电平，注册MISC驱动。在platform驱动的remove函数中注销MISC驱动并释放gpio编号占用。

3、编写应用测试代码，测试完全成功。此项实验与以前实验的不同是，用户不再是一一实现申请设备号、初始化添加cdev并创建类和设备节点，取而代之的是初始化MISC设备驱动。MISC设备驱动初始化需要次设备号、设备名称和文件操作集，由于MISC驱动的设备的主设备号都是10，所以设置以上三项后，MISC初始化过程中会自行申请设备号、初始化添加cdev并创建类和设备节点。这样的好处是用户的代码大大简化，并且注册的设备主设备号都是10，节约了设备号资源（以前实验都是注册一个设备占一整个主设备号），坏处也是此方法只能注册MISC下的设备（主设备号只能是10）。

**Linux INPUT子系统实验（Ubuntu：21\_input文件夹）**

按键、鼠标、键盘、触摸屏等都属于输入设备，Linux内核专门做了一个input子系统框架来处理输入事件。输入设备本质还是字符设备，只是在此基础上套了input框架。input子系统分为input驱动层、input核心层、input事件处理层，内核已经搭建好了input驱动框架，用户需要做的是按照框架要求完善具体的输入设备的驱动代码。

使用input子系统时，首先通过input\_allocate\_device函数申请一个input\_dev结构体表示申请一个input设备。然后初始化input\_dev结构体，其中evbit数组变量用来设置输入事件的类型（同步EV\_SYN、按键EV\_KEY、相对坐标EV\_REL、绝对坐标EV\_ABS、重复EV\_REP），keybit数组变量用来设置按键事件使用的位图（内核中已经定义了大量的键码供用户使用），后面跟着还有其他输入事件类型对应的位图（使用数组是因为一个input设备可能有着多种类型的输入和键码）。初始化完成后通过input\_register\_device函数注册一个input设备。同理，用input\_unregister\_device函数注销input设备，input\_free\_device函数释放申请的input\_dev结构体。

设置完input设备后，用户就需要编写具体的驱动代码来获取输入值或输入事件，然后将输入事件上报给Linux内核（比如，在某个按键的gpio中断处理函数中上报按键事件）。这样，Linux系统就可以得到正确的输入。上报事件函数input\_event函数用于上报指定事件和对应值（通用），input\_report\_key函数用于上报按键事件，同理还有input\_report\_rel函数、input\_report\_abs函数等等。在上报事件以后还需要调用input\_sync函数上报一个同步事件来通知Linux内核上报结束。

由于驱动使用了input框架并且事件上报给了内核，所以应用程序可以直接访问input子系统创建的/dev/input/eventX（X=0,1,2…）设备节点来访问输入事件，通过内核提供的input\_event结构体来文件读取输入事件数据。input\_event结构体里保存了事件发生时间（32位的秒和32位的微妙）、16位的事件类型、16位的事件码（按键事件里就是键码）和32位的值（按键事件里就是按下/释放状态的按键值），这个也就是输入事件的数据格式。

实验步骤：

1、设置设备树，将按键设备树节点的interrupt-parent、interrupts等属性设置好。

2、编写模块和驱动框架。在irq实验的基础上，在设备信息结构体中添加一个input\_dev结构体指针，在加载模块函数中申请input\_dev，初始化input\_dev，最后注册input\_dev结构体。在卸载模块函数中注销input\_dev，释放申请input\_dev结构体。在按键中断函数中开启的消抖定时器处理函数中，根据按键的电平值向Linux内核上报按键事件，最后上报同步事件表明上报结束。注意，此实验无需注册字符设备驱动框架，因为input子系统已经自行创建了设备节点，而应用程序的操作只需要对此设备节点进行文件操作即可。

3、应用测试程序编写。因为驱动使用了input框架，所以input子系统会自行创建/dev/input/eventX（X=0,1,2…）设备节点，应用程序只需要打开此文件进行读取即可。input\_event结构体就是读取的输入事件数据的格式，所以直接read函数写入到input\_event结构体中，就得到了所有数据信息。测试完全成功。

4、Linux自带也有按键驱动程序（这就相当于开机自启动的驱动模块）。在图形化配置界面中使能KEY驱动，就将按键驱动编译进了内核（是个标准的platform驱动），驱动匹配的兼容性值为"gpio-keys"。在设备树中添加对应的按键设备树节点，每个按键设备树节点都该是兼容性值为"gpio-keys"的设备树节点的子节点。父节点设置autorepeat表示使能重复事件，设置#address-cells为<1>和#size-cells为<0>。按键设备树子节点中设置按键标签label，设置pinctrl，设置linux,code = <28>表示按键的键码为28（回车键），设置GPIO引脚编号gpios。通过hexdump /dev/input/event1测试，成功读到了回车键的输入事件（不同板子可能是eventX，X是其他数字）。

**Linux LCD驱动实验（Ubuntu：）**

LCD是Linux系统的非常常见的外设，其搭配QT这样的GUI库可以制作出非常精美的UI界面。和裸机一样，Linux还是通过操作显存来在LCD上进行显示，但Linux系统的内存是需要申请的，并且要保证驱动程序和应用程序访问的显存是同一个物理地址。为此，Linux将显存抽象成设备——帧缓冲framebuffer（fb），当应用程序需要在LCD上显示时就直接读写操作framebuffer，而不需要了解framebuffer具体是如何驱动LCD的。当编写好LCD驱动后会生成一个名为/dev/fbX（X=0-31）的设备，应用程序就可以通过/dev/fbX设备访问LCD（Linux内核默认使能了LCD驱动，有一个/dev/fb0）。

Linux自带的LCD驱动主要分两个部分，LCD控制器驱动（不同芯片的LCD控制寄存器的操作不同）和LCD屏幕参数（不同LCD屏幕的屏幕参数不同）。因为系统移植的是NXP配置的Linux内核，所以内核自带的LCD驱动的LCD控制器驱动是和本芯片符合的，主要是要根据自己屏幕在设备树中修改屏幕设备参数。Linux内核LCD驱动通过读取兼容的设备树节点，初始化配置fb\_info结构体，最后通过register\_framebuffer函数来注册framebuffer设备，注销函数就是unregister\_framebuffer函数。

Linux内核自带设置了10min未操作LCD息屏功能，可以键盘按键唤醒、Linux源码里drivers/tty/vt/vt.c里修改blankinterval变量修改息屏时间、编写应用关闭息屏功能。

实验步骤：

1、LCD屏幕设备树节点设置。在NXP提供的设备树中已经有默认的LCD设备树节点，在其上面修改自己的LCD设备树节点。将pinctrl里LCD的引脚电气属性改为0x49（比原版降低了驱动能力，主要原因是此板子上引脚高驱动会影响到网络功能）。按照自己LCD的参数将正确数据填写到timing0的参数属性中（在设备树节点里没有保存引脚信息，是因为只要将引脚配置为LCDIF功能后，就可以直接控制LCDIF的寄存器来控制这些引脚）。没有设置LCD背光功能，因为背光引脚和NXP原版是相同的，驱动中已经有了背光设置（背光属性是在另一个设备树节点里的）。make dtbs后，就可以看到Linux成功把自己的LCD驱动起来，左上角绘制出默认配置的开机企鹅。输入echo hello linux > /dev/tty1就可以在LCD上打印出来hello linux字符串。

2、将LCD作为终端。在uboot中环境变量bootargs里添加一个console=tty1，表明了tty1和ttymxc0都为控制台，也就是LCD和串口都为控制台，都会打印log信息。打开开发板的根文件系统中的/etc/inittab文件加入tty1::askfirst:-/bin/sh语句，表明tty1也作为shell程序的终端。接上键盘就可以在LCD上进行正常的输入输出（键盘鼠标等驱动都是现代Linux内核默认自带，插上即可驱动设备，但鼠标还需要UI界面才能显示出来）。在/sys/devices/platform/backlight/backlight/backlight里可以查看当前屏幕背光等级，echo 0-7 > brightness可以修改背光等级，也就是调节LCD屏幕的亮度。

**Linux RTC驱动实验（Ubuntu：）**

RTC驱动也是标准字符设备驱动，应用程序通过文件操作来完成对RTC设备的操作。Linux内核将RTC设备抽象成rtc\_device结构体，所以申请并初始化rtc\_device结构体，并将rtc\_device注册到Linux内核里就完成了RTC设备的注册。rtc\_device结构体里有rtc\_class\_ops作为rtc操作集合，最常用的函数为read\_time读时间函数、set\_time设置时间函数、read\_alarm读定时函数、set\_alarm设置定时函数、alarm\_irq\_enable使能定时中断函数。

NXP官方已经写好了自己芯片的RTC驱动，并且编译进了内核中。应用程序直接使用ioctl函数即可完成设置/读取时间、设置/读取定时。

实验步骤：

1、开启开发板。RTC驱动已经在运行了，设备节点为/dev/rtc0。使用date命令可以查看当前时间，使用date --help命令可以查看date命令帮助，使用date -s/--set "2021-09-09 15:15:15"命令可以设置当前时间，使用hwclock -w命令可将当前时间写入到RTC中，下次启动后时间依旧会是实时时间（但要保证有个电量充足的纽扣电池维持RTC工作）。

**Linux IIC驱动实验（Ubuntu：22\_iic文件夹）**

Linux中IIC驱动分为IIC控制器驱动和IIC设备驱动（分离与分层的思维），IIC控制器驱动由半导体原厂已经编写好了，用户需要编写的就是IIC设备的驱动。IIC是物理存在的总线，所有IIC驱动使用的是I2C总线。

IIC控制器驱动核心就是申请并初始化i2c\_adapter结构体，设置i2c\_adapter结构体中i2c\_algorithm结构体的master\_xfer函数，最后注册此i2c\_adapter结构体。i2c\_algorithm结构体里包含的就是访问IIC设备的API接口函数，半导体原厂实现这些函数，用户负责调用这些函数去访问IIC设备，其中，master\_xfer函数就是IIC主机收发数据的API函数。（i2c控制器和i2c控制器驱动是platform总线上的匹配）

IIC设备驱动部分使用i2c\_client结构体来描述一个IIC设备，每检测到一个IIC设备就会给它分配一个i2c\_client结构体。不需要自己创建i2c\_client结构体，而是通过创建i2c设备树节点下的具体设备子节点时（想挂载在哪个i2c控制器下就在哪个节点下面创建子节点，需要哪些具体属性看绑定文档），内核会根据设备树自行创建此设备的i2c\_client结构体。i2c\_client结构体里有i2c\_adapter结构体成员，就是用来表明此IIC设备挂载在哪一个IIC控制器上。IIC设备的驱动由i2c\_driver结构体描述，需要用户自行初始化和注册（注册使用i2c\_register\_driver函数或者i2c\_add\_driver函数，注销使用i2c\_del\_driver函数）。当i2c\_driver和i2c\_client匹配时，i2c\_driver结构体中probe函数会执行；当i2c\_driver和i2c\_client取消匹配时，i2c\_driver结构体中remove函数会执行。（i2c设备和i2c设备驱动是i2c总线上的匹配）

IIC设备驱动对IIC设备的读写依靠i2c\_transfer这个API函数，它有三个参数，第一个是对应的IIC控制器，第二个参数是IIC的传输数据结构体i2c\_msg指针，第三个参数是传输数据结构体i2c\_msg的个数。IIC读需要先发送寄存器地址，再读取数据；而IIC写只需要依次发送寄存器地址和需要发送的数据。

实验步骤：

1、修改设备树。检查i2c的pinctrl的复用引脚是否板子i2c的连接引脚，不是的话需要修改为自己板子连接引脚的i2c复用。在i2c设备树节点下添加ap3216c@1e子节点，设置compatible兼容值属性和reg从机地址属性。注意，一个i2c设备树节点下不允许出现两个从机地址相同的设备子节点出现。make dtbs编译后拷贝到nfs/目录下，开发板开机后，在/sys/bus/i2c/devices/0-001e/目录下就会出现ap3216c的设备信息。

2、编写模块和i2c驱动框架，设置i2c\_driver结构体中兼容值与i2c设备匹配。在匹配成功的probe函数中注册字符设备驱动（remove函数卸载字符设备驱动），以此让应用程序可以文件操作i2c设备。利用i2c\_transfer这个API函数编写i2c读取一个或多个寄存器值，i2c写入一个或多个寄存器值的函数。按照i2c设备的芯片手册，在文件open操作中初始化i2c设备，在read函数中读取所有需要的值，发送给应用程序。应用程序中和往常一样文件打开，循环读取并显示出来。

3、测试没有结果，输出全是最大值，因为mini开发板上没有这个iic设备芯片。。。

**Linux SPI驱动实验（Ubuntu：23\_spi文件夹）**

SPI控制器驱动和IIC控制器驱动一样，内核用一个spi\_master结构体来表示SPI控制器（主机）驱动。控制器驱动代码的核心依旧是申请spi\_master结构体，初始化结构体，最后向内核注册spi\_master结构体，而spi\_master结构体中的transfer函数就是SPI控制器传输数据的函数。与IIC控制器驱动一样，这个部分已经由半导体原厂编写好了，不需要用户编写。（同样，主机（控制器）与主机驱动（控制器驱动）的匹配是在platform总线上的）

SPI设备驱动也类似于IIC设备驱动，Linux内核使用spi\_driver结构体来表示spi设备驱动，同样有着匹配的probe函数、取消匹配的remove函数和包含设备驱动信息的driver。spi\_driver初始化后需要向Linux内核通过spi\_register\_driver函数注册，注销使用spi\_unregister\_driver函数。spi设备和spi设备驱动的匹配也十分相同，无设备树时匹配设备名字和spi\_device\_id的name，有设备树时匹配设备compatible属性和of\_device\_id的compatible属性。（同样，spi设备与spi设备驱动的匹配是在spi总线上的）

Linux内核用spi\_transfer结构体来描述SPI需要发送的数据，结构体里有接收数据数组指针、发送数据数组指针和数据长度（SPI通信规定收发数据的长度必须一样）。Linux内核用spi\_message结构体来描述SPI发送的消息队列，spi\_transfer里的数据的发送需要先将自己添加到spi\_message队列尾部，然后将spi\_message交给内核进行SPI发送。spi\_message结构体通过spi\_message\_init函数初始化，通过spi\_message\_add\_tail函数将spi\_transfer添加到spi\_message的队列尾部，同步传输spi\_sync函数和异步传输函数spi\_async函数（区别是同步传输会阻塞等待传输完成，而异步传输不会等待）。

输入如下命令使能硬件浮点编译：arm-linux-gnueabihf-gcc -march=armv7-a -mfpu=neon -mfloat-abi=hard icm20608App.c -o icm20608App。

实验步骤：

1、在设备树中添加有关spi3的引脚复用pinctrl。在ecspi3节点下创建设备icm20608@0节点。在ecspi3节点中追加片选数量、片选引脚cs-gpio、pinctrl和status值为okay。在icm20608@0子节点中添加兼容性值、最大时钟频率和spi通道号。

2、编写模块和spi驱动框架。与上个实验类似，初始化spi\_driver的probe函数、remove函数、无设备树的匹配表id\_table和driver里的name、有设备树的匹配表of\_match\_table。在probe函数中，首先设置spi\_device里的mode为SPI\_MODE\_0，表示spi为模式0，然后通过spi\_setup函数配置spi\_device，最后将他作为字符设备的私有数据。其次，从spi\_device里的device得到spi设备的设备树节点，然后通过of\_get\_parent函数得到其母节点，再得到母节点的cs-gpio的编号，申请编号并电平拉高，表示未使能片选。最后，创建字符设备驱动。remove函数里相应地释放所有资源。在文件操作open函数中，初始化icm20608，在read函数中读取icm20608的所有测量数据，它们都可使用一套通用的spi读写函数来完成。

编写通用的spi读写数据函数，读函数中，创建spi\_message和spi\_transfer，设置spi\_transfer发送数据数组、接收数据数组和数据长度，第一步先发送读取的寄存器地址，在spi\_message通过spi\_message\_init函数初始化后，将spi\_transfer通过spi\_message\_add\_tail函数添加到spi\_message的队列中，通过spi\_sync函数传输出去，第二步再接收数据，同样方式设置spi\_tansfer和spi\_message并传输；在写函数中，创建spi\_message和spi\_transfer，设置spi\_transfer发送数据数组、接收数据数组和数据长度，第一步先发送写入的寄存器地址，在spi\_message通过spi\_message\_init函数初始化后，将spi\_transfer通过spi\_message\_add\_tail函数添加到spi\_message的队列中，通过spi\_sync函数传输出去，第二步再发送具体数据，同样方式设置spi\_tansfer和spi\_message并传输。寄存器地址为7位，发送的寄存器地址最高位是1表示读取，最高位是0表示写入。

3、编写应用程序。和上个实验一样，打开文件，循环读取并显示读到的值。

4、测试没有结果，输出不会发生变化，因为mini开发板上没有这个spi设备芯片。。。

**Linux 串口驱动实验（Ubuntu：）**

与IIC、SPI一样，Linux也提供了串口驱动框架，用户只需要按照框架编写代码即可。串口驱动没有主机端和设备端，就只有一个串口驱动，而且NXP原厂已经编写好了，用户只要在设备树中添加串口设备节点信息，系统就会自动实现设备和串口驱动的匹配，生成/dev/ttymxcX（X=0…n）节点。串口驱动的核心依旧是初始化uart\_driver结构体，向uart\_driver结构体中添加一个uart\_port结构体（uart\_port里的uart\_ops有着具体的原厂编写的串口驱动函数），最后注册uart\_driver结构体。（uart驱动和uart设备的匹配是在platform总线上的匹配）

由于实验只是使用内核的uart驱动，实现串口通信，所以不需要用户编写代码，NXP原厂的串口驱动就能实现收发数据。相比于IIC和SPI，这个原厂的驱动已经将串口表示为设备文件，应用程序可以直接对其读写了。这归功于串口就只是读取对方发送来的数据，不像IIC、SPI那样需要自己指定读取写入哪个寄存器，所以串口可以将通信协议包装到原厂的驱动中，而IIC、SPI需要在设备驱动中实现具体的读写。

实验步骤：

1、创建uart3的设备树节点。设备树文件默认只有uart1和uart2的节点信息，需要自己添加uart3的节点信息。在iomuxc中添加uart3两个引脚的IO复用配置pinctrl子节点，确保只有一个pinctrl节点配置了这两个引脚。然后添加uart3的信息，只需添加pinctrl和status="okay"即可，uart3的兼容值属性等其他有关属性在dtsi文件中就已经配置好了。启动开发板，操作/dev/ttymxc2设备文件就是在操作uart3。

2、系统中出现了/dev/ttymxc2设备文件，但是没有具体串口设备，实验无法测试。。。

**Linux 多点电容触摸屏驱动实验（Ubuntu：24\_multitouch文件夹）**

电容触摸屏为IIC接口，控制芯片为FT5426，其实就是与此芯片进行IIC通信，并且芯片提供中断引脚来通知触摸信息，提供的信息为触摸的绝对位置和是否按下状态。

在input子系统中，触摸点信息通过一系列的ABS\_MT事件上报给Linux内核，在这些众多的ABS\_MT事件中，最常用的就是ABS\_MT\_SLOT、ABS\_MT\_POSITION\_X、ABS\_MT\_POSITION\_Y和ABS\_MT\_TRACKING\_ID。其中ABS\_MT\_POSITION\_X和ABS\_MT\_POSITION\_Y用来上报触摸点的(X,Y)坐标信息，ABS\_MT\_SLOT用来上报触摸点ID，对于Type B类型的设备，需要用到ABS\_MT\_TRACKING\_ID事件来区分触摸点（Type B类型指的是能追踪并区分触摸点的触摸设备）。此外，还有ABS\_MT\_TOOL\_TYPE事件用于上报触摸工具类型，目前支持的是手指、笔、手掌。

Type B类型的设备的上报时序是，首先使用input\_mt\_slot函数上报ABS\_MT\_SLOT事件，把要上报的触摸点的slot先上报，slot由触摸芯片提供；其次使用input\_mt\_report\_slot\_state函数用来添加、修改、删除触摸点（如果是添加，内核会给触摸点自动分配ABS\_MT\_TRACKING\_ID用于区分不同触摸点）；使用input\_report\_abs函数上报触摸点X/Y轴坐标；然后重复上述操作来上报其他触摸点；最后全部上报完毕后使用input\_sync函数表示上报完毕。此外，还需要input\_mt\_init\_slots函数用于初始化MT的输入slots，编写MT驱动的时候必须先调用此函数初始化slots；当芯片追踪的触摸点数量多于驱动上报的最大数量时，需要调用input\_mt\_report\_pointer\_emulation函数将use\_count参数设置为false来告知用户空间芯片追踪的触摸点总数量。

实验步骤：

1、修改设备树。触摸芯片用到了4个引脚，一个复位IO、一个中断IO和I2C的SCL与SDA，所以先添加pinctrl复用设备树节点。在iomuxc下的pinctrl\_tsc节点下添加中断引脚复用（pinctrl\_tsc节点是默认就有的，用于电阻触摸屏的引脚初始化，现改为电容触摸屏引脚复用），复位引脚由于是SNVS\_TMPER9，所有在iomuxc\_snvs节点里创建其的pinctrl子节点。最后，芯片连接的是I2C2的引脚，它们默认已经在pinctrl\_i2c2里被配置了，不需要修改。在I2C2下添加FT5426设备子节点，配置pinctrl、reg、interrupt和gpio。

2、编写触摸屏驱动。同样编写模块加载卸载函数调用i2c\_drvier加载卸载函数，初始化i2c\_drvier，编写probe函数和remove函数。在probe函数中获取中断和复位引脚，将复位引脚拉低再拉高完成芯片的复位，i2c通信初始化芯片，初始化并注册input设备，初始化中断和中断处理函数（注册中断处理函数使用了devm\_request\_threaded\_irq函数，表示资源系统自动释放，并且中断处理线程化，可以被调度）。remove函数中只需要注销input设备即可，因为申请IO编号使用的devm\_gpio\_request\_one函数，有devm前缀的函数表示申请的资源系统会自动释放，不需要手动释放。在中断处理函数中，连续读出芯片29寄存器的值，组合拼接出5个触摸点的触摸类型、触摸点id和X/Y的绝对坐标，将它们上报给内核即可。

3、实验测试，hexdump /dev/input/event2可以查看到触摸屏被触摸时上报给内核的事件，完全成功。尝试将驱动编译进内核，将ft5426.c文件拷贝到内核源码目录下drivers/下的对应路径下（比如：drivers/input/touchscreen/），在此路径下的Makefile文件中加入一行obj-y = ft5426.o并保存。重新编译内核后，启动开发板，该驱动已经默认运行了，虽然变成了/dev/input/event1，但测试依旧完全正确。

**Linux USB驱动实验（Ubuntu：）**

USB只能进行主机和设备（从机）之间的通信，所以使用不同的插头和插座来区分USB主机和设备，主机一般使用USB A插座，而从机提供Mini USB、Micro USB或者USB TypeC。USB主机最多可以支持127个设备，USB主控制器支持128个地址，地址0不分配给设备，所以最多分配127个地址（因此可以通过USB HUB扩展USB接口）。USB主机和从机之间的通信通过管道(Pipe)来完成，任何一个USB设备一旦上电就会存在一个管道，也就是默认管道，USB主机通过管道来获取从机的描述符、配置等信息。在主机端管道其实就是一组缓冲区，用来存放主机数据，在设备端管道对应一个特定的端点。

支持USB OTG功能的USB接口一般需要有5条线，其中一条为ID线，因此才既可以做主机（ID线高电平），也可以做从机（ID线低电平）。

当USB设备与USB主机连接以后主机就会对USB设备进行枚举，通过枚举来获取设备的描述符信息，主机得到这些信息以后就知道该加载什么样的驱动、如何进行通信等。USB枚举过程如下：

①当USB主机检测到USB设备插入以后会发出总线复位信号来复位设备。USB设备复位完成以后地址为0，主机向地址0的端点0发送数据，请求设备的描述符。设备得到请求以后就会按照主机的要求将设备描述符发送给主机，主机得到设备发送过来的设备描述符确认无误后，就会向设备返回一个确认数据包(ACK)。

②主机再次复位设备，进入地址设置阶段。主机向地址0的端点0发送设置地址请求数据包，新的设备地址就包含在这个数据包中，因此没有数据过程。设备进入状态过程，等待主机请求状态返回，设备收到以后就会向主机发送一个0字节状态数据包，表明设备已经设置好地址了，主机收到这个0字节状态数据包以后会返回一个确认包(ACK)。设备收到主机发送的ACK包后就会使用这个新的设备地址，至此设备就得到了一个唯一的地址。

③主机向新的设备地址端点0发送请求设备描述符数据包，这一次主机要获取整个设备描述符，一共是18个字节。

④和第③步类似，接下来依次获取配置描述符、配置集合、字符串描述符等等。

实验步骤：

1、USB HOST实验（开发板作为主机，外接USB设备）。内核已经默认使能了USB键盘鼠标驱动（图形化界面里的Generic HID driver和USB HID transport layer），支持键盘、鼠标、摇杆、绘图板等HID设备，但此驱动不能和HIDBP键盘鼠标驱动一起使用。接上鼠标/键盘都可以进行在/dev/input/eventX中读取输入事件，键盘还可以操控LCD上的控制台。内核已经默认使能了U盘驱动（图形化界面里的SCSI disk support和USB Mass Storage support），接上U盘，内核打印出U盘容量和设备节点/dev/sd\*，通过mount挂载就可以正常读写U盘了，拔出前先sync同步，再umount卸载U盘。

2、USB OTG从机实验（OTG主机和USB HOST一样，所以这里是开发板作为从机，接入USB主机）。将开发板模拟成U盘设备，在设备树中pinctrl\_hog\_1里添加GPIO1\_IO00的复用为OTG1\_ID（GPIO1\_IO00作为ID线要默认下拉，因为OTG默认工作主机模式），图形化配置界面中将USB Gadget Support和Mass Storage Gadget编译成[M]模块，然后编译内核。启动开发板，电脑和开发板接上mini USB的线，开发板外接个U盘，然后按顺序加载libcomposite.ko、usb\_f\_mass\_storage.ko模块，最后加载modprobe g\_mass\_storage.ko file=/dev/sda1 removable=1并且指定大容量存储设备文件为/dev/sda1，也就是外界的U盘，不能指定EMMC或者NAND，因为EXT3/EXT4文件格式windows不兼容会被格式化。电脑可以成功通过开发板读写U盘，表明可以把开发板作为一个USB从机。

**Linux 块设备驱动实验（Ubuntu：25\_ramdisk\_request文件夹、26\_ramdisk\_norequest文件夹）**

前面的实验都是字符设备驱动，而块设备驱动比字符设备驱动复杂的多，不同类型的存储设备又对应不同的驱动子系统。块设备针对存储设备，比如SD卡、EMMC、NAND Flash、Nor Flash、SPI Flash、机械硬盘、固态硬盘等等。对于不同的存储设备，Linux有着不同的I/O算法。

块设备只能以块为单位进行读写访问，块是Linux虚拟文件系统（VFS）基本的数据传输单位，块设备在结构上可以随机访问，使用缓冲区暂时存放数据。（字符设备以字节为单位的顺序的数据流设备，不需要缓冲，实时访问）

内核使用block\_device结构体来表示块设备，其中有个叫做bd\_disk的gendisk结构体指针用于指向通用磁盘结构gendisk。和字符设备注册注销一样，注册块设备register\_blkdev函数，注销块设备unregister\_blkdev函数。gendisk结构体用于描述一个磁盘设备，其中的信息主要有主设备号、第一个次设备号、次设备号数量、磁盘分区表、块设备操作集、磁盘的请求队列等。alloc\_disk函数申请gendisk，del\_disk函数释放gendisk，add\_disk函数将gendisk添加到内核，set\_capacity函数设置gendisk磁盘容量，get\_disk函数增加gendisk的引用计数，put\_disk函数减少gendisk的引用计数（引用计数一般不需要驱动操作，gendisk结构体感觉相当于字符设备的cdev结构体，但是比它复杂得多）。和字符设备的file\_operations结构体一样，块设备操作集为block\_device\_operations结构体，主要有owner变量表明结构体从属，open函数打开块设备，release函数关闭块设备，rw\_page函数读写指定页，compat\_ioctl和ioctl函数块设备的I/O控制，getgeo函数获取磁盘信息。

内核把对块设备的读写操作都发送到其对应的gendisk里的request\_queue请求队列中，request\_queue请求队列包含一系列的request请求结构体，每个request结构体里有多个bio结构体，具体读写数据信息和地址就保存在bio中。一个bio对应的块设备地址是连续的，不同的块设备地址需要多个bio，bio有最大数据量限制，超过限制就需要多个bio，每个bio里有多个bio\_vec，bio\_vec里保存着缓冲区起始页地址、偏移地址、数据长度，所以一个bio\_vec对应的缓冲区地址是连续的。（blk\_cleanup\_queue函数可以清除请求队列）

request\_queue： 块设备地址不连续 缓冲区地址不连续 数据量大 读写

->request： 块设备地址连续 缓冲区地址不连续 数据量大 读写

->bio： 块设备地址连续 缓冲区地址不连续 数据量小 读/写

->bio\_vec： 块设备地址连续 缓冲区地址连续 数据量小 读/写

对于机械硬盘这样的随机访问会降低效率的存储设备，需要给请求队列分配一个I/O调度器，通过blk\_init\_queue函数进行队列的初始化，并且绑定一个请求处理函数。应用读写此类块设备的操作会将bio提交给I/O调度器，I/O调度器会将bio合并到队列现有的request里或者包装成新的request插入队列，再由请求处理函数处理队列（目的是优化此类设备的访问顺序，提高读写效率）。

而对于EMMC、SD卡这样可以完全随机访问的存储设备，不需要I/O调度器和请求队列（但会申请队列用于绑定制造请求函数），通过blk\_alloc\_queue函数申请队列，再通过blk\_queue\_make\_request函数绑定一个制造请求函数。应用读写此类块设备的操作提交的bio会由制造请求函数直接处理。

在请求处理函数中，会从请求队列里一个个取出请求，然后得到每个请求的bio，最后根据bio的描述去读写块设备，blk\_fetch\_request函数用于请求的获取和开启。

在制造请求函数中，因为不需要I/O调度器和请求队列，所以直接去处理每个bio，bio\_for\_each\_segment函数遍历bio的所有bio\_vec段，bio\_endio函数通知内核bio处理结果。

实验步骤：

1、请求队列实验（针对于机械硬盘这样需要I/O调度的块设备）。编写模块框架，在模块加载函数中申请一段内存（当作是一个块设备），注册块设备的设备号，初始化请求队列和队列需要的锁与请求处理函数，最后初始化并注册gendisk（操作集实现open、release、getgeo）。模块卸载函数中就反顺序依次释放申请的gendisk、清除请求队列、注销块设备、释放申请的内存。在请求处理函数中，循环得到队列中每一个请求，处理并判断请求是否正确处理完毕，直到获取完所有请求，对于每个请求读取其起始地址、操作的字节数以及数据缓冲区，判断请求是读还是写，如果是读，从申请内存地址+起始地址处拷贝到数据缓冲区；如果是写，从数据缓冲区拷贝到申请内存地址+起始地址处。测试完全成功，fdisk -l可以显示出此块设备的信息，并且可以格式化、挂载、在里面改写文件。

2、不使用请求队列实验（针对于EMMC、SD卡这样完全随机访问的块设备）。代码框架几乎完全一样，只是在初始化请求队列和队列需要的锁与请求处理函数这里，改为分配并初始化请求队列和绑定请求制造函数。编写请求制造函数，在函数里面获取bio的块设备地址，然后循环处理bio中的每个bio\_vec段的读写操作并偏移块设备地址，全部处理完后，数据传输完成标志置位，向内核返回处理结果。测试完全成功，fdisk -l可以显示出此块设备的信息，并且可以格式化、挂载、在里面改写文件。

3、和字符设备驱动相比，对块设备的操作依靠的，除了gendisk关联一个操作集外，多了一个gendisk关联请求队列的请求处理函数/制造请求函数，猜测是因为机械硬盘需要I/O调度提高效率而并不是按调用的先后顺序进行读写，所以多了一个按队列顺序读写块设备的函数，可以随机访问的块设备也还是统一沿用了此框架。

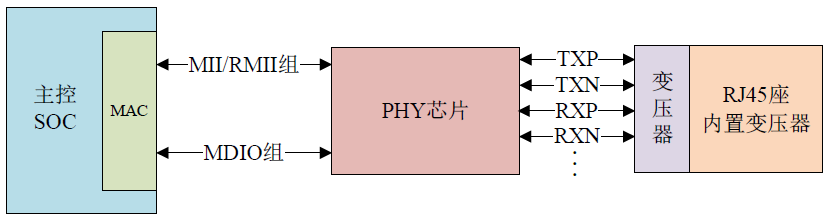
**Linux 网络设备驱动实验（Ubuntu：）**

以前计算机上网需要网卡，而现在技术的发展，只需要一个芯片就能完成以前网卡的功能。嵌入式网络硬件分为两部分：MAC和PHY。一款芯片如果说自己支持网络表明其内置了MAC（MAC类似IIC、SPI控制器这样的外设），但也需要外置PHY芯片才能实现网络功能，主控芯片和PHY芯片通过MII/RMII接口传输数据，通过MDIO接口进行配置控制。没有内置MAC的芯片可以选择外置MAC+PHY一体化网络芯片（有些芯片甚至集成了硬件TCP/IP协议栈，简化了联网方案），缺点是MAC+PHY网络芯片比PHY芯片贵很多，主控芯片网络效率不高（内置MAC的主控芯片一般会有网络加速引擎，比如专用DMA）。

MII接口一共16条线，RMII接口是MII接口的精简版本，只需要7根线：发送使能信号线、2\*发送数据信号线、2\*接收数据信号线、接收有效/载波侦听信号线、参考时钟线。

MDIO接口有两根线：MDIO数据线和MDC时钟线，驱动可通过这两根线访问PHY芯片任意寄存器。和IIC一样，MDIO根据器件地址访问对应的PHY芯片，最多支持32个PHY芯片，但同一时刻只能操作一个PHY芯片。

插入网线的座子叫RJ45座，RJ45座通过一个网络变压器隔离滤波，连接到PHY芯片上，现在也有内置网络变压器的RJ45座（比如：HR911105A）。PHY芯片控制RJ45的两个灯，绿灯亮表示网络连接正常，黄灯闪烁表示正在网络通信。



PHY芯片寄存器地址空间为5位，共32个寄存器，IEEE定义了前16个寄存器的功能，后16个由厂家定义，无论是哪个厂家的PHY芯片，通过控制定义相同的前16个寄存器都可以驱动起来PHY芯片，所以Linux内核中有着通用的PHY驱动。但由于有些厂商的PHY芯片有着更加强大的性能，厂商会提供自己PHY芯片的驱动源码，所以Linux内核中也有着一些具体PHY芯片的驱动。

Linux内核使用net\_device结构体来表示一个网络设备，网络驱动的核心就是初始化net\_device的各个成员变量，然后将初始化完成的net\_device注册到Linux内核中。net\_device结构体里的netdev\_ops是网络设备的操作集（类似字符设备的file\_operations），ethtool\_ops是网络管理工具的操作集（用于配置网卡），header\_ops是头部操作集（创建、解析、缓冲等）。通过alloc\_netdev函数申请net\_device（此函数对网络设备通用，对于不同类型的网络也有对应封装的申请函数），ether\_setup函数对以太网类型设备的net\_device进行初步初始化，通过free\_netdev函数释放申请的net\_device，通过register\_netdev函数注册net\_device，通过unregister\_netdev函数注销net\_device。netdev\_ops里主要有加载网络设备时执行的ndo\_init函数、卸载网络设备时执行的ndo\_uninit函数、打开网络设备ndo\_open函数、关闭网络设备ndo\_stop函数、发送数据ndo\_start\_xmit函数。sk\_buff结构体用于保存网络数据，发送时各个协议层在sk\_buff里添加自己的协议头，最后由底层驱动将其发送出去，接收时过程恰好相反。上层网络数据发送函数使用dev\_queue\_xmit函数，网络数据接收使用函数netif\_rx函数。为某网络设备申请sk\_buff的函数为netdev\_alloc\_skb函数，释放sk\_buff函数为dev\_kfree\_skb函数，skb\_put函数扩展sk\_buff的尾部数据区，skb\_push函数扩展sk\_buff的头部数据区，skb\_pull函数删除sk\_buff的头部数据区，sbk\_reserve函数调整头部缓冲区的大小。

接收数据一般有两种方法：中断和轮询。Linux提出了另一种网络数据接收的方法NAPI，核心思想是不全部采用中断来读取网络数据，而是采用中断来唤醒数据接收服务程序，在接收服务程序中采用POLL的方法来轮询处理数据。这种方法的好处就是可以提高短数据包的接收效率，减少中断处理的时间。在使用NAPI前，要通过netif\_napi\_add函数初始化napi\_struct结构体，netif\_napi\_del函数删除NAPI，napi\_enable函数使能NAPI，napi\_disable函数关闭NAPI，napi\_schedule\_prep函数检查NAPI是否可以调度，\_\_napi\_schedule函数NAPI调度，napi\_schedule函数检查NAPI是否可以调度可以则调度（napi\_schedule=napi\_schedule\_prep+\_\_napi\_schedule），napi\_complete函数标记NAPI处理完成。

Linux的网络驱动非常复杂，也是分为MAC控制器驱动和PHY设备驱动。控制器驱动由NXP官方提供，主要实现了初始化并注册platform\_driver结构体，在paltform总线上与设备树中fecX节点匹配，然后获取各种属性，初始化并注册net\_device结构体，初始化enet并添加napi，初始化mii，初始化并注册phy\_device结构体（超级复杂！！）。PHY设备驱动由Linux自带的驱动源码提供（需要图形化界面使能驱动），主要实现了初始化并注册phy\_driver结构体，在MDIO总线上与phy\_device结构体匹配，实现了大量驱动函数。

实验步骤：

1、由于控制器驱动和PHY设备驱动都有提供（主要是编写实在是太复杂困难了），所以不需要自己编写代码。在图形化配置界面使能Drivers for SMSC PHYs（开发板使用的LAN8720A芯片是SMSC公司的），编译内核并启动开发板。

2、已有驱动的修改，默认的驱动还不能成功启动网络，而修改方案在Linux内核移植的网络问题时已经解决好了（主要是控制器驱动里需要多配置两个引脚）。打开开发板，可以看到PHY的驱动信息。

3、修改ip地址。嵌入式Linux系统修改ip地址，没有普通Linux系统那些简单的配置文件。设置静态ip方法，一种是在uboot环境变量bootargs里把需要设置网卡的ip地址直接设置好（如ip=192.168.0.50:192.168.0.66:192.168.0.1:255.255.255.0::eth0:off）；另一种是利用ifconfig命令，如ifconfig eth0 192.168.0.66 netmask 255.255.255.0，这样设置的ip地址重启会失效，所以可以将此命令写入/etc/init.d/rcS中开机自动执行。设置动态ip的方法，在busybox源码中找到examples/udhcp/simple.script，将其拷贝到开发板的/usr/share/udhcpc目录下(没有则自行创建此目录)，并重命名为default.script，这样之后就可以输入ifconfig eth0 up命令打开eth0网卡，输入udhcpc -i eth0命令来为eth0网卡申请动态ip。测试完全成功，电脑、虚拟机、网站均可ping通。（动态ip设置不要去修改连接虚拟机文件系统的那个网卡，不然会与虚拟机断开连接，系统无响应）

**Linux PWM驱动实验（Ubuntu：）**

IMX6ULL有8路PWM输出，对应着8个PWM控制器，设备树下有8个PWM控制器节点。Linux提供了PWM子系统的框架，核心是利用pwm\_chip结构体控制PWM，结构体中的pwm\_ops结构体是PWM控制器的操作集，主要函数为配置周期和占空比的config函数、使能pwm的enable函数、关闭pwm的disable函数。PWM子系统驱动核心为初始化pwm\_chip结构体，并通过pwmchip\_add函数向内核注册PWM设备，注销使用pwmchip\_remove函数。（这些驱动NXP原厂已经编写好了，因为PWM的输出不需要关心连接的是什么具体设备，所以PWM可以直接被PWM子系统驱动包装成文件）

实验步骤：

1、修改设备树。要将GPIO1\_IO04作为PWM3输出，首先iomuxc下添加GPIO1\_IO04复用PWM3的引脚信息。然后在pwm3节点下追加pinctrl初始化，设置ipg和per时钟，状态处于开启。检查没有GPIO1\_IO04的其他引脚配置和使用后，make dtbs编译设备树，用新设备树来启动系统。

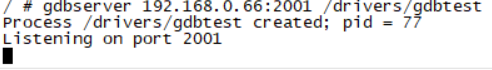
2、使能PWM驱动。在图形化配置界面中，使能i.MX PWM support选项，保存并编译系统，用新的zImage来启动系统。（默认情况下就是开启的）

3、GPIO1\_IO04引脚将输出pwm信号，通过echo 0 > /sys/class/pwm/pwmchip2/export命令调出pwm0子目录（pwmchip0-7对应的就是芯片的PWM1-8），通过命令进入pwm0子目录（cd /sys/class/pwm/pwmchip2/pwm0/）。echo 1 > enable命令使能PWM3功能，echo 0 > enable命令关闭PWM3功能，echo 50000 > enable命令设置pwm的周期值（单位ns，50000ns就是20KHz），echo 10000 > duty\_cycle命令设置pwm的高电平时间（单位ns，10000ns高电平在50000ns的周期中就是20%占空比）。由于没有示波器，无法测试引脚的输出是否和设置一样，但应该是准确的。。。

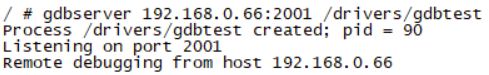
**嵌入式Linux GDB调试工具**

前面这么多实验都是通过编译报错或运行报错，通过打印信息逐渐地查找代码bug，而gdb工具是GNU项目调试器，基于命令行。和其他的一样，可以使用gdb来一行行的运行程序、单步执行、跳入/跳出函数、设置断点查看变量等，它是UNIX/LINUX操作系统下强大的程序调试工具。桌面Linux系统可以直接运行gdb工具，但是嵌入式Linux系统性能较弱，只能通过在开发板移植gdbserver并通过网络和桌面端的gdb通信进行调试。一般交叉编译器就自带了gdb和gdbserver，所以只需要将桌面端交叉编译器的gdbserver拷贝到开发板的/bin目录下就可以了。

编写一个简单的gdbtest.c文件，交叉编译此gdbtest.c文件，注意：想要用gdb调试必须加上-g选项，比如arm-linux-gnueabihf-gcc gdbtest.c -o gdbtest –g，将得到的gdbtest拷贝到开发板中。启动开发板，输入命令gdbserver 192.168.0.66:2001 /drivers/gdbtest测试Ubuntu和开发板的网络通信（192.168.0.66是Ubuntu的IP地址，2001是端口号，可以任意指定一个端口号，/drivers/gdbtest是编译并拷贝到开发板的可执行文件）。出现这样表明通信成功：



在Ubuntu中输入arm-linux-gnueabihf-gdb gdbtest启动gdb调试工具（第一次启动时报错缺少共享库，通过sudo apt install libpython2.7命令安装了此库就可以启动了），在gdb中输入target remote 192.168.0.50:2001连接到开发板（192.168.0.50是开发板的IP地址，2001是gdbserver设置的端口号），连接成功会显示（由于网络问题响应可能会比较缓慢，要耐心！！）：



此时就可以在桌面端进行调试，开发板会根据命令运行可执行文件。

* l（list）命令，列出程序源码，没有打印完可以重复l命令继续往下打印。list 函数名，将显示叫此“函数名”的函数的源代码。
* r（run）命令，开始运行直到结束或遇到断点停止。（嵌入式远程调试不能使用）
* start命令，运行程序并停在第一条执行语句上。（嵌入式远程调试不能使用）
* b（break）命令，设置断点，b 函数名或者b 行号。info b（info breakpoints）命令可以显示断点信息。
* delete命令，delete 断点号n，删除第n个断点，delete breakpoints，清除所有断点。
* clear命令，clear 行号n，清除第n行的断点
* disable命令，disable 断点号n，关闭第n个断点。
* enable命令，enable 断点号n，开启第n个断点
* c（continue）命令，继续运行直到断点停止。
* s（step）命令，单步运行，会进入函数里面。
* n（next）命令，单步运行，不会进入函数里面。
* p（print）命令，打印某个变量值。
* whatis命令，打印某个变量类型。
* set命令，设置某个变量的值，set 变量名=值。
* watch命令，设置观察某个变量，当其发生变化时会打印出其变化前后的值并且暂停程序。（嵌入式远程调试不能使用）
* until命令，运行程序直到退出循环体停止，会被断点打断。
* finish命令，运行程序直到当前函数完成返回后停止，并打印函数返回时的堆栈地址、返回值及参数值等信息，会被断点打断。
* call命令，call 函数(参数)，调用程序中可见的函数，并传递“参数”。
* bt（backtrace）命令，显示当前调用堆栈。
* x命令，显示内存值，x/<n/f/u> <addr>，n为数量、f为格式、u为单位，且可以取变量地址，如x/4xb &i（表示十六进制格式显示从变量i地址开始的4个字节的内存值，其中x表示十六进制格式显示，b表示单位为btye，&i表示取变量i的地址）。
* q（quit）命令，退出gdb调试工具，开发板的gdbserver也会停止。
* 空回车命令，表示重复上个命令。

通过给VSCode配置插件，还可以实现桌面端用图形化界面调试嵌入式Linux程序。给VSCode安装远程调试插件“Remote Development”，点击“运行”->“添加配置”，然后选择“C++(GDB/LLDB)”选项，选择“默认配置”选项，会自动生成默认的launch.json文件。launch.json文件中有几个项目是需要根据实际情况配置的：

① name：调试的项目名。

② program：指定调试的应用程序可执行文件。

③ cwd：指定调试的应用程序源码路径。

④ miDebuggerPath：此项需要手动添加，指定所使用的交叉编译器gdb工具。

⑤ miDebuggerServerAddress：此项需要手动添加，指定远程gdbserver服务器地址。

VSCode设置好后，开发板输入gdbserver 192.168.0.66:2001 /drivers/gdbtest开启监听，就可以点击VSCode的“运行”->“启动调试”来开启调试了。图形化的调试操作也就是运行/暂停、单步跳过、单步跳入、单步跳出、重新开始、停止、添加断点和监视变量等这些基本操作。（启动调试时连接非常缓慢一定要耐心！！调试过程中VSCode右下角会出现报错无法打开文件，主要是因为程序暂停在了系统文件的代码处，而VSCode无法打开此文件，直接跳出即可，并不影响调试）

**嵌入式Linux配置python3.8**

在嵌入式中配置python的目的是，让py代码文件可以在ARM架构的嵌入式中运行，为嵌入式运行机器学习类的代码实现边缘计算做好基础。

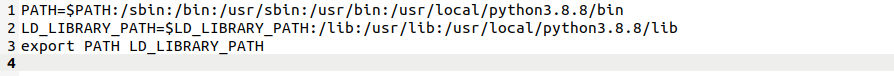
1、通过wget https://www.python.org/ftp/python/3.8.8/Python-3.8.8.tgz命令，将Python-3.8.8.tgz下载到Ubuntu主机中，tar -zxvf Python-3.8.8.tgz命令解压，并cd Python-3.8.8进入解压后的文件夹中。

2、编译python需要先得到脚本解释器，所以要先编译能够在PC主机上能够运行的脚本解释器。依次完成信息配置./configure、编译python程序和脚本解释器make python Parser/pgen、重命名python程序mv python hostpython、重命名脚本解释器mv Parser/pgen Parser/hostpgen、清理工程make distclean。（将得到的运行在PC上的python程序和脚本解释器重命名是为了后续编译不被覆盖。）

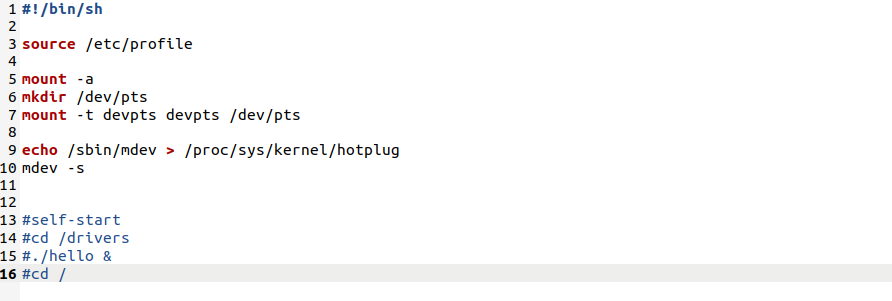
3、由于交叉编译中，需要明确如下两个变量的值，因此，这里必须人工定义。执行echo ac\_cv\_file\_\_dev\_ptmx=yes > config.site、echo ac\_cv\_file\_\_dev\_ptc=yes >> config.site、export CONFIG\_SITE=config.site这三个指令。

4、最终开始交叉编译，执行CC=arm-linux-gnueabihf-gcc CXX=arm-linux-gnueabihf-g++ AR=arm-linux-gnueabihf-ar RANLIB=arm-linux-gnueabihf-ranlib ./configure --host=arm-linux --build=x86\_64-linux-gnu --prefix=/home/ftyxt/python3.8/arm-python --disable-ipv6；接着给予setup.py文件最高权限chmod 777 ./setup.py；然后执行make HOSTPYTHON=./hostpython HOSTPGEN=./Parser/hostpgen BLDSHARED="arm-linux-gnueabihf-gcc -shared" CROSS\_COMPILE=arm-linux-gnueabihf- CROSS\_COMPILE\_TARGET=yes HOSTARCH=arm-linux BUILDARCH=x86\_64-linux-gnu；最后执行make install HOSTPYTHON=./hostpython BLDSHARED="arm-linux-gnueabihf-gcc -shared" CROSS\_COMPILE=arm-linux-gnueabihf- CROSS\_COMPILE\_TARGET=yes prefix=/home/ftyxt/python3.8/arm-python。这样之后，在PC上的/home/ftyxt/python3.8/arm-python目录下就有了编译好的运行在ARM嵌入式Linux下的可执行文件和库，将arm-python文件夹拷贝到嵌入式ARM的文件系统中。（具体位置可以自定义，我选择了/usr/local/，并重命名为python3.8.8）

5、最后在ARM嵌入式Linux中，将python添加入环境变量。打开/etc/profile（不存在就自行创建），输入



然后在/etc/init.d/rcS中添加语句source /etc/profile，如果已经存在此语句就不用添加了。



注意：以前环境变量配置是直接将这些语句写在rcS中的，但这样配置的环境变量只对自己shell和它的子shell有效，对其他shell无效，所以进入新shell后环境变量就失效了，所以只有通过source命令去运行这些语句，配置的环境变量才会对所有shell有效。

6、测试功能。在嵌入式Linux中输入python3命令，进入python编写模式，语句编写成功，导入模块成功。编写测试的test.py文件，python3 test.py也运行成功。