BEREMIZ IEC61131-3 IDE

INTEGRATING C CODE IN BEREMIZ

Written by Benoit BOUCHEZ from KissBox for Beremiz community

http://www.kiss-box.nl

Contents

1 - Introduction

Beremiz IDE permits to create Programmable Logic Controllers using IEC61131-3 standard. It heavily relies on MatIEC to generate C source code that can be compiled on many platforms,

However, the weak point of Beremiz / MatIEC is the lack of complete documentation, especially on C and Python extensions.

Please note : this document is made to be shared for free, and to help all Beremiz users. The only requirement is to keep me as the original author. Thanks...

2 - Using pragmas with Beremiz

In compilers world, a pragma is a specific instruction which controls the compiler itself but does not generate executable code by itself. For example, with most C/C++ compilers, the "pack push" and "pack pop" pragmas control dynamically the memory alignment of structures.

The IEC61131-3 norm also defines "pragma" (see section 6.2 of norm for details). The norm tells that "Syntax and semantics to build a particuliar pragma are specific to integrator". In other terms, the norm lets implementors decide what a pragma should do, and every IEC61131-3 development environment will react differently to the content of pragmas.

The programmers behind Beremiz and MatIEC got a very clever idea : they decided to use IEC61131-3 pragma to allow to put C code directly in the PLC code. Simply said, you can mix C and ST in any POU or program !

2.1 - Pragmas in IEC61131-3

A IEC61131-3 pragma is delimited by brace symbols (also known as "curly brackets"). Pragma starts with the opening brace "{" and ends with closing brace "}". Everything located between the two braces is excluded from ST code analysis.

Pragmas are allowed everywhere in IEC61131-3 programs where spaces are allowed, except in string litterals (otherwise you could not use the braces in IEC61131-3 strings)

2.2 - C functions vs. IEC61131-3 languages

The IEC61131-3 norm specifies that pragmas are part of "common element", like identifiers, comments, etc... So, even if ST appears as the best adapted language to include pragmas, it is possible in fact to use them in any of the IEC61131-3 languages : ST, SFC, FBD, LD and even IL.

Don't forget that Beremiz translates all languages into ST, so your C code will appear at the exact place in the processing sequence as it has been placed in the source code.

2.3 - The key rule : POU = C function

MatIEC translates all POUs in all programs as C functions. So, programmers shall always keep in mind that POU are coded as "pass-through" functions, since they are executed ("repeated") cyclically when the task controlling the program is triggered.

Consequently, there are limitations on what can be written in C using pragmas.

- Avoid loops in the C except if you know exactly what you do (by the way, this requirement applies to all languages from IEC61131-3, as a loop may lead to overruns, if the CPU remains stuck in a realtime POU for a too long time)
- Avoid any system call from embedded C source code from a realtime task except if you are absolutely sure that the system call will return in a given time (which is not the case with most OS). If your C code performs a system call, it must be performed in a dedicated task without realtime requirements.
- Do not forget that PLCs work by calling cyclically the different POUs under control of the tasks. POUs are read from top to bottom, as fast as possible.

To have a better understanding of these limitations, let's see how a ST POU is translated in C by MatIEC.

The POU with a pragma written in ST (color are used only to highlight the pragma) :

```
1  IF Reset THEN
2        Cnt :=  ResetCounterValue;
3  ELSE
4        Cnt := Cnt + 1;
5  END_IF;
6  {call_C_function();}
7  Out := Cnt;
```

And the same POU translated into C by MatIEC

```
8    if (__GET_VAR(data__->RESET,)) {
9      __SET_VAR(data__->,CNT,,__GET_EXTERNAL(data__->RESETCOUNTERVALUE,));
10   } else {
11     __SET_VAR(data__->,CNT,,(__GET_VAR(data__->CNT,) + 1));
12   };
13   #define GetFbVar(var,...) __GET_VAR(data__->var,__VA_ARGS__)
14   #define SetFbVar(var,val,...) __SET_VAR(data__->,var,__VA_ARGS__,val)
15 call_C_function();
16   #undef GetFbVar
17   #undef SetFbVar
18 ;
19   __SET_VAR(data__->,OUT,,__GET_VAR(data__->CNT,));
```

As you can see, the C source code embedded in the pragma is entirely copied in the function body generated by MatIEC, exactly at the same place as in the ST code.

2.4 - Accessing PLC variables from C code

C code in a pragma can access any IEC61131-3 global or local variable. However, you have to take into account the way MatIEC encapsulates variables.

You can see below how a local Function Block variable is passed to a C function :

- MatIEC uses a pointer to a structure called data__ to reach local variables in a Function Block
- All identifiers are converted to UPPER CASE even if they are lower case in ST source code. In the example below, the VAR_INPUT variable named **Reset** in ST is named **__data->RESET** in C.

```
20 FUNCTION_BLOCK CompteurST
21   VAR_INPUT
22     Reset : BOOL;
23   END_VAR
24   VAR_OUTPUT
```

```
25      Out : INT;
26    END_VAR
27    VAR
28      Cnt : INT;
29    END_VAR
30    VAR_EXTERNAL CONSTANT
31      ResetCounterValue : INT;
32    END_VAR
33
34    IF Reset THEN
35      Cnt :=  ResetCounterValue;
36    ELSE
37      Cnt := Cnt + 1;
38    END_IF;
39
40    {my_C_function(data__->RESET);}
41
42    Out := Cnt;
43  END_FUNCTION_BLOCK
```

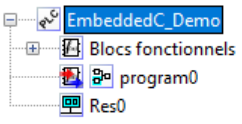
2.5 - Multiple lines pragmas

A pragma can represent one or multiple lines of C source code. It is then possible to use two different approaches if the C source code counts more than one line :

- use one pragma per line of C source code
- embed the Carriage Return / Line Feed sequence between each source code line within the pragma

The two approaches work equally and can even be mixed if necessary, the main criteria being the readability of the POU.

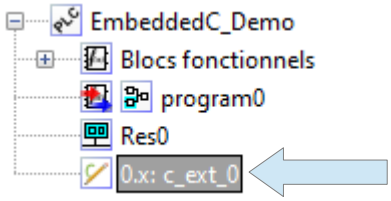
3 - C extensions



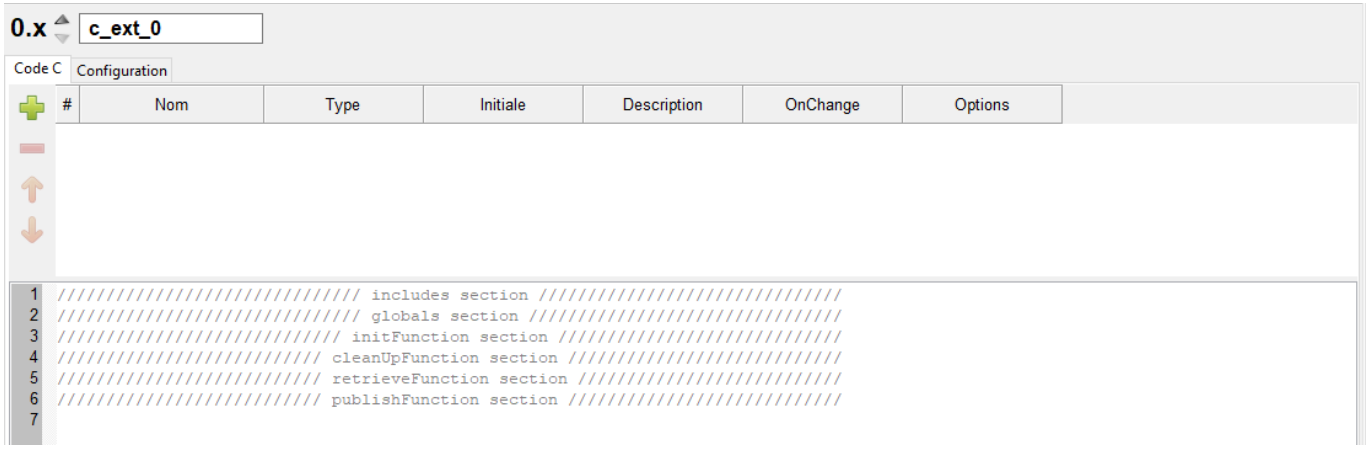
The C functions called from POUs must be declared in "C extensions", included in the project. To create a C extension, click on the "+" button in the Beremiz Editor project window, then choose "C extension"

A C extension item is then added to the project tree. Note that this item is global to the project, it does not

depend on Function Blocks, POUs or Ressources. It is possible to create as many C extensions as needed, so you can easily organize your work.



When you click on a C extension, its parameters and content are displayed in the edition area.



The name of the C extension can be changed using the edition box on the top. Note that the name given here is only for information, it does not impact the C code.

You will notice a 0.x indicator on top left of the window, with two arrow buttons next to it. Each time a C extension is created, it gets a different number, which can be changed using the arrows.

These numbers are used within the c_ext to identify functions (the number is added as suffix to the various default functions)

Using the green "+" button, it is possible to create local variables for the C extension, exactly in the same way as local variables are created for POUs.

IMPORTANT : the various sections you see in the Editor are really distinct ones, even if they seem to appear to be from a single file. You have to take care to place your C code in the correct section otherwise it will be impossible to compile your project. Moreover, never remove the section headers (includes section, globals section, etc...) or your project structure will be broken.

3.1 - Includes section

The "Includes section" shall contain header files needed to compile the C functions declared in the C extension. Place here all the header files needed to compile the C code within the C extension.

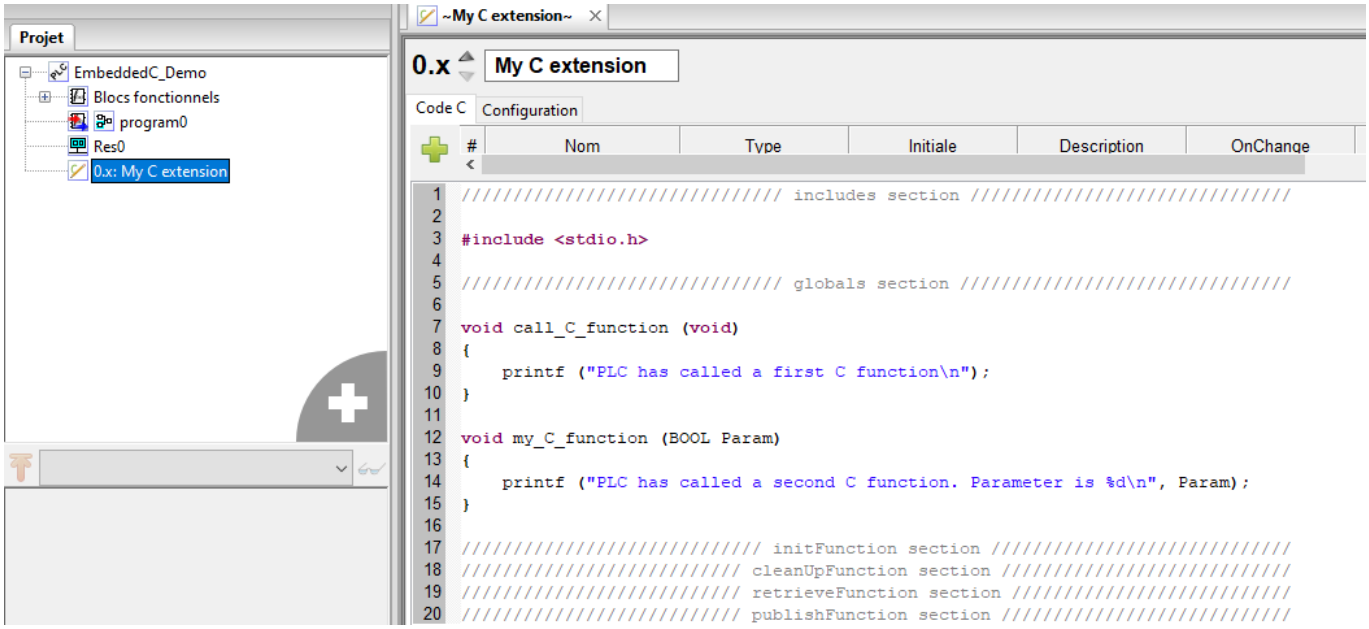
Note that C code generator of Beremiz includes automatically two files in each C extension :

- stdio.h
- iec_types_all.h

The last include file allows C functions to have direct access to all IEC61131-3 types

3.2 - Globals section

The "Globals section" contain bodies of C functions which can be called from PLC function blocks.



3.3 - initFunction section

By default, Beremiz generates the following code (the 0 suffix is the C extension number, defined when the C extension is created. It is then different for each extension) :

```
44 int __init_0(int argc,char **argv)
45 {
46   return 0;
47 }
```

Any source entered in this section is inserted in the __init_0 function (so do not write a full function here, like in the Globals section, otherwise it will not compile)

Example : following line entered in the initFunction section

TestCode (0);

produce the following code in the c_ext file

```
48 int __init_0(int argc,char **argv)
49 {
50 TestCode (0);
51 return 0;
52 }
```

3.4 - cleanUpFunction section

By default, Beremiz generates the following code :

```
53 void __cleanup_0(void)
54 {
55 }
```

Any source entered in this section is inserted in the __cleanup_0 function (so do not write a full function here, like in the Globals section, otherwise it will not compile)

3.5 - retrieveFunction section

By default, Beremiz generates the following code :

```
56 void __retrieve_0(void)
57 {
58
59 }
```

3.6 - publishFunction section

By default, Beremiz generates the following code :

```
60 void __publish_0(void)
61 {
62
63 }
```

4 - Creating a function block to access C functions

In order to access the C functions declared in C extensions, a wrapper is needed as IEC61131-3 does not provide direct interface to C.

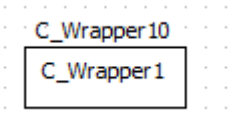
This wrapper will take the form of a Function Block, typically written in ST, which will describe all input/output parameters for the C function, and pass them to the C function.

4.1 - Special case : functions without parameters and return value

If the C function has the following declaration :

void ***functionName*** (void)

then the IEC61131-3 wrapper will not have parameters too, giving a Function Block looking like this :



Such a Function Block is perfectly legal in IEC61131-3.

Such a Function Block should appear like this when you declare them (note the absence of parameters on top of window)

However, current versions of Beremiz have a small issue when they deal with such Function Blocks, as Beremiz expects all functions blocks to have at least one parameter. If you try to generate the ST code without any parameters, Beremiz will report an error ("No variable declared in POU") and will refuse to generate the PLC code.

The workaround is very easy : just declare a single variable as external (it does not even need to exist). The cleanest way is simply to declare a global variable as the needed parameter.

5 - Document revisions

Date	Auteur	Version	Description
05/02/2020	Benoit Bouchez	1.0	First version
05/03/2022	Benoit BOUCHEZ	1.1	Updated document properties Corrected pagination
21/03/2022	Benoit BOUCHEZ	2.0	Major update to provide details about c_ext structure and usage in PLC code

Prepared with OpenOffice Writer software.