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 М313D

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: х0 – 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).

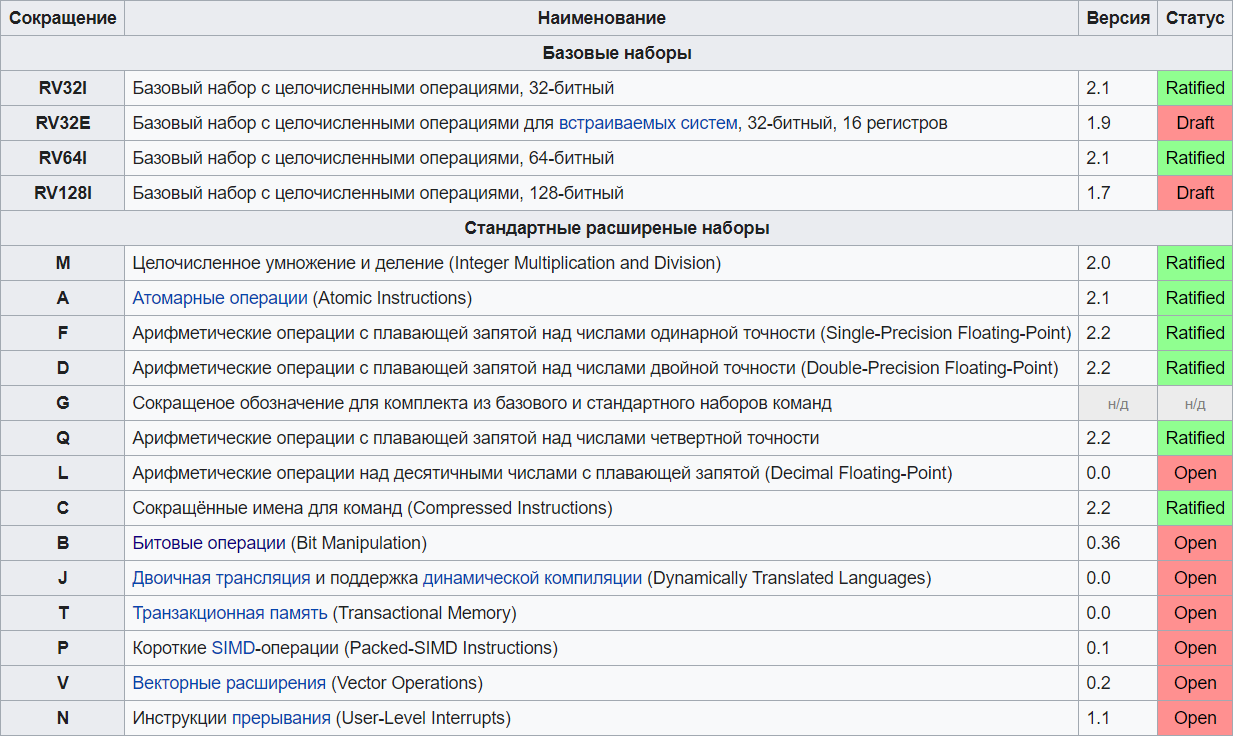


Рисунок-таблица №1

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).

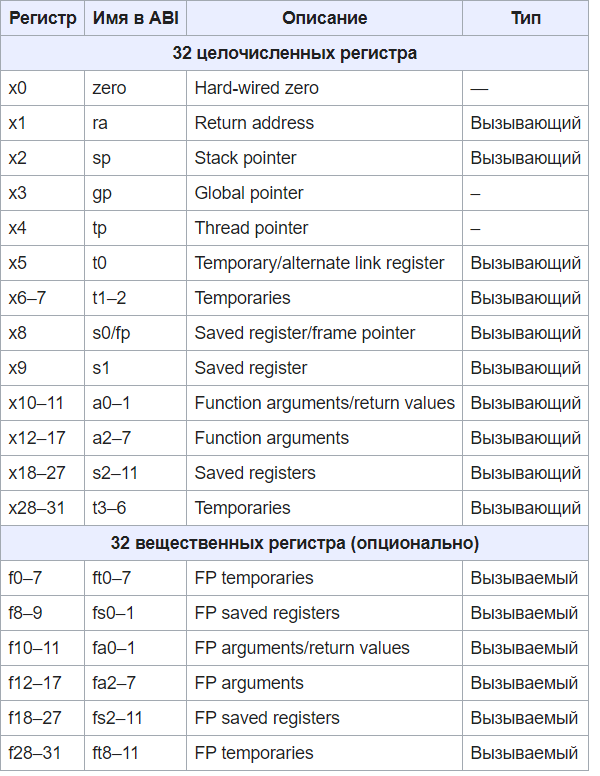


Рисунок-таблица №2

**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).

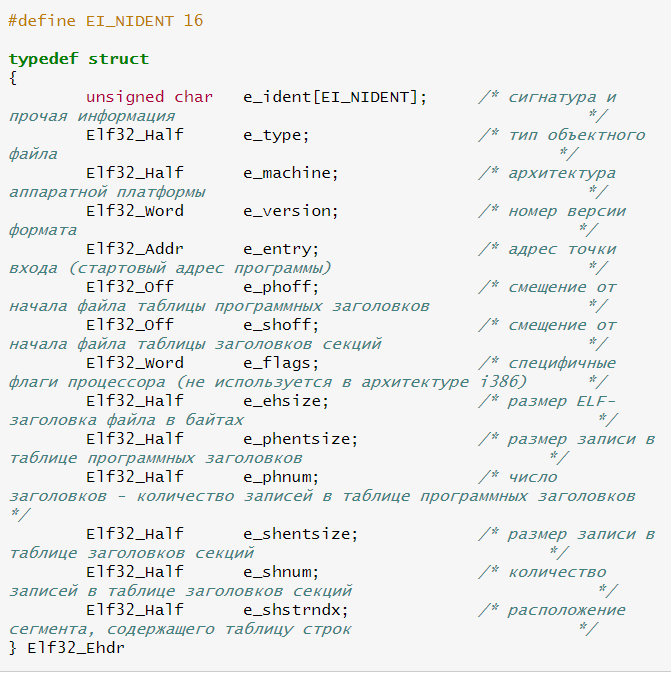


Рисунок №1

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

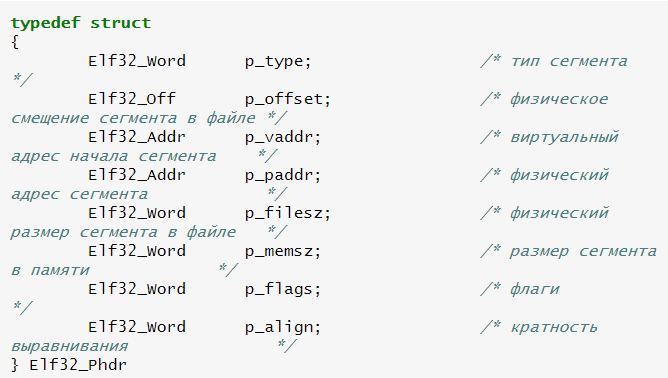


Таблица заголовков секций характеризует секции файла. Таблица секция является обязательной для компоновщика и необязательной для системного загрузчика. Компоновщик комбинирует секции с похожими атрибутами и оптимальным образом размещает их по сегментам при сборке файла (смотрите на рисунок № 3).

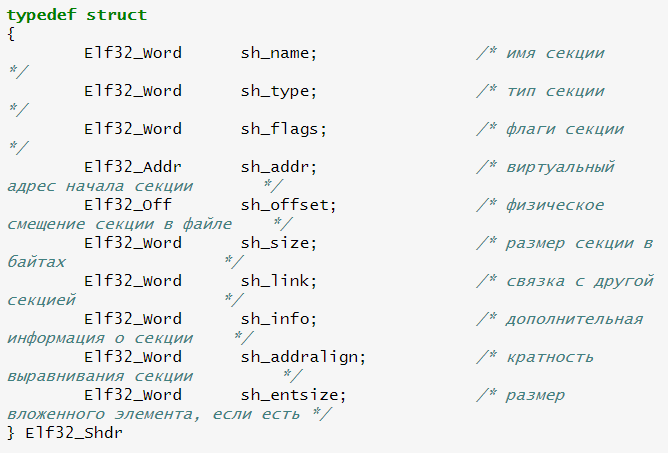


Рисунок №3

**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**

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">Dynamic 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;  
 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**

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.  
 \*/* 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) {  
 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**

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**

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**

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;  
 */\*\*  
 \* 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;  
  
 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**

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**

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];  
 }  
}