# Magic of Macros

## Using C-preprocessor as code generator?

András Gáspár, Dr. László Gianone, Dr. Gábor Tevesz

**Developing software for embedded systems restricts usage of possibilities trivially available in PC-environment. One field is handling of consistent data structures. E.g. identifying all CAN messages and applying their attributes consistently. One way of handling such configuration constructions is to create PC based configurator applications that generate all the needed embedded code constructions. But there is another way to use the preprocessor for the same purpose.**

Software development for embedded systems in series production usually suffers from shortage of resources, since hardware design is always optimized to save costs. Thus, RAM and ROM usage as well as necessary CPU load must be kept as low as possible. Optimizing usage of resources can be challenging and advanced software development methods may be required. This is especially true, if reusable code parts shall be included in the software.

Since reused software modules always contain parts, which are not necessary in the current project, support of conditionally compiled source code – i.e. usage of compiler switches – is obviously necessary. But not only the intensive usage of compiler switches makes a source code hard to maintain. Some software modules implement handling of relatively large amount of data. If the large data structures contain consistent segments, manual management of them can become rather difficult.

Typical such functions in an embedded system are ECU diagnostics, NVRAM parameter handling, layout of communication channels, handling of diagnostic trouble codes, etc. Though the actual implementation of these software modules can be always evaluated according to the agreed guidelines of the project, maintaining all data properly can be problematic in case of frequent changes of the data contents.

Instead of direct source code editing, storing all necessary data attributes in a proper database and generating all related source code sections for the embedded CPU by a code generator tool can be a clean and preferred solution in long term. Due to the practically unlimited amount of data storage and programming capabilities of a PC environment finding or even creating a source code generator tool is easy. However, the allowance of using a source code generator tool is sometimes not obvious.
In case an embedded application has active role in safety critical functions, further requirements must be fulfilled, which enforce further limitations of available development methods. Apart from keeping strict software coding guidelines, severe restrictions may exist for the building and maintaining environment. One of the most common restrictions is the assurance of reliability of the used tools. This means, not only the development of source code of the embedded system itself shall be secured, but all tools participating actively in the building environment must be certified properly.

ISO 26262 standard defines the necessary level of certification of the used tools in development of safety critical applications. See part 8, section 11: *Confidence in the use of software tools.*
In case of a system having components at ASIL-C or ASIL-D, reaching the expected confidence of a generator tool is rather difficult. This confidence must be certainly verified for the used compilation tool-chain, which creates the binaries to be downloaded in the FLASH of the embedded system. However, the final decision of a project may be to completely reject the usage of further code generator tools. In this case all source files must be maintained manually by keeping all rules of the regular software development process.

If we have to maintain a complex software structure manually without active help of any tool, the risk of making mistakes is large. Even if proper development process, appropriate reviews and tests are defined to avoid possible failures, the software development may become difficult.
Current article presents a possible solution in software projects written in ANSI C language to resolve or mitigate the contradiction of requirements for optimization, safety and maintenance. The central idea is to generate source code by using the C preprocessor. Since the C preprocessor is part of the already certified C compiler, its operation is reliable and its usage is allowed even in the most safety critical application.

**Magic of Macros**
In order to understand how C preprocessor can perform source code generation we analyze a simple example of keeping consistency between data structures. The file a.c contains a list of constants, which shall match to their index names defined in file a.h as literals of an enumeration:

a.c
```
const uint16 ad[A_SIZE] = {
  15,
  8
}
```

a.h
```
enum {
  A_1,
  A_2,
  A_SIZE
}
```

Certainly, our target is to gain access to the constant values with the means of the literals:

```
x = ad[A_2];
```

In order to be able to generate both the constant array and enumeration from the same source in an easily maintainable way, we need assignment between indices and values. This assignment can be solved in C syntax in the following way:

```
MAC(A_1, 15)
MAC(A_2, 8)
```

This is a function like statement, but the literal MAC can be also understood as a macro name for C preprocessor. Using a C preprocessor macro instead of a direct C function call has the advantage to be able to calculate something during compilation instead of using computation power of the target CPU.

Now the question is how we can transfer the information from these C macros to real C code. The clue is the #include directive of C preprocessor, which technically simply merges the referred file contents at the current location. The above mentioned macros shall be moved to a specific file mac.inc, and instead of the data this file shall be merged in the original files via #include directives:

a.c
```
#define MAC(a,b) b,
const uint16 ad[A_SIZE] = {
  #include “mac.inc”
}
```

a.h
```
#define MAC(a,b) a,
enum {
  #include “mac.inc”
  A_SIZE
}
```

mac.inc
```
MAC(A_1, 15)
MAC(A_2, 8)
```

In order to gain working source code the macro MAC must be defined by #define directives before the #include directives are reached. The necessary definitions can be observed above too.

The source file a.c must include its header a.h, because definition of A_SIZE is necessary for successful compilation. In this case the first macro definition of MAC inherited from file a.h must be removed by #undef directive before the current definition can be made in a.c:

```
#include "a.h"
...
#undef MAC
#define MAC(a,b) b,
const uint16 ad[A_SIZE] = {
  #include “mac.inc”
}
```

It is also possible to define partially different behavior for specific items. In case the macro MAC cannot always fully configure all necessary details, further macro EXT can be introduced for mac.inc:

a.c
```
#include "a.h"
...
#undef MAC
#undef EXT
```

mac.inc
```
MAC(A_1, 15)
MAC(A_2, 8)
EXT(A_3, 2, 4)
```

```
#define MAC(a,b) b,
#define EXT(a,b,c) b+c,
const uint16 ad[A_SIZE] = {
  #include “mac.inc”
}
```

This is identical to:

a.c

```
const uint16 ad[3] = {
  15,
  8,
  2+4,
}
```

At this point file a.c has already become rather hard to follow. If we had more macros and more data structures in the same file, the source code could become more difficult to understand. Therefore, further restructuring is suggested to be able to use such a structure efficiently. We introduce a new file a.inc for all macro definitions:

a.inc

```
#ifdef MAC
  #undef MAC
#endif
#ifdef DEF_A_TABLE
  #define MAC(a,b) b,
#elif defined DEF_A_INDEX
  #define MAC(a,b) a,
#else
  #define MAC(a,b)
#endif

#ifdef EXT
  #undef EXT
#endif
#ifdef DEF_A_TABLE
  #define EXT(a,b,c) b+c,
#elif defined DEF_A_INDEX
  #define EXT(a,b,c) a,
#else
  #define EXT(a,b,c)
#endif

#include "mac.inc"
```

Now, the original source code becomes this simple:

a.c
```
#include "a.h"
...
const uint16 a[A_SIZE] = {
  #define DEF_A_TABLE
  #include “a.inc”
  #undef DEF_A_TABLE
}
```

a.h
```
enum {
  #define DEF_A_INDEX
  #include “a.inc”
  #undef DEF_A_INDEX
  A_SIZE
}
```

At this stage we have managed to get rid of consistent data maintenance in the original source files, but they are kept readable. All the variable information is collected in one common file mac.inc, which is very simple to maintain. By simply adding a new macro statement or removing an old one in this file all consistent source codes are automatically modified without needing further attention. The macro definition file a.inc is also well structured, and easy to add or remove a new macro definition group.

Operating with macros in this way implements real source code generation. This means, not only constant arrays and enumerations can be maintained so, but using #include in a different context real executable code can be also created. The complexity of the generated output can be increased even on a large scale. However the more complicated structure is defined the more difficulties will be experienced when the software is traced in a debugger.

**Evaluation of implementation**

Looking at the original source files, and comparing them to the latest codes, the increase of complexity is obvious. Having no experience in using preprocessor statements in such a way, understanding the source code is more difficult. Debugging software parts using e.g. constant tables generated by macros can be also slightly disturbing.

On the other hand, the original target has been reached successfully: source code with consistent data structures are possible to be generated by the means of a certified tool. Furthermore, maintenance of the data becomes really simple. Even complicated relationships and many parallel structures can be easily maintained now, and potential human mistakes can be reduced dramatically.

It is also important to notice the allocation of data in the source code. While the original solution contains data in a distributed way in different files, the new structure contains all data at one common place. Furthermore, the centralized data contains no additional overhead, which cannot be avoided in many cases due to the limitations of ANSI C syntax. This way the file mac.inc becomes a very clean configuration for the current project.

In order to be able to judge whether using this technique gives real benefits or just makes the source code more complex, the expected amount of data must be considered.

Convincing advantages can be experienced in case of large data quantity, which must be updated frequently in the life time of a software project.

Using macros and #include directives in this way is unusual, but not prohibited. Syntactically it is fully compliant to all ANSI C standards. In practice all tested C compilers and static code analyzer tools can understand these statements with no errors and warnings.

Considering regulations of MISRA C 2012, there are 3 guidelines, which are violated:

1. MISRA 2012, Dir 4.9 (advisory): *A function should be used in preference to a function-like macro where they are interchangeable.*
2. MISRA 2012, Rule 20.1 (advisory): *#include directives should only be preceded by preprocessor directives or comments*
3. MISRA 2012, Rule 20.5 (advisory): *#undef should not be used*

It is important to notice, that all affected rules are advisory. Advisory rules are defined by MISRA this way:

*6.2.3 Advisory guidelines*
*These are recommendations. However, the status of "advisory" does not mean that these items can be ignored, but rather that they should be followed as far as is reasonably practical. Formal deviation is not necessary for advisory guidelines but, if the formal deviation process is not followed, alternative arrangements should be made for documenting non-compliances. […]*
This means, the presented structure is not fully in line with MISRA recommendations, however the level of violation is minor, and according to decision of software quality assurance these deviations can be accepted.

**Summary**
The message of this article may not be the invention of a new guideline. It may not be considered as an obviously recommended solution for state of the art software technology. The new software structure can be considered more difficult to understand and to debug.
However, as long as source code generator tools cannot be easily introduced in a safety critical embedded application, such or similar compromise solutions may be accepted. A project decision may be both to accept and to reject the usage of such programming style.
The acceptance of the method can be based on individual preference, but the benefits are inevitable. A well-structured overview is gained on most constant data elements clearly separated from the core software functions. All other software parts can be kept completely untouched in case of data related modifications.

The clean and well separated structure gives the most important advantage: reviewing and testing modifications on the common data file is straightforward and simple. In long term the faster and more reliable software modification practice can lead to a professionally organized and well managed software development.

**Abbreviations**

| | |
|---|---|
| ASIL | Automotive Safety Integrity Level (ISO26262) |
| ECU | Electronic Control Unit |
| NVRAM | Non-Volatile Random Access Memory |

**Bibliography**


| | |
|---|---|
| ISO 26262 | Road vehicles – Functional safety |
| MISRA C 2012 | Guidelines for the use of the C language in critical systems |


**Authors**


András Gáspár has been involved in embedded software development for safety critical applications for more than 14 years in the automotive industry. Direct relationship has been established to Budapest Technical University, Department of Automation and Applied Informatics by tutoring students and as external lecturer. Since 2013 he is also Automotive Functional Safety Professional (ISO26262).



Dr. László Gianone has been involved in embedded software development for safety critical applications for 20 years in the automotive industry. He is also an external lecturer at Budapest Technical University, Department of Automation and Applied Informatics. He did his Ph.D. in 1995 in the field of Control Theory at Computer and Automation Institute HAS after he finished his M.Sc. studies in 1992 in Electrical Engineering at Budapest TU.



Dr. Gábor Tevesz has been involved in embedded hardware and software development for microcontrollers for about 35 years with a strong focus on microcontroller-based process control systems. He did his Ph.D. in 1999 in the field of modern process control systems. He is an associate professor at Budapest University of Technology and Economics, Department of Automation and Applied Informatics. In recent times Tevesz's research has focused on controlling and path planning of mobile robots respectively intelligent control of autonomous vehicles.