1-Page Overview Most Important Codingrules for Driver Assistance

"Programming is the art of telling another human being what one wants the computer to do." (Donald Knuth)
"...in programming simplicity and clarity – in short: what mathematicians call elegance –
are not a dispensable luxury, but a crucial matter that decides between success and failure" (Edsger Dijkstra)

Header.hpp:

Most important rule:
Write code for humans, not for machines!
Your code will be read and used many times by many people (including your future self!). So write clean code for them!

Variable names, function names and class names need to be meaningful enough to let every developer understand the code fast (**no abbreviations in names**).

Write code which is as self-documenting as possible.
It should be possible to understand a piece of code just by itself.

Fix all compiler warnings. Compiler warnings contain valuable information. Learn to understand the warning and fix it in a clean way. Do not cheat the compiler. Keep the code clean!

Use comments to describe the *intent* of your code. Describe the high-level ideas which are not obvious out of the code. **Keep comments up to date, remove outdated code and comments.**

Avoid global variables. Encapsulate variables as much as possible in classes and namespaces.

Order class content by access level in this order:
public ⇨ protected ⇨ private

Use spaces, not tabs (3 spaces per level) ⇨ Easier compare if it's always the same. Do NOT indent contents of namespaces, do NOT indent access modifiers (public...).

Source.cpp:

Limit the size of methods:
Each method should do one thing and only one thing.
The code of a method should fit easily (unzoomed) on one screen.

Limit code to 110 characters per line (for easy compare).

Implement methods in source files, not in headers
(exception: setters/getters).

Check for Misra compliance:
Remove all Misra-Warnings > Level 2.
Always fix the warning if it is technically possible to fix it.
It is only acceptable to deactivate a warning when fixing it would negatively affect files of other components / teams or when it is technically impossible to fix it.

Check for HIS metrics:
Reduce cylomatic complexity (STCYC) to <= 20 for all methods.
It is only acceptable to have STCYC>20 when it is due to extensive switch-case that is not nested. **Nested switch-case / if-then-else should be disassembled into multiple separate methods.**

```
#ifndef SERVICEFUNCTIONS_HPP_INCLUDED
#define SERVICEFUNCTIONS_HPP_INCLUDED
namespace ServiceFunctions
{
enum class ServiceEnums
{
   serviceA = 0, ///< Short inline description of serviceA
   serviceB,     ///< Short inline description of serviceB
   serviceC      ///< Short inline description of serviceC
};

/// This is a short description of the following class.
///
/// More detailed description. This should include a high-level
/// description of functionality and motivation of the class. You should
/// also explain its intended usage. Giving code examples can be a
/// good idea. You can wrap code segments like this:
/// \code
///     vfc::int32_t x = 1;
///     vfc::int32_t y = x + 2;
/// \endcode
/// Then, doxygen will generate a nice html output of this code.
class ServiceClass
{
public:
   explicit ServiceClass(vfc::int32_t x) : counter1(x)
   {
      // NOT: counter1 = x;
   }

   vfc::int32_t getServiceBValue() const
   {
      return static_cast<vfc::int32_t>(ServiceEnums::serviceB);
   }

protected:
   vfc::int32_t start(vfc::int32_t myparam1, vfc::int32_t myparam2);

private:
   vfc::int32_t counter1; ///< Short inline comment for member counter1
   static vfc::int32_t counter2; // NOT: vfc::int32_t counter1, counter2;
};

}  // end namespace 'ServiceFunctions'
#endif // SERVICEFUNCTIONS_HPP_INCLUDED
```

Always add include-protection to header-files.

Avoid #defines and macros (only use for include protection). Instead use (static) const values, enums or inline functions (to replace macro functions) where possible. Never #undef a macro, never define TRUE / FALSE.

Only use the new strongly typed enums (enum class). Use an explicit cast if you want to access the value of an enum.

Only use C++ style comments:
// or /// : /// and ///< for inline comments will be visible in doxygen documentation.
Avoid /* ... */ where possible.
Always document in English.

Use doxygen comments to document classes and their use in header-files.
Also use doxygen to document types, variables, members, etc. in header-files.

Use member initializer lists in constructors.

Use the explicit keyword for constructors with only one parameter. This forbids to use ServiceClass a = 3 but requires ServiceClass a(3).

Declare a method as const if it doesn't modify the object (avoid non-const getter methods). **Declare variables as const when they are not supposed to be changed after initialization.**

Use only one declaration per line.

```
#include <servicefunctions.hpp>

namespace ServiceFunctions
{
/// This is a doxygen comment for a method.
/// \param myparam1  Description of myparam1.
/// \param myparam2  Description of myparam2.
/// \returns Describe what the method returns.
vfc::int32_t ServiceClass::start(vfc::int32_t param1, vfc::int32_t param2)
{
   MyClass* pointerToObject;      // NOT: MyClass *pointerToObject
   MyClass& referenceToObject;    // NOT: MyClass &referenceToObject
   vfc::int32_t x = param1 + 3;   // NOT: vfc::int32_t x=param1+3;

   if (param1 != param2)          // NOT: if ( param1 != param2 ),
                                  // NOT: if(...) , if (param1!=param2)
   {
      doThisAndThat(param1);
   }

   for (vfc::uint32_t i = 0; i < N; i++) // NOT: for(...), and NOT:
                                         // for (vfc::uint32_t i=0;i<N;i++)
   {
      // ...
   }

   // This is a comment which spans over more than one line.
   // Here comes the second line. We describe what the next code lines do.
   vfc::int32_t y = 2 * (param2 + param1);

   return x + y;
}
vfc::int32_t ServiceClass::counter2 = 42;
}  // end namespace 'ServiceFunctions'
```

Keep the size of files in check:
1 class per file where reasonable.

Minimize includes:
If an include is only needed in the source code, don't add it to its header file!

Use doxygen comments to document functions and methods in source files. **Describe all parameters and return values.** Use \command, not @command .

Do not use (unsigned) short / int / long or float / double, **use vfc-types** instead (bool / void is ok).

Check plausibility of input parameters before using them (if they come from outside context).

Don't use struct if you need any logic to work on this data.
Use a class instead and add the logic for the data to the class.

Put curly braces { and } in their own lines. **ALWAYS use curly braces, NEVER omit them** (e.g. don't omit them for 1-liners).

Try to avoid non-const static variables. If you have to use them make them multicore-safe!

Explicitly define initial values of all static variables (that have been declared in .hpp files) in .cpp files.