这里所说的ARM系统基本文件格式，都是在基于ARM的嵌入式系统开发中常会碰到的文件格式。 ARM系统基本文件格式有三种：1) BIN，平板式二进制格式，一般用于直接烧写到Flash中，也可以用于加载到monitor程序中。2) ELF，EXECUTABLE AND LINKABLE FORMAT，一种通用的OBJECT文件格式，一般由GNU COMPILER COLLECTION (GCC)产生。3) AXF，BIN格式的扩展版,主体部分同BIN，在文件头和尾加入了调试用的信息，用于AXD。 本文主要讨论BIN与ELF。 首先说明，ELF格式是一种OBJECT文件格式。一般OBJECT文件都可以分成三类：可重定位OBJECT文件，可执行OBJECT文件，共享OBJECT文件。ELF格式文件也可以分成这三种。 首先说说可重定位OBJECT文件。这种OBJECT文件一般由GCC中的ASSEMBLER(as)产生(请不要认为GCC只是编译器)，里面除了二进制的机器代码，还有一些可用于进行重定位的信息。它主要是作为LINKER(ld)的输入，LINKER将跟据这些信息，将需要重定位的符号重定位，进而产生可执行的OBJECT文件。ELF格式的可重定位OBJECT文件由header与section组成。 Header 包括 ELF header 与 section header. ELF header 位于文件的头部，用于存储目标机器的架构，大小端配置，ELF header大小，object文件类型，section header 在文件中的偏移，section header 的大小，section header 中的项目数等信息。Section header 则定义了文件中每个section 的类型，位置，大小等信息。 Linker就是通过查找ELF header，找到section header 的入口，再在section header 中找到相应的 section 入口，进而定位到目标section 的。

Section 包括 .text ：经过编译的机器代码。.rodata ：只读的数据，例如printf(“hello!”)中的字符串hello。.data ：已初始化的全局变量，局部变量将在运行时被存放在堆栈中，不会在.data或 .bss段中出现。.bss ：未初始化的全局变量，在这里只是一个占位符，在object文件中并没有实际的存储空间。.symtab ：符号表，用于存放程序中被定义的或被引用到的全局变量和函数的信息。.rel.text ：一个保存着一系列在.text中的位置的列表。这些位置将在linker把这个文件与其它object文件合并时被修改，一般来说，这些位置都是保存着一些引用到全局变量或者外部函数的指令。引用局部变量或者本地函数的指令是不需要被修改的，因为局部变量和本地函数的地址一般都是使用PC相对偏移地址的。需要注意的是，这个section 和下面的.rel.data在运行时并不需要，生成可执行的ELF object文件时会去掉这个section。.rel.data ：保存全局变量的重定位信息。一般来说，如果一个全局变量它的初始化值是另一个全局变量的地址，或者是外部函数的地址，那么它就需要被重定位。.debug ：保存debug信息。.strtab ： 一个字符串表，保存着.symtab和.debug ,和各个section的名字。.symtab，.debug 和section table里面，凡是保存name的域，其实都是保存了一个偏移值，通过这个偏移值在这个字符串表里面可以找到相应得字符串。 下面仔细讨论一下.symtab： 每一个可重定位的object文件，都会有一个.symtab。这个符号表保存了在这个object文件中所有被定义的和被引用的符号。当源程序是C 语言程序时，.symtab 中的符号直接来源于C编译器(cc1)。这里所说的符号主要有三种：1) 在这个object文件中被定义的可以被其他object文件全局符号。在C语言源程序中，主要就是那些非静态（没有static 修饰的）的全局变量和非静态的函数。在ARM汇编语言中，就是那些 被EXPORT 指令导出的变量。2) 在这个object文件中引用到，但是在其他文件中定义的全局变量。在ARM汇编语言中就是通过IMPORT命令引入的变量3) 本地变量。本地变量只在本object文件内可见。这里的本地变量指的是连接器本地变量，应该和一般的程序本地变量作区别。这里所指的本地变量，包括用 static 修饰的全局变量，object文件中section名称，源代码文件名称。一般意义上的本地变量，是在运行时由系统的运行时环境管理的，linker并不关心。 每个符合上面条件的符号在.symtab文件中都会有一个数据项。这个数据项的数据结构是：Typedef struct{ int name;//符号名称，其实就是.strtab的偏移值 int value;//在section中的位置，以相对section地址的偏移表示 int size;//大小 char type;//类型，一般是数据或函数 char binding;//是本地变量还是全局变量 char reserved;//保留的位 char section;//符号所属的section。可选有：.text(用数字1代表)，.data(用数 //3代表)，ABS（不应被重定位的符号），UND（在本object文件 //中未定义的符号，可能在别的文件中定义），COM（一般的未初//始化的变量符号）。}ELF\_sym 现在假设组成应用的各个模块都已经被汇编，构建出了可重定位的object文件。这些object的结构都是一样的，有各自的.text, .data section, 有各自的.symtab. GCC下一步要做的就是使用linker (ld),把这些object文件，加上必要的库连接成具有绝对运行时地址的可执行文件，就是可执行的ELF格式的文件。 Linker 的连接动作可以分为两部分：1) 符号解析。确定引用符号的指向。2) 符号重定位。合并section, 分配运行时环境地址，引用符号重定位。 符号解析： 在一个object文件中，有指令定义了符号，也有指令引用了符号。可能存在这样一种情况，一个被引用到的符号，有多重的定义。符号解析的作用就是确定，在这个object文件中，一个符号引用真正引用的是哪个符号。 在编译的时候，除了在本文件中定义的全局变量会由编译器生成一个符号表项之外，当发现一个被引用到的符号在本文件中并没有被定义，编译器也会自动产生一个符号表项，把确定这些引用的工作留给linker。汇编器在汇编时将读取这些符号表项，生成.symtab。在读取的过程中，如果发现有在无法确定的符号引用项，汇编器会为这些符号额外生成一个数据项，称作重定位数据项，存放于rel.text或rel.data section中，交由linker确定。下面是重定位数据项（relocation entry）的数据结构：Typedef struct{ int offset;//指明需要被重定位的引用在object中的偏移，实际上就是需要被重定位的引用 //在object中的实际位置 int symbol;//这个被重定位的引用真实指向的符号 int type;//重定位类型：R\_ARM\_PC24:使用24bit的PC相对地址重定位引用 //R\_ARM\_ABS32:使用32bit绝对地址重定位引用}Elf32\_Rel Linker 需要解析的，就是那些被生成了重定位数据项的引用。Linker将根据C语言定义的规则，对于每一个重定位数据项，在输入的各个object文件中查找适合的符号，把这个符号填入symbol项中。但是由于还不知道这个符号的真实地址，所以现在就算知道了引用的真实指向，但我们还是不能确定这个引用指向的地址。 符号重定位： 符号重定位用来解决上面的问题。Linker首先进行section 的合并。Linker合并 object文件的过程很简单，一般就是相同属性的section合并，例如不同object文件的.text section 将被合并成一个.text。同样，.symtab section也被合并成一个.symtab。这里面涉及到两个问题：1) 各个object文件合并的顺序。这个问题涉及到最终指令和符号的运行地址。最为重要的是，究竟是哪个section排在最前头？在ARM RAW 系统得开发过程中，这个最为重要。 ARM系统CPU上电后，系统会自动的从0x00000000地址取指令并执行，这个地址上映射着存储器。这个动作是不可编程的。所以排在最前面的 section一定要包含有程序的入口点，否则系统无法正常运行。2) 输入段与输出端之间的对应关系。理论上，任何section,都可以被随意的映射到一个输出段中。一个.data section是可以与一个.text section 组成输出一个.text的。当然这样的动作毫无意义。我们必须告诉linker使用那些section作为输入，产生一个输出section. 以上这两个问题，都是通过一个称为连接脚本的文件控制的。Linker通过读取连接脚本，来决定section 从输入到输出的映射，设置程序的入口点，设置哪个section应该在整个可执行文件的头部等问题。 连接脚本还有另外一个作用，那就是指定每个section的地址。在section 合并完成后，linker将跟据.symtab，对符号进行统一的编址，分配一个绝对的运行时地址。这个地址是以section地址作为基地址的。假设.text section的地址是0x00000000，那么.text里面的符号将以0x00000000这个地址作为基准地址。指定section地址的工作也是由连接脚本完成。在嵌入式开发中常见的在编译工程时需指定的text\_base, data\_base等参数，最后会被加入到连接脚本中，从而完成section的地址分配。 以上两步完成后，linker 执行引用符号重定位操作。Linker遍历.rel section (包括.rel text 和 .rel data)，对于其中的每个数据项，根据symbol域到.symtab 中查出相应的引用的真实地址(经过上面的地址分配，现在.symtab里面的符号都具有绝对的运行地址)，再根据offset域提供的偏移，将这个地址填入相应的位置上。 至此，符号重定位工作全部完成。Linker删除用于保存重定位信息的rel.text和rel.data section，加入一个segment header和 一个.init section。生成可执行的ELF格式的object文件。 Segment header 保存了用于操作系统内存映射的信息。.init section 包含了一个\_init 的函数。程序加载时，操作系统的程序加载器通过读取 segment header，将程序加载到用户内存空间，并根据segment header里面映射信息，分别将.text 段和.data段映射到适当的地址上。然后再调用.init中的\_init函数，完成初始化工作。 由于ELF文件具有通用性强的优点，现在流行的开发模式是，先通过编译工具生成ELF文件格式的可执行文件，在使用外部工具，抽离出ELF文件中的相应部分，生成BIN文件。例如著名的 GNU bootloader U-Boot，就采用了这种做法，编译器工具集是GCC，BIN生成工具是elf2bin。ARM公司著名的开发环境 ADS，虽然使用的是自家的armcc,和armcpp编译器，但他们的工作方式却是与GNU GCC如出一辙