ELF pickles for GNU poke

by Jose E. Marchesi

Table of Contents

GNU poke is an interactive, extensible editor for binary data. Not limited to editing basic entities such as bits and bytes, it provides a full-fledged procedural, interactive programming language designed to describe data structures and to operate on them.

This manual explains how to use the ELF pickles distributed at `https://jemarch.net/poke-elf`.

1 Introduction

This manual documents the pickles shipped in the poke-elf package. It not only describes the data structures implemented in these pickles, but also shows examples of techniques to how to make best use of them.

1.1 Who is this manual for?

This manual assumes that the reader is familiar with both the `poke` program and the Poke programming language.

1.2 Approach used to describe Poke data structures

When describing Poke types (such as for example `Elf64_Chdr`) this manual takes the approach of first showing a simplified or stripped-out version of the type, like this:

```
type Elf64_Chdr =
  struct
  {
    Elf_Word ch_type;
    Elf_Word ch_reserved;
    offset<Elf64_Xword,B> ch_size;
    offset<Elf64_Xword,B> ch_addralign;
  };
```

Generally speaking, these stripped versions of the type do not contain comments, constraints, variables, functions nor methods. However, there are exceptions to this rule in the particular cases where we want to draw your attention to some particular aspect involving some constraint, method, etc.

Following the simplified version of the type, its fields get discussed in detail. Then, the overall data structure gets discussed, and examples on how to use it to poke at data are shown. Finally, the methods offered by the type, if any, are described in detail along with usage examples.

2 Installation

Welcome! This section should get you up and running to enjoy poking at nasty ELF data in no time.

2.1 Build Requirements

These are the build requirements if you are building from a distribution source tarball:

– A recent enough version of GNU poke is necessary in order to run the test suite of this package. This is checked at configure time. If no suitable poke is found, the test suite is not run.

2.2 Fetching and unpacking poke-elf

The first step to install `poke-elf` is to fetch a copy of it. Like all GNU poke pickles, `poke-elf` releases are distributed as source tarballs:

```
$ wget https://ftp.gnu.org/gnu/poke/poke-elf-version.tar.gz
```

Where *version* is the version you want to install. Next step is to untar the tarball. It will expand to a `poke-elf-version` directory:

```
$ tar xvf poke-elf-version.tar.gz
$ cd poke-elf-version/
```

2.3 Configuring the sources

It is time now to configure the sources. You do that by invoking the `configure` script that is in the root directory of the distribution.

```
$ ./configure
```

By default the `configure` script will configure the source in order to be installed under `/usr/local`, which is a system location. If you want to install the pickles in some other location, you can pass a `--prefix` command line option to the script. For example:

```
$ ./configure --prefix=$HOME/.poke.d
```

Now that the sources are configured, it is time to build them and check the distribution.

2.4 Building and checking

```
$ make
$ make check
```

There should be no errors. If any of the tests fail, please re-run `make check` but this time enabling verbose output:

```
$ make check VERBOSE=1
```

And file a bug report at `https://sourceware.org/bugilla` including both the contents of your `config.log` file and the output you get on the terminal when you run `make check`. Please file the bug report for product “poke” and component “elf-pickle”.

Note that the testsuite will only be executed if a recent enough `poke` was found during configure.

2.5 Installing

The last step is to install the pickles in your system:

```
$ make install
```

Note that the installed poke will find the installed pickles only if these are installed under the same prefix than poke. If you install the pickles in some other location (like under `~/.poke.d` for example, you will have to set the environment variable `POKE_LOAD_PATH`. Just put something like this in your `.bashrc` or similar file:

```
export POKE_LOAD_PATH=$HOME/.poke.d
```

And that's it! Now run poke, load the pickles and enjoy!

```
$ poke /bin/ls
(poke) load elf
(poke) var elf = Elf64_File @ 0#B
...
```

3 Pickles Overview

This chapter provides an high-level overview of all the pickles distributed by this package. These are all developed in subsequent chapters.

`elf.pk` This is the main ELF pickle, and the one that is intended to be loaded by the user. It does little more than loading the rest of the `elf-*` pickles, but it does it in the right order!

`elf-build.pk`
This pickle contains information generating during building the `poke-elf` package. In particular, the version.

`elf-config.pk`
This pickle implements the *ELF configuration registry*, which is used by the ELF pickles in order to maintain a database of the very varied set of different configuration parameters supported by the ELF specification: machine types, section types, segment types, file flags, *etc.* See Chapter 4 [ELF Configurations], page 6.

`elf-common.pk`
This pickle contains definitions which are common to both 32-bit and 64-bit ELF. It also registers configuration parameters that are common to all machine types.

`elf-os-`*`os`*`.pk`
These pickles contain definitions and configuration parameters for the different operating systems supported in the ELF specification. For example, `elf-os-gnu.pk` covers the GNU extensions documented in the GNU gabi extensions[1].

`elf-mach-aarch64.pk`
`elf-mach-arm.pk`
`elf-mach-bpf.pk`
`elf-mach-mips.pk`
`elf-mach-riscv.pk`
`elf-mach-sparc.pk`
`elf-mach-x86-64.pk`
These pickles contain definitions and configuration parameters for the different *machine types* supported in the ELF specification.

`elf-32.pk`
This pickle contains definitions for 32-bit ELF files. Among these is the definition of the `Elf32_File` type, which corresponds to an entire ELF-32 file.

`elf-64.pk`
This pickle contains definitions for 64-bit ELF files. Among these is the definition of the `Elf64_File` type, which corresponds to an entire ELF-64 file.

[1] `https://sourceware.org/gnu-gabi`

4 ELF Configurations

4.1 ELF Configuration Parameters

The ELF object format specification, unlike most (all?) its predecessors, was designed with the goal of being extremely flexible and extensible, in order to cover the needs of any conceivable hardware architecture and operating system.

In order to achieve this goal, many of the entities that appear in ELF files, such as sections, symbols, or segments, are pretty generic and configurable. For example, consider the following (simplified) definition of an ELF64 relocation:

```
type Elf64_RelInfo =
  struct
  {
    uint<32> r_sym;
    uint<32> r_type;
  };
```

Where `r_type` contains a code identifying the type of the relocation. The ELF specification itself doesn't say what values may go in `r_type`; it is the different supplements for particular architectures (or *machines* in ELF parlance) that list the relocation types used in their machines.

Relocation types, understood as the set of valid codes to be set in a `r_type` field, is just one example of what in poke-elf we call an ELF *configuration parameter*.

There are many other configuration parameters: section flags, section types, symbol types, and a large *etc*. They are often dependent of particular machines and OSes, and there can be *many* of them: there are often literally hundreds of different relocation types defined by some particular architecture.

4.2 The ELF Configuration Registry

As the different ELF pickles get loaded, they populate a registry of configuration parameters. This registry is a value of the struct type `Elf_Config` that is defined in `elf-config.pk`, and is stored in the global variable `elf_config`.

There are two kinds of configuration parameters: *enumerations* and *masks*. The registry contains several collections of them:

- One set of common enumerations.
- One set of common masks.
- One set of enumerations per machine type.
- One set of masks per machine type.

4.3 Enumeration configuration parameters

Enumeration configuration parameters, or *enums* for short, are sets of numbers or codes. Each entry in an enum represents an alternative value for some parameter. New enum entries are constructed using the `Elf_Config_UInt` struct type:

```
type Elf_Config_UInt =
  struct
  {
    uint<32> value;
    string name;
    string doc;
  };
```

Where `name` is a short and descriptive name for the parameter value and `doc` is an English statement describing the meaning of this particular *value*.

For example, this is how the definition of a X86_64 relocation type looks like:

```
Elf_Config_UInt { value = ELF_R_X86_64_PC32, name = "pc32",
                  doc = "PC relative 32 bit signed." }
```

Adding new enum configuration parameters to the registry is done by using the `add_enum` method of `Elf_Config`:

```
method add_enum = (int<32> machine = -1,
                   string class = "",
                   Elf_Config_UInt[] entries = Elf_Config_UInt[]()) void:
```

Where `machine` is either -1 or an ELF machine code (likely one of the `ELF_EM_*` values defined in `elf-common.pk`). If the former, the new parameter is added to the set of common enums. Otherwise it is added to the set of enums defined for the specified machine type. Finally, `entries` is an array of the different values this parameter may adopt.

The `class` argument is a string that gives a name to the new configuration parameter. These have names like `reloc-types` or `file-classes`. Our ELF pickles use a definite set of names, documented below, but nothing prevents you to use your own.

This is how we would add a couple of common relocation types to the register (note the actual ELF specification has none of these, they are all machine-specific):

```
elf_config.add_enum
  :class "reloc-types"
  :entries [Elf_Config_UInt { value = 0, name = "null reloc" },
            Elf_Config_UInt { value = 1, name = "PC-relative 16-bit displacement." }];
```

And this is how we would add relocation types for the X86_64 architecture:

```
elf_config.add_enum
  :class "relocation_types"
  :entries [Elf_Config_UInt { value = ELF_R_X86_64_PC32, name = "pc32",
                              doc = "PC relative 32 bit signed." },
            Elf_Config_UInt { value = ELF_R_X86_64_GOT32, name = "got32",
                              doc = "32 bit GOT entry." },
            ...];
```

4.4 Mask configuration parameters

Mask configuration parameters are sets of bit-masks. Each entry is an unsigned number determining some valid configuration of bits for the value of some parameter. New mask entries are constructed using the `Elf_Config_Mask` type:

```
type Elf_Config_Mask =
  struct
  {
    uint<64> value;
    string name;
    string doc;
  };
```

Where `name` is a short an descriptive name summarizing the quality of the bit of bits set in `value`, and `doc` is an English statement describing the meaning of these particular bits.

For example, this is how the definition of an ARM section flag looks like:

```
Elf_Config_Mask { value = ELF_SHF_ARM_PURECODE, name = "purecode",
                  doc = "Section contains only code and no data." }
```

Adding new mask configuration parameters to the registry is done by using the `add_mask` method of `Elf_Config`:

```
method add_mask = (int<32> machine = -1,
                   string class = "",
                   Elf_Config_Mask[] entries = Elf_Config_Mask[]()) void:
```

Where `machine` is either -1 or an ELF machine code. If the former, the new mask is added to the set of common masks. Otherwise it is added to the set of masks defined for the specified machine type. Finally, `entries` is an array of the different sub-masks this parameter may adopt.

As with enums, the `class` argument is a string that gives a name to the new configuration parameter. Masks have names like `"section-flags"` or `"segment-flags"`.

This is how we would register a couple of common section flags:

```
elf_config.add_mask
  :class "section-flags"
  :entries [Elf_Config_Mask { value = ELF_SHF_WRITE, name = "write" },
            Elf_Config_Mask { value = ELF_SHF_ALLOC, name = "alloc" }];
```

And this is how we would register section flags for the ARM architecture:

```
elf_config.add_mask
  :machine ELF_EM_ARM
  :class "section-flags"
  :entries [Elf_Config_Mask { value = ELF_SHF_ARM_ENTRYSECT, name = "entrysect",
                              doc = "Section contains an entry point." },
            Elf_Config_Mask { value = ELF_SHF_ARM_PURECODE, name = "purecode",
                              doc = "Section contains only code and no data." },
            ...];
```

4.5 Configuration parameters used by this pickle

As we have mentioned, it is possible to register new configuration parameters in the registry, with arbitrary names. This is certainly useful to the happy poker that is working on some weird ELF extension, or simply playing around.

However, the set of `elf-*pk` pickles are designed to work with a closed set of configuration parameters. Having extra parameters in the registry is perfectly ok, but if you mess with the parameters below, you are gonna have to face the consequences :)

Note however that adding support for a new machine type or a new operating system shouldn't require extending the set of configuration parameters: just to add new values to them.

The enum configuration parameters used by this pickle are:

`elf-machines`
: Valid values in `e_machine` fields.

`file-osabis`
: Valid values in `ei_osabi` fields.

`file-encodings`
: Valid values in ei_data fields.

`file-classes`
: Valid values in ei_class fields.

`file-types`
: Valid values in e_type fields.

`section-types`
: Valid values in sh_type fields.

`section-indices`
: Indices in the file section header table with special meanings.

`section-other`
: Valid values in sh_other fields.

`segment-types`
: Valid values in p_type fields.

`reloc-types`
: Valid values in r_type fields.

`dynamic-tag-types`
: Valid values in d_tag fields.

`symbol-types`
: Valid values in st_type fields.

`symbol-bindings`
: Valid values in st_bind fields.

`symbol-visibilities`
: Valid values in st_visibility fields.

`note-tags`
: Valid tags for notes stored in notes sections.

`gnu-properties`
: Valid values for `pr_type` fields in GNU properties.

The mask configuration parameters used by this pickle are:

`file-flags`
: Valid bits in e_flags fields.

`section-flags`
: Valid bits in sh_flags fields.

`segment-flags`
: Valid bits in p_flags fields.

The architecture-specific enum configuration parameters used by this pickle are:

`mips-abis`
: Valid values in the `ELF_EF_MIPS_ABI` bits of `e_flags` in MIPS machines.

`mips-machines`
: Valid values in the `ELF_EF_MIPS_MACH` bits of `e_flags` in MIPS machines.

`mips-architectures`
: Valid values in the `ELF_EF_MIPS_ARCH` bits of `eflags` in MIPS machines.

The architecture-specific mask configuration parameters used by this pickle are:

`mips-l-flags`
: Valid bits in `l_flags` fields.

4.6 Getting printed representations of configuration parameters

The `format_enum` and `format_mask` methods of `Elf_Config` return the user-friendly printed representation of the given alternative value or bitmap. They have the following prototypes:

```
method format_enum = (string class, uint<16> machine,
                      uint<32> value) string:
method format_mask = (string class, uint<16> machine,
                      uint<64> value) string:
```

The printed representation of an enum is simply the `name` that was provided when registering it. For example:

```
(poke) elf_config.format_enum ("reloc-types", ELF_EM_X86_64, 2)
"pc32"
```

The printed representation of a mask is a sequence of the names given to the different bitmaps at registration time, separated by comma (`,`) characters. For example:

```
(poke) elf_config.format_mask ("section-flags", ELF_EM_X86_64, 0xf00)
"os-nonconforming,group,tls,compressed"
```

4.7 Checking valid configuration parameters

The `check_enum` and `check_mask` methods of `Elf_Config` check whether the given values are valid for some particular configuration parameter. They have the following prototypes:

```
method check_enum = (string class, uint<16> machine,
                     uint<32> value) int<32>:
method check_mask = (string class, uint<16> machine,
                     uint<64> value) int<32>:
```

Where `machine` specifies the machine type and `class` the name of the configuration parameter. They determe wether `value` is a valid `class`.

For example, this is how we would check whether 57 identifies a valid relocation type in RISCV:

```
(poke) elf_config.check_enum ("reloc-types", ELF_EM_RISCV, 57)
0
```

Turns out it doesn't! :D

4.8 Using configuration parameters in types

The formatting and checking methods described above are mainly used in the ELF pickles in order to implement pretty-printers and data integrity constraints in the several ELF structures holding such values.

For example:

```
type Elf64_Shdr =
  struct
  {
    Elf_Word sh_type : elf_config.check_enum ("section-types", elf_mach, sh_type);
    Elf64_Xword sh_flags : elf_config.check_mask ("section-flags", elf_mach, sh_flags)

    [...]

    method _print_sh_type = void:
    {
      printf "#<%s>", elf_config.format_enum ("section-types", elf_mach, sh_type);
```

```
    }

    method _print_sh_flags = void:
    {
      printf "#<%s>", elf_config.format_mask ("section-flags", elf_mach, sh_flags);
    }
  };
```

However, they are also very useful to the user while poking at existing data ("if these bytes were to be interpreted as ELF section flags in some given arch, which ones they would be?"), composing new data and also when generating reports and statistics.

4.9 Debugging the registry

If you want to get a trace of the configuration parameters as they are being added to the registry, simply set the `elf_config_debug` variable to a non zero value and reload the ELF pickles:

```
(poke) elf_config_debug = 1
(poke) load elf
```

5 ELF Basic Types

The encoding of the simple fields in the ELF data structures is abstracted in the following Poke types.

Types used in both 32-bit and 64-bit ELF:

`type Elf_Half = uint<16>`
: An ELF *unsigned medium integer*.

`type Elf_Word = uint<32>`
: An ELF *unsigned integer*.

`type Elf_Sword = int<32>`
: An ELF *signed integer*.

Types used in 32-bit ELF only:

`type Elf32_Addr = offset<uint<32>,B>`
: An ELF *unsigned program address*.

`type Elf32_Off = offset<uint<32>,B>`
: An ELF *unsigned file offset*.

Types used in 64-bit ELF only:

`type Elf64_Xword = uint<64>`
: An ELF *unsigned long integer*.

`type Elf64_Sxword = int<64>`
: An ELF *signed long integer*.

`type Elf64_Addr = offset<uint<64>,B>`
: An ELF *unsigned program address*.

`type Elf64_Off = offset,uint<64>,B>`
: And ELF *unsigned file offset*.

6 ELF File

The Poke types provided to denote ELF64 and ELF32 files are `Elf64_File` and `Elf32_File` respectively.

6.1 Overview

```
type Elf32_File =
  struct
  {
    Elf32_Ehdr ehdr;

    if (ehdr.e_shnum > 0)
      Elf32_Shdr[ehdr.e_shnum] shdr  ehdr.e_shoff;

    if (ehdr.e_phnum > 0)
      Elf32_Phdr[ehdr.e_phnum] phdr  ehdr.e_phoff;
  };

type Elf64_File =
  struct
  {
    Elf64_Ehdr ehdr;

    if (ehdr.e_shnum > 0)
      Elf64_Shdr[ehdr.e_shnum] shdr  ehdr.e_shoff;

    if (ehdr.e_phnum > 0)
      Elf64_Phdr[ehdr.e_phnum] phdr  ehdr.e_phoff;
  };
```

6.2 Fields

`ehdr` Is the header of the ELF file, of type `Elf64_File`. This always exists and is always located at the beginning of the ELF file.

`shdr` Is the optional section header table of the ELF file. This is an optional field that is an array of `Elf64_Shdr` (or `Elf32_Shdr`) values, describing the ELF sections present in the file.

This table, if it exists, can be located anywhere in the ELF file. The ELF header determines the size and location of the table.

`phdr` Is the optional program section header table of the ELF file. This is an optional field that is an array of `Elf64_Phdr` (or `Elf32_Phdr`) describing ELF segments present in the file.

This table, if it exists, can be located anywhere in the ELF file. The ELF header determines the size and location of the table.

6.3 Methods

6.3.1 Methods related to sections

`File_Elf64.section_name_p = (string` *name*`) int<32>`
`File_Elf32.section_name_p = (string` *name*`) int<32>`

Given a section *name*, return whether a section with that name exists in the ELF file.

`File_Elf64.get_sections_by_name = (string` *name*`) Elf64_Shdr[]`
`File_Elf32.get_sections_by_name = (string` *name*`) Elf32_Shdr[]`

Given the *name* of a section, return an array of section headers in the ELF file having that name. The returned array may of course be empty.

For example, this is how you can get an array of all the sections in the file with name `.text`:

```
(poke) elf.get_sections_by_name (".text")
[Elf64_Shdr {
  sh_name=141U#B,
  sh_type=#<progbits>,
  sh_flags=#<alloc,execinstr>,
  sh_addr=18144UL#B,
  sh_offset=18144UL#B,
  sh_size=80574UL#B,
  sh_link=0U,
  sh_info=0U,
  sh_addralign=16UL,
  sh_entsize=0UL#B
}]
```

`File_Elf64.get_sections_by_type = (Elf_Word` *stype*`) Elf64_Shdr[]`
`File_Elf32.get_sections_by_type = (Elf_Word` *stype*`) Elf32_Shdr[]`

Given a section type (one of the `ELF_SHT_*` value) return an array of section headers in the ELF file with that type. The returned array may be empty.

6.3.2 Methods related to string tables

`Elf64_File.get_section_name = (offset<Elf_Word,B>` *offset*`) string`
`Elf32_File.get_section_name = (offset<Elf_Word,B>` *offset*`) string`

Given an offset into the ELF file's section string table, return the string starting at that *offset*. This uses one particular string table that is linked from the ELF header via the `e_shstrndx` field.

For example, this is how we would print the name of the second section in an ELF file[1]:

```
(poke) elf.get_section_name (elf.shdr[1].sh_name)
".interp"
```

`Elf64_File.get_symbol_name = (Elf64_Shdr` *symtab*`, offset<Elf_Word,B>` *offset*`) string`
`Elf64_File.get_symbol_name = (Elf32_Shdr` *symtab*`, offset<Elf_Word,B>` *offset*`) string`

Given the section header of a section that contains a symbol table *symtab*, and an *offset*, return the corresponding string stored at the symbol table associated string table.

[1] The first section in an ELF file is the "null" section and has an empty name.

`Elf64_File.get_string = (offset<Elf_Word,B> offset) string`
`Elf32_File.get_string = (offset<Elf_Word,B> offset) string`

Given an *offset*, return the string stored at that offset in the "default" string table of the ELF file.

The default string table is contained in a section named `.strtab`. If such a section doesn't exist, or if it exists but it doesn't contain a string table, then this function raises `E_inval`.

6.3.3 Methods related to section groups

`Elf64_File.get_group_signature = (Elf64_Shdr section) string`
`Elf32_File.get_group_signature = (Elf32_Shdr section) string`

Return the signature corresponding to a given group *section*, characterized by its entry in the section header table. If the given section header doesn't correspond to a group section then raise `E_inval`.

`Elf64_File.get_group_signatures = string[]`
`Elf32_File.get_group_signatures = string[]`

Return an array of strings with the signatures of the section groups present in this ELF file.

`Elf64_File.get_section_group = (string name) Elf64_Shdr[]`
`Elf32_File.get_section_group = (string name) Elf32_Shdr[]`

Given the *name* of a section group, return an array with the section headers corresponding to all the sections in that group. If the given name doesn't identify a section group in the ELF file then return an empty array.

6.3.4 Methods related to loaded contents

`Elf64_File.get_load_base = Elf64_Addr`

Determine the base where the contents of the ELF file are loaded, understood as the lower virtual address where segments get loaded. If there are no loadable segments in the ELF file then this method raises `E_inval`.

`Elf64_File.vaddr_to_sec = (Elf64_Addr vaddr) Elf64_Addr`

Given a virtual address, return the index in the section header table of the section whose loaded contents cover the given address. If no such section is found this method returns -1.

Consider for example a relocation which points to some content that is stored in some section in a loadable ELF file. The corresponding `r_offset` field in the relocation will not contain a file offset, but a loaded address. This method can be then used to determine the section the relocation is applied to.

`Elf64_File.vaddr_to_file_offset = (Elf64_Addr vaddr) Elf64_Addr`

If some of the contents of the file sections are to be loaded in *vaddr*, this method returns the file offset to these contents.

6.4 Usage

Poking at an ELF file usually starts by opening some IO space and mapping a `Elf64_File` (or `Elf32_File`):

```
(poke) .file /bin/ls
(poke) var elf = Elf64_File @ 0#B
```

Once mapped, we can access any of the above fields. For example, let's see how many sections and segments this file has:

```
(poke) elf.shdr'length
```

```
30UL
(poke) elf.phdr'length
11UL
```

In case the file didn't have a program header table, which always happens with object files, we would have got an exception if we tried to access the absent field `phdr`:

```
$ echo '' | gcc -c -xc -o foo.o -
$ poke foo.o
(poke) load elf
(poke) (Elf64_File @ 0#B).phdr
unhandled invalid element exception
```

6.4.1 Working with sections

Unlike in older object formats (like a.out for example) the sections present in ELF files are not fixed nor they have fixed pre-defined names: there can be any number of them (including none) and they can have any arbitrary name. Also, more than one section in the file can have the same name.

So when it comes to ELF files, the process to determine whether one or more section with a given name exists in the file is a bit laborious: one has to traverse the section header table, fetch the section names from whatever appropriate string table, etc.

The following methods, that you can use in your own pickles, scripts, or at the prompt, are handy to look at particular sections in the file.

6.4.2 Working with string tables

The names of several entities in ELF files are stored in different string table, which are themselves stored in different sections. There are different rules establishing where exactly the name of certain entities (sections, symbols, ...) are to be found.

These rules are not trivial and require traversing several data structures. Therefore the `Elf64_File` (and `File32_File`) type provides several methods in order to easily determine the name of these entities.

6.4.3 Working with section groups

ELF supports grouping several sections in a *section group*. This is useful when several sections have to go together, because they rely on each other somehow.

A section of type `SHT_GROUP` defines a section group. Groups are univocally identified by a *group signature*, which is the name associated with a symbol that is stored in a particular symbol table, linked from the section header of the group defining section.

Again, it is not exactly trivial to determine, for example, which of the sections in the ELF file pertain to which group. Therefore the pickle provides the methods below:

7 ELF Header

The ELF headers are always to be found at the beginning of an ELF file. However, it is also common to find ELF data embedded in other container formats (such as an ELF section!) and sometimes ELF headers are used to describe non-conformance ELF contents. Therefore poking at headers directly is not that uncommon.

The Poke types provided to denote ELF headers are `Elf64_Ehdr` and `Elf32_Ehdr`, for 64-bit and 32-bit ELF files respectively.

7.1 Overview

```
type Elf32_Ehdr =
  struct
  {
    Elf_Ident e_ident;
    Elf_Half e_type;
    Elf_Half e_machine;
    Elf_Word e_version = ELF_EV_CURRENT;
    Elf32_Addr e_entry;
    Elf32_Off e_phoff;
    Elf32_Off e_shoff;
    Elf_Word e_flags;
    offset<Elf_Half,B> e_ehsize;
    offset<Elf_Half,B> e_phentsize;
    Elf_Half e_phnum;
    offset<Elf_Half,B> e_shentsize;
    Elf_Half e_shnum;
    Elf_Half e_shstrndx;
  };
type Elf64_Ehdr =
  struct
  {
    Elf_Ident e_ident;
    Elf_Half e_type;
    Elf_Half e_machine;
    Elf_Word e_version = ELF_EV_CURRENT;
    Elf64_Addr e_entry;
    Elf64_Off e_phoff;
    Elf64_Off e_shoff;
    Elf_Word e_flags;
    offset<Elf_Half,B> e_ehsize;
    offset<Elf_Half,B> e_phentsize;
    Elf_Half e_phnum;
    offset<Elf_Half,B> e_shentsize;
    Elf_Half e_shnum;
    Elf_Half e_shstrndx;
  };
```

7.2 Fields

`e_ident` Is a field that describes the encoding of the contents that follow in the ELF file. The data in this field is encoded in a clever way that only requires to read the

information byte by byte. This is necessary, because part of the information stored in `e_ident` is precisely the encoding used by the data in the ELF file:

```
type Elf_Ident =
  struct
  {
    byte[4] ei_mag == [0x7fUB, 'E', 'L', 'F'];
    byte ei_class;
    byte ei_data;
    byte ei_version;
    byte ei_osabi;
    byte ei_abiversion;
    byte[7] ei_pad;
  };
```

Where:

`ei_mag` Is the magic number identifying the ELF file. It is always 0x7F.

`ei_class` Determines the class of the ELF file. This can be one of `ELF_CLASS_NONE`, `ELF_CLASS_32` or `ELF_CLASS_64` denoting and "invalid class", a 32-bit ELF file and a 64-bit ELF file respectively.

I personally have never come across an ELF file with `ELF_CLASS_NONE`. But if such class is found, it shall be considered as a data integrity error. That is the approach implemented in this pickle.

`ei_data` Determines the encoding of the data in the file. This can be one of `ELF_DATA_NONE`, `ELF_DATA_2LSB` or `ELF_DATA_2MSB`, denoting no encoding, 2's complement and little endian, and 2's complement and big endian.

Note that at this point the only supported encoding for signed numbers in ELF files is 2's complement.

This pickle considers an ELF file with encoding `ELF_DATA_NONE` as a data integrity error.

`ei_version` Is the ELF header version number. This must be `ELF_EV_CURRENT`.

`ei_osabi` Identifies the ABI or operating system (these concepts are mixed in ELF) used by the ELF file. This must be one of the `ELF_OSABI_*` values defined in `elf-common.pk`.

The ELF specification recommends this field to be `ELF_OSABI_NONE`, which actually identifies the "UNIX System V ABI".

`ei_abiversion` Identifies the version of the ABI to which the ELF file is targeted. The ELF spec points out that the purpose of this field is to distinguish among incompatible versions of an ABI, and that its interpretation ultimately depends on the value of `ei_osabi`.

`ei_pad` Are unused bytes. These bytes may be used for some particular purpose in future versions of the ELF specification, and currently they must be set to zero.

`e_type` Identifies the kind of ELF file: whether it is an object file, an executable, a dynamic object or a core dump.

This field is checked against the `file-types` configuration parameter, and pretty-printed accordingly.

`e_machine`
: Identifies the machine type on which the elf file is supposed to run.

  When poke maps or constructs a `Elf64_Ehdr` (or `Elf32_Edhr`) struct, it sets the global ELF machine to the value of this field.

  This field is checked against the `machine-types` configuration parameter, and pretty-printed accordingly.

`e_version`
: Identifies the ELF version the ELF file conforms to. It must hold `ELF_EV_CURRENT`.

`e_entry`
: Is the virtual memory address of the entry point of a process executing the program in this ELF file. This can be `0#B`.

`e_phoff`
: Is the file offset of the program header table. If the ELF file doesn't contain any segment, then the table is empty and this field contains `0#B`.

`e_shoff`
: Is the file offset of the section header table. If the ELF file doesn't contain any section, then the table is empty and this field contains `0#B`.

`e_flags`
: Is a bitmap of file flags. This field contains ORed `ELF_EF_*` values.

  This field is checked against the `filed-flags` configuration parameter, and pretty-printed accordingly.

`e_ehsize`
: Is the size in bytes of the ELF header.

`e_phentsize`
: Is the size in bytes of one entry in the program header table.

`e_phnum`
: Is the number of entries in the program header table.

`e_shentsize`
: Is the size in bytes of one entry in the section header table.

`e_shnum`
: Is the number of entries in the section header table.

`e_shstrndx`
: Is the index in the section header table of the entry associated with the string table that contains the names of the sections stored in the file.

  If the ELF file doesn't contain a section name string table (which is uncommon but certainly possible) then this field contains `ELF_SHN_UNDEF`.

7.3 Usage

XXX

8 ELF Section Headers

Sections can be stored anywhere in an ELF file. They can also be of any size, of any type, have any name (or no name) and their contents are free. The ELF file therefore contains a table, called the *section header table*, whose entries describe each section. This table is sized and linked from the ELF header via the `e_shoff` field. As we have seen, the section header table is available in the `shdr` field of `Elf32_File` and `Elf64_File`.

The Poke types denoting entries in the section header table are `Elf32_Shdr` and `Elf64_Shdr` for ELF32 and ELF64 respectively.

8.1 Overview

```
type Elf32_Shdr =
  struct
  {
    offset<Elf_Word,B> sh_name;
    Elf_Word sh_type;
    Elf_Word sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off sh_offset;
    offset<Elf_Word,B> sh_size;
    Elf_Word sh_link;
    Elf_Word sh_info;
    Elf_Word sh_addralign;
    offset<Elf_Word,B> sh_entsize;
  };

type Elf64_Shdr =
  struct
  {
    offset<Elf_Word,B> sh_name;
    Elf_Word sh_type;
    Elf64_Xword sh_flags;
    Elf64_Addr sh_addr;
    Elf64_Off sh_offset;
    offset<Elf64_Xword,B> sh_size;
    Elf_Word sh_link;
    Elf_Word sh_info;
    Elf64_Xword sh_addralign;
    offset<Elf64_Xword,B> sh_entsize;
  };
```

8.2 Fields

`sh_name` Is the offset to the name of this section in the file's section string table. Two or more sections can share the same name.

`sh_type` Is a code identifying the type of the section. This is one of the `ELF_SHT_*` values.

The type of a section determines what kind of contents (if any) a section has: relocations, a symbol table, a string table, executable compiled code, etc. These are the types defined in the base spec:

`ELF_SHT_NULL`
: This marks "unused" entry in the section header table. The first entry in the table seems to always be an unused entry. Unused entries have empty names.

`ELF_SHT_PROGBITS`
: Section is what the spec calls "program specific (private) data." In practice, this basically means executable code. The prototypical progbits section is `.text`.

`ELF_SHT_SYMTAB`
: Section contains a symbol table. Each symbol table is an array of `ELF64_Sym` (`Elf32_Sym` in ELF32) values spanning for `sh_size` bytes. See Chapter 10 [ELF Symbols], page 25.

`ELF_SHT_STRTAB`
: Section contains a string table. Each string table is an array of NULL terminated strings spanning for `sh_size` bytes.

`ELF_SHT_RELA`
`ELF_SHT_REL`
: Section contains ELF relocations, with or without explicit addend. Each section contains an array of `Elf64_Rela` or `Elf64_Rel` (`Elf32_Rela` or `Elf32_Rel` in ELF32) values spanning for `sh_size` bytes. See Chapter 12 [ELF Relocations], page 29.

`ELF_SHT_HASH`
: Section contains a symbol hash table.

`ELF_SHT_DYNAMIC`
: Section contains dynamic linking information in the form of a sequence of *dynamic tags*. This is an array of `Elf64_Dyn` (`Elf32_Dyn` in ELF32) values spanning for `sh_size` bytes. See Chapter 13 [ELF Dynamic Info], page 31.

`ELF_SHT_NOTE`
: Section contains *notes*. These are flexible annotations that are usually used in order to reflect certain "auxiliary" attributes of the ELF file. For example, the name and full version of the compiler that generated it. The format in which the notes are encoded is well defined, and supported by the elf pickles. See Chapter 11 [ELF Notes], page 28.

`ELF_SHT_SHLIB`
: This value for `sh_type` is reserved by the ELF specification and has undefined semantics.

`ELF_SHT_DYNSYM`
`ELF_SHT_NOBITS`
: The section contents occupy no bits in the file.

`ELF_SHT_INIT_ARRAY`
`ELF_SHT_FINI_ARRAY`
`ELF_SHT_PREINIT_ARRAY`
: Section contains an array of pointers to initialization/finalization/pre-initialization functions, which are parameter-less procedures that do not return any value. This is an array of `offset<uint<64>,B>` (`offset<uint<32>,B>` in ELF32) values spanning for `sh_size` bytes.

`ELF_SHT_GROUP`
: Section contains the definition of an ELF section group. See Chapter 6 [ELF File], page 13.

`ELF_SHT_SYMTAB_SHNDX`
: Section contains indices for `SHN_XINDEX` entries.

The ELF supplements for architectures/machines and operating systems introduce their own additional section types. See Chapter 14 [ELF Machines], page 32.

This field is checked against the `section-types` configuration parameter, and pretty-printed accordingly.

`sh_flags` Is a bitmap where each enabled bit flags some particular property of the section. This is one of the `ELF_SHF_*` values. These are the flags defined in the base spec:

`ELF_SHF_WRITE`
: The section contains data that should be writable during process execution.

`ELF_SHF_ALLOC`
: The section contents are actually loaded into memory during process execution.

`ELF_SHF_EXECINSTR`
: The section contains executable machine instructions.

`ELF_SHF_MERGE`
: The section contents can be merged to eliminate duplication. The ELF spec provides an algorithm (to be implemented by link editors) that explains how to merge sections flagged with this flag. The algorithm covers two cases: merge-able sections containing elements of fixed size, and string tables.

`ELF_SHF_STRINGS`
: The section contains a string table.

`ELF_SHF_INFO_LINK`
: The `sh_info` field of this section header contains a section header table index.

`ELF_SHF_LINK_ORDER`
: This section is to be ordered in a particular way by link editors. The order to use is specified by a link to other section header table via `sh_info`. See the ELF spec for details.

`ELF_SHF_OS_NONCONFORMING`
: This section requires special OS support to be linked.

`ELF_SHF_OS_TLS`
: This section holds *thread-local storage*.

`ELF_SHF_COMPRESSED`
: This section contents are compressed. Sections flagged as compressed cannot have the flag `ELF_SHF_ALLOC` set. Also, sections of type `ELF_SHT_NOBITS` cannot be compressed.

The ELF supplements for architectures/machines and operating systems introduce their own additional section types. See Chapter 14 [ELF Machines], page 32.

This field is checked against the `section-flags` configuration parameter, and pretty-printed accordingly.

9 ELF Program Headers

Segments can be stored anywhere in an ELF file. In case of relocatable objects, both sections and segments are present in the file, and they most certainly overlap. The ELF file contains a table, called the *program header table*, whose entries describe each segment. This table is sized and linked from the ELF header via the `e_phoff` field. The program header table is available in the `phdr` field of `Elf32_File` and `Elf64_File`.

The Poke types denoting entries in the program header table are `Elf32_Phdr` and `Elf64_Phdr` for ELF32 and ELF64 respectively.

9.1 Overview

```
type Elf32_Phdr =
  struct
  {
    Elf_Word p_type;
    Elf32_Off p_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    offset<Elf_Word,B> p_filesz;
    offset<Elf_Word,B> p_memsz;
    Elf_Word p_flags;
    offset<Elf_Word,B> p_align;
  };

  type Elf64_Phdr =
  struct
  {
    Elf_Word p_type;
    Elf_Word p_flags;
    Elf64_Off p_offset;
    Elf64_Addr p_vaddr;
    Elf64_Addr p_paddr;
    offset<Elf64_Xword,B> p_filesz;
    offset<Elf64_Xword,B> p_memsz;
    offset<Elf64_Xword,B> p_align;
  };
```

9.2 Fields

`p_type` Is a code identifying the type of the segment. This is one of the `ELF_PT_*` values.

The type of a segment determines what kind of contents a segment has. These are the types defined in the base spec:

`ELF_PT_NULL`
: This entry in the program header table is unused, and is ignored by ELF readers.

`ELF_PT_LOAD`
: The segment is loadable.

  The stored file size is in `p_filesz`, and the loaded size is in `p_memsz`. These sizes can be different in certain situations; for example, when the loaded data has to fulfill different alignment constraints than the stored

data. However, the stored size shall not be larger than the loaded size. This is checked by a constraint.

`ELF_PT_DYNAMIC`
: The segment contains dynamic linking information in the form of a sequence of dynamic tags. This is an array of `Elf64_Dyn` or `Elf32_Dyn`.

`ELF_PT_INTERP`
: The segment contains a null-terminated path name that the kernel uses to invoke as an interpreter.

  This segment should not occur more than once in a file. If it is present, it must precede any loadable segment entry. There is a constraint in `Elf32_File` and `Elf64_File` that checks for this.

`ELF_PT_NOTE`
: The segment contains *notes*. These are flexible annotations that are usually used in order to reflect certain "auxiliary" attributes of the ELF file. For example, the name and full version of the compiler that generated it. The format in which the notes are encoded is well defined, and supported by the elf pickles. See Chapter 11 [ELF Notes], page 28.

`ELF_PT_SHLIB`
: This value for `p_type` is reserved by the ELF specification and has undefined semantics.

`ELF_PT_PHDR`
: Segment contains the program header table itself, in both file and memory.

  This segment type may not occur more than once in a file. If it is present, it must precede any loadable segment entry. There is a constraint in `Elf32_File` and `Elf64_File` that checks for this.

`ELF_PT_TLS`
: The segment contains a thread local storage template.

`p_flags` Is a bitmap where each enabled bit flags some particular property of the segment described by this entry. This is one of the `ELF_PF_*` values. These are the segment flags defined in the base spec:

`ELF_PF_X` The segment is executable.

`ELF_PF_W` The segment is writable.

`ELF_PF_R` The segment is readable.

`p_offset` This is the file offset of the start of the segment contents.

`p_vaddr` This is the virtual address of the start of the loaded segment contents.

`p_paddr` This is the physical address of the start of the loaded segment. Since sys-v ignores physical addressing for application programs (which use virtual memory) this field has unspecified contents in executables and shared objects.

`p_filesz` Size of the segment in the file in bytes. This may be zero for some segments.

`p_memsz` Loaded size of the segment in memory. This can be bigger than `p_filesz`. See above.

`p_align` This is the alignment of the segment contents in both file and memory.

If this field is either 0 or 1, no alignment is applied. Otherwise it must contain a power of two, and `p_vaddr == p_offset % p_align`. This is checked by a constraint in `Elf32_Phdr` and `Elf64_Phdr`.

10 ELF Symbols

ELF symbols are implemented by the `Elf32_Sym` and `Elf64_Sym` struct types.

10.1 Overview

```
type Elf32_Sym =
  struct
  {
    offset<Elf_Word,B> st_name;
    Elf32_Addr st_value;
    offset<Elf_Word,B> st_size;
    Elf_Sym_Info st_info;
    Elf_Sym_Other_Info st_other;
    Elf_Half st_shndx;
  };
type Elf64_Sym =
  struct
  {
    offset<Elf_Word,B> st_name;
    Elf_Sym_Info st_info;
    Elf_Sym_Other_Info st_other;
    Elf_Half st_shndx;
    Elf64_Addr st_value;
    Elf64_Xword st_size;
  };
```

10.2 Fields

`st_name` Index into the file symbol string table. If this entry is zero it means the symbol has no name.

`st_info` The type and the binding attributes of the symbol.

```
type Elf_Sym_Info =
  struct uint<8>
  {
    uint<4> st_bind;
    uint<4> st_type;
  };
```

Where:

`st_bind` Specifies how the symbol binds. This must be one of `ELF_STB_LOCAL`, `ELF_STB_GLOBAL` or `ELF_STB_WEAK`.

`st_type` Specifies the type of the symbol. This must be one of the `ELF_STT_*` values.

The following symbol types are defined by the core specification:

`ELF_STT_NOTYPE`
: The symbol's type is not specified.

`ELF_STT_OBJECT`
: The symbol is associated with a data object, such as a variable, an array and so on.

`ELF_STT_FUNC`
: The symbol is associated with a function or other executable code.

`ELF_STT_SECTION`
: The symbol is associated with a section. This is primarily used for relocations.

`ELF_STT_FILE`
: By convention, this symbol's name gives the name of the source file associated with the object file.

  A file symbol has local binding, its section index is `ELF_SHN_ABS` and it precedes the other local symbols for the file. This is currently not checked by the pickles.

`ELF_STT_COMMON`
: The symbol labels an uninitialized common block.

`ELF_STT_TLS`
: The symbol specifies a Thread-Local Storage entity, in the form of an offset.

`st_other` This field specifies the symbol's visibility. This is one of the `ELF_STV_*` values.

The list of symbol visibility defined by the core spec are:

`ELF_STV_DEFAULT`
: The visibility of this symbol is defined by its binding. Global and weak symbols are visible outside of heir defining component. Local symbols are hidden.

`ELF_STV_PROTECTED`
: This symbol is visible in other components but it is not preemptable.

  A symbol with local binding may not have protected visibility. This is checked by a constraint in `Elf_Sym_Info`.

`ELF_STV_HIDDEN`
: This symbol is not visible to other components.

`ELF_STV_INTERNAL`
: The meaning of this attribute, if any, is processor specific.

Some machine types define other values that can be used in `st_other`. See Chapter 14 [ELF Machines], page 32.

`st_shndx` Every symbol table entry is defined in relation to some section. This holds the index into the section header table of the section related to this symbol.

However, some values for this field indicate special meanings. These are the `ELF_SHN_*` values. The core specification defines the following:

`ELF_SHN_UNDEF`
: The symbol is undefined.

`ELF_SHN_ABS`
: The symbol is absolute, meaning its value will not change because of relocation.

`ELF_SHN_COMMON`
: The symbol refers to a common block that has not yet been allocated.

`ELF_SHN_XINDEX`
: The symbol refers to a specific location within a section, but the section header index for that section is too large to e represented directly in this entry. The actual section header index is found in the associated `SHT_SYMTAB_SHNDX` section.

Some machine types define additional values with special meanings for `st_shndx`. See Chapter 14 [ELF Machines], page 32.

`st_value` Offset from the beginning of the section identified by `st_shndx`.

`st_size` Size associated with the symbol. For example, the size of a data object. Symbols that have no associated size, or unknown size, have zero in this field.

11 ELF Notes

ELF notes provide a generic mechanism for adding metadata to ELF files in the form of *notes* stored in sections. ELF notes are implemented by the `Elf_Note` struct type.

11.1 Overview

```
type Elf_Note =
  struct
  {
    Elf_Word namesz;
    Elf_Word descsz;
    Elf_Word _type;
    byte[namesz] name;
    byte[descsz] desc;
  };
```

11.2 Fields

`namesz` The first `namesz` bytes in `name` contain a NULL-terminated character representation of the entry's owner or originator.

`descsz` The first `descsz` bytes in `desc` hold the note descriptor. The ABI places no constraints on a descriptor's contents.

`_type` This word gives the interpretation of the descriptor. Each originator controls its own types. The ABI does not define what descriptors mean.

`name` Note name.

`desc` Note descriptor.

12 ELF Relocations

ELF supports two kind of relocations: relocations without addend (*REL* relocations) and relocations with addend (*RELA* relocations).

REL relocations are implemented by the `Elf32_Rel` and `Elf64_Rel` types. RELA relocations are implementd by the `Elf32_Rela` and `Elf64_Rela` types.

12.1 Overview

```
type Elf32_RelInfo =
  struct Elf_Word
    {
      uint<24> r_sym;
      uint<8> r_type;
    };

type Elf32_Rel =
  struct
  {
    Elf32_Addr r_offset;
    Elf32_RelInfo r_info;
  };

type Elf32_Rela =
  struct
  {
    Elf32_Addr r_offset;
    Elf32_RelInfo r_info;
    Elf_Sword r_addend;
  };

type Elf64_RelInfo =
  struct Elf64_Xword
  {
    uint<32> r_sym;
    uint<32> r_type;
  };

type Elf64_Rel =
  struct
  {
    Elf64_Addr r_offset;
    Elf64_RelInfo r_info;
  };

type Elf64_Rela =
  struct
  {
    Elf64_Addr r_offset;
    Elf64_RelInfo r_info;
    Elf64_Sxword r_addend;
  };
```

12.2 Fields

`r_offset` This field specifies the location at which to apply the relocation action, which itself depends on the specific kind of relocation.

This is the byte offset from the beginning of the section whose contents are to be relocated. In executables and shared objects this offset is a virtual address; in all other ELF files this refers to the stored data.

`r_info` XXX

`r_sym` XXX

`r_type` XXX

`r_addend` XXX

13 ELF Dynamic Info

XXX

14 ELF Machines

XXX

15 ELF OSes

XXX

Appendix A Indices

A.1 Concept Index

E

O

P