# Developer Guide

The Header Importer Architecture

Mohammad R. Basirati

September 14, 2015

# Contents

# 1 Introduction

This document gives a complete description about how the tool header importer works. Furthermore it describes how to install needed tools for extending the importer. It assumes that you are familiar with lexical scanner, lexical parser and Jetbrains MPS[1].

Header Importer is a tool to import declarations from C headers into an mbeddr[2] project, so developers can use them in their code.

The process of importing a header file into an mbeddr project goes in three steps: 1. Scanning header files 2. Parsing scanner tokens 3. Importing declarations into mbeddr external module structure. We will discuss each step in detail in the following sections.

Step one and two work along in a Java project to prepare the required input for step three. The header importer tool Java project is located in folder bparser. The scanner generator produces lexer.java file and the parser generator produces two files: sym.java and parser.java. Lexer.java will tokenize the input file. Parser.java file can recognize the tokens by their identifier which has been defined in sym.java file. The last part of the Java project consists of classes which will be used to keep header file declarations. At the last stage, MPS importer gets the declarations and imports them into an mbeddr external module.

This document will explain each step of importing seperately: the scanner, the parser, and the MPS project. Then it will describe the details of every type of declaration in a C header. For every type of declaration, the document gives a brief definition, the parsing process, the importing process of parsed declarations into an mbeddr file, and finally the open issues. At last, there is a list of all open issues of the importer.

# 2 Scanner

The first stage is to tokenize the header file. For this task the importer uses JFlex[3] scanner. You can find the scanner files for the header importer inside the `"scanner_parser"` folder. For compatibility issue of our tool over different versions of header files, we enhanced the scanner to be able to tokenize the gcc stdio.

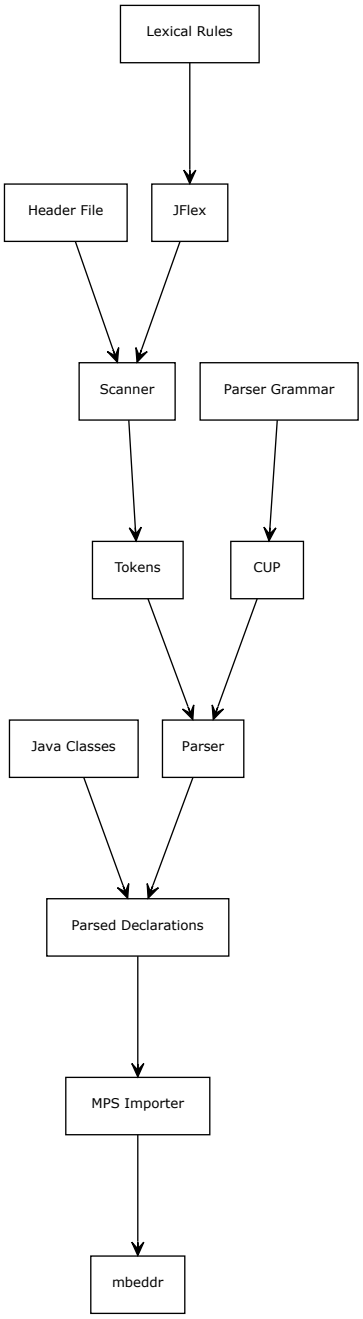
## 2.1 Installing JFlex Scanner

All installation guides provided here are from JFlex website installing guide[4]. After downloading JFlex proceed as follows:

*Windows:*

[1] More information about MPS on https://www.jetbrains.com/mps/
[2] More information about mbeddr on http://mbeddr.com/
[3] JFlex Web Site http://jflex.de/
[4] http://jflex.de/installing.html

Figure 1: Importing Process



- Unzip the file you downloaded into the directory you want JFlex in.
- Edit the file bin\jflex.bat (e.g. ..\jflex-1.6.1\bin\jflex.bat) such that

  JAVA_HOME contains the directory where your Java JDK is installed (for instance C:\java)

  and

  JFLEX_HOME the directory that contains JFlex (in the example: C:\jflex-1.6.1)
- Include the bin\directory of JFlex in your path(the one that contains the start script, ..\jflex-1.6.1\bin).

*Mac/Unix:* Using 'apt-get install jflex' should work OK, but in any case it didn't work, you can install it as follows:

- Decompress the archive into a directory of your choice with GNU tar, for instance to /usr/share:

  tar -C /usr/share -xvzf jflex-1.6.1.tar.gz
- Make a symbolic link from somewhere in your binary path to bin/jflex, for instance:

  ln -s /usr/share/jflex-1.6.1/bin/jflex /usr/bin/jflex

  If the Java interpreter is not in your binary path, you need to supply its location in the script bin/jflex.

After installation you could be enable to run JFlex. The simplest way to run JFlex without any option is:
jflex THE_SCANNER_FILE.jflex

## 2.2 The Scanner Structure

The scanner has three states which can identify one line comments, multiple lines comments, and strings. All the charachter sequences tokens that the scanner returns consists of: all possible operators in c, define, undef, if, ifdef, ifndef, else, endif, include, extern, pragma, typedef, struct, all c types (int, char, etc.), numbers , and identifiers.

The scanner can recognize the compiler preprocessing words which begin with double underscore and returns the token COMPWORD for them. Other compiler preprocessing phrases will be ignored by the scanner.

The lines that begin with define keyword for declaring macros and constants, scanner returns the whole line to the parser and doesn't tokenize their expression. In the parser section we will discuss this issue in detail.

# 3 Parser

The parser has been generated by CUP[5]. JFlex scanner is designed to work along with CUP. Basically the parser gets tokens from the scanner and matches them by the grammers which are defined in .cup file. The parser files of header importer located in `"scanner_parser"` folder.

## 3.1 CUP Parser Installation

All installation guides provided here are from CUP website installing guide[6]. After downloading CUP proceed as follows:

CUP comes with a generator part and a runtime part. Working with CUP comprises two phases:

Parser generation Parser generation is done by creating a .cup file, that represents a parser specification. These specifications need to be processed by the parser generator. For this end, you should place java-cup-11a.jar besides the your_parser.cup file and run this command:

java -classpath java-cup-11a.jar java_cup.Main ¡ your_parser.cup

This will create the corresponding parser as a .java file, that can be integrated into Java projects as usual.

Parser runtime During parser runtime, the generated parser needs a few classes, on which it is dependent. These classes are contained in the java-cup-11b-runtime.jar file. This Java package needs to be included into the target project's classpath. The original generator package java-cup-11b.jar however does not need to be distributed alongside the target project.

## 3.2 The Parser Structure

The grammer of the parser consists of two main parts: preprocessing steps and general declarations. Preprocessing steps comprise of the define, if, ifdef, ifndef, else, endif, include, and COMPWORD token that is the token returned by scanner for preprocessing words used by compilers.

**program ::= program code_part |code_part;**

**code_part ::= pre_process_step |general_declaration SEMI |EXTERN general_declaration SEMI |error;**

General declaration part, consists of 4 type of declaration: typedef declaration, struct declaration, variable declaration, and function prototype declaration.

[5]CUP Web Site http://www2.cs.tum.edu/projects/cup/

[6]http://www2.cs.tum.edu/projects/cup/install.php

### 3.3 Declaration Classes

The IDEA Java project[7] has several classes for managing and keeping the declarations. These classes are used by parser to define declarations and used by MPS importer part to get the declarations and import them into an mbeddr module. Descriptions for each class and how they colaborate in the parsing phase are described blow:

**CodeGenerator** CodeGenerator class is responsible for being a proxy between the parser and MPS part. The parser export parsed declarations by calling CodeGenerator methods.On the other hand, the MPS part gets the declarations from CodeGenerator list of all declarations. Besides, CodeGenerator class has the responsibility of taking care of if blocks and structs.

**Declaration** Declaration is an abstract class which all other classes execpt for CodeGenerator, extend it.

**ConditionalBlock** This class is used for defining conditional directives(e.g. #ifdef) blocks. It has two lists of declarations: true block and false block.

**Define** It is used for define declarations(constants and macros).

**Function** It keeps the information for function declarations and function pointers. It has a list of parameters, a return type, and a name.

**Variable** It defines a variable declaration.

**Struct** It has a list of struct declarations and a name.

**Typedef** It defines a typedef declaration.

**Include** It holds a name for what should be included.

## 4 MPS Importer Project

### 4.1 Installing MPS

For installing MPS you just need to download it from MPS website download page [8] and unpack it.

### 4.2 The MPS Part Structure

The MPS project part has five main classes:

**ParserAdapter** ParserAdapter initiates a new java Parser class and parses the given file address. It returns the CodeGenerator class as the result which has all the parsed declarations.

[7] Intelij IDEA Web Site https://www.jetbrains.com/idea/
[8] https://www.jetbrains.com/mps/download/index.html

Importer The importer class coordinates all the process of importing a parsed header file into the mbeddr like creating a new SupportVariability file, adding needed references to the modules, etc.

ImporterCore The ImporterCore class is responsible to add declarations based on their types(struct, function prototype, etc.) to an mbeddr module.

VariabilityImporter The VariabilityImporter class provides all needed functions to handle conditional directives in the variability mechanism of the mbeddr.

Tyeper The Typer class is responsible for managing the types in mbeddr. It creates new user defined types(e.g. structs, typedefs, etc.) and keeps the track of existing types in an mbeddr module.

# 5 Integrating Scanner, Parser, and MPS Project

Generally the generated scanner and parser java files should be included in a java project. The importer has these files in bparser IDEA java project to work with each other. There are three files: lexer.java, parser.java, and sym.java. Besides the bparser project consists of other needed java classes for handling the import process. Then the bparser .jar file should be included in the MPS project.

# 6 Structs and Unions

## 6.1 Background

Header files can have definitions of structs and unions. Structures and unions are defined exactly with the same syntax. The difference between a struct and a union is the way C stores them. A struct is a record structure and each member in the struct is allocated a new space, while for a union only one chunk of memory is allocated for all its members. Therefore writing to a struct member changes only the value of the member written to, while writing to a member of an union will render the value of all other members invalid.

## 6.2 Overview on Importer Implementation

Since struct and union definition has exactly the same syntax, so the way importer treats them is exactly the same and both are kept in a same java class.

## 6.3 Parsing Structs and Unions

The scanner identifies the 'struct' and 'union' tokens and returns their corresponding tokens to the parser. Struct and union definition is under general declaration part of the parser. Every struct and union has an id and a body of

declarations. The body part can have variable declarations and preprocessing steps(e.g. conditional directives).

Structs and unions are both kept in a java class called 'Struct'. The struct class keeps a list of declarations of a struct or an union memebrs, an id, and a boolean attribute 'isUnion' which defines whetere class belongs to a struct or an union.

### 6.4 Importing Parsed Structs and Unions to mbeddr

The process of importing structs and unions to mbeddr is straight forward, as mbeddr has these concepts completely.

## 7 Typedefs

### 7.1 Background

Typedefs are used to declare new names for existing types. These typedef names are aliases for existing types, and are not declarations of new types and cannot change the meaning of an existing type.

### 7.2 Overview on Importer Implementation

The hard task in importing a typedef is at parsing phase as the typedefs can have very complex expressions. The rest of the process is straight forward, a java class is responsible for keeping the typedef information: a typedef-name and a existing type. Right now the importer can import typedefs with only one pair of typedef-name and existing type. The existing type can be any basic types, pointers, structs, unions, and enums. If there is a more complicated type definition, the importer will fail to import it.

### 7.3 Parsing Typedefs

The scanner identifies typedef keyword and returns a token to the parser. The parser count a typedef as its general declarations. A typedef expression is expected to be a TYPEDEF token followin a general variable declaration, a struct definition or type, or an enum declaration.

The information of a typedef is saved in a java class called 'Typedef'. A boolean attribute called 'isStruct' defines whetere a typedef is defining a new struct or not.

### 7.4 Importing Parsed Typedefs to mbeddr

The process of importing typedefs to mbeddr is straight forward, as mbeddr has this concept completely.

### 7.5 Open Issues

- *Parsing Typedef Expressions:* The importer now lacks parsing all possible typedef expressions and the parser part needs to be improved in future.

## 8 Variable Declarations

### 8.1 Background

Every variable in C has a unique name and a type. A variable is a name given to a some location in memory that holds a value which the program can manipulate it.

### 8.2 Overview on Importer Implementation

The importer can recognize all basic variable types: char, int, float, double, short, short int, long, long int, long long, long long int, long double. Moreover the importer can recognize all possible more complicated types: pointers, function pointers,signed and unsigned types, and arrays. The user defined types like structs, unions and typedefs are supported by the importer too.

### 8.3 Parsing Variables Declarations

The scanner recognizes all basic variabe types and returns a symbol for each of them. The importer can handle pointers, signed and unsigned types, and arrays. A char sequence can be recognized by the parser as a user defined type.

### 8.4 Importing Parsed Variable Declarations to mbeddr

The importer has a class called 'Typer' which crates new types based on parsed types. Besides this class is responsible for recognizing predefined types in a header file. For example if there is a typedef, the Typer class could be able to identify the typedef's name as a new type, but there are some situations which the importer cannot import a typedef because of inability to parse it completely. In those situations, in the rest of the header file, the Typer class will not recognize the new type name defined by the typedef, so the importer will add a new typedef with the corresponding type name and put *void** as the type.

In general the importer will import all the variable types and if there is a type which is a user defined type, but the importer cannot recognize it, the importer adds a new typedef with that type's name and put *void** as the type.

# 9 Function Prototypes

## 9.1 Background

A function prototype declaration specifies name of the function, the return type and type of the arguments, but it lacks the function body. In a function prototype, arguments' names are optional.

## 9.2 Overview on Importer Implementation

The importer identifies function prototypes and import them to mbeddr. The types used in a function prototype are imported into mbeddr with the same mechanism of variable declaration types.

## 9.3 Parsing Function Prototypes

The scanner identifies different parts of a function prototype seperately and returns its corresponding token to the parser. Function prototypes are considered as the general declarations of the parser. Every function prototype has a type, aname and a parameters part which is surrounded by paranthesis. The parameters part are only types seperated by commas.

## 9.4 Importing Parsed Function Prototypes to mbeddr

If there is a type in defining a function prototype which is missing for any reason, the imported adds a new typedef with the name of that type and *void** as the type. The mechanism of importing types is exactly the same as importing variable declarations.

In mbeddr is necessary that arguments have names, so the importer adds a name for them sequentially(e.g. p0, p1, p2, ...).

## 9.5 Open Issues

- *Parsing Arguments' Names:* Right now the importer's parser cannot parse the function prototypes which their arguments have names as an option. The parser should be improved in future works.

# 10 Constants and Macros

## 10.1 Background

Constants and macros are defined by #define directive. By defining a constant, whenever the identifier name of the constant shows up in the file, it will be replaced by its corresponding value. Macro is the association of an identifier or parameterized identifier with a token string. After the macro is defined, the compiler can substitute the token string for each occurrence of the identifier in the source file.

### 10.2 Overview on Importer Implementation

A #define statement consists of two part: 1. an identifier which can be parametrized(in case of macro) 2. a value. In mbeddr mechanism, the correct declaration of the identifier part is mandatory for using it in an 'Implementation Module', but it's not necessary to declare the corresponding value part, as it will be linked from the original C header file later. Therfore importing constants and macros doesn't need importing of the exact value's expression. The importer parses the identifier part and import it as a constant or macro to mbeddr and leave the value part as an empty string.

### 10.3 Parsing #define Directives

JFlex scanner identifies #define directive and it returns the 'DEFINE' symbole to the parser and the exact text of its line.

The #define directive is parsed as a preprocess step. The parser creates a new instance of a java class called 'Define'. Define class has a function 'declare-Define' which extract the identifier part of the #define directive and saves it in the class' attributes.

### 10.4 Importing Parsed #define Directives to mbeddr

The importer defines a macro or a constant(in mbeddr #constant or #alias) with its corresponding identifier. If the value of a constant is integer, it will be imported, but if it's an expression or a non-integer(e.g. string), it will be imported as a string. Mbeddr will use the original C header file at the build time, so importing the value part is not necessary.

### 10.5 Open Issues

- *Importing Correct Values:* Importing the correct value parts of constants and macros can be a future work.

## 11 Preprocessor Conditional Directives

### 11.1 Background

A conditional directive instructs the preprocessor to select whether or not to include a chunk of code in the final token stream passed to the compiler. Preprocessor conditionals can test arithmetic expressions, or whether a name is defined as a macro, or both simultaneously using the special defined operator. Conditional directives are: #ifdef, #ifndef, #if, #else, #elif, #endif.

### 11.2 Overview on Importer Implementation

Importing conditional direcrtives is one of the hardest task in importing a header file. Mbeddr doesn't support conditional directives, so the implementation of

importer used mbeddr's abilities in a creative way. Conditional directive are transalted to 'Feature's in mbeddr and each import makes a unique 'VariabilitySupport' file that holds all conditional directives information as mbeddr features. Then each line of declarations in External Module file of mbeddr, has a 'Presence Condition' which defines this declaration is under which feature, i.e. conditional directive, of previously automatically created VariabilitySupport file.

## 11.3 Parsing Conditional Directives

JFlex scanner identifies #if, #ifdef, and #ifndef directives by a JFlex expression which ignores the whole rest of line front of them and returns a token IF, IFDEF, or IFNDEF to parser with the corresponding exact text of their line. For #else and #endif tokens, scanner returns ELSE and ENDIF respectively by identifying the exact text 'else' and 'endif'. The sharp sign # itself is recognized as a different symbole seperately.

All conditional directives are parsed as a preprocess step. IF, IFDEF, and IFNDEF tokens initiates a new class in java called 'ConditionalBlock' which handles the information of a conditional directive. ConditionalBlock keeps a list of all decelaration under it and its corresponding else block. Each ConditionalBlock has a boolean attribute called 'condition' which identifies that we are on if block or else.

Conditional directives can have complex arithmetic and logical expressions, so the parsing phase could get problematic. Since the importer's implementation mechanism of conditional directives in mbeddr doesn't treat the arithmetic expressions as they are, instead as a string which is a name, it is unnecessary for importer to parse and import all those expressions exactly. A conditional directive's expression is kept in ConditionalBlock class's id attribue which later will be the name of the feature(will be discussed later) added in mbeddr project. This id is exactly the normalized version of the conditional directive line in the original header file. This normalization is done due to naming rules in mbeddr. For example a line is normalized by replacing '!' by 'NOT and '&&' by 'AND'.

## 11.4 Importing Parsed Conditional Directives to mbeddr

Mbeddr doesn't support conditional directives, so the whole idea has been translated to a new available concept of mbeddr feature model. Like conditional directives, every feature of a feature model instructs the builder at the build time to select whether or not to include a chunk of code. Feature models are held in VariabilitySupport files. A VariabilitySupport file consists of two parts: 1. feature declaration 2. configuration model declaration.

Features are declared with their names in a feature-sub-feature hierarchy; that means a feature can be declared as a child of another feature. Conditional directives are imported to mbeddr as features. Nesting of conditional directives is supported by declaring a feature as a child of another feature.

The link between a line of code and a feature is connected with a presence condition. Presence conditions are logical expressions based on features that define when a piece of code is included at build time. Every declaration in a C header that is under a conditional directive is imported in mbeddr with a presence condition that defines based on which feature, i.e. conditional directive, the declaration should be included or not.

Mbeddr features don't hold any logical expression, so expression of a conditional directive is imported as the name of its corresponding feature. There is no way that mbeddr can check whether the condition(conditional directive's logical expression) holds or not, so this task goes to the developer side by defining a Configuration Model.

A configuration model is used to define single configurations by selecting a subset of features from a feature model. A developer should define a configuration model to identify which features, i.e. conditional directives, are hold and so on which declarations should be included at build time. As a start, the importer automatically defines a configuration model for each import that consists of all features, i.e. all conditional directives are held for the true block.

## 11.5 Open Issues

There are still some problems in importing conditional directives due to the mbeddr limitations.

- *Duplicated Feature Name:* In mbeddr features' names must be unique, but there are conditions which the importer adds some features with the same name. It happens when a C header file has a conditional directive which is repeated in different parts. For example the following code will lead to a duplicated feature name error:

  #ifndef `_USE_FILE_OFFSET64`

  # ifdef **REDIRECT**

  # endif

  #endif

  #if !defined `_USE_GNU`

  # ifdef **REDIRECT**

  # endif

  #endif

- *Defining an Appropriate Configuration Model:* Right now the importer makes a configuration model which holds all the features, but not necessarily a legitimate subset. Selecting a correct subset of features, i.e. chunck of a C headr's declarations, is current problem of the importer.

- *Importing #elif Directive:* The importer doesn't support importing the #elif directive right now.

# 12 List of All Open Issues

- *Importing Pragmas:* Right now, the importer doesn't support importing pragmas(The pragma lines are ignored by the scanner).
- *Managing #include Directives:* A C header can have several #include directives which import other headers to use declarations of other types, structs, and so on. Right now the importer ignores the #include directives. An idea could be to import other header used in #include directives .
- *Importing Type Casts:* For now type casts are not supported by the importer.
- *Duplicated Variable Names:* It's possible for a C header to have declarations with the same name based on different conditional directives. The importer will import these declarations with different features(based on their conditional directives), but they are reported as 'duplicated names' error. It cannot be resolved right now, because it's a MPS constraint.
- *Parsing Typedef Expressions:* The importer now lacks parsing all possible typedef expressions and the parser part needs to be improved in future.
- *Parsing Arguments' Names:* Right now the importer's parser cannot parse the function prototypes which their arguments have names as an option. The parser should be improved in future works.
- *Importing Correct Values:* Importing the correct value parts of constants and macros can be a future work.
- *Duplicated Feature Name:* In mbeddr features' names must be unique, but there are conditions which the importer adds some features with the same name. It happens when a C header file has a conditional directive which is repeated in different parts. For example the following code will lead to a duplicated feature name error:

  #ifndef `_USE_FILE_OFFSET64`

  # ifdef **REDIRECT**

  # endif

  #endif

  #if !defined `_USE_GNU`

  # ifdef **REDIRECT**

  # endif

  #endif

- *Defining an Appropriate Configuration Model:* Right now the importer makes a configuration model which holds all the features, but not necessarily a legitimate subset. Selecting a correct subset of features, i.e. chunck of a C headr's declarations, is current problem of the importer.

- *Importing #elif Directive:* The importer doesn't support importing the #elif directive right now