The Automated Analysis of Header Files for Support of the Standardization Process

Eugene Novikov
ISP RAS
joker@ispras.ru

Denis Silakov
ISP RAS
silakov@ispras.ru

Abstract

This paper considers the method of the header files automated analysis. The method is intended for the LSB standardization process support. The suggested approach is based on the usage of the cpp preprocessor and gcc compiler high-level representation, their extensions and additional analyzers. The basic work stages of tool implementing the suggested method are introduced. Also tool application for the Qt library header files analysis is considered.

1. Introduction

Now in the world a large number of Linux distributions exists [5]. They are widely applied in various systems. All distributions in their basis have the same components. However, components versions depend on system. Also distribution developers make some specific changes for their systems. Therefore at the binary level Linux distributions are not completely compatible and programs, created for one operating system, sometimes can not be started in another without recompilation from their source code. So application developers, who don't publish source code, should either make many efforts and spend much time for maintenance of compatibility at the binary level or release their products for the limited number of distributions [1]. Because of such complexities there are no fully reliable and valid applications in Linux for some software areas [6].

The similar problem of Unix distributions fragmentation has led to POSIX standard creation [7]. This standard contains requirements for core interfaces (about 1000 functions) and does not describe other interfaces. For example, it does not describe graphical interface functions that are used by the most of modern applications.

Project LSB (Linux Standard Base) [2], [4] is aimed to help developers to create and support portable applications for Linux operating system. In all distributions, supporting the LSB standard, binary files of these applications will be executed equally. In comparison with POSIX, LSB standardizes essentially more interfaces (about 40000 functions). The basis of the whole LSB standardization process is the LSB specification database. From its information it is generated:

- LSB standard text;
- primitive tests;
- environment for LSB-compatible applications development.

To estimate the scale, basic objects of the LSB database and their quantity at the current moment are shown in Table 1.

Table 1. The basic objects of the LSB database

Libraries	~60
Header files	~900
Interfaces	~47000
Types	~16000
Macros	~12000
Classes	~1600

From Table 1 it becomes clear that data volume is large. In addition Linux operating system and applications are constantly developed. So the LSB standard covers more and more libraries. Therefore it is necessary to support the database in up to date state. This is done by updating of stored information and uploading of the new one. Thus today the LSB database population automation problem is very actual.

2. The review of existing methods and tools for LSB database population

Data collection can be performed on the basis of following sources:

- libraries binary files (with debugging information and without it);
- libraries header files.

At the moment the LSB database is populated in general with data obtained from the libraries binary files analysis. Considerable data volume is modified and uploaded manually.

There are tools that automate both binary and header files analysis. libtodb and libtodb2 [8] allow to process binary files. These tools are based on the readelf system utility [11]. For header files processing the tool headertodb [8] based on the ctags program [9] may be used. These tools allow to automate the process substantially but they do not give all needed functionality.

The main advantages of the binary files analysis are possibilities to obtain:

- the list of interfaces exported by library;
- binary symbols versions.

This data is important to provide with compatibility at the binary level support. Disadvantages of the libtodb and libtodb2 analysis are:

- absence of some data at the binary level (e.g. inline functions);
- insufficient C++ support at the binary level;
- absence of interrelations between complex types at the binary level;
- complexity of the large data volumes processing by means of libtodb.

The tool headertodb allows to obtain a considerable part of data from header files but, due to insufficient analysis of the ctags program, it does not provide demanded completeness and quality. The C++ analysis also is not supported at the necessary level.

In this paper the header files analysis method that allows to eliminate the most of disadvantages specified above is described. This method makes analysis and population the LSB database LSB with obtained information essentially automated. Also the tool implementing the suggested method is described. It is necessary to notice that many features of the method and the tool are connected with necessity of their application for Qt library header files processing [12]. These header files are written in C++ language. Previously they were processed mainly by hand. This paper considers practical results of these header files automated analysis.

3. The method of the header files analysis

To obtain data from header files the cpp preprocessor and the gcc compiler parser [11] are used. The suggested approach has such advantages in comparison with the usage of other programs (e.g. ctags mentioned above) and the creation of the own analyzer as:

- Presence of analyzers (cpp and gcc) which are constantly supported by third-party developers. These analyzers allow to obtain data from the high-level representations that posses considerably more simplified and formalized structure in comparison with initial representation of header files (i.e. a standard text). This was the key feature in choosing of the method to be used.
- Maintenance of the most detailed analysis of Linux libraries header files. It is due to very close connection of the cpp preprocessor and the gcc compiler with the given operating system. Header files are processed by them with specific Linux features and different extensions, many of which are made especially for the given preprocessor and compiler. In addition, the third-party analyzers automatically carry out the code correctness check and connect header files with each other.
- The openness of the cpp and gcc source code. This important feature is necessary for more accurate understanding of their high-level representations structure. Also this allows their modification needed to obtain additional data.

Disadvantages of such approach are:

- Absence of the detailed documentation on the used high-level representations. Syntax and semantics of these representations were explored by testing on different examples. It is important to notice that all the necessary, what may be obtained from these representations for the Qt library header files analysis, was studied and implemented. Necessary researches were performed basically directly on header files from the given library and also on more simple examples, including header files written in C language.
- Incompleteness of data obtained from the high-level representations. First of all this comes from the fact that cpp and gcc are intended to compile programs and do not specialize in these representations generation. Therefore there may be no some needed information in the preprocessor and compiler representations. In that case some cpp and gcc source code modifications, extending these high-level representations, and additional analyzers are made. Extensions and additional analyzers will be introduced below.
- Possible preprocessor and compiler representations changes by third-party developers require changes in their analysis. It is necessary to tell that during the time, while the tool processing these representations was developed (about one year), syntax has not changed at all while small changes in semantics for some entities have facilitated their interrelations analysis.

Thus, it becomes clear that disadvantages of the suggested method may be eliminated and in many respects it was already done. Therefore it is possible to enjoy advantages of the approach. Following sections describe this.

3.1. The cpp preprocessor high-level representation structure

For parsing tree generation discussed below the compiler gcc uses header files source code after preprocessing by cpp. So the parsing tree does not contain any information on preprocessor directives. The cpp preprocessor output and the additional analyzer are used for the preprocessor directives analysis. The preprocessor output is used to obtain the lists of:

- macrodefinitions declarations (directive `#define`);
- removals of macrodefinitions declarations (directive `#undef`);
- included header files (directive `#include`).

In this output there are auxiliary strings between which there are directives `#define`, `#undef` and `#include` and preprocessed code of the analyzed header file and included header files. It is important to notice that:

- Comments and non-significant spaces (for example spaces between # and a directive name) are removed from directives.
- Multiline directives are transformed into one-line ones with corresponding additional empty strings.
- Lines numbers in the cpp output and directly in header files coincide with each other. Also each directive is fully placed on its own string. These features are important to order preprocessor directives with other entities from header files.

Here formats of auxiliary strings and directives `#define`, `#undef` and `#include` are described:

- `# a line number in a header file “an absolute path to a header file”`
  - Subsequent strings, up to the next auxiliary string or to the end of file, are content of the analyzed or included to it header file. This content begins with the given line number.
  - Absolute paths uniqueness allows to distinguish header files with identical short names, e.g. time.h and sys/time.h. So this gives the way to order entities from different header files. In addition, absolute paths to included header files are remained for their unique identification.
- `# 1 “<built-in>”`
  - Following strings contain system macrodefinitions that are declared by the preprocessor implicitly.
  - This list may be obtained by means of the headertodb3 tool described below.
- `# 1 “<command-line>”`
  - Following strings contain macrodefinitions that are declared by the user in the cpp preprocessor command-line.
  - User macrodefinitions may be declared by means of the headertodb3 special option. In particular, it is possible to redefine system macrodefinitions.
- `#define a macrodefinition name a macrodefinition value`
  - Macrodefinition declaration.
  - A macrodefinition value is arbitrary string, in particular it may be absent.
  - Functional macrodefinitions are declared in form: `a functional macrodefinition name (parameters names divided by commas) a functional macrodefinition value`.
- `#undef a macrodefinition name`
  - Macrodefinition declaration removal.
- `#include “a header file name”` or `#include <a header file name>`
  - This means that a currently processed header file includes another one. Included files are searched according to the standard preprocessor rules. Therefore it is important to specify correct paths to directories that will be explored for included headers.
  - Following two strings contain absolute paths to including and to included header files.

3.2. The gcc compiler high-level representation structure

Preprocessed header files are processed by means of the gcc compiler parser. This parser allows to obtain the parsing tree. The parsing tree is the internal compiler structure that is used to represent information on source code. The tree is generated by gcc after lexical, syntactic and semantic analyses. Also the compiler may write the parsing tree in the text representation. The parsing tree text representation structure is not well documented. Research of this structure on examples allowed to make its description.

The parsing tree consists of nodes and their attributes. Nodes correspond to entities. Attributes describe their kinds and properties. In the text representation all nodes are written at the beginning of strings in the form `@integer`, e.g. `@475`. After a node name a list of attributes corresponding to this node is written. The first attribute specifies an entity kind. For example, to represent the integer type the first attribute `integer_type` is used. The first attribute has a name and has no value. Below various entities kinds will be characterized directly by the first attribute.

Subsequent attributes describe entity properties. In the text representation they have the following form `an attribute name: an attribute value`, e.g. `name: @249`. An attribute value may be one of the following kinds:

- a reference to another tree node (e.g. `@12`);
- some text information (e.g. `long double`);
- a place in a header file where an entity is declared (e.g. `/usr/include/time.h:412:12`).

A reference to another tree node specifies that some entity property is described by means of another entity represented by another node. For example, the function declarations `function_decl` may have attribute `name: @11` that refers to the identifier entity `identifier_node`. In turn the identifier has the attribute with the function name value. The function name is written in the usual text form and belongs to the second kind of attributes. An entity declaration place is described in the following form `an absolute path to a header file: a line number in this file: a column number in this file`. Such form is the gcc compiler extensions. These extensions were made since the original declaration place form looked like `a header file name: a line number in this file`. It appeared that it is not enough both to uniquely identify header files (for example, entities from header files time.h and sys/time.h were concerned as belonging to the same header file time.h) and to order entities placed at the same string. Thanks to compilers extensions these disadvantages were eliminated. To designate system entities `<built-in>` is used as a header file name. System entities are, for example, `intrinsic` types such as `void`, `int`, `bool`, etc. Thus attributes values belong to one of three classes each of which is processed in the appropriate way. Some attributes may have identical names, e.g. `note` and `spec`. For such attributes all their values are obtained as values vectors. Despite attributes names coincidence for some entities all properties are unambiguously and consistently defined by a pair `an attribute name, an attribute value`.

Entities, important to obtain the necessary information from header files, belong to one of the following classes:

- declarations;
- types;
- constants;
- auxiliary entities.

The complete description of all entities structure and the method of their analysis is huge and is not considered within the bounds of this paper. Nevertheless the description of four representatives for each class is given below. For such description the analysis of entities attributes and properties that may be obtained is done.

3.2.1. Function declaration. The first attribute of this entity is `function_decl`. Subsequent attributes names and values are:

- `name` is a reference to an identifier `identifier_node`, a function name.
- `type` is a reference to a functional type `function_type`, a complete description of a function signature.
- `scpe` is a reference to a function scope, whether a namespace declaration `namespace_decl` or a parent class `record_type`.
- `srcp` is a function definition place in a header file.
- `note` (optional and multiple-valued) is `member` (for a class method) or `constructor` (a class constructor) or `destructor` (a class destructor) or `operator an operator name`.
- `accs` (optional) is access to a class method. It may be `pub`, `priv` or `prot` that accordingly designates public, private and protected.
- `spec` (optional) is a class method specifier. It may be `virt` or `pure` that is designations for virtual and pure virtual accordingly.
- `args` is a reference to `parm_decl`, the first element of a function arguments list containing their types, names and qualifiers.

To obtain additional needed information on a function declaration the following extended attributes are added:

- `ext_qual` (optional) is a class method qualifier. It may be `const` or `volatile`.
- `ext_note` (optional) is `explicit` (an explicit class constructor), `extern` or `static` (a function specifier), `inline` (an inline function), `throw` (a throw function).
- `ext_body` (optional) is a reference to the beginning of a tree expression corresponding to a function body.
- `ext_body_open_brace` (optional) is an opening brace place, the beginning of a function body.
- `ext_body_close_brace` (optional) is a closing brace place, the end of a function body.

According to attributes following information on a function declaration is obtained:

- A function name or an overloaded operator name.
- Return value type.

- For function arguments their types, names, default values and qualifiers are obtained. If a function arguments list has not variable length, last parameter is always `intrinsic` type `void`.
- A function parent, a namespace or a class.
- Whether a function is extern or static, inline, throw, virtual.
- For methods access to them, whether a method is const or volatile, whether a method is a constructor or a destructor are obtained. For constructors explicit property is obtained.
- An inline function body place in a header file. It is necessary to take corresponding block directly from a header file without expression parsing.

3.2.2. One-dimensional array type. The first attribute of this entity is `array_type`. Subsequent attributes names and values are:

- `elts` is a reference to an arbitrary type, a type of one-dimensional array elements.
- `domn` (optional) is a reference to an integer constant `integer_cst`, the number of one-dimensional array elements.

It is important to notice that multidimensional arrays are formed as a one-dimensional arrays sequence. A base types sequence is 'natural' i.e. from the fullest type to the basic one. Therefore processing of multidimensional arrays is recursive. In case of empty array usage (for example, `int []`) `array_type` has not attribute `domn`, containing the number of elements. Also it is necessary to pay attention, that in one-dimensional (multidimensional) array passing to function as argument, it is transformed to a pointer (an array of pointers). For example, `int [10][20][30]` becomes `int * [20][30]`.

3.2.3. Integer constant. The first attribute of this entity is `integer_cst`. Subsequent attributes names and values are:

- `type` is a reference to an integer type `integer_type` or to an enumeration type `enumeral_type` that is an integer constant type.
- `low` is some arbitrary integer number, an integer constant value.

Integer constants are enumerations elements values, arrays and bitfields sizes, global variables default values and function arguments or templates parameters default values. In the case when used integer constant belongs to some enumeration type, its name must be used instead of corresponding numerical value.

3.2.4. One-linked list element. The first attribute of this entity is `tree_list`. Subsequent attributes names and values are:

- `purp`:
  - for an enumeration type `enumeral_type` is a reference to an identifier `identifier_node`, an enumeration element name;
  - for a functional type `function_type` (optional) is a reference to an arbitrary constant, a function argument default value;
  - for a template declaration `template_decl`, an attribute `inst` value is a reference to `tree_vec`, an attribute `prms` value is a reference to `integer_cst`.
- `valu`:
  - for an enumeration type `enumeral_type` is a reference to an integer constant `integer_cst`, an enumeration element value;
  - for a functional type `function_type` is a reference to an arbitrary type, a function argument type;
  - for a template declaration `template_decl`, an attribute `inst` value is a reference to `record_type` (a template instance), an attribute `prms` value is a reference to `tree_vec`;
- `chan` (optional) is a reference to `tree_list`, a following element of an one-linked list.

One-linked lists are processed depending on a context in which they are used.

3.3. The additional analyzers

There are not preprocessor conditional compilation directives in the cpp output. Therefore they are analyzed directly through header files. The following conditional compilation directives groups are processed:

- *Conditions* are preprocessor directives specifying the beginning of conditional compilation. `#if`, `#ifdef` and `#ifndef` belong to this group. Conditional expressions must be for these directives. A conditional expression follows a directive name.
- *Branches* are instructions about possible variants in execution. Directives `#else` and `#elif` belong to this group. Conditional expression must be for directive `#elif`.
- *The end of conditional compilation* is expressed by means of directive `#endif`.

During the conditional compilation directives analysis non-significant spaces and multiline strings are processed. Such analyzer does not possess the cpp preprocessor completeness. Therefore in future

some corresponding preprocessor output extensions will be probably made.

Besides, auxiliary comments, having the special form, are processed:

- *LSB parameters* are instructions about how included header files must be analyzed. These comments are placed at the beginning of header files from the beginning of lines. They have such form `/* LSB PARAM: a parameter name */`. At the moment it is necessary to process the following *LSB parameters*:
  - *fresh* means that entities from an included header file should be processed as well as entities of an analyzed header file.
  - *print* means that *LSB IDs* discussed below are printed for entities from included header file.
  - *end* means the end of a *LSB parameters* section.
- *LSB IDs* are unique integers which are associated with entities from included header files. These integers are obtained during the corresponding header files analysis and kept into LSB database. Comments are placed on strings before corresponding entities. They have such form `/* LSB ID: an integer number */`. Then *LSB IDs* are used to refer on already processed entities.

Special comments are written into header files mainly automatically. This is done for files that were already analyzed and generated on the basis of the LSB database. These comments may be located in header files of any acceptable inclusion depth. So they are analyzed for all these levels. The latter is possible because of the preprocessor output has all included header files content. In the future new special comments may occur to provide additional needed functionality.

3.4. The high-level representations extensions

To obtain some additional information a number of gcc parser extensions was made. Some extensions were already mentioned above. At the moment the full extensions list is the following:

- More detailed entities definitions place in header files.
- Distinction between structure and class types joined by the compiler together into the general internal type `record_type`.
- Attributes correct form i.e. `an attribute name: an attribute value` (e.g. attribute `bitfield` for bitfields).
- Access to type declarations scoped in a class.
- Function declarations properties (inline, throw, const, volatile, etc.).
- Function bodies trees included in the general parsing tree.
- Function bodies in the form of their places in header files.
- Exception types lists.
- Escape-sequences in string constants.
- Function definitions and type declarations prototypes places.
- Absence of the unnecessary precompiled header files generation.
- Absence of information on the unnecessary gcc system functions in the tree text representation.

Extensions are made on the basis of the gcc compiler source code analysis. Usually they do not demand any considerable changes and do not affect the gcc work. Extensions supplement the compiler high-level representation and give more detailed set of entities properties. It allows to avoid the difficult manual analysis. So it automates populating of LSB database with data in very considerable degree. Some extensions optimize the compiler analyzer work. Therefore in future the extensions list will be increased to perform more and more detailed and qualitative analysis as fast as possible.

4. The headertodb3 tool

The tool called headertodb3 was developed to implement the high-level representations analysis. Tool input is libraries header files. A great number of various options allow to control the headertodb3 analysis and output. Also many settings are available through the special configuration file. Headertodb3 works automatically after options specifying. If corresponding options are enabled then headertodb3 informs about its actions and displays additional debugging information. In case of some critical error occurrence (e.g. error in cpp or gcc work) the tool finishes work with the corresponding return value. During the work the tool performs the following steps:

- Command-line options processing. Depending on these options the tool performs different analysis kinds. Below the headertodb3 standard work scheme is presented.
- Initializations that headertodb3 uses in the work. The main is the primitives initialization. Primitives consist of `intrinsic` types, C and C++ keywords and some auxiliary information. Primitives are used during the parsing tree analysis to connect tree conceptions with the language ones.
- The preprocessor directives analysis on the basis of the cpp output.

- The special comments analysis on the basis of the cpp output with comments. It is made by means of the additional analyzer.
- The conditional compilation directives analysis by means of the additional analyzer.
- The gcc compiler parsing tree generation. Its conversion to the tool internal representation that will be used in the further parsing tree analysis.
- The parsing tree analysis. This is the main stage during the tool work. The general work scheme at this stage is the following:
  - entities ordering in that sequence in which they are encountered in header files;
  - the ordered entities analysis by means of the special handlers.
- Temporary folders and files removal.

During its work the tool generates the following files into the special folders:

- The text file with information on preprocessor directives. This information is used by other tools to obtain inclusion interrelations between header files. It is needed to construct a sequence in which header files will be analyzed by headertodb3. By means of the tool special option this analysis may be standalone.
- The major SQL-script with information on all entities, their properties and interrelations. The LSB database may be directly populated with this script. It is necessary to notice that some additional tables are used in the database to store information from C++ header files. Then other tools use information from the LSB database to check different dependencies and to generate header files and corresponding to them HTML pages.
- The text file with information on function bodies places in header files. This may be used for the following functions bodies processing.
- The text file containing information on errors that the tool faced during the work. Information on failures kinds and failures locations is printed here.
- In addition by means of the tool special options it is possible to obtain text and HTML representation of the gcc parsing tree. The parsing tree in the HTML page form is convenient for navigation between nodes. Therefore everybody can walk quickly through the parsing tree by means of a usual browser.

The tool was developed by degrees. The tool first version is headertodb2. It is intended to process header files written in C language. During the headertodb3 creation the corresponding experience was taken into account and many innovations were brought. Tools were tested on real header files from C and C++ libraries. Also the test scenarios were made on the basis of different real situations and arisen errors. They consists of about 1000 various test cases that cover C and C++ languages and the compiler extensions. All these test cases may be passed automatically. It is very important to immediately trace malfunctions that may occur during tools development. Thanks to the automated testing system both headertodb3 and other tools errors were found out.

At the moment the most important check of headertodb3 tool is its application in Qt4 library [12] header files processing. By means of the tool information shown in Table 2 was obtained.

Table 2. Headertodb3 application in the Qt4 library header files analysis.

	libQtCore	**libQtGui**
Header files	85	183
Interfaces and their parameters	4450, 4520	9720, 10360
Classes, structures, unions and their fields	440, 385	990, 610
Enumerations and their constants	160, 1975	320, 2180
Templates, their parameters, instances and specializations	240, 340, 130, 80	155, 155, 150, 0
Type declarations by means of typedef	290	240
Macrodefinitions	290	190
Included header files	190	460
Entities including auxiliary ones	22760	40310
Properties and interrelations of entities	87830	161030

Also headertodb3 was applied in other Qt4 header files processing and in Qt3 library processing.

5. Conclusion

At the moment the header files analysis needed for populating the LSB database with data is one of the most important stages of the LSB standardization process. Initially header files written in C++ language were processed either manually or with the usage of the analyzers which do not correspond to all demands. As a result huge human resources were required to solve this problem.

This paper considers the method that allows to automate the C++ header files analysis. The approach is based on the usage of the cpp preprocessor and gcc compiler high-level representations. It allows to use analyzers for header files from third-party developers and to process and extend the more strictly formalized and simpler high-level representations. The suggested method was implemented in the headertodb3 tool.

The tool was used to populate the LSB database with data within the Qt library standardization process. On the basis of tool application results it is possible to make the following general conclusions:

- The method allows to analyze header files with high quality.
- The approach provides required C++ support.
- The tool does not demand considerable computing resources and time expenses.
- The tool makes the header files analysis substantially automated.

In the future additional high-level representations extensions are supposed. It will allow to analyze header files in the more qualitative and automated way.

References

[1]. A.I.Grinevich, D.A.Markovcev, V.V.Rubanov. Linux systems compatibility problems. The Institute for System Programming proceedings, the 10th volume: “Linux systems reliability and compatibility ensuring”. In Russian.

[2]. A.V.Horoshilov. Linux Standard Base: success history? The Institute for System Programming proceedings, the 10th volume: “Linux systems reliability and compatibility ensuring”. In Russian.

[3]. D.V.Silakov. Current state and perspectives of the LSB infrastructure development. The Institute for System Programming proceedings, the 13th volume, the 1st part. In Russian.

[4]. Linux Standard Base. http://www.linux-foundation.org/en/LSB

[5]. The Linux distributions list. http://www.lwn.net/Distributions

[6]. D.Shurupov. The programs lack as a barrier to the Linux popularization. In Russian. http://www.nixp.ru/articles/plugging_linux_holesD.Shurupov

[7]. IEEE POSIX® Certification Authority. http://standards.ieee.org/regauth/posix

[8]. The LSB Infrastructure Project. http://ispras.linux-foundation.org

[9]. The ctags program. http://www.ctags.sourceforge.net

[10]. The cpp preprocessor and the gcc compiler. http://www.gcc.gnu.org

[11]. The readelf system utility. http://www.opensourcemanuals.org/manual/readelf

[12]. The Qt library. http://www.qtsoftware.com