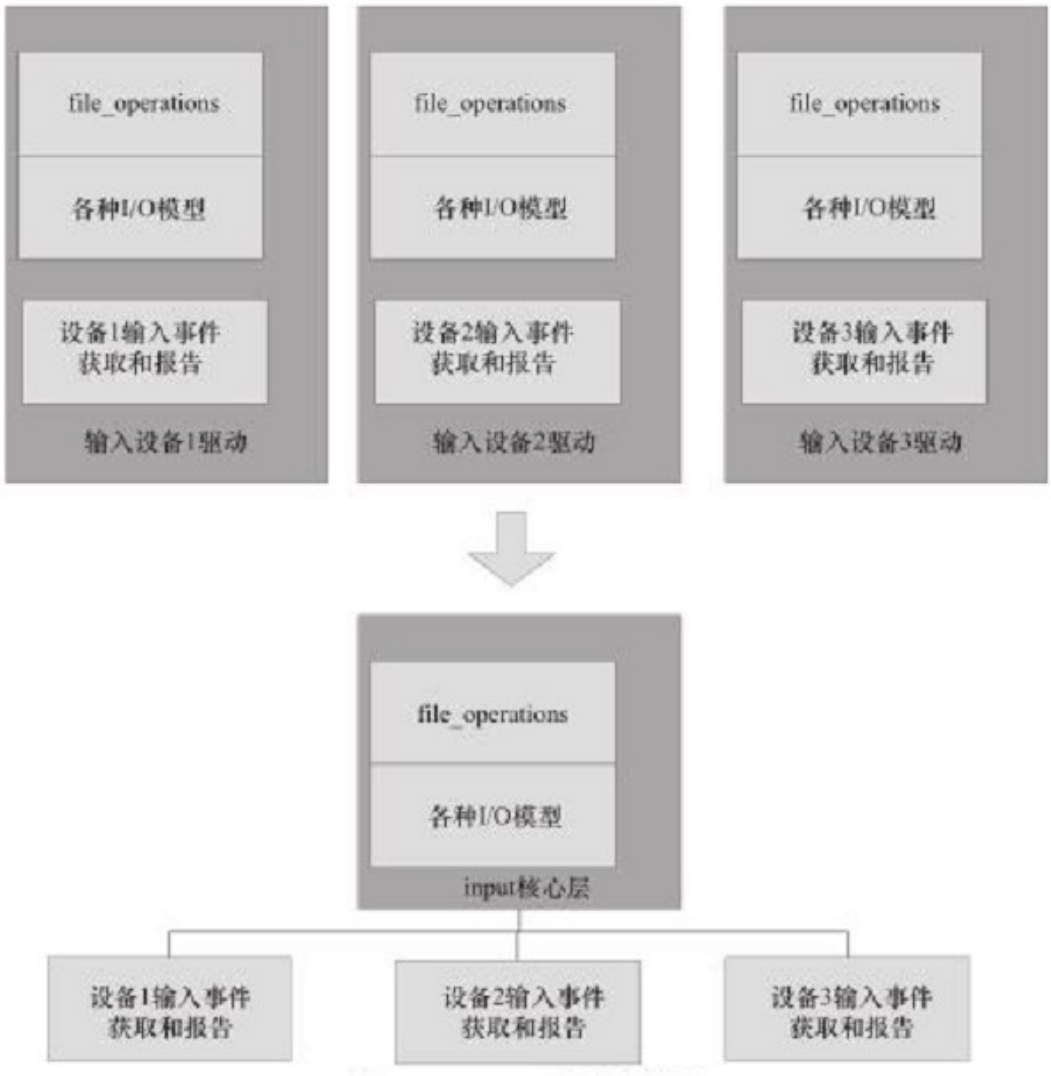
linux 设备驱动的软件架构上，实现了三个层次的分离。
按照由上而下，由粗到细的顺序：
**1.上层应用和驱动层的分离**
**2.控制器驱动和外设驱动的分离**
**3.驱动（包括控制器驱动和外设驱动）和设备的分离**

下面由最上层，逐级往下看，分析这个软件架构的组成：

**1.上层应用和驱动层的分离**
从最上层看，可以看到，从用户空间去访问外设，一般来说是通过字符/块设备/socket 来完成
以读取键盘输入为例，用户空间通过操作/dev 下的字符设备，到达内核的 input 子系统，再由 input 子系统进一步到达驱动本身。




**2.控制器驱动和外设驱动的分离**
从硬件上说，外设能挂载到 soc 上工作，是因为 soc 上有针对外设的控制器。
linux 的设计思想是把针对外设的驱动，和针对控制器的驱动，分离开来。
这两者之间，用内核的子系统分隔开来。（例如 I2C 子系统，SPI 子系统）

这样做的好处，是控制器的驱动的编写，无需关心外设，只管向内核子系统注册（例如 i2c_add_numbered_adapter()，spi_register_master()）。
而外设的驱动，无需关心控制器的驱动的细节，直接使用内核子系统的 API 操作外设（例如 spi_write()，i2c_smbus_write_byte()）。
针对一个控制器，可以去连接/删除不同数量/型号的外设，而不用去改到控制器的驱动代码。
这部分描述的内容相当于“上层应用和驱动层的分离.png”图中的《输入设备 1 的驱动》这个图框。把这个图框放放大，就可以看到：

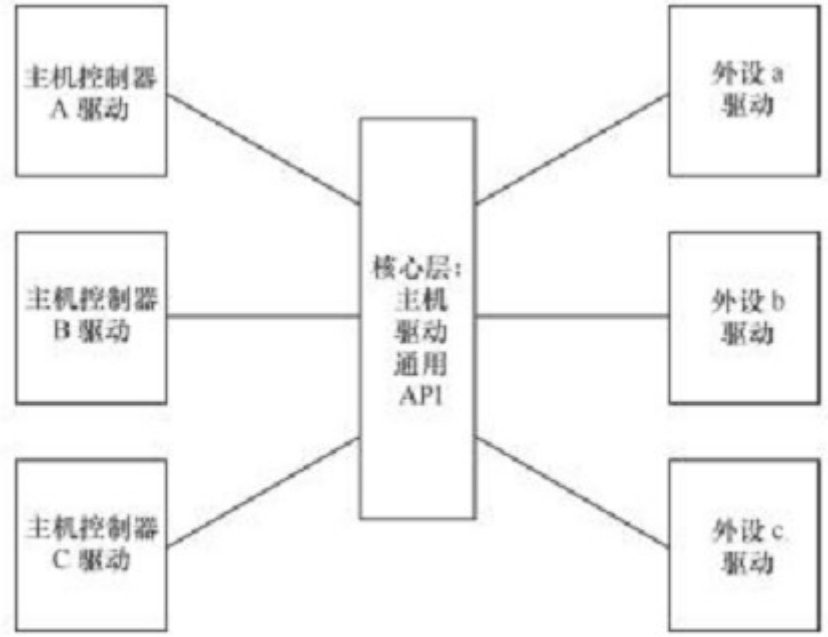
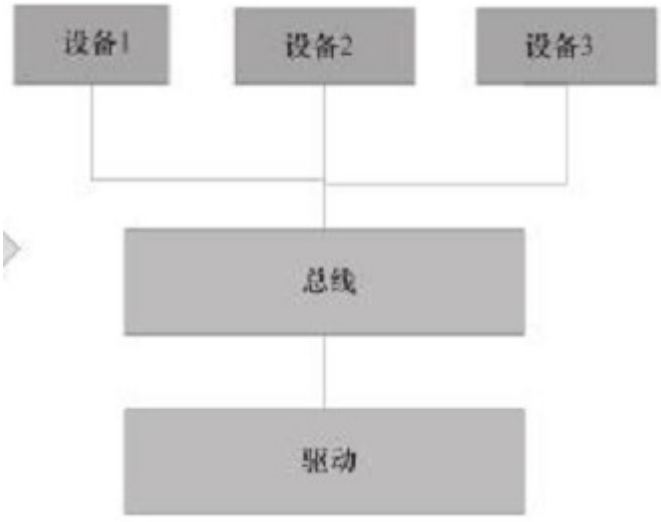


图 12.4　Linux 设备驱动的主机、外设驱动分离

**3，驱动（包括控制器驱动和外设驱动）和设备的分离**
这部分描述的内容相当于“控制器驱动和外设驱动的分离.png”” 图中的《外设 A 的驱动》这个图框，或者《主机控制器 A 驱动》。把这个图框放放大，就可以看到：



这部分分离，主要分为两块来讲。一个是针对外设的，一个是针对 soc 的控制器的。
以一个 i2c 外设驱动为例，其实内核内部，还会进一步把这个驱动，分为三部分：
i2c_device 即设备
i2c_bus 即总线
i2c_driver 即驱动
device 和 driver，有一对一的匹配的关系。而这个匹配的过程，其实就是由 i2c_bus 这一层来完成 match 的工作（通常是依赖 device 和 driver 拥有一致的 id 来匹配）。匹配成功，则执行 driver.probe 的初始化函数。
在一个 i2c_bus 上，可以挂载多对 i2c_device 和 i2c_driver，反应出物理上实际的电路结构。
当然这里重点还是 driver 部分。至于 device 部分，3.0 内核以前，在板级初始化代码里，是静态声明结构体，3.0 内核以后，则化身为 dts 里面的结点。
也可以参考“Kernel\驱动\I2C\i2c 驱动层次图.png"
上面是外设驱动的例子。
那如果是控制器的驱动呢？
控制器的驱动其实也是按照这个思路细分的。
唯一的区别是控制器的驱动，如果说到总线的话，其实控制器基本上是挂载在 soc 内部的数据总线上的。
内核针对这种情况，做了一个抽象，就是凡是属于 soc 上的控制器，一律归类为挂载在 platform_bus_type 这种总线上
而对应的驱动和设备，统称为 platform_device 和 platform_driver。
即：
platform_device 即设备
platform_bus 即总线
platform_driver 即驱动
和外设的情况类似，platform 的 device 和 driver，都是通过 platform_bus_type 的 platform_match 函数进行匹配的。匹配成功，则执行 driver.probe 的初始化函数。