嵌入式 Linux 驱动开发的方方面面，学习了！

嵌入式大杂烩 2023-02-09 21:30 发表于广东

01
嵌入式驱动开发到底学什么

嵌入式大体分为以下四个方向：

一、嵌入式硬件开发：熟悉电路等知识，非常熟悉各种常用元器件，掌握模拟电路和数字电路设计的开发能力。熟练掌握嵌入式硬件知识，熟悉硬件开发模式和设计模式，熟悉 ARM32位处理器嵌入式硬件平台开发、并具备产品开发经验。精通常用的硬件设计工具：Protel/PADS(PowerPCB)/Cadence/OrCad。一般需要有4~8层高速PCB设计经验。

二、嵌入式驱动开发：熟练掌握 Linux 操作系统、系统结构、计算机组成原理、数据结构相关知识。熟悉嵌入式 ARM 开发，至少掌握 Linux 字符驱动程序开发。具有单片机、ARM嵌入式处理器的移植开发能力，理解硬件原理图，能独立完成相关硬件驱动调试，具有扎实的硬件知识，能够根据芯片手册编写软件驱动程序。

三、嵌入式系统开发：掌握Linux系统配置，精通处理器体系结构、编程环境、指令集、寻址方式、调试、汇编和混合编程等方面的内容;掌握Linux文件系统制作，熟悉各种文件系统格式(YAFFS2、JAFFS2、RAMDISK等);熟悉嵌入式Linux启动流程，熟悉 Linux 配置文件的修改；掌握内核裁减、内核移植、交叉编译、内核调试、启动程序Bootloader编写、根文件系统制作和集成部署 Linux 系统等整个流程；熟悉搭建 Linux 软件开发环境(库文件的交叉编译及环境配置等);

四、嵌入式软件开发：精通Linux操作系统的概念和安装方法、Linux下的基本命令、管理配置和编辑器，包括 vi 编辑器，gcc 编译器，GDB 调试器和 Make 项目管理工具等知识；精通C语言的高级编程知识，包括函数与程序结构、指针、数组、常用算法、库函数的使用等知识、数据结构的基础内容，包括链表、队列等；掌握面向对象编程的基本思想，以及C++语言的基础内容；精通嵌入式Linux下的程序设计，精通嵌入式Linux开发环境，包括系统编程、文件I/O、多进程和多线程、网络编程、GUI图形界面编程、数据库；熟悉常用的图形库的编程，如QT、GTK、miniGUI、fltk、nano-x等。

公司的日常活动还是看公司的规模，大一点的一般只是让你负责一个模块，这样你就要精通一点。若是公司比较小的话估计要你什么都做一点。还要了解点硬件的东西。

那么看了这么多，嵌入式和纯软最大的区别在于：

纯软学习的是一门语言，例如C,C++,java,甚至Python，语言说到底只是一门工具，就像学会英语法语日语一样。

但嵌入式学习的是软件+硬件，通俗的讲，它学的是做系统做产品，讲究的是除了具体的语言工具，更多的是如何将一个产品分解为具体可实施的软件和硬件，以及更小的单元。

不少人问，将来就业到底是选驱动还是选应用？只能说凭兴趣，并且驱动和应用并不是截然分开的。

PART 01

我们说的驱动，其实并不局限于硬件的操作，还有操作系统的原理、进程的休眠唤醒调度等概念。想写出一个好的应用，想比较好的解决应用碰到的问题，这些知识大家应该都懂。

PART 02

做应用的发展路径个人认为就是业务纯熟。比如在通信行业、IPTV行业、手机行业，行业需求很了解。

PART 03

做驱动，其实不能称为“做驱动”，而是可以称为“做底层系统”，做好了这是通杀各行业。比如一个人工作几年，做过手机、IPTV、会议电视，但是这些产品对他毫无差别，因为他只做底层。当应用出现问题，解决不了时，他就可以从内核角度给他们出主意，提供工具。做底层的发展方向，应该是技术专家。

PART 04

其实，做底层还是做应用，之间并没有一个界线，有底层经验，再去做应用，会感觉很踏实。有了业务经验，再了解一下底层，很快就可以组成一个团队。

02
嵌入式Linux底层系统包含哪些东西?

嵌入式 Linux 里含有 bootloader, 内核, 驱动程序、根文件系统这4大块。

一、bootloader

它就是一个稍微复杂的裸板程序。但是要把这裸板程序看懂写好一点都不容易。Windows下好用的工具弱化了我们的编程能力。很多人一玩嵌入式就用ADS、KEIL。能回答这几个问题吗？

Q:一上电，CPU从哪里取指令执行？
A:一般从Flash上指令。

Q:但是Flash一般是只能读不能直接写的，如果用到全局变量，这些全局变量在哪里？
A:全局变量应该在内存里。

Q:那么谁把全局变量放到内存里去？
A:长期用ADS、KEIL的朋友，你能回答吗？这需要"重定位"。在ADS或KEIL里，重定位的代码是制作这些工具的公司帮你写好了。你可曾去阅读过？

Q:内存那么大，我怎么知道把"原来存在Flash上的内容"读到内存的"哪个地址去"？
A:这个地址用"链接脚本"决定，在ADS里有scatter文件，KEIL里也有类似的文件。但是，你去研究过吗？

Q:你说重定位是把程序从Flash复制到内存，那么这个程序可以读Flash啊？
A:是的，要能操作Flash。当然不仅仅是这些，还有设置时钟让系统运行得更快等等。

先自问自答到这里吧，对于 bootloader 这一个裸板程序，其实有3部分要点：

①对硬件的操作

对硬件的操作，需要看原理图、芯片手册。这需要一定的硬件知识，不要求能设计硬件，但是至少能看懂；不求能看懂模拟电路，但是要能看懂数字电路。这方面的能力在学校里都可以学到，微机原理、数字电路这2本书就足够了。想速成的话，就先放掉这块吧，不懂就GOOGLE、发贴。另外，芯片手册是肯定要读的，别去找中文的，就看英文的。开始是非常痛苦，以后就会发现那些语法、词汇一旦熟悉后，读任何芯片手册都很容易。

②对ARM体系处理器的了解

对ARM体系处理器的了解,可以看杜春蕾的<ARM体系架构与编程>，里面讲有汇编指令，有异常模式、MMU等。也就这3块内容需要了解。

③程序的基本概念：重定位、栈、代码段数据段BSS段等

程序的基本概念，王道当然是去看编译原理了。可惜，这类书绝对是天书级别的。若非超级天才还是别去看了。可以看韦东山的<嵌入式Linux应用开发完全手册>。

对于bootloader，可以先看<ARM体系架构与编程>，然后自己写程序把各个硬件的实验都做一遍，比如GPIO、时钟、SDRAM、UART、NAND。把它们都弄清楚了，组台在一起就很容易看懂u-boot了 。

总结一下，看懂硬件原理图、看芯片手册，这都需要自己去找资料。

二、内核

想速成的人，先跨过内核的学习，直接学习怎么写驱动。
想成为高手，内核必须深刻了解。注意，是了解，要对里面的调度机制、内存管理机制、文件管理机制等等有所了解。

推荐两本书：

1. 通读<linux内核完全注释>,请看薄的那本

2. 选读<Linux内核情景分析>，想了解哪一块就读哪一节

三、驱动

驱动包含两部分：硬件本身的操作、驱动程序的框架。
又是硬件，还是要看得懂原理图、读得懂芯片手册，多练吧

①硬件本身的操作

说到驱动框架，有一些书介绍一下。LDD3，即<Linux设备驱动>，老外写的那本，里面介绍了不少概念，值得一读。但是，它的作用 也就限于介绍概念了。入门之前可以用它来熟悉一下概念。

②驱动程序的框架

驱动方面比较全的介绍，应该是宋宝华的<linux设备驱动开发详解>了。要想深入了解某一块，<Linux内核情景分析>绝对是超5星级推荐。别指望把它读完，1800多页，上下两册呢。某一块不清楚时，就去翻一下它。任何一部分，这书都可以讲上2、3百页，非常详细。并且是以某个目标来带你分析内核源码。它以linux2.4为例，但是原理相通，同样适用于其它版本的linux。

把手上的开发板所涉及的硬件，都去尝试写一个驱动吧。有问题就先"痛苦地思考"，思考的过程中会把很多不相关的知识串联起来，最终贯通。

四、根文件系统

大家有没有想过这2个问题：

Q:对于Linux做出来的产品，有些用作监控、有些做手机、有些做平板。那么内核启动后，挂载根文件系统后，应该启动哪一个应用程序呢？

A:内核不知道也不管应该启动哪一个用户程序。它只启动 init 这一个应用程序，它对应/sbin/init。

显然，这个应用程序就要读取配置文件，根据配置文件去启动用户程序(监控、手册界面、平板界面等等，这个问题提示我们，文件系统的内容是有一些约定的，比如要有 /sbin/init，要有配置文件 。

Q:你写的hello,world程序，有没有想过里面用到的printf是谁实现的？

A:这个函数不是你实现的，是库函数实现的。它运行时，得找到库。

这个问题提示我们，文件系统里还要有库。

简单的自问自答到这里，要想深入了解，可以看一下 busybox 的 init.c，就可以知道 init 进程做的事情了。

当然，也可以看<嵌入式Linux应用开发完全手册>里构建根文件系统那章。

03
驱动程序设计的5个方法

1.使用设计模式

设计模式是一个用来处理那些在软件中会重复出现的问题的解决方案。开发人员可以选择浪费宝贵的时间和预算从无到有地重新发明一个解决方案，也可以从他的解决方案工具箱中选择一个最适合解决这个问题的方案。在微处理器出现之初，底层驱动已经很成熟了，那么，为什么不利用现有的成熟的解决方案呢?

驱动程序设计模式大致分属以下4个类别：Bit bang、轮询、中断驱动和直接存储器访问(DMA)。

Bit bang模式：当微控制器没有内外设去执行功能的时候，或者当所有的内外设都已经被使用了，而此时又有一个新的请求，那么开发者就应该选择Bit bang设计模式。Bit bang模式的解决方案很有效率，但通常需要大量的软件开销来确保其实施的能力。Bit bang模式可以让开发者手动完成通信协议或外部行为。

轮询模式用于简单地监视一个轮询调度方式中的事件。轮询模式适用于非常简单的系统，但许多现代应用程序都需要中断。

中断可以让开发者在事件发生时进行处理，而不用等代码手动检查。

DMA(直接存储器访问)模式允许其它外围设备来处理数据传输的需求，而不需要驱动的干预。

2.了解实时行为

一个实时系统是否能满足实时需求取决于它的驱动程序。写入能力差的驱动是低效的，并可能使不知情的开发者放弃系统的性能。设计者需要考虑驱动的两个特点：阻塞和非阻塞。一个阻

塞的驱动程序在其完成工作之前会阻止其他任何软件执行操作。例如，一个USART驱动程序可以把一个字符装入传输缓冲区，然后一直等到接收到传输结束标志符才继续执行下一步操作。

另一方面，非阻塞驱动则是一般利用中断来实现它的功能。中断的使用可以防止驱动程序在等待一个事件发生时拦截其他软件的执行操作。USART 的驱动程序可以将一个字符装入传输缓冲区然后等主程序发布下一个指令。传输结束标志符的设置会导致中断结束，让驱动进行下一步操作。

无论哪种类型，为了保持实时性能，并防止系统中的故障，开发人员必须了解驱动的平均执行时间和最坏情况下的执行时间。一个完整的系统可能会因为一个潜在的风险而造成更大的安全问题。

3. 重用设计

在时间和预算都很紧张的情况下为什么还要再造轮子呢?在驱动程序开发中，重用、便携性和可维护性都是驱动设计的关键要求。这里面的许多特征可以通过硬件抽象层的设计和使用来说明。

硬件抽象层(HAL)为开发人员提供一种方式来创建一个标准接口去控制微控制器的外设。抽象隐藏实现细节，取而代之的是提供了可视化功能，如 Usart_Init和 Usart_Transmit。这个方法就是让任何 USART、SPI、PWM 或其他外设具备所有微控制器都支持的共同特点。使用 HAL 隐藏底层、特定设备的细节，让应用程序开发人员专注于应用的需求，而不是关注底层的硬件是如何工作的。同时 HAL 提供了一个重用的容器。

4.参考数据手册

微控制器在过去的几年里变得越来越复杂。以前想要完全了解一个微控制器需要掌握由一个大约包含500页组成的单一数据手册。而如今，一个32位微控制器通常包含由部分的数据手册、整个微控制器系列的资料表、每个外设数以百计的资料以及所有的勘误表组成的数据手册。开发人员如果想要完全掌握这部分的内容需要了解几千页的文件。

不幸的是，所有这些数据手册都是一个驱动程序能真正合理实现所需要的。开发人员在一开始就要对每个数据手册中包含的信息进行收集和排序。通常它们中的每一个都需要被访问以使外设启动和运行。关键信息被分散(或隐藏)在每种类型的数据手册中。

5.谨防外设故障

最近我刚好有机会把一系列的微控制器驱动移植到其他的微处理器上。制造商和数据手册都表明PWM外设在这两个系列的微控制器之间是相同的。然而，实际情况却是在运行PWM驱动器的时候两者之间有很大的不同。该驱动程序只能在原来的微控制器工作，而在新系列的微控制器上却无效。

在反复翻看数据手册之后，我在数据手册中一个完全不相关的注脚里发现了PWM外设上电时会处于故障状态，需要将一个隐藏在寄存器中的标志位清零。在驱动程序实现的开始，确认外设可能出现的故障并查看其他看似无关的寄存器错误。

04
大牛对于嵌入式驱动开发的建议

1) 为了今后的发展，你除了考虑广度以外，更重要的是注意知识的深度。

譬如，做过网络驱动，那么是不是只停留在会写驱动的表层上，有没有对Linux 内核的网络结构，TCP/IP 协议作过深入的了解。

2） 在Linux下开发很多时候都要利用现成的东西，没必要什么都自己搞。关键是变成自己的驱动后是否了解原作者编写时背后的一些东西。你应该不只是简单的让它工作。写驱动的时候就要考虑它的性能问题，并给出测试的方法（当然可以利用现成的许多工具，譬如测试网络性能的netperf等）。

当你写过 Flash 驱动，可能会知道 Flash 的性能有时候有多重要。

3） C程序的自我修炼，是否考虑到软件工程方面的一些东西，程序的可维护性和扩展性，譬如LCD驱动，是不是从Sharp到NEC的只需要集中修改很少的几个地方？

对于不同品牌的Flash，如果使得Flash的驱动做的更具有灵活性。

4） 如果有时间结余，可以关注Linux内核的发展。譬如LCD的驱动有没有考虑到V4L2通用架构，譬如网络驱动用到了NAPI了吗？当然在此之前，假设已经对LDD3， ULK2理解的比较熟了。

5） 现在所作的这些驱动还算不得非常核心的东西。如果你想有更好的发展，可以考虑往audio，video，net方面发展，你应该多注意真个行业需要什么样的人才，上述每一项都需要很厚的底蕴，譬如video，需要了解MPEG4， H264等，怎么也要个1到2年才能算个入行阿，所以我建议不要只顾闷头做东西，要适当关注目前的一些应用。

6） 对硬件知识的补给，做嵌入式Linux这一行不可能不读硬件的Spec，如果你对硬件的工作机制理解的比较透，会有助你写出性能好的驱动程序。

顺便提一点，适时的提高你的英语水平，对你的职业生涯绝对有帮助。（不要等需要的时候再补，来不及）

7） 如果有时间，平时注意对Linux应用程序编写的了解/积累，也将有助于你写出很好功能很好的驱动程序。

8） 永远不能以为自己做了很多东西，就驱动而言，像TVIN/TVOUT， USB， SDIO等等，好多未知领域呢。在问题还没有解决之前很难说清是哪里不对了。

有时候是 datasheet 里面的一句话没有注意，还有好几次调不出来最后查到是 PCB 的问题，所以有时候特别晕。

05
嵌入式驱动自学者的感受

经过了多年的嵌入式自学，可谓是不断在绝望中求生。性格使然，我是一个我也不知这种性格的学名叫什么，就是学习一种东西，非得想要能理解每一处的含义作用为什么，要这样做没有其他办法了吗等等问题。并且当一个问题找不到让我能接受的解释时，那么我的学习路程也就几乎要停在这里了，大概是因为我讨厌一知半解。

可能是小时候被老师教导不要做书呆子的教育有关，小时候，听话孩子，认真，长辈的教育对孩子的影响真的是非常的大，很多影响如果你不细心的观察自己，你根本不能察觉这些进入了你骨子的观念，在我成长过程中，这些长辈的教育除了某些让我自己经历到并彻底认识到某个观念并不正确时，我才会形成自己的观点，自己的观念，但这些自己的观念在所有的价值观中，犹如沧海一粟。

这种讨厌一知半解的性格，在现在这个社会来说，可以说是极端的，因为现在你学习使用的很多东西，他都不是从零开始的，就好比，你编程使用的是高级语言而不是低级语言不是机器码，所以我的整个学习过程是非常缓慢缓慢地进行着，这么说吧，前面说我经过了半年多的学习，但是到现在为止，我接触嵌入式已经有两个年头了，也就是说，学习期间，我有一年多是在停滞着。

学习嵌入式，或者说学习现代的计算机编程，如果你想学好，有一个比较要求，那就是你能接受它的设定、它的模式。反过来说，当你真正接受它的设定、它的模式，并记住它们时，我认为，你已经学好了。

昨天，我又置之死地而后生了一次。最近一直在搞驱动，一个LCD驱动搞得我几乎要放弃继续走嵌入式这条路。昨夜，睡不觉，打开嵌入学习视频，躺在已关灯很久的房间的床上，大概凌晨3，4点吧。之前我一直都是学习着驱动自编源码的教学，是那种几乎和裸板程序没多大区别的编程方式，只是多使用了一些向内核注册的接口函数。

而最近我想换一下，因为很多设备驱动，内核都是自带的，而且是各种平台的设备驱动都有，我想如果能熟悉掌握内核自带驱动的编程，那以后要做某个设备的驱动时，我只需要在自带驱动中修改一下便好了，通过学习LCD平台设备的驱动，我了解了其编程想法，同时也认同这种想法，甚至让我疑惑，学习资料中教自编驱动的意义，为何不直接教如果修改内核源码驱动？

于是，继续按着书去修改内核驱动源码，但问题是，书中说他们这种修改，代码成功运行了，但我这，无论怎么调试都失败，我反复检测，我的修改是否与书中一致，检测了很多遍依然没发现哪一步不同，不过，有一点发现是，书中的内核源码和我内核使用的源码有一点点区别（当然书里并没有把所有的源码都贴上，只是修改部分附近会联带着一些，这就是发现，这些联带的没需要修改的源码和我的源码有点区别，比如，我的源码中多了一些设置（看似无关紧要的设置））。

与书核对无误但失败后，我又与成功运行的自编驱动核对，我陆续发现我修改的内涵源码中，没有去启动设备，也更没有去点亮背光，而在显存分配后的寄存器设置似乎也有问题，因为这里的地址使用各种宏定义不同的累加或计算，最后算得地址和我的寄存器地址也不知是否吻合，因为驱动源码中最后计算得到的是虚拟地址。于是我对比自编驱动，一点点修改尝试，到睡觉前都没成功。

我是想学得理直气壮一点的，最后是能一眼就能找到问题，并迅速轻松解决问题的，我也承认自己确实是有些浮躁。但是经过了昨晚床上的一点绝望的思考挣扎后，我好像想通了：为什么

嵌入式学习视频老师要教自编驱动。

下面我说下自编驱动与内核驱动源码各自的问题：

自编驱动：

程序简单简洁，它只能驱动特定的某个设备。如果设备换了需要支持另一款设备，那么你需要重新修改该驱动；如果需要系统同时支持两种LCD，那么它就会变成复杂并且对于内核驱动的简洁优势会削弱不少；如果你想驱动支持多种设备，那自编驱动，相对于内核驱动源码的简洁优势会变成了劣势，因为编程思想的适用范围不同而产生的结果。

内核自带驱动源码：

①从系统层次去考量，变量、宏定义使用多，甚至有些宏定义的值为了方便能让各种在不同的阶段需要不同的值调用，把简单的一个赋值调用变成了需要进行多次运算才能检测到该值是否满足使用要求，因为我们不是该驱动的编码者，不清楚这样做的好处，也或许是内核驱动源码的开发者从整个系统的编程简洁性去考量，这样做或许也是为了让整个系统代码更少，简洁的一种做法，因为每个设备你都给它赋具体的值的话，整个系统中有几百种驱动设备源码，给所有设备的这个位置参数都赋一个值的话，那各设备关于这个值的代码就要多了几百行了，所以还不如，让各设备根据各种平台去对某个宏进行各自的计算来得到合适的值，但某些计算中相同的算法的也整合在一起，这样就减少了系统不少行代码。所以系统中驱动源码是系统开发者对系统源码的整合，是基于系统层的整合。因此，对于我这种对单个设备驱动编码的人，就会觉得系统源码有好多不人性化的地方，会觉得简单的地方也被弄得很复杂。

②内核自带驱动还有一些代码是为了兼容以前的版本而添加了，比如以前硬件内存资源稀少，需要使用调色板的方法来减少程序运行时的内存使用量，这也会真假代码的复杂性，这一步虽不是必要的，但如果没弄好，那LCD驱动也不能正常使用。

③程序复杂，为了适用在多种设备型号，更简单地添加不同型号的设备驱动，内核对驱动抽象分离，把驱动分为平台管理部分，驱动代码部分（与硬件无关码），和设备代码部分（硬件相关代码）。用户添加新型号设备驱动时，只需要在平台管理部分检查添加设备的匹配信息，和提供一个硬件设备相关的代码（有格式）文件即可。

现在，站在驱动开发者而非系统开发者的角度去衡量。

①自编的驱动，简洁，要点明确。这个对于驱动开发者的用处就是：无论你使用的是哪个版本的内核，哪个芯片平台，你可以通过自编码比较简单方便地就可以确认硬件设备的情况，是否正常。如果自编码通过，那可以试用自编码上使用的参数去与内核进行核对、修改，然后再去测试。如果不成功，对于内核中多余的设置（这些大多可能是提供内核用做基本判断的变量）可以先屏蔽，编译出错了，根据提示，找到出错的位置修改添加。因为这些多余的设置，设置对了还行，设置错了，你又不好去定位错在哪。

自编的驱动在此处的用处，调试时，可以让你排除多余的失败可能性问题，在较少的代码去查出错误位置，如果你确定你的设置满足了该设备的必需设置，还是失败，你可以比较放心地去怀疑是硬件问题了。如果自编码成功，那个又可以当做你修改内核驱动的一个标准。

②内核驱动源码支持管理多种型号的设备的优势，是我用使用它的原因。先了解本版本平台的设备驱动结构，如果是添加型号支持，那就根据自编驱动的参数与设置即可，如果是第一次启动这类设备，你就还需要检测结构是完整性，如果结构完整，参数无误依旧错误，那就把内核驱动源码精简到自编码的简单粗暴设置吧。最终就变成了在基于内核驱动架构下的自编驱动。如果还不行，那无疑是结构性问题了。

所以自编驱动，还是有其存在价值的。内核驱动源码内容会变，平台会变，但自编驱动是变得最小的一个，也是最容易实现驱动目的的一个。是一码打天下不可缺少的重要组成部分。

06

如何编写嵌入式Linux设备驱动程序?

一、Linux device driver 的概念

系统调用是操作系统内核和应用程序之间的接口，设备驱动程序是操作系统内核和机器硬件之间的接口。设备驱动程序为应用程序屏蔽了硬件的细节，这样在应用程序看来，硬件设备只是一个设备文件，应用程序可以象操作普通文件一样对硬件设备进行操作。设备驱动程序是内核的一部分，它完成以下的功能:

1、对设备初始化和释放；

2、把数据从内核传送到硬件和从硬件读取数据；

3、读取应用程序传送给设备文件的数据和回送应用程序请求的数据；

4、检测和处理设备出现的错误。

在linux操作系统下有三类主要的设备文件类型，一是字符设备，二是块设备，三是网络设备。字符设备和块设备的主要区别是:在对字符设备发出读/写请求时，实际的硬件I/O一般就紧接着发生了，块设备则不然，它利用一块系统内存作缓冲区，当用户进程对设备请求能满足用户的要求，就返回请求的数据，如果不能，就调用请求函数来进行实际的I/O操作。块设备是主要针对磁盘等慢速设备设计的，以免耗费过多的CPU时间来等待。

已经提到，用户进程是通过设备文件来与实际的硬件打交道。每个设备文件都都有其文件属性(c/b)，表示是字符设备还是块设备?另外每个文件都有两个设备号，第一个是主设备号，标识驱动程序，第二个是从设备号，标识使用同一个设备驱动程序的不同的硬件设备，比如有两个软盘，就可以用从设备号来区分他们。设备文件的的主设备号必须与设备驱动程序在登记时申请的主设备号一致，否则用户进程将无法访问到驱动程序。

最后必须提到的是，在用户进程调用驱动程序时，系统进入核心态，这时不再是抢先式调度。也就是说，系统必须在你的驱动程序的子函数返回后才能进行其他的工作。如果你的驱动程序陷入死循环，不幸的是你只有重新启动机器了，然后就是漫长的fsck。

二、实例剖析

我们来写一个最简单的字符设备驱动程序。虽然它什么也不做，但是通过它可以了解Linux的设备驱动程序的工作原理。把下面的C代码输入机器，你就会获得一个真正的设备驱动程序。

由于用户进程是通过设备文件同硬件打交道，对设备文件的操作方式不外乎就是一些系统调用，如 open，read，write，close...， 注意，不是fopen， fread，但是如何把系统调用和驱动程序关联起来呢?这需要了解一个非常关键的数据结构：

```
struct file_operations {
int (*seek) (struct inode * , struct file *,  off_t , int);
int (*read) (struct inode * , struct file *,  char , int);
int (*write) (struct inode * , struct file *,  off_t , int);
int (*readdir) (struct inode * , struct file *,  struct dirent * , i
int (*select) (struct inode * , struct file *,  int , select_table *
int (*ioctl) (struct inode * , struct file *,  unsined int , unsigne
int (*mmap) (struct inode * , struct file *,  struct vm_area_struct
int (*open) (struct inode * , struct file *);
int (*release) (struct inode * , struct file *);
int (*fsync) (struct inode * , struct file *);
int (*fasync) (struct inode * , struct file *, int);
int (*check_media_change) (struct inode * , struct file *);
int (*revalidate) (dev_t dev);

```

```
}
```

这个结构的每一个成员的名字都对应着一个系统调用。用户进程利用系统调用在对设备文件进行诸如 read/write 操作时，系统调用通过设备文件的主设备号找到相应的设备驱动程序，然后读取这个数据结构相应的函数指针，接着把控制权交给该函数。这是 linux 的设备驱动程序工作的基本原理。既然是这样，则编写设备驱动程序的主要工作就是编写子函数，并填充 file_operations 的各个域。

下面就开始写子程序。

```
#include <linux/types.h> 基本的类型定义
#include <linux/fs.h> 文件系统使用相关的头文件
#include <linux/mm.h>
#include <linux/errno.h>
#include <asm/segment.h>
unsigned int test_major = 0;
static int read_test(struct inode *inode, struct file *file, char *b
{
int left; 用户空间和内核空间
if (verify_area(VERIFY_WRITE, buf, count) == -EFAULT )
return -EFAULT;
for(left = count ; left > 0 ; left--)
{
__put_user(1, buf, 1);
buf++;
}
return count;
}

```

这个函数是为read调用准备的。当调用read时，read_test()被调用，它把用户的缓冲区全部写1。buf 是read调用的一个参数。它是用户进程空间的一个地址。但是在read_test被调用时，系统进入核心态。所以不能使用buf这个地址，必须用__put_user()，这是kernel提供的一个函数，用于向用户传送数据。另外还有很多类似功能的函数。请参考，在向用户空间拷贝数据之前，必须验证buf是否可用。这就用到函数verify_area。为了验证BUF是否可以用。

```
static int write_test(struct inode *inode, struct file *file, const
{
return count;
}
static int open_test(struct inode *inode, struct file *file )
{
MOD_INC_USE_COUNT; 模块计数加以，表示当前内核有个设备加载内核当中去
return 0;
}
static void release_test(struct inode *inode, struct file *file )
{
MOD_DEC_USE_COUNT;
}

```

这几个函数都是空操作。实际调用发生时什么也不做，他们仅仅为下面的结构提供函数指针。

```
struct file_operations test_fops =
```

```
{
  read_test,
  write_test,
  open_test,
  release_test,
};
```

设备驱动程序的主体可以说是写好了。现在要把驱动程序嵌入内核。驱动程序可以按照两种方式编译。一种是编译进kernel，另一种是编译成模块(modules)，如果编译进内核的话，会增加内核的大小，还要改动内核的源文件，而且不能动态的卸载，不利于调试，所以推荐使用模块方式。

```
int init_module(void)
{
int result;
result = register_chrdev(0, "test", &test_fops); 对设备操作的整个接
if (result < 0) {
printk(KERN_INFO "test: can't get major number\n");
return result;
}
if (test_major == 0) test_major = result; /* dynamic */
return 0;
}

```

在用insmod命令将编译好的模块调入内存时，init_module 函数被调用。在这里，init_module只做了一件事，就是向系统的字符设备表登记了一个字符设备。register_chrdev需要三个参数，参数一是希望获得的设备号，如果是零的话，系统将选择一个没有被占用的设备号返回。参数二是设备文件名，参数三用来登记驱动程序实际执行操作的函数的指针。

如果登记成功，返回设备的主设备号，不成功，返回一个负值。

```
void cleanup_module(void)
{
unregister_chrdev(test_major, "test");
}
```

在用rmmod卸载模块时，cleanup_module函数被调用，它释放字符设备test在系统字符设备表中占有的表项。

一个极其简单的字符设备可以说写好了，文件名就叫test.c吧。

下面编译：

```
$ gcc -O2 -DMODULE -D__KERNEL__ -c test.c
```

—c 表示输出制定名，自动生成.o文件。

得到文件test.o就是一个设备驱动程序。

如果设备驱动程序有多个文件，把每个文件按上面的命令行编译，然后

```
ld -r file1.o file2.o -o modulename
```

驱动程序已经编译好了，现在把它安装到系统中去。

```
$ insmod -f test.o
```

如果安装成功，在/proc/devices文件中就可以看到设备test，并可以看到它的主设备号。要卸载的话，运行：

```
$ rmmod test
```

下一步要创建设备文件。

```
mknod /dev/test c major minor
```

c 是指字符设备，major是主设备号，就是在/proc/devices里看到的。

用shell命令

```
$ cat /proc/devices
```

就可以获得主设备号，可以把上面的命令行加入你的 shell script 中去。

minor是从设备号，设置成 0 就可以了。

我们现在可以通过设备文件来访问我们的驱动程序。写一个小小的测试程序。

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
main()
{
  int testdev;
  int i;
  char buf[10];
  testdev = open("/dev/test", O_RDWR);
  if ( testdev == -1 )
  {
    printf("Cann't open file \n");
    exit(0);
  }
  read(testdev, buf, 10);
  for (i = 0; i < 10;i++)
    printf("%d\n", buf[i]);
  close(testdev);
}
```

编译运行，看看是不是打印出全1 ？

以上只是一个简单的演示。真正实用的驱动程序要复杂的多，要处理如中断，DMA，I/O port等问题。这些才是真正的难点。上述给出了一个简单的字符设备驱动编写的框架和原理，更为复杂的编写需要去认真研究LINUX内核的运行机制和具体的设备运行的机制等等。希望大家好好掌握LINUX设备驱动程序编写的方法。

07
嵌入式驱动的结构分析

在Linux系统上编写驱动程序，说简单也简单，说难也难。难在于对算法的编写和设备的控制方面，是比较让人头疼的；说它简单是因为在Linux下已经有一套驱动开发的模式，编写的时候只需要按照这个模式写就可以了，而这个模式就是它事先定义好的一些结构体，在驱动编写的时候，只要对这些结构体根据设备的需求进行适当的填充，就实现了驱动的编写。

首先在Linux下，视一切事物皆为文件，它同样把驱动设备也看成是文件，对于简单的文件操作，无非就是 open/close/read/write，在Linux对于文件的操作有一个关键的数据结构：file_operation，它的定义在源码目录下的include/linux/fs.h 中，内容如下：

[cpp] view plain copy

```
1. struct file_operations {
2.     struct module *owner;
3.     loff_t (*llseek) (struct file *, loff_t, int);
4.     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
5.     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
6.     ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);

7.     ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);

8.     int (*readdir) (struct file *, void *, filldir_t);
9.     unsigned int (*poll) (struct file *, struct poll_table_struct *);
10.     int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
11.     long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
12.     long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
13.     int (*mmap) (struct file *, struct vm_area_struct *);
14.     int (*open) (struct inode *, struct file *);
15.     int (*flush) (struct file *, fl_owner_t id);
16.     int (*release) (struct inode *, struct file *);
17.     int (*fsync) (struct file *, int datasync);
18.     int (*aio_fsync) (struct kiocb *, int datasync);
19.     int (*fasync) (int, struct file *, int);
20.     int (*lock) (struct file *, int, struct file_lock *);
21.     ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
22.     unsigned long (*get_unmapped_area)
(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
23.     int (*check_flags)(int);
24.     int (*flock) (struct file *, int, struct file_lock *);
25.     ssize_t (*splice_write)
(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
26.     ssize_t (*splice_read)
(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
27.     int (*setlease)(struct file *, long, struct file_lock **);
28. };
```

对于这个结构体中的元素来说，大家可以看到每个函数名前都有一个“*”，所以它们都是指向函数的指针。目前我们只需要关心

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
int (*open) (struct inode *, struct file *);
int (*release) (struct inode *, struct file *);
```

这几条，因为这篇文章就叫简单驱动。就是读（read）、写（write）、控制（ioctl）、打开（open）、卸载（release）。这个结构体在驱动中的作用就是把系统调用和驱动程序关联起来，它本身就是一系列指针的集合，每一个都对应一个系统调用。

但是毕竟file_operation是针对文件定义的一个结构体，所以在写驱动时，其中有一些元素是用不到的，所以在2.6版本引入了一个针对驱动的结构体框架：platform，它是通过结构体platform_device来描述设备，用platform_driver描述设备驱动，它们都在源代码目录下的include/linux/platform_device.h中定义，内容如下：

[cpp] view plain copy

```
1. struct platform_device {
2.     const char   * name;
3.     int      id;
4.     struct device    dev;
5.     u32      num_resources;
6.     struct resource * resource;
7.     const struct platform_device_id *id_entry;
8.     /* arch specific additions */
9.     struct pdev_archdata     archdata;
10. };
11. struct platform_driver {
12.     int (*probe)(struct platform_device *);
13.     int (*remove)(struct platform_device *);
14.     void (*shutdown)(struct platform_device *);
15. int (*suspend)(struct platform_device *, pm_message_t state);
16.     int (*resume)(struct platform_device *);
17.     struct device_driver driver;
18.     const struct platform_device_id *id_table;
19. };
```

对于第一个结构体来说，它的作用就是给一个设备进行登记作用，相当于设备的身份证，要有姓名，身份证号，还有你的住址，当然其他一些东西就直接从旧身份证上copy过来，这就是其中的struct device dev，这是传统设备的一个封装，基本就是copy的意思了。对于第二个结构体，因为Linux源代码都是C语言编写的，对于这里它是利用结构体和函数指针，来实现了C语言中没有的“类”这一种结构，使得驱动模型成为一个面向对象的结构。对于其中的struct device_driver driver，它是描述设备驱动的基本数据结构，它是在源代码目录下的include/linux/device.h中定义的，内容如下：

[cpp] view plain copy

```
1. struct device_driver {
2.     const char       *name;
3.     struct bus_type      *bus;
4.     struct module        *owner;
```

```
5.      const char      *mod_name;  /* used for built-in modules */
6.      bool suppress_bind_attrs;   /* disables bind/unbind via sysfs */
7. #if defined(CONFIG_OF)
8.      const struct of_device_id   *of_match_table;
9. #endif
10.     int (*probe) (struct device *dev);
11.     int (*remove) (struct device *dev);
12.     void (*shutdown) (struct device *dev);
13.     int (*suspend) (struct device *dev, pm_message_t state);
14.     int (*resume) (struct device *dev);
15.     const struct attribute_group **groups;
16.     const struct dev_pm_ops *pm;
17.     struct driver_private *p;
18. };
```

依然全部都是以指针的形式定义的所有元素，对于驱动这一块来说，每一项肯定都是需要一个函数来实现的，如果不把它们集合起来，是很难管理的，而且很容易找不到，而且对于不同的驱动设备，它的每一个功能的函数名必定是不一样的，那么我们在开发的时候，需要用到这些函数的时候，就会很不方便，不可能在使用的时候去查找对应的源代码吧，所以就要进行一个封装，对于函数的封装，在C语言中一个对好的办法就是在结构体中使用指向函数的指针，这种方法其实我们在平时的程序开发中也可以使用，原则就是体现出一个“类”的感觉，就是面向对象的思想。

在Linux系统中，设备可以大致分为3类：字符设备、块设备和网络设备，而每种设备中又分为不同的子系统，由于具有自身的一些特殊性质，所以有不能归到某个已经存在的子类中，所以可以说是便于管理，也可以说是为了达到同一种定义模式，所以linux系统把这些子系统归为一个新类：misc，以结构体 miscdevice 描述，在源代码目录下的include/linux/miscdevice.h中定义，内容如下：

[cpp] view plain copy

```
1. struct miscdevice  {
2.      int minor;
3.      const char *name;
4.      const struct file_operations *fops;
5.      struct list_head list;
6.      struct device *parent;
7.      struct device *this_device;
8.      const char *nodename;
9.      mode_t mode;
10. };
```

对于这些设备，它们都拥有一个共同主设备号10，所以它们是以次设备号来区分的，对于它里面的元素，大应该很眼熟吧，而且还有一个我们更熟悉的list_head的元素，这里也可以应证我之前说的list_head就是一个桥梁的说法了。

其实对于上面介绍的结构体，里面的元素的作用基本可以见名思意了，所以不用赘述了。其实写一个驱动模块就是填充上述的结构体，根据设备的功能和用途写相应的函数，然后对应到结构体中的指针，然后再写一个入口一个出口（就是模块编程中的init和exit）就可以了，一般情况下入口程序就是在注册platform_device和platform_driver（当然，这样说是针对以platform模式编写驱动程序）。

08
嵌入式书籍推荐

1. 硬件方面的书：

微机原理、数字电路，高校里的教材。

2. Linux方面的书：

<ARM体系架构与编程>
<嵌入式Linux应用开发完全手册>
<Linux设备驱动>，老外写的那本
<linux设备驱动开发详解>
<linux内核完全注释>
<Linux内核情景分析>

在做驱动的时候，肯定会用到与内核相关的东西，或者需要和内核中的某些模块配合，这样你也要理解内核的某些部分是如何实现的，最后，你应该可以很好的掌握linux的内核整体框架是什么。

这些都是进步，都是在你一次又一次的开发中需要总结的东西，如果你不总结，永远都是从头开始（或者说永远都是还没看懂别人代码为什么这么做的时候，就去改它，然后可以工作了），就完事了，这样你永远也不可能提高，最后你就有了现在的这种感觉，觉得自己什么都不是，什么都不懂。

还有一点要说明的，现在有许多人搞linux开发，却不去用linux系统做为自己工作的平台，在这种情况下，你很难理解linux内核的实现机制，以及为什么要采用这种方式实现。

你都没用过linux系统，就想去实现一个与linux运行机理相符合的项目，这是不可能的。就是你这个项目成功了，它也肯定不是最优的，或者是不符合linux的使用习惯的（包括内核的扩展和应用程序的实现）。

所以，最后想说的是，你一定要定期总结，总结你这段时间做了什么，你从中得到了什么，为了你以后可以更好的做好类似的工作，你应该去看些其它的什么东西；二是你一定至少要在工作的开发环境中使用 linux 作为你的平时工作平台，而不要使用虚拟机和服务器，因为你只有完全了解了linux的使用，你才可以为它开发符合它规则的项目。

猜你喜欢：

一个很棒的智能配网方案！

分享一份嵌入式软件工具清单！

分享一款嵌入式人必备绘图工具！

分享一款小巧好用的代码对比工具

一个300多行代码实现的多任务管理的OS

分享嵌入式中几个实用的shell脚本！

点击**阅读原文**，查看更多分享。

点个  呗

喜欢此内容的人还喜欢

太好看了！东北大叔雪地作画引关注，网友：高手在民间

春晖学府

“我真的跑路了”！又一公司高调暴雷，创始人称钱早就洗干净了

易简财经

我家发生重大人员变动了

菜刀曦曦