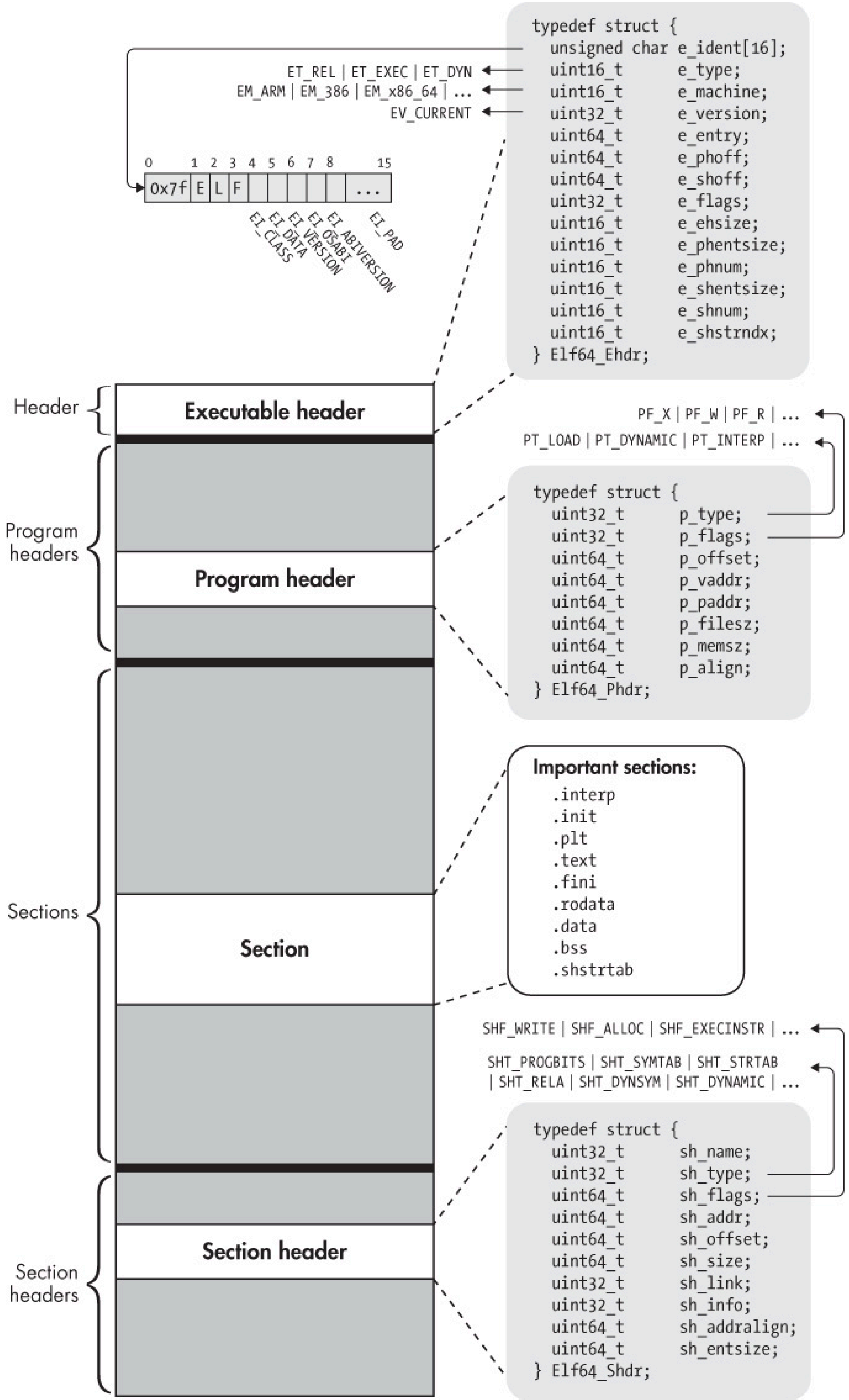
ELF Format Cheatsheet

elf_format_cheatsheet.md

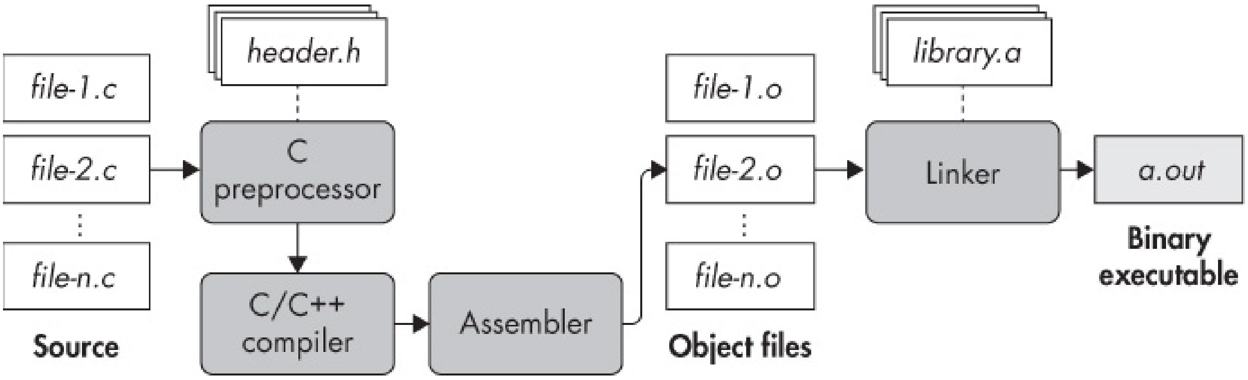
ELF Format Cheatsheet

Introduction

Executable and Linkable Format (ELF), is the default binary format on Linux-based systems.


typedef struct {
unsigned char e_ident[16];
uint16_t e_type;
uint16_t e_machine;
uint32_t e_version;
uint64_t e_entry;
uint64_t e_phoff;
uint64_t e_shoff;
uint32_t e_flags;
uint16_t e_ehsize;
uint16_t e_phentsize;
uint16_t e_phnum;
uint16_t e_shentsize;
uint16_t e_shnum;
uint16_t e_shstrndx;
} Elf64_Ehdr;
ET_REL | ET_EXEC | ET_DYN
EM_ARM | EM_386 | EM_x86_64 | ...
EV_CURRENT
0 1 2 3 4 5 6 7 8 15
0x7f E L F ...
EI_CLASS
EI_DATA
EI_VERSION
EI_OSABI
EI_ABIVERSION
EI_PAD
Header
Executable header
Program headers
Program header
Sections
Section
Section headers
Section header
PF_X | PF_W | PF_R | ...
PT_LOAD | PT_DYNAMIC | PT_INTERP | ...
typedef struct {
uint32_t p_type;
uint32_t p_flags;
uint64_t p_offset;
uint64_t p_vaddr;
uint64_t p_paddr;
uint64_t p_filesz;
uint64_t p_memsz;
uint64_t p_align;
} Elf64_Phdr;
Important sections:
.interp
.init
.plt
.text
.fini
.rodata
.data
.bss
.shstrtab
SHF_WRITE | SHF_ALLOC | SHF_EXECINSTR | ...
SHT_PROGBITS | SHT_SYMTAB | SHT_STRTAB
| SHT_RELA | SHT_DYNSYM | SHT_DYNAMIC | ...
typedef struct {
uint32_t sh_name;
uint32_t sh_type;
uint64_t sh_flags;
uint64_t sh_addr;
uint64_t sh_offset;
uint64_t sh_size;
uint32_t sh_link;
uint32_t sh_info;
uint64_t sh_addralign;
uint64_t sh_entsize;
} Elf64_Shdr;

Compilation



Executable Headers (Ehdr)

This is the only part of the ELF that must be in an specific location (at the starting of the ELF file).

It defines basic information, such as the file magic number to know whether a file is an ELF or another type. Also it defines type of ELF, architecture and some options that will link it to other parts of the ELF file.

32-bit struct:

```
#define EI_NIDENT (16)

typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info
*/
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address
*/
  Elf32_Off     e_phoff;                /* Program header table file
offset */
  Elf32_Off     e_shoff;                /* Section header table file
offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry
size */
  Elf32_Half    e_phnum;                /* Program header table entry
count */
  Elf32_Half    e_shentsize;            /* Section header table entry
size */
  Elf32_Half    e_shnum;                /* Section header table entry
count */
  Elf32_Half    e_shstrndx;             /* Section header string table
```

```
index */
} Elf32_Ehdr;
```

64-bit struct:

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info
*/
  Elf64_Half    e_type;                 /* Object file type */
  Elf64_Half    e_machine;              /* Architecture */
  Elf64_Word    e_version;              /* Object file version */
  Elf64_Addr    e_entry;                /* Entry point virtual address
*/
  Elf64_Off     e_phoff;                /* Program header table file
offset */
  Elf64_Off     e_shoff;                /* Section header table file
offset */
  Elf64_Word    e_flags;                /* Processor-specific flags */
  Elf64_Half    e_ehsize;               /* ELF header size in bytes */
  Elf64_Half    e_phentsize;            /* Program header table entry
size */
  Elf64_Half    e_phnum;                /* Program header table entry
count */
  Elf64_Half    e_shentsize;            /* Section header table entry
size */
  Elf64_Half    e_shnum;                /* Section header table entry
count */
  Elf64_Half    e_shstrndx;             /* Section header string table
index */
} Elf64_Ehdr;
```

The `EI_NIDENT`, is the size in bytes of the first struct entry, the `e_type`.

It is the ELF magic headers and some basic specifications of the file.

Values:

- `e_ident`: It is a 16-byte array that identifies the ELF object, it always starts with "\x7fELF".
- `e_type`: Specifies the ELF type:
  - `ET_NONE` (Undefined): ELF Format unknown or not specified.
  - `ET_EXEC`: (Executable file): An ELF executable.
  - `ET_DYN`: (Shared object): A library or a dynamically-linked executable.
  - `ET_REL` (Relocatable file): Relocatable files (.o object files).
  - `ET_CORE` (Core dump): A core dump file.
- `e_machine`: Target architecture.

- `e_version` : ELF file version.
- `e_entry` : Entry point address.
- `e_phoff` : Phdr offset.
- `e_shoff` : Shdr offset.
- `e_flags` : Processor-specific flags.
- `e_ehsize` Ehdr size (in bytes). (Usually 64 bytes in 64-bit ELF and 52 bytes for 32 bits)
- `e_phentsize` : Phdr entry size.
- `e_phnum` : Phdr entries.
- `e_shentsize` : Shdr entry size.
- `e_shnum` : Shdr entries.
- `e_shstrndx` : Shdr string table index (`.shstrtab` , it contains null terminated-strings with the name of each section)

Note: `e_phoff` and `e_shoff` are offsets of the ELF file, `e_entry` instead is a virtual address.

---- Needed type definitions ----

e_type defines:

```
#define ET_NONE         0               /* No file type */
#define ET_REL          1               /* Relocatable file */
#define ET_EXEC         2               /* Executable file */
#define ET_DYN          3               /* Shared object file */
#define ET_CORE         4               /* Core file */
#define ET_NUM          5               /* Number of defined types */
#define ET_LOOS         0xfe00          /* OS-specific range start */
#define ET_HIOS         0xfeff          /* OS-specific range end */
#define ET_LOPROC       0xff00          /* Processor-specific range
start */
#define ET_HIPROC       0xffff          /* Processor-specific range end
*/
```

e_machine defines:

```
#define EM_NONE          0      /* No machine */
#define EM_M32           1      /* AT&T WE 32100 */
#define EM_SPARC         2      /* SUN SPARC */
#define EM_386           3      /* Intel 80386 */
#define EM_68K           4      /* Motorola m68k family */
#define EM_88K           5      /* Motorola m88k family */
#define EM_IAMCU         6      /* Intel MCU */
#define EM_860           7      /* Intel 80860 */
#define EM_MIPS          8      /* MIPS R3000 big-endian */
```

```
#define EM_S370          9      /* IBM System/370 */
#define EM_MIPS_RS3_LE  10      /* MIPS R3000 little-endian */
                                /* reserved 11-14 */
#define EM_PARISC       15      /* HPPA */
                                /* reserved 16 */
#define EM_VPP500       17      /* Fujitsu VPP500 */
#define EM_SPARC32PLUS  18      /* Sun's "v8plus" */
#define EM_960          19      /* Intel 80960 */
#define EM_PPC          20      /* PowerPC */
#define EM_PPC64        21      /* PowerPC 64-bit */
#define EM_S390         22      /* IBM S390 */
#define EM_SPU          23      /* IBM SPU/SPC */
                                /* reserved 24-35 */
#define EM_V800         36      /* NEC V800 series */
#define EM_FR20         37      /* Fujitsu FR20 */
#define EM_RH32         38      /* TRW RH-32 */
#define EM_RCE          39      /* Motorola RCE */
#define EM_ARM          40      /* ARM */
#define EM_FAKE_ALPHA   41      /* Digital Alpha */
#define EM_SH           42      /* Hitachi SH */
#define EM_SPARCV9      43      /* SPARC v9 64-bit */
#define EM_TRICORE      44      /* Siemens Tricore */
#define EM_ARC          45      /* Argonaut RISC Core */
#define EM_H8_300       46      /* Hitachi H8/300 */
#define EM_H8_300H      47      /* Hitachi H8/300H */
#define EM_H8S          48      /* Hitachi H8S */
#define EM_H8_500       49      /* Hitachi H8/500 */
#define EM_IA_64        50      /* Intel Merced */
#define EM_MIPS_X       51      /* Stanford MIPS-X */
#define EM_COLDFIRE     52      /* Motorola Coldfire */
#define EM_68HC12       53      /* Motorola M68HC12 */
#define EM_MMA          54      /* Fujitsu MMA Multimedia Accelerator */
#define EM_PCP          55      /* Siemens PCP */
#define EM_NCPU         56      /* Sony nCPU embeeded RISC */
#define EM_NDR1         57      /* Denso NDR1 microprocessor */
#define EM_STARCORE     58      /* Motorola Start*Core processor */
#define EM_ME16         59      /* Toyota ME16 processor */
#define EM_ST100        60      /* STMicroelectronic ST100 processor */
#define EM_TINYJ        61      /* Advanced Logic Corp. Tinyj emb.fam */
#define EM_X86_64       62      /* AMD x86-64 architecture */
#define EM_PDSP         63      /* Sony DSP Processor */
#define EM_PDP10        64      /* Digital PDP-10 */
#define EM_PDP11        65      /* Digital PDP-11 */
#define EM_FX66         66      /* Siemens FX66 microcontroller */
#define EM_ST9PLUS      67      /* STMicroelectronics ST9+ 8/16 mc */
#define EM_ST7          68      /* STmicroelectronics ST7 8 bit mc */
#define EM_68HC16       69      /* Motorola MC68HC16 microcontroller */
#define EM_68HC11       70      /* Motorola MC68HC11 microcontroller */
#define EM_68HC08       71      /* Motorola MC68HC08 microcontroller */
#define EM_68HC05       72      /* Motorola MC68HC05 microcontroller */
#define EM_SVX          73      /* Silicon Graphics SVx */
#define EM_ST19         74      /* STMicroelectronics ST19 8 bit mc */
#define EM_VAX          75      /* Digital VAX */
```

```
#define EM_CRIS         76      /* Axis Communications 32-bit emb.proc
*/
#define EM_JAVELIN      77      /* Infineon Technologies 32-bit emb.proc
*/
#define EM_FIREPATH     78      /* Element 14 64-bit DSP Processor */
#define EM_ZSP          79      /* LSI Logic 16-bit DSP Processor */
#define EM_MMIX         80      /* Donald Knuth's educational 64-bit
proc */
#define EM_HUANY        81      /* Harvard University machine-
independent object files */
#define EM_PRISM        82      /* SiTera Prism */
#define EM_AVR          83      /* Atmel AVR 8-bit microcontroller */
#define EM_FR30         84      /* Fujitsu FR30 */
#define EM_D10V         85      /* Mitsubishi D10V */
#define EM_D30V         86      /* Mitsubishi D30V */
#define EM_V850         87      /* NEC v850 */
#define EM_M32R         88      /* Mitsubishi M32R */
#define EM_MN10300      89      /* Matsushita MN10300 */
#define EM_MN10200      90      /* Matsushita MN10200 */
#define EM_PJ           91      /* picoJava */
#define EM_OPENRISC     92      /* OpenRISC 32-bit embedded processor */
#define EM_ARC_COMPACT  93      /* ARC International ARCompact */
#define EM_XTENSA       94      /* Tensilica Xtensa Architecture */
#define EM_VIDEOCORE    95      /* Alphamosaic VideoCore */
#define EM_TMM_GPP      96      /* Thompson Multimedia General Purpose
Proc */
#define EM_NS32K        97      /* National Semi. 32000 */
#define EM_TPC          98      /* Tenor Network TPC */
#define EM_SNP1K        99      /* Trebia SNP 1000 */
#define EM_ST200        100     /* STMicroelectronics ST200 */
#define EM_IP2K         101     /* Ubicom IP2xxx */
#define EM_MAX          102     /* MAX processor */
#define EM_CR           103     /* National Semi. CompactRISC */
#define EM_F2MC16       104     /* Fujitsu F2MC16 */
#define EM_MSP430       105     /* Texas Instruments msp430 */
#define EM_BLACKFIN     106     /* Analog Devices Blackfin DSP */
#define EM_SE_C33       107     /* Seiko Epson S1C33 family */
#define EM_SEP          108     /* Sharp embedded microprocessor */
#define EM_ARCA         109     /* Arca RISC */
#define EM_UNICORE      110     /* PKU-Unity & MPRC Peking Uni. mc
series */
#define EM_EXCESS       111     /* eXcess configurable cpu */
#define EM_DXP          112     /* Icera Semi. Deep Execution Processor
*/
#define EM_ALTERA_NIOS2 113     /* Altera Nios II */
#define EM_CRX          114     /* National Semi. CompactRISC CRX */
#define EM_XGATE        115     /* Motorola XGATE */
#define EM_C166         116     /* Infineon C16x/XC16x */
#define EM_M16C         117     /* Renesas M16C */
#define EM_DSPIC30F     118     /* Microchip Technology dsPIC30F */
#define EM_CE           119     /* Freescale Communication Engine RISC
*/
#define EM_M32C         120     /* Renesas M32C */
```

```
                                  /* reserved 121-130 */
#define EM_TSK3000      131       /* Altium TSK3000 */
#define EM_RS08         132       /* Freescale RS08 */
#define EM_SHARC        133       /* Analog Devices SHARC family */
#define EM_ECOG2        134       /* Cyan Technology eCOG2 */
#define EM_SCORE7       135       /* Sunplus S+core7 RISC */
#define EM_DSP24        136       /* New Japan Radio (NJR) 24-bit DSP */
#define EM_VIDEOCORE3   137       /* Broadcom VideoCore III */
#define EM_LATTICEMICO32 138      /* RISC for Lattice FPGA */
#define EM_SE_C17       139       /* Seiko Epson C17 */
#define EM_TI_C6000     140       /* Texas Instruments TMS320C6000 DSP */
#define EM_TI_C2000     141       /* Texas Instruments TMS320C2000 DSP */
#define EM_TI_C5500     142       /* Texas Instruments TMS320C55x DSP */
#define EM_TI_ARP32     143       /* Texas Instruments App. Specific RISC
*/
#define EM_TI_PRU       144       /* Texas Instruments Prog. Realtime Unit
*/
                                  /* reserved 145-159 */
#define EM_MMDSP_PLUS   160       /* STMicroelectronics 64bit VLIW DSP */
#define EM_CYPRESS_M8C  161       /* Cypress M8C */
#define EM_R32C         162       /* Renesas R32C */
#define EM_TRIMEDIA     163       /* NXP Semi. TriMedia */
#define EM_QDSP6        164       /* QUALCOMM DSP6 */
#define EM_8051         165       /* Intel 8051 and variants */
#define EM_STXP7X       166       /* STMicroelectronics STxP7x */
#define EM_NDS32        167       /* Andes Tech. compact code emb. RISC */
#define EM_ECOG1X       168       /* Cyan Technology eCOG1X */
#define EM_MAXQ30       169       /* Dallas Semi. MAXQ30 mc */
#define EM_XIMO16       170       /* New Japan Radio (NJR) 16-bit DSP */
#define EM_MANIK        171       /* M2000 Reconfigurable RISC */
#define EM_CRAYNV2      172       /* Cray NV2 vector architecture */
#define EM_RX           173       /* Renesas RX */
#define EM_METAG        174       /* Imagination Tech. META */
#define EM_MCST_ELBRUS  175       /* MCST Elbrus */
#define EM_ECOG16       176       /* Cyan Technology eCOG16 */
#define EM_CR16         177       /* National Semi. CompactRISC CR16 */
#define EM_ETPU         178       /* Freescale Extended Time Processing
Unit */
#define EM_SLE9X        179       /* Infineon Tech. SLE9X */
#define EM_L10M         180       /* Intel L10M */
#define EM_K10M         181       /* Intel K10M */
                                  /* reserved 182 */
#define EM_AARCH64      183       /* ARM AARCH64 */
                                  /* reserved 184 */
#define EM_AVR32        185       /* Amtel 32-bit microprocessor */
#define EM_STM8         186       /* STMicroelectronics STM8 */
#define EM_TILE64       187       /* Tileta TILE64 */
#define EM_TILEPRO      188       /* Tilera TILEPro */
#define EM_MICROBLAZE   189       /* Xilinx MicroBlaze */
#define EM_CUDA         190       /* NVIDIA CUDA */
#define EM_TILEGX       191       /* Tilera TILE-Gx */
#define EM_CLOUDSHIELD  192       /* CloudShield */
#define EM_COREA_1ST    193       /* KIPO-KAIST Core-A 1st gen. */
```

```
#define EM_COREA_2ND    194     /* KIPO-KAIST Core-A 2nd gen. */
#define EM_ARC_COMPACT2 195     /* Synopsys ARCompact V2 */
#define EM_OPEN8        196     /* Open8 RISC */
#define EM_RL78         197     /* Renesas RL78 */
#define EM_VIDEOCORE5   198     /* Broadcom VideoCore V */
#define EM_78KOR        199     /* Renesas 78KOR */
#define EM_56800EX      200     /* Freescale 56800EX DSC */
#define EM_BA1          201     /* Beyond BA1 */
#define EM_BA2          202     /* Beyond BA2 */
#define EM_XCORE        203     /* XMOS xCORE */
#define EM_MCHP_PIC     204     /* Microchip 8-bit PIC(r) */
                                /* reserved 205-209 */
#define EM_KM32         210     /* KM211 KM32 */
#define EM_KMX32        211     /* KM211 KMX32 */
#define EM_EMX16        212     /* KM211 KMX16 */
#define EM_EMX8         213     /* KM211 KMX8 */
#define EM_KVARC        214     /* KM211 KVARC */
#define EM_CDP          215     /* Paneve CDP */
#define EM_COGE         216     /* Cognitive Smart Memory Processor */
#define EM_COOL         217     /* Bluechip CoolEngine */
#define EM_NORC         218     /* Nanoradio Optimized RISC */
#define EM_CSR_KALIMBA  219     /* CSR Kalimba */
#define EM_Z80          220     /* Zilog Z80 */
#define EM_VISIUM       221     /* Controls and Data Services VISIUMcore
*/
#define EM_FT32         222     /* FTDI Chip FT32 */
#define EM_MOXIE        223     /* Moxie processor */
#define EM_AMDGPU       224     /* AMD GPU */
                                /* reserved 225-242 */
#define EM_RISCV        243     /* RISC-V */

#define EM_BPF          247     /* Linux BPF -- in-kernel virtual
machine */
#define EM_CSKY         252     /* C-SKY */

#define EM_NUM          253

/* Old spellings/synonyms.  */

#define EM_ARC_A5       EM_ARC_COMPACT

/* If it is necessary to assign new unofficial EM_* values, please
   pick large random numbers (0x8523, 0xa7f2, etc.) to minimize the
   chances of collision with official or non-GNU unofficial values.  */

#define EM_ALPHA        0x9026
```

e_version defines:

```
#define EV_NONE         0               /* Invalid ELF version */
#define EV_CURRENT      1               /* Current version */
```

```
#define EV_NUM          2
```

Section Headers (Shdr)

The code and data is divided into contiguous non-overlapping chunks called sections.

It is just an space to store data or code, which its specifications are in a section header specifying needed details such as the size and offset.

Every section has a section header which defines it.

32-bit struct:

```
typedef struct
{
  Elf32_Word    sh_name;                /* Section name (string tbl
index) */
  Elf32_Word    sh_type;                /* Section type */
  Elf32_Word    sh_flags;               /* Section flags */
  Elf32_Addr    sh_addr;                /* Section virtual addr at
execution */
  Elf32_Off     sh_offset;              /* Section file offset */
  Elf32_Word    sh_size;                /* Section size in bytes */
  Elf32_Word    sh_link;                /* Link to another section */
  Elf32_Word    sh_info;                /* Additional section
information */
  Elf32_Word    sh_addralign;           /* Section alignment */
  Elf32_Word    sh_entsize;             /* Entry size if section holds
table */
} Elf32_Shdr;
```

64-bit struct:

```
typedef struct
{
  Elf64_Word    sh_name;                /* Section name (string tbl
index) */
  Elf64_Word    sh_type;                /* Section type */
  Elf64_Xword   sh_flags;               /* Section flags */
  Elf64_Addr    sh_addr;                /* Section virtual addr at
execution */
  Elf64_Off     sh_offset;              /* Section file offset */
  Elf64_Xword   sh_size;                /* Section size in bytes */
  Elf64_Word    sh_link;                /* Link to another section */
  Elf64_Word    sh_info;                /* Additional section
information */
  Elf64_Xword   sh_addralign;           /* Section alignment */
```

```
  Elf64_Xword   sh_entsize;             /* Entry size if section holds
table */
} Elf64_Shdr;
```

Values:

- `sh_name` : Index into the string table, if zero it means it has no name. (`.shstrtab`).
- `sh_type` : Type of section.
  - `SHT_NULL` Section table entry unused.
  - `SHT_PROGBITS` : Program data (Such as machine instructions or constants).
  - `SHT_SYMTAB` : Symbol table. (Static symbol table)
  - `SHT_STRTAB` : String table.
  - `SHT_RELA` : Relocation entries with addends.
  - `SHT_HASH` : Symbol hash table.
  - `SHT_DYNAMIC` : Dynamic linking information.
  - `SHT_NOTE` : Notes.
  - `SHT_NOBITS` : Uninitialized data.
  - `SHT_REL` : Relocation entries without addends.
  - `SHT_SHLIB` : Reserved.
  - `SHT_DYNSYM` Dynamic linker symbol table. (Dynamic-linker-used symbol table)
- `sh_flags` : Describes additional information about a section.
  - `SHF_WRITE` : Writable at runtime.
  - `SHF_ALLOC` : The section will be loaded to virtual memory at runtime.
  - `SHF_EXECINSTR` : Contains executable instructions.
- `sh_addr` : Section virtual address at execution.
- `sh_offset` : Section offset in ELF file.
- `sh_size` : Section size (in bytes).
- `sh_link` : Link to another section (Eg.: `SHT_SYMTAB` , `SHT_DYNSYM` , or `SHT_DYNAMIC` has an associated string table which contains the symbolic names for the symbols in question. Relocation sections (type `SHT_REL` or `SHT_RELA`) are associated with a symbol table describing the symbols involved in the relocations.).
- `sh_info` : Additional section information.
- `sh_addralign` : Section alignment.
- `sh_entsize` : Entry size if section holds table. (Some sections, such as symbol tables or relocation tables, contain a table of well-defined data structures

(such as `ElfN_Sym` or `ElfN_Rela`). For such sections, the sh_entsize field indicates the size in bytes of each entry in the table. When the field is unused, it is set to zero).

All the section headers which defines sections, are contained in the section header table.

To load and execute a binary in a process, you need a different organization of the code and data in the binary. For this reason, ELF executables specify another logical organization, called segments, which are used at execution time (as opposed to sections, which are used at link time).

The sections are optional, it is just metadata for debuggers. The program headers are what decides onto how an ELF binary gets loaded in memory.

Then the section headers are not loaded into memory.

---- Type definitions ----

sh_type defines:

```
#define SHT_NULL          0              /* Section header table entry
unused */
#define SHT_PROGBITS      1              /* Program data */
#define SHT_SYMTAB        2              /* Symbol table */
#define SHT_STRTAB        3              /* String table */
#define SHT_RELA          4              /* Relocation entries with
addends */
#define SHT_HASH          5              /* Symbol hash table */
#define SHT_DYNAMIC       6              /* Dynamic linking information
*/
#define SHT_NOTE          7              /* Notes */
#define SHT_NOBITS        8              /* Program space with no data
(bss) */
#define SHT_REL           9              /* Relocation entries, no
addends */
#define SHT_SHLIB         10             /* Reserved */
#define SHT_DYNSYM        11             /* Dynamic linker symbol table
*/
#define SHT_INIT_ARRAY    14             /* Array of constructors */
#define SHT_FINI_ARRAY    15             /* Array of destructors */
#define SHT_PREINIT_ARRAY 16             /* Array of pre-constructors */
#define SHT_GROUP         17             /* Section group */
#define SHT_SYMTAB_SHNDX  18             /* Extended section indeces */
#define SHT_NUM           19             /* Number of defined types.  */
#define SHT_LOOS          0x60000000     /* Start OS-specific.  */
#define SHT_GNU_ATTRIBUTES 0x6ffffff5    /* Object attributes.  */
#define SHT_GNU_HASH      0x6ffffff6     /* GNU-style hash table.  */
#define SHT_GNU_LIBLIST   0x6ffffff7     /* Prelink library list */
#define SHT_CHECKSUM      0x6ffffff8     /* Checksum for DSO content.  */
```

```
#define SHT_LOSUNW        0x6ffffffa    /* Sun-specific low bound.  */
#define SHT_SUNW_move     0x6ffffffa
#define SHT_SUNW_COMDAT   0x6ffffffb
#define SHT_SUNW_syminfo  0x6ffffffc
#define SHT_GNU_verdef    0x6ffffffd    /* Version definition section.
*/
#define SHT_GNU_verneed   0x6ffffffe    /* Version needs section.  */
#define SHT_GNU_versym    0x6fffffff    /* Version symbol table.  */
#define SHT_HISUNW        0x6fffffff    /* Sun-specific high bound.  */
#define SHT_HIOS          0x6fffffff    /* End OS-specific type */
#define SHT_LOPROC        0x70000000    /* Start of processor-specific
*/
#define SHT_HIPROC        0x7fffffff    /* End of processor-specific */
#define SHT_LOUSER        0x80000000    /* Start of application-specific
*/
#define SHT_HIUSER        0x8fffffff    /* End of application-specific
*/
```

sh_flags defines:

```
#define SHF_WRITE            (1 << 0)   /* Writable */
#define SHF_ALLOC            (1 << 1)   /* Occupies memory during
execution */
#define SHF_EXECINSTR        (1 << 2)   /* Executable */
#define SHF_MERGE            (1 << 4)   /* Might be merged */
#define SHF_STRINGS          (1 << 5)   /* Contains nul-terminated
strings */
#define SHF_INFO_LINK        (1 << 6)   /* `sh_info' contains SHT index
*/
#define SHF_LINK_ORDER       (1 << 7)   /* Preserve order after
combining */
#define SHF_OS_NONCONFORMING (1 << 8)   /* Non-standard OS specific
handling
                                           required */
#define SHF_GROUP            (1 << 9)   /* Section is member of a group.
*/
#define SHF_TLS              (1 << 10)  /* Section hold thread-local
data.  */
#define SHF_COMPRESSED       (1 << 11)  /* Section with compressed data.
*/
#define SHF_MASKOS           0x0ff00000 /* OS-specific.  */
#define SHF_MASKPROC         0xf0000000 /* Processor-specific */
#define SHF_ORDERED          (1 << 30)  /* Special ordering requirement
                                           (Solaris).  */
#define SHF_EXCLUDE          (1U << 31) /* Section is excluded unless
                                           referenced or allocated
(Solaris).*/
```

Sections

The first entry in the section header table of every ELF file is defined by the ELF standard to be a NULL entry. The type of the entry is `SHT_NULL`, and all fields in the section header are zeroed out.

Sections:

- `.init` : Executable code that performs initialization tasks and needs to run before any other code in the binary is executed (Then it has `SHF_EXECINSTR` flag) The system executes the code in the .init section before transferring control to the main entry point of the binary.
- `.fini` : The contrary as `.init`, it has executable code that must run after the main program completes.
- `.text` : Is where the main code of the program resides (Then it has `SHF_EXECINSTR` flag), it is `SHT_PROGBITS` because it has user-defined code.
- `.bss` : It contains uninitialized data (Type `SHT_NOBITS`). It does not occupy space at disk to avoid space consuming, then all the data is usually initialized to zero at runtime. It is writable.
- `.data` : Program initialized data, it is writable. (Type `SHT_PROGBITS`).
- `.rodata` : It is read-only data, such as strings used by the code, if the data should be writable then `.data` is used instead. Data that goes here can be for example hardcoded strings used for a `printf`.
- `.plt` : Stands for Procedure Linkage Table. It is code used for dynamic linking purposed that helps to call external functions from shared libraries with the help of the GOT (Global Offset Table).
- `.got.plt` : It is a table where resolved addresses from external functions are stored. It is by default writable as by default Lazy Binding is used. (Unless Relocation Read-Only is used or LD_BIND_NOW env var is exported to resolve all the imported functions at the program initialization).
- `.rel.*` : Contains information about how parts of an ELF object or process image need to be fixed up or modified at linking or runtime (Type `SHT_REL`).
- `.rela.*` : Contains information about how parts of an ELF object or process image need to be fixed up or modified at linking or runtime (with addend) (Type `SHT_RELA`).
- `.dynamic` : Dynamic linking structures and objects. Contains a table of `ElfN_Dyn` structures. Also contains pointers to other important information required by the dynamic linker (for instance, the dynamic string table, dynamic symbol table, `.got.plt` section, and dynamic relocation section pointed to by tags of type `DT_STRTAB`, `DT_SYMTAB`, `DT_PLTGOT`, and `DT_RELA`, respectively

- `.init_array` : Contains an array of pointers to functions to use as constructors (each of these functions is called in turn when the binary is initialized). In `gcc` , you can mark functions in your C source files as constructors by decorating them with `__attribute__((constructor)`. By default, there is an entry in `.init_array` for executing `frame_dummy` .
- `.fini_array` : Contains an array of pointers to functions to use as destructors.
- `.shstrtab` : Is simply an array of NULL-terminated strings that contain the names of all the sections in the binary.
- `.symtab` : Contains a symbol table, which is a table of `ElfN_Sym` structures, each of which associates a symbolic name with a piece of code or data elsewhere in the binary, such as a function or variable.
- `.strtab` : Contains strings containing the symbolic names. These strings are pointed to by the `ElfN_Sym` structures.
- `.dynsym` : Same as `.symtab` but contains symbols needed for dynamic-linking rather than static-linking.
- `.dynstr` : Same as `.strtab` but contains strings needed for dynamic-linking rather than static-linking.
- `.interp` : RTLD embedded string.
- `.rel.dyn` : Global variable relocation table.
- `.rel.plt` : Function relocation table.

Older `gcc` version sections:

- `.ctors` : Equivalent of `.init_array` produced by older versions of `gcc` .
- `.dtors` : Equivalent of `.fini_array` produced by older versions of `gcc` .

Program Headers (Phdr)

The program header table provides a segment view of the binary, as opposed to the section view provided by the section header table. The section view of an ELF binary, is meant for static-linking purposes only.

In contrast, the segment view, is used by the operating system and dynamic-linker when loading an ELF into a process for execution to locate the relevant code and data and decide what to load into virtual memory.

Segments provide an execution view, they are needed only for executable ELF files and not for nonexecutable files such as relocatable objects.

32-bit struct:

```
typedef struct
{
  Elf32_Word    p_type;                 /* Segment type */
  Elf32_Off     p_offset;               /* Segment file offset */
  Elf32_Addr    p_vaddr;                /* Segment virtual address */
  Elf32_Addr    p_paddr;                /* Segment physical address */
  Elf32_Word    p_filesz;               /* Segment size in file */
  Elf32_Word    p_memsz;                /* Segment size in memory */
  Elf32_Word    p_flags;                /* Segment flags */
  Elf32_Word    p_align;                /* Segment alignment */
} Elf32_Phdr;
```

64-bit struct:

```
typedef struct
{
  Elf64_Word    p_type;                 /* Segment type */
  Elf64_Word    p_flags;                /* Segment flags */
  Elf64_Off     p_offset;               /* Segment file offset */
  Elf64_Addr    p_vaddr;                /* Segment virtual address */
  Elf64_Addr    p_paddr;                /* Segment physical address */
  Elf64_Xword   p_filesz;               /* Segment size in file */
  Elf64_Xword   p_memsz;                /* Segment size in memory */
  Elf64_Xword   p_align;                /* Segment alignment */
} Elf64_Phdr;
```

Values:

- `p_type` : Type of segment.
  - `PT_NULL` : Program header table entry unused (usually first entry of Program Header Table).
  - `PT_LOAD` : Loadable program segment.
  - `PT_DYNAMIC` : Dynamic linking information (holds the `.dynamic` section).
  - `PT_INTERP` : Program interpreter (holds `.interp` section).
  - `PT_GNU_EH_FRAME` : This is a sorted queue used by the GNU C compiler (gcc). It stores exception handlers. So when something goes wrong, it can use this area to deal correctly with it.
  - `PT_GNU_STACK` : This header is used to store stack information.
- `p_flags` : Flags that defines permissions of the segment in memory.
  - `PF_X` : Segment is executable.
  - `PF_W` : Segment is writable.
  - `PF_R` : Segment is readable.
- `p_offset` : Offset of ELF file to the segment.

- `p_vaddr` : Segment virtual address (for loadable segments, `p_vaddr` must be equal to `p_offset` , modulo the page size (which is typically 4,096 bytes).
- `p_paddr` : Segment physical address (on some systems, it is possible to use the p_paddr field to specify at which address in physical memory to load the segment. On modern operating systems such as Linux, this field is unused and set to zero since they execute all binaries in virtual memory).
- `p_filesz` : Segment size in disk (in bytes).
- `p_memsz` : Segment size in memory (in bytes). (some sections only indicate the need to allocate some bytes in memory but do not actually occupy these bytes in the binary file, such as `.bss`).
- `p_align` : Segment alignment (is analogous to the `sh_addralign` field in a section header).

---- type defines ----

p_type defines:

```
#define PT_NULL         0               /* Program header table entry
unused */
#define PT_LOAD         1               /* Loadable program segment */
#define PT_DYNAMIC      2               /* Dynamic linking information
*/
#define PT_INTERP       3               /* Program interpreter */
#define PT_NOTE         4               /* Auxiliary information */
#define PT_SHLIB        5               /* Reserved */
#define PT_PHDR         6               /* Entry for header table itself
*/
#define PT_TLS          7               /* Thread-local storage segment
*/
#define PT_NUM          8               /* Number of defined types */
#define PT_LOOS         0x60000000      /* Start of OS-specific */
#define PT_GNU_EH_FRAME 0x6474e550      /* GCC .eh_frame_hdr segment */
#define PT_GNU_STACK    0x6474e551      /* Indicates stack executability
*/
#define PT_GNU_RELRO    0x6474e552      /* Read-only after relocation */
#define PT_LOSUNW       0x6ffffffa
#define PT_SUNWBSS      0x6ffffffa      /* Sun Specific segment */
#define PT_SUNWSTACK    0x6ffffffb      /* Stack segment */
#define PT_HISUNW       0x6fffffff
#define PT_HIOS         0x6fffffff      /* End of OS-specific */
#define PT_LOPROC       0x70000000      /* Start of processor-specific
*/
#define PT_HIPROC       0x7fffffff      /* End of processor-specific */
```

p_flags defines:

```
#define PF_X            (1 << 0)        /* Segment is executable */
#define PF_W            (1 << 1)        /* Segment is writable */
#define PF_R            (1 << 2)        /* Segment is readable */
#define PF_MASKOS       0x0ff00000      /* OS-specific */
#define PF_MASKPROC     0xf0000000      /* Processor-specific */
```

Segments

Division of segments / sections:

- Text Segment
  - `.text`
  - `.rodata`
  - `.hash`
  - `.dynsym`
  - `.dynstr`
  - `.plt`
  - `.rel.got`
- Data segment
  - `.data`
  - `.dynamic`
  - `.got.plt`
  - `.bss`

Symbols

Symbols are a symbolic reference to some type of data or code such as a global variable or function.

32-bit struct:

```
typedef struct
{
  Elf32_Word    st_name;                /* Symbol name (string tbl
index) */
  Elf32_Addr    st_value;               /* Symbol value */
  Elf32_Word    st_size;                /* Symbol size */
  unsigned char st_info;                /* Symbol type and binding */
  unsigned char st_other;               /* Symbol visibility */
  Elf32_Section st_shndx;               /* Section index */
} Elf32_Sym;
```

64-bit struct:

```
typedef struct
{
  Elf64_Word    st_name;                /* Symbol name (string tbl
index) */
  unsigned char st_info;                /* Symbol type and binding */
  unsigned char st_other;               /* Symbol visibility */
  Elf64_Section st_shndx;               /* Section index */
  Elf64_Addr    st_value;               /* Symbol value */
  Elf64_Xword   st_size;                /* Symbol size */
} Elf64_Sym;
```

Values:

- `st_name` : Symbol name.
- `st_info` : Symbol type and binding. It is calculated using macros.
- `st_other` : Symbol visibility.
  - `STV_DEFAULT` : For default visibility symbols, its attribute is specified by the symbol's binding type.
  - `STV_PROTECTED` : Symbol is visible by other objects, but cannot be preempted.
  - `STV_HIDDEN` : Symbol is not visible to other objects.
  - `STV_INTERNAL` : Symbol visibility is reserved.
- `st_shndx` : Section index.
- `st_value` : Symbol value.
- `st_size` : Symbol size.

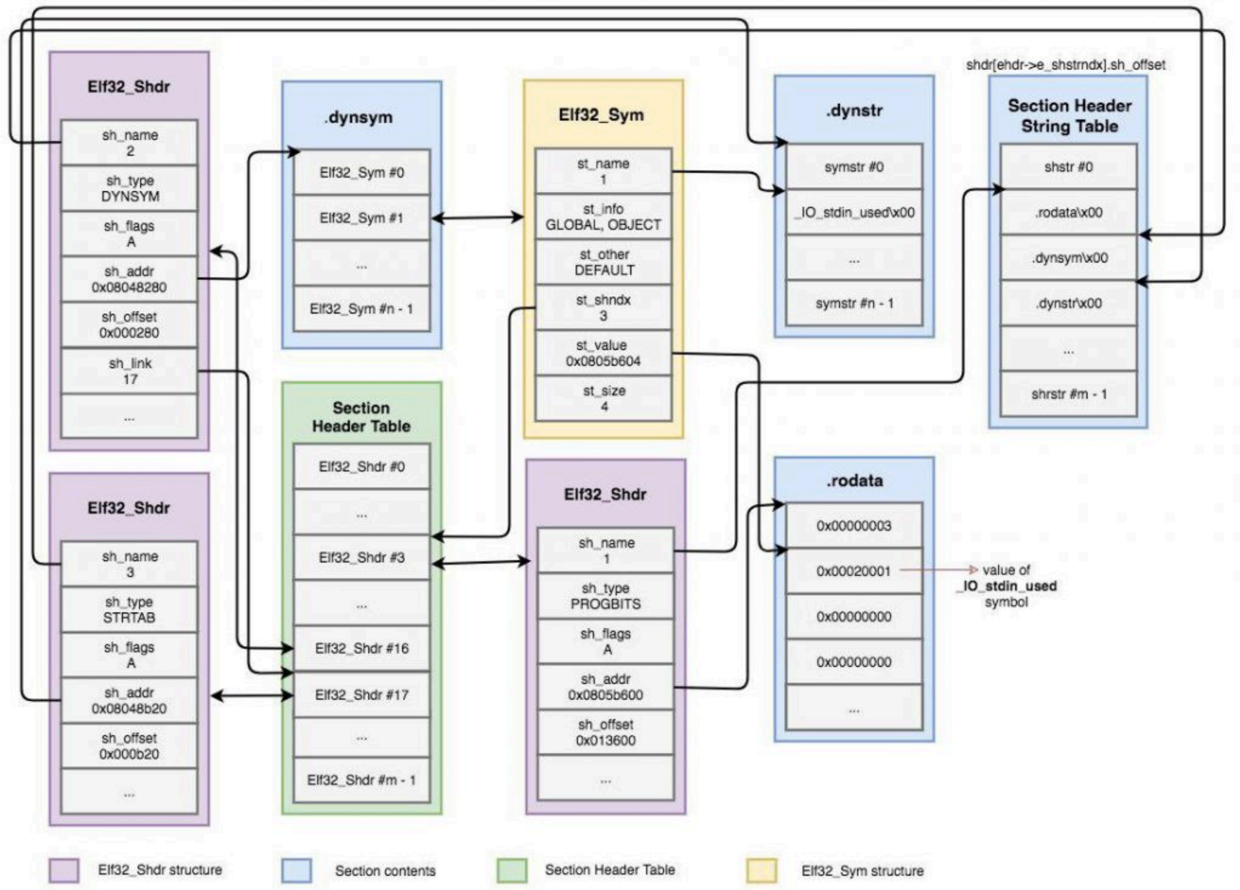
st_info Values:

- `st_bind` : Symbol binding.
  - `STB_LOCAL` : Local symbols are not visible outside the object file containing their definition, such as a function declared static.
  - `STB_GLOBAL` : Global symbols are visible to all object files being combined.
  - `STB_WEAK` : Similar to global binding, but with less precedence, meaning that the binding is weak and may be overridden by another symbol (with the same name) that is not marked as `STB_WEAK` .
- `st_type` : Symbol type.
  - `STT_NOTYPE` : The symbols type is undefined.
  - `STT_FUNC` : The symbol is associated with a function or other executable code.
  - `STT_OBJECT` : The symbol is associated with a data object.

- `STT_SECTION` : The symbol is a section.

Macros:

- `ELFN_ST_BIND(st_info)` : Get `st_bind` value given `st_info` .
- `ELFN_ST_TYPE(st_info)` : Get `st_type` value given `st_info` .
- `ELFN_ST_INFO(st_bind, st_type)` : Get `st_info` value given `st_type` and `st_bind` .



---- type defines ----

st_info macros:

```
#define ELF32_ST_BIND(val)          (((unsigned char) (val)) >> 4)
#define ELF32_ST_TYPE(val)          ((val) & 0xf)
#define ELF32_ST_INFO(bind, type)   (((bind) << 4) + ((type) & 0xf))

#define ELF64_ST_BIND(val)          ELF32_ST_BIND (val)
#define ELF64_ST_TYPE(val)          ELF32_ST_TYPE (val)
#define ELF64_ST_INFO(bind, type)   ELF32_ST_INFO ((bind), (type))
```

st_bind defines:

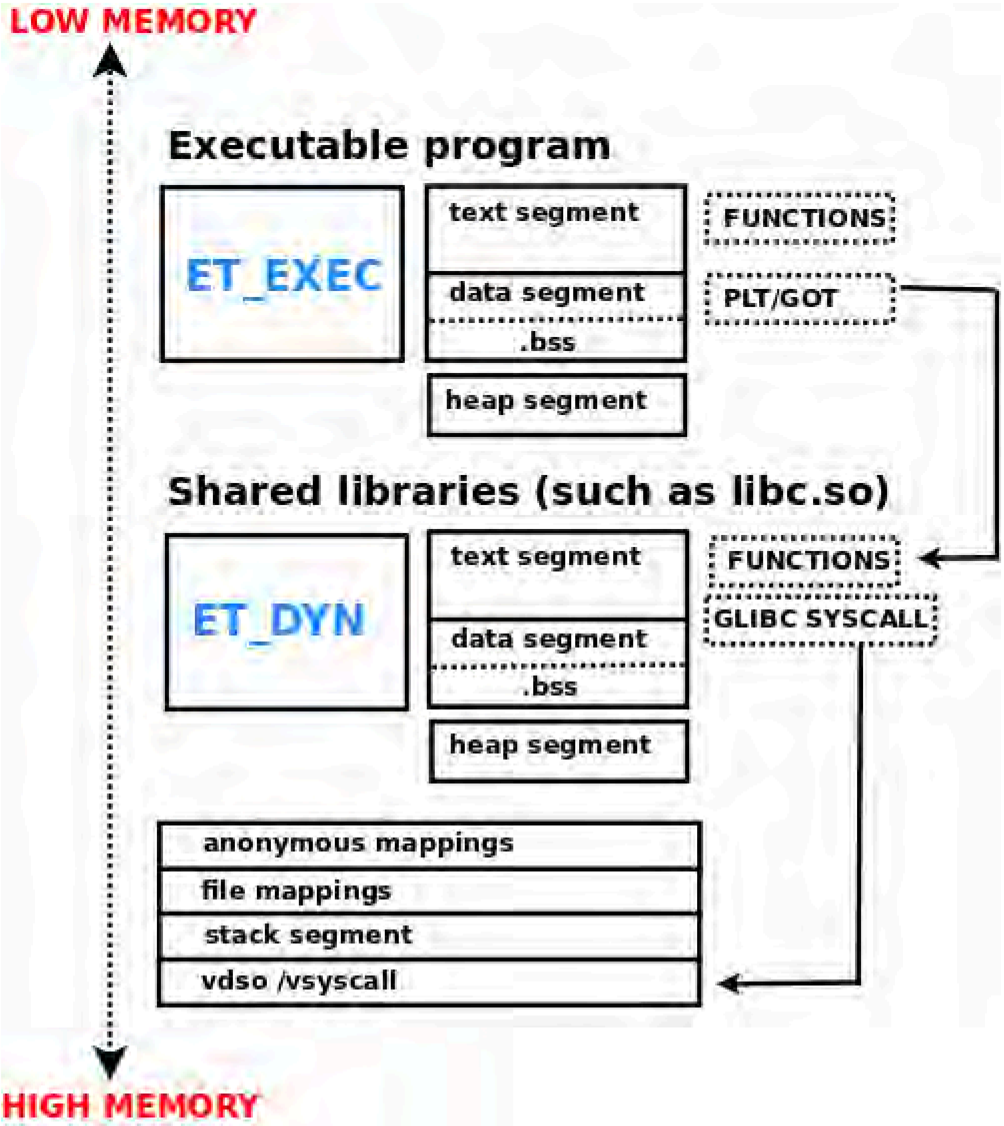
```
#define STB_LOCAL       0           /* Local symbol */
#define STB_GLOBAL      1           /* Global symbol */
```

```
#define STB_WEAK        2               /* Weak symbol */
#define STB_NUM         3               /* Number of defined types.  */
#define STB_LOOS        10              /* Start of OS-specific */
#define STB_GNU_UNIQUE  10              /* Unique symbol.  */
#define STB_HIOS        12              /* End of OS-specific */
#define STB_LOPROC      13              /* Start of processor-specific
*/
#define STB_HIPROC      15              /* End of processor-specific */
```

st_type defines:

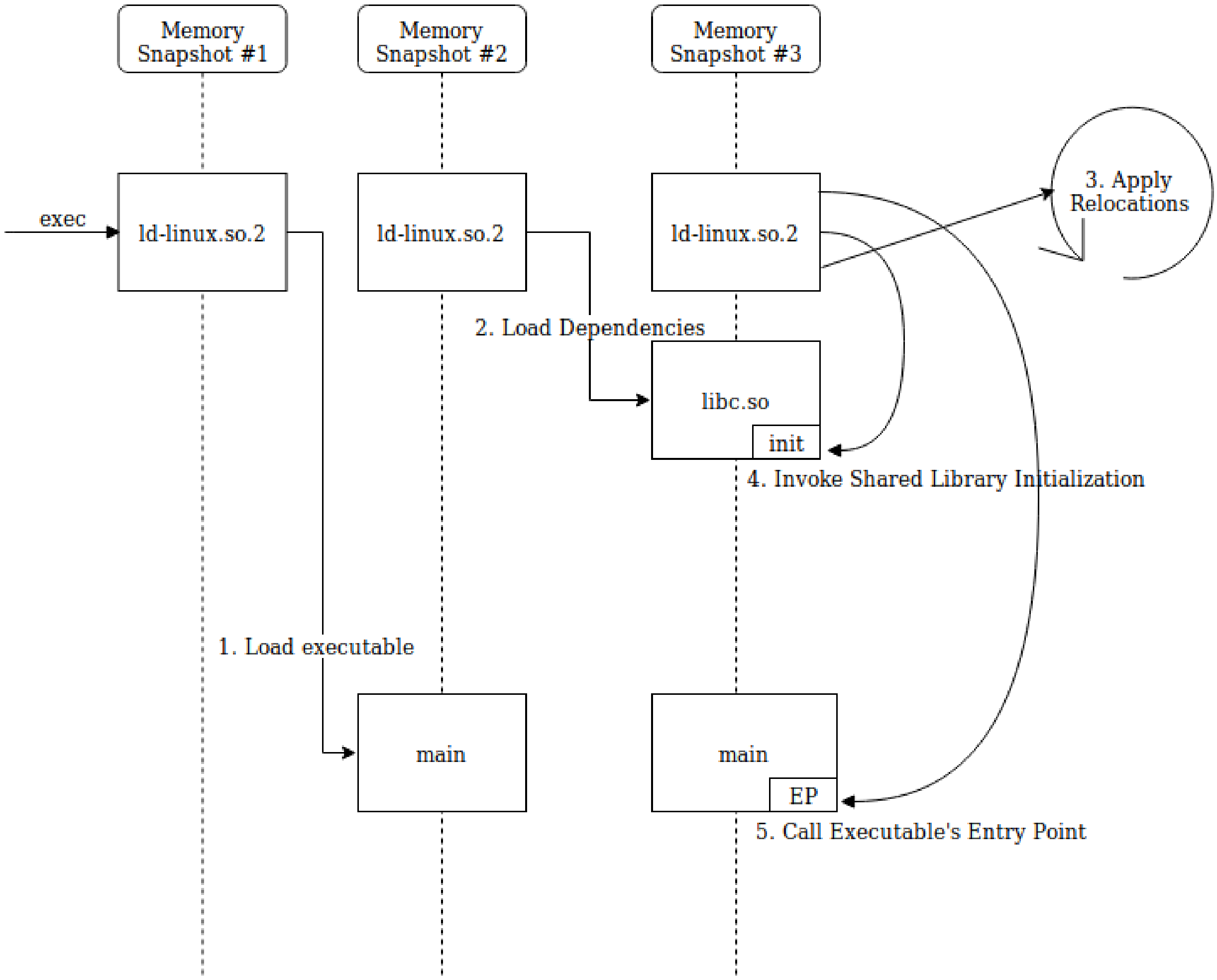
```
#define STT_NOTYPE      0               /* Symbol type is unspecified */
#define STT_OBJECT      1               /* Symbol is a data object */
#define STT_FUNC        2               /* Symbol is a code object */
#define STT_SECTION     3               /* Symbol associated with a
section */
#define STT_FILE        4               /* Symbol's name is file name */
#define STT_COMMON      5               /* Symbol is a common data
object */
#define STT_TLS         6               /* Symbol is thread-local data
object*/
#define STT_NUM         7               /* Number of defined types.  */
#define STT_LOOS        10              /* Start of OS-specific */
#define STT_GNU_IFUNC   10              /* Symbol is indirect code
object */
#define STT_HIOS        12              /* End of OS-specific */
#define STT_LOPROC      13              /* Start of processor-specific
*/
#define STT_HIPROC      15              /* End of processor-specific */
```

Dynamic Linking



Dynamic linking is the process in which we resolve functions from external libraries (shared objects).

By default, lazy binding is used, which is resolving functions at the time they are called first, at next calls it will be saved in the GOT (GLobal offset table). Then the PLT entry just have to jmp onto the address contained in the GOT entry for that function.



We can avoid lazy binding using `LD_BIND_NOW` env var, or using `RELRO` (or `Relocation Read-Only`).

When an external function is called from the code, instead of the real function, the PLT entry for that function is called.

The PLT is code that uses the GOT to jump and resolve with the help of the linker the external functions.

There is a relocation needed for fgets which will be resolved by the linker, as the address resolved must be written somewhere, in the offset value, it points to the GOT entry, for `fgets()` . Then the linker once the function is resolved will write that address on it.

```
Offset      Info        Type            SymValue    SymName
...
0804a000   00000107   R_386_JUMP_SLOT   00000000    fgets
...
```

`0x0804a000` is the GOT entry for `fgets()` .

When a function like `fgets` is called first:

```
objdump -d ./prog
...
8048481: e8 da fe ff ff    call 0x8048360 <fgets@plt>
...
```

`fgets@plt` is called.

PLT entry:

```
...
08048360 <fgets@plt>:
/* A jmp into the GOT */
8048360:  ff 25 00 a0 04 08   jmp *0x804a000
8048366:  68 00 00 00 00      push $0x0
804836b:  e9 e0 ff ff ff      jmp  0x8048350 <_init+0x34>
...
```

In the first instruction it does an indirect jump to the address contained in the GOT entry for `fgets`.

The address contained in the GOT at that time is the next instruction of that jmp, so the `push 0x0` instruction gets executed, that pushes onto the stack the index at GOT where `fgets` is located, take care that the first 3 entries are reserved, so actually it would be the 4th.

Reserved GOT entries:

- `GOT[0]`: Contains an address that points to the dynamic segment of the executable, which is used by the dynamic linker for extracting dynamic linking-related information.
- `GOT[1]`: Contains the address of the `link_map` structure that is used by the dynamic linker to resolve symbols.
- `GOT[2]`: Contains the address to the dynamic linkers `_dl_runtime_resolve()` function that resolves the actual symbol address for the shared library function.

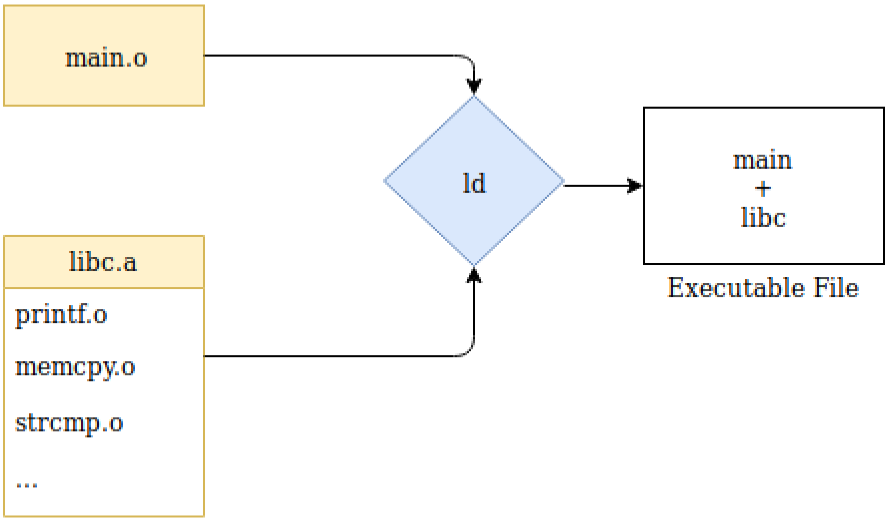
The last instruction in the `fgets()` PLT stub is a `jmp 0x8048350`. This address points to the very first PLT entry in every executable, known as PLT-0.

```
8048350: ff 35 f8 9f 04 08      pushl  0x8049ff8
8048356: ff 25 fc 9f 04 08      jmp   *0x8049ffc
804835c: 00 00                  add    %al,(%eax)
```

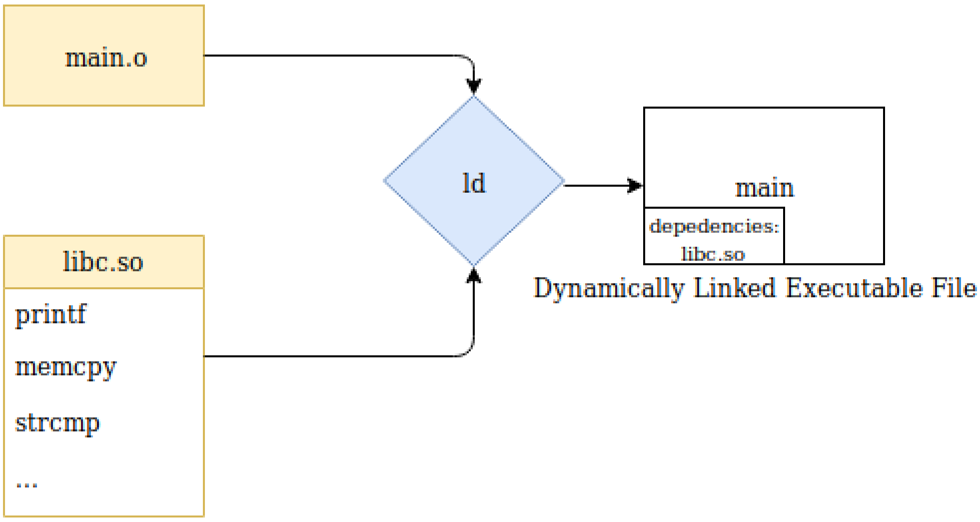
The first `pushl` instruction pushes the address of the second GOT entry, `GOT[1]`, onto the stack, which, as noted earlier, contains the address of the `link_map` structure.

The `jmp *0x8049ffc` performs an indirect jmp into the third GOT entry, `GOT[2]`, which contains the address to the dynamic linkers `_dl_runtime_resolve()` function, therefore transferring control to the dynamic linker and resolving the address for `fgets()`. Once `fgets()` has been resolved, all future calls to the PLT entry for `fgets()` will result in a jump to the `fgets()` code itself, rather than pointing back into the PLT and going through the lazy linking process again.

Static Linking:



Dynamic Linking:



Dynamic

32-bit struct:

```
typedef struct
{
  Elf32_Sword   d_tag;                  /* Dynamic entry type */
  union
    {
      Elf32_Word d_val;                 /* Integer value */
      Elf32_Addr d_ptr;                 /* Address value */
    } d_un;
} Elf32_Dyn;
```

64-bit struct:

```
typedef struct
{
  Elf64_Sxword  d_tag;                  /* Dynamic entry type */
  union
    {
      Elf64_Xword d_val;                /* Integer value */
      Elf64_Addr d_ptr;                 /* Address value */
    } d_un;
} Elf64_Dyn;
```

Values:

- `d_tag` : Contains a tag.
  - `DT_NEEDED` : Holds the string table offset to the name of a needed shared library.
  - `DT_SYMTAB` : Contains the address of the dynamic symbol table also known by its section name `.dynsym` .
  - `DT_HASH` : Holds the address of the symbol hash table, also known by it's section name `.hash` (or sometimes named `.gnu.hash`).
  - `DT_STRTAB` : Holds the address of the symbol string table, also known by its section name `.dynstr` .
  - `DT_PLTGOT` : Holds the address of the global offset table.
- `d_val` : Holds an integer value that has various interpretations such as being the size of a relocation entry to give one instance.
- `d_ptr` : Holds a virtual memory address that can point to various locations needed by the linker; a good example would be the address to the symbol table for the d_tag `DT_SYMTAB` .

---- type defines ----

d_tag defines:

```
#define DT_NULL         0               /* Marks end of dynamic section
*/
#define DT_NEEDED       1               /* Name of needed library */
#define DT_PLTRELSZ     2               /* Size in bytes of PLT relocs
*/
#define DT_PLTGOT       3               /* Processor defined value */
#define DT_HASH         4               /* Address of symbol hash table
*/
#define DT_STRTAB       5               /* Address of string table */
#define DT_SYMTAB       6               /* Address of symbol table */
#define DT_RELA         7               /* Address of Rela relocs */
#define DT_RELASZ       8               /* Total size of Rela relocs */
#define DT_RELAENT      9               /* Size of one Rela reloc */
#define DT_STRSZ        10              /* Size of string table */
#define DT_SYMENT       11              /* Size of one symbol table
entry */
#define DT_INIT         12              /* Address of init function */
#define DT_FINI         13              /* Address of termination
function */
#define DT_SONAME       14              /* Name of shared object */
#define DT_RPATH        15              /* Library search path
(deprecated) */
#define DT_SYMBOLIC     16              /* Start symbol search here */
#define DT_REL          17              /* Address of Rel relocs */
#define DT_RELSZ        18              /* Total size of Rel relocs */
#define DT_RELENT       19              /* Size of one Rel reloc */
#define DT_PLTREL       20              /* Type of reloc in PLT */
#define DT_DEBUG        21              /* For debugging; unspecified */
#define DT_TEXTREL      22              /* Reloc might modify .text */
#define DT_JMPREL       23              /* Address of PLT relocs */
#define DT_BIND_NOW     24              /* Process relocations of object
*/
#define DT_INIT_ARRAY   25              /* Array with addresses of init
fct */
#define DT_FINI_ARRAY   26              /* Array with addresses of fini
fct */
#define DT_INIT_ARRAYSZ 27              /* Size in bytes of
DT_INIT_ARRAY */
#define DT_FINI_ARRAYSZ 28              /* Size in bytes of
DT_FINI_ARRAY */
#define DT_RUNPATH      29              /* Library search path */
#define DT_FLAGS        30              /* Flags for the object being
loaded */
#define DT_ENCODING     32              /* Start of encoded range */
#define DT_PREINIT_ARRAY 32             /* Array with addresses of
preinit fct*/
#define DT_PREINIT_ARRAYSZ 33           /* size in bytes of
DT_PREINIT_ARRAY */
#define DT_SYMTAB_SHNDX 34              /* Address of SYMTAB_SHNDX
section */
#define DT_NUM          35              /* Number used */
#define DT_LOOS         0x6000000d      /* Start of OS-specific */
```

```
#define DT_HIOS         0x6ffff000      /* End of OS-specific */
#define DT_LOPROC       0x70000000      /* Start of processor-specific
*/
#define DT_HIPROC       0x7fffffff      /* End of processor-specific */
#define DT_PROCNUM      DT_MIPS_NUM     /* Most used by any processor */
```

Relocation

Relocation is the process of connecting symbolic references with symbolic definitions. Relocatable files must have information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. Relocation entries are these data.

Rel 32-bit struct:

```
typedef struct
{
  Elf32_Addr    r_offset;               /* Address */
  Elf32_Word    r_info;                 /* Relocation type and symbol
index */
} Elf32_Rel;
```

Rel 64-bit struct:

```
typedef struct
{
  Elf64_Addr    r_offset;               /* Address */
  Elf64_Xword   r_info;                 /* Relocation type and symbol
index */
} Elf64_Rel;
```

Rela 32-bit struct:

```
typedef struct
{
  Elf32_Addr    r_offset;               /* Address */
  Elf32_Word    r_info;                 /* Relocation type and symbol
index */
  Elf32_Sword   r_addend;               /* Addend */
} Elf32_Rela;
```

Rela 64-bit struct:

```
typedef struct
{
  Elf64_Addr    r_offset;               /* Address */
  Elf64_Xword   r_info;                 /* Relocation type and symbol
index */
  Elf64_Sxword  r_addend;               /* Addend */
} Elf64_Rela;
```

Values:

- `r_offset` : Points to the location that requires the relocation action.
  - For `ET_REL` type binaries, this value denotes an offset within a section header. in which the relocations have to take place.
  - For `ET_EXEC` type binaries, this value denotes a virtual address affected by a relocation.
- `r_info` : Gives both the symbol table index with respect to which the relocation must be made and the type of relocation to apply.
- `r_addend` : Specifies a constant addend used to compute the value stored in the relocatable field.

x86 Relocation types:

Name	Value	Field	Calculation
R_386_NONE	0	None	None
R_386_32	2	dword	S + A
R_386_PC32	1	dword	S + A – P
R_386_GOT32	3	dword	G + A
R_386_PLT32	4	dword	L + A – P
R_386_COPY	5	None	Value is copied directly from shared object
R_386_GLOB_DAT	6	dword	S
R_386_JMP_SLOT	7	dword	S
R_386_RELATIVE	8	dword	B + A
R_386_GOTOFF	9	dword	S + A – GOT
R_386_GOTPC	10	dword	GOT + A – P
R_386_32PLT	11	dword	L + A
R_386_16	20	word	S + A
R_386_PC16	21	word	S + A – P
R_386_8	22	byte	S + A
R_386_PC8	23	byte	S + A – P
R_386_SIZE32	38	dword	z + A

x86_64 Relocation types:

Name	Value	Field	Calculation
R_X86_64_NONE	0	None	None
R_X86_64_64	1	qword	S + A
R_X86_64_PC32	2	dword	S + A – P
R_X86_64_GOT32	3	dword	G + A
R_X86_64_PLT32	4	dword	L + A – P
R_X86_64_COPY	5	None	Value is copied directly from shared object
R_X86_64_GLOB_DAT	6	qword	S
R_X86_64_JUMP_SLOT	7	qword	S
R_X86_64_RELATIVE	8	qword	B + A
R_X86_64_GOTPCREL	9	dword	G + GOT + A – P
R_X86_64_32	10	dword	S + A
R_X86_64_32S	11	dword	S + A
R_X86_64_16	12	word	S + A
R_X86_64_PC16	13	word	S + A – P
R_X86_64_8	14	word8	S + A
R_X86_64_PC8	15	word8	S + A – P
R_X86_64_PC64	24	qword	S + A – P
R_X86_64_GOTOFF64	25	qword	S + A – GOT
R_X86_64_GOTPC32	26	dword	GOT + A – P
R_X86_64_SIZE32	32	dword	Z + A
R_X86_64_SIZE64	33	qword	Z + A

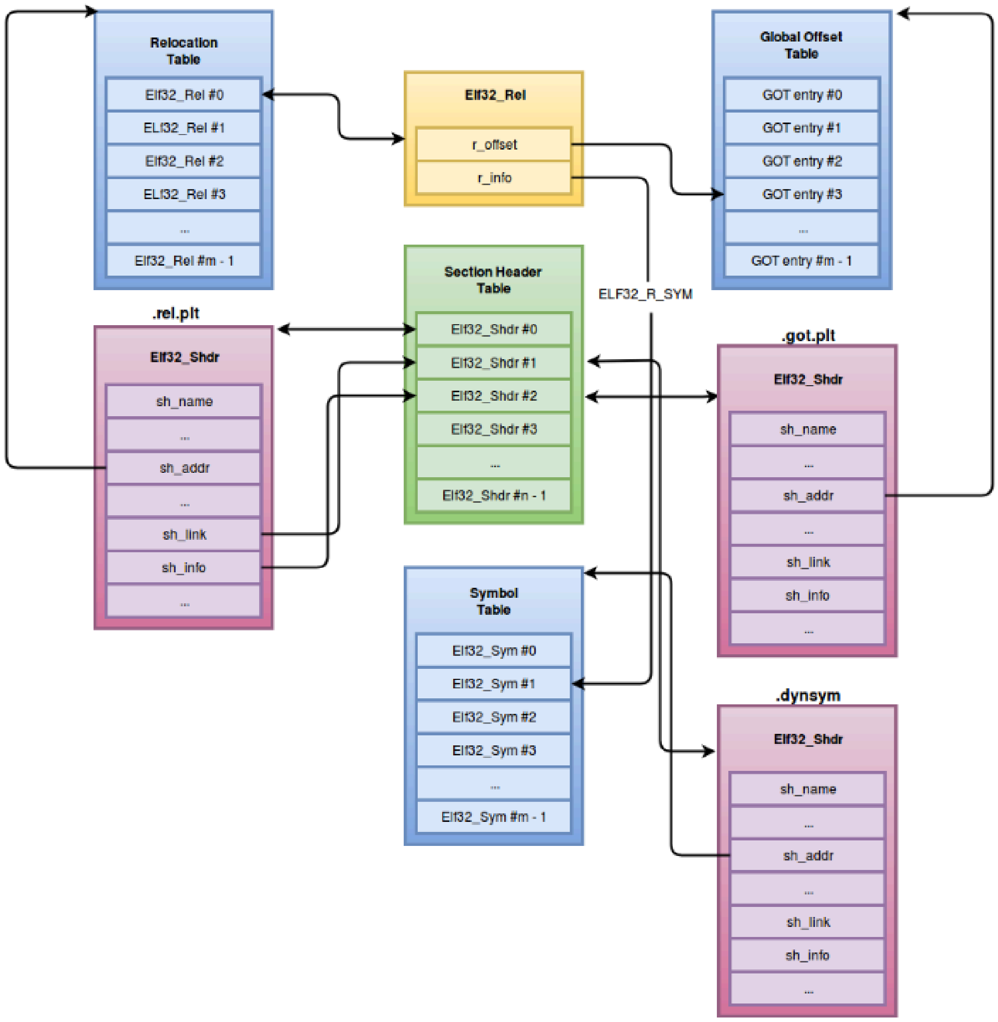
Values:

- `A` : This means the addend used to compute the value of the relocatable field.
- `B` : This means the base address at which a shared object has been loaded into memory during execution. Generally, a shared object file is built with a 0 base virtual address, but the execution address will be different.
- `G` : This means the offset into the global offset table at which the address of the relocation entry's symbol will reside during execution.
- `GOT` : This means the address of the global offset table.

- `L` : This means the place (section offset or address) of the procedure linkage table entry for a symbol. A procedure linkage table entry redirects a function call to the proper destination. The link editor builds the initial procedure linkage table, and the dynamic linker modifies the entries during execution.
- `P` : This means the place (section offset or address) of the storage unit being relocated (computed using `r_offset`).
- `S` : This means the value of the symbol whose index resides in the relocation entry.

Generic relocation suffixes:

- `_NONE` : Neglected entry.
- `_64` : qword relocation value.
- `_32` : dword relocation value.
- `_16` : word relocation value.
- `_8` : byte relocation value.
- `_PC` : relative to program counter.
- `_GOT` : relative to GOT.
- `_PLT` : relative to PLT (Procedure Linkage Table).
- `_COPY` : value copied directly from shared object at load-time.
- `_GLOB_DAT` : global variable.
- `_JMP_SLOT` : PLT entry.
- `_RELATIVE` : relative to image base of program's image.
- `_GOTOFF` : absolute address within GOT.
- `_GOTPC` : program counter relative GOT offset.



Sections:

- `.rel.bss` : Contains all the `R_386_COPY` relocs.
- `.rel.plt` : Contains all the `R_386_JMP_SLOT` relocs these modify the first half of the GOT elements.
- `.rel.got` : Contains all the `R_386_GLOB_DATA` relocs these modify the second half of the GOT elements.
- `.rel.data` : Contains all the `R_386_32` and `R_386_RELATIVE` relocs.
- `.rela.dyn` : Contains dynamic relocations for variables.
- `.rela.plt` : Contains dynamic relocations for functions.

Stripped binaries

Stripped binaries are those that it's symbols got removed.

Symbols in general are not needed by the loader to load an ELF executable, except from the dynamic linking ones.

They generally are used for debugging purposes, and they make the reverse engineering task easier as they give function names and a lot of information about an ELF file structure.

But, as dynamic symbols are still present, you can view the imported functions from external libraries like glibc.

Differences between 32-bit and 64-bit ELF objects

The main differences are:

- In the ELF header, the `e_machine` changes.
- The sizes of the values along the ELF file changes too.

Sections VS Segments

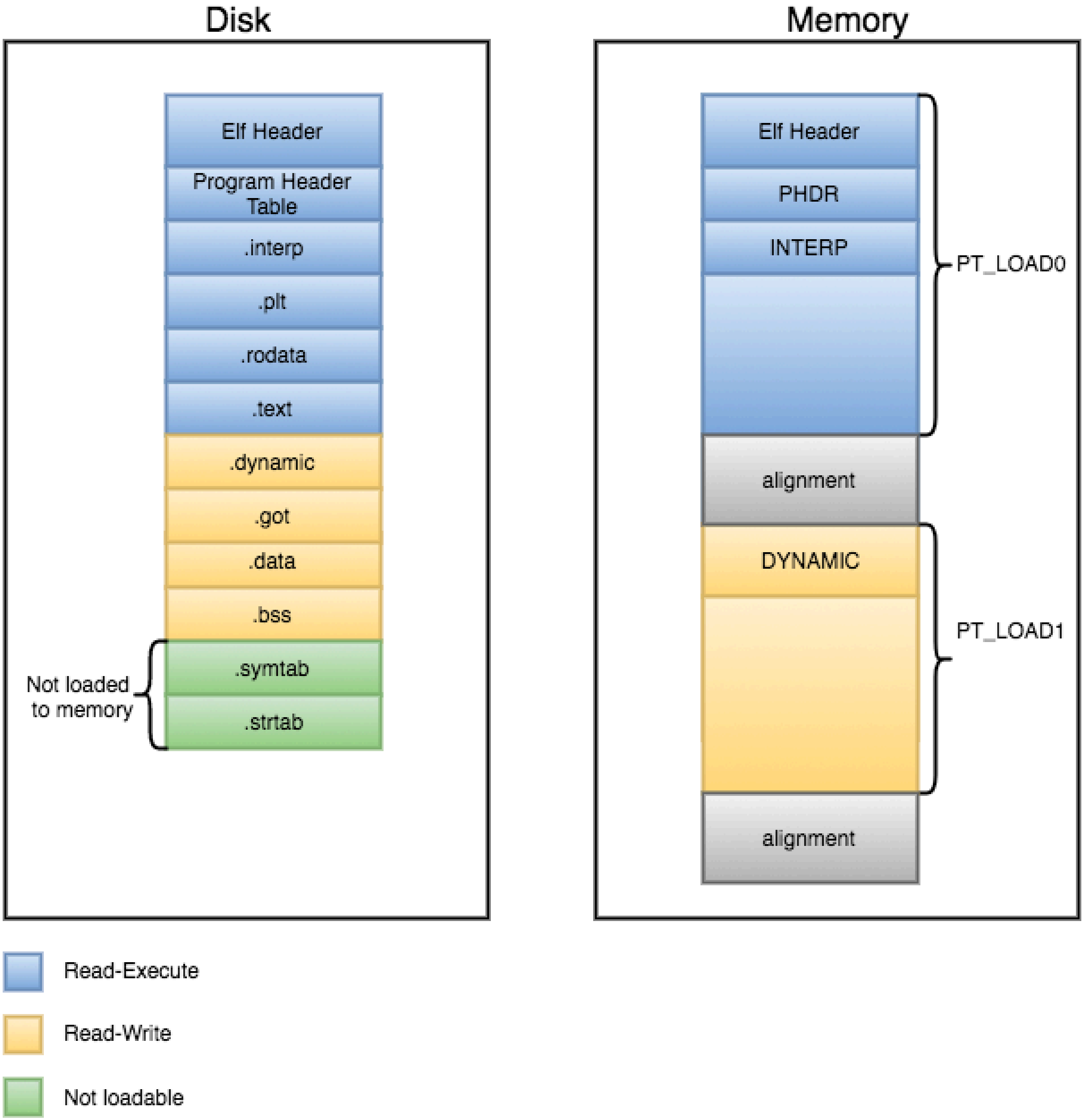
Segments are divided into sections, each section has an utility for the ELF file.

Sections per se, are not useful at runtime, so they are only useful at link time.

Segments are used for creating a block of memory, with some specific permissions and store there some content.

In contrast from other File formats, ELF files are composed of sections and segments. As previously mentioned, sections gather all needed information to link a given object file and build an executable, while Program Headers split the executable into segments with different attributes, which will eventually be loaded into memory.

In order to understand the relationship between Sections and Segments, we can picture segments as a tool to make the linux loader's life easier, as they group sections by attributes into single segments in order to make the loading process of the executable more efficient, instead of loading each individual section into memory. The following diagram attempts to illustrate this concept:



In-memory loaded ELF VS ELF file

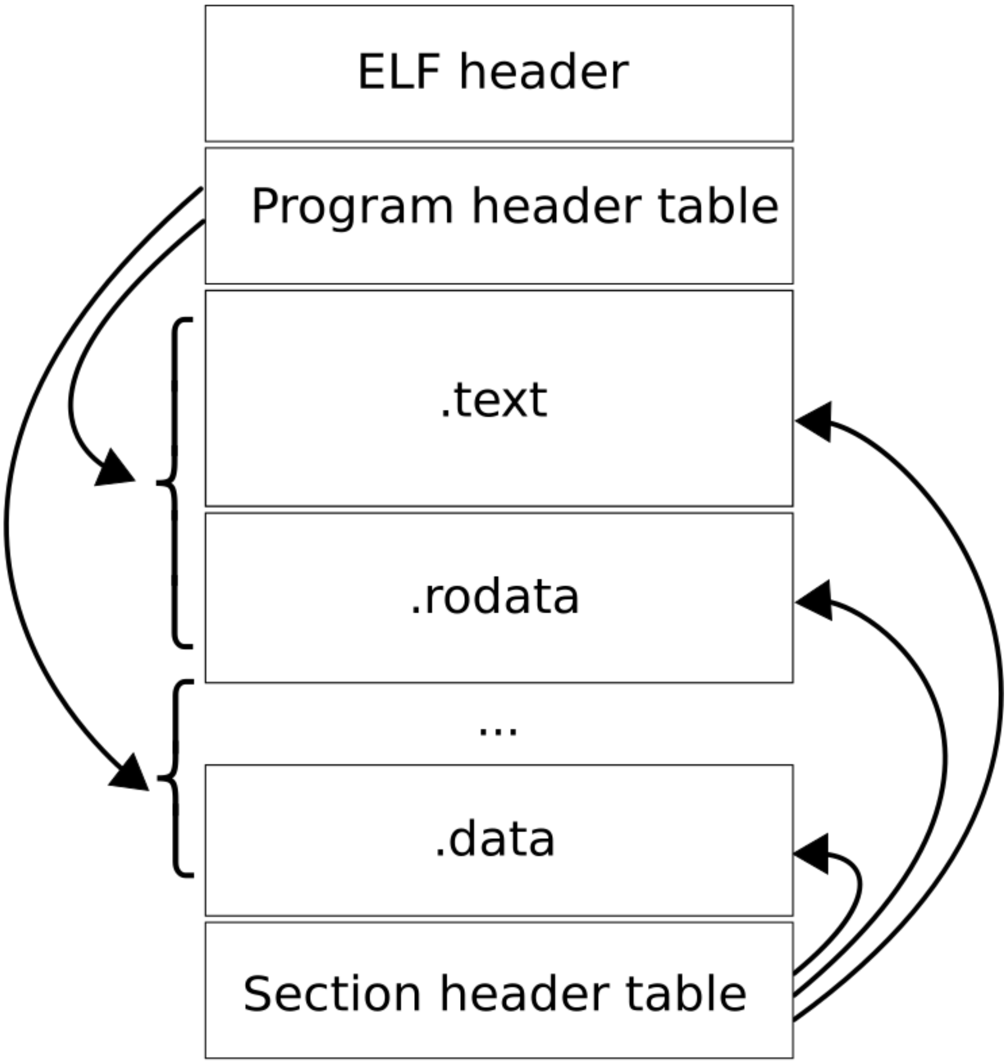
ELF files in disk are just a format that defines how to load it in memory to work fine.

In disk it specifies some not neccesary useful information such as .symtab, .strtab, they are not used at runtime and are there just for debugging purposes.

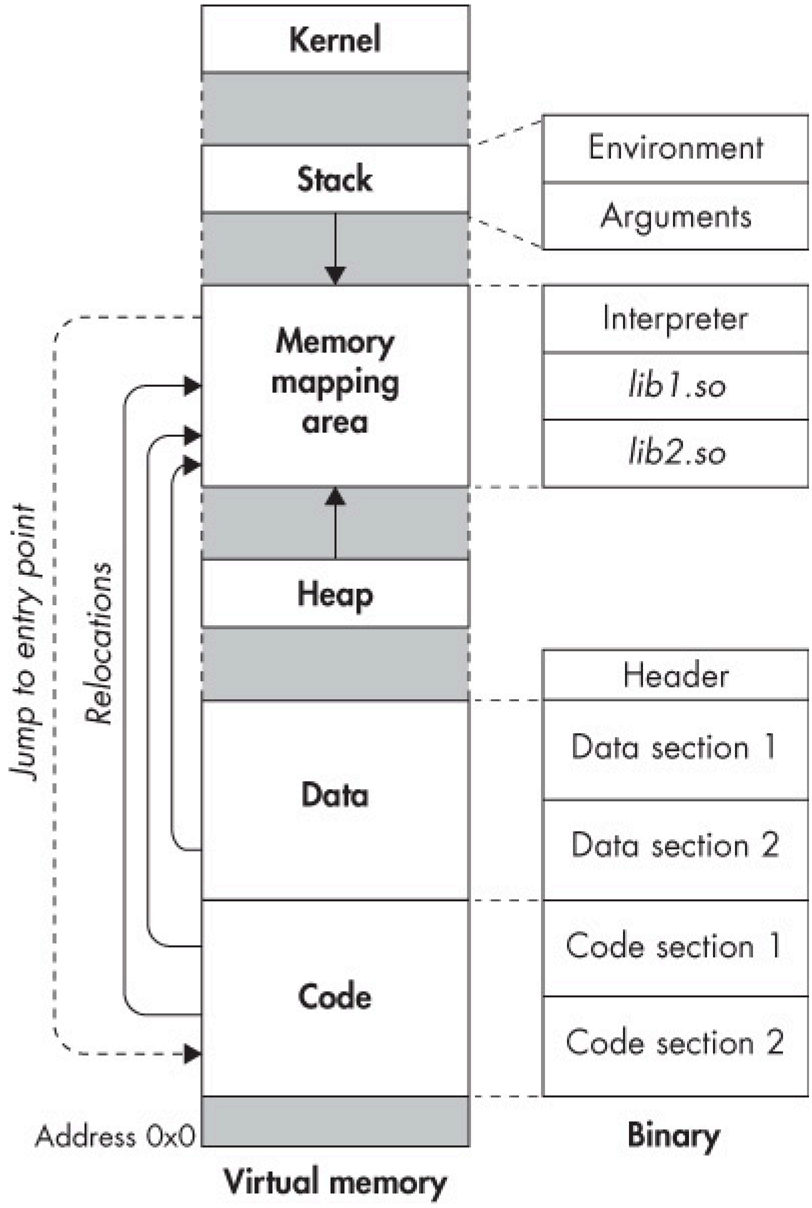
Size in memory is usually different than in disk, for example, someone can define uninitialized variables (stored at bss). In disk you just have to specify it's size without occupying that space. Once loaded in memory you have to fill that space somehow, for example with zeroes, so when loading the storage needed to allocate the ELF increases.

Basic overview:

ELF file in disk:



ELF loaded in memory:



Differences between ELF objects

Object Files

Object files are relocatable files, they are used to link them with another object files.

It provides information to the linker to, once it's time to link it to the rest of object files, allow the relocation and make it easier.

The object file content's is different from the other ELF files such as `ET_EXEC` and `ET_DYN`.

It usually have `.rela.text` and `.rela.eh_frame` sections.

As it is not a completely formed ELF yet, no specific sections has been created, therefore you will find just common code and data sections, and symbols.

Statically-linked executable files

Executable files are those that do not depend from external libraries, then no relocations should be pending for them as they can load without external objects.

They do not need `.dynamic` or the Dynamic segment, they do not need the GOT or PLT as function calls are done directly to the function address and without any intermediate.

Then in this type of ELF files you will find common code and data sections, and symbols (which can be removed).

As they are static, if they use libc functions the total size will be considerably long.

Dynamically-linked executable files

They are still executables, but as they are dynamically linked they are PIC (Process Independient Code).

They need GOT and PLT as intermediates to use external functions from shared-libraries such as `printf()`.

In this type of executables you will usually find common code and data sections, the GOT, the PLT, Dynamic-linking symbol sections such as `.dynsym` and `.dynstr` (As well as static symbols which are not needed).

You will also find the `.dynamic` section, which is crucial for dynamic linking, and `.rela.dyn`, `.rela.plt`.

Shared libraries

They get loaded in a process memory to provide functions to the executable which is going to use them.

They are similar to dynamically-linked executables, but not equal.

Here there is no `PT_INTERP` segment, as the shared-library is not loaded by the kernel but by the linker.

Also, local functions are included also in `.dynsym` (Not just in `.symtab`), and `__libc_start_main` is not imported.

The other structure is mostly the same as dynamically-linked executables.

Step-by-step ELF loading for each object type, ASLR and PIC/PIE

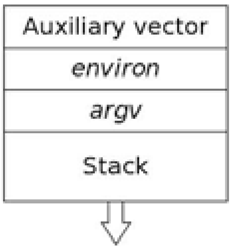
Relocatable files

They are not supposed to be loaded as some relocations are pending to create a fully working executable first.

Statically-linked executable files

First, when we decide to run an executable the kernel set up a process and give it a virtual memory space, an stack etc.

The stack for that process address space is set up in a very specific way to pass information to the dynamic linker. This particular setup and arrangement of information is known as the auxiliary vector or auxv.



Struct:

```
typedef struct
{
    uint64_t a_type;
    union
    {
        uint64_t a_val;
    } a_un;
} Elf64_auxv_t;
```

Auxv type:

```
/* Legal values for a_type (entry type).  */
#define AT_NULL         0               /* End of vector */
#define AT_IGNORE       1               /* Entry should be ignored */
#define AT_EXECFD       2               /* File descriptor of program */
#define AT_PHDR         3               /* Program headers for program
*/
#define AT_PHENT        4               /* Size of program header entry
*/
#define AT_PHNUM        5               /* Number of program headers */
```

```
#define AT_PAGESZ       6               /* System page size */
#define AT_BASE         7               /* Base address of interpreter
*/
#define AT_FLAGS        8               /* Flags */
#define AT_ENTRY        9               /* Entry point of program */
#define AT_NOTELF       10              /* Program is not ELF */
#define AT_UID          11              /* Real uid */
#define AT_EUID         12              /* Effective uid */
#define AT_GID          13              /* Real gid */
#define AT_EGID         14              /* Effective gid */
#define AT_CLKTCK       17              /* Frequency of times() */
/* Pointer to the global system page used for system calls and other
nice things.  */
#define AT_SYSINFO      32
#define AT_SYSINFO_EHDR 33
```

The auxiliary vector is a special structure that is for passing information directly from the kernel to the newly running program. It contains system specific information that may be required, such as the default size of a virtual memory page on the system or hardware capabilities; that is specific features that the kernel has identified the underlying hardware has that userspace programs can take advantage of.

Then the operating system maps an interpreter into the process's virtual memory (Usually `ld-linux.so`). Then reads the interpreter code and starts it from it's entry point. The interpreter can be retrieved by the `.interp` section in the ELF file.

The interpreter loads the binary, and gives the control to the entry point of the binary.

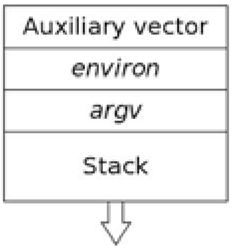
Summary:

- The kernel maps the program in memory (and the vDSO);
- The kernel sets up the stack and registers (passing information such as the argument and environment variables) and calls the main program entry point.
- The executable is loaded at a fixed address and no relocation is needed.

Dynamically-linked executable files

First, when we decide to run an executable the kernel set up a process and give it a virtual memory space, an stack etc.

The stack for that process address space is set up in a very specific way to pass information to the dynamic linker. This particular setup and arrangement of information is known as the auxiliary vector or auxv.




```
position            content                     size (bytes) + comment
  ------------------------------------------------------------------------
  stack pointer ->  [ argc = number of args ]     4
                    [ argv[0] (pointer) ]         4   (program name)
                    [ argv[1] (pointer) ]         4
                    [ argv[..] (pointer) ]        4 * x
                    [ argv[n - 1] (pointer) ]     4
                    [ argv[n] (pointer) ]         4   (= NULL)

                    [ envp[0] (pointer) ]         4
                    [ envp[1] (pointer) ]         4
                    [ envp[..] (pointer) ]        4
                    [ envp[term] (pointer) ]      4   (= NULL)

                    [ auxv[0] (Elf32_auxv_t) ]    8
                    [ auxv[1] (Elf32_auxv_t) ]    8
                    [ auxv[..] (Elf32_auxv_t) ]   8
                    [ auxv[term] (Elf32_auxv_t) ] 8   (= AT_NULL vector)

                    [ padding ]                   0 - 16

                    [ argument ASCIIZ strings ]   >= 0
                    [ environment ASCIIZ str. ]   >= 0

  (0xbffffffc)      [ end marker ]                4   (= NULL)

  (0xc0000000)      < bottom of stack >           0   (virtual)
  ------------------------------------------------------------------------
```

Sample view:

```
$LD_SHOW_AUXV=1 /bin/true
AT_SYSINFO_EHDR:      0x7ffff7fd3000
AT_HWCAP:             bfebfbff
AT_PAGESZ:            4096
AT_CLKTCK:            100
AT_PHDR:              0x555555554040
AT_PHENT:             56
AT_PHNUM:             11
AT_BASE:              0x7ffff7fd4000
AT_FLAGS:             0x0
AT_ENTRY:             0x555555556390
AT_UID:               1000
AT_EUID:              1000
AT_GID:               1000
AT_EGID:              1000
AT_SECURE:            0
AT_RANDOM:            0x7fffffffe3f9
AT_HWCAP2:            0x0
AT_EXECFN:            /bin/true
AT_PLATFORM:          x86_64
```

Struct:

```
typedef struct
{
    uint64_t a_type;
    union
    {
        uint64_t a_val;
    } a_un;
} Elf64_auxv_t;
```

Auxv type:

```
/* Legal values for a_type (entry type).  */
#define AT_NULL         0               /* End of vector */
#define AT_IGNORE       1               /* Entry should be ignored */
#define AT_EXECFD       2               /* File descriptor of program */
#define AT_PHDR         3               /* Program headers for program
*/
#define AT_PHENT        4               /* Size of program header entry
*/
#define AT_PHNUM        5               /* Number of program headers */
#define AT_PAGESZ       6               /* System page size */
#define AT_BASE         7               /* Base address of interpreter
*/
#define AT_FLAGS        8               /* Flags */
#define AT_ENTRY        9               /* Entry point of program */
```

```
#define AT_NOTELF       10              /* Program is not ELF */
#define AT_UID          11              /* Real uid */
#define AT_EUID         12              /* Effective uid */
#define AT_GID          13              /* Real gid */
#define AT_EGID         14              /* Effective gid */
#define AT_CLKTCK       17              /* Frequency of times() */
/* Pointer to the global system page used for system calls and other
nice things.  */
#define AT_SYSINFO      32
#define AT_SYSINFO_EHDR 33
```

The auxiliary vector is a special structure that is for passing information directly from the kernel to the newly running program. It contains system specific information that may be required, such as the default size of a virtual memory page on the system or hardware capabilities; that is specific features that the kernel has identified the underlying hardware has that userspace programs can take advantage of.

After the program code has been loaded into memory as described previously, the ELF handler also loads the ELF interpreter program into memory with `load_elf_interp()` . This process is similar to the process of loading the original program: the code checks the format information in the ELF header, reads in the ELF program header, maps all of the `PT_LOAD` segments from the file into the new program's memory, and leaves room for the interpreter's `BSS` segment. The interpreter can be retrieved by the `.interp` section in the ELF file.

The execution start address for the program is also set to be the entry point of the interpreter, rather than that of the program itself. When the `execve()` system call completes, execution then begins with the ELF interpreter, which takes care of satisfying the linkage requirements of the program from user space — finding and loading the shared libraries that the program depends on, and resolving the program's undefined symbols to the correct definitions in those libraries. Once this linkage process is done (which relies on a much deeper understanding of the ELF format than the kernel has), the interpreter can start the execution of the new program itself, at the address previously recorded in the `AT_ENTRY` auxiliary value.

We mentioned previously that system calls are slow, and modern systems have mechanisms to avoid the overheads of calling a trap to the processor.

In Linux, this is implemented by a neat trick between the dynamic loader and the kernel, all communicated with the `AUXV` structure. The kernel actually adds a small shared library into the address space of every newly created process which contains a function that makes system calls for you. The beauty of this system is that if the underlying hardware supports a fast system call mechanism the kernel (being the creator of the library) can use it, otherwise it can use the old scheme of generating a trap. This library is named linux-gate.so.1, so called because it is a gateway to the inner workings of the kernel.

When the kernel starts the dynamic linker it adds an entry to the auxv called AT_SYSINFO_EHDR, which is the address in memory that the special kernel library lives in. When the dynamic linker starts it can look for the AT_SYSINFO_EHDR pointer, and if found load that library for the program. The program has no idea this library exists; this is a private arrangement between the dynamic linker and the kernel.

The interpreter loads the binary, and parse it to find which libraries does the binary need, and maps them with mmap or similar options and then performs any necessary last-minute relocations in the binary's code sections to fill in the correct addresses for references to the dynamic libraries.

The dynamic linker will jump to the entry point address as given in the ELF binary.

The entry point is the `_start` function in the binary. At this point we can see in the disassembley some values are pushed onto the stack. The first value is the address of `__libc_csu_fini` function, another is the address of `__libc_csu_init` and then finally the address of `main()` function. After this the value `__libc_start_main` function is called.

At this stage we can see that the `__libc_start_main function` will receive quite a few input paramaters on the stack. Firstly it will have access to the program arguments, environment variables and auxiliary vector from the kernel. Then the initalization function will have pushed onto the stack addresses for functions to handle `init` , `fini` , and finally the address of the `main()` function itself.

The last value pushed onto the stack for the `__libc_start_main` was the initialisation function `__libc_csu_init` . If we follow the call chain through from `__libc_csu_init` we can see it does some setup and then calls the `_init` function in the executable. The `_init` function eventually calls some functions called `__do_global_ctors_aux` , `frame_dummy` and `call_gmon_start` .

Once `__libc_start_main` has completed with the `_init` call it finally calls the `main()` function. Remember that it had the stack setup initially with the arguments and environment pointers from the kernel; this is how main gets its `argc` , `argv[]` , `envp[]` arguments. The process now runs and the setup phase is complete.

Finally, call end functions and calls `exit()` with the return value from `main()` .

The linker's next work will be resolving with lazy binding all the library functions when they are called. Using the library's symbols and the dynamic symbols from you executable, and relocations for the GOT, the dynamic linking will be performed successfully.

Summary:

- locate and map all dependencies (as well as shared object specified in LD_PRELOAD);
- relocate the files.

This is a very high level overview as I understand it:

- the kernels initialises the process:
  - it maps the main program, the interpreter (dynamic linker) segments and the vDSO in the virtual address space;
  - it sets up the stack (passing the arguments, environment) and calls the dynamic linker entry point;
- the dynamic linker loads the different ELF objects and binds them together
  - it relocates itself (!);
  - it finds and loads the necessary libraries;
  - it does the relocations (which binds the ELF objects);
  - it calls the initialisation functions functions of the shared objects;
- Those functions are specified in the DT_INIT and DT_INIT_ARRAY entries of the ELF objects.
  - it calls the main program entry point;
  - The main program entry point is found in the AT_ENTRY entry of the auxiliary vector: it has been initialised by the kernel from the e_entry ELF header field.

   - the executable then initialises itself.

Shared libraries

As explained previously, they get loaded in the process memory space, and the linker does the dynamic-linking work.

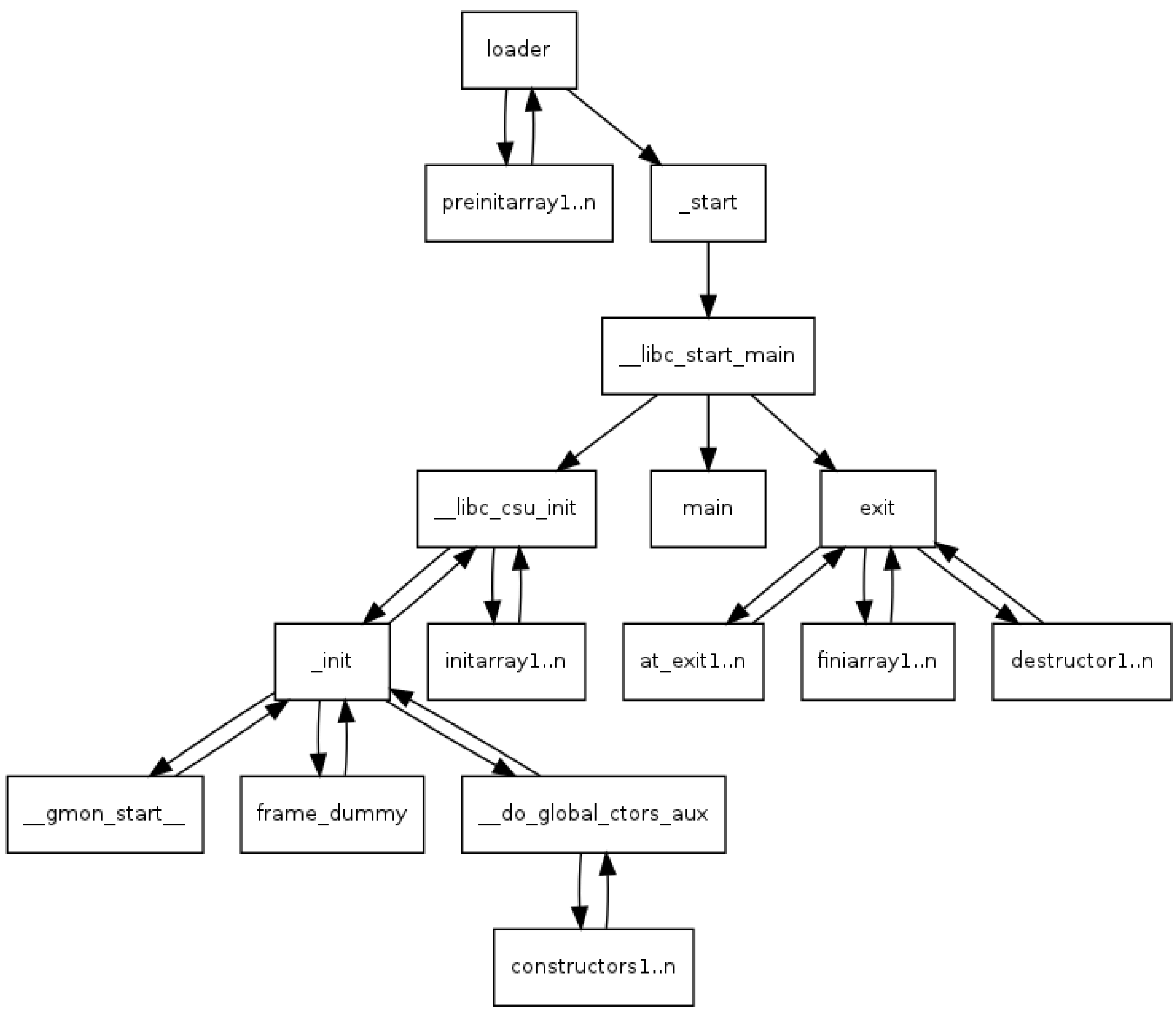
Common objects and functions

- `frame_dummy` : This function lives in the `.init` section. It is defined as `void frame_dummy ( void )` and its whole point in life is to call `__register_frame_info_bases` which has arguments.
- `_start` : This is where `e_entry` points to, and first code to be executed.
- `_init` : The dynamic loader executes the (INIT) function before control is passed _start function and executes the (FINI) function just before control is passed back to the OS kernel. The _init function is the default function used for the (INIT) tag. It calls several functions like `__gmon_start__` , `frame_dummy` , `__do_global_ctors_aux` .
- `_fini` : The dynamic loader executes the (FINI) function just before control is passed back to the OS kernel.
- `.init` : Code to be executed when the program starts.
- `.fini` : Code to be executed at the end of the program.
- `.init_array` : Array of pointers to use as constructors.
- `.fini_array` : Array of pointers to use as destructors.
- `__libc_start_main` : Libc functions that set up some stuff and calls `main()` .
- `deregister_tm_clones` : Transactional memory is intended to make programming with threads simpler. It is an alternative to lock-based synchronization. These routines tear down and setup, respectively, a table used by the library (libitm) which supports these functions.
- `register_tm_clones` : Transactional memory is intended to make programming with threads simpler. It is an alternative to lock-based synchronization. These routines tear down and setup, respectively, a table used by the library (libitm) which supports these functions.
- `__register_frame_info_bases` :
- `__stack_chk_fail` : Stack smashing Protector function.
- `__do_global_dtors_aux` : Runs all the global destructors on exit from the program on systems where `.fini_array` is not available.
- `__do_global_dtors_aux_fini_array_entry` and `__init_array_end` : These mark the end and start of the `.fini_array` section, which contains pointers to all the program-level finalizers.

- `__frame_dummy_init_array_entry` and `__init_array_start` : These mark the end and start of the `.init_array` section, which contains pointers to all the program-level initializers.
- `__libc_csu_init` : These run any program-level initializers (kind of like constructors for your whole program).
- `__libc_csu_fini` : These run any program-level finalizers (kind of like destructors for your whole program).
- `main` : For libc-linked programs, this is the default library being called by `__libc_start_main` and where the first user-custom code is executed.
- `.eh_frame` : DWARF-based debugging features such as stack unwinding.

Summary:

- `_start` calls the libc `__libc_start_main` ;
- `__libc_start_main` calls the executable `__libc_csu_init` (statically-linked part of the libc);
- `__libc_csu_init` calls the executable constructors (and other initialisatios);
- `__libc_start_main` calls the executable `main()` ;
- `__libc_start_main` calls the executable `exit()` .



FAQ (Frequently Asked Questions)

Why do we need sections?

Sections are there just to make the linker's work easier. For example, when you, in a relocation want to specify a relocation for `ET_REL` files, you specify the offset within that section.

How does the compiler make dynamically-linked executables (DT_NEEDED)?

When the compiler compiles for a dynamically-linked executable, instead of compiling it to a .a library and linking it statically, it creates in the `.dynamic` section specified by `DT_NEEDED` a string with the library name (Eg.: libc.so.6).

When the binary is executed on another system, the interpreter tries to find that library by name and load it to memory to start the dynamic-linking process.

When using PIC/PIE executables, how do the addresses get patched so the offset is added?

-- TO DO --

What is the difference between .got, .plt.got, .plt and .got.plt?

`.got` is for relocations regarding global 'variables' while `.got.plt` is an auxiliary section to act together with `.plt` when resolving procedures absolute addresses.

Where is mmap space located?

-- TO DO --

Where is ld loaded?

-- TO DO --

Where are needed libraries loaded?

-- TO DO --

What is the difference between Rel and Rela?

Rel is used in 32-bit systems, instead, Rela is used in 64-bit ones.

Rela, has an addend, Rel doesn't.

How is process address selected?

-- TO DO --

How does alignment work?

-- TO DO --

How are other segments included in PT_LOAD ones?

-- TO DO --

What happens if we include more than one shared-library?

-- TO DO --

What happens if A (program) which uses libc, imports also B (library) which also uses libc?

-- TO DO --

When a() (local) calls b() (libc) and b() calls c() (libc too) is c() imported in .dynsym?

-- TO DO --

References

- Practical Linux Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly By Dennis Andriesse
- Learning Linux Binary Analysis By Ryan "elfmaster" O'Neill
- https://web.stanford.edu/~ouster/cgi-bin/cs140-winter13/pintos/specs/sysv-abi-update.html/ch4.eheader.html
- https://hydrasky.com/malware-analysis/elf-file-chapter-2-relocation-and-dynamic-linking/
- https://www.intezer.com/blog/research/executable-linkable-format-101-part1-sections-segments/
- https://man7.org/linux/man-pages/man8/ld.so.8.html
- https://lwn.net/Articles/631631/
- https://codywu2010.wordpress.com/2014/11/29/about-elf-pie-pic-and-else/

- https://www.it-swarm-es.tech/es/c/que-funciones-agrega-gcc-al-linux-elf/822753373/
- https://gcc.gnu.org/onlinedocs/gccint/Initialization.html
- https://gcc.gnu.org/wiki/TransactionalMemory
- http://pmarlier.free.fr/gcc-tm-tut.html
- https://github.com/gcc-mirror/gcc/blob/master/libgcc/crtstuff.c
- https://www.bottomupcs.com/starting_a_process.xhtml
- https://www.gabriel.urdhr.fr/2015/01/22/elf-linking/
- https://web.archive.org/web/20191210114310/http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html
- `/usr/include/elf.h`
- `ELF(5)` man pages