内部资料

技术笔记

PKU/SSDB-03-TN-005

JBEOS-TN-03-005

2003 年 5 月

ELF 文件格式分析

滕启明

北京大学信息科学技术学院操作系统实验室

2003 年 5 月

声　明

本研究工作是在国家 863 计划软件重大专项《构件化嵌入式操作系统及开发环境》资助下开展的。

摘要　嵌入式操作系统应用领域广，硬件环境复杂多样，降低开发成本、缩短开发周期、提高产品质量是工业界和学术界共同关注的问题。借鉴软件复用的思想，采用基于构件的软件开发思路来开发嵌入式操作系统是一条可行的途径。本文是作者在探索系统软件构件的复用技术的过程中生成的技术笔记，重点分析了 UNIX 类操作系统中普遍采用的目标文件格式 ELF（Executable and Linkable Format），目的是研究操作系统中二进制级软件构件的静态、动态组装技术。本文首先介绍 ELF 文件格式规范，然后结合一个简单的 C 语言程序，分析编译、链接后生成的可重定位、可执行格式实例。

关键词　操作系统 编译 链接 目标文件 软件构件

目 录

图形目录

1 简介

可执行链接格式（Executable and Linking Format）最初是由 UNIX 系统实验室（UNIX System Laboratories，USL）开发并发布的，作为应用程序二进制接口（Application Binary Interface，ABI）的一部分。工具接口标准（Tool Interface Standards，TIS）委员会将还在发展的 ELF 标准选作为一种可移植的目标文件格式，可以在 32 位 Intel 体系结构上的很多操作系统中使用[1, 2]。

ELF 标准的目的是为软件开发人员提供一组二进制接口定义，这些接口可以延伸到多种操作环境，从而减少重新编码、重新编译程序的需要。接口的内容包括目标模块格式、可执行文件格式以及调试记录信息与格式等。

TIS 给出的 Portable Formats Specification 1.1 版本中主要针对三种不同类型的目标文件作了规定，并规定了程序加载与动态链接相关过程细节，给出了标准 ANSI C 和 libc 例程必须提供的符号[1]。在该组织随后发布的 Executable and Linking Format(ELF) Specification 1.2 版本中则分如下三个部分，主要的不同点是对与操作系统相关的部分进行了重新组织：

- Book I: Executable and Linking Format，描述 ELF 目标文件格式;
- Book II: Processor Specific(Intel Architecture)，描述 ELF 中与硬件相关的信息;
- Book III: Operating System Specific，描述 ELF 中与操作系统相关的部分，例如 System V Release 4 信息等。

2 相关标准

2.1 System V ABI

System V Application Binary Interface(ABI)为已编译的应用程序定义了系统接口，同时也为安装脚本支持提供了一个最小环境。其目的是为应用程序提供一个标准的二进制接口，使得这些程序能够运行在符合 *X/Open Commen Application Environment Specification, Issue 4.2* 和 *System V Interface Definition, Fourth Edition* 的操作系统上。二进制要包含与不同处理器体系结构相关的信息，所以 ABI 并不是只有一个规范，而是一个规范体系。System V ABI 由两个部分组成：一个通用的部分，描述 System V 在所有硬件平台上都一致的接口[3]；一个处理器相关的部分，描述特定于某个处理器体系结构的具体实现[4]。

System V ABI 的主要参考标准包括：

- 目标系统处理器的体系结构手册
- System V Interface Definition（接口定义），第 4 版
- IEEE POSIX 1003.1-1990 标准操作系统规范
- X/Open Common Application Environment Specification(CAE), Issue 4.2
- International Standard, ISO/IEC 9899:1990(E), Programming Languages – C, 12/15/90.
- X Window System™, X Version 11, Release 5, 图形用户界面规范

2.2 LSB

由于我们所关心的主要是 Linux 平台上目标文件的格式，所以 Linux 标准 LSB (Linux Standard Base)也是重要的参考资料。LSB 的目标是增强不同的 Linux 发布版本之间的兼容性，与 ABI 类似，也由两个部分组成：

- gLSB（Generic LSB） 适用于所有体系结构
- archLSB（Architecture Specific LSB） 特定某种体系结构的 LSB

目前，LSB 由 SourceForge 开放源码项目社区提供支持。

3 ELF 文件格式

3.1 简介

目标文件有三种类型：

- 可重定位文件（Relocatable File） 包含适合于与其他目标文件链接来创建可执行文件或者共享目标文件的代码和数据。
- 可执行文件（Executable File） 包含适合于执行的一个程序，此文件规定了 exec() 如何创建一个程序的进程映像。
- 共享目标文件（Shared Object File） 包含可在两种上下文中链接的代码和数据。首先链接编辑器可以将它和其它可重定位文件和共享目标文件一起处理，生成另外一个目标文件。其次，动态链接器（Dynamic Linker）可能将它与某个可执行文件以及其它共享目标一起组合，创建进程映像。

目标文件全部是程序的二进制表示，目的是直接在某种处理器上直接执行。

3.1.1 目标文件中的数据表示

目标文件格式支持 8 位字节/32 位体系结构。不过这种格式是可以扩展的，目标文件因此以某些机器独立的格式表达某些控制数据，使得能够以一种公共的方式来识别和解释其内容。目标文件中的其它数据使用目标处理器的编码结构，而不管文件在何种机器上创建。

表 1 ELF 中常用数据格式

名称	大小	对齐	目的
Elf32_Addr	4	4	无符号程序地址
Elf32_Half	2	2	无符号中等整数
Elf32_Off	4	4	无符号文件偏移
Elf32_SWord	4	4	有符号大整数
Elf32_Word	4	4	无符号大整数
unsigned char	1	1	无符号小整数

目标文件中的所有数据结构都遵从“自然”大小和对齐规则。如果必要，数据结构可以包含显式的补齐，例如为了确保 4 字节对象按 4 字节边界对齐。数据对齐同样适用于

文件内部。

3.2 目标文件格式

目标文件既要参与程序链接又要参与程序执行。出于方便性和效率考虑，目标文件格式提供了两种并行视图，分别反映了这些活动的不同需求。

图 1 目标文件格式

<table>
<tr><th>链接视图</th><th>执行视图</th></tr>
<tr><td>ELF 头部</td><td>ELF 头部</td></tr>
<tr><td>程序头部表（可选）</td><td>程序头部表</td></tr>
<tr><td>节区 1</td><td rowspan="2">段 1</td></tr>
<tr><td>...</td></tr>
<tr><td>节区 n</td><td rowspan="2">段 2</td></tr>
<tr><td>...</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>节区头部表</td><td>节区头部表（可选）</td></tr>
</table>

文件开始处是一个 *ELF 头部（ELF Header）*，用来描述整个文件的组织。节区部分包含链接视图的大量信息：指令、数据、符号表、重定位信息等等。

程序头部表（Program Header Table），如果存在的话，告诉系统如何创建进程映像。用来构造进程映像的目标文件必须具有程序头部表，可重定位文件不需要这个表。

节区头部表（Section Heade Table）包含了描述文件节区的信息，每个节区在表中都有一项，每一项给出诸如节区名称、节区大小这类信息。用于链接的目标文件必须包含节区头部表，其他目标文件可以有，也可以没有这个表。

> 注意：尽管图中显示的各个组成部分是有顺序的，实际上除了 ELF 头部表以外，其他节区和段都没有规定的顺序

3.3 ELF Header 部分

文件的最开始几个字节给出如何解释文件的提示信息。这些信息独立于处理器，也独立于文件中的其余内容。ELF Header 部分可以用下图中的数据结构表示：

图 2 ELF Header 数据结构

```
#define EI_NIDENT 16
typedef struct{
    unsigned char   e_ident[EI_NIDENT];
    Elf32_Half  e_type;
    Elf32_Half  e_machine;
    Elf32_Word  e_version;
    Elf32_Addr  e_entry;
    Elf32_Off   e_phoff;
    Elf32_Off   e_shoff;
```

```
    Elf32_Word  e_flags;
    Elf32_Half  e_ehsize;
    Elf32_Half  e_phentsize;
    Elf32_Half  e_phnum;
    Elf32_Half  e_shentsize;
    Elf32_Half  e_shnum;
    Elf32_Half  e_shstrndx;
}Elf32_Ehdr;
```

其中，e_ident 数组给出了 ELF 的一些标识信息，这个数组中不同下标的含义如表 2 所示：

表 2 e_ident[] 标识索引

名称	取值	目的
EI_MAG0	0	文件标识
EI_MAG1	1	文件标识
EI_MAG2	2	文件标识
EI_MAG3	3	文件标识
EI_CLASS	4	文件类
EI_DATA	5	数据编码
EI_VERSION	6	文件版本
EI_PAD	7	补齐字节开始处
EI_NIDENT	16	e_ident[]大小

这些索引访问包含以下数值的字节：

表 3 e_ident[]的内容说明

索引	说明
EI_MAG0 到 EI_MAG3	魔数（Magic Number），标志此文件是一个 ELF 目标文件。 **名称 取值 位置** EI_MAG0 0x7f e_ident[EI_MAG0] EI_MAG1 'E' e_ident[EI_MAG1] EI_MAG2 'L' e_ident[EI_MAG2] EI_MAG3 'F' e_ident[EI_MAG3]
EI_CLASS	标识文件的类别，或者说，容量。 **名称 取值 位置** ELFCLASSNONE 0 非法类别 ELFCLASS32 1 32 位目标 ELFCLASS64 2 64 位目标 ELFCLASS32 支持虚存范围 4 GB。ELFCLASS64 是为 64 位预留的，不过文件中的其他内容都没有针对 64 位定义。
EI_DATA	字节 e_ident[EI_DATA] 给出处理器特定数据的数据编码方式。

	名称	**取值**	**位置**
	ELFDATANONE	0	非法数据编码
	ELFDATA2LSB	1	高位在前
	ELFDATA2MSB	2	低位在前
EI_VERSION	ELF 头部的版本号码，不前此值必须是 EV_CURRENT。		
EI_PAD	标记 e_ident 中未使用字节的开始。初始化为 0。		

某些目标文件控制结构可以增长，因为 ELF 头部能够给出它们的实际大小。如果目标文件格式确实发生改变，可能会发生控制结构比预期的大或者小的情况。在这种情况下，忽略这些信息是允许的。至于对“缺失”信息的处理方式则要依赖于上下文，如果定义了扩展的话，将会对这些做出规定。

注意：在 32 位 Intel 体系结构上要求：
1、标志

位置	**取值**
e_ident[EI_CLASS]	ELFCLASS32
e_ident[EI_DATA]	ELFDATA2LSB

2、处理器标识（e_machine）成员必须是 EM_386。

ELF Header 中各个字段的说明如表 4：

表 4 ELF Header 中各个字段的含义

成员	说明
e_ident	目标文件标识
e_type	目标文件类型： **名称 取值 含义** ET_NONE 0 未知目标文件格式 ET_REL 1 可重定位文件 ET_EXEC 2 可执行文件 ET_DYN 3 共享目标文件 ET_CORE 4 Core 文件（转储格式） ET_LOPROC 0xff00 特定处理器文件 ET_HIPROC 0xffff 特定处理器文件 ET_LOPROC 和 ET_HIPROC 之间的取值用来标识与处理器相关的文件格式。
e_machine	给出文件的目标体系结构类型 **名称 取值 含义** EM_NONE 0 未指定 EM_M32 1 AT&T WE 32100 EM_SPARC 2 SPARC EM_386 3 Intel 80386 EM_68K 4 Motorola 68000 EM_88K 5 Motorola 88000

	EM_860 7 Intel 80860 EM_MIPS 8 MIPS RS3000 其它值都是保留的。特定处理器的 ELF 名称会使用机器名来进行区分。
e_version	目标文件版本 **名称 取值 含义** EV_NONE 0 非法版本 EV_CURRENT 1 当前版本
e_entry	程序入口的虚拟地址。如果目标文件没有程序入口，可以为 0。
e_phoff	程序头部表格（Program Header Table）的偏移量（按字节计算）。如果文件没有程序头部表格，可以为 0。
e_shoff	节区头部表格（Section Header Table）的偏移量（按字节计算）。如果文件没有节区头部表格，可以为 0。
e_flags	保存与文件相关的，特定于处理器的标志。标志名称采用 EF_machine_flag 的格式。
e_ehsize	ELF 头部的大小（以字节计算）。
e_phentsize	程序头部表格的表项大小（按字节计算）。
e_phnum	程序头部表格的表项数目。可以为 0。
e_shentsize	节区头部表格的表项大小（按字节计算）。
e_shnum	节区头部表格的表项数目。可以为 0。
e_shstrndx	节区头部表格中与节区名称字符串表相关的表项的索引。如果文件没有节区名称字符串表，此参数可以为 SHN_UNDEF。

3.4 节区（Sections）

节区中包含目标文件中的所有信息，除了：ELF 头部、程序头部表格、节区头部表格。节区满足以下条件：

(1). 目标文件中的每个节区都有对应的节区头部描述它，反过来，有节区头部不意味着有节区。
(2). 每个节区占用文件中一个连续字节区域（这个区域可能长度为 0）。
(3). 文件中的节区不能重叠，不允许一个字节存在于两个节区中的情况发生。
(4). 目标文件中可能包含非活动空间（INACTIVE SPACE）。这些区域不属于任何头部和节区，其内容未指定。

3.4.1 节区头部表格

ELF 头部中，e_shoff 成员给出从文件头到节区头部表格的偏移字节数；e_shnum 给出表格中条目数目；e_shentsize 给出每个项目的字节数。从这些信息中可以确切地定位节区的具体位置、长度。

节区头部表格中比较特殊的几个下标如下：

表 5 节区头部表格中的特殊下标

名称	取值	说明
SHN_UNDEF	0	标记未定义的、缺失的、不相关的，或者没有含义的

		节区引用
SHN_LORESERVE	OXFF00	保留索引的下界
SHN_LOPROC	0XFF00	保留给处理器特殊的语义
SHN_HIPROC	0XFF1F	
SHN_ABS	OXFFF1	包含对应引用量的绝对取值。这些值不会被重定位所影响
SHN_COMMON	OXFFF2	相对于此节区定义的符号是公共符号。如 FORTRAN 中 COMMON 或者未分配的 C 外部变量。
SHN_HIRESERVE	0XFFFF	保留索引的上界

介于 SHN_LORESERVE 和 SHN_HIRESERVE 之间的表项不会出现在节区头部表中。

3.4.2 节区头部

每个节区头部可以用如下数据结构描述：

图 3 节区头部数据结构

```
typedef struct{
    Elf32_Word sh_name;
    Elf32_Word sh_type;
    Elf32_Word sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off  sh_offset;
    Elf32_Word sh_size;
    Elf32_Word sh_link;
    Elf32_Word sh_info;
    Elf32_Word sh_addralign;
    Elf32_Word sh_entsize;
}Elf32_Shdr;
```

对其中各个字段的解释如下：

表 6 节区头部字段说明

成员	说明
sh_name	给出节区名称。是节区头部字符串表节区（Section Header String Table Section）的索引。名字是一个 NULL 结尾的字符串。
sh_type	为节区的内容和语义进行分类。参见节区类型。
sh_flags	节区支持 1 位形式的标志，这些标志描述了多种属性。
sh_addr	如果节区将出现在进程的内存映像中，此成员给出节区的第一个字节应处的位置。否则，此字段为 0。
sh_offset	此成员的取值给出节区的第一个字节与文件头之间的偏移。不过，SHT_NOBITS 类型的节区不占用文件的空间，因此其 sh_offset 成员给出的是其概念性的偏移。
sh_size	此成员给出节区的长度（字节数）。除非节区的类型是 SHT_NOBITS，否则节区占用文件中的 sh_size 字节。类型为

	SHT_NOBITS 的节区长度可能非零，不过却不占用文件中的空间。
sh_link	此成员给出节区头部表索引链接。其具体的解释依赖于节区类型。
sh_info	此成员给出附加信息，其解释依赖于节区类型。
sh_addralign	某些节区带有地址对齐约束。例如，如果一个节区保存一个 doubleword，那么系统必须保证整个节区能够按双字对齐。sh_addr 对 sh_addralign 取模，结果必须为 0。目前仅允许取值为 0 和 2 的幂次数。数值 0 和 1 表示节区没有对齐约束。
sh_entsize	某些节区中包含固定大小的项目，如符号表。对于这类节区，此成员给出每个表项的长度字节数。 如果节区中并不包含固定长度表项的表格，此成员取值为 0。

索引为零（SHN_UNDEF）的节区头部也是存在的，尽管此索引标记的是未定义的节区引用。这个节区的内容固定如下：

表 7 SHN_UNDEF(0)节区的内容

字段名称	取值	说明
sh_name	0	无名称
sh_type	SHT_NULL	非活动
sh_flags	0	无标志
sh_addr	0	无地址
sh_offset	0	无文件偏移
sh_size	0	无尺寸大小
sh_link	SHN_UNDEF	无链接信息
sh_info	0	无辅助信息
sh_addralign	0	无对齐要求
sh_entsize	0	无表项

3.4.2.1 节区类型—sh_type 字段

节区类型定义如表 8：

表 8 节区类型定义

名称	取值	说明
SHT_NULL	0	此值标志节区头部是非活动的，没有对应的节区。此节区头部中的其他成员取值无意义。
SHT_PROGBITS	1	此节区包含程序定义的信息，其格式和含义都由程序来解释。
SHT_SYMTAB	2	此节区包含一个符号表。目前目标文件对每种类型的节区都只能包含一个，不过这个限制将来可能发生变化。 一般，SHT_SYMTAB 节区提供用于链接编辑（指 ld 而言）的符号，尽管也可用来实现动态链接。
SHT_STRTAB	3	此节区包含字符串表。目标文件可能包含多个字符串表节区。
SHT_RELA	4	此节区包含重定位表项，其中可能会有补齐内容（addend），例如 32 位目标文件中的 Elf32_Rela 类型。目标文件可能拥有多个重定位节区。

SHT_HASH	5	此节区包含符号哈希表。所有参与动态链接的目标都必须包含一个符号哈希表。目前，一个目标文件只能包含一个哈希表，不过此限制将来可能会解除。
SHT_DYNAMIC	6	此节区包含动态链接的信息。目前一个目标文件中只能包含一个动态节区，将来可能会取消这一限制。
SHT_NOTE	7	此节区包含以某种方式来标记文件的信息。
SHT_NOBITS	8	这种类型的节区不占用文件中的空间，其他方面和 SHT_PROGBITS 相似。尽管此节区不包含任何字节，成员 sh_offset 中还是会包含概念性的文件偏移
SHT_REL	9	此节区包含重定位表项，其中没有补齐（addends），例如 32 位目标文件中的 Elf32_rel 类型。目标文件中可以拥有多个重定位节区。
SHT_SHLIB	10	此节区被保留，不过其语义是未规定的。包含此类型节区的程序与 ABI 不兼容。
SHT_DYNSYM	11	作为一个完整的符号表，它可能包含很多对动态链接而言不必要的符号。因此，目标文件也可以包含一个 SHT_DYNSYM 节区，其中保存动态链接符号的一个最小集合，以节省空间。
SHT_LOPROC	0X70000000	这一段（包括两个边界），是保留给处理器专用语义的。
SHT_HIPROC	0X7FFFFFFF	
SHT_LOUSER	0X80000000	此值给出保留给应用程序的索引下界。
SHT_HIUSER	0X8FFFFFFF	此值给出保留给应用程序的索引上界。

其它的节区类型是保留的。

3.4.2.2 sh_flags 字段

sh_flags 字段定义了一个节区中包含的内容是否可以修改、是否可以执行等信息。如果一个标志位被设置，则该位取值为 1。未定义的各位都设置为 0。

表 9 节区头部的 sh_flags 字段取值

名称	取值
SHF_WRITE	0x1
SHF_ALLOC	0x2
SHF_EXECINSTR	0x4
SHF_MASKPROC	0xF0000000

其中已经定义了的各位含义如下：

- SHF_WRITE: 节区包含进程执行过程中将可写的数据。
- SHF_ALLOC: 此节区在进程执行过程中占用内存。某些控制节区并不出现于目标文件的内存映像中，对于那些节区，此位应设置为 0。
- SHF_EXECINSTR: 节区包含可执行的机器指令。
- SHF_MASKPROC: 所有包含于此掩码中的四位都用于处理器专用的语义。

3.4.2.3 sh_link 和 sh_info 字段

根据节区类型的不同，sh_link 和 sh_info 的具体含义也有所不同：

表 10 sh_link 和 sh_info 字段解释

sh_type	sh_link	sh_info
SHT_DYNAMIC	此节区中条目所用到的字符串表格的节区头部索引	0
SHT_HASH	此哈希表所适用的符号表的节区头部索引	0
SHT_REL SHT_RELA	相关符号表的节区头部索引	重定位所适用的节区的节区头部索引
SHT_SYMTAB SHT_DYNSYM	相关联的字符串表的节区头部索引	最后一个局部符号（绑定 STB_LOCAL）的符号表索引值加一
其它	SHN_UNDEF	0

3.4.3 特殊节区

很多节区中包含了程序和控制信息。下面的表中给出了系统使用的节区，以及它们的类型和属性。

表 11 常见特殊节区

名称	类型	属性	含义
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE	包含将出现在程序的内存映像中的为初始化数据。根据定义，当程序开始执行，系统将把这些数据初始化为 0。此节区不占用文件空间。
.comment	SHT_PROGBITS	(无)	包含版本控制信息。
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE	这些节区包含初始化了的数据，将出现在程序的内存映像中。
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE	
.debug	SHT_PROGBITS	(无)	此节区包含用于符号调试的信息。
.dynamic	SHT_DYNAMIC		此节区包含动态链接信息。节区的属性将包含 SHF_ALLOC 位。是否 SHF_WRITE 位被设置取决于处理器。
.dynstr	SHT_STRTAB	SHF_ALLOC	此节区包含用于动态链接的字符串，大多数情况下这些字符串代表了与符号表项相关的名称。
.dynsym	SHT_DYNSYM	SHF_ALLOC	此节区包含了动态链接符号表。
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR	此节区包含了可执行的指令，是进程终止代码的一部分。程序正常退出时，系统将安排执行这里的代码。
.got	SHT_PROGBITS		此节区包含全局偏移表。

.hash	SHT_HASH	SHF_ALLOC	此节区包含了一个符号哈希表。
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR	此节区包含了可执行指令，是进程初始化代码的一部分。当程序开始执行时，系统要在开始调用主程序入口之前（通常指 C 语言的 main 函数）执行这些代码。
.interp	SHT_PROGBITS		此节区包含程序解释器的路径名。如果程序包含一个可加载的段，段中包含此节区，那么节区的属性将包含 SHF_ALLOC 位，否则该位为 0。
.line	SHT_PROGBITS	（无）	此节区包含符号调试的行号信息，其中描述了源程序与机器指令之间的对应关系。其内容是未定义的。
.note	SHT_NOTE	（无）	此节区中包含注释信息，有独立的格式。
.plt	SHT_PROGBITS		此节区包含过程链接表（procedure linkage table）。
.rel*name*	SHT_REL		这些节区中包含了重定位信息。如果文件中包含可加载的段，段中有重定位内容，节区的属性将包含 SHF_ALLOC 位，否则该位置 0。传统上 name 根据重定位所适用的节区给定。例如 .text 节区的重定位节区名字将是：.rel.text 或者 .rela.text。
.rela*name*	SHT_RELA		
.rodata	SHT_PROGBITS	SHF_ALLOC	这些节区包含只读数据，这些数据通常参与进程映像的不可写段。
.rodata1	SHT_PROGBITS	SHF_ALLOC	
.shstrtab	SHT_STRTAB		此节区包含节区名称。
.strtab	SHT_STRTAB		此节区包含字符串，通常是代表与符号表项相关的名称。如果文件拥有一个可加载的段，段中包含符号串表，节区的属性将包含 SHF_ALLOC 位，否则该位为 0。
.symtab	SHT_SYMTAB		此节区包含一个符号表。如果文件中包含一个可加载的段，并且该段中包含符号表，那么节区的属性中包含SHF_ALLOC 位，否则该位置为 0。
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR	此节区包含程序的可执行指令。

在分析这些节区的时候，需要注意如下事项：

- 以“.”开头的节区名称是系统保留的。应用程序可以使用没有前缀的节区名称，以避免与系统节区冲突。
- 目标文件格式允许人们定义不在上述列表中的节区。
- 目标文件中也可以包含多个名字相同的节区。
- 保留给处理器体系结构的节区名称一般构成为：处理器体系结构名称简写 + 节区名称。
- 处理器名称应该与 e_machine 中使用的名称相同。例如 .FOO.psect 街区是由 FOO 体系结构定义的 psect 节区。

另外，有些编译器对如上节区进行了扩展，这些已存在的扩展都使用约定俗成的名称，如：

- .sdata
- .tdesc
- .sbss
- .lit4
- .lit8
- .reginfo
- .gptab
- .liblist
- .conflict
-

3.5 字符串表（String Table）

字符串表节区包含以 NULL（ASCII 码 0）结尾的字符序列，通常称为字符串。ELF 目标文件通常使用字符串来表示符号和节区名称。对字符串的引用通常以字符串在字符串表中的下标给出。

一般，第一个字节（索引为 0）定义为一个空字符串。类似的，字符串表的最后一个字节也定义为 NULL，以确保所有的字符串都以 NULL 结尾。索引为 0 的字符串在不同的上下文中可以表示无名或者名字为 NULL 的字符串。

允许存在空的字符串表节区，其节区头部的 sh_size 成员应该为 0。对空的字符串表而言，非 0 的索引值是非法的。

例如：对于各个节区而言，节区头部的 sh_name 成员包含其对应的节区头部字符串表节区的索引，此节区由 ELF 头的 e_shstrndx 成员给出。下图给出了包含 25 个字节的一个字符串表，以及与不同索引相关的字符串。

表 12 字符串表示例

索引	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

表 12 中包含的字符串如下：

索引	字符串
0	(无)
1	name.
7	Variable
11	able
16	able
24	(空字符串)

在使用、分析字符串表时，要注意以下几点：

- 字符串表索引可以引用节区中任意字节。
- 字符串可以出现多次
- 可以存在对子字符串的引用
- 同一个字符串可以被引用多次。
- 字符串表中也可以存在未引用的字符串。

3.6 符号表（Symbol Table）

目标文件的符号表中包含用来定位、重定位程序中符号定义和引用的信息。符号表索引是对此数组的索引。索引 0 表示表中的第一表项，同时也作为未定义符号的索引。

符号表项的格式如下：

图 4 符号表项格式定义

```
typedef struct {
    Elf32_Word  st_name;
    Elf32_Addr  st_value;
    Elf32_Word  st_size;
    unsigned char   st_info;
    unsigned char   st_other;
    Elf32_Half  st_shndx;
 } Elf32_sym;
```

其中各个字段的含义说明如表 13：

表 13 符号表项字段

字段	说明
st_name	包含目标文件符号字符串表的索引，其中包含符号名的字符串表示。如果该值非 0，则它表示了给出符号名的字符串表索引，否则符号表项没有名称。 注：外部 C 符号在 C 语言和目标文件的符号表中具有相同的名称。
st_value	此成员给出相关联的符号的取值。依赖于具体的上下文，它可能是一个绝对值、一个地址等等。
st_size	很多符号具有相关的尺寸大小。例如一个数据对象的大小是对象中包含的字节数。如果符号没有大小或者大小未知，则此成员为 0。
st_info	此成员给出符号的类型和绑定属性。下面给出若干取值和含义的绑定关系。
st_other	该成员当前包含 0，其含义没有定义。
st_shndx	每个符号表项都以和其他节区间的关系的方式给出定义。此成员给出相关的节区头部表索引。某些索引具有特殊含义。

3.6.1 关于 st_info 的说明

st_info 中包含符号类型和绑定信息，操纵方式如：

图 5 符号表项的 st_info 字段合成

```
#define ELF32_ST_BIND(i)    ((i)>>4)
#define ELF32_ST_TYPE(i)    ((i)&0xf)
#define ELF32_ST_INFO(b, t) (((b)<<4) + ((t)&0xf))
```

从中可以看出，st_info 的低四位表示符号绑定，用于确定链接可见性和行为。具体的绑定类型如：

表 14 符号的绑定类型

名称	取值	说明
STB_LOCAL	0	局部符号在包含该符号定义的目标文件以外不可见。相同名称的局部符号可以存在于多个文件中，互不影响。
STB_GLOBAL	1	全局符号对所有将组合的目标文件都是可见的。一个文件中对某个全局符号的定义将满足另一个文件对相同全局符号的未定义引用。
STB_WEAK	2	弱符号与全局符号类似，不过他们的定义优先级比较低。
STB_LOPROC	13	处于这个范围的取值是保留给处理器专用语义的。
STB_HIPROC	15	

全局符号与弱符号之间的区别主要有两点：

(1). 当链接编辑器组合若干可重定位的目标文件时，不允许对同名的 STB_GLOBAL 符号给出多个定义。
另一方面如果一个已定义的全局符号已经存在，出现一个同名的弱符号并不会产生错误。链接编辑器尽关心全局符号，忽略弱符号。
类似地，如果一个公共符号（符号的 st_shndx 中包含 SHN_COMMON），那么具有相同名称的弱符号出现也不会导致错误。链接编辑器会采纳公共定义，而忽略弱定义。

(2). 当链接编辑器搜索归档库（archive libraries）时，会提取那些包含未定义全局符号的档案成员。成员的定义可以是全局符号，也可以是弱符号。
连接编辑器不会提取档案成员来满足未定义的弱符号。
未能解析的弱符号取值为 0。

在每个符号表中，所有具有 STB_LOCAL 绑定的符号都优先于弱符号和全局符号。符号表节区中的 sh_info 头部成员包含第一个非局部符号的符号表索引。

3.6.2 符号类型

符号类型（ELF32_ST_TYPE）定义如下：

表 15 符号类型

名称	取值	说明
STT_NOTYPE	0	符号的类型没有指定
STT_OBJECT	1	符号与某个数据对象相关，比如一个变量、数组等等
STT_FUNC	2	符号与某个函数或者其他可执行代码相关
STT_SECTION	3	符号与某个节区相关。这种类型的符号表项主要用于重定位，通常具有 STB_LOCAL 绑定。
STT_FILE	4	传统上，符号的名称给出了与目标文件相关的源文件的名称。文件符号具有 STB_LOCAL 绑定，其节区索引是

		SHN_ABS，并且它优先于文件的其他 STB_LOCAL 符号（如果有的话）
STT_LOPROC	13	此范围的符号类型值保留给处理器专用语义用途。
STT_HIPROC	15	

在共享目标文件中的函数符号（类型为 STT_FUNC）具有特别的重要性。当其他目标文件引用了来自某个共享目标中的函数时，链接编辑器自动为所引用的符号创建过程链接表项。类型不是 STT_FUNC 的共享目标符号不会自动通过过程链接表进行引用。

如果一个符号的取值引用了某个节区中的特定位置，那么它的节区索引成员（st_shndx）包含了其在节区头部表中的索引。当节区在重定位过程中被移动时，符号的取值也会随之变化，对符号的引用始终会“指向”程序中的相同位置。

3.6.3 特殊的节区索引

某些特殊的节区索引具有不同的语义：

- SHN_ABS: 符号具有绝对取值，不会因为重定位而发生变化。
- SHN_COMMON: 符号标注了一个尚未分配的公共块。符号的取值给出了对齐约束，与节区的 sh_addralign 成员类似。就是说，链接编辑器将为符号分配存储空间，地址位于 st_value 的倍数处。符号的大小给出了所需要的字节数。
- SHN_UNDEF: 此节区表索引值意味着符号没有定义。当链接编辑器将此目标文件与其他定义了该符号的目标文件进行组合时，此文件中对该符号的引用将被链接到实际定义的位置。

3.6.4 STN_UNDEF 符号

如上所述，符号表中下标为 0（STN_UNDEF）的表项被保留。其中包含如下数值：

表 16 STN_UNDEF 符号

名称	取值	说明
st_name	0	无名称
st_value	0	0 值
st_size	0	无大小
st_info	0	无类型，局部绑定
st_other	0	无附加信息
st_shndx	0	无节区

3.6.5 符号取值

不同的目标文件类型中符号表项对 st_value 成员具有不同的解释：

(1). 在可重定位文件中，st_value 中遵从了节区索引为 SHN_COMMON 的符号

的对齐约束。

(2). 在可重定位的文件中，st_value 中包含已定义符号的节区偏移。就是说，st_value 是从 st_shndx 所标识的节区头部开始计算，到符号位置的偏移。

(3). 在可执行和共享目标文件中，st_value 包含一个虚地址。为了使得这些文件的符号对动态链接器更有用，节区偏移（针对文 件的解释）让位于虚拟地址（针对内存的解释），因为这时与节区号无关。

尽管符号表取值在不同的目标文件中具有相似的含义，适当的程序可以采取高效的数据访问方式。

3.7 重定位信息

重定位是将符号引用与符号定义进行连接的过程。例如，当程序调用了一个函数时，相关的调用指令必须把控制传输到适当的目标执行地址。

3.7.1 重定位表项

可重定位文件必须包含如何修改其节区内容的信息，从而允许可执行文件和共享目标文件保存进程的程序映像的正确信息。重定位表项就是这样一些数据。

重定位表项的格式如图 6：

图 6 重定位表项的格式

```
typedef struct {
    Elf32_Addr r_offset;
    Elf32_Word r_info;
} Elf32_Rel;

typedef struct {
    Elf32_Addr r_offset;
    Elf32_Word r_info;
    Elf32_Word r_addend;
} Elf32_Rela;
```

其中，各个字段的说明如下表：

表 17 重定位表项字段说明

成员	说明
r_offset	此成员给出了重定位动作所适用的位置。对于一个可重定位文件而言，此值是从节区头部开始到将被重定位影响的存储单位之间的字节偏移。对于可执行文件或者共享目标文件而言，其取值是被重定位影响到的存储单元的虚拟地址。
r_info	此成员给出要进行重定位的符号表索引，以及将实施的重定位类型。例如一个调用指令的重定位项将包含被调用函数的符号表索引。如果索引是 STN_UNDEF，那么重定位使用 0 作为“符号值”。重定位类

	型是和处理器相关的。当程序代码引用一个重定位项的重定位类型或者符号表索引，则表示对表项的 r_info 成员应用 ELF32_R_TYPE 或者 ELF32_R_SYM 的结果。 #define ELF32_R_SYM(i) ((i)>>8) #define ELF32_R_TYPE(i) ((unsigned char)(i)) #define ELF32_R_INFO(s, t) (((s)<<8) + (unsigned char)(t))
r_addend	此成员给出一个常量补齐，用来计算将被填充到可重定位字段的数值。

如上所述，只有 Elf32_Rela 项目可以明确包含补齐信息。类型为 Elf32_Rel 的表项在将被修改的位置保存隐式的补齐信息。 依赖于处理器体系结构，各种形式都可能存在，甚至是必需的。因此，对特定机器的实现可以仅使用一种形式，也可以根据上下文使用不同的形式。

重定位节区会引用两个其它节区：符号表、要修改的节区。节区头部的 sh_info 和 sh_link 成员给出这些关系。不同目标文件的重定位表项对 r_offset 成员具有略微不同的解释。

(1). 在可重定位文件中，r_offset 中包含节区偏移。就是说重定位节区自身描述了如何修改文件中的其他节区；重定位偏移 指定了被修改节区中的一个存储单元。

(2). 在可执行文件和共享的目标文件中，r_offset 中包含一个虚拟地址。为了使得这些文件的重定位表项对动态链接器更为有用，节区偏移（针对文件的解释）让位于虚地址（针对内存的解释）。

尽管对 r_offset 的解释会有少许不同，重定位类型的含义始终不变。

3.7.2 重定位类型

重定位表项描述如何修改后面的指令和数据字段。一般，共享目标文件在创建时，其基本虚拟地址是 0，不过执行地址将随着动态加载而发生变化。

重定位的过程，按照如下标记：

- A 用来计算可重定位字段的取值的补齐。
- B 共享目标在执行过程中被加载到内存中的位置（基地址）。
- G 在执行过程中，重定位项的符号的地址所处的位置 —— 全局偏移表的索引。
- GOT 全局偏移表（GOT）的地址。
- L 某个符号的过程链接表项的位置（节区偏移/地址）。过程链接表项把函数调用重定位到正确的目标位置。链接编辑器构造初始的过程链接表，动态链接器在执行过程中修改这些项目。
- P 存储单位被重定位（用 r_offset 计算）到的位置（节区偏移或者地址）。
- S 其索引位于重定位项中的符号的取值。

重定位项的 r_offset 取值给定受影响的存储单位的第一个字节的偏移或者虚拟地址。重定位类型给出那些位需要修改以及如何计算它们的取值。

SYSTEM V 仅使用 Elf32_Rel 重定位表项，在被重定位的字段中包含补齐量。补齐量和计算结果始终采用相同的字节顺序。

X86 体系结构下常见的重定位类型：

表 18 x86 体系结构下常见的重定位类型

名称	数值	字段	计算	说明
R_386_NONE	0	(无)	(无)	
R_386_32	1	word32	S+A	
R_386_PC32	2	word32	S+A-P	
R_386_GOT32	3	word32	G+A-P	此重定位类型计算从全局偏移表基址到符号的全局偏移表项之间的距离。它会通知连接编辑器构造一个全局偏移表。
R_386_PLT32	4	word32	L+A-P	此重定位类型计算符号的过程链接表项的地质，并通知链接编辑器构造一个过程链接表。
R_386_COPY	5	(无)	(无)	链接编辑器创建这种重定位类型的目的是支持动态链接。其偏移量成员引用某个可写段中的某个位置。符号表索引规定符号应该既存在于当前目标文件中，也存在于某个共享目标中。在执行过程中，动态链接器把与共享目标的符号相关的数据复制到由偏移给出的位置。
R_386_GLOB_DAT	6	word32	S	此重定位类型用来把某个全局偏移表项设置为给定符号的地址。这种特殊的重定位类型允许确定符号与全局偏移表项之间的关系。
R_386_JMP_SLOT	7	word32	S	链接编辑器创建这种重定位类型主要是为了支持动态链接。其偏移地址成员给出过程链接表项的位置。动态链接器修改过程链接表项的内容，把控制传输给指定符号的地址。
R_386_RELATIVE	8	word32	B+A	链接编辑器创建这种重定位类型是为了支持动态链接。其偏移地址成员给出共享目标中的一个位置，在该位置包含了代表相对地址的一个数值。动态链接器通过把共享目标被加载到的虚地址和相对地址相加，计算对应的虚地址。这种类型的重定位项必须设置符号表索引为 0。
R_386_GOTOFF	9	word32	S+A-GOT	这种重定位类型会计算符号取值与全局偏移表地址间的差。并通知链接编辑器创建一个全局偏移表。
R_386_GOTPC	10	word32	GOT+A-P	此重定位类型与 R_386_PC32 类似，只不过它在计算时采用全局偏移表的地址。在此重定位项中引用的符号通常是_GLOBAL_OFFSET_TABLE_，这种类型也会暗示连接编辑器构造全局偏移表。

3.8 程序加载和动态链接

实现程序加载和动态链接的主要技术有：

- 程序头部（Program Header）：描述与程序执行直接相关的目标文件结构信息。用来在文件中定位各个段的映像。同时包含其他一些用来为程序创建进程映像所必需的信息。
- 程序加载：给定一个目标文件，系统加载该文件到内存中，启动程序执行。
- 动态链接：系统加载了程序以后，必须通过解析构成进程的目标文件之间的符号引用，以便完整地构造进程映像。

3.8.1 程序头部（Program Header）

可执行文件或者共享目标文件的程序头部是一个结构数组，每个结构描述了一个段或者系统准备程序执行所必需的其它信息。目标文件的“段”包含一个或者多个“节区”，也就是“段内容（Segment Contents）”。程序头部仅对于可执行文件和共享目标文件有意义。

可执行目标文件在 ELF 头部的 e_phentsize 和 e_phnum 成员中给出其自身程序头部的大小。程序头部的数据结构如下图：

```
typedef struct {
    Elf32_Word p_type;
    Elf32_Off  p_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    Elf32_Word p_filesz;
    Elf32_Word p_memsz;
    Elf32_Word p_flags;
    Elf32_Word p_align;
} Elf32_phdr;
```

图 7 程序头部数据结构

其中各个字段说明如下：

- p_type　此数组元素描述的段的类型，或者如何解释此数组元素的信息。具体如表 19。
- p_offset　此成员给出从文件头到该段第一个字节的偏移。
- p_vaddr　此成员给出段的第一个字节将被放到内存中的虚拟地址。
- p_paddr　此成员仅用于与物理地址相关的系统中。因为 System V 忽略所有应用程序的物理地址信息，此字段对与可执行文件和共享目标文件而言具体内容是未指定的。
- p_filesz　此成员给出段在文件映像中所占的字节数。可以为 0。
- p_memsz　此成员给出段在内存映像中占用的字节数。可以为 0。
- p_flags　此成员给出与段相关的标志。
- p_align　可加载的进程段的 p_vaddr 和 p_offset 取值必须合适，相对

于对页面大小的取模而言。此成员给出段在文件中和内存中如何对齐。数值 0 和 1 表示不需要对齐。否则 p_align 应该是个正整数，并且是 2 的幂次数，p_vaddr 和 p_offset 对 p_align 取模后应该相等。

3.8.1.1 段类型

可执行 ELF 目标文件中的段类型如表 19 所示：

表 19 段类型

名字	取值	说明
PT_NULL	0	此数组元素未用。结构中其他成员都是未定义的。
PT_LOAD	1	此数组元素给出一个可加载的段，段的大小由 p_filesz 和 p_memsz 描述。文件中的字节被映射到内存段开始处。如果 p_memsz 大于 p_filesz，“剩余”的字节要清零。p_filesz 不能大于 p_memsz。可加载的段在程序头部表格中根据 p_vaddr 成员按升序排列。
PT_DYNAMIC	2	数组元素给出动态链接信息。
PT_INTERP	3	数组元素给出一个 NULL 结尾的字符串的位置和长度，该字符串将被当作解释器调用。这种段类型仅对与可执行文件有意义（尽管也可能在共享目标文件上发生）。在一个文件中不能出现一次以上。如果存在这种类型的段，它必须在所有可加载段项目的前面。
PT_NOTE	4	此数组元素给出附加信息的位置和大小。
PT_SHLIB	5	此段类型被保留，不过语义未指定。包含这种类型的段的程序与 ABI 不符。
PT_PHDR	6	此类型的数组元素如果存在，则给出了程序头部表自身的大小和位置，既包括在文件中也包括在内存中的信息。此类型的段在文件中不能出现一次以上。并且只有程序头部表是程序的内存映像的一部分时才起作用。如果存在此类型段，则必须在所有可加载段项目的前面。
PT_LOPROC	0x70000000	此范围的类型保留给处理器专用语义。
PT_HIPROC	0x7fffffff	

3.8.1.2 基地址（Base Address）

基地址用来对程序的内存映像进行重定位。可执行文件或者共享目标文件的基地址是在执行过程中从三个数值计算的：

- 内存加载地址
- 最大页面大小
- 程序的可加载段的最低虚地址。

程序头部中的虚拟地址可能不能代表程序内存映像的实际虚地址。要计算基地址，首先要确定与 PT_LOAD 段的最低 p_vaddr 值相关的内存地址。通过对内存地址向最接近的最大页面大小截断，就可以得到基地址。根据要加载到内存中的文件的类型，内存地址可能与 p_vaddr 相同也可能不同。

如前所述，“.bss”节区的类型为 SHT_NOBITS。尽管它在文件中不占据空间，却会占据段的内存映像的空间。通常，这些未初始化的数据位于段的末尾，所以 p_memsz 会

比 p_filesz 大。

3.8.1.3 注释节区（Note Section）

类型为 SHT_NOTE 的节区和类型为 PT_NOTE 的节区可以用作特殊信息的存放。节区和程序头部中的注释信息可以有任意多个条目，每个条目都是一个按目标处理器格式给出的 4 字节字的数组。

例如：

namesz
descsz
type
name
...
desc
...

图 8 注释节区示例

其中：

- **namesz** 和 **name** 注释信息的 name 部分前 namesz 字节包含一个 NULL 结尾的字符串，表示该项的属主或者发起者。没有正式的机制来避免名字冲突。如果没有名字，namesz 中包含 0。如果需要的话，可以用补零来确保描述符以 4 字节对齐。这类补齐并不包含在 namesz 中。
- **descsz** 和 **desc** desc 中的前 descsz 字节包含注释信息的描述。ABI 对此没有作出约束。如果需要，可以用补零来确保 4 字节边界对齐。补零的字节数不计算在 descsz 中。
- **type** 此 word 给出描述符的解释。每个发起者都要负责对自己的类型进行控制；同一类型值可以包含多种不同解释。因此程序必须能够识别名称和类型，才能理解描述符。类型取值必须非负。ABI 并不定义描述符的含义[3]。

3.8.2 程序加载

进程除非在执行过程中引用到相应的逻辑页面，否则不会请求真正的物理页面。进程通常会包含很多未引用的页面，因此，延迟物理读操作通常会避免这类费力不讨好的事情发生，从而提高系统性能。要想实际获得这种效率，可执行文件和共享目标文件必须具有这样的段：其文件偏移和虚拟地址对页面大小取模后余数相同。

例如，可执行文件布局如图 9：

文件偏移 文件 虚拟地址

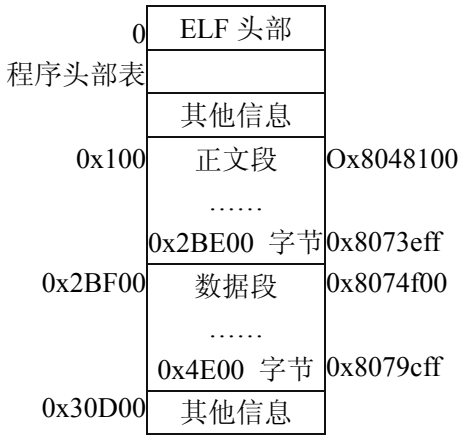

图 9 可执行文件布局示例

该可执行文件中程序头部段如所示：

表 20 可执行文件程序头部段示例

成员	正文段	数据段
p_type	PT_LOAD	PT_LOAD
p_offset	0X100	0x2bf00
p_vaddr	0x8048100	0x8074f00
p_paddr	未指定	未指定
p_filesz	0x2BE00	0x4e00
p_memsz	0x2BE00	0x5e24
p_flags	PF_R + PF_X	PF_R + PF_W + PF_X
p_align	0x1000	0x1000

在这个例子中，至多四个文件页面包含非纯粹的正文或者数据。

(1). 第一个页面中包含 ELF 头部、程序头部表、以及其它信息
(2). 最后一个页面包含数据开始部分的一个副本
(3). 第一数据页面包含正文段的末尾部分
(4). 最后一个数据页面可能包含与运行进程无关的文件信息

不过系统对这些页面一般会做两次映射，以保证每个段的内存访问许可是相同的。数据段的末尾需要对未初始化数据进行特殊处理，系统应该将这些初始化为 0。

可执行文件与共享目标文件之间的段加载之间有一点不同。可执行文件的段通常包含绝对代码，为了能够让进程正确执行，所使用的段必须是构造可执行文件时所使用的虚拟地址。因此系统会使用 p_vaddr 作为虚拟地址。

另外，共享目标文件的段通常包含与位置无关的代码。这使得段的虚拟地址在不同的进程中不同，但不影响执行行为。尽管系统为每个进程选择独立的虚拟地址，仍能维持段的相对位置。因为位置独立的代码在段与段之间使用相对寻址，内存虚地址之间的差异必须与文件中虚拟地址之间的差异相匹配。

下表给出共享目标文件的针对不同进程的一种虚拟地址指定方案，说明了这种重定位问题。

表 21 虚拟地址指定方案示例

源	正文	数据	基地址
文件	0x200	0x2a400	0x0
进程 1	0x80000200	0x8002a400	0x80000000
进程 2	0x80081200	0x800ab400	0x80081000
进程 3	0x900c0200	0x900ea400	0x900c0000
进程 4	0x900c6200	0x900f0400	0x900c6000

3.8.3 动态链接

3.8.3.1 程序解释器

可执行文件可以包含 PT_INTERP 程序头部元素。在 exec() 期间，系统从 PT_INTERP 段中检索路径名，并从解释器文件的段创建初始的进程映像。也就是说，系统并不使用原来可执行文件的段映像，而是为解释器构造一个内存映像。接下来是解释器从系统接收控制，为应用程序提供执行环境。

解释器可以有两种方式接受控制。

- 接受一个文件描述符，读取可执行文件并将其映射到内存中
- 根据可执行文件的格式，系统可能把可执行文件加载到内存中，而不是为解释器提供一个已经打开的文件描述符。

解释器可以是一个可执行文件，也可以是一个共享目标文件。

共享目标文件被加载到内存中时，其地址可能在各个进程中呈现不同的取值。系统在 mmap 以及相关服务所使用的动态段区域创建共享目标文件的段。因此，共享目标解释器通常不会与原来的可执行文件的原始段地址发生冲突。

可执行文件被加载到内存中固定地址，系统使用来自其程序头部表的虚拟地址创建各个段。因此，可执行文件解释器的虚拟地址可能会与原来的可执行文件的虚拟地址发生冲突。解释器要负责解决这种冲突。

3.8.3.2 动态加载程序

在构造使用动态链接技术的可执行文件时，连接编辑器向可执行文件中添加一个类型为 PT_INTERP 的程序头部元素，告诉系统要把动态链接器激活，作为程序解释器。系统所提供的动态链接器的位置是和处理器相关的。

Exec() 和动态链接器合作，为程序创建进程映像，其中包括以下动作：

(1). 将可执行文件的内存段添加到进程映像中；
(2). 把共享目标内存段添加到进程映像中；
(3). 为可执行文件和它的共享目标执行重定位操作；
(4). 关闭用来读入可执行文件的文件描述符，如果动态链接程序收到过这样的文件描述符的话；
(5). 将控制转交给程序，使得程序好像从 exec 直接得到控制。

链接编辑器也会构造很多数据来协助动态链接器处理可执行文件和共享目标文件。这些数据包含在可加载段中，在执行过程中可用。如：

- 类型为 SHT_DYNAMIC 的 .dynamic 节区包含很多数据。位于节区头部的结构保存了其他动态链接信息的地址。
- 类型为 SHT_HASH 的 .hash 节区包含符号哈希表。
- 类型为 SHT_PROGBITS 的 .got 和 .plt 节区包含两个不同的表：全局偏移表和过程链接表。

因为任何符合 ABI 规范的程序都要从共享目标库中导入基本的系统服务，动态链接器会参与每个符合 ABI 规范的程序的执行。

3.8.3.3 动态节区

如果一个目标文件参与动态链接，它的程序头部表将包含类型为 PT_DYNAMIC 的元素。此“段”包含.dynamic 节区。该节区采用一个特殊符号_DYNAMIC 来标记，其中包含如下结构的数组。

图 10 动态节区符号结构

```
typedef struct {
    Elf32_Sword d_tag;
    union {
        Elf32_Word  d_val;
        Elf32_Addr  d_ptr;
    } d_un;
} Elf32_Dyn;

extern Elf32_Dyn  _DYNAMIC[];
```

对每个这种类型的对象，d_tag 控制 d_un 的解释含义：

- d_val 此 Elf32_Word 对象表示一个整数值，可以有多种解释。
- d_ptr 此 Elf32_Addr 对象代表程序的虚拟地址。如前所述，文件的虚拟地址可能与执行过程中的内存虚地址不匹配。在解释包含于动态结构中的地址时，动态链接程序基于原来文件值和内存基地址计算实际地址。为了保持一致性，文件中不包含用来“纠正”动态结构中重定位项地址的重定位项目。

下面的表格总结了可执行文件和共享目标文件对标志的要求。如果标志被标记为“必需”，那么符合 ABI 规范的文件的动态链接数组必须包含一个该类型表项。“可选”意味着该标志可以出现，但不是必需的。

表 22 动态项标志说明

名称	数值	d_un	可执行	共享目标	说明
DT_NULL	0	忽略	必需	必需	标记为 DT_NULL 的项目标注了整个 _DYNAMIC 数组的末端。
DT_NEEDED	1	d_val	可选	可选	此元素包含一个 NULL 结尾的字符串的字符串表偏移，该字符串给出某个需要的库的名称。所使用的字符串表根据 DT_STRTAB 项目中记录的内容确定。所谓的偏移即是指在该表中的下标。动态数组

					中可以包含多个这种类型的条目。这些条目的相对顺序很重要，尽管他们与其他类型条目间的顺序没有很大关系。
DT_PLTRELSZ	2	d_val	可选	可选	此元素给出了与过程链接表（PLT）相关联的重定位项的总计大小（按字节）。如果存在 DT_JMPREL 类型的条目，必须有与之配合的 DT_PLTRELSZ 条目。
DT_PLTGOT	3	d_ptr	可选	可选	此元素给出一个与过程链接表（PLT）与/或全局偏移表相关联的一个地址。
DT_HASH	4	d_ptr	必需	必需	此元素包含符号哈希表的地址。此哈希表指的是被 DT_SYMTAB 元素引用的符号表。
DT_STRTAB	5	d_ptr	必需	必需	此元素包含字符串表的地址，符号名、库名、和其他字符串都包含在此表中。
DT_SYMTAB	6	d_ptr	必需	必需	此元素包含符号表的地址。对 32 位的文件而言，这个符号表中的条目是 Elf32_Sym 类型。
DT_RELA	7	d_ptr	必需	可选	此元素包含重定位表的地址。此表中的元素包含显式的补齐，例如 32 位文件中的 Elf32_Rela。目标文件可能有多个重定位节区。在为可执行文件或者共享目标文件构造重定位表时，连接编辑器将这些节区连接起来，形成一个表格。尽管在目标文件中这些节区保持相互独立，动态链接器所看到的仍然是一个表。在动态链接器为可执行文件创建进程映像或者向一个进程映像中添加某个共享目标时，要读取重定位表，并执行相关的动作。如果此元素存在，动态结构必须也包含 DT_RELASZ 和 DT_RELAENT 元素。如果对于某个文件来说，重定位能力是必需的，那么 DT_RELA 或者 DT_REL 都可能存在（二者都是允许存在但不要求存在的）。
DT_RELASZ	8	d_val	必需	可选	此元素包含 DT_RELA 重定位表的大小（按字节数计算）。
DT_RELAENT	9	d_val	必需	可选	此元素包含 DT_RELA 重定位项的大小（按字节计算）。
DT_STRSZ	10	d_val	必需	必需	此元素给出字符串表的大小，按字节数计算。
DT_SYMENT	11	d_val	必需	必需	此元素给出符号表项的大小，按字节数计算。
DT_INIT	12	d_ptr	可选	可选	此元素包含初始化函数的地址。
DT_FINI	13	d_ptr	可选	可选	此元素包含结束函数（Termination Function）的地址。
DT_SONAME	14	d_val	忽略	可选	此元素给出一个 NULL 结尾的字符串的字符串表偏移，字符串是某个共享目标的名称。该偏移实际上是 DT_STRTAB 项目所记录的表格的索引。
DT_RPATH	15	d_val	可选	忽略	此元素包含 NULL 结尾的字符串的字符串表偏移，字符串是搜索库时使用的搜索路径。该偏移实际上是 DT_STRTAB 项目所记录的表格的索引。
DT_SYMBOLIC	16	忽略	忽略	可选	此元素出现于某个共享目标库中时，将改变动态链

					接器在该库中解析引用时使用的符号解析算法。动态链接器不再从可执行文件中开始搜索符号，而是从共享目标中开始搜索。如果共享目标未能提供所引用的符号，动态链接器才会和平常一样搜索可执行文件和其他共享目标。
DT_REL	17	d_ptr	必需	可选	此元素与 DT_RELA 类似，只是其表格中包含隐式的补齐，对 32 位文件而言，就是 Elf32_Rel。如果文件中包含此元素，那么动态结构中也必须包含 DT_RELSZ 和 DT_RELENT 元素。
DT_RELSZ	18	d_val	必需	可选	此元素包含 DT_REL 重定位表的总计大小，按字节数计算。
DT_RELENT	19	d_val	必需	可选	此元素包含 DT_REL 重定位项的大小，按字节数计算。
DT_PLTREL	20	d_val	可选	可选	此成员给出过程链接表所引用的重定位项的类型。根据具体情况，d_val 成员包含 DT_REL 或者 DT_RELA。过程链接表中的所有重定位都必须采用相同的重定位方式。
DT_DEBUG	21	d_ptr	可选	忽略	此成员用于调试。ABI 未规定其内容，访问这些条目的程序与 ABI 不兼容。
DT_TEXTREL	22	忽略	可选	可选	如果文件中不包含此成员，则表示没有任何重定位表项能够引起对不可写段的修改，正如程序头部表中段许可所规定的。如果存在此成员，则存在若干重定位项要求对不可写段进行修改，动态链接器因此可以作相应的准备。
DT_JMPREL	23	d_ptr	可选	可选	如果存在这种成员，则表示条目的 d_ptr 成员包含了某个重定位项的地址，并且该重定位项仅与过程链接表相关。把重定位项分开有利于让动态链接器在进程初始化时忽略它们，当然后期绑定必须可行。如果存在此成员，相关的 DT_PLTRELSZ 和 DT_PLTREL 必须也存在。
DT_LOPROC	0x70000000	未指定	未指定	未指定	这个范围的表项，包括 DT_LOPROC 和 DT_HIPROC 都是保留给处理器特定的语义的。
DT_HIPROC	0x7fffffff	未指定	未指定	未指定	

注：

- 没有出现在此表中的标记值是保留的。
- 除了数组末尾的 DT_NULL 元素以及 DT_NEEDED 元素的相对顺序约束以外，其他项目可以以任意顺序出现。

3.8.3.4 共享目标的依赖关系

在链接编辑器处理某个归档库时，它会从库中提取成员并将它们复制到输出目标文件中。这些静态链接的服务都是在执行中可用的，不需要动态链接器的参与。共享目标

所提供的服务则必须由动态链接器附加到进程映像中。因此可执行文件和共享目标文件描述了它们的特定的依赖关系。

在动态链接器为某个目标文件创建内存段时，依赖关系（记录于动态结构的 DT_NEEDED 表项中）能够提供需要哪些目标来提供程序服务的信息。通过不断地将被引用的共享目标与他们的依赖之间建立连接，动态链接器构造出完整的进程映像。在解析符号引用时，动态链接程序使用宽度优先算法检查符号表。

就是说首先检查可执行程序自身的符号表，然后检查 DT_NEEDED 条目（按顺序）的符号表，接着在第二级 DT_NEEDED 条目上搜索。共享目标文件必须对进程而言可读，不需要其他权限。

即使某个共享目标在依赖表中出现多次，动态链接器也仅会对其连接一次。

存在于依赖表中的名称或者是 DT_SONAME 字符串的副本，或者是用来构造目标文件的共享目标的路径名称。例如，如果连接编辑器在构造某个可执行文件时使用的一个共享目标中包含 lib1 的 DT_SONAME 项，并且使用了路径名为 /usr/llib/lib2 的共享目标库，那么可执行文件将在其依赖表中包含 lib1 和 /usr/lib/lib2。

如果共享目标的名称中包含一个或者多个斜线（/），那么动态链接器将使用该字符串作为路径名称。如果名称中没有斜线，则对共享目标的路径搜索按如下顺序进行：

- 动态数组标记 DT_RPATH 中可能包含若干字符串，这些字符串用“:”分隔，用来通知动态链接器从哪里开始搜索。默认情况下最后搜索当前目录。
- 进程环境中可能包含一个名为 LD_LIBRARY_PATH 的变量，其中也包含若干用“:”分隔的路径名。变量可以以“;”结尾。所有 LD_LIBRARY_PATH 都在 DT_RPATH 之后被搜索。尽管某些程序对分号前后的列表的处理有所不同，动态链接程序并不这样，它能够接受分号，语义如上。
- 最后，如果上面两组目录搜索都失败，未能找到所需要的库，则对/usr/lib 进行搜索。

注意：

出于安全性考虑，动态链接器会针对 SUID 或者 SGID 的程序忽略环境搜索规范（如 LD_LIBRARY_PATH）。不过仍然会搜索 DT_RPATH 和/usr/lib 路径。

3.8.4 全局偏移表（GOT）

位置独立的代码一般不能包含绝对的虚拟地址。全局偏移表在私有数据中包含绝对地址，从而使得地址可用，并且不会影响位置独立性和程序代码的可共享性。程序使用位置独立的寻址引用其全局偏移表，并取得绝对值，从而把位置独立的引用重定向到绝对位置。

全局偏移表中最初包含其重定位项中要求的信息。在系统为可加载目标创建内存段以后，动态链接器要处理重定位项，其中有一些重定位项的类型是 R_386_GLOB_DAT，是对全局偏移表的引用。动态链接器确定相关的符号取值，计算其绝对地质，并将相应的内存表格项目设置为正确的数值。尽管在链接编辑器构造一个目标文件时还无法知道绝对地质，动态链接器清楚所有内存段的地址，因而能够计算其中所包含的符号的绝对地址。

如果程序需要直接访问某个符号的绝对地址，那么该符号就会具有一个全局偏移表项。由于可执行文件和共享目标具有独立的全局偏移表，一个符号的地址可能出现在多个表中。动态链接器在将控制交给进程映像中任何代码之前，要处理所有的全局偏移表重定位，因而确保了执行过程中绝对地址信息可用。

表项 0 是保留的，用来存放动态结构的地址，可以用符号_DYNAMIC 引用之。这样，类似动态链接器这种程序能够在尚未处理其重定位项的时候先找到自己的动态结构。对于动态链接器而言这点很重要，因为它必须能够在不依赖其他程序来对其内存映像进行重定位的前提下，初始化自己。在 32 位 Intel 体系结构下，全局偏移表中的表项 1 和 2 也是保留的。

系统可能在不同的程序中为相同的共享目标选择不同的内存段地址，甚至为统一程序的两次执行选择不同的库地址。尽管如此，一旦进程映像被建立起来，内存段不会改变其地址。只有进程存在，其内存段都位于固定的虚地址。

全局偏移表的格式和解释都是和处理器相关的。对于 32 位 Intel 体系结构而言，符号 _GLOBAL_OFFSET_TABLE_ 可以用来访问该表。

```
extern Elf32_Addr _GLOBAL_OFFSET_TABLE[];
```

符号 _GLOBAL_OFFSET_TABLE_ 可能存在于 .got 节区的中间，允许使用负的/非负的下标来访问地址数组。

3.8.5 过程链接表（PLT）

全局偏移表（GOT,）用来将位置独立的地址计算重定向到绝对位置，与此相似，过程链接表（PLT）能够把位置独立的函数调用重定向到绝对位置。链接编辑器物能解析从一个可执行文件/共享目标到另一个可执行文件/共享目标控制转移（例如函数调用）。因此，链接编辑器让程序把控制转移给过程链接表中的表项。在 System V 中，过程链接表位于共享正文中，不过它们使用位于私有的全局偏移表中的地址。

动态链接器能够确定目标处的绝对地址，并据此修改全局偏移表的内存映像。动态链接器因此能够对表项进行重定位，并且不会影响程序代码的位置独立性和可共享性。可执行文件和共享目标文件拥有各自独立的过程链接表。

```
.PLT0: pushl got_plus_4
       jmp   *got_plus_8
       nop;  nop
       nop;  nop
.PLT1: jmp   *name1_in_GOT
       pushl $offset@PC
.PLT2: jmp   *name2_in_GOT
       pushl $offset
       jmp   .PLT0@PC
       ...
```

图 11 绝对过程链接表

```
.PLT0: pushl 4(%ebx)
       jmp   *8(%ebx)
       nop;  nop
       nop;  nop
.PLT1: jmp   *name1@GOT(%ebx)
       pushl $offset
       jmp   .PLT0@PC
.PLT2: jmp   *name2@GOT(%ebx)
       pushl $offset
       jmp   .PLT0@PC
       ...
```

图 12 位置独立的过程链接表

如图所示，过程链接表命令针对绝对代码（图 11）和位置独立的代码（图 12）使用不同的操作数寻址模式。尽管如此，它们对动态链接器的接口还是相同的。

动态链接器和程序“合作”，通过过程链接表和全局偏移表解析符号引用：

1. 在第一次创建程序的内存映像时，动态链接器为全局偏移表的第二和第三项设置特殊值。
2. 如果过程链接表是位置独立的，全局偏移表必须位于 %bx 中，进程映像中的每个共享目标文件都有自己的过程链接表，控制向过程链接表项的传递仅发生在同一个目标文件中。因此，调用函数用负责在调用过程链接表项之前设置全局偏移表的基址寄存器。
3. 出于说明的目的，假定程序调用了 name1，name1 将控制传输给标号 .PLT1。
4. 第一条指令跳转到 name1 的全局偏移表项的地址。最初，全局偏移表中包含后面的 pushl 指令的地址，而不是 name1 的真实地址。
5. 接下来，程序将重定位偏移（offset）压栈。重定位偏移是一个 32 位非负数，是在重定位表中的字节偏移量。指定的重定位表项的类型为 R_386_JMP_SLOT，其偏移将给出在前面的 jmp 指令中使用的 GOT 表项。重定位项也包含一个符号表索引，借以告诉动态链接器被引用的符号是什么，在这里是 name1。
6. 在将重定位偏移压栈后，程序会跳转到 .PLT0，也就是过程链接表的第一项。pushl 指令把第二个全局偏移表项（got_plus_4 或者 4(%ebx)）压入堆栈，因而为动态链接器提供了识别信息的机会。程序然后跳转到第三个 GOT 表项内保存的地址（got_plus_8 或者 8(%ebx)），后者将控制传递给动态链接器。
7. 当动态链接器得到控制后，它恢复堆栈，查看指定的重定位项，寻找符号的值，将 name1 的“真实”地址存储于全局偏移表项中，并将控制传递给期望的目的地。

8. 过程链接表项的后续执行将把控制直接传递给 name1，不会再次调用动态链接器。就是说 .PLT1 处的 jmp 将控制传递给 name1，而不会执行后面的 pushl 指令。

环境变量 LD_BIND_NOW 可以更改动态链接行为。如果其取值非空，动态链接器会在控制传递给程序之前，对过程链接表项进行计算。就是说动态链接器会在进程初始化的过程中处理类型为 R_386_JMP_SLOT 的重定位项。否则，动态链接器会对过程链接表实行懒惰计算，延迟符号解析和重定位，直到某个表项的第一次执行。

懒惰绑定通常会提供整体的应用性能，因为未使用的符号不会引入额外的动态链接开销。尽管如此，有些应用情形会使得懒惰绑定不太合适。首先，对共享目标函数的第一次引用花的时间会超出后续调用，因为动态链接器要截获调用以便解析符号。一些应用不能容忍这种不可预测性。第二，如果发生了错误，动态链接器无法解析某个符号，动态链接器会终止程序。在懒惰绑定下，这类事情可能会发生任意多次。某些应用也可能无法容忍这种不可预测性。通过关闭懒惰绑定，动态链接器会迫使所有错误都发生在进程初始化期间，而不是应用程序接收控制以后。

3.8.6 哈希表（Hash Table）

由 Elf32_Word 对象组成的哈希表支持符号表访问。下面的例子有助于解释哈希表组织，不过不是规范的一部分。

图 13 符号哈希表的组织

nbucket
nchain
bucket[0] ... bucket[nbucket-1]
chain[0] ... chain[nchain-a]

bucket 数组包含 nbucket 个项目，chain 数组包含 nchain 个项目，下标都是从 0 开始。bucket 和 chain 中都保存符号表索引。Chain 表项和符号表存在对应。符号表项的数目应该和 nchain 相等，所以符号表的索引也可用来选取 chain 表项。哈希函数能够接受符号名并且返回一个可以用来计算 bucket 的索引。

因此，如果哈希函数针对某个名字返回了数值 X，则 bucket[X%nbucket] 给出了一个索引 y，该索引可用于符号表，也可用于 chain 表。如果符号表项不是所需要的，那么 chain[y] 则给出了具有相同哈希值的下一个符号表项。我们可以沿着 chain 链一直搜索，直到所选中的符号表项包含了所需要的符号，或者 chain 项中包含值 STN_UNDEF。

3.8.6.1 哈希函数

ELF 实现中常用的哈希函数如下，有时候会作一些优化（比如 Linux）。

```
unsigned long
elf_hash (const unsigned char *name)
{
    unsigned long h = 0, g;
    while (*name)
    {
        h = (h << 4) + *name++;
        if (g = h & 0xf0000000)
            h ^= g >> 24;
        h &= ~g;
    }
    return h;
}
```

3.8.7 初始化和终止函数

在动态链接器构造了进程映像，并执行了重定位以后，每个共享的目标都获得执行某些初始化代码的机会。这些初始化函数的被调用顺序是不一定的，不过所有共享目标初始化都会在可执行文件得到控制之前发生。

类似地，共享目标也包含终止函数，这些函数在进程完成终止动作序列时，通过 atexit() 机制执行。动态链接器对终止函数的调用顺序是不确定的。

共享目标通过动态结构中的 DT_INIT 和 DT_FINI 条目指定初始化/终止函数。通常这些代码放在.init 和.fini 节区中。

> 注意：
>
> 尽管 atexit() 终止处理通常会被执行，在进程消亡时并不能保证被执行。特别地，如果进程调用了 _exit 或者进程因为收到某个它既未捕捉又未忽略的信号而终止时，不会执行终止处理。

3.9 C 库

3.9.1 关于 C 库函数

C 库（libc）包含所有 libsys 中的符号，另外还包含下面两个表中列举的例程。第一个表中的例程来自于 ANSI C 标准。

表 23 libc 内容中无同义词的名字

abort	fputc	isprint	putc	strncmp
abs	fputs	ispunct	putchar	strncpy
asctime	fread	isspace	puts	strpbrk
atof	freopen	isupper	qsort	strrchr

atoi	frexp	isxdigit	raise	strspn
atol	fscanf	labs	rand	strstr
bsearch	fseek	ldexp	rewind	strtod
clearerr	fsetpos	ldiv	scanf	strtok
clock	ftell	localtime	setbuf	strtol
ctime	fwrite	longjmp	setjmp	strtoul
difftime	getc	mblen	setvbuf	tmpfile
div	getchar	mbstowcs	sprintf	tmpnam
fclose	getenv	mbtowc	srand	tolower
feof	gets	memchr	sscanf	toupper
ferror	gmtime	memcmp	strcat	ungetc
fflush	isalnum	memcpy	strchr	vfprintf
fgetc	isalpha	memmove	strcmp	vprintf
fgetpos	iscntrl	memset	strcpy	vsprintf
fgets	isdigit	mktime	strcspn	wcstombs
fopen	isgraph	perror	strlen	wctomb
fprintf	islower	printf	strncat	

表 24 Libc 中有同义词的名字

__assert	getdate	lockf	sleep	tell
cfgetispeed	getopt	lsearch	strdup	tempnam
cfgetospeed	getpass	memccpy	swab	tfind
cfsetispeed	getsubopt	mkfifo	tcdrain	toascii
cfsetospeed	getw	mktemp	tcflow	_tolower
ctermid	hcreate	monitor	tcflush	tsearch
cuserid	hdestroy	nftw	tcgetattr	_toupper
dup2	hsearch	nl_langinfo	tcgetpgrp	twalk
fdopen	isascii	pclose	tcgetsid	tzset
__filbuf	isatty	popen	tcsendbreak	_xftw
fileno				

除了在表 24 中定义的符号以外，name 项的形式为_name 的符号没有列出，但也存在。因而 libc 中包含 getopt 也包含_getopt。

在前面列举的例程中，以下例程没有在其他地方定义：

- int __fillbuf(FILE *f);

  此函数为 f 返回下一个输入字符，并适当地填充其缓冲区。如果发生错误，则返回 EOF。

- int __flsbuf(FILE *f);

  此函数冲洗 f 的输出字符，就好像 putc(x, f) 被调用一样，并且将 x 的取值追加到结果输出流中。如果发生错误，则返回 EOF，否则返回 x。

- int _xftw(int, char*, int(*)(char *, struct stat *, int), int);

  对 ftw() 的调用会在应用程序被编译时被映射到此函数。此函数与 ftw()

相同，只是 _xftw() 多了第一个参数，该参数必须取值为 2。

注意：
可能要求库函数对 SVID、ANSI C、POSIX 兼容。

3.9.2 全局数据符号

libc 要求定义某些全局外部变量来确保其中的例程能够正确工作。 libsys 库所需要的所有数据符号都必须通过 libc 来提供，正如下面的表格所给出。

要查看这些符号所代表的数据对象的正式定义，可参考 *System V Interface Definition, 3rd Edtion* 或者 *System V ABI* 中第 6 章相关处理器材料的“Data Definition”节。

对于下表中以 name - _name 形式出现的条目，每一对条目中的符号都代表相同的数据。带下划线的同义词是用来满足 ANSI C 标准的。

表 25 libc 中的全局外部数据符号

getdate_err	optarg
_getdate_err	opterr
__iob	optind
	optopt

参考文献

[1] Tool Interface Standards(TIS) Committee, Portable Formats Specification, Version 1.1.

[2] Tool Interface Standards(TIS) Committee, Executable and Linking Format(ELF) Specification, Version 1.2, May, 1995.

[3] AT&T, The Santa Cruz Operation, Inc. *System V Application Binary Interface,* Edtion 4.1. DRAFT COPY, March 18, 1997.

[4] AT&T, The Santa Cruz Operation, Inc. *System V Application Binary Interface, Intel386™ Architecture Processor Supplement. Fourth Edition.*

[5] Free Standards Group, Linux Standard Base, Version 1.3, http://www.linuxbase.org/spec