Decoding ELF Format Files

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.1	August 11, 2013	Initial coding.	GAM
1.0	August 23, 2013	Initial release.	GAM

Contents

Introduction **1**

Scope . . . 1

Background . . . 1

Representing an ELF File **1**

constructor . . . 1

readFile . . . 3

readChan . . . 4

decodeData . . . 4

getSectionHeaderByName . . . 4

getSectionHeaderByIndex . . . 5

foreachSectionHeader . . . 5

getSectionDataByIndex . . . 5

getSectionDataByName . . . 6

getSymbolByName . . . 6

getSymbolTable . . . 6

ReadELFHeader . . . 6

ReadSecHeaders . . . 7

ReadProgHeaders . . . 9

ReadStringTable . . . 9

ReadSymTable . . . 10

Seek . . . 12

Advance . . . 12

Read . . . 13

Conversion Specification . . . 13

Convert . . . 15

ConvertIdentifier . . . 15

Scan . . . 16

Enumerated Values **16**

Class Data . . . 17

Object File Types . . . 17

Section Types Types . . . 18

Section Indices . . . 19

Program Segment Types . . . 20

Symbol Binding . . . 20

Symbol Types . . . 21

Bit Encoded Values **21**

Section Attribute Flags 22

Program Attribute Flags 23

Helper Procedures **23**

Examples **23**

Code Organization **24**

Index **26**

List of Figures

1 ELF File Format 1

Introduction

This document describes a Tcl package named, **elfdecode**. This document is also a literate program and contains both the descriptive material as well as the source code for the package. The **elfdecode** package is used to decode ELF files that contain program executable information. ELF is a common executable format used in Unix and Linux operating systems.

Scope

The primary use case for this package is to obtain symbolic information about an executable. The package does *not* contain any capabilities to generate ELF files. This package does not attempt to be comprehensive in the same way that `libelf` is.

Background

An ELF file consists of a header and a set of sections and segments. The figure below shows the general layout. We do not describe the format in detail in this document. That information available in the ELF file format specifications.

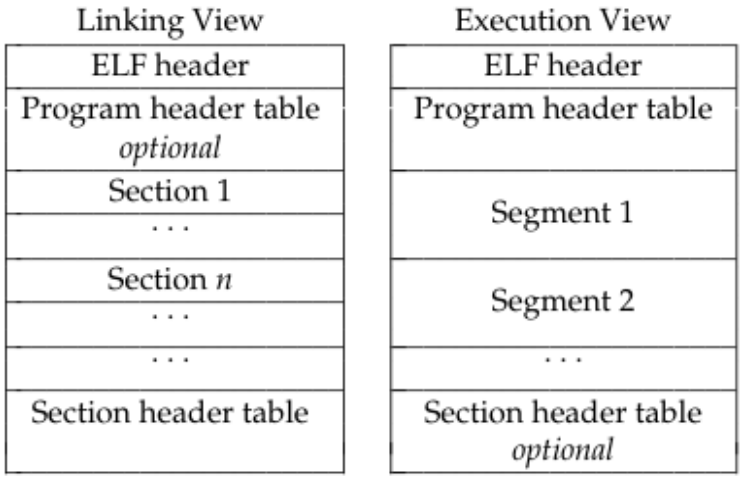

Figure 1: ELF File Format

Representing an ELF File

The **elfdecode** package uses TclOO to define a class that can represent the ELF file as an object.

```
<<elffile class>>=
::oo::class create ::elfdecode::elffile {
    <<elffile methods>>
}
```

The design of this class is to decode the binary ELF data into internal data structures which can then be used to query information. The following sections describe the methods available to **elffile** objects.

Exported Methods

constructor

The constructor for the `elffile` class only performs initialization. Other methods are used to read in the contents of an ELF file. It does create some internal data structures that are used for subsequent queries.

```
<<elffile methods>>=
constructor {} {
    set svcname ::elfdecode[self]
    ::logger::init $svcname
    ::logger::import -all -namespace log $svcname

    namespace import ::ral::*
    namespace import ::ralutil::*

    <<elffile data structures>>
}
```

elffile Variables We hold the ELF data as a large string and use a variable to track our `position` in that string where binary decoding operations take place. Several methods are used to track the data and implement the notion of a cursor into the data.

```
<<elffile data structures>>=
my variable elfdata
my variable position
```

The ELF format defines a set of data structures and, naturally enough, the names and structures used here mirror, but are not identical to, the one defined in the ELF format specification

The ELF file starts with a header. We hold the header as a `dict` with the following keys:

- `ei_class`
- `ei_data`
- `ei_version`
- `e_type`
- `e_machine`
- `e_version`
- `e_entry`
- `e_phoff`
- `e_shoff`
- `e_flags`
- `e_ehsize`
- `e_phentsize`
- `e_phnum`
- `e_shentsize`
- `e_shnum`
- `e_shstrndx`

The meaning of the keys is available from the ELF format specification.

The remainder of the ELF data is in the form of sections. There are section headers, program headers and a symbol table. Naming information is in the form of integer offsets into a string table. All the string information is collected into the string table. Since the header information is very tabular oriented, we use TclRAL `relvars` as the internal data structures to hold the parsed ELF data. The `relvars` are also convenient for query operations.

ELF Section Header

```
<<elffile data structures>>=
relvar create ElfSecHeader {
    sh_index int
    sh_name string
    sh_type string
    sh_flags string
    sh_addr int
    sh_offset int
    sh_size int
    sh_link int
    sh_info int
    sh_addralign int
    sh_entsize int
} sh_index
```

Each section in the ELF file is described by a section header. The ELF section header attributes names are taken from the ELF format specification. The `sh_name` attribute is the name of the section as a string and has been resolved through the string table.

ELF Program Header

```
<<elffile data structures>>=
relvar create ElfProgHeader {
    p_index int
    p_type string
    p_offset int
    p_vaddr int
    p_paddr int
    p_filesz int
    p_memsz int
    p_flags string
    p_align int
} p_index
```

The ELF program header describes how an executable is to be loaded into memory. The ELF program header attributes names are also taken from the specification. Since we are not particularly concerned with running the program program contained in the ELF file, we only parse the header.

ELF Symbol Table

```
<<elffile data structures>>=
relvar create ElfSymbolTable {
    st_index int
    st_name string
    st_value int
    st_size int
    st_info int
    st_bind string
    st_type string
    st_other int
    st_shndx string
} st_index
```

Finally, the symbol table attribute names are also taken from the specification. The `st_name` is a string table resolved name for the symbol. Note also that symbol names are *not* unique, only the index into the symbol table is guaranteed to be unique.

readFile

Several methods are provided to obtain the data from and ELF file. The `readFile` method will read the data directly from a named file. This method returns a dictionary containing the ELF file header.

```
<<elffile methods>>=
method readFile {fname} {
    set elfchan [::open $fname rb]
    try {
        return [my decodeData [chan read $elfchan]]
    } finally {
        chan close $elfchan
    }
}
```

readChan

The `readChan` method allow you to open a Tcl channel by some other means and have it supply the ELF file data. This method returns a dictionary containing the ELF file header.

```
<<elffile methods>>=
method readChan {chan} {
    # Make sure we are in binary mode!
    chan configure $chan -translation binary
    return [my decodeData [chan read $chan]]
}
```

decodeData

And finally, the `decodeData` method does the real work of sequencing the decoding of the binary ELF data and building up the internal data structures. This method returns a dictionary containing the ELF file header.

```
<<elffile methods>>=
method decodeData {data} {
    my variable elfdata
    set elfdata $data
    my variable position
    set position 0
    my variable elfheader
    set elfheader [my ReadElfHeader]
    log::info "ELF header = \"$elfheader\""
    dict with elfheader {
        my ReadSecHeaders $e_shoff $e_shnum $e_shentsize $e_shstrndx
        my ReadProgHeaders $e_phoff $e_phnum $e_phentsize
        my ReadSymTable
    }
    return $elfheader
}
```

After reading in the header, the section headers, program headers and symbol table are parsed out of the binary data.

getSectionHeaderByName

The `getSectionHeaderByName` method returns a list of dictionaries of attributes of the section header given by `name`. ELF section names are not guaranteed to be unique only section indices are unique. So it is possible that the returned list has more than one element.

```
<<elffile methods>>=
method getSectionHeaderByName {name} {
    return [pipe {
        relvar set ElfSecHeader |
```

```
        relation restrictwith ~ {$sh_name eq $name} |
        relation body ~
    }]
}
```

The implementation is a query on the `ElfSecHeader` relvar, returning the body of the resulting relation, which is a list of dictionaries.

getSectionHeaderByIndex

The `getSectionHeaderByIndex` method returns a dictionary of the attribute of the section header given by `index`. ELF section header indices are unique.

```
<<elffile methods>>=
method getSectionHeaderByIndex {index} {
    set header [relvar restrictone ElfSecHeader sh_index $index]
    if {[relation isempty $header]} {
        error "no section header with index, \"$index\""
    }
    return [tuple get [relation tuple $headers]]
}
```

foreachSectionHeader

The `foreachSectionHeader` method provides a control structure that iterates across each section header. The `varname` argument is the name of a variable to which a dictionary value is assigned that contains a section heading. The `script` is then executed.

```
<<elffile methods>>=
method foreachSectionHeader {varname script} {
    upvar 1 $varname header
    foreach header [relation body [relvar set ElfSecHeader]] {
        try {
            uplevel 1 $script
        } on error {result opts} {
            return -options $opts $result
        } on break {
            return -level 0 -code break
        } on continue {
            return -level 0 -code continue
        }
    }
}
```

getSectionDataByIndex

The `getSectionData` method returns a string containing the binary information contained in the section given by `index`.

```
<<elffile methods>>=
method getSectionDataByIndex {index} {
    set header [my getSectionHeaderByIndex $index]
    my Seek [dict get $header sh_offset]
    return [my Read [dict get $header sh_size]]
}
```

getSectionDataByName

The `getSectionData` method returns a string containing the binary information contained in the section given by `name`. If the section given by `name` is not unique, then an error is thrown.

```
<<elffile methods>>=
method getSectionDataByName {name} {
    set headers [my getSectionHeaderByName $name]
    if {[llength $headers] != 1} {
        error "cannot find unique section named, \"$name\""
    }
    set header [lindex $headers 0]
    my Seek [dict get $header sh_offset]
    return [my Read [dict get $header sh_size]]
}
```

getSymbolByName

The `getSymbolByName` method returns a list of dictionaries of attributes of the symbol given by `name`. Since symbol names are not unique, the returned list may have a length greater than one.

```
<<elffile methods>>=
method getSymbolByName {name} {
    return [pipe {
        relvar set ElfSymbolTable |
        relation restrictwith ~ {$st_name eq $name} |
        relation body ~
    }]
}
```

getSymbolTable

The `getSymbolTable` method returns a TclRAL relation value that holds the ELF symbol table for the file. This value can then be used to perform arbitrary queries on the symbols.

```
<<elffile methods>>=
method getSymbolTable {} {
    return [relvar set ElfSymbolTable]
}
```

Unexported Methods

ReadELFHeader

The `ReadELFHeader` method reads the header that is the beginning of an ELF file and converts the binary information into other data types. It checks that the ELF identification is correct to insure that we are actually dealing with an ELF formatted file.

```
<<elffile methods>>=
method ReadElfHeader {} {
    # Read the 16 byte identifier field and make sure we are dealing with an
    # ELF file. The header is of a fixed 16 byte format.  We need to squirrel
    # away the class and data type information so we can distinguish between
    # the various file types and byte orders.
    my variable ei_class ei_data
    my ConvertIdentifier\
        elfmag0 elfmagName ei_class ei_data ei_version ei_osabi\
```

```
        ei_abiversion ei_pad

    if {$elfmag0 != 0x7f || $elfmagName ne "ELF"} {
        error "bad ELF magic number, [format %#x $elfmag0] $elfmagName"
    }
    # At this point we know we have a good ELF file.
    # Convert some numbers into more meaningful strings.
    set ei_class [my XlateEnumValue ei_class $ei_class]
    set ei_data [my XlateEnumValue ei_data $ei_data]

    # Convert the remainder of the header.
    my Convert header\
        e_type e_machine e_version e_entry e_phoff e_shoff e_flags\
        e_ehsize e_phentsize e_phnum e_shentsize e_shnum e_shstrndx
    my Advance [expr {$e_ehsize - 16}]

    set e_type [my XlateEnumValue e_type $e_type]
    set e_version [expr {$e_version == 1 ? "EV_CURRENT" : "EV_NONE"}]

    return [dict create\
        ei_class $ei_class\
        ei_data $ei_data\
        ei_version $ei_version\
        ei_osabi $ei_osabi\
        ei_abiversion $ei_abiversion\
        e_type $e_type\
        e_machine $e_machine\
        e_version $e_version\
        e_entry $e_entry\
        e_phoff $e_phoff\
        e_shoff $e_shoff\
        e_flags $e_flags\
        e_ehsize $e_ehsize\
        e_phentsize $e_phentsize\
        e_phnum $e_phnum\
        e_shentsize $e_shentsize\
        e_shnum $e_shnum\
        e_shstrndx $e_shstrndx\
    ]
}
```

ReadSecHeaders

The `ReadSecHeaders` method reads the section headers from the ELF file. This method also resolves the section name into a string rather than just leaving it as an index into the string table. Section header names are found in the string table given by the `e_shstrndx` value in the ELF header.

The strategy here is to parse out the section header information into a list of dictionaries. Then we create a relation value using the list of dictionaries as the body of the relation. Next, we read the string table that contains the section header names. Finally, the string table index values are replaced with names and the `ElfSecHeader` relvar is populated.

```
<<elffile methods>>=
method ReadSecHeaders {e_shoff e_shnum e_shentsize e_shstrndx} {
    if {$e_shnum == 0} {
        return
    }
    # Seek to where the section header table is located.
    my Seek $e_shoff

    set sectuples [list]
```

```
    # Read and convert the entire array of section headers.
    for {set secNo 0} {$secNo < $e_shnum} {incr secNo} {
        my Convert section\
                sh_name sh_type sh_flags sh_addr sh_offset\
                sh_size sh_link sh_info sh_addralign sh_entsize

        set sh_type [my XlateEnumValue sh_type $sh_type]
        # The first entry in the section header has a type of 0 and
        # does not contain any useful information.
        if {$sh_type eq "SHT_NULL"} {
            continue
        }
        set sh_flags [my XlateFlagBits sh_flags $sh_flags]
        set secheader [list\
            sh_index        $secNo\
            sh_name         $sh_name\
            sh_type         $sh_type\
            sh_flags        $sh_flags\
            sh_addr         $sh_addr\
            sh_offset       $sh_offset\
            sh_size         $sh_size\
            sh_link         $sh_link\
            sh_info         $sh_info\
            sh_addralign    $sh_addralign\
            sh_entsize      $sh_entsize\
        ]
        log::debug "section header: $secheader"
        lappend sectuples $secheader
    }
    # Create a relation with all the section header information.
    set sections [relation create {
        sh_index int
        sh_name int
        sh_type string
        sh_flags string
        sh_addr int
        sh_offset int
        sh_size int
        sh_link int
        sh_info int
        sh_addralign int
        sh_entsize int
    } {*}$sectuples]
    #log::info \n[relformat $sections sections]

    # Now we want to update the information stored with the section
    # headers to replace the string index used as the name with
    # the string that is the section name.
    set shstrsec [relation restrictwith $sections\
            {$sh_index == $e_shstrndx}]
    if {[relation isnotempty $shstrsec]} {
        relation assign $shstrsec sh_type sh_offset sh_size
        if {$sh_type eq "SHT_STRTAB"} {
            set shnames [my ReadStringTable $sh_offset $sh_size]
            #log::info \n[relformat $shnames SectionNames]
            relvar set ElfSecHeader [pipe {
                relation join $sections $shnames -using {sh_name st_index} |
                relation eliminate ~ sh_name |
                relation rename ~ st_string sh_name
            }]
        }
    }
```

```
    log::info \n[relformat [relvar set ElfSecHeader] ElfSecHeader]
}
```

ReadProgHeaders

The `ReadProgHeaders` method reads in the program headers. Program headers don't have naming information, so we can insert into the `ElfProgHeader` relvar directly.

```
<<elffile methods>>=
method ReadProgHeaders {e_phoff e_phnum e_phentsize} {
    if {$e_phnum == 0} {
        return
    }
    # Seek to where the section header table is
    my Seek $e_phoff

    for {set phNo 0} {$phNo < $e_phnum} {incr phNo} {
        my Convert prog\
                p_type p_offset p_vaddr p_paddr\
                p_filesz p_memsz p_flags p_align
        # Turns some numbers into strings.
        set p_type [my XlateEnumValue p_type $p_type]
        set p_flags [my XlateFlagBits p_flags $p_flags]

        relvar insert ElfProgHeader [list\
            p_index     $phNo\
            p_type      $p_type\
            p_offset    $p_offset\
            p_vaddr     $p_vaddr\
            p_paddr     $p_paddr\
            p_filesz    $p_filesz\
            p_memsz     $p_memsz\
            p_flags     $p_flags\
            p_align     $p_align\
        ]
    }
    log::info \n[relformat [relvar set ElfProgHeader] ElfProgHeader]
}
```

ReadStringTable

The `ReadStringTable` method parses a string table section and returns a relation value containing the section information.

Note
Initially this was implemented by finding the `NUL` terminating characters using `string first` and then scanning the non-`NUL` characters. This turned out to be rather slow, so this implementation pulls the string table into a separate variable and uses `split` to convert the binary data into a list of strings.

```
<<elffile methods>>=
method ReadStringTable {sh_offset sh_size} {
    # seek to the file location
    my Seek $sh_offset
    set strings [my Read $sh_size]
    set strOff 0
    # The strings are just packed NUL terminated ASCII strings.
    # Just split on the NUL character to unbundle the them.
```

```
    set tuples [list]
    foreach s [split $strings "\0"] {
        if {[string length $s] != 0} {
            lappend tuples [list st_index $strOff st_string $s]
        }
        incr strOff [expr {[string length $s] + 1}]
    }

    return [relation create {st_index int st_string string} {*}$tuples]
}
```

ReadSymTable

The `ReadSymTable` method parses of the ELF file symbols and stores them in a `relvar`. Like `ReadSecHeaders`, this method resolves indices into a string table into the actual strings so that the symbol name is easily accessible. According to the ELF specification, the section named, ".symtab", is a special section name that contains the symbol table.

```
<<elffile methods>>=
method ReadSymTable {} {
    set symsecs [my getSectionHeaderByName .symtab]
    if {[llength $symsecs] != 1} {
        log::info "cannot find symbol table section named, \".symtab\""
        return
    }
    set symheader [lindex $symsecs 0]
    dict with symheader {
        if {$sh_type ne "SHT_SYMTAB"} {
            error "expected symbol table section type of \"SHT_SYMTAB\",\
                    got \"$sh_type\""
        }
        if {$sh_entsize == 0} {
            error "expected non-zero symbol table entry size,\
                    got \"$sh_entsize\""
        }
        my Seek $sh_offset
        set nSyms [expr {$sh_size / $sh_entsize}]
        my variable ei_class
        # Accumulate a list of dictionaries to use as a relation value body.
        set symtuples [list]
        for {set symindex 0} {$symindex < $nSyms} {incr symindex} {
            # As it turns out, the ordering of fields is different for 32-bit
            # vs. 64-bit ELF files. The field contents are the same, but we
            # slightly different procs to read the symbol table entries.
            set sym [expr {$ei_class eq "ELFCLASS32" ?\
                [my ReadElf32Sym] : [my ReadElf64Sym]}]
            my Advance $sh_entsize
            if {[dict size $sym] == 0} {
                log::notice "discarding unnamed symbol table entry,\
                        \"$symindex\""
                continue
            }
            dict set sym st_index $symindex
            # Decode the st_info field and translate the numbers into strings.
            dict set sym st_bind [my XlateEnumValue st_bind\
                [expr {[dict get $sym st_info] >> 4}]]
            dict set sym st_type [my XlateEnumValue st_type\
                [expr {[dict get $sym st_info] & 0xf}]]
            # Decode the symbol section index reference
            dict set sym st_shndx [my XlateEnumValue shndx\
                [dict get $sym st_shndx]]
```

```
            lappend symtuples $sym
        }
        # Create a relation with the symbol information.
        set syms [relation create {
            st_index int
            st_nindex int
            st_value int
            st_size int
            st_info int
            st_bind string
            st_type string
            st_other int
            st_shndx string} {*}$symtuples]
        #log::debug \n[relformat $syms syms]

        # Now we want to read the string table and resolve the index values
        # into actual names.  First, find the string table. Its name is
        # also special.
        set strsecs [my getSectionHeaderByName .strtab]
        if {[llength $strsecs] != 1} {
            log::info "cannot find string table section named, \".strtab\""
            return
        }
        set strheader [lindex $strsecs 0]
        dict with strheader {
            if {$sh_type ne "SHT_STRTAB"} {
                error "expected symbol table section type of \"SHT_STRTAB\",\
                        got \"$sh_type\""
            }
            if {$sh_entsize != 0} {
                error "expected zero string table entry size,\
                        got \"$sh_entsize\""
            }
            set symnames [my ReadStringTable $sh_offset $sh_size]
        }
    }
    # Join the string table to the symbol table and replace the string indices
    # with symbol names.
    relvar set ElfSymbolTable [pipe {
        relation join $syms $symnames -using {st_nindex st_index} |
        relation eliminate ~ st_nindex |
        relation rename ~ st_string st_name
    }]
    log::info \n[relformat [relvar set ElfSymbolTable] ElfSymbolTable]
    return
}
```

The only real difference in parsing a 32-bit symbol and a 64-bit symbol is the order of the fields. Since our `Convert` method assigns scanned fields in a particular order, we use different procedures for the conversion.

```
<<elffile methods>>=
method ReadElf32Sym {} {
    my Convert symbol st_index st_value st_size st_info st_other st_shndx
    # The "zero" string index is undefined and implies the symbol
    # has not name. We discard those.
    if {$st_index == 0} {
        return
    }
    return [list\
        st_nindex   $st_index\
        st_value    $st_value\
```

```
        st_size     $st_size\
        st_info     $st_info\
        st_other    $st_other\
        st_shndx    $st_shndx\
    ]
}
```

```
<<elffile methods>>=
method ReadElf64Sym {} {
    my Convert symbol st_index st_info st_other st_shndx st_value st_size
    # The "zero" string index is undefined and implies the symbol
    # has not name. We discard those.
    if {$st_index == 0} {
        return
    }
    return [list\
        st_nindex   $st_index\
        st_value    $st_value\
        st_size     $st_size\
        st_info     $st_info\
        st_other    $st_other\
        st_shndx    $st_shndx\
    ]
}
```

Managing a Cursor Into the ELF Data

Since we store the binary ELF data as one big string, it is useful to be able to track our location in that string and to implement the notion of a cursor where the next action will take place. The following methods accomplish that.

Seek

```
<<elffile methods>>=
method Seek {offset} {
    my variable elfdata position

    if {$offset < 0} {
        set position 0
    } elseif {$offset > [string length $elfdata]} {
        set position [string length $elfdata]
    } else {
        set position $offset
    }
    #log::debug "seek to $position"
}
```

Advance

```
<<elffile methods>>=
method Advance {offset} {
    my variable elfdata
    my variable position
    set newpos [expr {$position + $offset}]

    if {$newpos < 0} {
        set position 0
```

```
    } elseif {$newpos > [string length $elfdata]} {
        set position [string length $elfdata]
    } else {
        set position $newpos
    }
    #log::debug "advance to $position"
}
```

Read

```
<<elffile methods>>=
method Read {count} {
    my variable elfdata
    my variable position

    set end [expr {$position + $count - 1}]
    if {$end >= [string length $elfdata]} {
        error "attempt to read beyond end of ELF data"
    }
    set data [string range $elfdata $position $end]
    set position [expr {$end + 1}]
    return $data
}
```

Conversion Specification

In order to support both 64 bit and 32 bit ELF files and both little endian and big endian formats, we need different format specifiers to `binary scan`. We will organize the format specifiers by name with variations for 32 / 64 bits and for little / big endian. Since this information is invariable for all `elffile` objects, we place it in the class namespace so all `elffile` objects can have handy access.

```
<<elffile class data>>=
namespace eval [info object namespace ::elfdecode::elffile] {
    namespace import ::ral::*
    namespace import ::ralutil::*

    <<elffile data definitions>>
}
```

To convert ELF header data, you need to know the type of header, whether the file is 32-bit or 64-bit and whether the data is little endian or bit endian. So we build up a relvar that is identified by these three values and contains the `binary scan` format string needed for the conversion. The size of the entry is also included. Sometimes this `Size` attribute is set to zero. This implies that the size is known from some other information. For example, the size of symbol table entries is known from the section header. For those headers with a fixed entry size, then the `Size` attribute value can be used to advance the implicit data cursor to the next entry.

```
<<elffile data definitions>>=
relvar create ElfFormatSpec {
    Name        string
    Class       string
    DataType    string
    Format      string
    Size        int
} {Name Class DataType}

relvar insert ElfFormatSpec {
    Name        header
```

```
    Class       ELFCLASS32
    DataType    ELFDATA2LSB
    Format      {su su iu iu iu iu iu su su su su su su}
    Size        0
} {
    Name        header
    Class       ELFCLASS32
    DataType    ELFDATA2MSB
    Format      {Su Su Iu Iu Iu Iu Iu Su Su Su Su Su Su}
    Size        0
} {
    Name        header
    Class       ELFCLASS64
    DataType    ELFDATA2LSB
    Format      {su su iu wu wu wu iu su su su su su su}
    Size        0
} {
    Name        header
    Class       ELFCLASS64
    DataType    ELFDATA2MSB
    Format      {Su Su Iu Wu Wu Wu Iu Su Su Su Su Su Su}
    Size        0
} {
    Name        section
    Class       ELFCLASS32
    DataType    ELFDATA2LSB
    Format      {iu iu iu iu iu iu iu iu iu iu}
    Size        40
} {
    Name        section
    Class       ELFCLASS32
    DataType    ELFDATA2MSB
    Format      {Iu Iu Iu Iu Iu Iu Iu Iu Iu Iu}
    Size        40
} {
    Name        section
    Class       ELFCLASS64
    DataType    ELFDATA2LSB
    Format      {iu iu wu wu wu wu iu iu wu wu}
    Size        64
} {
    Name        section
    Class       ELFCLASS64
    DataType    ELFDATA2MSB
    Format      {Iu Iu Wu Wu Wu Wu Iu Iu Wu Wu}
    Size        64
} {
    Name        prog
    Class       ELFCLASS32
    DataType    ELFDATA2LSB
    Format      {iu iu iu iu iu iu iu iu}
    Size        32
} {
    Name        prog
    Class       ELFCLASS32
    DataType    ELFDATA2MSB
    Format      {Iu Iu Iu Iu Iu Iu Iu Iu}
    Size        32
} {
    Name        prog
    Class       ELFCLASS64
    DataType    ELFDATA2LSB
```

```
    Format      {iu iu wu wu wu wu wu wu}
    Size        56
} {
    Name        prog
    Class       ELFCLASS64
    DataType    ELFDATA2MSB
    Format      {Iu Iu Wu Wu Wu Wu Wu Wu}
    Size        56
} {
    Name        symbol
    Class       ELFCLASS32
    DataType    ELFDATA2LSB
    Format      {iu iu iu cu cu su}
    Size        0
} {
    Name        symbol
    Class       ELFCLASS32
    DataType    ELFDATA2MSB
    Format      {Iu Iu Iu cu cu Su}
    Size        0
} {
    Name        symbol
    Class       ELFCLASS64
    DataType    ELFDATA2LSB
    Format      {iu cu cu su wu wu}
    Size        0
} {
    Name        symbol
    Class       ELFCLASS64
    DataType    ELFDATA2MSB
    Format      {Iu cu cu Su Wu Wu}
    Size        0
}
```

Convert

The **Convert** method finds the conversion format, convert the binary data and advances our notion of where we are in the data.

```
<<elffile methods>>=
method Convert {fmtname args} {
    my variable ei_class ei_data
    set fmtrel [relvar restrictone [classns]::ElfFormatSpec\
        Name        $fmtname\
        Class       $ei_class\
        DataType    $ei_data\
    ]
    if {[relation isempty $fmtrel]} {
        error "cannot find format specification,\
                \"$fmtname $ei_class $ei_data\""
    }
    my Scan [relation extract $fmtrel Format] $args
    my Advance [relation extract $fmtrel Size]
}
```

ConvertIdentifier

The first 16 bytes of an ELF file contain a file identifier. We convert it with a separate method since it is always of a fixed size and format.

```
<<elffile methods>>=
method ConvertIdentifier {args} {
    my Scan {cu a3 cu cu cu cu cu cu7} $args
    my Advance 16
}
```

Scan

The **Scan** method is a simple control structure wrapper around the **binary scan** command. Here we perform the conversion into variables that are passed by name. This gives us a chance to check that the conversion did indeed happen properly.

```
<<elffile methods>>=
method Scan {fmt arglist} {
    my variable elfdata
    my variable position

    foreach arg $arglist {
        upvar 2 $arg $arg
    }
    set cvtd [binary scan $elfdata "@$position $fmt" {*}$arglist]
    if {$cvtd != [llength $arglist]} {
        error "expected to convert \"[llength $arglist]\" values,\
            actually converted \"$cvtd\""
    }
}
```

Enumerated Values

Since Tcl encourages the use of strings and strings are easier to understand than integer encoded data, there are several conversions from binary that are best represented as string data. These are fields and flags that have enumerated values defined by the ELF format specification. The data consists of a set of constant values and so we define them directly into the namespace of the class.

It is often the case that these enumerated values have both a fixed set of enumerators plus allowances for vendors and operating systems to include specific values within ranges of the enumeration. The decoding of these specific values is determined by the generator of the ELF file, but they must fall within a given numeric range. Here we define some data structures that can be used to decode this type of symbolic information.

```
<<elffile data definitions>>=
relvar create ElfEnumerator {
    SymbolType  string
    Enumerator  string
    Value       int
} {SymbolType Enumerator} {SymbolType Value}
```

Enumeration ranges require another attribute to define the upper and lower bound.

```
<<elffile data definitions>>=
relvar create ElfEnumRange {
    SymbolType  string
    Enumerator  string
    LowValue    int
    HighValue   int
} {SymbolType Enumerator} {SymbolType LowValue}
```

The processing to do the decoding looks first in the set of distinct enumerated values and then in the range values. If nothing is found, then the original value is simply returned.

```
<<elffile methods>>=
method XlateEnumValue {symtype etor} {
    set enum [relvar restrictone [classns]::ElfEnumerator\
            SymbolType $symtype Value $etor]
    if {[relation isnotempty $enum]} {
        return [relation extract $enum Enumerator]
    }
    set range [pipe {
        relvar set [classns]::ElfEnumRange |
        relation restrictwith ~ {
            $SymbolType eq $symtype && $etor >= $LowValue &&\
            $etor <= $HighValue}
    }]
    if {[relation isnotempty $range]} {
        relation assign $range Enumerator LowValue
        return $Enumerator\[[format %#x $etor]\]
    } else {
        # If all else fails, just return the original value.
        return $etor
    }
}
```

In the sections below, we define the tediously long list of enumerated values for the various header and section fields. We follow the convention of giving the symbol types the same name as the field we are converting.

Class Data

```
<<elffile data definitions>>=
relvar insert ElfEnumerator {
    SymbolType  ei_class
    Enumerator  ELFCLASS32
    Value       1
} {
    SymbolType  ei_class
    Enumerator  ELFCLASS64
    Value       2
} {
    SymbolType  ei_data
    Enumerator  ELFDATA2LSB
    Value       1
} {
    SymbolType  ei_data
    Enumerator  ELFDATA2MSB
    Value       2
}
```

Object File Types

```
<<elffile data definitions>>=
relvar insert ElfEnumerator {
    SymbolType  e_type
    Enumerator  ET_NONE
    Value       0
} {
    SymbolType  e_type
    Enumerator  ET_REL
    Value       1
```

```
} {
    SymbolType  e_type
    Enumerator  ET_EXEC
    Value       2
} {
    SymbolType  e_type
    Enumerator  ET_DYN
    Value       3
} {
    SymbolType  e_type
    Enumerator  ET_CORE
    Value       4
}

relvar insert ElfEnumRange {
    SymbolType  e_type
    Enumerator  ET_LOOS
    LowValue    0xfe00
    HighValue   0xfeff
} {
    SymbolType  e_type
    Enumerator  ET_LOPROC
    LowValue    0xff00
    HighValue   0xffff
}
```

Section Types Types

```
<<elffile data definitions>>=
relvar insert ElfEnumerator {
    SymbolType  sh_type
    Enumerator  SHT_NULL
    Value       0
} {
    SymbolType  sh_type
    Enumerator  SHT_PROGBITS
    Value       1
} {
    SymbolType  sh_type
    Enumerator  SHT_SYMTAB
    Value       2
} {
    SymbolType  sh_type
    Enumerator  SHT_STRTAB
    Value       3
} {
    SymbolType  sh_type
    Enumerator  SHT_RELA
    Value       4
} {
    SymbolType  sh_type
    Enumerator  SHT_HASH
    Value       5
} {
    SymbolType  sh_type
    Enumerator  SHT_DYNAMIC
    Value       6
} {
    SymbolType  sh_type
    Enumerator  SHT_NOTE
```

```
    Value       7
} {
    SymbolType  sh_type
    Enumerator  SHT_NOBITS
    Value       8
} {
    SymbolType  sh_type
    Enumerator  SHT_REL
    Value       9
} {
    SymbolType  sh_type
    Enumerator  SHT_SHLIB
    Value       10
} {
    SymbolType  sh_type
    Enumerator  SHT_DYNSYM
    Value       11
}

relvar insert ElfEnumRange {
    SymbolType  sh_type
    Enumerator  SHT_LOOS
    LowValue    0x60000000
    HighValue   0x6fffffff
} {
    SymbolType  sh_type
    Enumerator  SHT_LOPROC
    LowValue    0x70000000
    HighValue   0x7fffffff
}
```

Section Indices

```
<<elffile data definitions>>=
relvar insert ElfEnumerator {
    SymbolType  shndx
    Enumerator  SHN_UNDEF
    Value       0
} {
    SymbolType  shndx
    Enumerator  SHN_ABS
    Value       0xfff1
} {
    SymbolType  shndx
    Enumerator  SHN_COMMON
    Value       0xfff2
}

relvar insert ElfEnumRange {
    SymbolType  shndx
    Enumerator  SHN_LOPROC
    LowValue    0xff00
    HighValue   0xff1f
} {
    SymbolType  shndx
    Enumerator  SHT_LOOS
    LowValue    0xff20
    HighValue   0xff3f
}
```

Program Segment Types

```
<<elffile data definitions>>=
relvar insert ElfEnumerator {
    SymbolType  p_type
    Enumerator  PT_NULL
    Value       0
} {
    SymbolType  p_type
    Enumerator  PT_LOAD
    Value       1
} {
    SymbolType  p_type
    Enumerator  PT_DYNAMIC
    Value       2
} {
    SymbolType  p_type
    Enumerator  PT_INTERP
    Value       3
} {
    SymbolType  p_type
    Enumerator  PT_NOTE
    Value       4
} {
    SymbolType  p_type
    Enumerator  PT_SHLIB
    Value       5
} {
    SymbolType  p_type
    Enumerator  PT_PHDR
    Value       6
}

relvar insert ElfEnumRange {
    SymbolType  p_type
    Enumerator  PT_LOOS
    LowValue    0x60000000
    HighValue   0x6fffffff
} {
    SymbolType  p_type
    Enumerator  PT_LOPROC
    LowValue    0x70000000
    HighValue   0x7fffffff
}
```

Symbol Binding

```
<<elffile data definitions>>=
relvar insert ElfEnumerator {
    SymbolType  st_bind
    Enumerator  STB_LOCAL
    Value       0
} {
    SymbolType  st_bind
    Enumerator  STB_GLOBAL
    Value       1
} {
    SymbolType  st_bind
    Enumerator  STB_WEAK
    Value       2
```

```
}

relvar insert ElfEnumRange {
    SymbolType  st_bind
    Enumerator  STB_LOOS
    LowValue    10
    HighValue   12
} {
    SymbolType  st_bind
    Enumerator  STB_LOPROC
    LowValue    13
    HighValue   15
}
```

Symbol Types

```
<<elffile data definitions>>=
relvar insert ElfEnumerator {
    SymbolType  st_type
    Enumerator  STT_NOTYPE
    Value       0
} {
    SymbolType  st_type
    Enumerator  STT_OBJECT
    Value       1
} {
    SymbolType  st_type
    Enumerator  STT_FUNC
    Value       2
} {
    SymbolType  st_type
    Enumerator  STT_SECTION
    Value       3
} {
    SymbolType  st_type
    Enumerator  STT_FILE
    Value       4
}

relvar insert ElfEnumRange {
    SymbolType  st_type
    Enumerator  STT_LOOS
    LowValue    10
    HighValue   12
} {
    SymbolType  st_type
    Enumerator  STT_LOPROC
    LowValue    13
    HighValue   15
}
```

Bit Encoded Values

Some of the fields in an ELF file are bit encoded. Here we define a `relvar` to hold the bit field definition and its associated symbolic name.

```
<<elffile data definitions>>=
```

```
relvar create ElfBitSymbol {
    SymbolType  string
    SymbolName  string
    Offset      int
    Length      int
} {SymbolType SymbolName} {SymbolType Offset}
```

To translate a bit field, we find all the bit field definitions associated with the symbol and iterate across them to test if they are set. For all the bit fields that have a value, we include the symbolic name. In the end we have a list of symbol names for all the fields that are set.

```
<<elffile methods>>=
method XlateFlagBits {symtype value} {
    set result [list]
    set bitsyms [pipe {
        relvar set [classns]::ElfBitSymbol |
        relation restrictwith ~ {$SymbolType eq $symtype}
    }]
    relation foreach bitsym $bitsyms -ascending Offset {
        relation assign $bitsym SymbolName Offset Length
        # Compute a mask from the bit field definition.
        set mask [expr {((1 << $Length) - 1) <<  $Offset}]
        set mvalue [expr {$value & $mask}]
        if {$mvalue != 0} {
            # We treat single bit fields differently. Multi-bit fields have a
            # value associated with them.
            if {$Length == 1} {
                lappend result $SymbolName
            } else {
                lappend result "$SymbolName\([expr {$mvalue >> $Offset}]\)"
            }
        }
    }
    return $result
}
```

What follows is then the long and tedious definitions of exactly what all the bit fields are in the various flag components of the ELF data.

Section Attribute Flags

```
<<elffile data definitions>>=
relvar insert ElfBitSymbol {
    SymbolType  sh_flags
    SymbolName  SHF_WRITE
    Offset      0
    Length      1
} {
    SymbolType  sh_flags
    SymbolName  SHF_ALLOC
    Offset      1
    Length      1
} {
    SymbolType  sh_flags
    SymbolName  SHF_EXECINST
    Offset      2
    Length      1
} {
    SymbolType  sh_flags
    SymbolName  SHF_MASKPROC
```

```
    Offset      28
    Length      4
}
```

Program Attribute Flags

```
<<elffile data definitions>>=
relvar insert ElfBitSymbol {
    SymbolType  p_flags
    SymbolName  PF_X
    Offset      0
    Length      1
} {
    SymbolType  p_flags
    SymbolName  PF_W
    Offset      1
    Length      1
} {
    SymbolType  p_flags
    SymbolName  PF_R
    Offset      2
    Length      1
} {
    SymbolType  p_flags
    SymbolName  PF_MASKOS
    Offset      16
    Length      8
} {
    SymbolType  p_flags
    SymbolName  PF_MASKPROC
    Offset      24
    Length      8
}
```

Helper Procedures

It is convenient to be able to determine the namespace of the an `elffile` object so that we can access the class `relvars` that contain descriptive information. We accomplish that by adding a `proc` to the `::oo::Helpers` namespace.

```
<<helpers>>=
proc ::oo::Helpers::classns {} {
    return [info object namespace [uplevel 1 {self class}]]
}
```

Examples

A simple example shows how the class is used. Assume we have a file named, `myelffile`, that is an ELF format file. We can then look up the value of the `printf` symbol as shown below.

```
package require Tcl 8.6
package require elfdecode

elfdecode elffile create ef
ef readFile myelffile
```

```
set sym [ef getSymbolByName printf]
chan puts $sym
```

A more complicated example shows how to query the symbol table to obtain a list of file names contained in the ELF file.

```
package require ral
package require ralutil

namespace import ::ral::*
namespace import ::ralutil::*

set files [pipe {
    ef getSymbolTable |
    relation restrictwith ~ {$st_type eq "STT_FILE"} |
    relation list ~ st_name
}]

chan puts $files
```

Code Organization

This document is a literate program and the source code for the `elfdecode` package can be extracted from it using the `atangle` program (see mrtools). The primary source can be extracted from the root called, `elfdecode.tcl`.

```
<<elfdecode.tcl>>=
# DO NOT EDIT THIS FILE!
# THIS FILE IS AUTOMATICALLY GENERATED FROM A LITERATE PROGRAM SOURCE FILE.
#
# This software is copyrighted 2013 by G. Andrew Mangogna.
# The following terms apply to all files associated with the software unless
# explicitly disclaimed in individual files.
#
# The authors hereby grant permission to use, copy, modify, distribute,
# and license this software and its documentation for any purpose, provided
# that existing copyright notices are retained in all copies and that this
# notice is included verbatim in any distributions. No written agreement,
# license, or royalty fee is required for any of the authorized uses.
# Modifications to this software may be copyrighted by their authors and
# need not follow the licensing terms described here, provided that the
# new terms are clearly indicated on the first page of each file where
# they apply.
#
# IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR
# DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING
# OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES
# THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF
# SUCH DAMAGE.
#
# THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES,
# INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.  THIS SOFTWARE
# IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE
# NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS,
# OR MODIFICATIONS.
#
# GOVERNMENT USE: If you are acquiring this software on behalf of the
# U.S. government, the Government shall have only "Restricted Rights"
# in the software and related documentation as defined in the Federal
```

```
# Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2).  If you
# are acquiring the software on behalf of the Department of Defense,
# the software shall be classified as "Commercial Computer Software"
# and the Government shall have only "Restricted Rights" as defined in
# Clause 252.227-7013 (c) (1) of DFARs.  Notwithstanding the foregoing,
# the authors grant the U.S. Government and others acting in its behalf
# permission to use and distribute the software in accordance with the
# terms specified in this license.

package require Tcl 8.6
package require logger
package require ral
package require ralutil

namespace eval ::elfdecode {
    namespace export elffile
    namespace ensemble create

    logger::initNamespace [namespace current]

    variable version 1.0.1
}

<<helpers>>
<<elffile class>>
<<elffile class data>>
<<elffile class methods>>

package provide elfdecode $::elfdecode::version
```

We also provide a root chunk to extract a package index file.

```
<<pkgIndex.tcl>>=
package ifneeded elfdecode 1.0.1 [list source [file join $dir elfdecode.tcl]]
```

Index

A
Advance, 12

C
classes
    elffile, 1
constructor, 1
Convert, 15
ConvertIdentifier, 15

D
decodeData, 4

E
ElfBitSymbol, 21
ElfEnumerator, 16
ElfEnumRange, 16
elffile, 1
elffile methods
    exported
        constructor, 1
        decodeData, 4
        foreachSectionHeader, 5
        getSectionDataByIndex, 5
        getSectionDataByName, 5
        getSectionHeaderByIndex, 5
        getSectionHeaderByName, 4
        getSymbolByName, 6
        getSymbolTable, 6
        readChan, 4
        readFile, 3
    unexported
        Advance, 12
        Convert, 15
        ConvertIdentifier, 15
        Read, 13
        ReadElfHeader, 6
        ReadProgHeaders, 9
        ReadSecHeaders, 7
        ReadStringTable, 9
        ReadSymTable, 10
        Scan, 16
        Seek, 12
ElfFormatSpec, 13
ElfProgHeader, 3
ElfSecHeader, 2
ElfSymbolTable, 3
exported
    constructor, 1
    decodeData, 4
    foreachSectionHeader, 5
    getSectionDataByIndex, 5
    getSectionDataByName, 5
    getSectionHeaderByIndex, 5
    getSectionHeaderByName, 4
    getSymbolByName, 6
    getSymbolTable, 6
    readChan, 4
    readFile, 3

F
foreachSectionHeader, 5

G
getSectionDataByIndex, 5
getSectionDataByName, 5
getSectionHeaderByIndex, 5
getSectionHeaderByName, 4
getSymbolByName, 6
getSymbolTable, 6

R
Read, 13
readChan, 4
ReadElfHeader, 6
readFile, 3
ReadProgHeaders, 9
ReadSecHeaders, 7
ReadStringTable, 9
ReadSymTable, 10
relvar
    ElfBitSymbol, 21
    ElfEnumerator, 16
    ElfEnumRange, 16
    ElfFormatSpec, 13
    ElfProgHeader, 3
    ElfSecHeader, 2
    ElfSymbolTable, 3

S
Scan, 16
Seek, 12

U
unexported
    Advance, 12
    Convert, 15
    ConvertIdentifier, 15
    Read, 13
    ReadElfHeader, 6
    ReadProgHeaders, 9
    ReadSecHeaders, 7
    ReadStringTable, 9
    ReadSymTable, 10
    Scan, 16
    Seek, 12