蓝色字体代表自己的理解。红色字体代表疑问，紫色字体为代码示例

1. arm的版本号和用arm架构的CPU版本号是不一样的。
2. 内存可用的大小是由CPU寻址空间决定的，而且CPU的地址总线不是全部安排给内存。
3. ROM(FLASH)是用来放程序的，只可读；RAM是用来放数据的，可读写。
4. ARM是RISC架构，常用的指令只有二三十条（以基本指令为主），ARM是低功耗CPU。
5. ARM是32位CPU。

模拟

1. 地址映射就是指地址空间的分配。 （参考数据手册）
2. 1)ROM：read only memory（指不能通过总线去写，而是通过IO去写，硬盘，FLASH，对运行速度影响不大）

2)RAM:random access menmory 随机访问存储器（内存，与CPU直接进行数据交换，运行速度与此有关）。

3)IROM:internal rom 内部ROM，指的是集成到SoC内部的ROM

4)IRAM

5)DRAM:dynamic ram动态RAM

6)SRAM:静态RAM

8.FLSAH：NAND 、INAND、 U盘 、SSD（固态硬盘）、光盘。

9.CPU连接内存和外存的方式是不同的，与内存是通过地址总线和数据总线连接的，和外存是通过接口来连接的（需要经过时序）。

10.NorFlash可以总线式访问。

11. 1)SRAM:容量小，价格高，优点是不需要软件初始化直接上电就能用。

2）DRAM：与SRAM相对。

12.·现在的嵌入式系统大多采用的启动方案是：外接的大容量Nand+外接的大容量DRAM+SoC内置SRAM。

13. ARM约定：

1）HalfWord: 16bits

2) Word:32 bits(刚好ARM的CPU是32位的)

14.CPU各种模式的切换，可以是程序员通过代码主动切换（通过写CPSR寄存器）；也可以是CPU在某些情况下自动切换。

15.程序跑飞：PC寄存器内容错误。

16.异步：靠中断来实现。

17.异常会打断正在执行的工作，并且一般我们希望异常处理完成后继续回来执行原来的工作。

18.异常向量表是硬件向软件提供的处理异常的支持。

19.比较指令不用后面加s就可以影响cpsr中的标志位。

20.有返回值的函数重在结果，没有返回值的函数重在过程.

疑问：a.寄存器中的数据是32位的吗？为什么

b.一个地址对应的内存大小是8位吗？

19.伪指令是编译器环境提供的，目的是用来指导编译过程。

20.寄存器是通过名字指定的，而内存是通过地址指定的。

21.

22.条件后缀是否成立，取决于这句代码之前的代码运行后的结果；

而条件后缀本身是否成立又决定了本剧代码是否会被执行。

23.PC指向正在被取指的指令，比正在被执行的指令的地址晚两个周期（即多8个字节，一个周期4个字节）。

疑问：为什么说CPU与内存是通过总线连接的，但CPU访问内存还需要通过通用寄存器呢？

24.满栈=先传输再变地址。

A．满栈：进栈—先移动指针再存 出栈--先出数据再移动指针

B．空栈：xxx

C．减栈：进栈—指针向下移动 出栈—指针向上移动

D．增栈：xxxx

25.防止出错的小技巧：操作栈时用相同的后缀。

26.可以使用adr和ldr加载的地址比较来判断当前程序是否在链接时指定的地址运行。

27.由于笔记本电脑没有串口，所以笔记本电脑上看到的串口实际上是USB转串口后得到的虚拟串口。

28.写代码时需要查阅的和参考的资料有：核心板原理图、底板原理图、相应硬件的数据手册、S5PV210数据手册。

29.一个USB设备插到PC上，要么接到主机，要么接到虚拟机，默认接到主机。

30.iNand可以看成芯片化的SD卡或者MMC。

疑问：BL0、BL1是附属哪里的？

解答：S5PV210启动后先执行内部iROM中的BL0，外部设备的启动信息是为BL1;

疑问：从SD0启动时,BL0和BL1是否都在iROM?

31.从SD卡启动时，BL0首先会读取sd卡得到完整的镜像（完整的是指主体二进制代码和16字节的头），然后BL0会自己根据你的实际镜像来计算一个校验和checksum，然后和你完整镜像的头部中的checksum来对比。如果对应则执行BL1；如果不对应则启动失败（会转入执行2st启动，即SD2启动。（我们在通过破坏扇区数据实现外部SD卡启动时，实际上是破坏了uboot，导致BL0算出的校验和与uboot自带的校验和不对应---即SD checksum error。）

32.在申请比较大的内存时，一般是用malloc申请堆内存，并且要把这个申请到的堆内存清零（因为有可能这个堆里面有被写过—“脏“）。

33.ldr后面是#号时，ldr是指令；后面是=号时，ldr为伪指令。

疑问：合法立即数与非法立即数的概念？

34.代码示例：把0x11111111写入0xE0200240位置

ldr r0 =0x11111111 //使用ldr伪指令的目的是使用伪指令时，编译器会帮我们判断后面跟//的是合法立即数还是非法立即数。

ldr r1 =0xE0200240

str r0, [r1] //这一句是寄存器间接寻址：把r0中的值写到以r1中数据为地址的内存（寄存器）

35.CPU每个时钟周期都要执行程序。而我们实验中的裸机程序是直接在CPU上执行的，CPU执行完我们的程序之后还会继续运行，而且此后的行为是未定义的（“跑飞”），很危险，在裸机程序最后加一个死循环是这个问题的一种解决方法。

疑问：单片机程序会出现35中的问题吗？

36. .代表当前这一句指令的地址。

37.在对寄存器编程时，利用位运算可以提高代码的可读性。

38.汇编中主程序最后是一个死循环。（我认为c中main函数的最后也是一句没有任何操作的死循环，在运行最后看起来像是程序总在维持最后用户定义的状态）

39.代码实例：实现使一串二进制数据只有其中一位与其他不同（假设只让第n位为1）

--位取反：1.先使这串数据全为0

2.~（1<<(n-1)）

程序编译后编译器报出的warning一般不会影响运行效果。

40.编程操作一个硬件的步骤：1.分析硬件工作原理

2.分析原理图

3.分析数据手册

4.找到相关的SFR

5.写代码设置相关的寄存器得到想要的效果。

41.为什么需要反汇编：1.逆向破解

2.当你用非汇编语言写程序后，程序运行和预想不符的话，可以对生成的可执行程序进行反汇编得到程序的汇编语言版本，通过对比汇编版本程序与原始程序，有助于排查编程错误，同时可以加深对非汇编语言的理解。

42.反汇编工具是基于.elf文件进行反汇编的，.bin文件只有纯二进制数据，而.elf具有很多符号文件。（可以看成原始程序的另一个版本加链接需要的信息）

43.下载烧录执行的bin文件，内部其实是一条条的指令机器码。这些指令每一条都有一个指令地址，这个地址是链接时ld给指定的。

44.有时候我们写的链接脚本有误（或者我们不知道这个链接脚本会怎么样），这个时候可以通过看反汇编文件来分析这个链接脚本的效果，看是不是我们想要的，如果不是，可以改了再看。

45.电子设备上的程序经常会跑飞，可以通过看门狗来监视：程序正常时会定时喂狗，程序跑飞了之后不会喂狗的话，看门狗就会将设备复位，这个复位的动作既提醒了我们程序跑飞了，又阻止了程序跑飞后可能发生的危险动作。（无人值守的设备必须加看门狗）

46.c语言运行后是需要一定的环境的，在启动代码中这个环境由汇编构建。

47.整个系统中CPU的供应链由：寄存器+cache+DDR+硬盘/flash四阶组成，这是综合考虑了性能、成本后得到的妥协的结果。

48.cache的意义：指令平时是放在硬盘或flash中的。运行时读到DDR中，再从DDR中读给寄存器，再由寄存器送给CPU。但是DDR的速度和寄存器相差太大，如果CPU运行完一句再去DDR读取下一句，那么CPU的速度就完全被DDR拖慢了。解决方案就是icache。

icache在工作时，会把我们cpu正在运行的指令旁边的指令（因为CPU下一次执行旁边几条指令的可能性较高）事先读给到icache中。

下一句CPU要指令时，cache首先检查自己事先准备好的缓存指令有没有这句，如果有就直接拿给CPU去执行；如果没有则需要从DDR中重新去读取拿给CPU，并同时做一系列的动作：清缓存、重新缓存。（更新缓存的指令）

49.代码示例：将数据中的bit(n)置0或者置1

置0：bic r0,r0,#(1<<n)

置1：orr r0,r0,#(1<<n)

50.链接地址是我在对程序进行编译链接时认为并且指定的程序将来的运行地址，而运行地址是将来程序运行时的实际运行地址。（例子说明：实验中，makefile中用 -Ttext 0x0来指定链接地址是0x0.但实际上我们运行时的地址是0xd0020010，即dnw工具设置的运行地址。但这两个地址看似不同，实际上是相同的，因为S5PV210内部做了映射，把SRAM映射到0x0地址了）

51.位置有关代码·将来运行时如果运行地址不在链接地址处则不能运行，位置无关代码反之。

52.实验板官方推荐的启动方式：bootloader必须小于96KB并且大于16KB，假定bootloader为80KB，启动过程是这样子：先上电运行iRom的BL0，BL0会加载外部启动设备中的bootloader的前16KB（BL1）到SRAM中去运行，BL1运行时会加载BL2（Bootloder中80-16=64KB）到SRAM（从SRAM的16KB处开始用）中去运行；BL2运行时会初始化DDR并且将OS搬运到DDR去执行OS，启动完成。

53.uboot实际使用的方式：uboot大小随意，假定为200KB；启动过程是这样的：先开机上电运行BL0，BL0会加载外部启动设备中的前16KB到SRAM中去运行，BL1运行时会初始化DDR，然后将整个uboot搬运到DDR中，再用一句长跳转（从SRAM跳转到DDR）指令从SRAM直接跳转到DDR中继续执行uboot直到uboot执行结束。Uboot启动后在uboot命令行中去启动OS。

54.从源码到可执行程序的步骤：

1. 预编译：预编译器来执行。譬如C中的宏定义、#include、注释等就是由预编译器处理的。
2. 编译：编译器来执行。把源码.c、.s变成机器码.o文件（各个文件独立、各个函数也独立）。
3. 链接：链接器来执行。把.o文件中的各个函数（段）按一定规则（规则由链接脚本来指定）累积在一起，形成可执行文件。
4. Strip（过滤）：把可执行程序中的符号信息去掉。（相当于从debug版本变成release版本）
5. Objcopy：由可执行程序生成可烧录的镜像bin文件。

疑问：bin文件和可执行程序的release有什么区别？

55.C语言中全局变量如果未显式初始化，值是0。本质就是C语言把这类全局变量放在了bss段，从而保证了为0。

56.局部变量的值是随机的，因为局部变量是放在栈上的，而栈是被反复使用的，本次定义并且没有初始化的局部变量的值是上次被放在和这个局部变量在栈上相同位置的变量的值。

57.c运行时环境如何保证显式初始化为非0的全局变量的值在Main之前就被赋值了？

就是因为这类变量被放在了.data段中，而.data段会在main被执行之前被处理了。（初始化）

58.关于链接脚本：链接器会参考链接脚本，并且使用其中规定的规则来处理.o文件中的那些段，将其链接成一个可执行程序。

59.链接脚本的理解：

1. 链接脚本的关键内容有两部分：段名+地址（链接地址）
2. SECTION{ }这是整个链接脚本
3. .点号在链接脚本中代表当前位置。
4. =代表赋值。

60.链接器的指挥流程：看到了链接脚本中的段名，就根据这个段名去.o文件中找相应的段内容，然后再去看链接脚本里面段名对应的地址，最后把段内容放到这个对应的地址中去。

61.重定位代码的作用是：在PIC执行完之前（在代码中第一句位置有关码执行之前）必须将整个代码搬移到链接地址位置去执行，这就是重定位。

62.当链接地址和运行地址相同时，短跳转和长跳转实际效果是一样的；但是当链接地址不等于运行地址时，短跳转和长跳转就有差异了。

63.ldr和adr都是伪指令，区别是ldr是长加载、adr是短加载。

主要区别是adr指令在加载符号地址，加载的运行时地址；ldr指令在加载符号地址，加载的是链接地址。（可结合.dis文件中对应的语句、地址等来理解）

64.代码示例：

ldr r2 ,[r0],#4 //执行完这句后r0地址增长4字节

str r2 ,[r1],,#4 //执行完这句后r0地址增长4字节

65.重定位的实现过程：

1. 首先利用adr和ldr加载start地址，判断是否需要进行重定位。
2. 如果需要就将代码段和数据段复制到链接 地址。
3. 将链接地址对应的bss段清0；
4. 用一句长跳转跳转到链接地址。

66.定义汇编语言函数时，在函数的最后需要

例如（mov pc,lr）

进行函数返回。

67.代码示例：基址加变址寻址

ldr r0=xxx（基地址）

ldr r1=xxx（值）

str r1,[r0,xxx(偏移量)]

把r1的值加载到以（r0+偏移量）为地址的内存处

68.由于CPU和内部外设需要协同工作，SoC是同步系统。

69．PLL（锁相环）：用于倍频。

70.SoC内部有很多外设，这些外设不工作时最好关掉（这里的关掉指的是不使能它的时钟，而不是掉电），否则会作无谓的产热。

71.时钟来源的组成：晶振+时钟发生器+PLL+分频电路

72.210内部的各个外设都是接在（内部AMBA总线）总线上面的，AMBA总线有一条高频分支叫AHB，有一条低频分支叫APB。

AMBA总线上的各个域都有对应的HCLK\_XXX和PCLK\_XXX，其中HCLK\_XXX就是XXX这个域在中AHB总线的工作频率；PCLK\_\_XXX就是XXX这个域中APB总线的工作频率。

73.关于时钟频率的典型值：

当210刚上电时，默认是外部晶振+内部时钟发生器产生的24MHZ的时钟频率直接给ARMCLK。这时系统主频是24MHZ，运行很慢。

然后iROM代码初始化了时钟系统，给了系统一个默认的运行频率。这个频率是三星推荐的210工作性能和稳定性最佳的频率。

74.各个时钟频率的典型值如下（在数据手册里面有）：

• freq(ARMCLK) = 1000 MHz  
• freq(HCLK\_MSYS) = 200 MHz  
• freq(HCLK\_IMEM) = 100 MHz  
• freq(PCLK\_MSYS) = 100 MHz  
• freq(HCLK\_DSYS) = 166 MHz  
• freq(PCLK\_DSYS) = 83 MHz  
• freq(HCLK\_PSYS) = 133 MHz  
• freq(PCLK\_PSYS) = 66 MHz  
• freq(SCLK\_ONENAND) = 133 MHz, 166 MHz

75.差分信号传输比·电平信号传输更加稳定。

76.看起来似乎相同根数的通信线下，电平信号（一根线走一路电平信号）要比差分信号（两根线走一路差分信号）要快，但实际上还是差分信号传输更快，因为差分信号抗干扰能力强，因此一个发送周期更短。

76.实际上串行通信比并行通信更广，因为串行通信对（传输）线的要求更低（考虑成本），；如果想要提高速度也可以通过提高串行通信的传输速度来实现。

77.串口通信属于异步、串行、电平传输。

78.电平信号传输在长距离传输时压降比较明显。

79.电平信号是用信号线电平减去参考线电平，这个电压差决定了传输至是0还是1（要参考电平标准，例如RS232和TTL）

80.串口通信的一个通信单元（一个周期）：起始位+数据位+奇偶校验位+停止位。

81.数据位一般选择8位，因为我们通过串口传输的信息都是ASCII码中一个字符刚好编码为8位。

82．奇偶校验（把有效数据逐位相加）可以在一定程度上防止位反转（即0变成1,1变成0的情况）。

83.波特率为什么要收发双方一致：因为波特率约定了发送方发送一个信号的时间间隔、接收方接收一个信号的时间间隔，如果双方波特率不一致就会产生对信号的认识失误。即发送方一个1维持1s然后变成0又维持1s再变成1维持1s，这时接收方如果两秒读取一次就会认为发送方是发了两个1，而忽略中间过程的0信号。

84.通过串口发送数据首先要对待发送的数据进行编码，编码成二进制再进行逐个位的发送。

85.为什么需要流控：再发送方塑料比接收方快的适合流控可以保证发送和接收不会丢包。

86.由于典型串口通信需要需要CPU频繁地读写缓冲区（缓冲区大小只有1字节），这样导致CPU需要频繁地切换上下文，会增加很多开销。

对于这个问题有两个解决办法·：1）FIFO：CPU把很多字节的数据一次存到FIFO里面，FIFO再1字节1字节地和缓冲区交互，这种方法只能缓解问题，不能从根本解决问题。

2）DMA：DMA来处理数据的转移，直接把CPU解放出来，大大地减少了串口通信中CPU由于切换上下文而造成的开销。

87. uboot必须进行和硬件相对应的代码级别的更改和移植，才能保证可以从相应的启动介质启动。Uboot中第一阶段的start.S文件具体处理了这一块。

88.linux内核在设计的时候，设计为可以被传参，也就是说我们可以在uboot中事先给linux内核准备一些启动参数放在内存中特定位置然后传给内核，内核启动后会到这个特定位置去取uboot传给他的参数，然后在内核中解析这些参数，这些参数将被用来指导linux内核的启动过程。

89.环境变量和全局变量的区别：

A．环境变量被存储在Flash的另一块专门区域。

B . 全局变量的生命周期是在程序的一次运行中，开始运行时诞生，运行结束时死亡。

90．裸机程序uboot并不管理所有的内存，使用Uboot时注意内存管理，避免写数据时覆盖原来的数据。

91.go命令内部就是一个函数指针指向一个内存地址然后直接调用那个函数，即PC直接跳转到一个内存地址去运行。有一种调试裸机程序的方法就是事先启动uboot，然后在uboot中去下载裸机程序，用go命令去下载裸机程序。

92.由于uboot工作时没有操作系统支持，所以需要对Flash进行事先分区（实际上uboot和kernel都有个分区表）；

分区在系统移植前确定好，在uboot和kernel使用同一个分区表。

93.内存（DDR）的分区部署只在内核启动起来之前有作用，内核启动起来之后，内核的内存管理模块会接管整个内存空间。

94.决定uboot移植时要根据硬件做改动的程度原因：

针对于一款芯片，uboot官网会有一个对应的uboot1.0->SoC厂商基于这款芯片做出了一款SoC，于是他会基于uboot1.0做一个针对这款SoC的uboot2.0->开发板厂商基于这些SoC生产出自己的开发板，于是他会基于uboot2.0做一个针对这款开发板的uboot3.0。

95.uboot下的board文件夹下每一个文件夹都代表一个开发板，这个文件夹下面放的文件就是用来描述一个开发板的信息的。

也就是说，board目录下有多少个文件夹，就表示当前这个uboot已经被一直到多少个开发板了。

但由于开发板太多，后面又变成依据厂商来分类。

这一点影响的是uboot编译前配置的路径，

96. uboot下的cpu文件夹下每一个文件夹都代表一个SoC系列。



圈起来的是系列内共通的配置文件，划线的是系列内各个SoC各自不同的配置文件。

97.利用make 的参数O=输出目录或者export输出目录环境变量的方法可以将输出文件与源文件分离（Makefile中的OBJTREE和SRCTREE记录了源文件的根目录和输出文件的根目录），这样做的好处，一是避免了对源文件目录的污染，二是修改源代码编译时可以做到只影响一种配置的输出文件（即不同配置下的输出文件相互独立）。

98.Makefile中通过给mkconfig脚本传参来决定了mkconfig中导出的环境变量的值，其中的ARCH确定了CROSS\_COMPILE的值，通过CROSS\_COMPILE的值确定用什么cpu架构，

在这之后需要指定这个架构对应的交叉编译链的路径。（或者通过环境变量来指定）

99.config.mk中需要注意的地方：

1. 定义了编译工具。
2. 配置过程中生成的autoconfig.mk,里面的很多宏决定了uboot的条件编译，从而影响编译。

autoconfig.mk里面宏的值由源代码目录中的include/configs/xxx.h决定。每一个开发板的移植对应这个目录下的一个头文件。

100.Makefile的代码使得配置时可以将TEXT\_BASE写入config.mk中，这个TEXT\_BASE是uboot的链接地址（由于uboot启用了虚拟地址映射，所以链接地址的值要参考映射表），LDFLAGS=-Ttext+(TEXT\_BASE),由于-Ttext=0x0，所以LDFLAGS就是TEXT\_BASE。

101.mkconfig创建的符号链接：

1. 在include目录下创建asm文件，指向asm-arm。
2. 在include/asm-arm下创建一个arch文件，指向include/asm-arm/arch-s5pc110
3. 在include目录下创建regs.h文件，指向include/s5pc110.h，删除第二个。
4. 在include/asm-arm下创建一个arch文件，指向include/asm-arm/arch-s5pc11x
5. 在include/asm-arm下创建一个proc文件，指向include/asm-arm/proc-armv

102.mkconfig脚本里面会创建或者追加include/config.h文件，这个文件里面的内容是#include<configs/x210\_sd.h>,这个头文件是对我们所要移植的开发板的一些宏定义，这个x210\_sd.h会被用来生成一个autoconfiog.mk文件，这个autoconfig文件会被主Makefil引入，指导整个编译过程。

103．uboot链接脚本的代码段指明放在前面的代码，会在前16KB（BL1:进行初始化的工作）被调用。

104.基于三星s5pv210移植uboot:

1. 检查Makefile中的交叉编译工具链
2. 关于时钟可以在配置头文件（configs里面的头文件）里面修改（宏开关修改）
3. 可以在configs/xx.h里面:
4. 修改宏定义来改变作为控制台输入输出的串口。
5. 修改默认的环境变量（例如ip）
6. 行提示符

105. 总线式访问：可以通过寻址访问；

非总线式访问：要通过一定的时序访问。

106.基于三星官方开发板uboot进行uboot移植在网卡驱动上要做的修改有：

1. 网卡初始化。（网卡初始化函数）
2. 改驱动代码里的数据部分。（configs/xx.h）

107.uboot的configs/xx.h里面定义的CONFIG\_SETUP\_MEMORY\_TAGS和CONFIG\_CMDLINE\_TAG会影响uboot给内核传参（影响启动）

108.由于驱动程序工作在内核态，具有很高的权限，所以驱动程序漏洞会造成内核的不安全。

109.根文件系统提供根目录。

110.进程1存放在根文件系统里面

进程1是第一个用户态，是从内核态变为用户态的地方。

111.Linux系统的裁剪：linux内核在编译之前可以进行配置，配置时可以根据需求将组成内核的成千上万个模块进行裁剪。

112.系统的动态升级（不需要重新烧录系统，也不需要重新启动系统）是由模块化（条件编译来实现 ）来支持的。

113.选择SoC厂家移植版本会减少工作量。

114.linux内核中include里面有各种CPU架构共用的头文件，而各种CPU架构特有的头文件在arc目录中去找。

115.内核配置编译三部曲：

1. make xxx(首先得清楚你现在移植的内核是基于哪个内核的，比如是基于三星官方开发板移植成功的内核，那么这里的xxx就是arch/arm/configs/xxx，实际上就是继承了三星官方开发板的内核配置，之后在这基础上去做修改，这样做可以减少移植配置的工作量)
2. make menuconfig(提供了一个UI，供你在make xxx的配置基础上修改配置)
3. make，编译

116.linux内核中一个功能模块有三种编译选择：

1. 编入：将这个模块的代码直接编译链接zImage中去。
2. 去除：不将这个模块编译链接到zImage中。
3. 模块化：编译这个模块，但不将其链接到zImage中，而是将它单独链接成一个内核模块.ko文件，将来linux系统内核启动起来后可以动态的加载或者卸载这个模块。

117.kernel各级目录下的Kconfig的作用是向menuconfig提供配置项，menuconfig从.config中读取预配置信息，修改后的menuconfig会根据修改重新写.config。

118.内核源码目录树中每一个Kconfig都会source引入其所有子目录下的Kconfig，从而保证了所有的Kconfig项目都会被包含进menuconfig中。

·所以，如果自己在linux内核中添加了一个文件夹，一定要在这个文件夹中创建一个Kocndig,并且在这个文件夹的上一层目录的Kconfig中source引入这个新文件夹的Kconfig。

119.kernel的版本号在kernel的主Makefile最前面。

120.通过对链接脚本汇编文件的分析知道，内核启动的入口在head.S文件。

121.通过分析head.S可以知道内核运行的物理地址和虚拟地址。

因此，uboot中启动内核时要在上述的物理地址处启动。

122.内核的起始部分代码是被解压代码调用的。

Uboot启动内核后实际调用运行的是zImage前面那段未经压缩的解压代码，解压代码运行时先将zImage后段的内核解压开，然后再去调用运行真正的内核入口。

123.内核启动需要bootloader传参提供条件：

1. r0=0
2. r1=机器码
3. r2=uboot传参指针

（r0、r1、r2是arm的寄存器）

124.函数调用传参有两种方式：

1. 寄存器传参（arm体系就是通过寄存器传参）
2. 栈内存传参

125.内核最开始运行时MMU是关闭的，所以代码无法使用虚拟地址。

但整个内核使用的地址类型就是虚拟地址。

所以在MMU开启之前的内核代码必须是位置无关码，涉及到地址的必须使用物理地址。

126.开发板的机器码在arch/arm/tools.mach-types中查询。

127.内核刚运行时汇编所做的工作：

1. 检验
2. 建立段式页表
3. 开启MMU
4. 转入C阶段

128.进程1（init进程）完成了从内核态向用户态的转变：init进程刚开始运行时时内核态，它属于一个内核线程，然后它自己运行了一个用户态下面的程序强行把自己转成了用户态。

129.由于内核源代码下所有函数都处于内核态，进程1为了转成用户态，于是进程1挂载了根文件系统，并且找到了根文件系统下的init程序（用户态）来执行。

130.uboot传参（uboot环境变量）中有两个参数可以告诉内核根文件系统的位置和根文件系统的类型。

131.进程1如何找到根文件系统中的init程序？

1. 先去uboot传参（cmdline）指定的地方找。
2. 四个备用路径。

疑问：nfs文件系统是用服务器的资源还是客户端的资源？

132.在内核中把SoC内部外设相关的硬件操作代码就叫做平台设备驱动（里面是数据）。