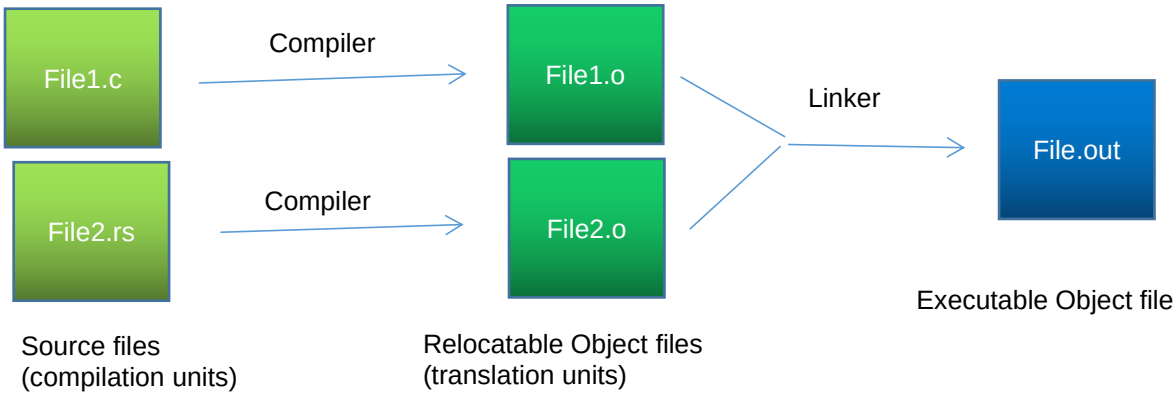
# Linker Project




Object file: File with binary content.
Relocatable: Independent of the final executable.

Linker = ( path-to-file1, path-to-file2, path-to-fileN, linker-script ) -> executable

Linker script: Configuration file with settings for linking process. Optional.

Linking process depends on:
- Input file types. This project supports Elf32 and Elf64 only. All input files must be of same type.
- Output file types. This project supports Elf32 and Elf64 only. Input and output file types must be same.
- Input file endianness. This project supports only LE.
- Input file target architecture. This project supports only Intel 80386 (also called i386, x86), and AMD x86-64. Linux 3+ can execute both architectures natively.

Note that input and output files are of the same type: Elf (executable and linkable format). Relocatable obj files can be distinguished from executables by examining the ELF Header of the file. Other possible ELF file types: Shared object files (.so), archive file (.a). These are not supported by this project. Relocatable object files generally have .o extension and executables have none. Extensions are part of the name and don't determine the file type.

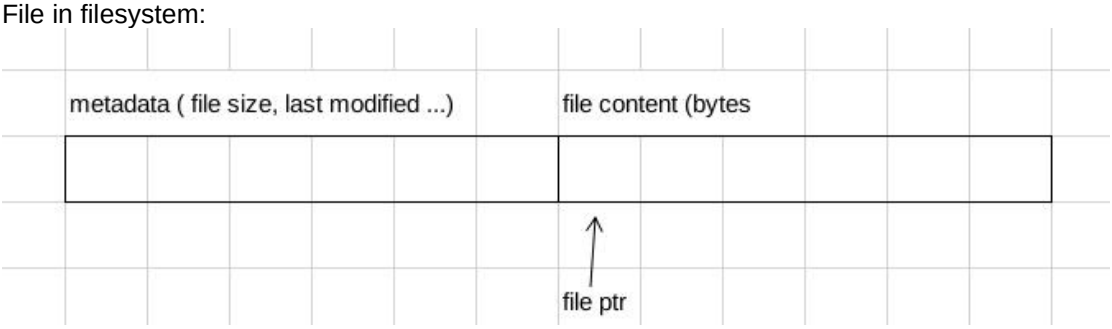
Linker has following cmd line signature

```
linker path-to-file1 path-to-file2 path-to-fileN
```

- If any of the files don’t exist, or cant be opened,
- If any of the files are not valid ELF files,
- If all files don’t have the same file format (Elf32 or Elf64),
- If any of the files is big endian,
- If all files don’t have the same target architecture (i386 or x86-64),
  Linker errors and exits

- If there is a file called `config` on pwd (current directory), assume it is the linker script, and parse it to get the settings. If it doesn’t exist, assume default settings.

File in filesystem:




Metadata format is dependent on OS and filesystem. It cannot be accessed by file i/o functions. It can be accessed by OS syscalls. Linker can ignore metadata and work on bytes only. End of file is not delimited by \0, so program needs to know file sizes ahead of time.

For simplicity, this project will not optimize from space by working on files one by one. Instead, it will load all input files to the memory entirely, then create objects which represent the data in files, and then close the files.

At the very beginning of ELF files is found the ELF header, which is always of fixed size. It has the following signature:

ELF32:

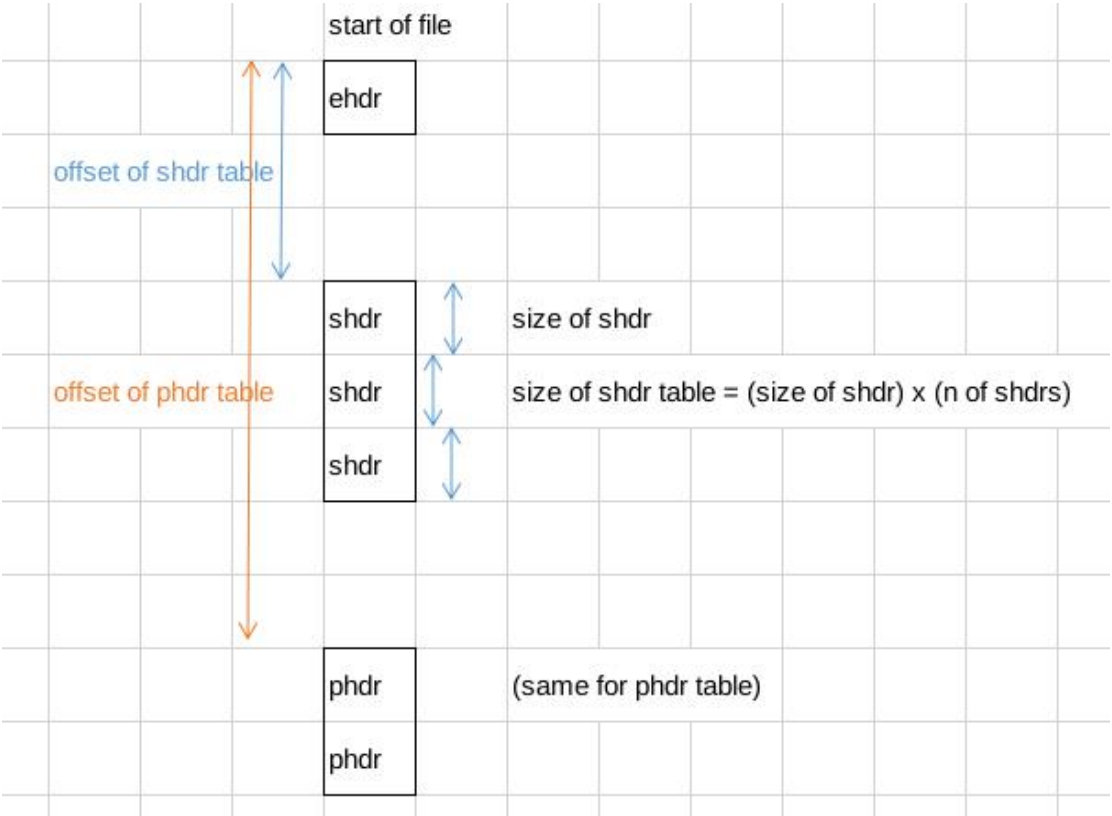
```
#define EI_NIDENT        16

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;
```

ELF64:

```
typedef struct
{
        unsigned char   e_ident[16];    /* ELF identification */
        Elf64_Half      e_type;         /* Object file type */
        Elf64_Half      e_machine;      /* Machine type */
        Elf64_Word      e_version;      /* Object file version */
        Elf64_Addr      e_entry;        /* Entry point address */
        Elf64_Off       e_phoff;        /* Program header offset */
        Elf64_Off       e_shoff;        /* Section header offset */
        Elf64_Word      e_flags;        /* Processor-specific flags */
        Elf64_Half      e_ehsize;       /* ELF header size */
        Elf64_Half      e_phentsize;    /* Size of program header entry */
        Elf64_Half      e_phnum;        /* Number of program header entries */
        Elf64_Half      e_shentsize;    /* Size of section header entry */
        Elf64_Half      e_shnum;        /* Number of section header entries */
        Elf64_Half      e_shstrndx;     /* Section name string table index */
} Elf64_Ehdr;
```

Refer to Elf docs for data type definitions. e_ident always occupies first 16 bytes, independent of elf file type (elf32 or elf64). e_ident [ ei_class ] determines the file type. Rest of the object file's structure depends on exact file type. (Eg. Elf32 header is 52 bytes, Elf64 header is 64 bytes).

Everything in relocatable files is expressed in terms of offsets, not absolute addresses. Eg, the offset of shdr table (section header table) is measured from start of file.



EHDR: Elf header (file header)
SHDR: Section header
PHDR: Program header
SHDR TBL: Section header table
PHDR TBL: Program header table

-Every ELF file: EHDR (elf header, or file header) is mandatory.
-Relocatable obj files: SHDR TBL is mandatory.
PHDR TBL is optional.
-Executable obj files: PHDR TBL is mandatory.
SHDR TBL is optional.

Note that there might be space between EHDR and other HDR tables, they don't have to come one after the other. Offsets are determined by compilers.

SHDR and PHDR hold offsets (again from start of file) to sections and segments, respectively. Note: section != segment.

First header (index 0) in SHDR and PHDR are called NULL headers. They hold all zeros, and are unused.

**Figure 1-8. Section Header**

```
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;
```

All sections have names, which are not unique. The names themselves are not on the header. Only an offset of the string is stored. (starting from shstrtab). String is delimited by \0.

Type of the section is important:
- PROGBITS type means section holds the program. `.text, .data` will have this section type.

- NOBITS type means section should hold space after loadtime. `.bss` will have this type.

- SYMTAB type means section holds symbols. More on this later

- STRTAB means section holds strings.

- REL, RELA types mean section holds relocations. More on this later

- NULL type means section should be ignored.

- For this project, all other section types (DYNSYM, HASH ... ) will be ignored.

All PROGBITS (text, data) and NOBITS (bss) will have three flags: write, allocate, execute. Write means content of section is writable, it must not be put to read-only memory by the loader. Allocate means space will be allocated in memory by the loader. Linker needs to think as if 0 bytes exist in the section. Execute means executable code. In some architectures executable sections must reside in special memory devices.

Section header [ address ] holds the final address, after linking process is done. Relocatable obj files will have this field 0. Executable files will hold address where loader needs to load the section.

Section header [ offset ] holds the offset of the section from start of file.

Section header [ size ] holds the size of the section in the file. If the section type is NOBITS, section will hold no space in the rel obj file, and the size will tell how many bytes to allocate after the loading phase.

Section header [ adr align ] holds alignment in bytes that section must have. Eg 4 byte alignment means address can be 4,8,12,...

Section header [ entity size ] only applies to SYMTAB, REL, RELA and other unused sections that are tables. For example for SYMTAB, this number holds size of each entry of the table (a symbol ). For REL, RELA it holds size of a relocation struct.

Section header [ link ] and [ info ] is only relevant for SYMTAB, REL and RELA. To be explained later...

Linker’s first job is to group sections from all translation units together and form segments. For this project, make sure that in every translation unit, all section names are unique. Also make sure that if there are sections with same name in different translation units, sections have same flags (write, alloc, exec).

In relocatable obj files, position is specified by offsets from some points (start of file). The reason compiler doesn't assign final addresses itself is to make library linking possible. Linker is given the final job of changing the offsets to addresses.

There are two kind of addresses: virtual address ( aka VMA), and physical address (aka LMA). For Linux and other UNIX System V compatible systems, virtual address needs to be specified and physical address can be ignored by the linker. It will be filled later by the OS. Linker can assign virtual addresses from 0 to MAX_ADDRESSABLE_WORD, which is 2^32 for i386 and 2^64 for x86-64.

From application writers' point of view, whole address space of 0 - 2^64 can be used by the application. Virtual addresses are then mapped to physical addresses by OS with the help of MMU. This makes it impossible to corrupt other programs' memory contents.

For cases where linker script is not specified, default settings will be used to assign virtual addresses. The procedure is:
- For all sections that need to be allocated ( ie. have alloc flag set):
- Chain sections with same name together. Since their flags must be same, output section will have same flags. When combining sections, make sure that section addresses are aligned to 0x1000.
- Group sections with same flags together. Groups of sections with same flags are called segments.
- Starting from address 0, put all sections one after the other.

In embedded applications, it is regular to not rely on linker's procedure of address assignment. Sometimes it is desirable to make sure that a section has a desired address. For these cases linker script is used.

All linkers have their own linker script format. For this project, linker script is a text file of following format. It only configures entry address, and section addresses.

```
Entry
Output_section_name virtual_address
Output_section_name virtual_address
....
```

where Entry can either be a name or a number. If a number, entry address is a fixed value, and if a name, a symbol (label in asm) will specify the entry point. Make sure that sections don't intersect. Also make sure that specified virtual addresses are all divisible by 0x1000.

In case where linker script is provied, procedure for section address assignment is as follows:
- For all sections that need to be allocated ( ie. have alloc flag set):
- Chain all sections with same name. Make sure they have same flags.
- First go through script and place sections that are there in specified addresses.
- For all remaining sections, group them according to flags.
- Starting from address 0, try to place a group. If it is intersecting, try a smaller one. If none fit, go to the next empty space, and try again. All groups should be aligned to 0x1000.

After placing sections in appropriate places, create an output_shdr_table and output_phdr_table. Only PHDR is required by OS, but Elf analyzer tools like `readelf` rely on SHDR, so for this project we will also generate an output SHDR.

**Figure 2-1. Program Header**

```
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;
```

First PHDR will be filled with zeros, and given type NULL. All the other PHDRs will specify a segment each, and will have type LOAD. Segments are contigious groupings of sections. Not all ssections need to be in a segment. Only sections which are allocated need to have segments.

Phdr [ offset ] specifies the offset of the segment in the executable file, from start of file.

Phdr [ vaddr ] specifies the address where to load to. This should already be determined at the section placement process.

Phdr [ paddr ] will be same as [ vaddr ]. Later it will be modified by OS.

Phdr [ filesz ] specifies the segment length in bytes in the file. Remember that bss section occupies no space in file, but does so in memory.

Phdr [ memsz ] is almost same as filesz, except that it also counts bss occupied space.

Phdr [ flags ] is the segment flags: exec, write, read. All segments will be readable, and other two flags will correspond with the section group’s write, exec flags.

Phdr [ align ] is the alignment. Set to 0x1000.

After having set all virtual addresses to sections and segments, next step is to form `.symtab, .strtab` and `.shstrtab` sections in the executable.

`.strtab` and `.shstrtab` are sections which hold strings, one after the other. Do not confuse these strings with strings that are written by programmers, and that are kept in `.data` section. `.strtab` holds strings, which are names of labels in asm. `shstrtab` holds strings, which are names of the sections (such as text, data, bss, symtab ... ). There can be only one `strtab` and `shstrtab` in every translation unit.

These names of labels are kept for debugging purposes, as well as for external references.

A symbol is a name (label) and a place where it points to. Names are kept in strtab, and places where they point to are specified by offsets or virtual addresses, depending on elf file type. In a relocatable obj file, positions are offsets, measured from start of section. Thus they must also know which section it is. All sections have an index in their file. Symbols keep these indices.

**Figure 1-15. Symbol Table Entry**

```
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;
```

Sym [ name ] is offset from strtab, and until \0 delimiter.

Sym [ value ] is an offset from a start of section, if it is a relocatable obj file, or a virtual address if it is an executable.

Sym [ size ] and Sym [ other ] are ignored and kept at 0 in this project.

Sym [ shndx ] specifies which section it is. It can also have special values like 0 (SHN_UNDEF), SHN_COMMON, SHN_ABS.

Sym [ info ] is a bitmap holding symbol type, and binding. Type should not be changed, and is not important to linker. Binding is one of local, weak and global.

When linker combines all symtabs into one big symtab for the output executable file, it checks their binding: there can't be two global symbols with same name.

After combining all symtabs into one, linker must change all offsets to virtual addresses in symbols' [ value ] fields. This should be possible because all sections are already assigned virtual addresses.

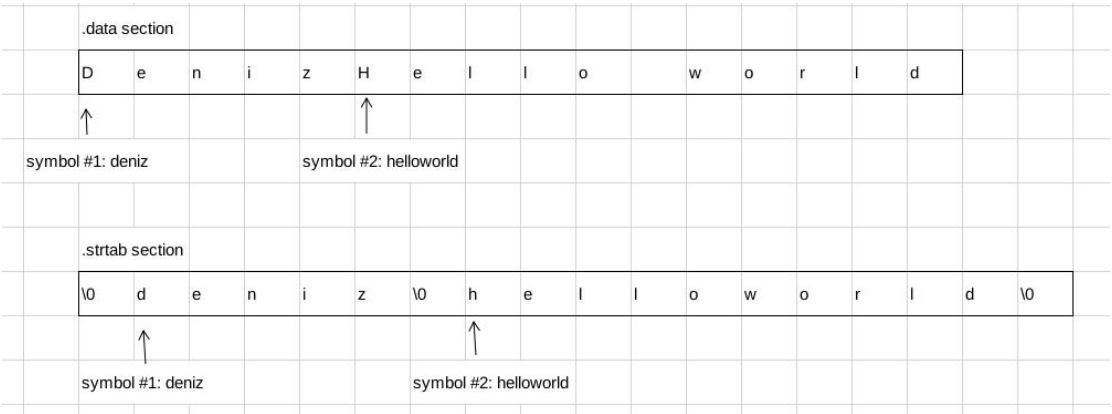
Lastly, symtab's section header's [ sh_link ] and [ sh_info ] fields need to be filled.
Shdr [ sh_link ] = index of associated strtab section in shdr table.
Shdr [ sh_info ] = index of last local symbol + 1

There can be multiple REL sections in every translation unit (or RELA in x86-64). They hold a list of relocations. A relocation is simply a copy paste operation. For example "copy 4 bytes from A to B". Typically A is a symbol: a symbol always points to an area in code somewhere. B is an offset and a starting point for relocatable obj files, and a virtual address for shared libraries. Since we don't deal with shared libraries in this project, B is always an offset from a section start.

For example, let's analyze the following hello world asm code

```
1
2   section .data
3
4   deniz: db "Deniz"
5   helloworld: db "Hello world", 10
6
7
8   section .text
9   mov eax, 4
10  mov ebx, 1
11  mov ecx, helloworld
12  mov edx, 12
13  int 80h
```

Data and strtab sections before linking



Text section before linking:

```
; asm code                  ; machine code
mov eax, 4          ->      b8 04 00 00 00
mov ebx, 1          ->      bb 01 00 00 00
mov ecx, helloworld ->      b9 ?? ?? ?? ??
mov edx, 12         ->      ba 0c 00 00 00
int 80h             ->      cd 80
```

Notice how compiler leaves ?? where there needs to be an address. Compiler fills ??s with 0s. Compiler adds a relocation entry:

`Relocation from where the helloworld symbol is (data section offset 5) to text section offset 11.`

After linking, say data section ends up at virtual address 1000, then ?? ?? ?? ?? will be replaced by 1005 (in little endian order).

Two important fields in REL section header are rel_shdr [ sh_link ] and rel_shdr [ sh_info ]. Just like with symtabs, these two fields hold special information for linking:

Rel_shdr [ sh_link ] : index of associated symtab in section header table
Rel_shdr [ sh_info ] : index of section in section header table where to apply (copy) the values. That is, the target section.

Now that we know which symtab we are working with, and which target section to apply relocations to, we need two things:
- Index of symbol in symtab,
- Offset from section start to the target bytes

**Figure 1-19. Relocation Entries**

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

typedef struct {
        Elf32_Addr      r_offset;
        Elf32_Word      r_info;
        Elf32_Sword     r_addend;
} Elf32_Rela;
```

R_offset of a relocation gives the offset from section start.

R_info is a bitmat holding r_sym and r_type. R_sym is the index of the symbol in the symbol table, as needed. R_type is the “relocation type”.

Relocation types depend on target arch (i386 or x86-64).

**Figure 1-3. Relocation Types**

| Name | Value | Field | Calculation |
|---|---|---|---|
| R_386_NONE | 0 | none | none |
| R_386_32 | 1 | word32 | S+A |
| R_386_PC32 | 2 | word32 | S+A-P |

A — *This means the addend used to compute the value of the relocatable field.*

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.*

*A relocation entry's* `r_offset` *value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values. The Intel architecture uses only* `Elf32_Rel` *relocation entries, the field to be relocated holds the addend. In all cases, the addend and the computed result use the same byte order.*

Relocation types specifies what value to put inside the target 4 or 8 bytes.

If the symbol of relocation is undefined (ie extern symbol is used), linker should scan all symbols and find a global, or a weak symbol with the same name. If it cannot find one, then it is a linking error.

When relocations are done, REL and RELA sections can be disarded, and all others used to construct the final executable. It should have file header, SHDR Table, PHDR Table, sections, segments, symtab, strtab, shstrtab.

Every executable ELF file must have eh [ entry ] specified for OS to start executing. The value should be selected procedurally:
- If linker script’s first line is a number, then it is the entry
- If linker script’s first line is a string, then find the symbol and use its virtual address
- If linker script’s first line is of form “string number” then no entry is specified. Also, if no script is provided, that also means no entry is specified. Default entry point is selected as following:
- If there is a global _start symbol, select it
- Else, start from first byte of the first segment with flag exec=true.

To assemble samples, use NASM assembler. Command to assemble but not to link:

```
nasm -f elf32 ./main.s -o main32.o
```

Here, change `elf32` with `elf64` where needed. `Main.s` is the source asm file.

To link using ld:

```
ld -m elf_i386 ./main32.o -o main32 -e baslangic
```

Change `elf_i386` with `elf_x86_64.` -e flag specifies which symbol to use as entry point.

To analyze elf:

```
readelf -a elf_file
```