emC Usage of `#define`

`#define` in C has 2 intension:
1) controlling of parts in sources to compile (compile switch)
2) text replacement

First intension:

```c
#define DEF_REFLECTION_NO
```

```c
#ifdef DEF_REFLECTION_FULL
    #include <emC/Base/genRefl/Time_emC.crefl>
#endif
```

Decision of usage outside of the concrete source, may be as compiler cmd argument

Different capabilities with unchanged source for different usage approaches

=> use a capability or not.
emC Usage of \#define

\#define in C has 2 intension:

1) controlling of parts in sources to compile (compile switch)
2) text replacement

First and second intension:

```
#define initReflection_ObjectJc(THIZ, ADDR, SIZE, REFL, IDENT)  
{ (THIZ)->idInstanceType = ((IDENT)<<16) + ((REFL)->ixType & 0xffff); }
#define initReflection_ObjectJc(THIZ, ADDR, SIZE, REFL, IDENT) 
{ (THIZ)->idInstanceType = ((IDENT)<<16); }
```

```
void ctor_Clock_MinMaxTime_emC(Clock_MinMaxTime_emC* thiz, int nrofEntries) {
initReflection_ObjectJc(&thiz->base.object, thiz, sizeof(*thiz), 
, &reflection_Clock_MinMaxTime_emC, 0);
```

Explicit error message if a decision is not possible in the given context
Different perculiarity
with unchanged – same sources
emC Usage of \#define

\#define in C has 2 intension:

1) controlling of parts in sources to compile (compile switch)
2) text replacement

Second intension: text replacement in style of an operation

Use paranthesis around arguments (!)

\#define ARRAYLEN_emC(ARRAY) (sizeof(ARRAY) / sizeof((ARRAY)[0]))

......
//usage:
int myArray[] = { 1,2,3 };  
int size = ARRAYLEN_emC(myArray);

* It is simple able to read and clarified
* An inline operation is not possible for that approach
* Do not count arguments in the source and use immediately numbers:
  int size = 3;
emC Usage of `#define`

**Negative and positive pattern of `#define`**

```c
#ifdef PLATFORM_A
  dosomething(...);
  ...
#else if PLATFORM_B
  #if DEF_XY
    doitother();
    ...
  #else
    thirdVariant();
    ...
#endif
#endif
```

- Different implementations, not the same concepts or slightly different
- Any platform has its own intension, different
  => Only (de)selecting capabilities
- Too many variants
  => It should have a concept
- Nesting is bad

```c
#ifdef DEF_USE_INSPECTOR
  init_Inspc(...);
  ...
#endif
```

- A clear decision
- The adequate clear content

www.vishia.org/Cj – C für embedded - Thesen
emC Usage of `#define`

**Writing style of `#define`**

```c
#include DEF_REFLECTION_FULL

#define add_MyType(A,B) ((A) + (B))
inline int add_MyType(int a, int b)
{ return a+b; }
```

- Compiler switches should be start with DEF and upper case written. Do not use `__DEF__` because gcc says „reserved keyword“
- Mark the associated endif
- Use paranthesis around arguments (!)
- Usage a macro as operation: It is possible, advantage: type-variable.
- Then write macro name in normal camelCase, it is an operation!
  Alternativly to an inline operation.

Advantage of inline: Better compiler error detection (inline in C since C99).
Advantage of macro: type-invariant, ignore arguments in special cases ...etc.

A macro should be well tested. Problems with usage possible.
A macro should not be too complex. It should be comprehensible.
emC Writing style in header

struct and class definition

Use typedef for C language

typedef struct MyType_T {
    /**Comment to element*/
    int32 val1;
    float val2;
    OtherType_s* aggregation;
} MyType_s;

int anyOperation_MyType(MyType_s* thiz, float arg);

#ifdef __cplusplus
class MyType : public MyType_s {
    int anyOperation(float arg) {
        anyOperation(this, arg);
    }
    ...
};
#else

offer the C function as class function.

Write the struct MyType_T with _T, using for forward declaration.

Write the C MyType_s with _s

Declare function prototypes accordingly to the data type, write „thiz“

Offer a class for C++ usage, it is better to handle, but with __cplusplus compilation condition => possible fall back to C for some usages.

offer the C function as class function.
C or C++, the question

Though C++ is available for the most embedded processors (for all) since 20 years more and more better, and C++ has taken a tutorial development
=> Some or many people attached to C. Why? Are they to old or stupid?

C is near to machine code.
A simple C++ is near to machine code too.
C is often used as meta language for code generation from graphical models.
Some or all C++ libraries are using dynamic memory.
It is worse for embedded, worse for safety long-running devices.
C++ is a language for PC programming and graphical applications (QT...),
The growth of C++ programs are for PC usage.
C++ with dynamic memory libraries has not so far experience in embedded
What about safety of a virtual table pointer inside the data for safety critical apps?

The discussion C vs C++ or a lightweight C++ for embedded is up to date.