[uboot链接脚本](http://blog.csdn.net/zzsfqiuyigui/article/details/7393365)

分类： [ARM 相关](http://blog.csdn.net/zzsfqiuyigui/article/category/1060473)2012-03-25 23:50 1211人阅读 [评论](http://blog.csdn.net/zzsfqiuyigui/article/details/7393365#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zzsfqiuyigui/article/details/7393365#report)

[cmd](http://www.csdn.net/tag/cmd)[脚本](http://www.csdn.net/tag/%e8%84%9a%e6%9c%ac)[编译器](http://www.csdn.net/tag/%e7%bc%96%e8%af%91%e5%99%a8)[存储](http://www.csdn.net/tag/%e5%ad%98%e5%82%a8)[flash](http://www.csdn.net/tag/flash)[工具](http://www.csdn.net/tag/%e5%b7%a5%e5%85%b7)

GNU编译器生成的目标文件缺省为elf格式，elf文件由若干段（section）组成，如不特殊指明，由C源程序生成的目标代码中包含如下段：.text(正文段)包含程序的指令代码；.data(数据段)包含固定的数据，如常量、字符串；.bss(未初始化数据段)包含未初始化的变量、数组等。C++源程序生成的目标代码中还包括.fini(析构函数代码)和.init(构造函数代码)等.链接器的任务就是将多个目标文件的.text、.data和.bss等段连接在一起，而连接脚本文件是告诉链接器从什么地址开始放置这些段.简而言之，由于一个工程中有多个.c文件，当它们生成.o文件后如何安排它们在可执行文件中的顺序，这就是链接脚本的作用.

(1).lds文件的完整形式  
   SECTIONS   
   {  
      ...  
      secname start BLOCK(align) (NOLOAD) : AT ( ldadr )  
    { contents } >region :phdr =fill  
      ...  
   }  
   <1>secname:段名，必须填写.  
   <2>contents:决定哪些内容可以放在该段，可以是整个目标文件，也可以是目标文件中的某段(代码段、数据段等).必须填写.  
   例程1:  
        SECTIONS   
       {   
          firtst 0x00000000 : { head.o init.o }   
          second 0x30000000 : AT(4096) { main.o }   
       }  
       head.o放在0x00000000地址开始处，init.o放在head.o后面，它们的运行地址是0x00000000，即连接和存储地址相同；main.o放在4096(0x1000，是AT指定的，存储地址)开始处，但它的运行地址在0x30000000，运行之前需要从0x1000(加载处)复制到0x30000000(运行处)，此过程也就用到了读取Nand flash。这就是存储地址和连接(运行)地址的不同，称为加载时域和运行时域，可以在.lds连接脚本文件中分别指定.  
   例程2:  
        ENTRY(begin)   
        SECTION   
       {   
          .=0x00300000;   
          .text : { \*(.text) }   
          .data: { \*(.data) }   
          .bss: { \*(.bss) }   
       }    
       ENTRY(begin)指明程序的入口点为begin标号；   
       .=0x00300000指明目标代码的起始地址为0x00300000，这一段地址可以是SDRAM的起始地址；  
       .text : { \*(.text) }表示从0x00300000开始放置所有目标文件的代码段，   
       .data: { \*(.data) }表示数据段从代码段的末尾开始，  
       再后是.bss段.

(2)二进制化  
   连接生成的elf文件还不能直接下载执行，通过objcopy工具可生成最终的二进制文件:  
       arm-linux-objcopy -O binary bootstrap.elf bootstrap.bin  
   其中: -O binary指定生成为二进制格式文件。  
         Objcopy还可以生成S格式的文件，只需将参数换成-O srec。  
   如果想将生成的目标代码反汇编，还可以用objdump工具：   
         arm-linux-objdump -D bootstrap.elf   
   至此，所生成的目标文件就可以直接写入Flash中运行了.

(3)uboot.lds文件分析  
  uboot的链接脚本文件放在arch\arm\cpu\arm920t下.具体内容如下:   
    
  OUTPUT\_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm"")  
     /\*指定输出可执行文件是elf格式、32位ARM指令、小端\*/  
 OUTPUT\_ARCH(arm) /\*指定输出可执行文件的平台为ARM\*/  
 ENTRY(\_start)    /\*指定输出可执行文件起始代码段为\_start\*/  
 SECTIONS  
 {  
   . = 0x00000000    /\*指明目标代码起始地址从0x0开始，"."代表当前位置\*/  
   . = ALIGN(4)     /\*代码以4字节对齐\*/  
   .text :         /\*指定代码段:必须将start.o文件放在代码段的开始位置，其它文件可任意放\*/  
   {  
      cpu/arm920t/start.o (.text)  /\*代码段第一部分，指明start.s是入口程序，被放到代码段开头\*/  
      \*(.text)                 /\*其它代码部分.其中，\*表示其它任意文件\*/  
   }  
   . = ALIGN(4)   
   .rodata : { \*(.rodata) }    /\*指定只读数据段，RO段\*/  
   . = ALIGN(4);  
   .data : { \*(.data) }       /\*指定读/写数据段，RW段\*/  
   . = ALIGN(4);  
   .got : { \*(.got) }        /\*指定got段, got段式是uboot自定义的一个段, 非标准段\*/  
     
   \_\_u\_boot\_cmd\_start = .     /\*把\_\_u\_boot\_cmd\_start赋值为当前位置, 即起始位置\*/  
   .u\_boot\_cmd : { \*(.u\_boot\_cmd) }   /\*指定u\_boot\_cmd段, uboot把所有的uboot命令放在该段\*/  
   \_\_u\_boot\_cmd\_end = .      /\*把\_\_u\_boot\_cmd\_end赋值为当前位置,即结束位置\*/  
   . = ALIGN(4);  
   \_\_bss\_start = .          /\*把\_\_bss\_start赋值为当前位置,即bss段的开始位置\*/  
   .bss : { \*(.bss) }       /\*指定bss段\*/  
   \_end = .               /\*把\_end赋值为当前位置,即bss段的结束位置\*/  
 }   
 通过上面的分析可以看出:  
 <1>由于在链接脚本中规定了文件start.o(对应于start.S)作为整个uboot的起始点，因此启动uboot时会执行首先执行start.S  
 <2>一般来说，内存空间可分为代码段、数据段、全局变量段、未初始化变量区、栈区、堆区等.其中，栈区由指针SP决定，堆区实质上是由C代码实现的，其它段则由编译器决定.从上面的分析可以看出，从0x00000000地址开始，编译器首先将代码段放在最开始的位置，然后是数据段，然后是bss段(未初始化变量区).   
文章出处：飞诺网(www.diybl.com):http://www.diybl.com/course/6\_system/linux/Linuxjs/20101230/550224.html