ELF Tutorial

From OSDev Wiki

This tutorial describes the steps to loading ELF files targeting the i386 (32-bit architecture, little-endian byte order). All code in the tutorial is in the form of C compatible C++ and strives to teach by example, by using simplified (and sometimes naive), neat, and functional snippets of code. It may later be expanded to cover other types of ELF files, or formats targeting other architectures or machine types.

Difficulty level

Medium

Executable Formats

Microsoft

16 bit:
COM
MZ
NE
32/64 bit:
PE
Mixed (16/32 bit):
LE

***nix**

A.out
ELF

Contents

Macro Definitions used in Examples

The example code provided in this tutorial relies on a couple of preprocessor definitions, which can be found below. In addition, the code relies on a printf()

function for printing debug information to the screen, malloc() for memory allocations, memset() for zeroing memory, and the stdint.h header for fixed width integer types.

```
# ifndef __cplusplus
#       include <stdbool.h>
#       define CLINK
#       define CEXTERN  extern
# else
#       define CLINK    extern "C"
#       define CEXTERN  extern "C"
# endif /* C++ */

# define WRITE(PRE, STR, ...)   \
        printf("[%s] %s() : ", (PRE), (__FUNCTION__));  \
        printf(STR, __VA_ARGS__);

# define DEBUG(STR, ...)        \
        WRITE("INFO", STR, __VA_ARGS__);

# define WARN(STR, ...)         \
        WRITE("WARN", STR, __VA_ARGS__);

# define ERROR(STR, ...)        \
        WRITE("ERR!", STR, __VA_ARGS__);
```

Some additional macros will be defined along with their relevant examples or sections.

ELF Data Types

```
# include <stdint.h>

typedef uint16_t Elf32_Half;    // Unsigned half int
typedef uint32_t Elf32_Off;     // Unsigned offset
typedef uint32_t Elf32_Addr;    // Unsigned address
typedef uint32_t Elf32_Word;    // Unsigned int
typedef int32_t  Elf32_Sword;   // Signed int
```

The ELF file format is made to function on a number of different architectures, many of which support different data widths. For support across multiple machine types, the ELF format provides a set of guidelines for fixed width types that make

up the layout of the section and data represented within object files. You may choose to name your types differently or use types defined in stdint.h directly, but they should conform to those shown above.

The ELF Header

The ELF file format has only one header with fixed placement: the ELF header, present at the beginning of every file. The format itself is extremely flexible as the positioning, size, and purpose of every header (save the ELF header) is described by another header in the file.

```
# define ELF_NIDENT     16

typedef struct {
        uint8_t         e_ident[ELF_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;
```

The ELF header is the first header in an ELF file and it provides important information about the file (such as the machine type, architecture and byte order, etc.) as well as a means of identifying and checking whether the file is valid. The ELF header also provides information about other sections in the file, since the can be appear in any order, vary in size, or may be absent from the file altogether. Universal to all ELF files are the first 4 bytes (the magic number) which are used identify the file. When working with the file through the **Elf32_Ehdr** type defined above, these 4 bytes are accessible from indexes 0 - 3 of the field **e_ident**.

```
enum Elf_Ident {
        EI_MAG0         = 0, // 0x7F
        EI_MAG1         = 1, // 'E'
        EI_MAG2         = 2, // 'L'
```

```
        EI_MAG3         = 3, // 'F'
        EI_CLASS        = 4, // Architecture (32/64)
        EI_DATA         = 5, // Byte Order
        EI_VERSION      = 6, // ELF Version
        EI_OSABI        = 7, // OS Specific
        EI_ABIVERSION   = 8, // OS Specific
        EI_PAD          = 9  // Padding
};

# define ELFMAG0        0x7F // e_ident[EI_MAG0]
# define ELFMAG1        'E'  // e_ident[EI_MAG1]
# define ELFMAG2        'L'  // e_ident[EI_MAG2]
# define ELFMAG3        'F'  // e_ident[EI_MAG3]

# define ELFDATA2LSB    (1)  // Little Endian
# define ELFCLASS32     (1)  // 32-bit Architecture
```

The first field in the header consists of 16 bytes, many of which provide important information about the ELF file such as the intended architecture, byte order, and ABI information. Since this tutorial focuses on implementing a x86 compatible loader, only relevant value definitions have been included.

```
enum Elf_Type {
        ET_NONE         = 0, // Unkown Type
        ET_REL          = 1, // Relocatable File
        ET_EXEC         = 2  // Executable File
};

# define EM_386         (3)  // x86 Machine Type
# define EV_CURRENT     (1)  // ELF Current Version
```

The file header also provides information about the machine type and file type. Once again, only the relevant definitions have been included above.

Checking the ELF Header

Before an ELF file can be loaded, linked, relocated or otherwise processed, it's important to ensure that the machine trying to perform the aforementioned is able to do. This entails that the file is a valid ELF file targeting the local machine's architecture, byte order and CPU type, and that any operating system spefic semantics are satisfied.

```
bool elf_check_file(Elf32_Ehdr *hdr) {
```

```
        if(!hdr) return false;
        if(hdr->e_ident[EI_MAG0] != ELFMAG0) {
                ERROR("ELF Header EI_MAG0 incorrect.\n");
                return false;
        }
        if(hdr->e_ident[EI_MAG1] != ELFMAG1) {
                ERROR("ELF Header EI_MAG1 incorrect.\n");
                return false;
        }
        if(hdr->e_ident[EI_MAG2] != ELFMAG2) {
                ERROR("ELF Header EI_MAG2 incorrect.\n");
                return false;
        }
        if(hdr->e_ident[EI_MAG3] != ELFMAG3) {
                ERROR("ELF Header EI_MAG3 incorrect.\n");
                return false;
        }
        return true;
}
```

Assuming that an ELF file has already been loaded into memory (either by the bootloader or otherwise), the first step to loading an ELF file is checking the ELF header for the magic number that should be present at the begining of the file. A minimal implementation of this could simply treat the image of file in memory as a string and do a comparision against a predefined string. In the example above, the comparision is done byte by byte through the ELF header type, and provides detailed feedback when the method encounters an error.

```
bool elf_check_supported(Elf32_Ehdr *hdr) {
        if(!elf_check_file(hdr)) {
                ERROR("Invalid ELF File.\n");
                return false;
        }
        if(hdr->e_ident[EI_CLASS] != ELFCLASS32) {
                ERROR("Unsupported ELF File Class.\n");
                return false;
        }
        if(hdr->e_ident[EI_DATA] != ELFDATA2LSB) {
                ERROR("Unsupported ELF File byte order.\n");
                return false;
        }
        if(hdr->e_machine != EM_386) {
                ERROR("Unsupported ELF File target.\n");
                return false;
```

```
        }
        if(hdr->e_ident[EI_VERSION] != EV_CURRENT) {
                ERROR("Unsupported ELF File version.\n");
                return false;
        }
        if(hdr->e_type != ET_REL && hdr->e_type != ET_EXEC) {
                ERROR("Unsupported ELF File type.\n");
                return false;
        }
        return true;
}
```

The next step to loading an ELF object is to check that the file in question is intended to run on the machine that has loaded it. Again, the ELF header provides the necessary information about the file's indended target. The code above assumes that you have implemented a function called **elf_check_file**() (or used the one provided above), and that the local machine is i386, little-endian and 32-bit. It also only allows for executable and relocatable files to be loaded, although this can be changed as necessary.

Loading the ELF File

```
static inline void *elf_load_rel(Elf32_Ehdr *hdr) {
        int result;
        result = elf_load_stage1(hdr);
        if(result == ELF_RELOC_ERR) {
                ERROR("Unable to load ELF file.\n");
                return NULL;
        }
        result = elf_load_stage2(hdr);
        if(result == ELF_RELOC_ERR) {
                ERROR("Unable to load ELF file.\n");
                return NULL;
        }
        // TODO : Parse the program header (if present)
        return (void *)hdr->e_entry;
}

void *elf_load_file(void *file) {
        Elf32_Ehdr *hdr = (Elf32_Ehdr *)file;
        if(!elf_check_supported(hdr)) {
                ERROR("ELF File cannot be loaded.\n");
                return;
        }
```

```
        switch(hdr->e_type) {
                case ET_EXEC:
                        // TODO : Implement
                        return NULL;
                case ET_REL:
                        return elf_load_rel(hdr);
        }
        return NULL;
}
```

The ELF Section Header

The ELF format defines a lot of different types of section and their relevant headers, not all of which are present in every file, and there's no guarantee on which order they are appear in. Thus, in order to parse and process these sections the format also defines section headers, which contains information such as section names, sizes, locations and other relevant information. The list of all the section headers in an ELF image is refered to as the section header table.

```
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;
```

The section header table contains a number of important fields, some of which have different meanings for different sections. Another point of interest is that the **sh_name** field does not point directly to a string, instead it gives the offset of a string in the section name string table (the index of the table itself is defined in the ELF header by the field **e_shstrndx**). Each header also defines the position of the actual section in the file image in the field **sh_offset**, as an offset from the begining of the file.

```
# define SHN_UNDEF      (0x00) // Undefined/Not present
```

```
enum ShT_Types {
        SHT_NULL        = 0,   // Null section
        SHT_PROGBITS    = 1,   // Program information
        SHT_SYMTAB      = 2,   // Symbol table
        SHT_STRTAB      = 3,   // String table
        SHT_RELA        = 4,   // Relocation (w/ addend)
        SHT_NOBITS      = 8,   // Not present in file
        SHT_REL         = 9,   // Relocation (no addend)
};

enum ShT_Attributes {
        SHF_WRITE       = 0x01, // Writable section
        SHF_ALLOC       = 0x02  // Exists in memory
};
```

Above are a number of constants that are relevant to the tutorial (a good deal more exist). The enumeration **ShT_Types** defines a number of different types of sections, which correspond to values stored in the field **sh_type** in the section header. Similarly, **ShT_Attributes** corresponds to the field **sh_flags**, but are bit flags rather than stand-alone values.

Accessing Section Headers

Getting access to the section header itself isn't very difficult: It's position in the file image is defined by **e_shoff** in the ELF header and the number of section headers is in turn defined by **e_shnum**. Notably, the first entry in the section header is a NULL entry; that is to say, fields in the header are 0. The section headers are continuous, so given a pointer to the first entry, subsequent entries can accessed with simple pointer arithmatic or array operations.

```
static inline Elf32_Shdr *elf_sheader(Elf32_Ehdr *hdr) {
        return (Elf32_Shdr *)((int)hdr + hdr->e_shoff);
}

static inline Elf32_Shdr *elf_section(Elf32_Ehdr *hdr, int idx) {
        return &elf_sheader(hdr)[idx];
}
```

The two methods above provide convinient access to section headers on a by-index basis using the principals noted above, and they will be used fruquently in the example code that follows.

Section Names

One notable procedure is accessing section names (since, as mentioned before, they header only provides an offset into the section name string table), which is also fairly simple. The whole operation can be broken down into a simple series of steps:

1. Get the section header index for the string table from the ELF header (stored in **e_shstrndx**). Make sure to check the index against **SHN_UNDEF**, as the table may not be present.
2. Access the section header at the given index and find the table offset (stored in **sh_offset**).
3. Calculate the position of the string table in memory using the offset.
4. Create a pointer to the name's offset into the string table.

An example of the process is shown in the two convinience methods below.

```
static inline char *elf_str_table(Elf32_Ehdr *hdr) {
        if(hdr->e_shstrndx == SHN_UNDEF) return NULL;
        return (char *)hdr + elf_section(hdr, hdr->e_shstrndx)->sh_
}

static inline char *elf_lookup_string(Elf32_Ehdr *hdr, int offset)
        char *strtab = elf_str_table(hdr);
        if(strtab == NULL) return NULL;
        return strtab + offset;
}
```

Note that before you attempt to access the name of a section, you should first check that the section has a name (The offset given by sh_name is not equal to **SHN_UNDEF**).

ELF Sections

ELF object files can have a very large number of sections, however, it is important to note that only some sections need to be processed during program loading, and not all of them may exist within the object file itself (ie. the BSS). This segment will describe a number of sections that should be processed during program loading (given they are present).

The Symbol Table

The symbol table is a section (or a number of sections) that exist within the ELF file and define the location, type, visibility and other traits of various symbols declared in the original source, created during compilation or linking, or

otherwise present in the file. Since an ELF object can have multiple symbol tables, it is necessary to either iterate over the file's section headers, or to follow a reference from another section in order to access one.

```
typedef struct {
        Elf32_Word              st_name;
        Elf32_Addr              st_value;
        Elf32_Word              st_size;
        uint8_t                 st_info;
        uint8_t                 st_other;
        Elf32_Half              st_shndx;
} Elf32_Sym;
```

Each symbol table entry contains a number of notable bits of information such as the symbol name (**st_name**, may be **STN_UNDEF**), the symbol's value (**st_value**, may be absolute or relative address of value), and the field **st_info** which conatins both the symbol type and binding. As an aside, the first entry in each symbol table is a NULL entry, so all of it's fields are 0.

```
# define ELF32_ST_BIND(INFO)    ((INFO) >> 4)
# define ELF32_ST_TYPE(INFO)    ((INFO) & 0x0F)

enum StT_Bindings {
        STB_LOCAL               = 0, // Local scope
        STB_GLOBAL              = 1, // Global scope
        STB_WEAK                = 2  // Weak, (ie. __attribute__((v
};

enum StT_Types {
        STT_NOTYPE              = 0, // No type
        STT_OBJECT              = 1, // Variables, arrays, etc.
        STT_FUNC                = 2  // Methods or functions
};
```

As mentioned above, **st_info** contains both the symbol type and biding, so the 2 macros above provide access to the individual values. The enumeration **StT_Types** provides a number of possible symbol types, and **StB_Bindings** provides possible symbol bindings.

Accessing the Value of a Symbol

Some operation such as linking and relocation require the value of a symbol (or

rather, the address thereof). Although the symbol table entries do define a field **st_value**, it may only contain a relative address. Below is an example of how to compute the absolute address of the value of the symbol. The code has been broken up into multple smaller section so that it is easier to understand.

```
static int elf_get_symval(Elf32_Ehdr *hdr, int table, uint idx) {
        if(table == SHN_UNDEF || idx == SHN_UNDEF) return 0;
        Elf32_Shdr *symtab = elf_section(hdr, table);

        if(idx >= symtab->sh_size) {
                ERROR("Symbol Index out of Range (%d:%u).\n", table
                return ELF_RELOC_ERR;
        }

        int symaddr = (int)hdr + symtab->sh_offset;
        Elf32_Sym *symbol = &((Elf32_Sym *)symaddr)[idx];
```

The above performs a check against both the symbol table index and the symbol index; if either is undefined, 0 is returned. Otherwise the section header entry for the symbol table at the given index is accessed. It then checks that the symbol table index is not outside the bounds of the symbol table. If the check fails an error message is displayed and an error code is returned, otherwise the symbol table entry at the given index is is retreived.

```
        if(symbol->st_shndx == SHN_UNDEF) {
                // External symbol, lookup value
                Elf32_Shdr *strtab = elf_section(hdr, symtab->sh_li
                const char *name = (const char *)hdr + strtab->sh_c

                extern void *elf_lookup_symbol(const char *name);
                void *target = elf_lookup_symbol(name);

                if(target == NULL) {
                        // Extern symbol not found
                        if(ELF32_ST_BIND(symbol->st_info) & STB_WEA
                                // Weak symbol initialized as 0
                                return 0;
                        } else {
                                ERROR("Undefined External Symbol :
                                return ELF_RELOC_ERR;
                        }
                } else {
                        return (int)target;
```

```
                }
```

If the section to which the symbol is relative (given by **st_shndx**) is equal to **SHN_UNDEF**, the symbol is external and must be linked to its definition. The string table is retreived for the current symbol table (the string table for a given symbol table is available in the table's section header in **sh_link**), and the symbol's name is found in the string table. Next the function **elf_lookup_symbol**() is used to find a symbol defintion by name (this function is not provided, a minimal implementation always return NULL). If the symbol definition is found, it is returned. If the symbol has the **STB_WEAK** flag (is a weak symbol) 0 is returned, otherwise an error message is displayed and an error code returned.

```
        } else if(symbol->st_shndx == SHN_ABS) {
                // Absolute symbol
                return symbol->st_value;
        } else {
                // Internally defined symbol
                Elf32_Shdr *target = elf_section(hdr, symbol->st_sh
                return (int)hdr + symbol->st_value + target->sh_off
        }
}
```

If the the value of **sh_ndx** is equal to **SHN_ABS**, the symbol's value is absolute and is reurned immidiately. If **sh_ndx** doesn't contain a special value, that means the symbol is defined in the local ELF object. Since the value given by **sh_value** is relative to a section defined **sh_ndx**, the relevant section header entry is accessed, and the symbol's address is computed by adding the address of the file in memory to the symbol's value with its section offset.

The String Table

The string table conceptually is quite simple: it's just a number of consecutive zero-terminated strings. String literals used in the program are stored in one of the tables. There are a number of different string tables that may be present in an ELF object such as .strtab (the default string table), .shstrtab (the section string table) and .dynstr (string table for dynamic linking). Anytime the loading process needs access to a string, it uses an offset into one of the string tables. The offset may point to the beginning of a zero-terminated string or somewhere in the middle or even to the zero terminator itself, depending on usage and scenario. The size of the string table itself is specified by **sh_size** in the corresponding section header entry.

The simplest program loader may copy all string tables into memory, but a more complete solution would omit any that are not necessary during runtime such, notably those not flagged with **SHF_ALLOC** in their respective section header (such as .shstrtab, since section names aren't used in program runtime).

The BSS and SHT_NOBITS

The BSS (the section named ".bss") is in the simplest way of describing it: A block of memory which has been zeroed. The BSS is the area in memory where variables with global lifetime that haven't been initialized (or have been initialized to 0 or NULL) are stored. The section header for the BSS defines its **sh_type** as **SHT_NOBITS**, which means that it isn't present in the file image, and must be allocated during runtime. A simple and naive way of allocating a BSS is to malloc some memory and zero it out with a memset. Failing to zero the BSS can cause unexpected behavious from any loaded programs. Another thing to note is that the BSS should be allocated before performing any operation that relies on relative addressing (such as relocation), as failing to do so can cause code to reference garbage memory or fault.

While the BSS is one specific example, any section that is of type **SHT_NOBITS** and has the attribute **SHF_ALLOC** should be allocated early on during program loading. Since this tutorial is intended to be general and unspecific, the example below will follow the trend and use the simplest example for allocating sections.

```
static int elf_load_stage1(Elf32_Ehdr *hdr) {
        Elf32_Shdr *shdr = elf_sheader(hdr);

        unsigned int i;
        // Iterate over section headers
        for(i = 0; i < hdr->e_shnum; i++) {
                Elf32_Shdr *section = &shdr[i];

                // If the section isn't present in the file
                if(section->sh_type == SHT_NOBITS) {
                        // Skip if it the section is empty
                        if(!section->sh_size) continue;
                        // If the section should appear in memory
                        if(section->sh_flags & SHF_ALLOC) {
                                // Allocate and zero some memory
                                void *mem = malloc(section->sh_size
                                memset(mem, 0, section->sh_size);

                                // Assign the memory offset to the
                                section->sh_offset = (int)mem - (in
                                DEBUG("Allocated memory for a secti
```

```
                        }
                }
        }
        return 0;
}
```

The example above allocates as much memory as necessary for the section, described by the **sh_size** field of the section's header. Although the function in the example only seeks out sections that needs to be allocated, it can be modified to perform other operation that should be performed early on into the loading process.

Relocation Sections

Relocatable ELF files have many uses in kernel programming, especially as modules and drivers that can be loaded at startup, and are especially useful because they are position independant, thus can easily be placed after the kernel or starting at some convinient address, and don't require their own address space to function. The process of relocation itself is conceptually simple, but may get more difficult with the introduction of complex relocation types.

Relocation starts with a table of relocation entries, which can be located using the relevant section header. There are actually two different kinds of relocation structures; one with an explicit added (section type **SHT_RELA**), one without (section type **SHT_REL**). Relocation entires in the table are continuous and the number of entries in a given table can be found by dividing the size of the table (given by **sh_size** in the section header) by the size of each entry (given by **sh_entsize**). Each relocation table is specific to a single section, so a single file may have multiple relocation tables (but all entries within a given table will be the same relocation structure type).

```
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;
```

The above are the defintions for the different structure types for relocations. Of

note if the value stored in **r_info**, as the upper byte designates the entry in the symbol table to which the relocation applies, whereas the lower byte stores the type of relocation that should be applied. Note that an ELF file may have multiple symbol tables, thus the index of the section header table that refers to the symbol table to which these relocation apply can be found in the **sh_link** field on this relocation table's section header. The value in **r_offset** gives the relative position of the symbol that is being relocated, within its section.

```
# define ELF32_R_SYM(INFO)      ((INFO) >> 8)
# define ELF32_R_TYPE(INFO)     ((uint8_t)(INFO))

enum RtT_Types {
        R_386_NONE              = 0, // No relocation
        R_386_32                = 1, // Symbol + Offset
        R_386_PC32              = 2  // Symbol + Offset - Section O
};
```

As previously mentioned, the **r_info** field in **Elf32_Rel**(**a**) refers to 2 seperate values, thus the set of macro functions above can be used to attain the individual values; **ELF32_R_SYM**() provides access to the symbol index and **ELF32_R_TYPE**() provides access to the relocation type. The enumeration **RtT_Types** defines the relocation typs this tutorial will encompass.

Relocation Example

Loading a relocatable ELF file entails processing all relocation entries present in the file (Remember to alloc allocate all **SHT_NOBITS** sections first!). This process starts with finding all the relocation tables in the file, which is done in the example code below.

```
# define ELF_RELOC_ERR -1

static int elf_load_stage2(Elf32_Ehdr *hdr) {
        Elf32_Shdr *shdr = elf_sheader(hdr);

        unsigned int i, idx;
        // Iterate over section headers
        for(i = 0; i < hdr->e_shnum; i++) {
                Elf32_Shdr *section = &shdr[i];

                // If this is a relocation section
                if(section->sh_type == SHT_REL) {
                        // Process each entry in the table
```

```
                        for(idx = 0; idx < section->sh_size / secti
                                Elf32_Rel *reltab = &((Elf32_Rel *)
                                int result = elf_do_reloc(hdr, relt
                                // On error, display a message and
                                if(result == ELF_RELOC_ERR) {
                                        ERROR("Failed to relocate s
                                        return ELF_RELOC_ERR;
                                }
                        }
                }
        }
        return 0;
}
```

Note that the code above only processes **Elf32_Rel** entries, but it can be modified to process entries with explicit addends as well. The code also relies on a function called **elf_do_reloc** which will be shown in the next example. This example function stops, displays an error message, and returns an error code if it's unable to process a relocation.

Relocating a Symbol

As the following function is fairly complex, it's been broken up into smaller managable chumks and explained in detail. Note that the code shown below assumes that the file being relocated is a relocatable ELF file (ELF executables and shared objects may also contain relocation entries, but are processed somewhat differently). Also note that **sh_info** for section headers of type **SHT_REL** and **SHT_RELA** stores the section header to which the relocation applies.

```
# define DO_386_32(S, A)        ((S) + (A))
# define DO_386_PC32(S, A, P)   ((S) + (A) - (P))

static int elf_do_reloc(Elf32_Ehdr *hdr, Elf32_Rel *rel, Elf32_Shdr
        Elf32_Shdr *target = elf_section(hdr, reltab->sh_info);

        int addr = (int)hdr + target->sh_offset;
        int *ref = (int *)(addr + rel->r_offset);
```

The above code defines the macro functions that are used to complete relocation calculations. It also retreives the section header for the section wherein the symbol exists and computes a reference to the symbol. The variable **addr** denotes the start of the symbol's section, and **ref** is created by adding the offset to the

symbol from the relocation entry.

```
        // Symbol value
        int symval = 0;
        if(ELF32_R_SYM(rel->r_info) != SHN_UNDEF) {
                symval = elf_get_symval(hdr, reltab->sh_link, ELF32
                if(symval == ELF_RELOC_ERR) return ELF_RELOC_ERR;
        }
```

Next the value of the symbol being relocated is accessed. If the symbol table index stored in **r_info** is undefined, then the value defaults to 0. The code also references a function called **elf_get_symval**(), which was implemented previously. If the value returned by the function is equal to **ELF_RELOC_ERR**, relocation is stoped and said error code is returned.

```
        // Relocate based on type
        switch(ELF32_R_TYPE(rel->r_info)) {
                case R_386_NONE:
                        // No relocation
                        break;
                case R_386_32:
                        // Symbol + Offset
                        *ref = DO_386_32(symval, *ref);
                        break;
                case R_386_PC32:
                        // Symbol + Offset - Section Offset
                        *ref = DO_386_PC32(symval, *ref, (int)ref);
                        break;
                default:
                        // Relocation type not supported, display e
                        ERROR("Unsupported Relocation Type (%d).\n"
                        return ELF_RELOC_ERR;
        }
        return symval;
}
```

Finally, this segment of code details the actual relocation process, performing the necessary calculating the relocated symbol and returning it's value on success. If the relocation type is unsupported an error message is displayed, relocation is stoped and the function returns an error code. Assuming no errors have occured, relocation is now complete.

The ELF Program Header

The program header is a structure that defines information about how the ELF program behaves once it's been loaded, as well as runtime linking information. ELF program headers (much like section headers) are all grouped together to make up the program header table.

```
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;
```

The program header table contains a continuous set of program headers (thus they can be accessed as if they were an array). The table itself can be accessed using the **e_phoff** field defined in the ELF header, assuming that it is present. The header itself defines a number of useful fields like **p_type** which distinguishes between headers, **p_offset** which stores the offset to the segment the header refers to, and **p_vaddr** which defines the address at which position-dependent code should exist.

TODO : Expand and Detail.

See Also

Articles

- ELF
- Modular Kernel
- System V ABI

External Links

- ELF Format Specifications (http://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-46512.html#scrolltoc) Detailed and up-to-date ELF information (including SPARC in depth) by Oracle.
- System V ABI (http://www.sco.com/developers/gabi/latest/contents.html)

about ELF
- LSB specifications (http://www.linuxfoundation.org/en/Specifications)
  See (generic or platform-specific) 'Core' specifications for additional ELF information.

Retrieved from "http://wiki.osdev.org/index.php?title=ELF_Tutorial&oldid=17007"
Category: Level 2 Tutorials
