

## **Suggestions for Coding Guidelines for "C"**

The programming language "C" is described in "The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie and also defined in ISO/IEC 9899:1990 "Programming languages: C" (English version EN 29899;1993). It is a very powerful language. Nevertheless IEC 61508, "Functional safety of E/E/PE safety related systems", does not recommend "C" in higher safety integrity levels (SIL 3 and SIL 4).

To overcome some of the known problems, programming rules must be established.

### **Important Note:**

- This is no recommendation to use "C" !
- The presented Guidelines are only examples.
- Known compiler failures have to be considered additionally.
- Take the unspecified, undefined and implementation defined behaviour into account (annex G1-G3 of ISO/IEC 9899)
- Every exception to the rules must be explained in the source code

### **Here are some suggested rules:**

#### **General requirements**

- Every exception in the rules must be explained in the source code
- No command line options, if possible use utilities like the program "make" to compile.
- Measurement of complexity metrics (e.g. Halstead, McCabe)
- Use of a static analysis tool
- Definition of maximum of encapsulated (nested) blocks (e.g. 7)
- No unreachable code (dead code)  
Bad examples:

```
if ( 7 == 4) {...}          /* block will never be reached */
for (i=1; i>x, i<100; i++) /* because of the comma operator, the
.....                    condition "i>x" has no influence on end
                           expression of the FOR loop */
```
- Only one statement per line
- Maximum of 1000 statements per file
- Absolute minimum use of the preprocessor
- No command line options, if possible use utilities like the program "make" to compile files or programs.
- On the following topics individual rules must be compiled:
  - Nomenclature:
    - unified definitions
    - use of underscore "\_" within identifiers
    - define own rules ; use common and practice-proofed guidelines and conventions
  - use of includes
  - use of global definitions
  - use of shift operations
  - use of bit operations
  - use of macros

#### **Control flow**

- No "GOTO"
- No "BREAK" in "FOR", "DO", "WHILE" loops
- No "CONTINUE" in "FOR", "DO", "WHILE" loops "DEFAULT" block mandatory in "SWITCH"

### Automation, Software and Information Technology

- Every branch in a "SWITCH" has to be finished by precisely one "BREAK"
- No nested "SWITCH"
- Use of curly braces {} to enclose the contents of IF-, ELSE-, FOR-, WHILE- and DO WHILE- statements, even though these braces are not required by the C language

Bad example:

```
if (condition1)
for(i = 0; i < n; i++)
if (condition2) {...}
else /* "else" belongs to "if (condition2)" rather than to "if
(condition1)" */
.....
```

- Floating point comparison for equality and also inequality and non-integral loop variables

Bad example:

```
float a, b;
...
if ( a == b) {...}
else {...}
/* calculation on a, b */
/* The result is not predictable in advance
and may differ */
/* from machine to machine*/
```

- No assignments or changes to loop variables of FOR loops

Bad example:

```
for (x=0 ; x<100; x++)
{ ...
x=x+1;
...}
```

- No use of loop variables outside a FOR loop

Bad example:

```
for (x=0; x<100; x++)
{...}
...
a[x]=1;
...
```

- No recursion

### Variables and constants

- No unused variables
- Explicit and complete declaration of types  
Example: signed char, unsigned char, signed long int, unsigned short int, ...
- Separation of declaration and initial assignment
- Only one declaration per line
- No redefinition (local overwrite) of types and variables

### Functions

- No unused functions
- Maximum of 7 parameters per function
- Definition of function prototype, whereby the number and type of arguments can be evaluated by the compiler
- No side effects in function arguments

Bad example:

```
void f(int i, int j)
{...}
...
f(n++, n); /* Argument evaluating from left to right ? Performs the
/* post-increment on the first argument, n++, before */
/* evaluating the second argument ? */
```

### Automation, Software and Information Technology

- Maximum of 100 statements per function
- Only one declaration part for types and variables per function

Bad example :

```
void function x (void)
{
  {int a ;
  ... }
  {float a;
  ... }
}
```

- Only one "RETURN" as last statement of functions

### Assignments

- No assignments within conditions

Bad examples:

"if (x=y)"; usually "if (x==y)" is meant by the programmer  
 if ((x==y) || ((y--) == 0)); execution of "(y--)" depends on the result of "x==y"

"while (x++ <= 15)"; really test on "(x+1) <= 15" or "x <= 15" followed by "x++"

- No multiple assignments (a=b=c)
- No conditional expressions (?:)

Bad example:

```
z=(x>y) ? x : y ;
```

- No coupled operations ("+=", "\*=", etc.)
- No nested assignments
- No implicit conversion of types
- Use parentheses in complex assignments

Bad example:

```
a * = b + 1; /* does it means 'a = a * b + 1' or 'a = a * (b + 1)' */
```

### Pointers

- Minimize the use of pointers (pointers have a shaky reputation in any language, the use of pointers is almost completely uninhibited by the language definition itself)
- Explicit declaration and initializations of pointers
- Conversion of pointers are not allowed, except conversion of "/" to the zero-pointer

Bad example :

```
int i, *p_i;
char c, *p_c;
char memblock[] = {'qwertzu'};
.....
p_c = memblock;          /* point to q in memblock */
p_c ++;                  /* point to w in memblock */
p_i = (int *) c_p;       /* Cast the pointer */
i = *p_i;                /* What goes into i? The content of i vary
                           dramatically */
/* from machine to machine and can simply crashes */
/* because int-addresses are supposed to be aligned on a */
/* word boundary */
```

- Dynamically memory allocation (malloc, calloc) and reallocation (free) should be avoided

---

**Automation, Software and Information Technology**

Evaluation order and Precedence must be considered separately for the correct evaluation of expressions. Some orders on evaluating operands can be still undefined, also if parentheses are used.

The sequence points of C laid down in the Standard are the following:

- The point of calling a function, after evaluating its arguments.
- The end of the first operand of the && operator.
- The end of the first operand of the || operator.
- The end of the first operand of the ?: conditional operator.
- The end of the each operand of the comma operator.
- Completing the evaluation of a full expression. They are the following:
  - Evaluating the initializer of an auto object.
  - The expression in an 'ordinary' statement — an expression followed by semicolon
  - The controlling expressions in do, while, if, switch or for statements.
  - The other two expressions in a for statement.
  - The expression in a return statement.

Examples:

```
a = 4 + 5 * 7;           // a = 39, * done before +
a = 4 + (5 * 7);        // a = 39, order explicit
a = (4 + 5) * 7;        // a = 63, order overridden
a = b * (c() + d());    // * done before +, however order of c() and d()
                        // is unspecified
a = b++ + c[b];         // Order in which operands are evaluated is
                        // unspecified
a = (b++) + c[b];       // Order is still unspecified
```

### Suggested Reading

- Safer "C", Les Hatton, McGraw-Hill Book Company, ISBN 0-07-707640-0
- "C", A Reference Manual, Samuel P. Harbison, Guy L. Steele Jr., Prentice Hall Software Series, ISBN 0-13-110933-2
- Guidelines for the use of C Language in vehicle based software, MISRA, ISBN 0 9524156 9 0