In House C-Coding Rules (IHRC)

L&R Ingeniería – Rev. 2e 12-02-22 - R. Oliva

1. INTRODUCTION

This document seeks to set a few internal rules, based on industry standards for C programming of embedded systems, mainly focused on programming the in-house designed CL2 [ref1_CL2b,2010] (AVR), CL3 [ref2_CL3,2018] (STM32F4) boards and M4/E and METEO (PSoC-1) architectures. It takes as its base the Barr C coding rules [ref3_Barr,2018] and the work from Kieras [ref4_Kieras, 2012], but merges other rule sets from Labrosse's work [ref9_JLabrosse, 2000], Samek [ref6_Samek,2013], and to a lesser extent Pont [ref10_PontEmbC,2002],[ref11_PontERES2,2017]. It is not explicitly intended for safety-critical systems as in [ref-MISRA-C, 2012] or to reach SIL level certification but mainly to enforce healthy programming habits that reduce the possibility of errors and enhances reliability of embedded systems. L&R's experience shows that in older projects with tight schedules, a great deal of poorly written code managed to get through and newer practices such as static code checking, TDD [ref12_GrenningTDD,2011], agile programming and Continuous Integration (CI) tools are now available to reduce defects on newer programs. The C programming language, even with latest additions from C99 [ref0_C99,1999] gives a great freedom to programmers, but can make programs unreliable and hard to maintain or upgrade. Legacy code can also be incrementally refactored introducing elements from this document and its references. Board CL2 has been historically programmed with CodevisionAVR C compilers v3 or newer [ref-CVAVR,2022] but although it is not strictly Arduino-hardware compatible it can use many Arduino-AVR libraries and can be programmed with C++ style Arduino IDE [ref-ArduinoIDE,2021] and Visual Studio Code [ref-VSC,2022] with PlatformIO[ref-PIO,2022], both using GNU avr-gcc [ref-AVRgcc,2021] compilers. Boards M4/E and METEO deploy Cypress/Infineon CY8C29466 parts with PSoC-1 CPU cores and use a specialized ImageCraft C compiler integrated into the PSoC-Designer IDE [ref-PSoCDes,2022], to support reconfigurable hardware analog and digital modules.

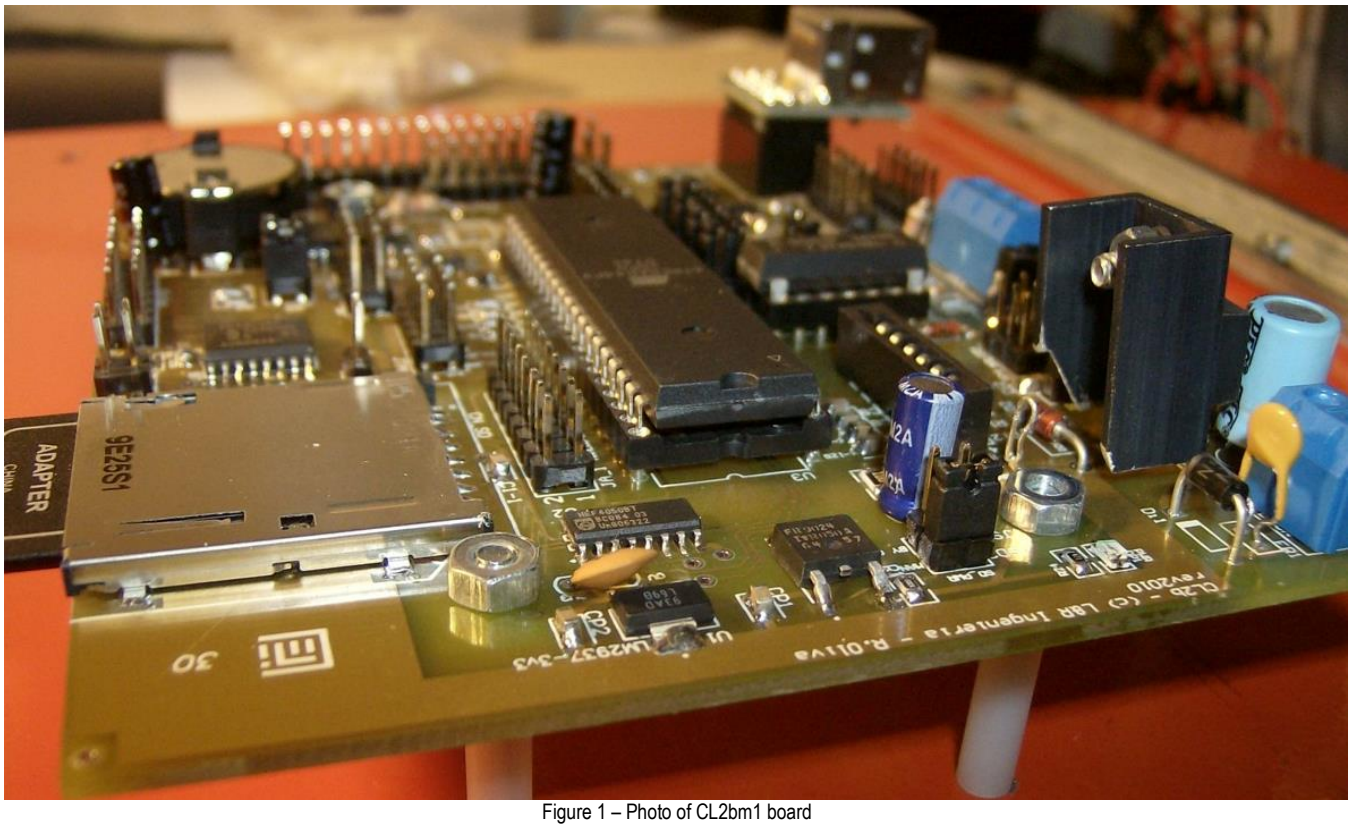

Figure 1 – Photo of CL2bm1 board

2. PART1 – KIERAS RULES [ref4_Kieras,2012]

Rule #1: Each module with its .h and .c file should correspond to a clear piece of functionality. Conceptually, a module is a group of declarations and functions can be developed and maintained separately from other modules, and perhaps even reused in entirely different projects. Don't force together into a module things that will be used or maintained separately, and don't separate things that will always be used and maintained together. The Standard Library modules math.h and string.h are good examples of clearly distinct modules.

Rule #2. Always use "include guards" in a header file. The most compact form uses #ifndef. Choose a guard symbol based on the header file name, since these symbols are easy to think up and the header file names are almost always unique in a project. Follow the convention of making the symbol all-caps. For example "event_handler.h" would start with:

```
#ifndef EVENT_HANDLER_H
#define EVENT_HANDLER_H
```

and end with:

```
#endif
```

Note: Do not start the guard symbol with an underscore! Leading underscore names are reserved for internal use by the C implementation – the preprocessor, compiler, and Standard Library – breaking this rule can cause unnecessary and very puzzling errors. The complete rule for leading underscores is rather complex; but if you follow this simple form you'll stay out of trouble.

Rule #3. All of the declarations needed to use a module must appear in its header file, and this file is always used to access the module. Thus #including the header file provides all the information necessary for code using the module to compile and link correctly. Furthermore, if module A needs to use module X's functionality, it should always `#include "X.h"`, and never contain hard-coded declarations for structure or functions that appear in module X. Why? If module X is changed, but you forget to change the hard-coded declarations in module A, module A could easily fail with subtle run-time errors that won't be detected by either the compiler or linker. This is a violation of the "One Definition Rule" [ref-odr1], [refodr2] which C compilers and linkers can't detect. Always referring to a module through its header file ensures that only a single set of declarations needs to be maintained, and helps enforce the One-Definition Rule.

Rule #4. The header file contains only declarations, and is included by the .c file for the module. Put only structure type declarations, function prototypes, and global variable extern declarations, in the .h file; put the function definitions and global variable definitions and initializations in the .c file. The .c file for a module must include the .h file; the compiler can detect discrepancies between the two, and thus help ensure consistency.

Rule #5. Set up program-wide global variables with an extern declaration in the header file, and a *defining declaration* in the .c file. For global variables that will be known throughout the program, place an extern declaration in the .h file, as in:

```
extern int g_number_of_entities;
```

The other modules `#include` only the .h file. The .c file for the module must include this same .h file, and near the beginning of the file, a defining declaration should appear - this declaration both defines and initializes the global variables, as in:

```
int g_number_of_entities = 0;
```

Of course, some other value besides zero could be used as the initial value, and static/global variables are initialized to zero by default; but initializing explicitly to zero is customary because it marks this declaration as the defining declaration, meaning that this is the unique point of definition. Note that different C compilers and linkers

will allow other ways of setting up global variables, but this is the accepted C++ method for defining global variables and it works for C as well to ensure that the global variables obey the One Definition Rule.

Figure 1 shows a concise graph of rules #1 to #5

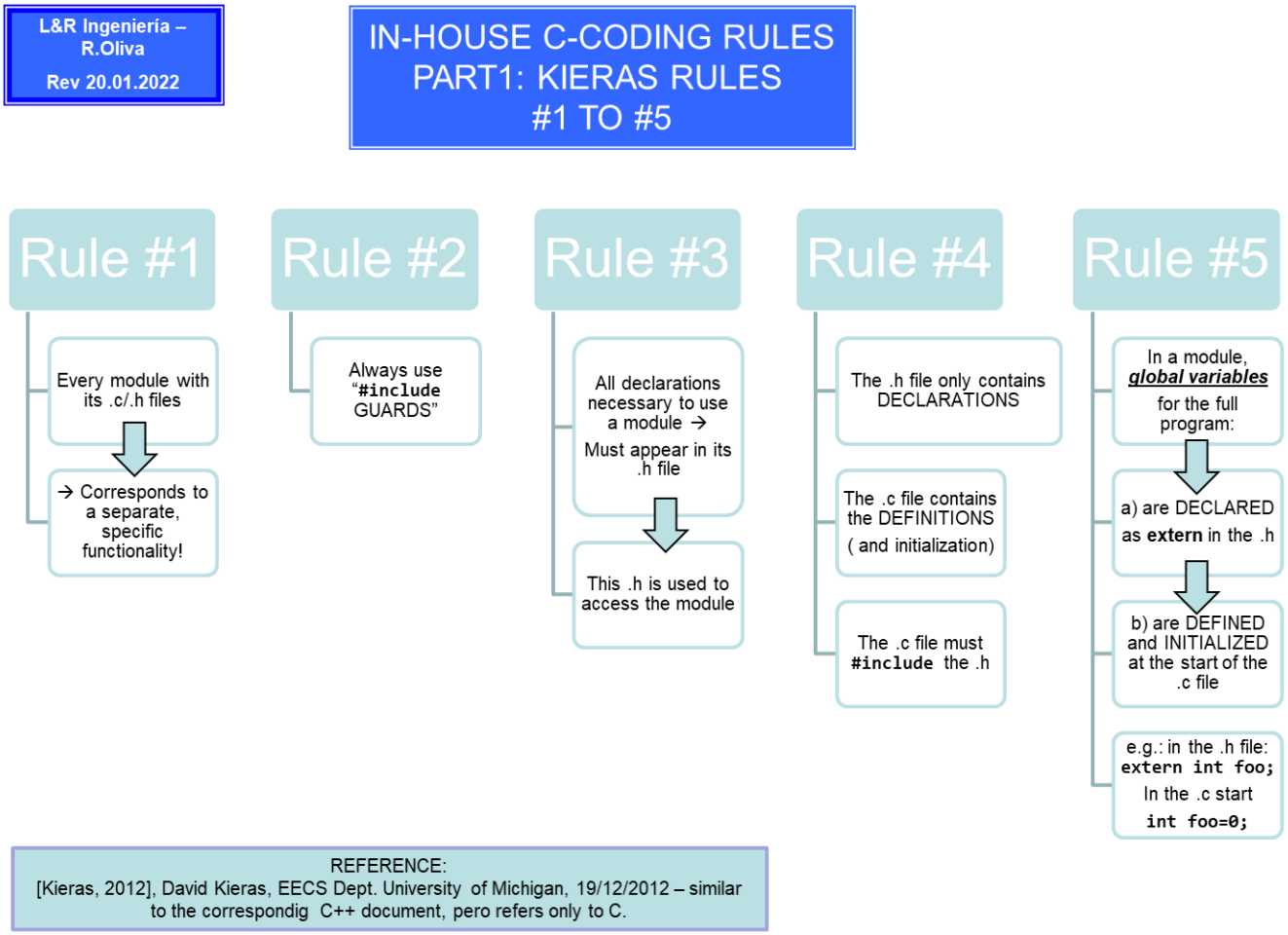

Figure 1 – Graphical representation of rules #1 to #5 by Kieras

Rule #6. Keep a module's internal declarations out of the header file. Sometimes a module uses strictly internal components that are not supposed to be accessed by other modules. If you need structure declarations, global variables, or functions that are used only in the code in the .c file, put their definitions or declarations near the top of the .c file and do not mention them in the .h file.
Furthermore, declare globals and functions `static` in the .c file to give them internal linkage. This way, other modules do not (and can not) know about these declarations, globals, or functions that are internal to the module. The internal linkage resulting from the static declaration will enable the linker to help you enforce your design decision.

Rule #7. Every header file A.h should #include every other header file that A.h requires to compile correctly, but no more. What is needed in A.h: If another structure type X is used as a member variable of a structure type A, then you must `#include X.h` in A.h so that the compiler knows how large the X member is. Do not include header files that only the .c file code needs.
E.g. `<math.h>` is usually needed only by the function definitions – `#include` it in .c file, not in the .h file.

Rule #8. If an incomplete declaration of a structure type X will do, use it instead of `#including` its header X.h. If a struct type X appears only as a pointer type in a structure declaration or its functions, and the code in the header file does not attempt to access any member variables of X, then you should not #include X.h, but instead make an incomplete declaration of X (also called a "forward" declaration) before the first use of X.

Rule #9. The content of a header file should compile correctly by itself. A header file should explicitly #include or forward declare everything it needs. Failure to observe this rule can result in very puzzling errors when other header files or `#includes` in other files are changed. Check your headers by compiling (by itself) a test.c that contains nothing more than `#include "A.h".`It should not produce any compilation errors. If it does, then something has been left out - something else needs to be included or forward declared. Test all the headers in a project by starting at the bottom of the include hierarchy and work your way to the top.
This will help to find and eliminate any accidental dependencies between header files.

Rule #10. The A.c file should first #include its A.h file, and then any other headers required for its code. Always `#include` A.h first to avoid hiding anything it is missing that gets included by other .h files. Then, if A's implementation code uses X, explicitly `#include` X.h in A.c, so that A.c is not dependent on X.h accidentally being `#included` somewhere else. There is no clear consensus on whether A.c should also #include header files that A.h has already included. Two suggestions:
• If the X.h file is a logically unavoidable requirement for the declaration in A.h to compile, then `#including` it in A.c is redundant, since it is guaranteed to be included by A.h. So it is OK to not `#include` X.h in A.c.
• Always `#including` X.h in A.c is a way of making it clear to the reader that we are using X, and helps make sure that X's declarations are available even if the contents of A.h changes due to the design changes. E.g. maybe we had a struct Thing member of a struct at first, then got rid of it, but still used Things in the implementation code. The `#include` of Thing.h saves us a compile failure. So it is OK to redundantly `#include` X.h in A.c. Of course, if X becomes completely unnecessary, all of the `#includes` of X.h should be removed.

Rule #11. Never `#include` a .c file for any reason! This happens occasionally and it is always a mess. Why does it happen? Sometimes you need to bring in a bunch of code that really should to be shared between .c files for ease of maintenance, so you put it in a file by itself. Because the code does not consist of "normal" declarations or definitions, you know that putting it in a .h file is misleading, so you are tempted to call it a ".c" file instead, and then write `#include "stuff.c"`. If it can't be treated like a normal header or source file, don't name it like one!
If you think you need to do something like this, first make sure that there isn't a more normal way to share the code (such as simply creating another module). If not, then name the special `#include` file with a different extension like ".inc" or ".inl".

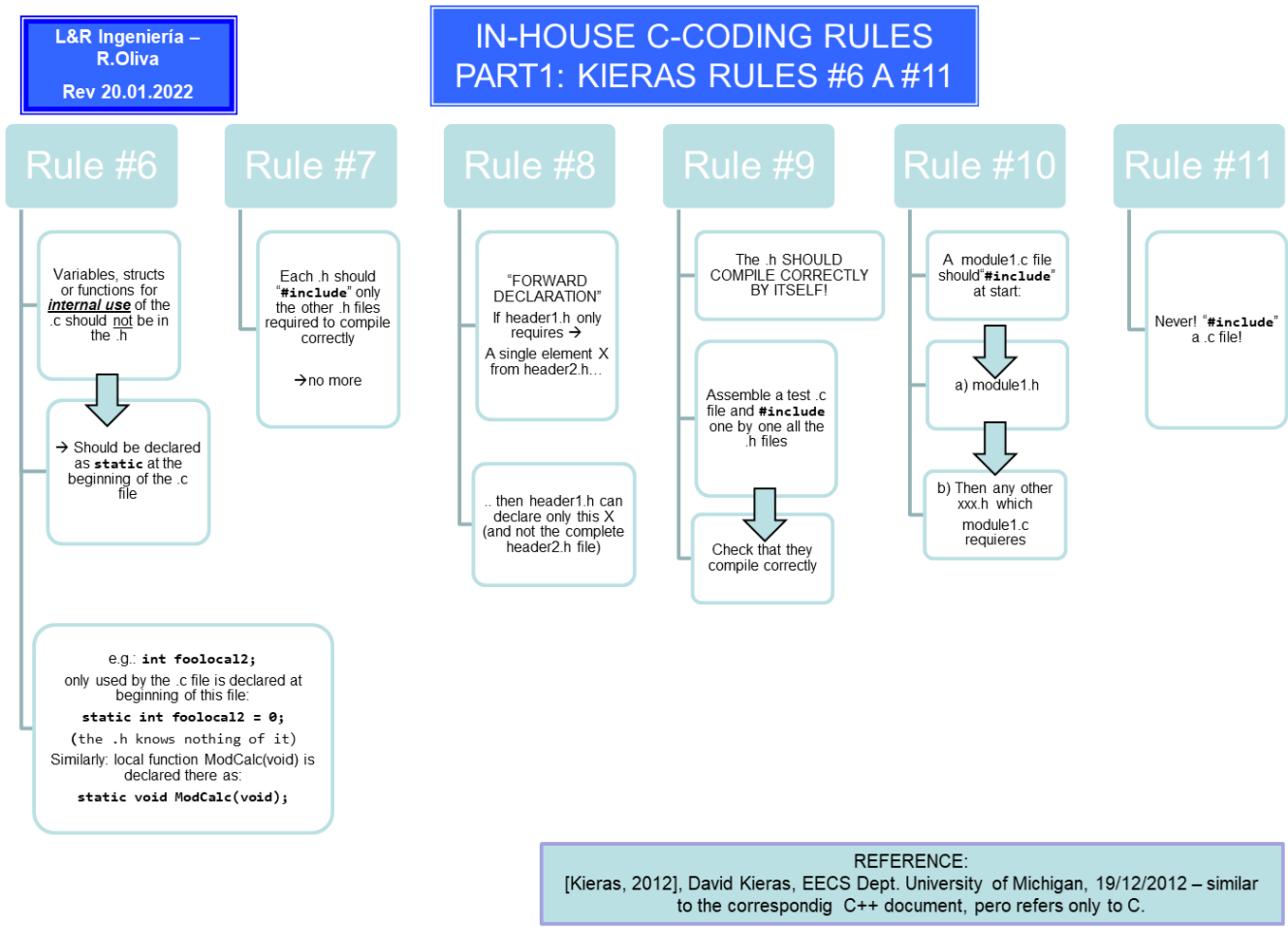

Figure 2 – Graphical representation of rules #6 to #11 by Kieras

Rule #12. No identifier should be longer than 31 characters: This is usually required since compilers usually enforce significance of only the first 31 characters for any variable, procedure or function name. In [ref_MISRA,2012] this is enforced in Rule 5.1 and also in [ref3_Barr,2018].

Rule #13. Type naming
13.1 - New data types (structures, unions and enumerations) shall be named via a typedef.
13.2 -Type names should (at least) start with an upper-case letter, consist of characters and internal underscores and end with '_t'. Note that '_t' is a convention – it isn't reserved or noted by the compiler, only inserted for clarity, following C99 criteria as in stdint.h.
13.3 - If public, these names should start with the module name and underscore.

This is a combination of C++ and C common practices. In C++ it is common use the convention to start type names with an uppercase, for example, MyNewType_t (as opposed to the C convention my_new_type_t). Barr in [ref3_Barr, 2018] focuses on the second option, and specifies only lower case.
e.g. For structures, in a PWRC2 system on CL2b board, the analog.h module would contain:

```
typedef struct fsample_holder {
    float32_t FV_Iaero;     // Current in EU - Sampled
    float32_t FV_Vbat;      // Bat. Voltage in EU - Sampled
```

```
    float32_t FV_Power;     // Power calculated in EU {W}
    float32_t FV_OWind;     // OutDoor WindSpeed [m/s] from METEO / COM1
    float32_t FV_OWDir;     // OutDoor WindDirection  [º] from METEO / COM1
    float32_t FV_OTemp;     // External Temp read from METEO+NOMAD2/COM1 in {°c}
    float32_t FV_OBaro;     // Barometric Pressure read from METEO+NOMAD2/COM1 in {mB}
    float32_t FV_RPM;       // RPM added
} Analog_FSHolder_t;
```

The name `fsample_holder` is redundant, but is kept for clarity. In the same PWRC2 system on CL2b board, the analog.c module would contain:

```
// Struct declaration in RAM - assign space to it.
Analog_FSHolder_t Analog_FS;
```

Rule #14. Procedure or function naming:

14.1 Modulename_action () is the preferred naming method for global functions (can be all capitals if usual for specific naming, for example RTC_ in /rtc hi level modules). For drivers, Modulename will usually be all capitals (example `void ADC_Init(uint8_t ADC_param);`

14.2 Camelcase naming for general functions

In general the noun-verb ordering [Samek, 2013] or modulename_action criteria [Barr, 2018] [Micrium, 2000] is the selected method for public global procedures/functions, and the discrepancies are regarding the use of (I) lower case characters+underscore (pure C, [Barr 2018]) or (II) capital letter first for module name followed by underscore for global functions or member functions of C-coded classes, and camelcase naming for general functions [Samek,2013].

Rule #15. Variable naming:

15.1 No use of restricted K&R, C99 names, follow Rule #12 (< 31 characters long), no underscores at start

15.2 Global variables start with Modulename_ and are declared (at start of .c module file) and initialized

As shown before, in the analog.c module we would declare variable Analog_FS as:

```
// Struct declaration in RAM - assign space to it.
Analog_FSHolder_t Analog_FS;
```

15.3 Local variables usually camelCase, if used within a module file should be declared as static

15.4 Constants – numeric macros or enumerations will be in upper case with underscores (e.g. FOO_BAR), if global prefixed with module name (ANALOG_FOO_BAR)

15.5 For counters of limited scope i, j,k and local declaration is allowed.

Rule #16. Module file and directory naming:

16.1 Module file names (.h and .c) shall consist of lowercase letters, numbers and underscores, first 8 characters should be unique for portability

16.2 No reserved (C or C++ standard libraries) names like stdio or math are allowed

16.3 Any module containing a main() function shal have the word main as part to its source file name.

This Rule follows naming conventions in part 4 of [Barr 2018], and is based on problems of portability with mixed case names and directories, and error-prone for programmers. The inclusion of “main” in a file name has proven useful in projects with multiple software configurations.

Rule #17. Equivalence Tests:

17.1 When evaluating the equality of a variable against a constant, the constant shall always be placed to the left of the equal-to operator (==).

Example:

```
// uses /modules/sdfile
if (SYS_FALSEVAL == SYS.FlagNoSD){
    printf("\n\rCheck Event SD File...\n\r");
    // in /modules/sdfile
    FileSD_checkOrCreateEventFile();
}
```

This Rule follows reasoning in part 8.6 of [ref3_Barr, 2018], and eases detection of possible typos and as many other coding defects as possible at compile-time. Defect discovery in later phases is not guaranteed and often also more costly. By following this rule, any compiler will reliably detect erroneous attempts to assign (i.e., = instead of ==) a new value to a constant.

5. References

[ref0_C99,1999] "Programming Languages - C (Formerly ANSI/ISO/IEC 9899-1999)"
https://webstore.ansi.org/standards/incits/incitsisoiec98991999r2005
[ref1_CL2b, 2010] CL2b board and interfaces: https://www.lyr-ing.com/Embedded/LyRAVR_CyEn.htm
[ref2_CL3, 2018] CL3 board: https://www.lyr-ing.com/Embedded/LyRCl3%2BM5E_En.htm
[ref3_Barr, 2018] Barr, Michael "Embedded C Coding Standard"
https://barrgroup.com/sites/default/files/barr_c_coding_standard_2018.pdf
[ref4_Kieras, 2012] David Kieras, "C Header File Guidelines"
http://websites.umich.edu/~eecs381/handouts/CHeaderFileGuidelines.pdf
[ref5_Kieras,2015] David Kieras, "C++ Header File Guidelines"
http://umich.edu/~eecs381/handouts/CppHeaderFileGuidelines.pdf
[ref6_Samek,2013], Miro Samek, Quantum Leaps Application Note "C/C++ Coding Standard" Rev J (2013) https://www.state-machine.com/doc/AN_QL_Coding_Standard.pdf
[ref7_C99] C99 WG14 Committee Draft http://www.open-std.org/jtc1/sc22/open/n2794/
[ref8_MicriumAN2000B,2000] https://d1.amobbs.com/bbs_upload782111/files_38/ourdev_630682QLHUQ2.pdf
[ref9_JLabrosse,2000] Jean Labrosse, "Embedded Systems Building Blocks, 2ed" R&D Books, 2000, ISBN 0-87930-604-1
[ref10_PontEmbC,2002] Michael J.Pont, "Embedded C" Pearson Ed., 2002, ISBN 0 201 79523 X (https://www.safetty.net/publications/embedded-c)
[ref11_PontERES2,2017] Michael J.Pont, "The Engineering of Reliable Embedded Systems, 2ed" SafeTTy Systems Ed., 2017, ISBN 978-0-9930355-4-8 (https://www.safetty.net/publications/the-engineering-of-reliable-embedded-systems-second-edition)
[ref12_GrenningTDD,2011], James W. Grenning "Test-Driven Development for Embedded C", The Pragmatic Bookshelf Ed. 2011, ISBN 9781934356623
[ref-odr1,2020] https://stackoverflow.com/questions/60085469/why-does-the-one-definition-rule-exist-in-c-c
[ref-odr2, 2014] "DCL60-CPP. Obey the one-definition rule"
https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL60-CPP.+Obey+the+one-definition+rule
[ref-CVAVR,2022] HP InfoTech / Codevision AVR C Compiler http://www.hpinfotech.ro/cvavr-features.html
[ref-ArduinoIDE,2021] Arduino IDE https://www.arduino.cc/en/software
[ref-VSC,2022] Microsoft Visual Studio Code https://code.visualstudio.com/
[ref-PIO,2022] Platform IO https://platformio.org/
[ref-AVRgcc,2021] AVR GCC compiler: from https://blog.zakkemble.net/avr-gcc-builds/ or with IDE, now Microchip Studio at https://www.microchip.com/en-us/tools-resources/develop/microchip-studio
[ref-PSoCDes,2022] https://www.infineon.com/cms/en/design-support/tools/sdk/psoc-software/psoc-designer/
[ref-MISRA-C, 2012] "MISRA C:2012 Guidelines for the use of the C language in critical systems," MISRA, March 2013 https://www.misra.org.uk/product/misra-c2012-third-edition-first-revision/

Teófilo de Loqui 58 (fdo)
9400 Rio Gallegos
Santa Cruz
ARGENTINA

L&R INGENIERIA
https://www.lyr-ing.com
https://www.lyringenieria.com.ar

TE: +54 (0) 2966 430923
e-mail: roliva@lyr-ing.com
roliva@lyringenieria.com.ar

NOTES: