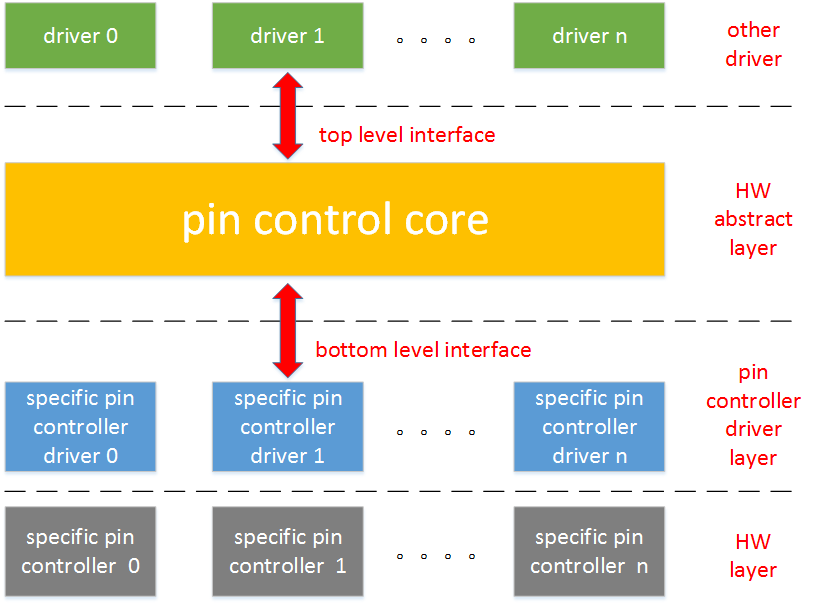
记录GPIO控制驱动的一些感想

在Linux内核驱动，GPIO控制是最简单驱动，公司内的很多产品都需要用到GPIO控制，例如LED灯的控制、继电器控制、矩阵键盘等，接下来，说说自己对江工写的GPIO控制驱动程序的一些感悟。

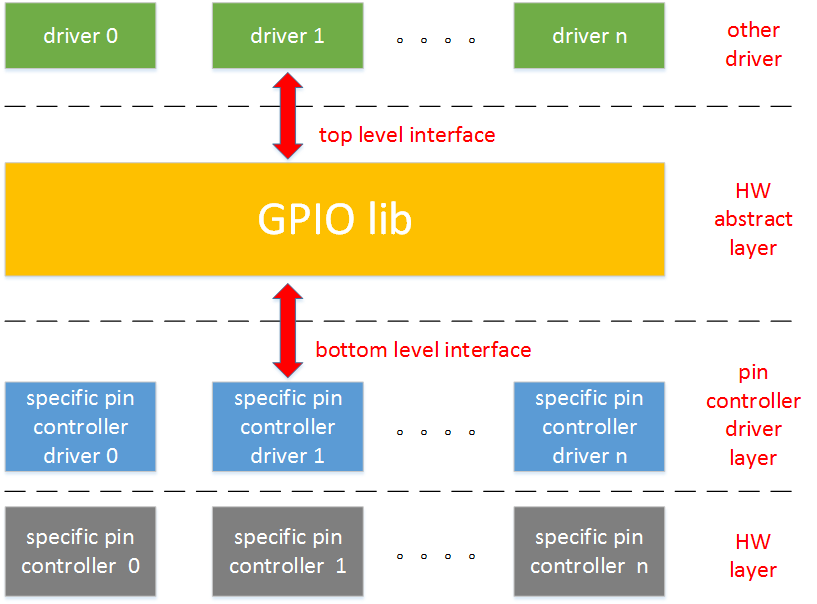
在Linux内核中有pinctrl子系统和GPIO子系统，当SoC的引脚被复用为GPIO功能（通用输入输出接口）时，我们便需要用到Linux内核中的GPIO子系统，做Linux底层驱动开发的初学者，必须要对这两个子系统具有一定的了解，包括其子系统的软件分层思想以及实现思路。

下图是内核中pinctrl子系统的软件分层架构，如下：



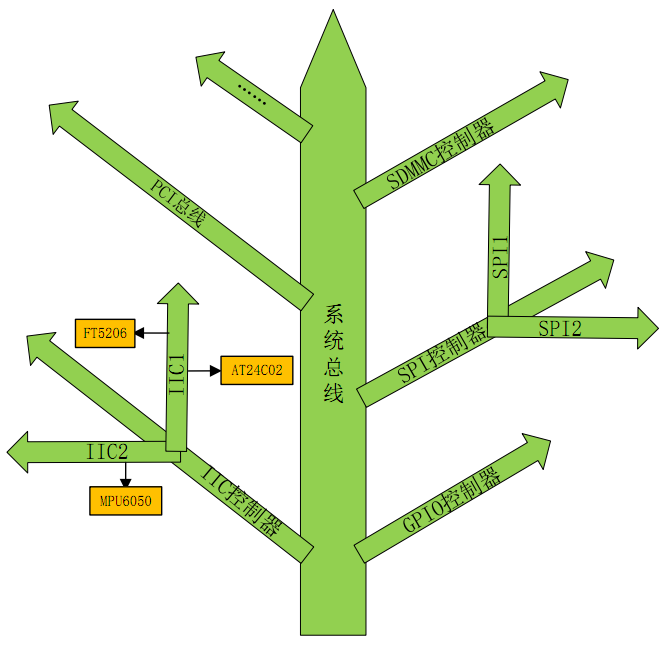
从上面这个图可以看到，Linux内核中的pinctrl子系统提供了一层硬件抽象层，它向上提供top level interface接口，该层接口给与其它驱动使用，我们作为SoC的使用者，则需要使用到这层API接口，pinctrl子系统向下提供bottom level interface接口，该接口给与SoC厂商使用，例如高通、全志这些研发芯片的公司，pin controller驱动层用于配置引脚的复用功能或者电气属性等，我们无须关注太多，作为初学者而言，更需要弄懂得是如何使用pinctrl子系统提供给我们的上层接口API。

在上面已经提到过了，当引脚复用为GPIO（通用输入输出）功能时，将会使用到Linux内核的GPIO子系统，该子系统的软件分层和pinctrl子系统类似，结构如下所示：



同样是将硬件层进行抽象了，并向上提供出顶层的API接口，这样做的好处在于，当我们要想使用GPIO子系统的功能时，例如简单IO口控制，将IO设置为输入或者输出模式，我们无须关心硬件底层的寄存器是怎么配置的，我们只需要调用GPIO子系统提供出来的API接口，便可以实现我们想要的功能，这便是其软件分层的思想所在。

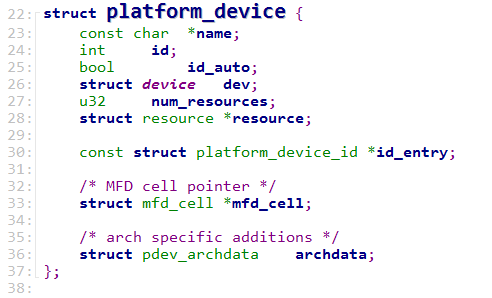
在Linux3.x版本开始，Linux内核中对于arm开发板的外设描述，开始采用设备树对板子上的外设进行描述，这是一个很大的概念，简单来说，设备树其实就是描述设备的一棵树，文件以dts或者dtsi结尾，结构类似如下图：



设备树文件中描述了我们板子的接口、设备、总线等所有东西，作为Linux驱动初学者，需要学习和掌握设备树的相关知识，学习如何在驱动程序中读取设备树的设备节点，如何使用Linux内核中提供的API接口，去获取设备树节点中的属性。

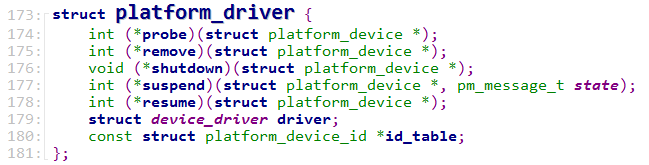
分析了Linux内核中的GPIO子系统框架图和设备树相关知识后，接下来分析一下Linux内核中提供的驱动框架，例如platform\_driver驱动框架，上面提到的GPIO控制驱动程序的实现也是基于此框架进行实现的。

先了解一下platform\_device结构体，如下：



该结构体是对平台设备的抽象，在Linux内核中，所有的设备至少有一个struct device结构体进行描述，而platform\_device则是对其的进一步封装。

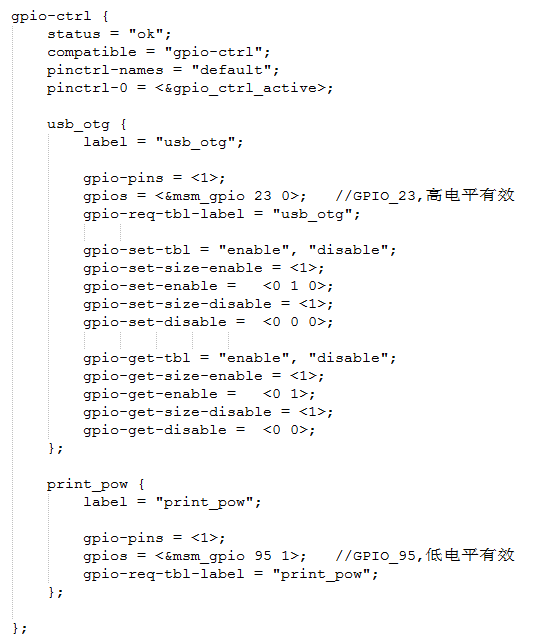
接下来就是platform\_driver结构体，作为驱动编写者，则需要关心此结构体，该结构体的描述如下所示：



Linux内核中，每一个设备驱动都有一个struct device\_driver进行描述，platform\_driver则是对其的进一步封装，id\_table这个变量非常重要，里面保存了设备和驱动进行匹配的属性值，当驱动加载后，探测到设备后，probe函数便会执行，完成设备的初始化，这部分知识点属于Linux内核中的设备驱动模型，知识的范围比较大，基本介绍到这里吧。

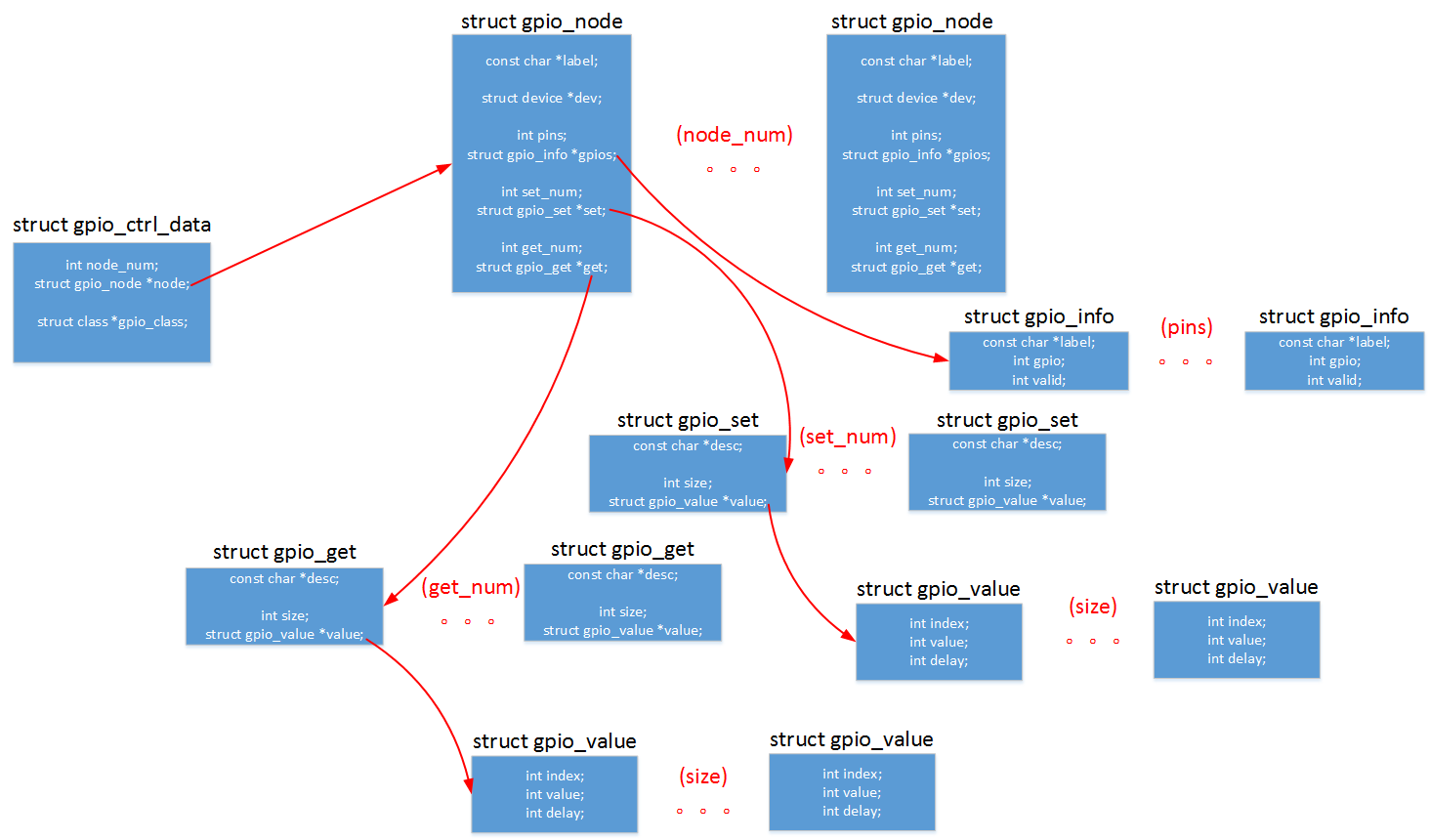
接下来分析一下，如何去编写一个GPIO控制驱动，以江工的程序为例子：

首先，需要在设备树文件中定义我们需要编写驱动的设备相关描述，例如，下面的GPIO控制设备定义：



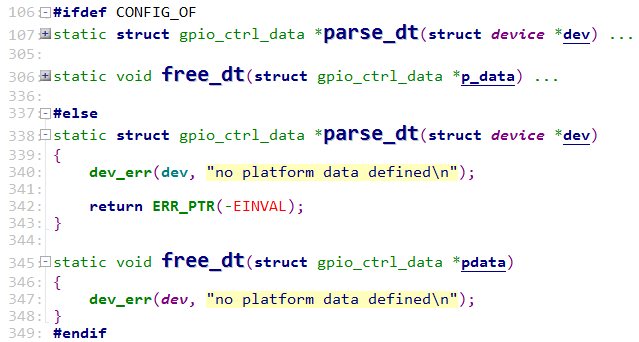
在上面的图中gpio-ctrl为一个大节点，compatible是和驱动进行匹配的属性值，usb\_otg和print\_pow是大节点的子节点，被描述为子设备，gpio-ctrl则可以看作是一个class，class底下将会有多个GPIO控制的子设备，由于是用到GPIO控制，所以必须要给出要控制的gpio号，gpios属性则是描述了要使用的gpio号。

完成了设备节点的定义后，则是要编写我们的驱动程序，使用的驱动框架为platform\_driver，先封装好我们的设备数据结构，下图则是江工为GPIO控制封装的设备数据结构，如下：



在图中。。。表示支持n个，struct gpio\_node则描述为一个GPIO控制设备节点，由此可以看出，该驱动程序能支持多个组GPIO引脚的控制，兼容性和可移植性都是比较好的，或许，驱动程序就是要实现这样的一个思想吧，更好的兼容性和可移植性，让驱动能发挥出更大的作用。

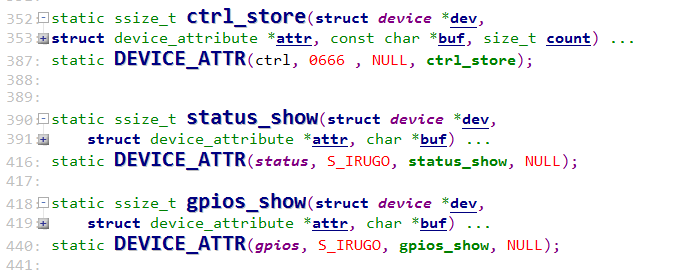
封装好设备数据结构后，因为我们的设备是以设备树节点进行描述的，因此，需要实现设备树节点解析函数，将我们定义好的设备树节点进行解析，并对我们的设备数据结构进行填充，如下所示：



函数比较长就没有直接贴了，函数parse\_dt()用于解析我们定义的设备树节点，并在内核中对我们封装的设备数据结构体进行内存分配，然后将指针进行返回。

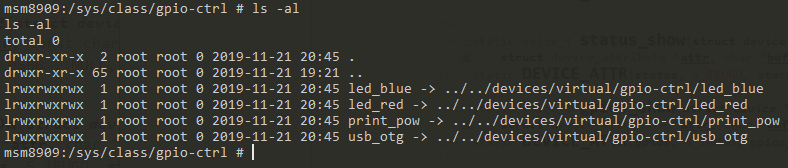
函数free\_dt()则是对解析设备树节点分配的内核内存进行释放，当驱动模块被移除时使用。

接下来就是定义我们的设备属性文件，在Linux内核中，一切皆文件，因此，对于我们的GPIO控制设备，必须要对外导出设备的属性文件，提供给应用层进行使用，使用DEVICE\_ATTR宏去对我们的设备属性文件进行定义，如下所示：

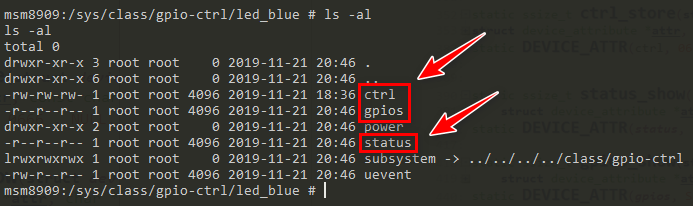


在上面的程序中，定义了3个属性文件，分别是ctrl、status和gpios属性文件，其中ctrl属性文件用来设置我们的GPIO的电平状态，status属性文件则是描述了当前的GPIO引脚电平状态，而gpios属性文件则是用来描述当前使用的gpio引脚号，当我们的驱动成功工作后，便会在相应的sysfs设备节点中生成对应的属性文件，使用echo或cat命令，便可以实现设备的控制。

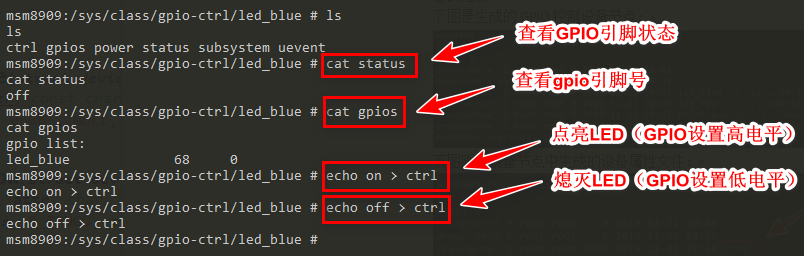
下图是生成的GPIO控制设备节点：



下图则是设备节点中生成的设备属性文件：

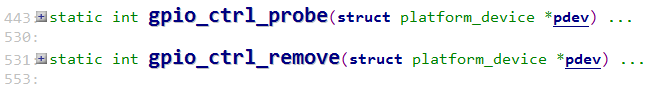


对于GPIO控制设备节点的访问，使用echo或者cat命令进行控制，如下所示：

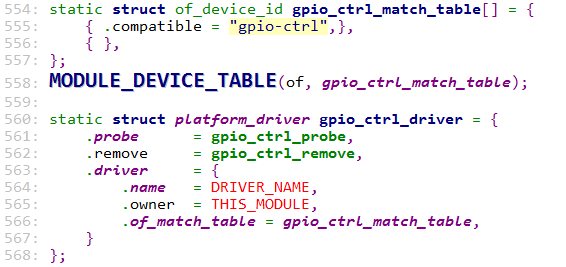


至于设备属性文件的用处便是这样，通过设备的属性文件，可以实现我们访问设备的功能。

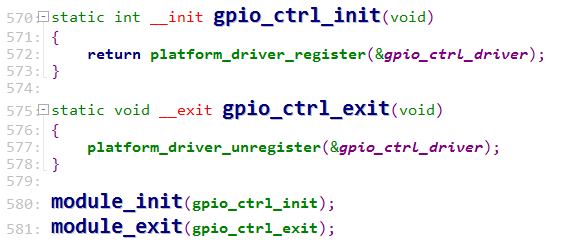
接下来，就是实现我们设备驱动的probe和remove函数，当驱动在系统中注册后，如果设备存在的话，probe函数便会执行，当设备下线后，remove函数便会执行，probe函数需要完成设备树节点的解析，GPIO资源的申请，GPIO控制设备的创建和注册，设备属性文件的生成等功能，remove函数则是逆操作，这两个函数的定义如下所示：



再接下来，则是定义of\_device\_id结构体数组，该数组元素的值用于设备节点和驱动匹配，该值，要和设备树节点中的compatible的属性值一样，例如，我们前面GPIO控制设备树节点中定义的compatible的属性值为“gpio-ctrl”，因此，在定义该结构体的时候，compatible的属性值也要定义为“gpio-ctrl”，另外，在GPIO控制驱动中，我们需要嵌入platform\_driver驱动框架，因此需要定义一个struct platform\_driver变量，并对其内部的成员进行填充，例如下图：



最后，则是模块加载函数和模块卸载函数的编写，如下，当使用insmod加载模块时，函数gpio\_ctrl\_init()函数将会执行，完成platform\_driver在系统中的注册，当使用rmmod卸载模块时，函数gpio\_ctrl\_exit()函数将会执行，完成platform\_driver系统中的注销。



以上，就是一个基本的驱动程序的编写流程，在Linux内核中的许多驱动程序都是基于这样的框架，例如I2C驱动程序、SPI驱动程序等，掌握其思路方法和框架是极为重要的，最后，附上一个驱动的大致框架图，如下所示：

