САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

По домашней работе № 4

Система набора команд RISC-V

Выполнил: Зайнидинов Мирзофирдавс Шавкатович

Студент группы M313D

Санкт-Петербург

2020

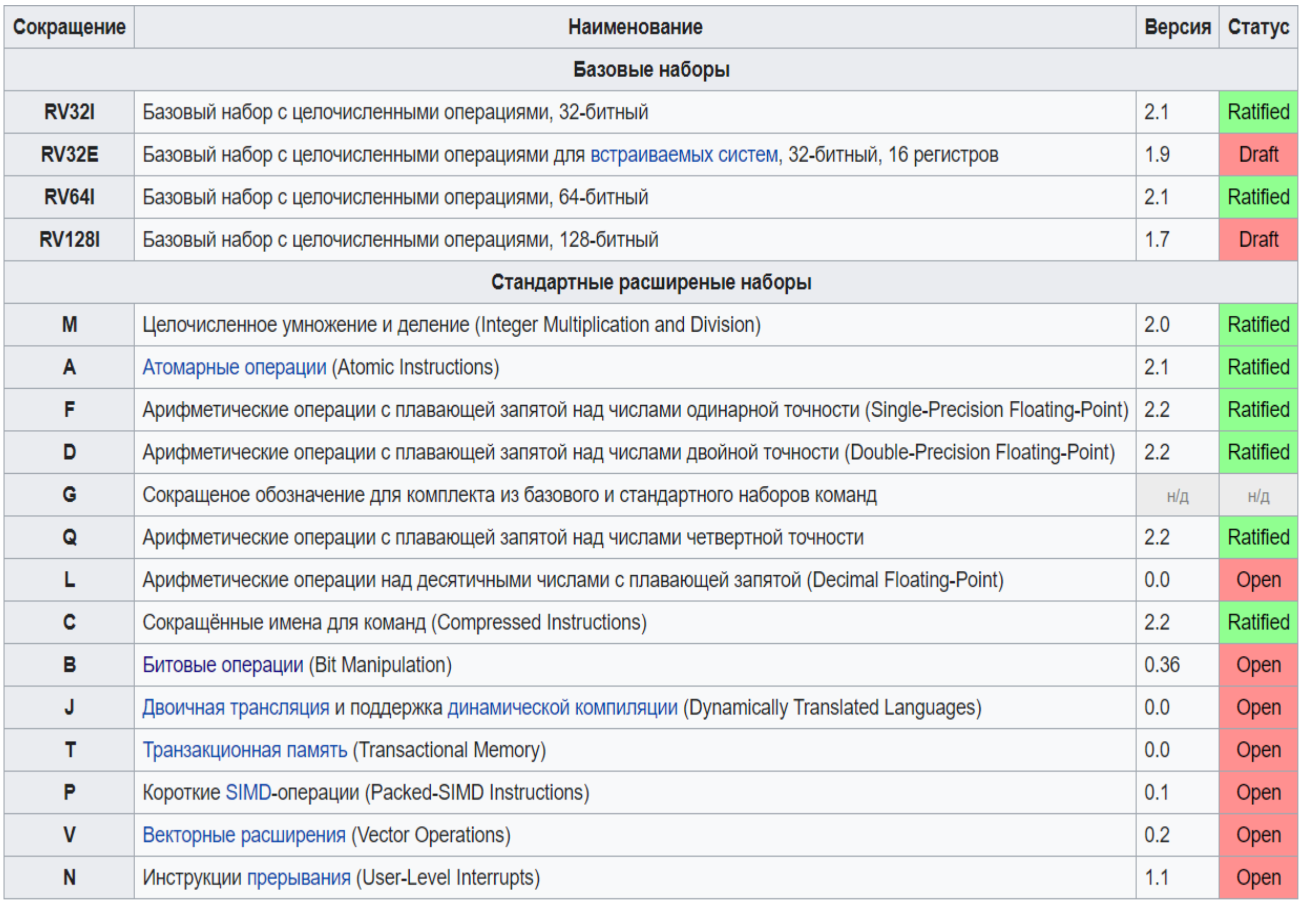
Цель работы: <знакомство со системой набора команд RISC-V>

Инструментарий и требования к работе: <Java>

Теоретическая часть

Знакомство со системой набора команд RISC-V

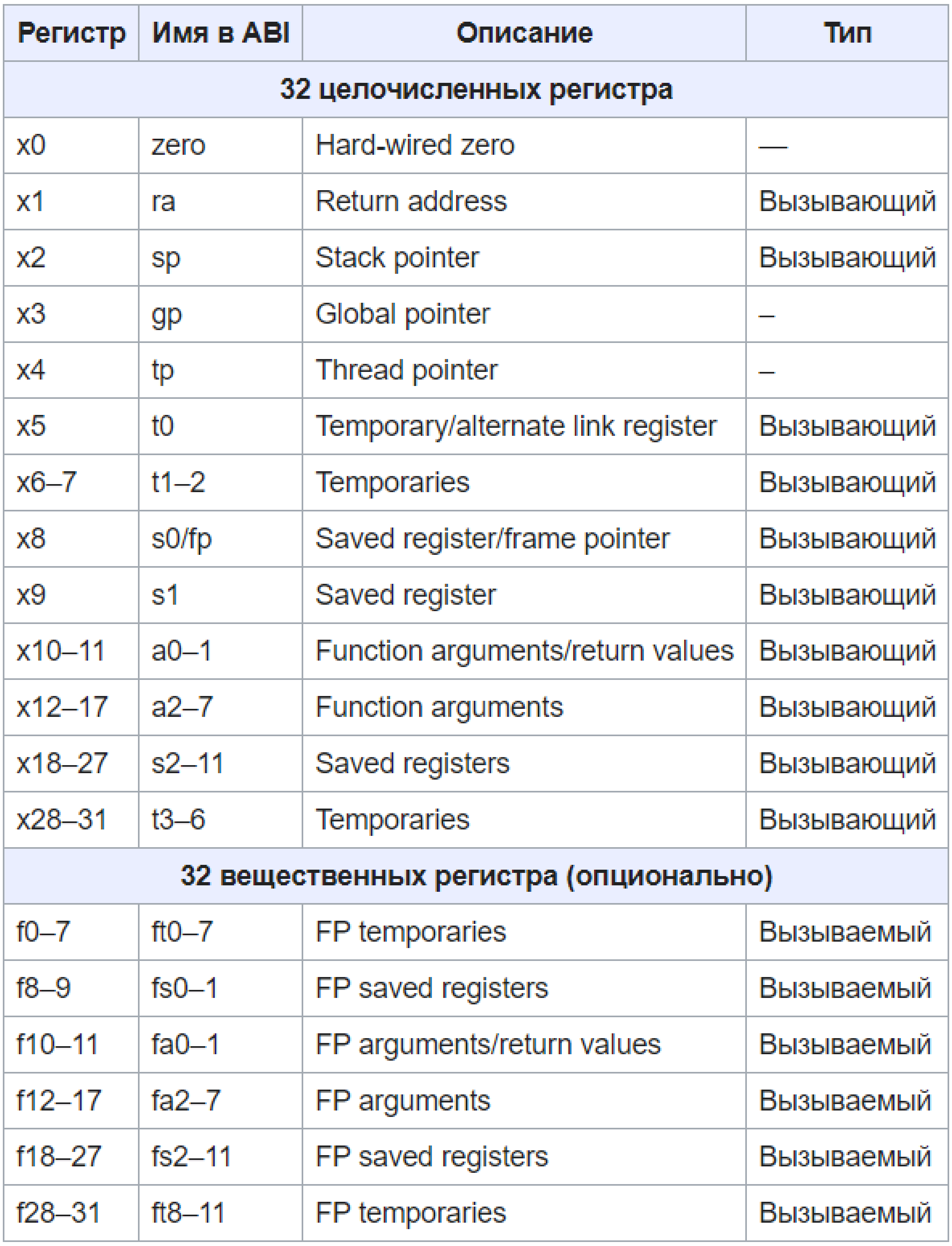
RISC-V – это открытая и свободная система команд (ISA) и процессорная архитектура на основе RISC для микропроцессоров и микроконтроллеров, которая была представлена в 2010 году исследователями калифорнийского университета в Беркли. Так как оно основано на базе RISC, нужно сначала узнать, что такое RISC. RISC – это компьютер с набором простых, коротких команд, за счет которых уменьшается время выполнения, чтобы декодирование было более простым. RISC-V определяет около 50 стандартных команд. В архитектуре RISC-V имеется обязательное для реализации небольшое количество команд и несколько стандартных расширений. В этот обязательный набор входят команды условной и безусловной передачи управления и ветвления, минимальный набор арифметической и битовых операций на регистрах, операции с памятью и еще несколько служебных команд. Базовое подмножество команд используют следующий набор регистров: x0 – специальный нулевой регистр, 31 целочисленный регистр для общего назначения, регистр счетчика, а также множество CSR (Control and Status Registers). Команды в RISC-V предусмотрены для реализации архитектур с 32 битными, 64 битными т 128 битными регистрами общего назначения. Разрядность регистровых операций зависит и соответствует от размера регистров, одни и те же значения в регистрах могут трактоваться целыми числами со знаком и без знака. Операции нигде не сохраняют биты переноса, так же не генерируют некоторые исключения, например деления на ноль. Все это должно быть осуществлено программно, а не аппаратно. Размер операнда может отличатся от размера регистра только в случаи операции с памятью. RISC-V использует только little – endian модель. Little-endian – это порядок записывания в памяти компьютера байтов, младшего к старшему или же справа – налево (Например число 123 было бы записана в число 321 при таком порядке). Один и тот – же код на RISC-V может запускаться на различных архитектурах RISC-V, поскольку кодировка базового набора не зависит от разрядности архитектур. Спецификацией RISC-V предусмотрено несколько областей в пространстве кодировок команд для пользовательских архитектуры которые поддерживаются на уровне ассемблера. Список набора команд RISC-V (смотрите на рисунок-таблицу №1).



Сокращение	Наименование	Версия	Статус
	Базовые наборы		
RV32I	Базовый набор с целочисленными операциями, 32-битный	2.1	Ratified
RV32E	Базовый набор с целочисленными операциями для встраиваемых систем, 32-битный, 16 регистров	1.9	Draft
RV64I	Базовый набор с целочисленными операциями, 64-битный	2.1	Ratified
RV128I	Базовый набор с целочисленными операциями, 128-битный	1.7	Draft
	Стандартные расширеные наборы		
M	Целочисленное умножение и деление (Integer Multiplication and Division)	2.0	Ratified
A	Атомарные операции (Atomic Instructions)	2.1	Ratified
F	Арифметические операции с плавающей запятой над числами одинарной точности (Single-Precision Floating-Point)	2.2	Ratified
D	Арифметические операции с плавающей запятой над числами двойной точности (Double-Precision Floating-Point)	2.2	Ratified
G	Сокращеное обозначение для комплекта из базового и стандартного наборов команд	н/д	н/д
Q	Арифметические операции с плавающей запятой над числами четвертной точности	2.2	Ratified
L	Арифметические операции над десятичными числами с плавающей запятой (Decimal Floating-Point)	0.0	Open
C	Сокращённые имена для команд (Compressed Instructions)	2.2	Ratified
B	Битовые операции (Bit Manipulation)	0.36	Open
J	Двоичная трансляция и поддержка динамической компиляции (Dynamically Translated Languages)	0.0	Open
T	Транзакционная память (Transactional Memory)	0.0	Open
P	Короткие SIMD-операции (Packed-SIMD Instructions)	0.1	Open
V	Векторные расширения (Vector Operations)	0.2	Open
N	Инструкции прерывания (User-Level Interrupts)	1.1	Open

Рисунок-таблица №1

Говоря о регистрах, RISC-V имеет 32 целочисленных регистра и еще 32 вещественных регистра при реализации вещественных видов команд. Для операций над числами в бинарных форматах с плавающей точкой используются дополнительный набор из 32 FPU (Floating Point Unit), которые совместно используются расширениями базового набора для трех вариантов точности: одинарный – 32 бита, двойной – 64 бита и четверной – 128 бит. Так же рассматривается вариант о добавлении в RISC-V набора из 32 векторных регистров с вариативной длиной обрабатываемых значений (Смотрите на рисунок-таблицу №2).



Регистр	Имя в ABI	Описание	Тип
32 целочисленных регистра			
x0	zero	Hard-wired zero	—
x1	ra	Return address	Вызывающий
x2	sp	Stack pointer	Вызывающий
x3	gp	Global pointer	–
x4	tp	Thread pointer	–
x5	t0	Temporary/alternate link register	Вызывающий
x6–7	t1–2	Temporaries	Вызывающий
x8	s0/fp	Saved register/frame pointer	Вызывающий
x9	s1	Saved register	Вызывающий
x10–11	a0–1	Function arguments/return values	Вызывающий
x12–17	a2–7	Function arguments	Вызывающий
x18–27	s2–11	Saved registers	Вызывающий
x28–31	t3–6	Temporaries	Вызывающий
32 вещественных регистра (опционально)			
f0–7	ft0–7	FP temporaries	Вызываемый
f8–9	fs0–1	FP saved registers	Вызываемый
f10–11	fa0–1	FP arguments/return values	Вызываемый
f12–17	fa2–7	FP arguments	Вызываемый
f18–27	fs2–11	FP saved registers	Вызываемый
f28–31	ft8–11	FP temporaries	Вызываемый

Рисунок-таблица №2

Структуры Elf-файла

ELF-файл (Executable and Linkable Format) – это формат двоичных файлов, которые используются во многих операционных системах, также этот формат используются во многих других системах. Этот формат был представлен Лабораторией UNIX, как часть двоичного интерфейса приложения ABI операционной системы UNIX-V. Потом он был выбран в качестве основного файлового формата в 32 – битном архитектуре Intel x86. Призван упростить жизнь программиста и представлять четкую и понятную структуру файла. Этот формат имеет несколько типов:

1. Перемещаемый файл, хранящий команды и данные, которые могут быт связаны с другими объектными файлами.
2. Разделяемый объектный файл. Он также содержит инструкции и данные, но может быть использован двумя способами. Первый способ, он может быть связан с перемещаемыми файлами и разделяемыми объектами и в итоге будет создан новый объектный файл. Второй способ, при запуске программы на выполнения операционная система может динамически связать его с исполняемым файлом программы, в итоге будет создан образ программы.
3. Исполняемый файл хранит полное описание, позволяющее системе создать образ процессора.

Структуру файла можно рассматривать с двух сторон:

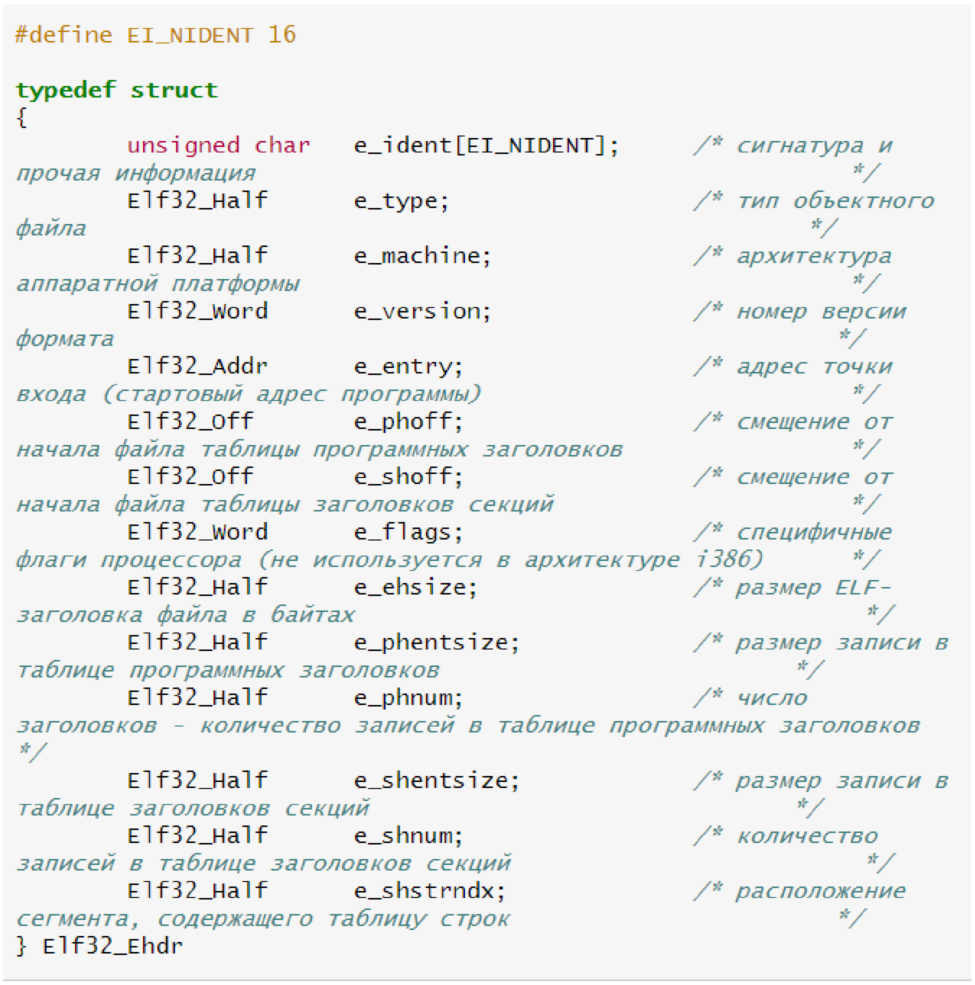
1. Со стороны компоновщика (L)
2. Со стороны загрузчика (E)

Любой ELF-файл состоит из:

1. ELF заголовка, в котором указаны все общие параметры (тип, архитектура процессора, виртуальный адрес и. т. д.).
2. Таблицы программных заголовок, которая служит для описания сегментов ELF-файла.

3. Таблицы заголовков секций, которая характеризует секции файла.

Сегмент – это непрерывная область адресного пространства со своими атрибутами доступа. Сегмент кода имеет атрибут исполнения, а сегмент данных имеет атрибуты чтения и записи. В самом ELF-файле сегменты не выравниваются и хранятся плотно прижатыми друг к другу. Сегмент ELF-файла можно разбит на несколько частей, эти части называются секциями. Сейчас ELF-файлы используются и на 32 – битных, и на 64 – битных системах и для машин с порядком Little – endian (справа налево), и для машин с порядком Big – endian (слева направо). Заголовок ELF-файла имеет фиксированное расположение в начале файла и содержит общее описание структуры файла и его основные характеристики, такие как: тип, версия формата, архитектура процессора, виртуальный адрес точки входа, размеры т смещения остальных частей ELF-файла (посмотрите на рисунок №1).



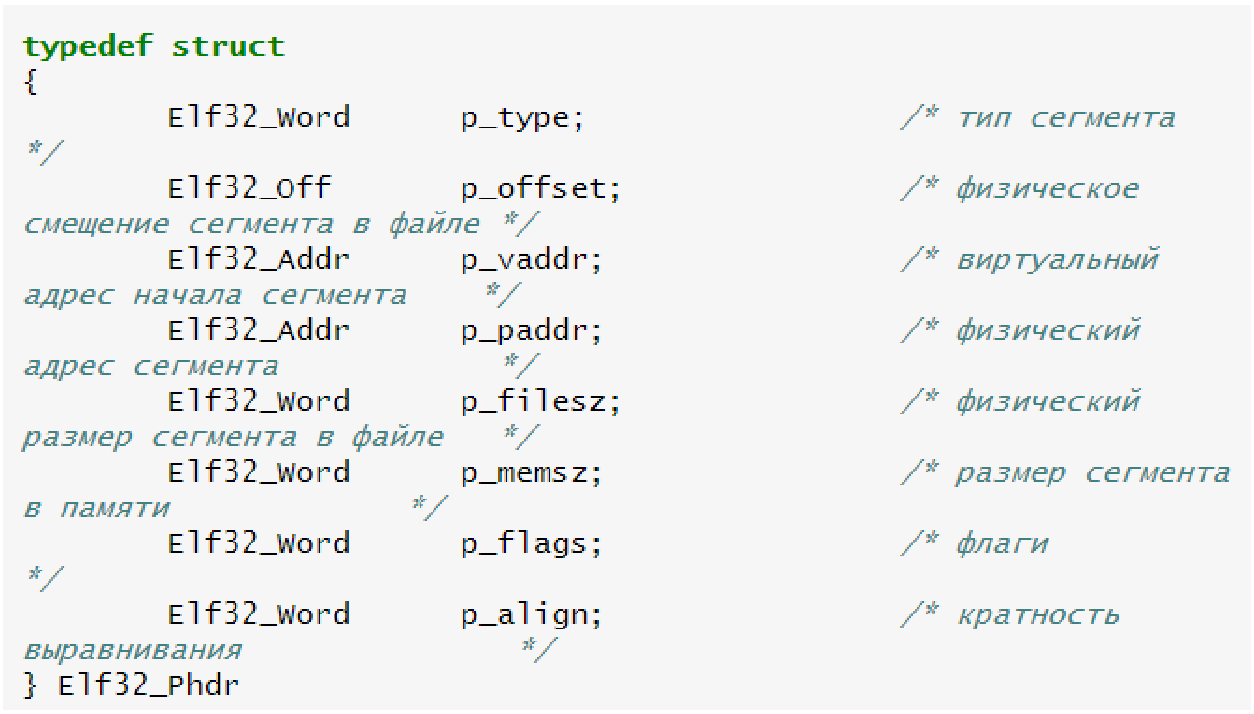
```
#define EI_NIDENT 16

typedef struct
{
        unsigned char   e_ident[EI_NIDENT];     /* сигнатура и
прочая информация                                          */
        Elf32_Half      e_type;                 /* тип объектного
файла                                                   */
        Elf32_Half      e_machine;              /* архитектура
аппаратной платформы                                       */
        Elf32_Word      e_version;              /* номер версии
формата                                                  */
        Elf32_Addr      e_entry;                /* адрес точки
входа (стартовый адрес программы)                          */
        Elf32_Off       e_phoff;                /* смещение от
начала файла таблицы программных заголовков                */
        Elf32_Off       e_shoff;                /* смещение от
начала файла таблицы заголовков секций                     */
        Elf32_Word      e_flags;                /* специфичные
флаги процессора (не используется в архитектуре i386)      */
        Elf32_Half      e_ehsize;               /* размер ELF-
заголовка файла в байтах                                    */
        Elf32_Half      e_phentsize;            /* размер записи в
таблице программных заголовков                          */
        Elf32_Half      e_phnum;                /* число
заголовков - количество записей в таблице программных заголовков
*/
        Elf32_Half      e_shentsize;            /* размер записи в
таблице заголовков секций                               */
        Elf32_Half      e_shnum;                /* количество
записей в таблице заголовков секций                         */
        Elf32_Half      e_shstrndx;             /* расположение
сегмента, содержащего таблицу строк                        */
} Elf32_Ehdr
```

Рисунок №1

Таблица заголовков программы расположена сразу после заголовка файла и содержит заголовки сегментов, каждый из которых описывает сегмент программы такие как:

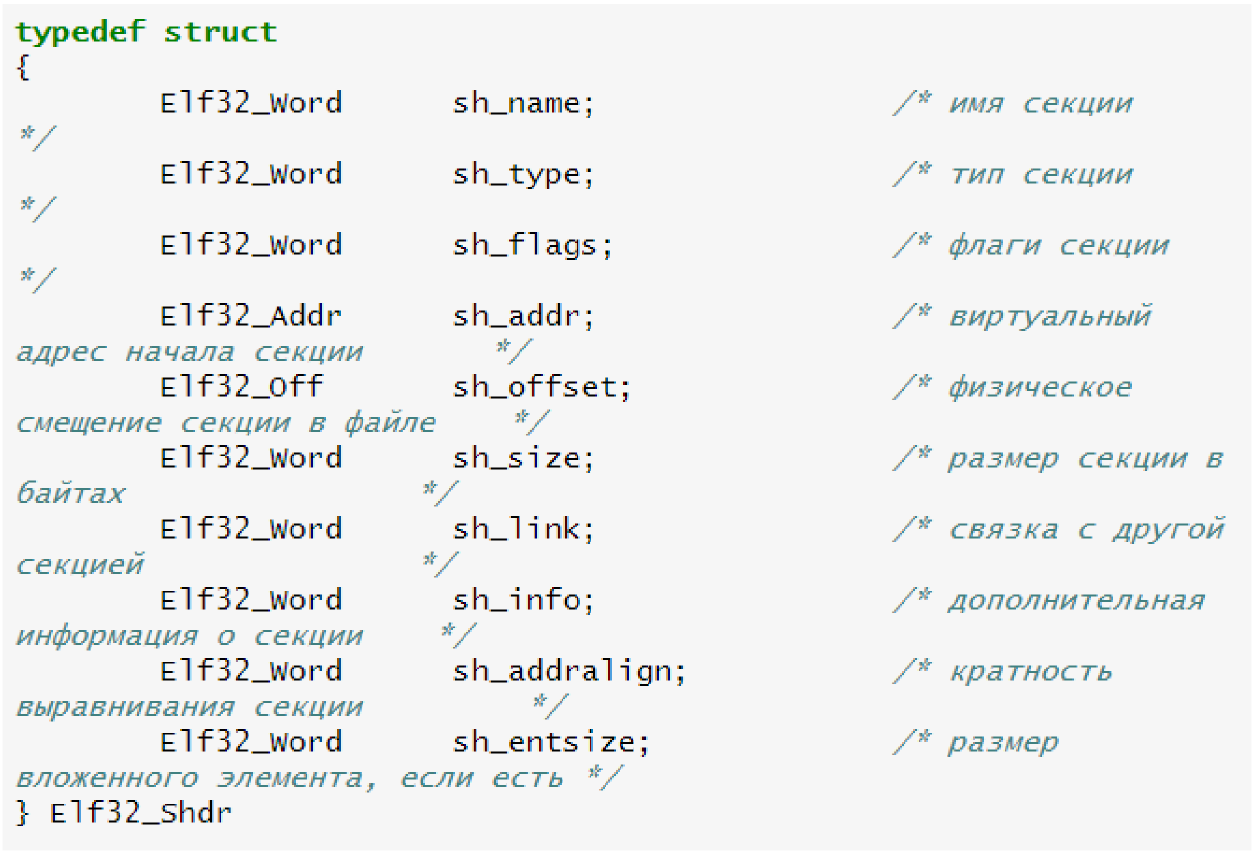
1. Тип сегмента и действия операционной системы с данным сегментом.
2. Расположение сегмента.
3. Точка входа сегмента.
4. Размер сегмента.
5. Флаги доступа к сегменту (запись, чтение, выполнение).

Смотрите на рисунок №2



```
typedef struct
{
        Elf32_Word      p_type;                     /* тип сегмента
*/
        Elf32_Off       p_offset;                   /* физическое
смещение сегмента в файле */
        Elf32_Addr      p_vaddr;                    /* виртуальный
адрес начала сегмента    */
        Elf32_Addr      p_paddr;                    /* физический
адрес сегмента           */
        Elf32_Word      p_filesz;                   /* физический
размер сегмента в файле    */
        Elf32_Word      p_memsz;                    /* размер сегмента
в памяти                */
        Elf32_Word      p_flags;                    /* флаги
*/
        Elf32_Word      p_align;                    /* кратность
выравнивания                */
} Elf32_Phdr
```

Таблица заголовков секций характеризует секции файла. Таблица секция является обязательной для компоновщика и необязательной для системного загрузчика. Компоновщик комбинирует секции с похожими атрибутами и оптимальным образом размещает их по сегментам при сборке файла (смотрите на рисунок № 3).



```
typedef struct
{
        Elf32_Word      sh_name;                /* имя секции
*/
        Elf32_Word      sh_type;                /* тип секции
*/
        Elf32_Word      sh_flags;               /* флаги секции
*/
        Elf32_Addr      sh_addr;                /* виртуальный
адрес начала секции       */
        Elf32_Off       sh_offset;              /* физическое
смещение секции в файле    */
        Elf32_Word      sh_size;                /* размер секции в
байтах                */
        Elf32_Word      sh_link;                /* связка с другой
секцией               */
        Elf32_Word      sh_info;                /* дополнительная
информация о секции    */
        Elf32_Word      sh_addralign;           /* кратность
выравнивания секции         */
        Elf32_Word      sh_entsize;             /* размер
вложенного элемента, если есть */
} Elf32_Shdr
```

Рисунок №3

Практическая часть

Решение этой задачи можно разделить на 3 части:

1. Основной код
2. Исполняемый код (Main)
3. Вспомогательная библиотека

Рассмотрим работу исполняемого кода.

По условию задачи нам даются 32-битные ELF-файлы, так что для начало надо проверить является она ELF-файлом RISC-V и является ли его формат 32 битным. В противном случаи программа бросает исключение. Потом находим регистры, точнее какие регистры были использованы. В случаи нахождение регистра, которого нету в RISC-V так – же бросается исключение. Далее задается формат строки, простой метод, который корректирует вывод. Далее ищем .text. После этого уже декодируем команды, точнее какие команды закодированы в этом ELF – файле и какие регистры были использованы и что на них записано. Все это мы делаем при помощи вспомогательной библиотеки, для парсинга ELF - файла. После компилируем тесты в классе Main.

Вывод программы

```
00000000:< main> addi sp, sp, 4064

00000004: sw ra, 28(sp)

00000008: sw s0, 24(sp)

0000000C: addi s0, sp, 32

00000010: addi a0, zero, 0

00000014: sw a0, 4084(s0)

00000018: addi a1, zero, 64

0000001C: sw a1, 4080(s0)

00000020: sw a0, 4076(s0)

00000024: addi a0, zero, 1

00000028: sw a0, 4072(s0)

0000002C: jal zero, 0 #0x0000002C

00000030: lw a0, s0, 4072

00000034: lw a1, s0, 4080

00000038: bge a0, a1, 0 #0x00000038

0000003C: jal zero, 0 #0x0000003C

00000040: lw a0, s0, 4072

00000044: mul a0, a0, a0

00000048: lw a1, s0, 4076

0000004C: add a0, a0, a1

00000050: sw a0, 4076(s0)

00000054: jal zero, 0 #0x00000054

00000058: lw a0, s0, 4072
```

```
0000005C: addi a0, a0, 1

00000060: sw a0, 4072(s0)

00000064: jal zero, 0 #0x00000064

00000068: lw a0, s0, 4076

0000006C: lw s0, sp, 24

00000070: lw ra, sp, 28

00000074: addi sp, sp, 32

00000078: jalr zero, ra, 0
```

RISCVDisassembler.java

```java
package file.EL;

import net.fornwall.jelf.*;

import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.InputMismatchException;

public class RISCVDisassembler {
    final ElfFile file;

    public RISCVDisassembler(ElfFile file) {
        if (file.objectSize != ElfFile.CLASS_32) {
            throw new InputMismatchException("This ELF-file is not 32 bit.");
        }
        if (file.arch != 0xF3) {
            throw new InputMismatchException("This ELF-file is not for RISC-V.");
        }
        this.file = file;
    }


    public void Launch(OutputStreamWriter output) {
        PrintWriter writer = new PrintWriter(output);
        doDisassemble(writer);
        writer.flush();
    }

    String getRegisterString(int register) {
        if (register == 0) {
            return "zero";
        } else if (register == 1) {
            return "ra";
        } else if (register == 2) {
            return "sp";
        } else if (register == 3) {
            return "gp";
        } else if (register == 4) {
            return "tp";
        } else if (5 <= register && register <= 7) {
            return "t" + (register - 5);
        } else if (register == 8) {
            return "s0";
        } else if (register == 9) {
            return "s1";
        } else if (10 <= register && register <= 17) {
            return "a" + (register - 10);
        } else if (18 <= register && register <= 27) {
            return "s" + (register - 18 + 2);
        } else if (28 <= register && register <= 31) {
            return "t" + (register - 28 + 3);
        } else {
            throw new AssertionError("RISC-V doesn't have this register");
        }
    }

    private String getSymbolForAddr(long cur) {
        ElfSymbol symb = file.getELFSymbol(cur);
        String locS = String.format("0x%08X", cur);
        if (symb != null && symb.st_value == cur && symb.section_type ==
ElfSymbol.STT_FUNC) {
            locS += " <" + symb.getName() + ">";
        }
        return locS;
    }

    private void doDisassemble(PrintWriter out) {
        ElfSection textSection = file.firstSectionByName(".text");
        if (textSection == null)
            throw new InputMismatchException("No .text found");
        file.getDynamicSymbolTableSection();
        file.getSymbolTableSection();
        int maxSymbolLen = 10;
        file.parser.seek(textSection.header.section_offset);
        for (int cur = 0; cur < textSection.header.size; cur += 4){
            long virtualAddress = cur + textSection.header.address;
            out.print(String.format("%08X:", virtualAddress));
            int instruction = file.parser.readInt();
            ElfSymbol symb = file.getELFSymbol(virtualAddress);
            if (symb != null && symb.st_value == virtualAddress && symb.section_type
== ElfSymbol.STT_FUNC) {
                out.printf("<%" + maxSymbolLen + "s> ", symb.getName());
            } else {
                out.print(" ".repeat(maxSymbolLen + 3));
            }
            int opcode = instruction & ((1 << 7) - 1);
            int rd = instruction >> 7 & ((1 << 5) - 1);
            int funct3 = instruction >> 12 & ((1 << 3) - 1);
            int rs1 = instruction >> 15 & ((1 << 5) - 1);
            int rs2 = instruction >> 20 & ((1 << 5) - 1);
            int imm110 = instruction >> 20 & ((1 << 12) - 1);
            int funct7 = instruction >> 25;
            if (instruction == 0b1110011) {
                out.printf("%6s%n", "ecall");
            } else if (opcode == 0b0110111) {
                out.printf("%6s %s, %s%n", "lui", getRegisterString(rd),
Integer.toUnsignedString((instruction >>> 12) << 12));
            } else if (opcode == 0b0010111) {
                out.printf("%6s %s, %s%n", "auipc", getRegisterString(rd),
Integer.toUnsignedString((instruction >>> 12) << 12));
            } else if (opcode == 0b1101111) {
                int imm = instruction >> 12;
                int offset = (((imm >>> 9) & ((1 << 10) - 1)) << 1) |
                        (((imm >>> 8) & 1) << 11) |
                        ((imm & ((1 << 8) - 1)) << 12) |
                        (((imm >>> 19) & 1) << 20);
                if ((offset & (1 << 20)) != 0) {
                    offset = -offset & ((1 << 20) - 1);
                }
                out.printf("%6s %s, %d #%s%n", "jal", getRegisterString(rd), offset,
getSymbolForAddr(virtualAddress + offset));
            } else if (opcode == 0b1100111 && funct3 == 0b000) {
                if ((imm110 & (1 << 11)) != 0) {
                    imm110 = -imm110 & ((1 << 11) - 1);
                }
                out.printf("%6s %s, %s, %d%n", "jalr", getRegisterString(rd),
getRegisterString(rs1), imm110);
            } else if (opcode == 0b1100011) {
                int offset = (((instruction >>> 8) & ((1 << 4) - 1)) << 1) |
                        (((instruction >>> 25) & ((1 << 6) - 1)) << 5) |
                        (((instruction >>> 7) & 1) << 11) |
                        (((instruction >>> 31) & 1) << 12);
                if ((offset & (1 << 12)) != 0) {
                    offset = -offset & ((1 << 12) - 1);
                }
                String instr = new String[]{"beq", "bne", "??", "??", "blt", "bge",
"bltu", "bgeu"}[funct3];
                out.printf("%6s %s, %s, %d #%s %n", instr, getRegisterString(rs1),
getRegisterString(rs2), offset, getSymbolForAddr(virtualAddress + offset));
            } else if (opcode == 0b0000011) {
                String instr = new String[]{"lb", "lh", "lw", "??", "lbu", "lhu",
"??", "??"}[funct3];
                out.printf("%6s %s, %s, %d%n", instr, getRegisterString(rd),
getRegisterString(rs1), imm110);
            } else if (opcode == 0b0100011) {
                String instr = new String[]{"sb", "sh", "sw", "??", "??", "??", "??",
"??"}[funct3];
                int imm = rd | ((imm110 >>> 5) << 5);
                out.printf("%6s %s, %d(%s)%n", instr, getRegisterString(rs2), imm,
getRegisterString(rs1));
            } else if (opcode == 0b0010011) {
                if (funct3 == 0b001) {
                    out.printf("%6s %s, %s, %d%n", "slli", getRegisterString(rd),
getRegisterString(rs1), imm110);
                } else if (funct3 == 0b101) {
                    if (funct7 == 0b0100000) {
                        out.printf("%6s %s, %s, %d%n", "srai", getRegisterString(rd),
getRegisterString(rs1), imm110 & ((1 << 5) - 1));
                    } else {
                        out.printf("%6s %s, %s, %d%n", "srli", getRegisterString(rd),
getRegisterString(rs1), imm110);
                    }
                } else {
                    String instr = new String[]{"addi", "??", "slti", "sltiu",
"xori", "??", "ori", "andi"}[funct3];
                    out.printf("%6s %s, %s, %d%n", instr, getRegisterString(rd),
getRegisterString(rs1), imm110);
                }
            } else if (opcode == 0b110011) {
                if (funct7 == 0b0100000) {
                    String instr = new String[]{"sub", "??", "??", "??", "??", "sra",
"??", "??"}[funct3];
                    out.printf("%6s %s, %s, %s%n", instr, getRegisterString(rd),
getRegisterString(rs2), getRegisterString(rs1));
                } else if (funct7 == 0) {
                    String instr = new String[]{"add", "sll", "slt", "sltu", "xor",
"srl", "or", "and"}[funct3];
                    out.printf("%6s %s, %s, %s%n", instr, getRegisterString(rd),
getRegisterString(rs2), getRegisterString(rs1));
                } else if (funct7 == 1) {
                    String instr = new String[]{"mul", "mulh", "mulhsu", "mulhu",
"div", "divu", "rem", "remu"}[funct3];
                    out.printf("%6s %s, %s, %s%n", instr, getRegisterString(rd),
getRegisterString(rs2), getRegisterString(rs1));
                }
            } else {
                out.printf("????%n");
            }
        }
    }
}
```

Main.java

```
import file.ELF.RISCVDisassembler;
import net.fornwall.jelf.ElfFile;

import java.io.*;

public class Main {
    public static void main(String[] args) {
        if (args.length < 1) {
            System.err.println("Usage: <input file> [<output file>]");
            return;
        }
        try {
            OutputStreamWriter output = null;
            try (BufferedInputStream stream = new BufferedInputStream(new
FileInputStream(args[0]))) {
                if (args.length > 1) {
                    output = new OutputStreamWriter(new FileOutputStream(args[1]));
                } else {
                    output = new OutputStreamWriter(System.out);
                }
                RISCVDisassembler disassembler = new
RISCVDisassembler(ElfFile.from(stream));
                disassembler.Launch(output);
            } finally {
                if (output != null) {
                    output.close();
                }
            }
        } catch (FileNotFoundException e) {
            System.err.println("File is not found.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Файлы готовой библиотеки для работы и Elf-файлом

BackingFile.java

```
package net.fornwall.jelf;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.nio.Buffer;
import java.nio.MappedByteBuffer;

class BackingFile {
    private final ByteArrayInputStream byteArray;
    private final MappedByteBuffer mappedByteBuffer;
    private final long mbbStartPosition;

    public BackingFile(ByteArrayInputStream byteArray) {
        this.byteArray = byteArray;
        this.mappedByteBuffer = null;
        this.mbbStartPosition = -1;
    }

    public BackingFile(MappedByteBuffer mappedByteBuffer) {
        this.byteArray = null;
        this.mappedByteBuffer = mappedByteBuffer;
        this.mbbStartPosition = 0;
        ((Buffer)mappedByteBuffer).position((int) mbbStartPosition);
    }

    public void seek(long offset) {
        if (byteArray != null) {
            byteArray.reset();
            if (byteArray.skip(offset) != offset) throw new ElfException("seeking 
outside file");
        } else if (mappedByteBuffer != null) {
            ((Buffer)mappedByteBuffer).position((int)(mbbStartPosition + offset)); // 
we may be limited to sub-4GB mapped filess
        }
    }

    public void skip(int bytesToSkip) {
        if (byteArray != null) {
            long skipped = byteArray.skip(bytesToSkip);
            if (skipped != bytesToSkip) {
                throw new IllegalArgumentException("Wanted to skip " + bytesToSkip + 
" bytes, but only able to skip " + skipped);
            }
        } else {
            ((Buffer)mappedByteBuffer).position(mappedByteBuffer.position() + 
bytesToSkip);
        }
    }

    short readUnsignedByte() {
        int val = -1;
        if (byteArray != null) {
            val = byteArray.read();
        } else if (mappedByteBuffer != null) {
            byte temp = mappedByteBuffer.get();
            val = temp & 0xFF; // bytes are signed in Java =_= so assigning them to a
longer type risks sign extension.
        }

        if (val < 0) throw new ElfException("Trying to read outside file");
        return (short) val;
    }

    public int read(byte[] data) {
        if (byteArray != null) {
            try {
                return byteArray.read(data);
            } catch (IOException e) {
                throw new RuntimeException("Error reading " + data.length + " bytes",
e);
            }
        } else if (mappedByteBuffer != null) {
            mappedByteBuffer.get(data);
            return data.length;
        }
        throw new RuntimeException("No way to read from file or buffer");
    }

}
```

ElfDynamicSection.java

```
package net.fornwall.jelf;

import java.util.ArrayList;
import java.util.List;

/**
 * An {@link ElfSection} with information necessary for dynamic linking.
 * <p>
 * Given an {@link ElfFile}, use {@link ElfFile#getDynamicSection()} to obtain the
dynamic section for it if one exists,
 * which it only does if the ELF file is an object file participating in dynamic
linking.
 * <p>
 * This dynamic linking section contains a list of {@link ElfDynamicStructure}:s.
 * <pre>
 * Name                        Value  d_un         Executable   Shared Object
 * ----------------------------------------------------------------------
 * DT_NULL                         0  ignored      mandatory    mandatory
 * DT_NEEDED                       1  d_val        optional     optional
 * DT_PLTRELSZ                     2  d_val        optional     optional
 * DT_PLTGOT                       3  d_ptr        optional     optional
 * DT_HASH                         4  d_ptr        mandatory    mandatory
 * DT_STRTAB                       5  d_ptr        mandatory    mandatory
 * DT_SYMTAB                       6  d_ptr        mandatory    mandatory
 * DT_RELA                         7  d_ptr        mandatory    optional
 * DT_RELASZ                       8  d_val        mandatory    optional
 * DT_RELAENT                      9  d_val        mandatory    optional
 * DT_STRSZ                       10  d_val        mandatory    mandatory
 * DT_SYMENT                      11  d_val        mandatory    mandatory
 * DT_INIT                        12  d_ptr        optional     optional
 * DT_FINI                        13  d_ptr        optional     optional
 * DT_SONAME                      14  d_val        ignored      optional
 * DT_RPATH*                      15  d_val        optional     ignored
 * DT_SYMBOLIC*                16  ignored      ignored      optional
 * DT_REL                      17  d_ptr        mandatory    optional
 * DT_RELSZ                    18  d_val        mandatory    optional
 * DT_RELENT                   19  d_val        mandatory    optional
 * DT_PLTREL                   20  d_val        optional     optional
 * DT_DEBUG                    21  d_ptr        optional     ignored
 * DT_TEXTREL*                 22  ignored      optional     optional
 * DT_JMPREL                   23  d_ptr        optional     optional
 * DT_BIND_NOW*                24  ignored      optional     optional
 * DT_INIT_ARRAY               25  d_ptr        optional     optional
 * DT_FINI_ARRAY               26  d_ptr        optional     optional
 * DT_INIT_ARRAYSZ             27  d_val        optional     optional
 * DT_FINI_ARRAYSZ             28  d_val        optional     optional
 * DT_RUNPATH                  29  d_val        optional     optional
 * DT_FLAGS                    30  d_val        optional     optional
 * DT_ENCODING                 32  unspecified  unspecified  unspecified
 * DT_PREINIT_ARRAY            32  d_ptr        optional     ignored
 * DT_PREINIT_ARRAYSZ          33  d_val        optional     ignored
 * DT_LOOS             0x6000000D  unspecified  unspecified  unspecified
 * DT_HIOS             0x6ffff000  unspecified  unspecified  unspecified
 * DT_LOPROC           0x70000000  unspecified  unspecified  unspecified
 * DT_HIPROC           0x7fffffff  unspecified  unspecified  unspecified
 * "*" Signifies an entry that is at level 2.
 * </pre>
 * <p>
 * Read more about dynamic sections at <a
href="https://refspecs.linuxbase.org/elf/gabi4+/ch5.dynamic.html#dynamic_section">Dyn
amic Section</a>.
 */
public class ElfDynamicSection extends ElfSection {

    /**
     * An entry with a DT_NULL tag marks the end of the _DYNAMIC array.
     */
    public static final int DT_NULL = 0;
    /**
     * This element holds the string table offset of a null-terminated string, giving
the
     * name of a needed library. The offset is an index into the table recorded in
the
     * {@link #DT_STRTAB} code.
     * <p>
     * See <a
href="https://refspecs.linuxbase.org/elf/gabi4+/ch5.dynamic.html#shobj_dependencies">
Shared Object Dependencies</a> for more information about these names.
     * <p>
     * The dynamic array may contain multiple entries with this type.
     * <p>
     * These entries' relative order is significant, though their relation to entries
of other types is not.
     */
    public static final int DT_NEEDED = 1;
    public static final int DT_PLTRELSZ = 2;
    public static final int DT_PLTGOT = 3;
    public static final int DT_HASH = 4;
    /**
     * DT_STRTAB entry holds the address, not offset, of the dynamic string table.
     */
    public static final int DT_STRTAB = 5;
    public static final int DT_SYMTAB = 6;
    public static final int DT_RELA = 7;
    public static final int DT_RELASZ = 8;
    public static final int DT_RELAENT = 9;
    /**
     * The size in bytes of the {@link #DT_STRTAB} string table.
     */
    public static final int DT_STRSZ = 10;
    public static final int DT_SYMENT = 11;
    public static final int DT_INIT = 12;
    public static final int DT_FINI = 13;
    public static final int DT_SONAME = 14;
    public static final int DT_RPATH = 15;
    public static final int DT_SYMBOLIC = 16;
    public static final int DT_REL = 17;
    public static final int DT_RELSZ = 18;
    public static final int DT_RELENT = 19;
    public static final int DT_PLTREL = 20;
    public static final int DT_DEBUG = 21;
    public static final int DT_TEXTREL = 22;
    public static final int DT_JMPREL = 23;
    public static final int DT_BIND_NOW = 24;
    public static final int DT_INIT_ARRAY = 25;
    public static final int DT_FINI_ARRAY = 26;
    public static final int DT_INIT_ARRAYSZ = 27;
    public static final int DT_FINI_ARRAYSZ = 28;
    public static final int DT_RUNPATH = 29;
    public static final int DT_FLAGS = 30;
    public static final int DT_PREINIT_ARRAY = 32;
    public static final int DT_GNU_HASH = 0x6ffffef5;
    public static final int DT_FLAGS_1 = 0x6ffffffb;
    public static final int DT_VERDEF = 0x6ffffffc; /* Address of version definition
*/
    public static final int DT_VERDEFNUM = 0x6ffffffd; /* Number of version
definitions */
    public static final int DT_VERNEEDED = 0x6ffffffe;
    public static final int DT_VERNEEDNUM = 0x6fffffff;

    public static final int DF_ORIGIN = 0x1;
    public static final int DF_SYMBOLIC = 0x2;
    public static final int DF_TEXTREL = 0x4;
    public static final int DF_BIND_NOW = 0x8;

    /**
     * Set RTLD_NOW for this object.
     */
    public static final int DF_1_NOW = 0x00000001;
    /**
     * Set RTLD_GLOBAL for this object.
     */
    public static final int DF_1_GLOBAL = 0x00000002;
    /**
     * Set RTLD_GROUP for this object.
     */
    public static final int DF_1_GROUP = 0x00000004;
    /**
     * Set RTLD_NODELETE for this object.
     */
    public static final int DF_1_NODELETE = 0x00000008;
    public static final int DF_1_LOADFLTR = 0x00000010;
    public static final int DF_1_INITFIRST = 0x00000020;
    /**
     * Object can not be used with dlopen(3)
     */
    public static final int DF_1_NOOPEN = 0x00000040;
```

```java
public static final int DF_1_ORIGIN = 0x00000080;
public static final int DF_1_DIRECT = 0x00000100;
public static final int DF_1_TRANS = 0x00000200;
public static final int DF_1_INTERPOSE = 0x00000400;
public static final int DF_1_NODEFLIB = 0x00000800;
/**
 * Object cannot be dumped with dldump(3)
 */
public static final int DF_1_NODUMP = 0x00001000;
public static final int DF_1_CONFALT = 0x00002000;
public static final int DF_1_ENDFILTEE = 0x00004000;
public static final int DF_1_DISPRELDNE = 0x00008000;
public static final int DF_1_DISPRELPND = 0x00010000;
public static final int DF_1_NODIRECT = 0x00020000;
public static final int DF_1_IGNMULDEF = 0x00040000;
public static final int DF_1_NOKSYMS = 0x00080000;
public static final int DF_1_NOHDR = 0x00100000;
public static final int DF_1_EDITED = 0x00200000;
public static final int DF_1_NORELOC = 0x00400000;
public static final int DF_1_SYMINTPOSE = 0x00800000;
public static final int DF_1_GLOBAUDIT = 0x01000000;
public static final int DF_1_SINGLETON = 0x02000000;
public static final int DF_1_STUB = 0x04000000;
public static final int DF_1_PIE = 0x08000000;

/**
 * For the {@link #DT_STRTAB}. Mandatory.
 */
public long dt_strtab_offset;

/**
 * For the {@link #DT_STRSZ}. Mandatory.
 */
public int dt_strtab_size;

private MemoizedObject<ElfStringTable> dtStringTable;
public final List<ElfDynamicStructure> entries = new ArrayList<>();

/**
 * An entry in the {@link #entries} of a {@link ElfDynamicSection}.
 * <p>
 * In the elf.h header file this represents either of the following structures:
 *
 * <pre>
 * typedef struct {
 *     Elf32_Sword d_tag;
 *     union {
 *         Elf32_Word      d_val;
 *         Elf32_Addr      d_ptr;
 *         Elf32_Off       d_off;
 *     } d_un;
 * } Elf32_Dyn;
 *
 * typedef struct {
 *     Elf64_Xword d_tag;
 *     union {
 *         Elf64_Xword d_val;
 *         Elf64_Addr d_ptr;
 *     } d_un;
 * } Elf64_Dyn;
 * </pre>
 */
```

```
    public static class ElfDynamicStructure {
        public ElfDynamicStructure(long d_tag, long d_val_or_ptr) {
            this.tag = d_tag;
            this.d_val_or_ptr = d_val_or_ptr;
        }

        /**
         * A tag value whose value defines how to interpret {@link #d_val_or_ptr}.
         * <p>
         * One of the DT_* constants in {@link ElfDynamicSection}.
         */
        public final long tag;
        /**
         * A field whose value is to be interpreted as specified by the {@link #tag}.
         */
        public final long d_val_or_ptr;

        @Override
        public int hashCode() {
            final int prime = 31;
            int result = 1;
            result = prime * result + (int) (tag ^ (tag >>> 32));
            result = prime * result + (int) (d_val_or_ptr ^ (d_val_or_ptr >>> 32));
            return result;
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj) return true;
            if (obj == null) return false;
            if (getClass() != obj.getClass()) return false;
            ElfDynamicStructure other = (ElfDynamicStructure) obj;
            if (tag != other.tag) return false;
            return d_val_or_ptr == other.d_val_or_ptr;
        }

        @Override
        public String toString() {
            return "ElfDynamicSectionEntry{tag=" + tag + ", d_val_or_ptr=" +
d_val_or_ptr + "}";
        }
    }

    public ElfDynamicSection(final ElfParser parser, ElfSectionHeader header) {
        super(parser, header);

        parser.seek(header.section_offset);
        int numEntries = (int) (header.size / 8);

        // Except for the DT_NULL element at the end of the array, and the relative
order of DT_NEEDED elements, entries
        // may appear in any order. So important to use lazy evaluation to only
evaluating e.g. DT_STRTAB after the
        // necessary DT_STRSZ is read.
        loop:
        for (int i = 0; i < numEntries; i++) {
            long d_tag = parser.readIntOrLong();
            final long d_val_or_ptr = parser.readIntOrLong();
            entries.add(new ElfDynamicStructure(d_tag, d_val_or_ptr));
            switch ((int) d_tag) {
                case DT_NULL:
                    // A DT_NULL element ends the array (may be following DT_NULL
values, but no need to look at them).
                    break loop;
                case DT_STRTAB: {
                    dtStringTable = new MemoizedObject<ElfStringTable>() {
                        @Override
                        protected ElfStringTable computeValue() throws ElfException {
                            long fileOffsetForStringTable =
parser.virtualMemoryAddrToFileOffset(d_val_or_ptr);
                            return new ElfStringTable(parser,
fileOffsetForStringTable, dt_strtab_size, null); // FIXME: null header
                        }
                    };
                    dt_strtab_offset = d_val_or_ptr;
                }
                break;
                case DT_STRSZ:
                    if (d_val_or_ptr > Integer.MAX_VALUE) throw new ElfException("Too
large DT_STRSZ: " + d_val_or_ptr);
                    dt_strtab_size = (int) d_val_or_ptr;
                    break;
            }
        }

    }

    private ElfDynamicStructure firstEntryWithTag(long desiredTag) {
        for (ElfDynamicStructure entry : this.entries) {
            if (entry.tag == desiredTag) return entry;
        }
        return null;
    }

    public List<String> getNeededLibraries() throws ElfException {
        ElfStringTable stringTable = dtStringTable.getValue();
        List<String> result = new ArrayList<>();
        for (ElfDynamicStructure entry : this.entries) {
            if (entry.tag == DT_NEEDED) result.add(stringTable.get((int)
entry.d_val_or_ptr));
        }
        return result;
    }

    public String getRunPath() {
        ElfDynamicStructure runPathEntry = firstEntryWithTag(DT_RUNPATH);
        return runPathEntry == null ? null : dtStringTable.getValue().get((int)
runPathEntry.d_val_or_ptr);
    }

    public long getFlags() {
        ElfDynamicStructure flagsEntry = firstEntryWithTag(DT_FLAGS);
        return flagsEntry == null ? 0 : flagsEntry.d_val_or_ptr;
    }

    public long getFlags1() {
        ElfDynamicStructure flagsEntry = firstEntryWithTag(DT_FLAGS_1);
        return flagsEntry == null ? 0 : flagsEntry.d_val_or_ptr;
    }

    @Override
    public String toString() {
        return "ElfDynamicStructure{entries=" + this.entries + "}";
    }
}
```

ElfException.java

```
package net.fornwall.jelf;

/**
 * Generic exception class for all exceptions which occur in this package. Since
 * there is no mechanism built into this library for recovering from errors, the
 * best clients can do is display the error string.
 */
public class ElfException extends RuntimeException {

   private static final long serialVersionUID = 1L;

   public ElfException(String message) {
      super(message);
   }

   public ElfException(Throwable cause) {
      super(cause);
   }

   public ElfException(String message, Throwable cause) {
      super(message, cause);
   }

}
```

ElfFile.java

```java
package net.fornwall.jelf;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.nio.MappedByteBuffer;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 * An ELF (Executable and Linkable Format) file that can be a relocatable,
executable, shared or core file.
 * <p>
 * Use one of the following methods to parse input to get an instance of this class:
 * <ul>
 *     <li>{@link #from(File)}</li>
 *     <li>{@link #from(byte[])}</li>
 *     <li>{@link #from(InputStream)}</li>
 *     <li>{@link #from(MappedByteBuffer)}</li>
 * </ul>
 * <p>
 * Resources about ELF files:
 * <ul>
 *  <li>http://man7.org/linux/man-pages/man5/elf.5.html</li>
 *  <li>http://en.wikipedia.org/wiki/Executable_and_Linkable_Format</li>
 *  <li>http://www.ibm.com/developerworks/library/l-dynamic-libraries/</li>
 *  <li>http://downloads.openwatcom.org/ftp/devel/docs/elf-64-gen.pdf</li>
 * </ul>
 */
public final class ElfFile {

    /**
     * Relocatable file type. A possible value of {@link #e_type}.
     */
    public static final int ET_REL = 1;
    /**
     * Executable file type. A possible value of {@link #e_type}.
     */
    public static final int ET_EXEC = 2;
    /**
     * Shared object file type. A possible value of {@link #e_type}.
     */
    public static final int ET_DYN = 3;
    /**
     * Core file file type. A possible value of {@link #e_type}.
     */
    public static final int ET_CORE = 4;

    /**
```

```
     * 32-bit objects.
     */
    public static final byte CLASS_32 = 1;
    /**
     * 64-bit objects.
     */
    public static final byte CLASS_64 = 2;

    /**
     * LSB data encoding.
     */
    public static final byte DATA_LSB = 1;
    /**
     * MSB data encoding.
     */
    public static final byte DATA_MSB = 2;

    /**
     * No architecture type.
     */
    public static final int ARCH_NONE = 0;
    /**
     * AT&amp;T architecture type.
     */
    public static final int ARCH_ATT = 1;
    /**
     * SPARC architecture type.
     */
    public static final int ARCH_SPARC = 2;
    /**
     * Intel 386 architecture type.
     */
    public static final int ARCH_i386 = 3;
    /**
     * Motorola 68000 architecture type.
     */
    public static final int ARCH_68k = 4;
    /**
     * Motorola 88000 architecture type.
     */
    public static final int ARCH_88k = 5;
    /**
     * Intel 860 architecture type.
     */
    public static final int ARCH_i860 = 7;
    /**
     * MIPS architecture type.
     */
    public static final int ARCH_MIPS = 8;
    public static final int ARCH_ARM = 0x28;
    public static final int ARCH_X86_64 = 0x3E;
    public static final int ARCH_AARCH64 = 0xB7;

    /**
     * Identifies the object file type. One of the ET_* constants in the class.
     */
    public final short e_type; // Elf32_Half
    /**
     * Byte identifying the size of objects, either {@link #CLASS_32} or {link
{@value #CLASS_64} .
     */
    public final byte objectSize;
    /**
     * Returns a byte identifying the data encoding of the processor specific data.
This byte will be either
     * DATA_INVALID, DATA_LSB or DATA_MSB.
     */
    public final byte encoding;

    public final byte elfVersion;
    public final byte abi;
    public final byte abiVersion;

    /**
     * The required architecture. One of the ARCH_* constants in the class.
     */
    public final short arch; // Elf32_Half
    /**
     * Version
     */
    public final int version; // Elf32_Word
    /**
     * Virtual address to which the system first transfers control. If there is no
entry point for the file the value is
     * 0.
     */
    public final long entry_point; // Elf32_Addr
    /**
     * e_phoff. Program header table offset in bytes. If there is no program header
table the value is 0.
     */
    public final long ph_offset; // Elf32_Off
    /**
     * e_shoff. Section header table offset in bytes. If there is no section header
table the value is 0.
     */
    public final long sh_offset; // Elf32_Off
    /**
     * e_flags. Processor specific flags.
     */
    public final int flags; // Elf32_Word
    /**
     * e_ehsize. ELF header size in bytes.
     */
    public final short eh_size; // Elf32_Half
    /**
     * e_phentsize. Size of one entry in the file's program header table in bytes.
All entries are the same size.
     */
    public final short ph_entry_size; // Elf32_Half
    /**
     * e_phnum. Number of {@link ElfSegment} entries in the program header table, 0
if no entries.
     */
    public final short num_ph; // Elf32_Half
    /**
     * e_shentsize. Section header entry size in bytes - all entries are the same
size.
     */
    public final short sh_entry_size; // Elf32_Half
    /**
     * e_shnum. Number of entries in the section header table, 0 if no entries.
     */
```

```java
    public final short num_sh; // Elf32_Half

    /**
     * Elf{32,64}_Ehdr#e_shstrndx. Index into the section header table associated
with the section name string table.
     * SH_UNDEF if there is no section name string table.
     */
    private short sh_string_ndx; // Elf32_Half

    /**
     * MemoizedObject array of section headers associated with this ELF file.
     */
    private MemoizedObject<ElfSection>[] sections;
    /**
     * MemoizedObject array of program headers associated with this ELF file.
     */
    private MemoizedObject<ElfSegment>[] programHeaders;

    /**
     * Used to cache symbol table lookup.
     */
    private ElfSymbolTableSection symbolTableSection;
    /**
     * Used to cache dynamic symbol table lookup.
     */
    private ElfSymbolTableSection dynamicSymbolTableSection;

    private ElfDynamicSection dynamicSection;

    /**
     * Returns the section header at the specified index. The section header at index
0 is defined as being a undefined
     * section.
     */
    public ElfSection getSection(int index) throws ElfException {
        return sections[index].getValue();
    }

    public List<ElfSection> sectionsOfType(int sectionType) throws ElfException {
        if (num_sh < 2) return Collections.emptyList();
        List<ElfSection> result = new ArrayList<>();
        for (int i = 1; i < num_sh; i++) {
            ElfSection section = getSection(i);
            if (section.header.type == sectionType) {
                result.add(section);
            }
        }
        return result;
    }


    /**
     * Returns the section header string table associated with this ELF file.
     */
    public ElfStringTable getSectionNameStringTable() throws ElfException {
        return (ElfStringTable) getSection(sh_string_ndx);
    }

    /**
     * Returns the string table associated with this ELF file.
     */
    public ElfStringTable getStringTable() throws ElfException {
        return findStringTableWithName(ElfSectionHeader.NAME_STRTAB);
    }

    /**
     * Returns the dynamic symbol table associated with this ELF file, or null if one
does not exist.
     */
    public ElfStringTable getDynamicStringTable() throws ElfException {
        return findStringTableWithName(ElfSectionHeader.NAME_DYNSTR);
    }

    private ElfStringTable findStringTableWithName(String tableName) throws
ElfException {
        // Loop through the section header and look for a section
        // header with the name "tableName". We can ignore entry 0
        // since it is defined as being undefined.
        return (ElfStringTable) firstSectionByName(tableName);
    }

    /**
     * The {@link ElfSectionHeader#SHT_SYMTAB} section (of which there may be only
one), if any.
     */
    public ElfSymbolTableSection getSymbolTableSection() throws ElfException {
        return (symbolTableSection != null) ? symbolTableSection :
(symbolTableSection = (ElfSymbolTableSection)
firstSectionByType(ElfSectionHeader.SHT_SYMTAB));
    }

    /**
     * The {@link ElfSectionHeader#SHT_DYNSYM} section (of which there may be only
one), if any.
     */
    public ElfSymbolTableSection getDynamicSymbolTableSection() throws ElfException {
        return (dynamicSymbolTableSection != null) ? dynamicSymbolTableSection :
(dynamicSymbolTableSection = (ElfSymbolTableSection)
firstSectionByType(ElfSectionHeader.SHT_DYNSYM));
    }

    /**
     * The {@link ElfSectionHeader#SHT_DYNAMIC} section (of which there may be only
one). Named ".dynamic".
     */
    public ElfDynamicSection getDynamicSection() {
        return (dynamicSection != null) ? dynamicSection : (dynamicSection =
(ElfDynamicSection) firstSectionByType(ElfSectionHeader.SHT_DYNAMIC));
    }

    public ElfSection firstSectionByType(int type) throws ElfException {
        for (int i = 1; i < num_sh; i++) {
            ElfSection sh = getSection(i);
            if (sh.header.type == type) return sh;
        }
        return null;
    }

    public <T extends ElfSection> T firstSectionByType(Class<T> type) throws
ElfException {
        for (int i = 1; i < num_sh; i++) {
            ElfSection sh = getSection(i);
            if (type.isInstance(sh)) return (T) sh;
        }
```

```
            return null;
        }

        public ElfSection firstSectionByName(String sectionName) throws ElfException {
            for (int i = 1; i < num_sh; i++) {
                ElfSection sh = getSection(i);
                if (sectionName.equals(sh.header.getName())) return sh;
            }
            return null;
        }

        /**
         * Returns the elf symbol with the specified name or null if one is not found.
         */
        public ElfSymbol getELFSymbol(String symbolName) throws ElfException, IOException
{
            if (symbolName == null) return null;

            // Check dynamic symbol table for symbol name.
            ElfSymbolTableSection sh = getDynamicSymbolTableSection();
            if (sh != null) {
                int numSymbols = sh.symbols.length;
                for (int i = 0; i < Math.ceil(numSymbols / 2); i++) {
                    ElfSymbol symbol = sh.symbols[i];
                    if (symbolName.equals(symbol.getName())) {
                        return symbol;
                    } else if (symbolName.equals((symbol = sh.symbols[numSymbols - 1 -
i]).getName())) {
                        return symbol;
                    }
                }
            }

            // Check symbol table for symbol name.
            sh = getSymbolTableSection();
            if (sh != null) {
                int numSymbols = sh.symbols.length;
                for (int i = 0; i < Math.ceil(numSymbols / 2); i++) {
                    ElfSymbol symbol = sh.symbols[i];
                    if (symbolName.equals(symbol.getName())) {
                        return symbol;
                    } else if (symbolName.equals((symbol = sh.symbols[numSymbols - 1 -
i]).getName())) {
                        return symbol;
                    }
                }
            }
            return null;
        }

        /**
         * Returns the elf symbol with the specified address or null if one is not found.
'address' is relative to base of
         * shared object for .so's.
         */
        public ElfSymbol getELFSymbol(long address) throws ElfException {
            // Check dynamic symbol table for address.
            ElfSymbol symbol;
            long value;

            ElfSymbolTableSection sh = getDynamicSymbolTableSection();
            if (sh != null) {
```

```java
            int numSymbols = sh.symbols.length;
            for (int i = 0; i < numSymbols; i++) {
                symbol = sh.symbols[i];
                value = symbol.st_value;
                if (address >= value && address < value + symbol.st_size) return
symbol;
            }
        }

        // Check symbol table for symbol name.
        sh = getSymbolTableSection();
        if (sh != null) {
            int numSymbols = sh.symbols.length;
            for (int i = 0; i < numSymbols; i++) {
                symbol = sh.symbols[i];
                value = symbol.st_value;
                if (address >= value && address < value + symbol.st_size) return
symbol;
            }
        }
        return null;
    }

    public ElfSegment getProgramHeader(int index) {
        return programHeaders[index].getValue();
    }

    public static ElfFile from(InputStream in) throws IOException {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        int totalRead = 0;
        byte[] buffer = new byte[8096];
        boolean firstRead = true;
        while (true) {
            int readNow = in.read(buffer, totalRead, buffer.length - totalRead);
            if (readNow == -1) {
                return from(baos.toByteArray());
            } else {
                if (firstRead) {
                    // Abort early.
                    if (readNow < 4) {
                        throw new ElfException("Bad first read");
                    } else {
                        if (!(0x7f == buffer[0] && 'E' == buffer[1] && 'L' ==
buffer[2] && 'F' == buffer[3]))
                            throw new ElfException("Bad magic number for file");
                    }
                    firstRead = false;
                }
                baos.write(buffer, 0, readNow);
            }
        }
    }

    public static ElfFile from(File file) throws ElfException, IOException {
        byte[] buffer = new byte[(int) file.length()];
        try (FileInputStream in = new FileInputStream(file)) {
            int totalRead = 0;
            while (totalRead < buffer.length) {
                int readNow = in.read(buffer, totalRead, buffer.length - totalRead);
                if (readNow == -1) {
                    throw new ElfException("Premature end of file");
                } else {
                    totalRead += readNow;
                }
            }
        }
        return from(buffer);
    }

    public static ElfFile from(byte[] buffer) throws ElfException, IOException {
        return new ElfFile(new BackingFile(new ByteArrayInputStream(buffer)));
    }

    public static ElfFile from(MappedByteBuffer mappedByteBuffer) throws
ElfException, IOException {
        return new ElfFile(new BackingFile(mappedByteBuffer));
    }

    public final ElfParser parser;

    private ElfFile(BackingFile backingFile) throws ElfException, IOException {
        parser = new ElfParser(this, backingFile);

        byte[] ident = new byte[16];
        int bytesRead = parser.read(ident);
        if (bytesRead != ident.length)
            throw new ElfException("Error reading elf header (read " + bytesRead +
"bytes - expected to read " + ident.length + "bytes)");

        if (!(0x7f == ident[0] && 'E' == ident[1] && 'L' == ident[2] && 'F' ==
ident[3]))
            throw new ElfException("Bad magic number for file");

        objectSize = ident[4];
        if (!(objectSize == CLASS_32 || objectSize == CLASS_64))
            throw new ElfException("Invalid object size class: " + objectSize);
        encoding = ident[5];
        if (!(encoding == DATA_LSB || encoding == DATA_MSB)) throw new
ElfException("Invalid encoding: " + encoding);
        elfVersion = ident[6];
        if (elfVersion != 1) throw new ElfException("Invalid elf version: " +
elfVersion);
        abi = ident[7]; // EI_OSABI, target operating system ABI
        abiVersion = ident[8]; // EI_ABIVERSION, ABI version. Linux kernel (after at
least 2.6) has no definition of it.
        // ident[9-15] // EI_PAD, currently unused.

        e_type = parser.readShort();
        arch = parser.readShort();
        version = parser.readInt();
        entry_point = parser.readIntOrLong();
        ph_offset = parser.readIntOrLong();
        sh_offset = parser.readIntOrLong();
        flags = parser.readInt();
        eh_size = parser.readShort();
        ph_entry_size = parser.readShort();
        num_ph = parser.readShort();
        sh_entry_size = parser.readShort();
        num_sh = parser.readShort();
        if (num_sh == 0) {
            throw new ElfException("e_shnum is SHN_UNDEF(0), which is not supported
yet"
                    + " (the actual number of section header table entries is
contained in the sh_size field of the section header at index 0)");
```

```
        }
        sh_string_ndx = parser.readShort();
        if (sh_string_ndx == /* SHN_XINDEX= */0xffff) {
            throw new ElfException("e_shstrndx is SHN_XINDEX(0xffff), which is not
supported yet"
                    + " (the actual index of the section name string table section is
contained in the sh_link field of the section header at index 0)");
        }

        sections = MemoizedObject.uncheckedArray(num_sh);
        for (int i = 0; i < num_sh; i++) {
            final long sectionHeaderOffset = sh_offset + (i * sh_entry_size);
            sections[i] = new MemoizedObject<>() {
                @Override
                public ElfSection computeValue() throws ElfException {
                    ElfSectionHeader elfSectionHeader = new ElfSectionHeader(parser,
sectionHeaderOffset);
                    switch (elfSectionHeader.type) {
                        case ElfSectionHeader.SHT_DYNAMIC:
                            return new ElfDynamicSection(parser, elfSectionHeader);
                        case ElfSectionHeader.SHT_SYMTAB:
                        case ElfSectionHeader.SHT_DYNSYM:
                            return new ElfSymbolTableSection(parser,
elfSectionHeader);
                        case ElfSectionHeader.SHT_STRTAB:
                            return new ElfStringTable(parser,
elfSectionHeader.section_offset, (int) elfSectionHeader.size, elfSectionHeader);
                        case ElfSectionHeader.SHT_HASH:
                            return new ElfHashTable(parser, elfSectionHeader);
                        case ElfSectionHeader.SHT_NOTE:
                            return new ElfNoteSection(parser, elfSectionHeader);
                        case ElfSectionHeader.SHT_RELA:
                            return new ElfRelocationSection(parser,
elfSectionHeader);
                        case ElfSectionHeader.SHT_GNU_HASH:
                            return new ElfGnuHashTable(parser, elfSectionHeader);
                        default:
                            return new ElfSection(parser, elfSectionHeader);
                    }
                }
            };
        }

        programHeaders = MemoizedObject.uncheckedArray(num_ph);
        for (int i = 0; i < num_ph; i++) {
            final long programHeaderOffset = ph_offset + (i * ph_entry_size);
            programHeaders[i] = new MemoizedObject<ElfSegment>() {
                @Override
                public ElfSegment computeValue() {
                    return new ElfSegment(parser, programHeaderOffset);
                }
            };
        }
    }

    /**
     * The interpreter specified by the {@link ElfSegment#PT_INTERP} program header,
if any.
     */
    public String getInterpreter() throws IOException {
        for (MemoizedObject<ElfSegment> programHeader : programHeaders) {
            ElfSegment ph = programHeader.getValue();
            if (ph.type == ElfSegment.PT_INTERP) return ph.getIntepreter();
        }
        return null;
    }

}
```

ElfGnuHashTable

```
package net.fornwall.jelf;

/**
 * An ELF section containing a hash table for lookup of dynamic symbols.
 *
 * Has the section type {@link ElfSectionHeader#SHT_GNU_HASH}.
 *
 * Replaces {@link ElfHashTable} on almost all modern Linux systems.
 *
 * See https://flapenguin.me/2017/05/10/elf-lookup-dt-gnu-hash/
 */
public class ElfGnuHashTable extends ElfSection {

    private final ElfParser parser;
    private final int ELFCLASS_BITS;
    // The number of .dynsym symbols skipped.
    int symbolOffset;
    int bloomShift;
    long[] bloomFilter;
    int[] buckets;
    int[] chain;

    ElfGnuHashTable(ElfParser parser, ElfSectionHeader header) {
        super(parser, header);
        this.parser = parser;

        ELFCLASS_BITS = parser.elfFile.objectSize == ElfFile.CLASS_32 ? 32 : 64;

        parser.seek(header.section_offset);
        int numberOfBuckets = parser.readInt();
        symbolOffset = parser.readInt();
        int bloomSize = parser.readInt();
        bloomShift = parser.readInt();
        bloomFilter = new long[bloomSize];
        buckets = new int[numberOfBuckets];

        for (int i = 0; i < bloomSize; i++) {
            bloomFilter[i] = parser.readIntOrLong();
        }
        for (int i = 0; i < numberOfBuckets; i++) {
            buckets[i] = parser.readInt();
        }
        // The chain is initialized on first use in lookupSymbol() due to it
requiring .dynsym size.
    }

    ElfSymbol lookupSymbol(String symbolName, ElfSymbolTableSection symbolTable) {
        if (chain == null) {
            int chainSize = ((ElfSymbolTableSection)
parser.elfFile.firstSectionByType(ElfSectionHeader.SHT_DYNSYM)).symbols.length -
symbolOffset;
            chain = new int[chainSize];
            parser.seek(header.section_offset + 4*4 +
bloomFilter.length*(ELFCLASS_BITS/8) + buckets.length * 4);
            for (int i = 0; i < chainSize; i++) {
                chain[i] = parser.readInt();
            }
        }

        final int nameHash = gnuHash(symbolName);

        long word =
bloomFilter[(Integer.remainderUnsigned(Integer.divideUnsigned(nameHash,
ELFCLASS_BITS), bloomFilter.length))];
        long mask = 1L << (long) (Integer.remainderUnsigned(nameHash, ELFCLASS_BITS))
                | 1L << (long) (Integer.remainderUnsigned((nameHash >>> bloomShift),
ELFCLASS_BITS));

        if ((word & mask) != mask) {
            // If at least one bit is not set, a symbol is surely missing.
            return null;
        }

        int symix = buckets[Integer.remainderUnsigned(nameHash, buckets.length)];
        if (symix < symbolOffset) {
            return null;
        }

        while (true) {
            int hash = chain[symix - symbolOffset];

            if ((((long) nameHash)|1L) == (((long) hash)|1L)) {
                // The chain contains contiguous sequences of hashes for symbols
hashing to the same index,
                // with the lowest bit discarded (used to signal end of chain).
                ElfSymbol symbol = symbolTable.symbols[symix];
                if (symbolName.equals(symbol.getName())) return symbol;
            }
            ElfSymbol symbol = symbolTable.symbols[symix];

            if ((hash & 1) != 0) {
                // Chain ends with an element with the lowest bit set to 1.
                break;
            }

            symix++;
        }

        return null;
    }

    static int gnuHash(String name) {
        int h = 5381;
        int nameLength = name.length();
        for (int i = 0; i < nameLength; i++) {
            char c = name.charAt(i);
            h = (h << 5) + h + c;
        }
        return h;
    }
}
```

ElfHashTable

```java
package net.fornwall.jelf;

/**
 * An ELF section containing a hash table for lookup of dynamic symbols.
 *
 * Note that this has been replaced with {@link ElfGnuHashTable} on modern Linux
systems.
 *
 * See https://flapenguin.me/2017/04/24/elf-lookup-dt-hash/
 */
public class ElfHashTable extends ElfSection {

    private final int[] buckets;
    private final int[] chain;

    ElfHashTable(ElfParser parser, ElfSectionHeader header) {
        super(parser, header);

        parser.seek(header.section_offset);

        int num_buckets = parser.readInt();
        int num_chains = parser.readInt();

        buckets = new int[num_buckets];
        for (int i = 0; i < num_buckets; i++) {
            buckets[i] = parser.readInt();
        }

        chain = new int[num_chains];
        for (int i = 0; i < num_chains; i++) {
            chain[i] = parser.readInt();
        }

        // Make sure that the amount of bytes we were supposed to read
        // was what we actually read.
        int actual = num_buckets * 4 + num_chains * 4 + 8;
        if (header.size != actual) {
            throw new ElfException("Error reading string table (read " + actual +
"bytes, expected to read " + header.size + "bytes).");
        }
    }

    public ElfSymbol lookupSymbol(String name, ElfSymbolTableSection symbolTable) {
        long hashValue = elfHash(name);
        int index = buckets[(int) (hashValue % buckets.length)];
        while (true) {
            if (index == 0) return null;
            ElfSymbol symbol = symbolTable.symbols[index];
            if (name.equals(symbol.getName())) return symbol;
            index = chain[index];
        }
    }

    static long elfHash(String name) {
        long hash = 0;
        int nameLength = name.length();
        for (int i = 0; i < nameLength; i++) {
            hash = (hash << 4) + name.charAt(i);
            long x = hash & 0xF0000000L;
            if (x != 0) hash ^= (x >> 24);
            hash &= ~x;
        }
        return hash;
    }

}
```

ElfNoteSection.java

```
package net.fornwall.jelf;

import java.io.IOException;

class ElfNoteSection extends ElfSection {

    /**
     * A possible value of the {@link #type} where the description should contain
{@link GnuAbiDescriptor}.
     */
    public static final int NT_GNU_ABI_TAG = 1;
    /**
     * A possible value of the {@link #type} for a note containing synthetic hwcap
information.
     *
     * The descriptor begins with two words:
     *    word 0: number of entries
     *    word 1: bitmask of enabled entries
     *    Then follow variable-length entries, one byte followed by a '\0'-terminated
hwcap name string.  The byte gives the bit
     *    number to test if enabled, (1U << bit) & bitmask.
     */
    public static final int NT_GNU_HWCAP = 2;
    /**
     * A possible value of the {@link #type} for a note containing build ID bits as
generated by "ld --build-id".
     *
     * The descriptor consists of any nonzero number of bytes.
     */
    public static final int NT_GNU_BUILD_ID = 3;

    /**
     * A possible value of the {@link #type} for a note containing a version string
generated by GNU gold.
     */
    public static final int NT_GNU_GOLD_VERSION = 4;

    /**
     * The descriptor content of a link {@link #NT_GNU_ABI_TAG} type note.
     *
     * Accessible in {@link #descriptorAsGnuAbi()}.
     */
    public final static class GnuAbiDescriptor {

        /** A possible value of {@link #operatingSystem}. */
        public static final int ELF_NOTE_OS_LINUX = 0;
        /** A possible value of {@link #operatingSystem}. */
        public static final int ELF_NOTE_OS_GNU = 1;
        /** A possible value of {@link #operatingSystem}. */
        public static final int ELF_NOTE_OS_SOLARIS2 = 2;
        /** A possible value of {@link #operatingSystem}. */
        public static final int ELF_NOTE_OS_FREEBSD = 3;

        /** One of the ELF_NOTE_OS_* constants in this class. */
        public final int operatingSystem;
        /** Major version of the required ABI. */
        public final int majorVersion;
        /** Minor version of the required ABI. */
        public final int minorVersion;
        /** Subminor version of the required ABI. */
        public final int subminorVersion;

        public GnuAbiDescriptor(int operatingSystem, int majorVersion, int
minorVersion, int subminorVersion) {
            this.operatingSystem = operatingSystem;
            this.majorVersion = majorVersion;
            this.minorVersion = minorVersion;
            this.subminorVersion = subminorVersion;
        }
    }

    public final /* uint32_t */ int nameSize;
    public final /* uint32_t */ int descriptorSize;
    public final /* uint32_t */ int type;
    private String name;
    private byte[] descriptorBytes;
    private final GnuAbiDescriptor gnuAbiDescriptor;

    ElfNoteSection(ElfParser parser, ElfSectionHeader header) throws ElfException {
        super(parser, header);

        parser.seek(header.section_offset);
        nameSize = parser.readInt();
        descriptorSize = parser.readInt();
        type = parser.readInt();
        byte[] nameBytes = new byte[nameSize];
        descriptorBytes = new byte[descriptorSize];
        int bytesRead = parser.read(nameBytes);
        if (bytesRead != nameSize) {
            throw new ElfException("Error reading note name (read=" + bytesRead + ",
expected=" + nameSize + ")");
        }
        parser.skip(bytesRead % 4);

        switch (type) {
            case NT_GNU_ABI_TAG:
                gnuAbiDescriptor = new GnuAbiDescriptor(parser.readInt(),
parser.readInt(), parser.readInt(), parser.readInt());
                break;
            default:
                gnuAbiDescriptor = null;
        }

        bytesRead = parser.read(descriptorBytes);
        if (bytesRead != descriptorSize) {
            throw new ElfException("Error reading note name (read=" + bytesRead + ",
expected=" + descriptorSize + ")");
        }

        name = new String(nameBytes, 0, nameSize-1); // unnecessary trailing 0
    }
    String getName() {
        return name;
    }

    byte[] descriptorBytes() {
        return descriptorBytes;
    }

    public String descriptorAsString() {
        return new String(descriptorBytes);
    }

    public GnuAbiDescriptor descriptorAsGnuAbi() {
        return gnuAbiDescriptor;
    }

}
```

ElfParser.java

```
package net.fornwall.jelf;

/**
 * Package internal class used for parsing ELF files.
 */
public class ElfParser {

    final ElfFile elfFile;
    private final BackingFile backingFile;
    private long readBytes;

    ElfParser(ElfFile elfFile, BackingFile backingFile) {
        this.elfFile = elfFile;
        this.backingFile = backingFile;
    }

    public void seek(long offset) {
        readBytes = 0;
        backingFile.seek(offset);
    }

    public void skip(int bytesToSkip) {
        readBytes = 0;
        backingFile.skip(bytesToSkip);
    }

    public long getReadBytes() {
        return readBytes;
    }

    /**
     * Signed byte utility functions used for converting from big-endian (MSB) to
little-endian (LSB).
     */
    short byteSwap(short arg) {
        return (short) ((arg << 8) | ((arg >>> 8) & 0xFF));
    }

    int byteSwap(int arg) {
        return ((byteSwap((short) arg)) << 16) | (((byteSwap((short) (arg >>> 16))))
& 0xFFFF);
    }

    long byteSwap(long arg) {
        return ((((long) byteSwap((int) arg)) << 32) | (((long) byteSwap((int) (arg
>>> 32))) & 0xFFFFFFFF));
    }

    short readUnsignedByte() {
        readBytes++;
        return backingFile.readUnsignedByte();
    }

    public short readShort() throws ElfException {
        int ch1 = readUnsignedByte();
        int ch2 = readUnsignedByte();
        short val = (short) ((ch1 << 8) + (ch2 << 0));
        if (elfFile.encoding == ElfFile.DATA_LSB) val = byteSwap(val);
        return val;
    }

    public int readInt() throws ElfException {
        int ch1 = readUnsignedByte();
        int ch2 = readUnsignedByte();
        int ch3 = readUnsignedByte();
        int ch4 = readUnsignedByte();
        int val = ((ch1 << 24) + (ch2 << 16) + (ch3 << 8) + (ch4));

        if (elfFile.encoding == ElfFile.DATA_LSB) val = byteSwap(val);
        return val;
    }

    public long readLong() {
        int ch1 = readUnsignedByte();
        int ch2 = readUnsignedByte();
        int ch3 = readUnsignedByte();
        int ch4 = readUnsignedByte();
        int val1 = ((ch1 << 24) + (ch2 << 16) + (ch3 << 8) + (ch4 << 0));
        int ch5 = readUnsignedByte();
        int ch6 = readUnsignedByte();
        int ch7 = readUnsignedByte();
        int ch8 = readUnsignedByte();
        int val2 = ((ch5 << 24) + (ch6 << 16) + (ch7 << 8) + (ch8 << 0));

        long val = ((long) (val1) << 32) + (val2 & 0xFFFFFFFFL);
        if (elfFile.encoding == ElfFile.DATA_LSB) val = byteSwap(val);
        return val;
    }

    /**
     * Read four-byte int or eight-byte long depending on if {@link
ElfFile#objectSize}.
     */
    public long readIntOrLong() {
        return elfFile.objectSize == ElfFile.CLASS_32 ? readInt() : readLong();
    }
    /**
     * Returns a big-endian unsigned representation of the int.
     */
    public long unsignedByte(int arg) {
        long val;
        if (arg >= 0) {
            val = arg;
        } else {
            val = (unsignedByte((short) (arg >>> 16)) << 16) | ((short) arg);
        }
        return val;
    }

    /**
     * Find the file offset from a virtual address by looking up the {@link
ElfSegment} segment containing the
     * address and computing the resulting file offset.
     */
    long virtualMemoryAddrToFileOffset(long address) {
        for (int i = 0; i < elfFile.num_ph; i++) {
            ElfSegment ph = elfFile.getProgramHeader(i);
            if (address >= ph.virtual_address && address < (ph.virtual_address +
ph.mem_size)) {
                long relativeOffset = address - ph.virtual_address;
                if (relativeOffset >= ph.file_size)
                    throw new ElfException("Can not convert virtual memory address "
+ Long.toHexString(address) + " to file offset -" + " found segment " + ph
                            + " but address maps to memory outside file range");
                return ph.offset + relativeOffset;
            }
        }
        throw new ElfException("Cannot find segment for address " +
Long.toHexString(address));
    }

    public int read(byte[] data) {
        return backingFile.read(data);
    }

}
```

ElfRelocationSection.java

```
package net.fornwall.jelf;

public class ElfRelocationSection extends ElfSection {

    public ElfRelocationSection(ElfParser parser, ElfSectionHeader header) {
        super(parser, header);

        int num_entries = (int) (header.size / header.entry_size);
    }

}
```

ElfSection.java

```java
package net.fornwall.jelf;

public class ElfSection {
    public final ElfSectionHeader header;
    private final ElfParser parser;

    public ElfSection(ElfParser parser, ElfSectionHeader header) {
        this.header = header;
        this.parser = parser;
    }

    public byte[] rawSection() {
        parser.seek(header.section_offset);
        byte[] data = new byte[(int) header.size];
        parser.read(data);
        return data;
    }
}
```

ElfSectionHeader.java

```java
package net.fornwall.jelf;

import java.io.IOException;

/**
 * Class corresponding to the Elf32_Shdr/Elf64_Shdr struct.
 *
 * <p>
 * An object file's section header table lets one locate all the file's sections. The 
section header table is an array
 * of Elf32_Shdr or Elf64_Shdr structures. A section header table index is a 
subscript into this array. The ELF header's
 * {@link ElfFile#sh_offset e_shoff member} gives the byte offset from the beginning 
of the file to the section header
 * table with each section header entry being {@link ElfFile#sh_entry_size 
e_shentsize} bytes big.
 *
 * <p>
 * {@link ElfFile#num_sh e_shnum} normally tells how many entries the section header 
table contains, but if the number
 * of sections is greater than or equal to SHN_LORESERVE (0xff00), e_shnum has the 
value SHN_UNDEF (0) and the actual
 * number of section header table entries is contained in the sh_size field of the 
section header at index 0 (otherwise,
 * the sh_size member of the initial entry contains 0).
 *
 * <p>
 * Some section header table indexes are reserved in contexts where index size is 
restricted, for example, the st_shndx
 * member of a symbol table entry and the e_shnum and e_shstrndx members of the ELF 
header. In such contexts, the
 * reserved values do not represent actual sections in the object file. Also in such 
contexts, an escape value indicates
 * that the actual section index is to be found elsewhere, in a larger field.
 */
public class ElfSectionHeader {
    /**
     * Marks the section header as inactive; it does not have an associated section.
Other members of the section header
     * have undefined values.
     */
    public static final int SHT_NULL = 0;
    /**
     * Section holds information defined by the program.
     */
    public static final int SHT_PROGBITS = 1;
    /**
     * The {@link #type} value for a section containing complete symbol table
information necessary for link editing.
     * <p>
     * See {@link ElfSymbolTableSection}, which is the class representing sections of
this type, for more information.
     */
    public static final int SHT_SYMTAB = 2;
    /**
     * Section holds string table information.
     */
    public static final int SHT_STRTAB = 3;
    /**
     * Section holds relocation entries with explicit addends.
     */
    public static final int SHT_RELA = 4;
    /**
     * Section holds symbol hash table.
     */
    public static final int SHT_HASH = 5;
    /**
     * Section holds information for dynamic linking. Only one per ELF file. The
dynsym is allocable, and contains the
     * symbols needed to support runtime operation.
     */
    public static final int SHT_DYNAMIC = 6;
    /**
     * Section holds information that marks the file.
     */
    public static final int SHT_NOTE = 7;
    /**
     * Section occupies no space but resembles TYPE_PROGBITS.
     */
    public static final int SHT_NOBITS = 8;
    /**
     * Section holds relocation entries without explicit addends.
     */
    public static final int SHT_REL = 9;
    /**
     * Section is reserved but has unspecified semantics.
     */
    public static final int SHT_SHLIB = 10;
    /**
     * The {@link #type} value for a section containing a minimal set of symbols
needed for dynamic linking at runtime.
     * <p>
     * See {@link ElfSymbolTableSection}, which is the class representing sections of
this type, for more information.
     */
    public static final int SHT_DYNSYM = 11;
    public static final int SHT_INIT_ARRAY = 14;
    public static final int SHT_FINI_ARRAY = 15;
    public static final int SHT_PREINIT_ARRAY = 16;
    public static final int SHT_GROUP = 17;
    public static final int SHT_SYMTAB_SHNDX = 18;

    /**
     * A hash table for fast lookup of dynamic symbols.
     * <p>
     * See {@link ElfGnuHashTable}.
     */
    public static final int SHT_GNU_HASH = 0x6ffffff6;
    public static final int SHT_GNU_verdef = 0x6ffffffd;
    public static final int SHT_GNU_verneed = 0x6ffffffe;
    public static final int SHT_GNU_versym = 0x6fffffff;

    /**
     * Lower bound of the range of indexes reserved for operating system-specific
semantics.
     */
    public static final int SHT_LOOS = 0x60000000;
    /**
     * Upper bound of the range of indexes reserved for operating system-specific
semantics.
     */
    public static final int SHT_HIOS = 0x6fffffff;
    /**
     * Lower bound of the range of indexes reserved for processor-specific semantics.
     */
    public static final int SHT_LOPROC = 0x70000000;
    /**
     * Upper bound of the range of indexes reserved for processor-specific semantics.
     */
    public static final int SHT_HIPROC = 0x7fffffff;
    /**
     * Lower bound of the range of indexes reserved for application programs.
     */
    public static final int SHT_LOUSER = 0x80000000;
    /**
     * Upper bound of the range of indexes reserved for application programs.
     */
    public static final int SHT_HIUSER = 0xffffffff;

    public static final short SHN_UNDEF = 0;
    public static final short SHN_LORESERVE = (short) 0xff00;
    public static final short SHN_LOPROC = (short) 0xff00;
    public static final short SHN_HIPROC = (short) 0xff1f;
    public static final short SHN_LOOS = (short) 0xff20;
    public static final short SHN_HIOS = (short) 0xff3f;
    public static final short SHN_ABS = (short) 0xfff1;
    public static final short SHN_COMMON = (short) 0xfff2;
    public static final short SHN_XINDEX = (short) 0xffff;
    public static final short SHN_HIRESERVE = (short) 0xffff;


    /**
     * Flag informing that this section contains data that should be writable during
process execution.
     */
    public static final int FLAG_WRITE = 0x1;
    /**
     * Flag informing that section occupies memory during process execution.
     */
    public static final int FLAG_ALLOC = 0x2;
    /**
     * Flag informing that section contains executable machine instructions.
     */
    public static final int FLAG_EXEC_INSTR = 0x4;
    /**
     * Flag informing that all the bits in the mask are reserved for processor
specific semantics.
     */
    public static final int FLAG_MASK = 0xf0000000;

    /**
     * Name for the section containing the string table.
     * <p>
     * This section contains a string table which contains names for symbol
structures
     * by being indexed by the {@link ElfSymbol#st_name} field.
     */
    public static final String NAME_STRTAB = ".strtab";
    /**
     * Name for the section containing the dynamic string table.
     */
    public static final String NAME_DYNSTR = ".dynstr";
    /**
     * Name for the section containing read-only initialized data.
     */
    public static final String NAME_RODATA = ".rodata";

    /**
     * Index into the section header string table which gives the name of the
section.
     */
    public final int name_ndx; // Elf32_Word or Elf64_Word - 4 bytes in both.
    /**
     * Section content and semantics.
     */
    public final int type; // Elf32_Word or Elf64_Word - 4 bytes in both.
    /**
     * Flags.
     */
    public final long flags; // Elf32_Word or Elf64_Xword.
    /**
     * sh_addr. If the section will be in the memory image of a process this will be
the address at which the first byte
     * of section will be loaded. Otherwise, this value is 0.
     */
    public final long address; // Elf32_Addr
    /**
     * Offset from beginning of file to first byte of the section.
     */
    public final long section_offset; // Elf32_Off
    /**
     * Size in bytes of the section. TYPE_NOBITS is a special case.
     */
    public final /* uint32_t */ long size;
    /**
     * Section header table index link.
     */
    public final /* uint32_t */ int link;
    /**
     * Extra information determined by the section type.
     */
    public final /* uint32_t */ int info;
```

```java
    /**
     * Address alignment constraints for the section.
     */
    public final /* uint32_t */ long address_alignment;
    /**
     * Size of a fixed-size entry, 0 if none.
     */
    public final long entry_size; // Elf32_Word

    private final ElfFile elfHeader;

    /**
     * Reads the section header information located at offset.
     */
    ElfSectionHeader(final ElfParser parser, long offset) {
        this.elfHeader = parser.elfFile;
        parser.seek(offset);

        name_ndx = parser.readInt();
        type = parser.readInt();
        flags = parser.readIntOrLong();
        address = parser.readIntOrLong();
        section_offset = parser.readIntOrLong();
        size = parser.readIntOrLong();
        link = parser.readInt();
        info = parser.readInt();
        address_alignment = parser.readIntOrLong();
        entry_size = parser.readIntOrLong();
    }

    /**
     * Returns the name of the section or null if the section has no name.
     */
    public String getName() {
        if (name_ndx == 0) return null;
        ElfStringTable tbl = elfHeader.getSectionNameStringTable();
        return tbl.get(name_ndx);
    }

    @Override
    public String toString() {
        return "ElfSectionHeader[name=" + getName() + ", type=0x" +
Long.toHexString(type) + "]";
    }

}
```

ElfSegment.java

```
package net.fornwall.jelf;

import java.io.IOException;

/**
 * Class corresponding to the Elf32_Phdr/Elf64_Phdr struct.
 *
 * An executable or shared object file's program header table is an array of
structures, each describing a segment or
 * other information the system needs to prepare the program for execution. An object
file segment contains one or more
 * sections. Program headers are meaningful only for executable and shared object
files. A file specifies its own
 * program header size with the ELF header's {@link ElfFile#ph_entry_size
e_phentsize} and {@link ElfFile#num_ph
 * e_phnum} members.
 *
 * http://www.sco.com/developers/gabi/latest/ch5.pheader.html#p_type
 * http://stackoverflow.com/questions/22612735/how-can-i-find-the-dynamic-libraries-
required-by-an-elf-binary-in-c
 */
public class ElfSegment {

   /** Type defining that the array element is unused. Other member values are
undefined. */
   public static final int PT_NULL = 0;
   /** Type defining that the array element specifies a loadable segment. */
   public static final int PT_LOAD = 1;
   /** The array element specifies dynamic linking information. */
   public static final int PT_DYNAMIC = 2;
   /**
    * The array element specifies the location and size of a null-terminated path
name to invoke as an interpreter.
    * Meaningful only for executable files (though it may occur for shared objects);
it may not occur more than once in
    * a file. If it is present, it must precede any loadable segment entry.
    */
   public static final int PT_INTERP = 3;
   /** The array element specifies the location and size of auxiliary information. */
   public static final int PT_NOTE = 4;
   /** This segment type is reserved but has unspecified semantics. */
   public static final int PT_SHLIB = 5;
   /**
    * The array element, if present, specifies the location and size of the program
header table itself, both in the
    * file and in the memory image of the program. This segment type may not occur
more than once in a file.
    */
   public static final int PT_PHDR = 6;
   /** The array element specifies the Thread-Local Storage template. */
   public static final int PT_TLS = 7;

   /** Lower bound of the range reserved for operating system-specific semantics. */
   public static final int PT_LOOS = 0x60000000;
   /** Upper bound of the range reserved for operating system-specific semantics. */
   public static final int PT_HIOS = 0x6fffffff;
   /** Lower bound of the range reserved for processor-specific semantics. */
   public static final int PT_LOPROC = 0x70000000;
   /** Upper bound of the range reserved for processor-specific semantics. */
   public static final int PT_HIPROC = 0x7fffffff;

   /** Elf{32,64}_Phdr#p_type. Kind of segment this element describes. */
   public final int type; // Elf32_Word/Elf64_Word - 4 bytes in both.
   /** Elf{32,64}_Phdr#p_offset. File offset at which the first byte of the segment
resides. */
   public final long offset; // Elf32_Off/Elf64_Off - 4 or 8 bytes.
   /** Elf{32,64}_Phdr#p_vaddr. Virtual address at which the first byte of the
segment resides in memory. */
   public final long virtual_address; // Elf32_Addr/Elf64_Addr - 4 or 8 bytes.
   /** Reserved for the physical address of the segment on systems where physical
addressing is relevant. */
   public final long physical_address; // Elf32_addr/Elf64_Addr - 4 or 8 bytes.

   /** Elf{32,64}_Phdr#p_filesz. File image size of segment in bytes, may be 0. */
   public final long file_size; // Elf32_Word/Elf64_Xword -
   /** Elf{32,64}_Phdr#p_memsz. Memory image size of segment in bytes, may be 0. */
   public final long mem_size; // Elf32_Word
   /**
    * Flags relevant to this segment. Values for flags are defined in
ELFSectionHeader.
    */
   public final int flags; // Elf32_Word
   public final long alignment; // Elf32_Word

   private MemoizedObject<String> ptInterpreter;

   ElfSegment(final ElfParser parser, long offset) {
      parser.seek(offset);
      if (parser.elfFile.objectSize == ElfFile.CLASS_32) {
         // 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;
         type = parser.readInt();
         this.offset = parser.readInt();
         virtual_address = parser.readInt();
         physical_address = parser.readInt();
         file_size = parser.readInt();
         mem_size = parser.readInt();
         flags = parser.readInt();
         alignment = parser.readInt();
      } else {
         // typedef struct {
         // Elf64_Word p_type;
         // Elf64_Word p_flags;
         // Elf64_Off p_offset;
         // Elf64_Addr p_vaddr;
         // Elf64_Addr p_paddr;
         // Elf64_Xword p_filesz;
         // Elf64_Xword p_memsz;
         // Elf64_Xword p_align;
         // } Elf64_Phdr;
         type = parser.readInt();
         flags = parser.readInt();
         this.offset = parser.readLong();
```

```
            virtual_address = parser.readLong();
            physical_address = parser.readLong();
            file_size = parser.readLong();
            mem_size = parser.readLong();
            alignment = parser.readLong();
         }

         switch (type) {
         case PT_INTERP:
            ptInterpreter = new MemoizedObject<String>() {
               @Override
               protected String computeValue() throws ElfException {
                  parser.seek(ElfSegment.this.offset);
                  StringBuilder buffer = new StringBuilder();
                  int b;
                  while ((b = parser.readUnsignedByte()) != 0)
                     buffer.append((char) b);
                  return buffer.toString();
               }
            };
            break;
         }
      }

      @Override
      public String toString() {
         String typeString;
         switch (type) {
         case PT_NULL:
            typeString = "PT_NULL";
            break;
         case PT_LOAD:
            typeString = "PT_LOAD";
            break;
         case PT_DYNAMIC:
            typeString = "PT_DYNAMIC";
            break;
         case PT_INTERP:
            typeString = "PT_INTERP";
            break;
         case PT_NOTE:
            typeString = "PT_NOTE";
            break;
         case PT_SHLIB:
            typeString = "PT_SHLIB";
            break;
         case PT_PHDR:
            typeString = "PT_PHDR";
            break;
         default:
            typeString = "0x" + Long.toHexString(type);
            break;
         }

         String pFlagsString = "";
         if (isReadable()) pFlagsString += (pFlagsString.isEmpty() ? "" : "|") + "read";
         if (isWriteable()) pFlagsString += (pFlagsString.isEmpty() ? "" : "|") +
   "write";
         if (isExecutable()) pFlagsString += (pFlagsString.isEmpty() ? "" : "|") +
   "execute";

         if (pFlagsString.isEmpty()) pFlagsString = "0x" + Long.toHexString(flags);
         return "ElfProgramHeader[p_type=" + typeString + ", p_filesz=" + file_size + ",
   p_memsz=" + mem_size + ", p_flags=" + pFlagsString + ", p_align="
               + alignment + ", range=[0x" + Long.toHexString(virtual_address) + "-0x" +
   Long.toHexString(virtual_address + mem_size) + "]]";
      }

      /** Only for {@link #PT_INTERP} headers. */
      public String getIntepreter() throws IOException {
         return (ptInterpreter == null) ? null : ptInterpreter.getValue();
      }

      public boolean isReadable() {
         return (flags & /* PF_R= */4) != 0;
      }

      public boolean isWriteable() {
         return (flags & /* PF_W= */2) != 0;
      }

      public boolean isExecutable() {
         return (flags & /* PF_X= */1) != 0;
      }
   }
```

ElfStringTable.java

```
package net.fornwall.jelf;

import java.io.IOException;

/**
 * String table sections hold null-terminated character sequences, commonly called
strings.
 *
 * The object file uses these strings to represent symbol and section names.
 *
 * You reference a string as an index into the string table section.
 */
final public class ElfStringTable extends ElfSection {

   /** The string table data. */
   private final byte[] data;
   public final int numStrings;

   /** Reads all the strings from [offset, length]. */
   ElfStringTable(ElfParser parser, long offset, int length, ElfSectionHeader header)
throws ElfException {
      super(parser, header);

      parser.seek(offset);
      data = new byte[length];
      int bytesRead = parser.read(data);
      if (bytesRead != length)
         throw new ElfException("Error reading string table (read " + bytesRead +
"bytes - expected to " + "read " + data.length + "bytes)");

      int stringsCount = 0;
      for (byte datum : data) if (datum == '\0') stringsCount++;
      numStrings = stringsCount;
   }

   public String get(int index) {
      int endPtr = index;
      while (data[endPtr] != '\0')
         endPtr++;
      return new String(data, index, endPtr - index);
   }
}
```

ElfSymbol.java

```
package net.fornwall.jelf;

/**
 * An entry in the {@link ElfSymbolTableSection}, which holds information needed to
locate and relocate a program's symbolic definitions and references.
 * <p>
 * In the elf.h header file the struct definitions are:
 *
 * <pre>
 * typedef struct {
 *     uint32_t      st_name;
 *     Elf32_Addr    st_value;
 *     uint32_t      st_size;
 *     unsigned char st_info;
 *     unsigned char st_other;
 *     uint16_t      st_shndx;
 * } Elf32_Sym;
 *
 * typedef struct {
 *     uint32_t      st_name;
 *     unsigned char st_info;
 *     unsigned char st_other;
 *     uint16_t      st_shndx;
 *     Elf64_Addr    st_value;
 *     uint64_t      st_size;
 * } Elf64_Sym;
 * </pre>
 */
public final class ElfSymbol {

    public enum Visibility {
        /**
         * The visibility of symbols with the STV_DEFAULT attribute is as specified
by the symbol's binding type.
         * <p>
         * That is, global and weak symbols are visible outside of their defining
component, the executable file or shared object.
         * Local symbols are hidden. Global and weak symbols can also be preempted,
that is, they may by interposed by definitions
         * of the same name in another component.
         */
        STV_DEFAULT,
        /**
         * This visibility attribute is currently reserved.
         */
        STV_INTERNAL,
        /**
         * A symbol defined in the current component is hidden if its name is not
visible to other components. Such a symbol is necessarily protected.
         * <p>
         * This attribute is used to control the external interface of a component.
An object named by such a symbol may still be referenced from another component if
its address is passed outside.
         * <p>
         * A hidden symbol contained in a relocatable object is either removed or
converted to STB_LOCAL binding by the link-editor when the relocatable object is
included in an executable file or shared object.
         */
        STV_HIDDEN,
        /**
         * A symbol defined in the current component is protected if it is visible in
other components but cannot be preempted.
         *
         * Any reference to such a symbol from within the defining component must be
resolved to the definition in that component, even if there is a definition in
another component that would interpose by the default rules. A symbol with STB_LOCAL
binding will not have STV_PROTECTED visibility.
         */
        STV_PROTECTED
    }

    /**
     * Binding specifying that local symbols are not visible outside the object file
that contains its definition.
     */
    public static final int BINDING_LOCAL = 0;
    /**
     * Binding specifying that global symbols are visible to all object files being
combined.
     */
    public static final int BINDING_GLOBAL = 1;
    /**
     * Binding specifying that the symbol resembles a global symbol, but has a lower
precedence.
     */
    public static final int BINDING_WEAK = 2;
    /**
     * Lower bound binding values reserved for processor specific semantics.
     */
    public static final int BINDING_LOPROC = 13;
    /**
     * Upper bound binding values reserved for processor specific semantics.
     */
    public static final int BINDING_HIPROC = 15;

    /**
     * Type specifying that the symbol is unspecified.
     */
    public static final byte STT_NOTYPE = 0;
    /**
     * Type specifying that the symbol is associated with an object.
     */
    public static final byte STT_OBJECT = 1;
    /**
     * Type specifying that the symbol is associated with a function or other
executable code.
     */
    public static final byte STT_FUNC = 2;
    /**
     * Type specifying that the symbol is associated with a section. Symbol table
entries of this type exist for
     * relocation and normally have the binding BINDING_LOCAL.
     */
    public static final byte STT_SECTION = 3;
    /**
     * Type defining that the symbol is associated with a file.
     */
    public static final byte STT_FILE = 4;
    /**
     * The symbol labels an uninitialized common block.
     */
    public static final byte STT_COMMON = 5;
    /**
     * The symbol specifies a Thread-Local Storage entity.
     */
    public static final byte STT_TLS = 6;

    /**
     * Lower bound for range reserved for operating system-specific semantics.
     */
    public static final byte STT_LOOS = 10;
    /**
     * Upper bound for range reserved for operating system-specific semantics.
     */
    public static final byte STT_HIOS = 12;
    /**
     * Lower bound for range reserved for processor-specific semantics.
     */
    public static final byte STT_LOPROC = 13;
    /**
     * Upper bound for range reserved for processor-specific semantics.
     */
    public static final byte STT_HIPROC = 15;

    /**
     * Index into the symbol string table that holds the character representation of
the symbols. 0 means the symbol has
     * no character name.
     */
    public final int st_name; // Elf32_Word
    /**
     * Value of the associated symbol. This may be a relative address for .so or
absolute address for other ELFs.
     */
    public final long st_value; // Elf32_Addr
    /**
     * Size of the symbol. 0 if the symbol has no size or the size is unknown.
     */
    public final long st_size; // Elf32_Word
    /**
     * Specifies the symbol type and binding attributes.
     */
    public final short st_info; // unsigned char
    /**
     * Currently holds the value of 0 and has no meaning.
     */
    public final short st_other; // unsigned char
    /**
     * Index to the associated section header. This value will need to be read as an
unsigned short if we compare it to
     * ELFSectionHeader.NDX_LORESERVE and ELFSectionHeader.NDX_HIRESERVE.
     */
    public final /* Elf32_Half */ short st_shndx;

    public final int section_type;

    /**
     * Offset from the beginning of the file to this symbol.
     */
    public final long offset;

    private final ElfFile elfHeader;
```

```java
ElfSymbol(ElfParser parser, long offset, int section_type) {
    this.elfHeader = parser.elfFile;
    parser.seek(offset);
    this.offset = offset;
    if (parser.elfFile.objectSize == ElfFile.CLASS_32) {
        st_name = parser.readInt();
        st_value = parser.readInt();
        st_size = parser.readInt();
        st_info = parser.readUnsignedByte();
        st_other = parser.readUnsignedByte();
        st_shndx = parser.readShort();
    } else {
        st_name = parser.readInt();
        st_info = parser.readUnsignedByte();
        st_other = parser.readUnsignedByte();
        st_shndx = parser.readShort();
        st_value = parser.readLong();
        st_size = parser.readLong();
    }

    this.section_type = section_type;

    switch (getType()) {
        case STT_NOTYPE:
            break;
        case STT_OBJECT:
            break;
        case STT_FUNC:
            break;
        case STT_SECTION:
            break;
        case STT_FILE:
            break;
        case STT_LOPROC:
            break;
        case STT_HIPROC:
            break;
        default:
            break;
    }
}

/**
 * Returns the binding for this symbol.
 */
public int getBinding() {
    return st_info >> 4;
}

/**
 * Returns the symbol type.
 */
public int getType() {
    return st_info & 0x0F;
}

/**
 * Returns the name of the symbol or null if the symbol has no name.
 */
public String getName() throws ElfException {
    // Check to make sure this symbol has a name.
```

```
        if (st_name == 0) return null;

        // Retrieve the name of the symbol from the correct string table.
        String symbol_name = null;
        if (section_type == ElfSectionHeader.SHT_SYMTAB) {
            symbol_name = elfHeader.getStringTable().get(st_name);
        } else if (section_type == ElfSectionHeader.SHT_DYNSYM) {
            symbol_name = elfHeader.getDynamicStringTable().get(st_name);
        }
        return symbol_name;
    }

    public Visibility getVisibility() {
        if (st_other < 0 || st_other > 3) throw new ElfException("Unsupported 
st_other=" + st_other);
        return Visibility.values()[st_other];
    }

    @Override
    public String toString() {
        String typeString;
        int typeInt = getType();
        switch (typeInt) {
            case STT_NOTYPE:
                typeString = "unspecified";
                break;
            case STT_OBJECT:
                typeString = "object";
                break;
            case STT_FUNC:
                typeString = "function";
                break;
            case STT_SECTION:
                typeString = "section";
                break;
            case STT_FILE:
                typeString = "file";
                break;
            case STT_LOPROC:
                typeString = "loproc";
                break;
            case STT_HIPROC:
                typeString = "hiproc";
                break;
            default:
                typeString = Integer.toString(typeInt);
                break;
        }

        return "ElfSymbol[name=" + getName() + ", type=" + typeString + ", size=" + 
st_size + "]";
    }
}
```

ElfSymbolTableSection.java

```java
package net.fornwall.jelf;

/**
 * An ELF section with symbol information.
 *
 * This class represents either of two section types:
 * <ul>
 *     <li>{@link ElfSectionHeader#SHT_DYNSYM}: For a minimal set of symbols adequate
for dynamic linking. Can be stripped and has no runtime cost (is non-allocable).
Normally named ".dynsym".</li>
 *     <li>{@link ElfSectionHeader#SHT_SYMTAB}: A complete symbol table typically
used for link editing. Can not be stripped (is allocable). Normally named
".symtab".</li>
 * </ul>
 */
public class ElfSymbolTableSection extends ElfSection {

    public final ElfSymbol[] symbols;

    public ElfSymbolTableSection(ElfParser parser, ElfSectionHeader header) {
        super(parser, header);

        int num_entries = (int) (header.size / header.entry_size);
        symbols = new ElfSymbol[num_entries];
        for (int i = 0; i < num_entries; i++) {
            final long symbolOffset = header.section_offset + (i *
header.entry_size);
            symbols[i] = new ElfSymbol(parser, symbolOffset, header.type);
        }
    }
}
```

MemoizedObject.java

```java
package net.fornwall.jelf;

import java.io.IOException;

/**
 * A memoized object. Override {@link #computeValue} in subclasses; call {@link
#getValue} in using code.
 */
abstract class MemoizedObject<T> {
   private boolean computed;
   private T value;

   /**
    * Should compute the value of this memoized object. This will only be called
once, upon the first call to
    * {@link #getValue}.
    */
   protected abstract T computeValue() throws ElfException;

   /** Public accessor for the memoized value. */
   public final T getValue() throws ElfException {
      if (!computed) {
         value = computeValue();
         computed = true;
      }
      return value;
   }

   @SuppressWarnings("unchecked")
   public static <T> MemoizedObject<T>[] uncheckedArray(int size) {
      return new MemoizedObject[size];
   }
}
```