PLM2C - PL/M to C Translator

Table of contents

INTRODUCTION

PLM2C accepts PL/M sources files and produces the corresponding ANSI C or K&R C source files ready to be compiled with a C compiler. The resultant C source code embodies the functionality of the original PL/M source code while adhering to modern C coding style. The translator is available on IBM PC or compatible running Windows 95 and above.
The current version of PLM2C handles PL/M 86, 286 and PL/M 51.

NOTEWORTHY FEATURES

- ☞ **ALL "based" declarations are handled by using the based pointer.**
- ☞ **ALL "AT" declarations that define "unions" in C are handled.**
- ☞ **ALL context sensitive translation of the "declare Literally" Macros. The "literally" macros (even nested or composed) which are being used for structure definition are translated to "typedef"s in C.**
  The "literally" macros which are being used for constants definition are translated to the "#define" macros in C.
- ☞ **ALL remarks are correctly placed in the translated C source code.**
- ☞ **ALL nested include files are translated.**

For more technical details it is highly recommended that you read the Appendixes at the end of this document. The technical descriptions will help you appreciate the state of the art of this translator.

Usage Manual And Invocation Flags

The translator is invoked for each file that should be translated. A batch file or a make file can be defined to automate the translation of a group of files. The destination directories for the translation should be defined in the configuration files as listed in the next section. All detected translation errors, if exist, are displayed on the screen with meaningful messages, and are also logged to a file named "plm2c.log" in the working directory. The following line is an example of invoking Plm2C:

```
Plm2c myFile.plm
```

The number of invocation flags is small since most of the translation options are defined in the configuration files. (See the next sections).

Invocation flags:

-k (plm2c –k filename.plm)

Forces the translation to produce K&R C source files. The default is an ANSI C translation.

-c (plm2c -c filename.plm)

Affects the translation of the compiler directives "$if" and "$else".
Suppose that the following statements appear in the PL/M source file:

```
$if NOT_INCLUDED
$include ( globals.lit)
$endif
```


If the flag NOT_INCLUDED is evaluated as TRUE, during the translation process, the translator will not translate the statements between the "$if" and "$end" directives if the invocation flag '-c' is used.
Using this invocation flag can help sometimes when the "$if" directive is used to avoid multiple inclusion of files, as in the above example.
In the above example the inclusion of the "globals.lit" file is controlled by the "NOT_INCLUDED" flag. If after the first inclusion of the file the "NOT_INCLUDED" flag is set to TRUE then the file "globals.lit" will not be included again. Since programmers use this mechanism to direct the compiler preprocessor, it is included also in the translator, in order to avoid multiple declarations errors.
The evaluation of the expressions in the "$if" directive is restricted to simple expressions that consist of only one flag, as in the above example.
If the "-c" flag is not specified in the invocation line, all statements between the "$if" and "$end" directives are translated.

-p51 (plm2c –p51 filename.plm)

Declarations in PL/M 51 with no specified memory type are translated with a "data" memory type.
Example:

```
PL/M:    DECLARE X1 byte;
C:       static BYTE data X1;
```

-c51 (plm2c –c51 filename.plm)

Declarations in PL/M 51 with a CONSTANT memory type are translated with a "code" memory type.
Example:

```
PL/M:    DECLARE TABLE(*) BYTE CONSTANT(1,2,3);
C:       static BYTE code table[] = {1, 2, 3};
```

Configuration Files

Before using the translator, several configuration files should be defined in order to define the environment in which the translation takes place. The following sections describes what each of these files should define.

The "plm2c.dat" configuration file

This file defines where to put the translated files. It should contain a path definition where to put the translated source files, and a path definition where to put the translated include files. each path definition is bounded by double quotes.

Example:

```
"\translated\src"
"\translated\inc"
```

The above example shows the content of a possible "plm.dat" file. The first line defines that the translated source files should be put in the directory "\translated\src" and the second line defines that the translated include files should be put in the "\translated\inc" directory. The first line always defines the path for the source files and the second line always defines the path for the include files.

The "plm2c.inc" configuration file

This file gives a solution when an include file declaration, in a PL/M source file, contains some path directive symbols which are interpreted by the operating system but are not known to the translator.

In the "**plm2c.inc**" file it is possible to declare the original path, its interpretation, and a destination path where to put the translated include file.
This definition overrides the path that was declared in "**plm2c.dat**" for the specific include path only.
This is a useful mechanism to help the translator resolve path directives even if the translation is not being executed in the same environment in which the PL/M compilation takes place.
This file should contain triplets, each in a different line. Each triplet is in the following format:

<"sourceIncludePath"> <"pathInterpretation"> <"destinationPath">

Example:

```
"${DRIVE1}/"  "\project\inc\"  "\project\translated\inc\"
```

In this example the "${DRIVE1}/" symbol is known to the OS (as an environment symbol for example). The above line informs the translator to replace this symbol by "\project\inc\" when trying to open an include file which have the "${DRIVE1}/" symbol in its path definition.
The last element in the triplet, the "\project\translated\inc\" path, defines the path where the translator should put the translated include files which have a path that match the first element in the triplet.
The "**plm2c.dat**" file defines the default destination path for the translated include files, while the "**plm2c.inc**" configuration file help customize the translator to handle interpretation of specific include paths.

The "plm2c.bf" configuration file

This configuration file controls the translation of built-in functions defined in the PL/M language. These functions can accept different types of parameters (byte, word, dword etc..) and handle them correctly.
When translating to C, each built-in function should be broken to several functions, each accepting a different parameter type. For example the "high" built-in function should be translated to "high_b" (for example) when it operate on a byte parameter, and to a "high_w" function when it operate on a word parameter.
This information should be declared in the "**plm2c.bf**" configuration file. This file contains quadruplets in each line. Each quadruplet associates information to a built-in function.
Each quadrate is described in the in the following BNF format:

<built-in function name> <parameter type> <translated name> <return type>

<built-in function name> :: STRING

<translated name> :: STRING

<parameter type> :: integer | real | byte | word | dword

<return type> :: integer | real | byte | word | dword

Example:

```
high  byte  high_b  byte
high  word  high_w  word
```

These two lines in the "**plm2c.bf**" file direct the translator to translate the "high" built-in function to "high_b" function name, when it encounters in the PL/M source file this built-in function with a byte type parameter. When it encounters this function name with a word type parameter, it will translate it to the "high_w" function name instead.
The translator creates automatically a "builtin.h" file in the include files directory, (as defined in the "**plm2c.dat**" file), and includes it in the translated source file. The "builtin.h" file contains all the prototypes of the functions that were declared in the "**plm2c.bf**" file.

The user should supply a library of the C functions, that were declared in the "**plm2c.bf**" file, which have the same functionality as the PL/M built-in functions. Typically these are usually very simple functions.
There are three built-in functions which are handled directly by the translator, and which cannot be redefined in the "**plm2c.bf**" file. These are the "SIZE" built-in function (which is translated to the "sizeof" C function), the "LENGTH" and the "LAST" built-in functions. Any attempt to redefine them in the "**plm2c.bf**" file is ignored. The translation of the "SIZE" built-in function is not trivial when it receives a "BASED" variable parameter, but is readable and clear.

PL/M Compiler Directives Translations

The PL/M language support a list of compiler directives. The compiler directives that have no affect on the translation (like printing controls) are ignored. The other ones are translated to their C preprocessor directives equivalents, or affect the translation in other way, if there is no real equivalent directive to be translated to.
Here is a list of all the handled compiler directives:

1. **$include**
2. **$leftmargin** - it affects the scanning of the source file.
3. **$if**
4. **$else**
5. **$endif**
6. **$set**
7. **$reset**
8. **$save** - affect the translation as it should affect the PL/M compiler. The translator maintains a stack for this purpose.
9. **$restore** - affect the translation since it relates to the $save directive (see $save).

Problematic And Interesting Translations

BOOLEAN EXPRESSIONS

In the PL/M language an expression is **True** if in its bit representation the rightmost bit is 1. The value is **False** when the rightmost bit is 0.
In the C language the convention is different. The **True** value is associated with a non zero value, and the **False** value with the zero value. In the translation a conversion is done in a readable and efficient manner.
In PL/M the Boolean operators "Or" and "And" are always bit operators, but the translation is context sensitive and detects the places where logical operators that are not bitwise, should be placed in the translation. **(See translations examples in the 'examples' subdirectory)**.

BASED VARIABLES

In the translation of based variables the based pointer is used explicitly in the translation to C. The type of the pointer is determined by the type of the based variable.

Example:

```
declare V1_ptr pointer;
declare V1 based ptr word;

V1 = 10;
```

is translated to:

```
WORD *V1_ptr;   /* WORD is declared as typedef of unsigned
                                short */
*V1_ptr = 10;
```

When two or more based variables use the same pointer, a correct casting is added whenever this pointer is accessed. **(See translation examples in the 'examples' subdirectory)**.

STRUCTURE TYPEDEF DECLARATION

A structure declaration is translated to a "typedef" declaration with a name that is generated by the translator. Structures that were implicitly declared by using the "declare literally" macro mechanism of the PL/M language, are explicitly declared as "typedef" declarations in C. Complex usage combinations of these "literally" macros, including nested declarations, are translated correctly to structure "typedefs" or appropriate "#define" statements. **(See Appendix A and translation examples in the 'examples' subdirectory).**

UNIONS

Variables that were declared "AT" another variable address are grouped in a C "union" declaration. This is done for the simple and complicated "at" declarations as well, and results in a readable and clear source code in C. **(See Appendix B and translation examples in the diskette).**

APPENDIX A - The "Literally" Declaration Translation

In PL/M one can define a macro by using the "declare literally" declaration.

examples:

```
declare Table_size literally  '10';
declare record literally 'structure (aa(10) byte , bb word)';
declare dcl literally 'declare';
```

The PL/M preprocessor expands each macro before calling the compiler.
For the first example above it means that whenever the compiler encounters the identifier "Table_size" it will replace it with the number 10. The macro "record" is expanded as a structure definition, and the third macro is expanded to the "declare" PL/M keyword.
The last macro is being used for the convenience of the programmer. Thus he/she can define shorter aliases names for keywords of the PL/M language itself.
We see that the PL/M macro declaration semantics is different in each of the above examples. The first is a constant definition, the second is like a structure "typedef" declaration in C, and the third is a simple alias name.
If we translate each of these declarations by just expanding the macro, as the PL/M preprocessor does, it will result in an unreadable and unmaintenable programs since the "**Table_size**" and "**record**" declarations will appear in many places. By looking at the above examples it is clear that we would like to interpret each of them differently in order to generate a high quality translation.
The first declaration should be translated as a "**#define**" macro in C. The second one should be translated as a "**typedef**" declaration of a structure in C, and the third one should only direct the translator to expand the alias name "dcl" to the string "declare" whenever it encounters it in the source code. This is a context sensitive translation which complicates the translation process, but is a "must" in order to generate a readable and maintainable C source code program from the original PL/M program.
If a context sensitive translation is applied the translated program is more readable even then the original PL/M program. The translator makes the interpretation automatically and generates different declarations in C, which clarify the meaning of the original PL/M declarations.
The usage of the "declare literally" declaration in PL/M, for declaring structures, can be much more complicated then the above example. Consider the following example:

```
declare rec literally  'structure (a byte, end_st' ;
declare end_st literally  'b word)' ;

      /* one instance of the above defined structure */
declare new_st  rec;
```

The above example translated to C will be:

```
typedef struct {
        BYTE a;
        WORD b;
}       rec;

/* one instance of the above defined structure */
static rec new_st;
```

In this example the structure declaration is made by using two "declare literally" macros, one is included (nested) in the other. The translator, on the other hand, generates one "typedef" deceleration in C as a result of these two macros.
The translator can handle any number of macros like these, and with any level of nesting, and will collect all of them to one clear "typedef" definition in C.
This clarifies even more the claim that the translated programs are even more readable and clear then the original programs in PL/M. Once again, this is due to the fact that C has more powerful and precise declaration mechanisms than PL/M, and that a correct context sensitive and automatic interpretation is done by the translator.
There can be a more complex structure declaration with a "declare literally" macro definition.

example:

```
declare st_struct literally  'structure (a byte,' ;
declare end_st literally  'b word)' ;

        /* one instance of the above defined structure */
declare new_st   st_struct  end_st;
```

The difference from the previous example is that in this case the macros are not nested, but are composed together for the declaration of a structure.
It is more complicated to detect that the two (or more) macros form together one structure declaration. The translator detect that this is one structure declaration only if the macro declarations ("st_struct" and "end_st" in this example) appear in the correct order in the file and one after the other. A correct order means that the structure declaration can be gathered when scanning the macros one after the other.
In the above example if the macro "end_st" would appear in the file before the macro "st_struct", no "typedef" definition in C will be generated but a syntax error will be signaled. In this case it suffice to put the macros that define the structure definition in the correct order and run the translator on the modified file. This case is rarely encountered since the programmer prefer to put these macros in the correct order in order to make the program readable and clear for himself.
The translation of the above example would be:

```
typedef struct {
        BYTE a;
        WORD b;
}       st_struct;
/* one instance of the above defined structure */
static st_struct new_st;
```

The "literally" macro declarations translation is a context sensitive translation that is aimed to produce readable and maintainable programs. Since the C language has a richer and more precise set of declarations, and since the translator makes a correct and automatic matching to this set of declarations, the translation is even more readable and clear than the original PL/M source code.

☞ **The original programmer of the PL/M program will find it easy to understand and navigate in the translated program.**

APPENDIX B - "AT" Declaration Translation

Variables that were declared "AT" another variable address are grouped in a C "union" declaration in the translated program. This is done for the simple and complicated "AT" declarations as well, and results in a readable and clear source code in C.
A simple "AT" declaration is a declaration of a variable "AT" the address of another variable.

example:

```
declare v1 word;
declare v2 byte at (@v1);
```

The variables "v1" and "v2" start at the same memory address. This is equivalent to the following "union" statement in C:

```
union {
      /* WORD and BYTE are declared as typedefs in the
     automatically generated "builtin.h" file */
       WORD  v1;
       BYTE  v2;
} Un1;
```

The Translator finds the group of the variables that start at the same address, as a result of the usage of the "AT" attribute, and creates a "union" statement that includes all of them. It generates a unique name for the union that always starts with the prefix "Un".
(Since all the translated names are in lower case letters the prefix "Un" will not collide with another identifier name).
Any reference, in the translated code, to a variable in the union will be as a union member reference. For example if in the PL/M code an assignment was made to the variable "v1" as follows:

```
v1 = 1;
```

The translated statement will be:

```
Un1.v1 = 1;
```

Things get more complicated when the "AT" attribute is being used to point to a member of an array.

example:

```
declare v1(10) word;
declare v2 byte at (@a(5));
```

In this example "v2" starts at the same address as the fifth member of the array v1. The union translation reflects this fact:

```
union {
      WORD v1[10];
      struct {
           WORD    Offset1[5];
           BYTE    v2;
      } St1;
} Un1;
```

The "v2" variable is declared in an intermediate structure within the union, with a preceding correct offset. Any reference to this variable is translated correctly as a member of a structure within a union. The same idea is applied to variables that are declared "AT" structure members.

example:

```
declare v1 structure ( a byte , bb word , xy dword);
declare v2 byte at (@v1.xy);
```

The translation will be:

```
typedef struct {
      BYTE a;
      WORD bb;
      DWORD xy;
} StrucT1;

union {
      StrucT1 v1;
      struct {
          BYTE    Offset1;
          WORD    Offset2;
          BYTE v2;
      } St1;
} Un1;
```

A reference to "v2" in the PL/M code like:

```
v2 = 5;
```

is translated to:

```
Un1.St1.v2 = 5;
```

Some examples of complex union translations are shown in the file "union.c" that is a translation of the file "union.plm". Please look for this file in the EXAMPLES\PLM directory.
Generated union names are unique in the file context. If at least one of the union members was declared as a "PUBLIC" variable in the original PL/M program, a unique name for the union should be generated in the project context since a reference to it can be made from variables in other files.

Suppose that "v1" was declared "PUBLIC" in PL/M as in the following example:

```
declare v1 word public;
declare v2 byte at (@v1);
```

The variable "v1" might be declared as "external" in other files and be referred to by variables in these files. The naming convention of the union declaration should be different since it should be unique in the project context and not only in a single file context.
The name of the generated union in this case would be "Un_v1" since "v1" is already unique in the project context . The above example translation will be:

```
union {
      WORD v1;
      BYTE v2;
} Un_v1;
```

The Grouping of all variables that were declared "AT" other variable addresses, in a single union in C, makes the program more readable and clear, since there might be some levels of "AT" declarations which are hard to follow.

Suppose that "v1" is declared "AT" the address of "v2" and "v2" is declared "AT" the address of "v3" and so on. If these declarations are not continuos in the PL/M file it is hard to follow that they all start at the same memory address.
The translator collects all these declarations, whether or not they appear continuously in the original PL/M file, and forms one union with them which reflects immediately the relation between them.

☞ **surprisingly the translation is even clearer and more maintainable than the original PL/M program.**