ELFIO
Tutorial

Serge Lamikhov-Center
(to_serge@users.sourceforge.net)

April 21, 2002

1 Introduction

ELFIO is a C++ library that permits you to read and generate files in ELF binary format. This library is not based on any other products and is platform independent. It uses only standard ANSI C++ language constructions and supposed to run on wide range of architectures.

While the library's implementation does it's best to make your work easy, a basic knowledge of ELF binary format is needed. You may read about ELF format in TIS (Tool Interface Standards) documentation that you received together with the sources of this library.

2 Getting started with ELFIO

2.1 Initialization

ELFIO library consists of two independent parts: ELF file reader (ELFI) and producer (ELFO). Each is represented by its own set of interfaces. The library doesn't contain any classes that we should explicitly instantiate. Instead, ELFIO provides us a set of interfaces that we should use to access the library functionality.

To make our program recognize all ELFIO interface classes, we need to include ELFIO.h header file. Doing this, we also define all standard definitions from TIS documentation.

```
#include <ELFIO.h>
```

In this chapter, we'll see how to work with the reader part of ELFIO library. The first step that we should do, is to get a pointer onto the ELF file reader:

```
IELFI* pReader;
ELFIO::GetInstance()->CreateELFI( &pReader );
```

Now, when we have a pointer on IELFI interface, we should initialize the object by loading ELF file itself:

```
char* filename = ''file.o'';
pReader->Load( filename );
```

From here, we have an access to an ELF header. We may "ask" such file parameters as encoding, machine type, entry point, etc... Let's get the encoding of our file:

```
unsigned char encoding = pReader->GetEncoding();
```

Please note, standard types and constants from TIS document are defined in ELFTypes.h header file. This file is included automatically into our project. For example ELFDATA2LSB and ELFDATA2MSB constant defines a value for little and big endian encoding respectively.

2.2 ELF file sections

ELF binary file consist from several sections. Each section has it's own responsibility. There are sections that contain executable code, and there are sections that describes you program dependencies, symbol tables and so on... Please see TIS documentation for the full description of all sections.

How can we know, how many sections our ELF file contains? What are their names? Their size? Let's see the next code:

```
int nSecNo = pReader->GetSectionsNum();
for ( int i = 0; i < nSecNo; ++i ) {    // For all sections
    const IELFISection* pSec = pReader->GetSection( i );
    std::cout << pSec->GetName() << '' ''
              << pSec->GetSize() << std::endl;
    pSec->Release();
}
```

First, we have got a number of sections. Next, we have received a pointer on IELFISection interface. Using this interface we may access different section attributes, like size, type, flags. address. To get a buffer, that contains section's bytes, we'll use GetData() member function of this interface. Please see IELFISection declaration for a full description of this interface.

2.3 Section readers

When we have got section data through GetData() function call, we can manipulate this data according to our wish. But there are special sections that provide information in predefined forms and ELFIO library is ready to help us to process such sections. The library provides a set of section readers that "know" predefined formats and how to process data. ELFIO.h header file currently defines the next types of readers:

```
enum ReaderType {
    ELFI_STRING,     // Strings reader
    ELFI_SYMBOL,     // Symbol table reader
    ELFI_RELOCATION, // Relocation table reader
    ELFI_NOTE,       // Notes reader
    ELFI_DYNAMIC,    // Dynamic section reader
    ELFI_HASH        // Hash
};
```

Let's see how we can use symbol table reader in our example.
First, we get symbol section:

```
const IELFISection* pSec = pReader->GetSection( ''.symtab'' );
```

Then, we create symbol section reader:

```
IELFISymbolTable* pSymTbl = 0;
pReader->CreateSectionReader( IELFI::ELFI_SYMBOL,
                              pSec,
                              (void**)&pSymTbl );
```

And finally, we use the section reader to process all entries (print operations omitted):

```
std::string   name;
Elf32_Addr    value;
Elf32_Word    size;
unsigned char bind;
unsigned char type;
Elf32_Half    section;
int nSymNo = pSymTbl->GetSymbolNum();
if ( 0 < nSymNo ) {
    for ( int i = 0; i < nSymNo; ++i ) {
        pSymTbl->GetSymbol( i, name, value, size,
                            bind, type, section );
    }
}
pSymTbl->Release();
pSec->Release();
```

2.4 Finalization

All interfaces that we get from ELFIO library should be freed after their use. Each interface has Release() function. It's not enough to free only high level interface. If one of sections or readers will be held, resources will not be cleared.

While we freed our interfaces immediately after their use, in this example, we should free only pReader object:

```
pReader->Release();
```

3 ELFDump utility

You can find source code of this ELF dumping utility in "Examples" directory. There you will see more examples how to use different ELFIO reader interfaces.

4 IELFO - ELF file producer interface

Now, let's look how ELFIO library can help you to build a very short ELF executable file. This executable file will run on x86 Linux machine and will print "Hello World!" onto your console.

Just as we did when we worked with the reader, our first step is to get a pointer onto the ELF file writer (producer):

```
IELFO* pELFO;
ELFIO::GetInstance()->CreateELFO( &pELFO );
```

Before we will continue, we should inform the library about main attributes of the executable file we are going to build. We should declare that our new executable ELF file will run on 32 bit x86 machine, has little endian encoding, and use current version of ELF file format:

```
// You can't proceed without this function call!
pELFO->SetAttr( ELFCLASS32, ELFDATA2LSB, EV_CURRENT,
                ET_EXEC, EM_386, EV_CURRENT, 0 );
```

Some sections of an ELF executable file should be resided in program segments. To create such loadable segment, we call AddSegment() function.

```
// Create a loadable segment
IELFOSegment* pSegment = pELFO->AddSegment( PT_LOAD,
                                            0x08040000,
                                            0x08040000,
                                            PF_X | PF_R,
                                            0x1000 );
```

This segment serves as a placeholder for our code section. To create code section we call AddSection() function:

```
// Create code section
IELFOSection* pTextSec = pELFO->AddSection( ".text",
                                SHT_PROGBITS,
                                SHF_ALLOC | SHF_EXECINSTR,
                                0,
                                0x10,
                                0 );
```

and add executable code for this section:

```
char text[] =
{ '\xB8', '\x04', '\x00', '\x00', '\x00',   // mov eax, 4
  '\xBB', '\x01', '\x00', '\x00', '\x00',   // mov ebx, 1
  '\xB9', '\xFD', '\x00', '\x04', '\x08',   // mov ecx, msg
  '\xBA', '\x0E', '\x00', '\x00', '\x00',   // mov edx, 14
  '\xCD', '\x80',                           // int 0x80
  '\xB8', '\x01', '\x00', '\x00', '\x00',   // mov eax, 1
  '\xCD', '\x80',                           // int 0x80
  '\x48', '\x65', '\x6C', '\x6C', '\x6F',   // db 'Hello'
  '\x2C', '\x20', '\x57', '\x6F', '\x72',   // db ', Wor'
  '\x6C', '\x64', '\x21', '\x0A'            // db 'ld!', 10
};
pTextSec->SetData( text, sizeof( text ) );
```

Now, put this code section into our loadable segment:

```
// Add code section into program segment
pSegment->AddSection( pTextSec );
pTextSec->Release();
pSegment->Release();
```

The only thing we have to do now is to define start address of our program and create the result file.

```
// Set program entry point
pELFO->SetEntry( 0x08040000 );
// Create ELF file
pELFO->Save( "test.elf" );
pELFO->Release();
```

Please don't forget to call Release() functions for each interface you use. This will free all resources that ELFIO library created for it's work.

When we will compile our program and run it, we will get a new ELF file called "test.elf". The length of this working executable file is only 267 bytes! Try to run it on your Linux box:

```
[serge@home Writer]$ ./Writer
[serge@home Writer]$ chmod +x test.elf
[serge@home Writer]$ ./test.elf
Hello, World!
[serge@home Writer]$
```

Full text for this program you can find in "Writer" directory. In "Examples" directory, you will find two other programs "WriteObj" and "WriteObj2" that demonstrate creation of ELF object files.