

Work Instruction

Project

Software Engineering


Topic

Coding Styles

Revision

5.1

Approval



Signature:

Dr. P. Geiselhart / E / 22.08.2005



Signature:

P. Vollmer / SPPQ / 22.08.2005

Table of Contents

1	Preface	5
1.1	Purpose	5
1.2	Validity and Audience	5
1.3	Lifetime	5
1.4	Changes	5
1.5	Acknowledgment	5
2	Introduction and Usage	6
2.1	Monitoring of the Rule Usage	6
2.2	Applicability of Rules	6
2.2.1	Existing Code	6
2.2.2	Warning Elimination	6
2.2.3	Generated Code	7
2.2.4	External Code	7
2.3	Rule Structure	7
2.4	Rule Priorities and Violation	7
2.5	Tailoring of Rules	8
3	Definitions	9
3.1	Accessor Method	9
3.2	Helper Class	9
3.3	Interface Class	9
3.4	Component	9
3.5	Package	9
3.6	Complex Expressions	9
3.7	Effectively Boolean Expression	9
4	Harman/Becker Rules	10
4.1	File Structure	10
	HB01 English Language	10
	HB02 Headers and Includes (C, C++)	10
	HB03 Location of Declarations (C, C++)	11
	HB04 Classes per File (C++)	11
	HB05 Class and Function Size (C, C++)	11
	HB06 Code Grouping (C, C++)	12
	HB07 Instructions per Line (C, C++)	12
	HB08 Blanks and Indentation (C, C++)	12
	HB09 Braces (C, C++)	13
4.2	Code Documentation	14
	HB10 Code Comments (C, C++)	14

HB11 File Header (C, C++).....	15
HB12 Interface Documentation (C, C++)	15
HB13 Component Documentation (C, C++)	16
4.3 Naming Conventions.....	17
HB14 File Names (C, C++)	17
HB15 File Suffixes (C, C++).....	17
HB16 Naming (C, C++).....	18
HB17 Underscore Character (C, C++).....	19
HB18 Name Prefixes (C, C++).....	19
4.4 Preprocessor-related Rules.....	20
HB19 Includes (C, C++).....	20
HB20 Conditional Compilation (C, C++)	20
HB21 Pragmas (C, C++).....	21
HB22 Preprocessor Defines (C++)	21
HB23 Macro Definition (C, C++)	21
4.5 Miscellaneous Rules	22
HB24 Intrinsic Types (C, C++, Embedded Systems).....	22
HB25 Null Pointers (C, C++).....	23
HB26 Const Classifier (C, C++)	23
HB27 Assertions (C, C++)	23
HB28 Array Lengths (C, C++).....	24
4.6 C++ specific Rules	24
HB29 Orthodox Canonical Form (C++).....	24
HB30 Namespaces (C++).....	25
HB31 Reference Parameter (C++)	25
HB32 Casts (C++).....	26
HB33 Inline Methods (C++)	26
HB34 Explicit Constructors (C++)	26
HB35 Inheritance (C++)	27
HB36 Code Bloat Prevention (C++, Embedded Systems).....	27
5 MISRA Rules	28
5.1 MISRA Rule List.....	28
M1 Environment.....	28
M2 Language Extensions	29
M3 Documentation.....	29
M4 Character Sets.....	29
M5 Identifiers	29
M6 Types	30
M7 Constants.....	30
M8 Declarations and Definitions	30
M9 Initialisation	31
M10 Arithmetic Type Conversions	31
M11 Pointer Type Conversions.....	32

M12 Expressions	32
M13 Control Statement Expressions	33
M14 Control Flow	33
M15 Switch Statements	35
M16 Functions	36
M17 Pointers and Arrays	37
M18 Structures and Unions	37
M19 Preprocessing Directives	37
M20 Standard Libraries.....	38
M21 Run-time Failures.....	39
5.2 Compliance Table	40
6 Effective C++ Rules	41
E01-E04 Effective C++: Shifting From C to C++	41
E05-E10 Effective C++: Memory Management.....	41
E11-E17 Effective C++: Constructors, Destructors and Assignment Operators	41
E18-E28 Effective C++: Classes and Functions: Design and Declaration.	41
E29-E34 Effective C++: Classes and Functions: Implementation.....	42
E35-E44 Effective C++: Inheritance and Object-Oriented Design	42
E45-E50 Effective C++: Miscellany.....	42
EE01-EE05 More Effective C++: Basics	43
EE06-EE08 More Effective C++: Operators.....	43
EE09-EE15 More Effective C++: Exceptions	43
EE16-EE24 More Effective C++: Efficiency	43
EE25-EE31 More Effective C++: Techniques	44
EE32-EE35 More Effective C++: Miscellany.....	44
7 Changes.....	45
7.1 Change History.....	45
7.2 Changes since Release 5.0	46
7.3 Changes since Release 4.6	47
8 Bibliography	48

1 Preface

1.1 Purpose

This document describes the Coding Styles for Software Engineering within Harman/Becker Automotive Systems GmbH with C and C++.

1.2 Validity and Audience

This document is valid for all software engineering activities in all locations of Harman/Becker Automotive Systems GmbH.

The intended audience includes software engineers using the C or C++ programming languages.

1.3 Lifetime

This document is valid until 2007-12-31 if not changed. Before that date, the Software Engineering Process Group shall start a committee that reviews this document and decides whether to extend the date of validity or to revise the contents.

1.4 Changes

All readers of this document are invited to send their experiences and change requests to the Software Engineering Process Group.

1.5 Acknowledgment

We would like to thank the reviewers and contributors to the release 5.1:

Dr. Frank Beeh, Markus Broghammer, Karl Bühler, Joachim Grill, Gerald Harris, Olaf Kleine, Steffen Lohse, Sven Scheible, Hartmut Schirmer, Henning Schröder, Markus Schwarz, Harald Wellmann.

We would further like to thank the authors of the previous releases and the reviewers and contributors to the release 5.0:

Dr. Frank Beeh, Andree Buschmann, Matthias Häusser, Gerald Harris, Christian Hillebrecht, Peter Jehle, Michael Kindel, Klaus Klee, Hans Leitenmeier, Leona Neufeld, Michael Rieck, Gerhard Rueß, Christian Schäfer, Hartmut Schirmer, Matthias Schulz, Markus Schwarz, Kurt Stege, Alfons Steier, Wolfgang Sitter, Prof. Dr. Tien Tran Manh.

2 Introduction and Usage

Programs shall be correct and easy to maintain. In order to reach these goals, the programs should have a consistent style, be easy to read and understand, be portable to other architectures, be free of common types of errors and be maintainable by different programmers.

Questions of design, such as how to design a class or a class hierarchy, are beyond the scope of this document. Recommended books on these subjects are indicated in the chapter entitled “References”.

In order to learn how to effectively deal with the most difficult aspects of C and C++, the examples of code that are provided should be carefully studied. C and especially C++ are difficult languages in which there may be a very fine line between a feature and a bug. Both languages allow a programmer to write compact and in some sense unreadable code. This places a large responsibility upon the programmer.

2.1 Monitoring of the Rule Usage

Adherence to these rules shall be checked by manual code reviews and code inspections, which have to be planned by the project leaders. For more information about inspections read the EE-Intranet at oekair02.becker.de/~sde/main/phpwiki/index.php?pagename=Inspections.

2.2 Applicability of Rules

In special situations, the applicability of this document may be restricted.

2.2.1 Existing Code

Code that existed before the date of approval does not have to be changed to conform to the new version of the coding styles.

If existing code is changed dramatically (a significant portion is changed), it is considered as new code and shall conform to the current coding styles.

If existing code has been designed for reuse (frameworks, libraries), it should be revised even if no other changes are made. This avoids confusion when this code is integrated with code that already uses the new coding styles.

2.2.2 Warning Elimination

The Zero Warnings Campaign aims at the systematic elimination of warnings issued by the target compiler or PC-Lint. This campaign should not affect the rules presented here. If there should ever be a conflict, the elimination of the

warnings has higher priority than any coding style. In such a case, please also inform the Software Engineering Process Group.

2.2.3 Generated Code

Generated code shall comply with these coding styles. If a code generator cannot be adapted in reasonable time or with reasonable effort in order to generate compliant code, then that generated code is not required to comply with these coding styles.

2.2.4 External Code

External code written specifically for Harman/Becker shall comply with these coding styles. Other external code is not required to comply.

2.3 Rule Structure

The rule definitions follow this scheme:

ID Title (Scope)

Rule description, possibly including `keywords`.

Additional comments, possibly including `code samples`.

Examples with `example code`.

The scope of a rule restricts the applicability to the given list of programming languages, application types or project types.

Comments to rules provide further explanation and clarification of special cases, references to other rules and standards and implementation proposals.

Examples for rules demonstrate the correct or incorrect use of these rules.

2.4 Rule Priorities and Violation

All rules shall only be violated with good reason and in exceptional cases. In such cases, the priority of rules defines the procedure to follow.

The priority of a rule is defined by the wording used in the rule description as follows.

A statement using “may” indicates an optional rule. An optional rule may be violated only if there is a good rationale.

A statement using “should” indicates a recommended rule. A recommended rule may be violated only if there is a good rationale and the rationale is documented in the code next to the violation.

A statement using “shall” indicates a mandatory rule. A mandatory rule may be violated only if there is a hard technical or cost-related rationale and the rationale is documented in the code next to the violation. Project-critical performance issues or otherwise irresolvable compatibility problems qualify as hard rationales.

2.5 Tailoring of Rules

Projects or product lines may define tailored guidelines, which then shall be documented in the quality assurance plan of the project or as a work instruction and shall be communicated to all developers concerned.

The tailored guidelines may select one of the alternatives which have been defined for some rules, may further narrow existing rules, may define additional rules which do not interfere with the existing ones, and may exclude single rules, given that rationales are provided that hold for the whole project or product line (see 2.4).

The project-specific coding styles should document these changes and can provide further information on how to fulfill specific rules. Project-specific coding styles shall be approved by the Software Project Architect.

The following rules require tailoring (examples are valid for Mocca V2):

- HB10 Location of the DoxyGen configuration files, containing e.g. the project header.
- HB14 Name convention for package prefixes / namespaces for the project.
- HB17 Name convention for the use of underscores in special prefixes for the project.
- HB24 Location of header defining intrinsic data types (e.g. `core/base/HBTypes.h`).
- HB27 Location of header defining the assert macro (e.g. `core/base/HBTrace.h`).
- HB28 Location of header defining the array length macro (e.g. `HBMacros.h`).

3 Definitions

This section defines terms that are not part of the usual C++ terminology.

3.1 Accessor Method

An Accessor Method, also known as “getter/setter”, encapsulates reading or writing access to a member variable and has little to no other functionality.

```
public:
    Int32 getSpeed() const
    {
        return m_speed;
    }
```

3.2 Helper Class

A class used to implement the details of another class. Helper classes are small (usually less than 60 lines of code) and not public.

Simple internal data structures (e.g. entries of a hash table) and algorithms (e.g. comparators) are typical Helper Classes.

3.3 Interface Class

An Interface Class is a class containing public or protected pure virtual methods and no member variables except constants.

3.4 Component

A component (also known as module) is an encapsulated subsystem with a defined interface. Specialized components can conform to a dedicated component environment such as CORBA or Mocca. Usually, a component is represented by a C compilation unit or C++ class.

3.5 Package

A package is a collection of components, usually represented by a directory.

3.6 Complex Expressions

MISRA defines the term “complex expression” as any expression that is not a constant expression, an L-value or the return value of a function.

3.7 Effectively Boolean Expression

MISRA defines an „effectively Boolean expression“ as the result of equality, logical or relational operators or a cast to a defined Boolean type.

4 Harman/Becker Rules

4.1 File Structure

HB01 English Language

1. Documentation and names shall be in US English.

This includes source comments, variable names, constants, file names and so on.

HB02 Headers and Includes (C, C++)

1. Each component shall define one public header file to expose the public or external declarations. All other files of the component (private headers and implementation files) shall go to a subdirectory named “private”.
2. Compilation units that include headers of other components shall use a pathname relative to the root directory of the project. Compilation units that include headers of their own component shall use a pathname relative to the component directory, except for generated headers which need to be included relative to the root directory of the project.
3. Components shall not include private headers of other components, except for the purpose of testing. Public headers should not include private headers of their own component.
4. Include paths shall contain only forward slashes (‘/’) as directory separator. Include paths shall not contain placeholders for the current (‘.’) or parent (‘..’) directory.
5. Each header shall be wrapped so that it will not be processed more than once.

These rules help to distinguish public interfaces and implementation details, make dependencies clearer and decrease the compile times for larger projects by reducing the number of search paths.

A compilation unit should not have multiple header files. However, essential global definitions (e.g. data types or constants) may be defined in extra headers.

When sources from several projects are merged, no `#include` directives have to be changed. Instead, the root directories of the subprojects have to be added to the include path list for the compilation itself.

For the sake of efficiency rather than information hiding, a compilation unit might need to include a private header of its own component. This makes internal definitions visible as an unwanted side-effect. In these cases, it is required to document the exact design rationale.

Compilation units that belong to test programs may need to include private headers of other components to gain access to internal interfaces.

An example directory (structure is also valid for C++ classes):

```
myproject/  
  mycomponent/  
    Alpha.h          /* "Alpha" is a public component */  
    private/  
      Alpha.c  
      Beta.h          /* the whole component "Beta" is private */  
      Beta.c  
      Defs.h          /* "Defs" contains definitions for Alpha & Beta.*/  
  mysubcomponent/  
    Gamma.h          /* "Gamma" is a public component */  
    private/  
      Gamma.c
```

in Alpha.h:

```
#ifndef MYPROJECT_MYCOMPONENT_ALPHA_H  
#define MYPROJECT_MYCOMPONENT_ALPHA_H  
... /* content here */  
#include "private/Defs.h" /* avoid private includes if possible */  
#endif
```

in Alpha.c:

```
#include "mycomponent/Alpha.h" /* no ".." in path */
```

HB03 Location of Declarations (C, C++)

1. Symbol reference with global linkage (such as global function prototypes or global variable references) should only be declared in header files.

Suppose the author of a component wants to modify some functions to provide an 'inline' definition for optimization reasons. This will fail unless the prototype declaration is obtained via the header file.

HB04 Classes per File (C++)

1. There should be only one class definition per compilation unit, with the exception of Helper Classes.

A relation between classes and files helps to locate class definitions, see also HB14.

Helper Classes may be defined together with a main class and may be defined locally (e.g. not in the header, as an inner class).

HB05 Class and Function Size (C, C++)

1. A function should not be larger than 60 lines of code with 80 characters.
2. A C++ class should not define more than 15 non-accessor methods.

"Lines of code" do not comprise comments or empty lines for structuring the code.

Smaller functions are much easier to understand, to test and to maintain since the complexity increases more than linearly with the amount of lines.

A class should be divided into multiple (possibly derived) classes if it has too many methods, since it is then likely that it has more than one objective.

If the given limit is exceeded, the rationale for this has to be documented.

HB06 Code Grouping (C, C++)

1. Contents of headers and sources shall be grouped as follows:

- a) preprocessor includes
- b) preprocessor defines (macros, constants)
- c) type definitions (enums, typedefs, structs, ...)
- d) constant and external variables
- e) function prototypes
- f) local variables (in implementation files)
- g) function definitions (in implementation files)
- h) class declarations (C++ only)
- i) friends (C++ only)
- j) public members (C++ only)
- k) protected members (C++ only)
- l) private members (C++ only)

2. Comments should separate these groups.

```
/*-----  
 *   MACROS  
 *----- */  
...
```

HB07 Instructions per Line (C, C++)

1. Each line shall contain at most one statement or variable declaration.

```
register(obj2); i++;    // BAD: easy to overlook the second statement
```

```
UInt8* buffer,buffer2; // BAD: buffer2 is not a pointer, it is a byte
```

HB08 Blanks and Indentation (C, C++)

1. Code lines shall be indented using blank characters. Indentation size should be 3 blanks.
2. There should be a single blank between binary and trinary operators, after commas and between keywords and parentheses.

3. There should be no blank between function names and parameter lists and between unary operators and operand.
4. `case` labels should not be indented, while `break` statements should.

Set up the editor to automatically replace tabs (ASCII 0x09) with blanks (ASCII 0x20).

Using source code formatting tools, these requirements can be met without great efforts.

```
while_(i_<_n)
{
    _c_+=_f(-i);
    _i++;
}
```

HB09 Braces (C, C++)

1. Curly braces “{” and “}” shall be placed in separate lines in the same column.
2. Braces shall be put after every branch and loop statement even if there is only one statement. Braces may be omitted after `case` labels.

These rules increase readability of the code. Putting also single statements into a block prevents problems with bad indentations and makes it easier to add further statements.

```
if (a == b)           // preferred formatting:
{
    doSomething();
}

if (a == b)
    doSomething();    // BAD: dangerous, use a block in any case

struct complex        // The formatting also applies to type definitions.
{
    float re;
    float im;
};

switch (state)
{
case A:               // An extra block is not necessary here
    dispatchToA();
    break;
default:
    break;
}
```

4.2 Code Documentation

HB10 Code Comments (C, C++)

1. Important assumptions that cannot be expressed as assertions shall be documented in the code.
2. Rationales for implementation choices that are not obvious shall be documented in the code.
3. Coding tricks that are not obvious shall be documented in the code.
4. Code passages that need rework shall be documented in the code starting with a `@todo` DoxyGen tag and one of the following keywords, where applicable:

`FIXME` – code is known to be defect or in need of polishing,

`WORKAROUND` – code needs rework as soon as some issue is resolved.

5. Commented out code shall contain a rationale why it is commented.

Important assumptions that need to be documented comprise predicates that are not computable or very difficult to code or too costly to execute as an assertion.

Implementation choices mainly concern the selection of internal data structures and algorithms.

The use of keywords in the comments facilitates the search for these code passages and clearly indicates the status of the concerned code passages.

The purpose of commented out code needs to be clear for the reader: Is it meant as an example (of good or bad use), is it a candidate for a code change in the future (possible design variant, possible functional augmentation, possible replacement of an existing work-around), or is it debug code that might be needed again in the future?

```
// assumption: isSorted(list)
index = binarySearch(list);

insertionSort(list); // fast for expected list size (<10 entries)

// check if n (>0) is a power of 2:
if ((n & (n - 1)) == 0)

UInt32 getStatus()
{
    ///<@todo implement proper error status
    return STATUS_OK;
}

// printf("*** x==%i", x); // debug code
```

HB11 File Header (C, C++)

1. Every source file shall provide a file header comment at the start of the file, using the template shown in the example section.

Only the second comment block defines the `@file` directive. Doxygen will ignore the first block - this information should already be part of the standard header that Doxygen will include in every generated documentation file anyway.

It is not recommended to include `perforce` tags in the header. This information is fully redundant and can be confusing (consider scenarios involving branching and integration).

Use the `@author` tag to indicate the primary author of the unit.

```
/* *****  
 * Project           Harman Car Multimedia System  
 * (c) copyright    2005  
 * Company          Harman/Becker Automotive Systems GmbH  
 *                  All rights reserved  
 * *****  
 **  
 * @file            CExampleWidget.hpp  
 * @ingroup          HMIComponent  
 * @author           Otto Mueller  
 * A useless, slow and ugly user interface element.  
 */
```

HB12 Interface Documentation (C, C++)

1. External and internal interface elements shall be documented consistently using Doxygen comments.
2. Doxygen comments should use the JavaDoc-style: Prefer the `/**` and `///` comments over the `/*!` and `//!` forms and the `@`-form of the documentation tags over the `\`-form. A short summary sentence should be written using the `JAVADOC_AUTOBRIEF` option. Comments should not use structural references. HTML tags shall be used for formatting.
3. The standard header for each page should be configured so that it includes the project name and copyright information (see also HB11).

Interface elements comprise defines, macros, typedefs, enums, structs, unions, classes, templates, variables, constants, member variables, functions, methods, constructors, destructors, operators, and namespaces.

The documentation should, for instance, describe the following:

- What is it supposed to do? What is the intention? What are side-effects?
- Are there any constraints?
- What instance or function is responsible for the deallocation of a pointer?

- What is the unit of a numeric data (e.g. length in inches or centimeters)?

Some interface elements have a declaration and a definition part. DoxyGen checks both locations, thus only one needs a comment, preferably in the header.

```
/**
 * Macro to compute the absolute value of the argument.
 * @param x an expression of numeric type; it is evaluated twice
 * and must not contain side-effects.
 * @return the absolute value of the argument.
 */
#define ABS(x) ((x)<0?-(x):(x))

/**
 * Test suite for X. ...
 * @param obj test object reference providing test environment.
 * @return true if the test succeeded, false otherwise.
 * @see testEverything()
 */
bool test(X &obj);

/**
 * Semaphore counter that controls access to this module.
 * Four concurrent accesses are allowed.
 */
Int32 gSema = 4;

/** @var Int32 gSema ... */    BAD: Structural references are problematic.

/**
 * Simple list class. ...
 * @param E the element type.
 */
template <class E> class List ...
```

HB13 Component Documentation (C, C++)

1. Components or packages should be documented in a separate text file `package.txt` with a single large DoxyGen block inside defining a group.
2. All files belonging to the component or package shall contain the proper tag: `@ingroup <component name>`.
3. The top level documentation shall give a project overview and include the project name, current date and version.

The component documentation may also document namespaces.

Some files (e.g. headers) may logically belong to several components; for that purpose, `@ingroup` accepts a list of names.

in package.txt:

```
/**
 * @defgroup mycomp MyComponent
 * Document your component here, give an overview and describe
 * points of interest, e.g. using the @see command.
 */
```

Put “@ingroup MyComponent” in all corresponding file headers of the files belonging to that component / package.

in project.txt:

```
/**
 * @mainpage
 *
 * Give an overview of your project here, provide name, current date
 * and version.
 */
```

4.3 Naming Conventions

HB14 File Names (C, C++)

1. C compilation unit names should start with a package shortcut to avoid conflicts with customer or supplier code.
2. C++ classes and their associated files should have the same name and spelling.

It is not recommended that the company or department name is used as prefix - use technical shortcuts instead, e.g. the abbreviation of the component.

The names of Helper Classes that are bundled with a major class may obviously differ from the file name of the major class.

HB15 File Suffixes (C, C++)

1. Files shall receive a suffix describing their contents:

.c	for C source files	.cpp	for C++ source files
.h	for C header files	.hpp	for C++ header files

It is much easier to handle a big amount of project files correctly within scripts and build specifications when the files are consistently tagged with standard suffixes.

Some projects mix C and C++ units; it is therefore highly recommended to use the .hpp suffix to facilitate the distinction between C++ and C headers.

MyModule.h	MyModule.c	
CMyModule.hpp	CMyModule.cpp	TMyTemplate.hpp

HB16 Naming (C, C++)

1. Names of variables, functions and types shall contain only letters and digits and shall be in mixed case with the first letter of each internal word capitalized. Names of variables shall start with a small letter. Names of functions shall start with a small letter and should contain a verb. Names of types shall start with a capital letter and should be nouns.
2. Names of namespaces shall contain only letters and digits.
3. Names of preprocessor defines (macros, constants) and general constants (enum values, global constant variables) shall only contain capital letters, digits and underscores.
4. Names of template parameters shall be treated like type names for type parameters, and like names of constants for non-type parameters.

Variables comprise local and global variables, parameters and member variables. Functions comprise global functions and methods. Types comprise structs, unions, classes, enums and typedefs.

Keep names simple and descriptive. Use whole words and avoid acronyms and abbreviations (unless these are widely used, such as URL or HTML). Abbreviations are admissible for names of namespaces or unit prefixes.

Names constructed by macros (using `##`-concatenation) sometimes cannot comply with these conventions; this is tolerable.

```
Int32 count;

bool isValid();

struct ExampleStruct exampleOne;

enum StandardColors
{
    BLACK = 0,
    WHITE = 0xffffffff
};

namespace NdvD ...

template <class Element, bool OPTION> ...

#define EV_KEY_PRESS    2

const UInt8 EV_KEY_PRESS = 2;

#define DISABLE_INTERRUPT() ...
```

HB17 Underscore Character (C, C++)

1. The underscore character ‘`_`’ shall neither be used at the beginning of a symbol nor in sequence with other underscores.
2. The underscore character shall be reserved for preprocessor constants and macros and to separate special name prefixes.

If used improperly, the underscore character in names may lead to conflicts with external libraries and compiler name mangling.

If the underscore character is used to separate prefixes, this should be kept consistent within the project: either use underscores for all name prefixes or do not use them for any name prefix. If nothing else has been defined, do not use underscores for prefixes.

```
MY_MACRO           // use underscores to separate words in macro names
mMemberVariable    // no underscores within other names
m_memberVariable    // it is admissible to separate special prefixes
m_member_variable   // BAD: do not use underscores inside names
```

HB18 Name Prefixes (C, C++)

1. Variables, types and namespaces shall receive a special prefix if they have one or several of the following properties:

<code>g</code>	global variable	<code>s</code>	static variable
<code>m</code>	member variable (C++)	<code>t</code>	custom defined type (typedef)
<code>C</code>	class (C++)	<code>I</code>	interface class (C++)
<code>T</code>	template (C++)	<code>N</code>	namespace (C++)
2. Prefixes for classes and namespaces may be followed by a company, project, component or package shortcut.
3. A capital letter or an underscore shall follow a prefix; any project shall define and follow one consistent convention (see HB17).

“Variables” also include parameters and member variables. It is no longer recommended to prefix variables with shortcuts of their types (the so-called Hungarian notation).

The special prefixes may violate rule HB16 (e.g. the `t` prefix is not a capital letter, the `N` prefix is not a small letter and the `T` prefix is not a noun), but the actual name must not.

```
extern UInt8 gFlag;           //or: g_flag
static UInt32 sDelay;         //or: s_delay
Int32 mMyMember;             //or: m_myMember
typedef struct AudioParam tAudioParam; //or: t_AudioParam
class CHBDesktopController;   //or: CHB_DesktopController
template<...> class TContainer; //or: T_Container
namespace Ndvd ...           //or: N_dvd
```

4.4 Preprocessor-related Rules

HB19 Includes (C, C++)

1. Headers shall be included with `#include <...>` when the associated sources will not be compiled in the project (e.g. for target OS).
2. Headers shall be included with `#include "..."` when the associated sources will or could be compiled in the project (e.g. internal libs).

`#include <...>` will locate files in the include path only while `#include "..."` will try to find files locally first, then use the include path.

```
#include <stdlib.h>
#include "MyComponent.h"
```

HB20 Conditional Compilation (C, C++)

1. `#if` should be used instead of `#ifdef`.
2. Conditional compilation should only be used for the distinction of host, target and operating systems and debug / release versions.
3. Types, functions or class members should not be declared dependent to compiler switches.

It is sometimes feasible to use a standard `if` statement instead of a preprocessor `#if`.

It is usually feasible to use an `#if` instead of `#ifdef`; `#if` avoids problems resulting from misspelled symbols.

Source code which uses conditional compilation is difficult to understand and to maintain.

Use source management system features to manage different variants of the same project. This requires a careful design that neatly separates commonalities and variations.

It can be extremely difficult to find errors that relate to differently compiled declarations. For instance, if a header defines a type dependent to a `NDEBUG` switch and is compiled differently within the same release, the addresses used in the calling and called functions do not correspond to each other and the v-tables may be sized differently.

```
#ifndef NDEBUG
    virtual void doSomething(void); // BAD
#endif

#ifdef HBTRACEMODE
    typedef int tMyType;           // BAD
#else
    typedef char tMyType;
#endif
```

HB21 Pragmas (C, C++)

1. Pragmas shall be wrapped with `#if` so that they will only be seen by the compiler for which they are intended.

See also MISRA rule M3.4.

HB22 Preprocessor Defines (C++)

1. Inline functions should be used instead of preprocessor-defined macros, constant variables instead of preprocessor-defined constants.

C++ inline functions and constant variables are type-safe and do not produce overhead compared with macros. See also MISRA rule M19.7 and Scott Meyers rule E1.

For int-like types with a limited domain, `enum` can also be used to define constants. Be aware of type conversion problems when using `enum` constants.

```
// BAD: macros and constants are not checked at all

#define DIST2(dx,dy)    ((dx)*(dx)+(dy)*(dy))

#define MAX_ENTRIES    ((2<<10)-1)

// functions and variables are structurally and type checked:

inline UInt32 dist2(Int32 dx, Int32 dy)
{
    return (UInt32)(dx * dx + dy * dy);
}

const Int32 maxEntries = (2 << 10) - 1;
```

HB23 Macro Definition (C, C++)

1. Macros shall be properly usable in all their intended contexts and with any intended arguments.

Macros that are used as statements should be wrapped in braces. There is a trick to wrap a macro in `do{ ... }while(0)` to make it behave like an ordinary statement concerning semicolon placements or dangling else branches. Use this at your own risk, as the constant expression involved can produce unwanted warnings.

Macros that are used as expressions shall be wrapped with parentheses to prevent issues with operator precedence.

Put every occurrence of a parameter in parentheses to prevent unpredictable operator precedence issues (see also MISRA rule M19.10). If a parameter occurs several times, document this explicitly to prevent insertion of arguments with side-effects.

```
#define SET_ONE_OR_ZERO(x) {(x) = (x) ? 1 : 0;}
/* properly in braces */

/** Compute absolute value. Argument will be evaluated twice. */
#define ABS(x) (x<0?x:-x)

// BAD: the documentation is okay, but without parentheses the final term
// of e.g. ABS(z+1) would expand to "-z + 1" which was not intended
```

4.5 Miscellaneous Rules

HB24 Intrinsic Types (C, C++, Embedded Systems)

1. Standard C/C++ intrinsic data types shall not be used directly. Instead, versions for integer types that state their exact or minimal bit-width explicitly in the type name shall be used. Types shall be defined for at least 8-bit, 16-bit and 32-bit integers for all target architectures needed.
2. Other intrinsic types shall only be used for interaction with external libraries, or for bit fields.
3. A boolean data type for C shall be defined.

This rule implements MISRA rule M6.3.

The H/B standard framework defines data types in the file “HBTypes.h”, other frameworks are also admissible (e.g. StarRec PAL) as long as they define the required minimum set of types. A valid alternative are the `stdint` types from the ISO 9899:1999 C standard.

It is always admissible to define further bit-sizes – e.g. 24, 48, 56, 64.

ANSI C dictates that bitfields have an `int` basetype. Since primarily the sign of the bitfield is important, it is recommended to simply declare “signed” or “unsigned”.

For most efficient loop counters and indices, it is admissible to define appropriate integer types with explicitly stated minimum bit sizes (such as `uint_fast16_t` in `stdint`).

H/B Standard Framework:

```
UInt8 *byteBuffer;

Int32 idArray[];

for (UInt16 i = 0; i < n; i++)

bool myFlag;

struct bit { unsigned b : 1; }
```

C'99 stdint.h:

```
uint8_t *byteBuffer;

int_least32_t idArray[];

for (uint_fast16_t i = 0; ...)
```

HB25 Null Pointers (C, C++)

1. For C++, 0 shall be used for invalid or uninitialized pointers.
2. For C, NULL shall be used for invalid or uninitialized pointers.

The definition of the C macro NULL can differ between compilers (e.g. 0, (void *)0, (__null) – the latter is not even equal to zero). C++ guarantees correct conversion of the integer 0 to an appropriate pointer type.

```
/* C: */                // C++:  
char *current = NULL;   char *mCurrent = 0;  
if (current != NULL) ... if (mCurrent != 0) ...
```

HB26 Const Classifier (C, C++)

1. The const classifier shall be used wherever feasible.

Possible locations are parameter types, function return types and C++ methods - ensuring that these parameters, return values or the this object cannot be modified in any way.

See also Scott Meyers, rule E21.

```
UInt8 const * const getView(const UInt8 &buffer) const;
```

HB27 Assertions (C, C++)

1. Important assumptions shall be made explicit using assertions.

Assertions check assumptions that the programmer requires to hold but does not want to or can not afford to check explicitly during run time of the production code. Assertions document those assumptions for both readers of the code and code analysis tools and also provide a fail-fast behavior during testing.

Use assertions to support the design by contract methodology by checking preconditions and postconditions of functions and methods and class invariants where this is feasible.

A proper error handling shall be preferred over the use of assertions; error handling can replace assertions but assertions shall never replace a proper error handling.

In the standard H/B framework for embedded systems, a macro ASSERT is defined in HBTrace.h and shall be used instead of the standard assert. Projects using other frameworks may provide alternative definitions or allow the standard assert macro.

Design-by-contract with assertions:

```
/**  
 * ...  
 * Neither the list nor the element may be null; the index must  
 * not exceed the list size. If the index matches the list size,  
 * the element is added at the end of the list. ...  
 */
```

```
void insertElement(tList *list, UInt32 index, tElement *element)
{
    ASSERT(list != 0);                // preconditions
    ASSERT(element != 0);
    const UInt32 oldSize = list->size;
    ASSERT(index <= oldSize);
    ...                               // implementation
    ASSERT(list->size == oldSize + 1); // postconditions
    ASSERT(list->get(index) == element);
}
```

HB28 Array Lengths (C, C++)

1. The `ARRAYLEN` macro should be used to determine the size of any statically allocated array.

The explicit use of the `ARRAYLEN` macro shows the intention of comparisons for array bound checking much better and it also helps tools such as Lint.

```
#define MAX_COUNT ...
Xyz* item[MAX_COUNT];

if (index < MAX_COUNT) ...
// BAD: is MAX_COUNT some threshold or the boundary of the array?

// better:
if (index < ARRAYLEN(item)) ...
```

The usual macro definition:

```
#define ARRAYLEN(a) (sizeof(a)/sizeof(a[0]))
```

4.6 C++ specific Rules

HB29 Orthodox Canonical Form (C++)

1. Each class shall define a default constructor, a copy constructor, an assignment operator and a destructor.
2. Destructors shall be `virtual` for all classes with virtual methods.
3. The assignment operator shall check equality of source and destination.

These definitions are necessary if an object shall be passed by value, it contains pointers that are reference counted, or the destructor deletes a data member. If some members make no sense for a particular class, they should be declared private with an empty body.

If default- and copy-constructors are not defined, code bloat in classes with virtual methods may happen. If copy constructor or assignment operators are not declared, the compiler creates them on demand. The default code copies the objects element-wise which might lead to unintended results with pointers (shallow vs. deep copy).

A class has virtual methods if it defines a virtual method or any of its base classes does. A non-virtual destructor in such a class can produce memory leaks if derived classes allocate additional memory and mistakenly only the base class destructor is called.

A typical error in implementations of the assignment operator is to omit a check whether source and destination are equal. In this case internal data of the own object may be freed after which this freed memory is copied, leading to erratic behavior.

See also Scott Meyers rules E11-E17.

```
CClassX& CClassX::operator=(const CClassX& theOther)
{
    if (this != &theOther)
    {
        ...
        // implement the copy operation
    }
    return *this;
}
```

HB30 Namespaces (C++)

1. C++ namespaces should be applied.
2. `using` declarations and directives shall not be used within the global namespace. `using` shall only be used after the last `#include`.

See also Scott Meyers rule E28.

Older compilers may not support namespaces. In that case, the naming convention for C units applies (see HB14).

Using `using` within the global namespace is very dangerous in combination with includes.

HB31 Reference Parameter (C++)

1. Parameters with non-intrinsic type shall be passed by reference whenever possible.

A reference always points to an existing object whereas simple pointers may point anywhere. If you know that a pointer may never be 0, declare a reference parameter. This makes the code more readable and makes some null pointer checks obsolete.

By passing an object (like `Test(CWString theString)`) the compiler will generate implicit calls to the copy-constructor and destructor, making this an expensive call.

See also Scott Meyers, rule E22.

```
void getPTYText(CWString &thePTYText);

bool CATunDesktopController::processEvent(const CEvent &theEvent)
```

HB32 Casts (C++)

1. C++-style casts should be used instead of C-style casts.
2. The rationale for any `reinterpret` or `const` casts shall be documented.
3. For Embedded Systems programming, the `dynamic_cast` shall not be used, since this would activate exception handling (see HB36).

See Scott Meyers, Rule EE2.

Frameworks may define alternative solutions to emulate dynamic casts. It is then recommended to use these solutions; fall back to the C-style cast otherwise.

```
const Int8 *buffer = cgFixedBuffer;
UInt8 *b = (UInt8*)buffer; // BAD: was removing the const intended?
UInt8 *b = static_cast<UInt8*>buffer; // leads to a compiler error
```

HB33 Inline Methods (C++)

1. Inline methods shall not be defined inside the class declaration.
2. Destructors shall not be defined inline.
3. Constructors should not be defined inline.

Some compilers generate memory overhead in special cases (e.g. templates). Inline destructors or constructors can lead to code bloat.

```
inline Int32 Cxyz::inlineMethod() // definition outside class declaration
{
    ...
}
```

HB34 Explicit Constructors (C++)

1. Constructors with exactly one parameter shall be `explicit`, with the exception of the copy constructor.

The implicit type conversions from converting constructors are error-prone.

```
CWString(UInt16 length); // BAD: implicit type conversions allowed
void test(const CWString& text);
CWString temp;
UInt16 tmp;
```

```
test(tmp);
```

/The above line contains a typo – it should have been `test(temp)`. Instead of issuing a warning, the compiler creates an anonymous string object, implicitly converting the `UInt16` to a `CWString`.

```
explicit CWString(UInt16 length); // prohibit implicit conversions
```

HB35 Inheritance (C++)

1. Protected inheritance shall not be used.
2. Private inheritance should not be used.
3. Multiple inheritance of non-Interface Classes should not be used.

Protected and private inheritance are techniques for code reuse and do not work too well for that purpose but are risky when combined with virtual methods.

The literature is full of problems of multiple inheritance especially the famous „diamond“ formations. Avoid multiple inheritance by changing the design: use delegation or derive from at most one implementation class and Interface Classes otherwise.

See also Scott Meyers, rules E42-43.

```
class CMyDerivedClass : public CSomeClass, public ISomeInterface
{
public:
    CMyDerivedClass();
    virtual void processEvent(CEvent* event);
    ...
}

CMyDerivedClass::CMyDerivedClass() : CSomeClass(), ISomeInterface()
{
}

CMyDerivedClass::processEvent(CEvent* event)
{
    CSomeClass::processEvent(event);
}
```

HB36 Code Bloat Prevention (C++, Embedded Systems)

1. Exceptions (i.e. `try`, `throw` and `catch`) shall not be used.
2. RTTI (Run Time Type Information) shall not be used.
3. The C++ library `iostreams` shall not be used.
4. The STL (Standard Template Library) shall not be used.

The overhead for exception handling or RTTI is too severe for embedded environments. Measurements showed a 50% size increase of binaries when exception handling is used.

The `iostream` library comes with very bulky localization support.

The STL activates RTTI, expects exception handling to be active and its iterators are not always properly inlined by compilers. The standard Harman/Becker framework contains optimized templates that are slimmer and faster than the corresponding STL versions.

5 MISRA Rules

MISRA rules apply to all H/B Embedded System projects. Where applicable, these rules also apply to C++.

An emerging number of vehicle manufacturers require us to implement software according to the “Guidelines for the Use of the C Language in Vehicle Based Software” published by MISRA. These Guidelines are often called MISRA C or just MISRA.

MISRA is the Motor Industries Software Reliability Association, located in the UK. A lot of the large and smaller vehicle manufacturers and some automotive suppliers are member of the MISRA consortium.

We apply these rules to C++ too because the intention of MISRA to provide a moderately safe way to deal with the risks of C also holds for C++.

5.1 MISRA Rule List

Some general exclusions are defined for H/B code, corresponding rationales are documented in small print. In some cases, conditions are defined for admissible situation- or project-specific **exclusions**. Rationales for these exclusions still need to be documented (see 2.4 and 2.5).

M1 Environment

1. (C only) All code shall conform to ISO 9899:1990 “Programming languages -- C”, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.

The following language features from the revised standard ISO 9899:1999 (“C9X”) are admissible, if all compilers relevant for the project understand them:

- a) The `int64` type may be defined using `long long int` or similar extensions.
- b) C++ one-line comments (“//”) may be used.

In the case of compatibility issues with C++-style comments, a script can replace these comments with C-style comments (“/* */”) in the project delivering the code.

2. No reliance shall be placed on undefined or unspecified behavior.
3. Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.
4. The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.
5. Floating-point implementations should comply with a defined floating-point standard.

M2 Language Extensions

1. Assembly language shall be encapsulated and isolated.
2. (C only) Source code shall only use `/* ... */` style comments.
3. The character sequence `/*` shall not be used within a comment.
4. Sections of code should not be "commented out".

M3 Documentation

1. All usage of implementation-defined behavior shall be documented.
2. The character set and the corresponding encoding shall be documented.
3. The implementation of integer division in the chosen compiler should be determined, documented and taken into account.
4. All uses of the `#pragma` directive shall be documented and explained.
5. The implementation-defined behavior and packing of bitfields shall be documented if being relied upon.
6. All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.

M4 Character Sets

1. Only those escape sequences that are defined in the ISO C standard shall be used.
2. Trigraphs shall not be used.
Also, C++ Digraphs shall not be used.

M5 Identifiers

1. Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
Generated code sometimes violates this rule; in that case, make sure that the compiler used does distinguish long identifiers and tailor your project-specific rules accordingly.
2. Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
3. A `typedef` name shall be a unique identifier.
4. A tag name shall be a unique identifier.
5. No object or function identifier with static storage duration should be reused.

6. No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.
7. No identifier name should be reused.

M6 Types

1. The plain `char` type shall be used only for storage and use of character values.
2. `signed` and `unsigned char` type shall be used only for the storage and use of numeric values.
3. `typedefs` that indicate size and signedness should be used in place of the basic types.
4. Bit fields shall only be defined to be of type `unsigned int` or `signed int`.
5. Bit fields of type `signed int` shall be at least 2 bits long.

M7 Constants

1. Octal constants (other than zero) and octal escape sequences shall not be used.

M8 Declarations and Definitions

1. Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
2. Whenever an object or function is declared or defined, its type shall be explicitly stated.
3. For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
4. If objects or functions are declared more than once their types shall be compatible.
5. There shall be no definitions of objects or functions in a header file.
6. Functions shall be declared at file scope.
7. Objects shall be defined at block scope if they are only accessed from within a single function.
8. An external object or function shall be declared in one and only one file.

9. An identifier with external linkage shall have exactly one external definition.
10. All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
11. The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
12. When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation.

M9 Initialisation

1. All automatic variables shall have been assigned a value before being used.
2. Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures.
3. In an enumerator list, the "=" construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised.

M10 Arithmetic Type Conversions

1. The value of an expression of integer type shall not be implicitly converted to a different underlying type if:
 - a) it is not a conversion to a wider integer type of the same signedness, or
 - b) the expression is complex, or
 - c) the expression is not constant and is a function argument, or
 - d) the expression is not constant and is a return expression.
2. The value of an expression of floating type shall not be implicitly converted to a different type if:
 - a) it is not a conversion to a wider floating type, or
 - b) the expression is complex, or
 - c) the expression is a function argument, or
 - d) the expression is a return expression.
3. The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.

4. The value of a complex expression of floating type may only be cast to a narrower floating type.
5. If the bitwise operators `~` and `<<` are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand.
6. A “U” suffix shall be applied to all constants of unsigned type.

M11 Pointer Type Conversions

1. Conversions shall not be performed between a pointer to a function and any type other than an integral type.
2. Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.
3. A cast should not be performed between a pointer and an integral type.
4. A cast should not be performed between a pointer to object type and a different pointer to object type.
Casts between class type pointers are admissible since there are no alignment issues.
5. A cast shall not be performed that removes any `const` or `volatile` qualification from the type addressed by a pointer.

M12 Expressions

1. Limited dependence should be placed on C's operator precedence rules in expressions.
2. The value of an expression shall be the same under any order of evaluation that the standard permits.
3. The `sizeof` operator shall not be used on expressions that contain side effects.
4. The right-hand operand of a logical `&&` or `||` operator shall not contain side effects.
5. The operands of a logical `&&` or `||` shall be primary-expressions.
6. The operands of logical operators (`&&`, `||` and `!`) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (`&&`, `||` and `!`).
7. Bitwise operators shall not be applied to operands whose underlying type is signed.

8. The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.
9. The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
10. The comma operator shall not be used.
11. Evaluation of constant unsigned integer expressions should not lead to wrap-around.
12. The underlying bit representations of floating-point values shall not be used.
13. The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

M13 Control Statement Expressions

1. Assignment operators shall not be used in expressions that yield a Boolean value.
2. Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
3. Floating-point expressions shall not be tested for equality or inequality.
4. The controlling expression of a `for` statement shall not contain any objects of floating type.
5. The three expressions of a `for` statement shall be concerned only with loop control.
6. Numeric variables being used within a `for` loop for iteration counting shall not be modified in the body of the loop.
7. Boolean operations whose results are invariant shall not be permitted.

M14 Control Flow

1. There shall be no unreachable code.
2. All non-null statements shall either:
 - a) have at least one side effect however executed, or
 - b) cause control flow to change.
3. Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.

4. The `goto` statement shall not be used.

The `goto` statement is desirable in rare occasions when leaving a nested loop from within an inner loop. The standard workaround is to insert an additional boolean variable that is checked in the outer loop. This can lead to a performance bottleneck if the compiler does not remove the additional variable and checks, and it is also harder to comprehend. A `goto` may only be used if the full workaround would be needed.

// A case for a desirable `goto`:

```
while (someCondition)
{
    while (otherCondition)
    {
        doSomething();
        if (furtherCondition)
        {
            /*lint -e{801}*/
            goto outOfLoops; // "goto" needed to avoid bottleneck
        }
        doSomethingElse();
    }
    doYetAnotherThing();
}
outOfLoops:
...
```

// The workaround idiom:

```
bool quickExit = false;
while (!quickExit && someCondition)
{
    while (!quickExit && otherCondition)
    {
        doSomething();
        quickExit = furtherCondition;
        if (!quickExit)
        {
            doSomethingElse();
        }
    }
    if (!quickExit)
    {
        doYetAnotherThing();
    }
}
```

5. The `continue` statement shall not be used.
6. For any iteration statement there shall be at most one `break` statement used for loop termination.
7. A function shall have a single point of exit at the end of the function.
8. The statement forming the body of a `switch`, `while`, `do ... while` or `for` statement shall be a compound statement.
9. An `if (expression)` construct shall be followed by a compound statement. The `else` keyword shall be followed by either a compound statement, or another `if` statement.
10. All `if ... else if` constructs shall be terminated with an `else` clause.

M15 Switch Statements

1. A `switch` label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.
2. **Excluded:** An unconditional `break` statement shall terminate every non-empty `switch` clause.

A `switch` clause may be non-terminated if this is documented. This „fall-through“ technique is allowed for code size optimization.

```
switch (x)
{
case ABC:
    statement;
    break;
case DEF:
case GHI:
    statement;
    //lint -fallthrough
default:
    statement;
    break;
}
```

3. The final clause of a `switch` statement shall be the `default` clause.

Some compilers complain about missing `default` statements while others complain about an unreachable `default` statement when the `switch` covers all values of an `enum` type. In that case, make the last case falling through to the default (see example below), or add an additional value “ILLEGAL” to the `enum` type.

If some cases need no special action, declare those with empty fallthroughs and a `break` and add a comment describing this situation.

```
switch (myState)
{
case START_STATE:
    ....
    break;
case IRRELEVANT_STATE:
case IRRELEVANT_STATE2:
    // no action needed
    break;
case FINAL_STATE:
    ....
    break;
default:
    FAIL("myState was in an illegal state");
    break;
}

switch (myState)
{
case START_STATE:
    ....
case FINAL_STATE:
default:
    ....
    break;
}
```

4. A `switch` expression shall not represent a value that is effectively Boolean.
5. Every `switch` statement shall have at least one `case` clause.

M16 Functions

1. Functions shall not be defined with variable numbers of arguments.
2. Functions shall not call themselves, either directly or indirectly.
3. Identifiers shall be given for all of the parameters in a function prototype declaration.
4. The identifiers used in the declaration and definition of a function shall be identical.
5. Functions with no parameters shall be declared with parameter type `void`.
6. The number of arguments passed to a function shall match the number of parameters.

7. A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.
8. All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
9. A function identifier shall only be used with either a preceding &, or with parenthesised parameter list, which may be empty.
10. If a function returns error information, then that error information shall be tested.

M17 Pointers and Arrays

1. Pointer arithmetic shall only be applied to pointers that address an array or array element.
2. Pointer subtraction shall only be applied to pointers that address elements of the same array.
3. >, >=, <, <= shall not be applied to pointer types except where they point to the same array.
4. Array indexing shall be the only allowed form of pointer arithmetic.
5. The declaration of objects should contain no more than 2 levels of pointer indirection.
6. The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

M18 Structures and Unions

1. All structure or union types shall be complete at the end of a translation unit.
2. An object shall not be assigned to an overlapping object.
3. An area of memory shall not be reused for unrelated purposes.
4. Unions shall not be used.

M19 Preprocessing Directives

1. `#include` statements in a file should only be preceded by other preprocessor directives or comments.
2. Non-standard characters should not occur in header file names in `#include` directives.

3. The `#include` directive shall be followed by either a `<filename>` or `"filename"` sequence.
4. C macros shall only expand to a braced initialiser, a constant, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct.
5. Macros shall not be `#define'd` or `#undef'd` within a block.
6. `#undef` shall not be used.
7. A function should be used in preference to a function-like macro.
8. A function-like macro shall not be invoked without all of its arguments.
9. Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
10. In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of `#` or `##`.
11. All macro identifiers in preprocessor directives shall be defined before use, except in `#ifdef` and `#ifndef` preprocessor directives and the `defined()` operator.
12. There shall be at most one occurrence of the `#` or `##` preprocessor operators in a single macro definition.
13. The `#` and `##` preprocessor operators should not be used.
14. The `defined` preprocessor operator shall only be used in one of the two standard forms.
15. Precautions shall be taken in order to prevent the contents of a header file being included twice.
16. Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.
17. All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related.

M20 Standard Libraries

1. Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.
2. The names of standard library macros, objects and functions shall not be reused.
3. The validity of values passed to library functions shall be checked.

4. **Excluded:** Dynamic heap memory allocation shall not be used.

Many of our major embedded system projects need dynamic memory allocation. This is admissible since memory capacity has grown significantly.

Since memory fragmentation is still an issue and there is no backing store if memory runs out, it is still recommended to avoid dynamic heap allocation where possible.

Implementation of a custom memory manager for known memory allocation schemes can also help to prevent fragmentation and reduce memory management overhead.

5. The error indicator `errno` shall not be used.

`errno` may be used in context where its behavior is welldefined and thread-safe. Projects relying on QNX qualify for this.

6. **Excluded:** The macro `offsetof`, in `<stddef.h>`, shall not be used.

The `offsetof` macro is a major asset for data marshalling (deserializing) and is therefore admissible. `offsetof` shall not be used in the context of bitfields.

7. The `setjmp` macro and the `longjmp` function shall not be used.

8. The signal handling facilities of `<signal.h>` shall not be used.

Signal handling routines may be used in contexts where their behavior is welldefined. Projects relying on QNX qualify for this.

9. The input/output library `<stdio.h>` shall not be used in production code.

Projects that have to rely upon functionality from `stdio` shall make sure that the functions used are thread-safe or are always called from one thread at a time only.

10. The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>` shall not be used.

11. The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` shall not be used.

These routines may be used in contexts where their behavior is welldefined. Projects relying on QNX qualify for this.

12. The time handling functions of library `<time.h>` shall not be used.

M21 Run-time Failures

1. Minimisation of run-time failures shall be ensured by the use of at least one of:

- a) static analysis tools/techniques;
- b) dynamic analysis tools/techniques;
- c) explicit coding of checks to handle run-time faults.

5.2 Compliance Table

The following table shows how the compliance to the rules are enforced. PC-Lint is able to check MISRA-C:2004 rules starting with version 8.00q. The checklist for code reviews contains the rules not covered by PC-Lint. Individual projects may use compiler switches or additional analysis tools to further increase the percentage of automatically enforced rules.

1.1	PC-Lint	7.1	PC-Lint	12.3	PC-Lint	15.2	Excluded	19.6	PC-Lint
1.2	Checklist	8.1	PC-Lint	12.4	PC-Lint	15.3	PC-Lint	19.7	Checklist
1.3	Checklist	8.2	PC-Lint	12.5	Checklist	15.4	PC-Lint	19.8	PC-Lint
1.4	PC-Lint	8.3	PC-Lint	12.6	Checklist	15.5	PC-Lint	19.9	PC-Lint
1.5	Checklist	8.4	PC-Lint	12.7	Checklist	16.1	PC-Lint	19.10	PC-Lint
2.1	PC-Lint	8.5	Checklist	12.8	PC-Lint	16.2	Checklist	19.11	PC-Lint
2.2	Checklist	8.6	PC-Lint	12.9	PC-Lint	16.3	PC-Lint	19.12	PC-Lint
2.3	PC-Lint	8.7	Checklist	12.10	PC-Lint	16.4	Checklist	19.13	PC-Lint
2.4	Checklist	8.8	PC-Lint	12.11	PC-Lint	16.5	PC-Lint	19.14	PC-Lint
3.1	Checklist	8.9	Checklist	12.12	Checklist	16.6	PC-Lint	19.15	Checklist
3.2	Checklist	8.10	PC-Lint	12.13	Checklist	16.7	PC-Lint	19.16	Checklist
3.3	Checklist	8.11	PC-Lint	13.1	PC-Lint	16.8	PC-Lint	19.17	Checklist
3.4	Checklist	8.12	Checklist	13.2	PC-Lint	16.9	Checklist	20.1	PC-Lint
3.5	Checklist	9.1	PC-Lint	13.3	PC-Lint	16.10	Checklist	20.2	Checklist
3.6	Checklist	9.2	PC-Lint	13.4	PC-Lint	17.1	Checklist	20.3	PC-Lint
4.1	PC-Lint	9.3	PC-Lint	13.5	Checklist	17.2	Checklist	20.4	Excluded
4.2	PC-Lint	10.1	PC-Lint	13.6	Checklist	17.3	PC-Lint	20.5	PC-Lint
5.1	PC-Lint	10.2	PC-Lint	13.7	PC-Lint	17.4	Checklist	20.6	Excluded
5.2	PC-Lint	10.3	Checklist	14.1	PC-Lint	17.5	Checklist	20.7	PC-Lint
5.3	PC-Lint	10.4	PC-Lint	14.2	PC-Lint	17.6	PC-Lint	20.8	PC-Lint
5.4	PC-Lint	10.5	PC-Lint	14.3	PC-Lint	18.1	PC-Lint	20.9	PC-Lint
5.5	PC-Lint	10.6	Checklist	14.4	PC-Lint	18.2	Checklist	20.10	PC-Lint
5.6	PC-Lint	11.1	Checklist	14.5	PC-Lint	18.3	Checklist	20.11	PC-Lint
5.7	PC-Lint	11.2	Checklist	14.6	Checklist	18.4	PC-Lint	20.12	PC-Lint
6.1	Checklist	11.3	PC-Lint	14.7	Checklist	19.1	PC-Lint	21.1	Checklist
6.2	Checklist	11.4	Checklist	14.8	PC-Lint	19.2	PC-Lint		
6.3	PC-Lint	11.5	Checklist	14.9	PC-Lint	19.3	PC-Lint		
6.4	PC-Lint	12.1	PC-Lint	14.10	PC-Lint	19.4	Checklist		
6.5	PC-Lint	12.2	PC-Lint	15.1	Checklist	19.5	PC-Lint		

6 Effective C++ Rules

Scott Meyers defines rules about the proper use of C++ in his books ‘Effective C++’ and ‘More effective C++’. These rules apply to each Harman/Becker C++ project. All rules are required unless marked as recommended.

PC-Lint can at least partially check for some of the rules of ECPP and MECPP (E1, E3-7, E9, E11-17, E19-20, E22-24, E26, E29-30, E35-38, E47, EE2, EE5-7, EE13, EE33).

E01-E04 Effective C++: Shifting From C to C++

1. Prefer const and inline to #define.
2. **Excluded for Embedded Systems (HB36.3):** Prefer iostream to stdio.h.
3. Prefer new and delete to malloc and free. (Recommended)
4. Prefer C++-style comments. (Recommended)

E05-E10 Effective C++: Memory Management

5. Use the same form in corresponding uses of new and delete.
6. Use delete on pointer members in destructors.
7. Be prepared for out-of-memory conditions.
8. Adhere to convention when writing operator new and operator delete.
9. Avoid hiding the “normal” form of new.
10. Write operator delete if you write operator new. (Recommended)

E11-E17 Effective C++: Constructors, Destructors and Assignment Operators

11. Declare a copy constructor and an assignment operator for classes with dynamically allocated memory.
12. Prefer initialization to assignment in constructors. (Recommended)
13. List members in an initialization list in the order in which they are declared.
14. Make destructors virtual in base classes. (Recommended)
15. Have operator return a reference to *this.
16. Assign to all data members in operator =. (Recommended)
17. Check for assignment to self in operator =.

E18-E28 Effective C++: Classes and Functions: Design and Declaration

18. Strive for class interfaces that are complete and minimal. (Recommended)
19. Differentiate among member functions, non-member functions, and friend functions. (Recommended)
20. Avoid data members in the public interface. (Recommended)

- 21. Use const whenever possible.
- 22. Prefer pass-by-reference to pass-by-value.
- 23. Do not try to return a reference when you must return an object.
- 24. Choose carefully between function overloading and parameter defaulting. (Recommended)
- 25. Avoid overloading on a pointer and a numerical type.
- 26. Guard against potential ambiguity.
- 27. Explicitly disallow use of implicitly generated member functions you do not want.
- 28. Partition the global namespace. (Recommended)

E29-E34 Effective C++: Classes and Functions: Implementation

- 29. Avoid returning “handles” to internal data.
- 30. Avoid member functions that return non-const pointers or references to members less accessible than themselves.
- 31. Never return a reference to a local object or to a dereferenced pointer initialized by new within the function.
- 32. Postpone variable definitions as long as possible. (Recommended)
- 33. Use inlining judiciously. (Recommended)
- 34. Minimize compilation dependencies between files. (Recommended)

E35-E44 Effective C++: Inheritance and Object-Oriented Design

- 35. Make sure public inheritance models “is-a”.
- 36. Differentiate between inheritance of interface and inheritance of implementation.
- 37. Never redefine an inherited nonvirtual function.
- 38. Never redefine an inherited default parameter value.
- 39. Avoid casts down the inheritance hierarchy. (Recommended)
- 40. Model “has-a” or “is-implemented-in-terms-of” through layering.
- 41. Differentiate between inheritance and templates.
- 42. Use private inheritance judiciously.
- 43. Use multiple inheritance judiciously.
- 44. Say what you mean; understand what you’re saying.

E45-E50 Effective C++: Miscellany

- 45. Know what functions C++ silently writes and calls.

46. Prefer compile-time and link-time errors to runtime errors. (Recommended)
47. Ensure that non-local static objects are initialized before they're used.
48. Pay attention to compiler warnings.
49. Familiarize yourself with the standard library.
50. Improve your understanding of C++.

EE01-EE05 More Effective C++: Basics

1. Distinguish between pointers and references.
2. Prefer C++-style casts. (Recommended)
3. Never treat arrays polymorphically.
4. Avoid gratuitous default constructors.
5. Be wary of user-defined conversion functions.

EE06-EE08 More Effective C++: Operators

6. Distinguish between prefix and postfix forms of increment and decrement operators.
7. Never overload `&&`, `||`, or `, .`
8. Understand the different meanings of `new` and `delete`.

EE09-EE15 More Effective C++: Exceptions

Excluded for Embedded Systems: Due to HB36.1, these rules are not applicable for Embedded Systems programming. They apply to other C++ programs, though.

9. Use destructors to prevent resource leaks.
10. Prevent resource leaks in constructors.
11. Prevent exceptions from leaving destructors.
12. Understand how throwing an exception differs from passing a parameter or calling a virtual function.
13. Catch exceptions by reference.
14. Use exception specifications judiciously.
15. Understand the costs of exception handling.

EE16-EE24 More Effective C++: Efficiency

16. Remember the 80-20 rule.
17. Consider using lazy evaluation. (Recommended)
18. Amortize the cost of expected computations. (Recommended)
19. Understand the origin of temporary objects. (Recommended)

- 20. Facilitate the return value optimization. (Recommended)
- 21. Overload to avoid implicit type conversions. (Recommended)
- 22. Consider using Op= instead of stand-alone op. (Recommended)
- 23. Consider alternative libraries. (Recommended)
- 24. Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI.

EE25-EE31 More Effective C++: Techniques

- 25. Virtualizing constructors and non-member functions. (Recommended)
- 26. Limiting the number of objects of a class. (Recommended)
- 27. Requiring or prohibiting heap-based objects. (Recommended)
- 28. Smart pointers. (Recommended)
- 29. Reference counting. (Recommended)
- 30. Proxy classes. (Recommended)
- 31. Making functions virtual with respect to more than one object. (Recommended)

EE32-EE35 More Effective C++: Miscellany

- 32. Program in the future tense. (Recommended)
- 33. Make non-leaf classes abstract. (Recommended)
- 34. Understand how to combine C++ and C in the same program.
- 35. Familiarize yourself with the language standard.

7 Changes

7.1 Change History

Rev.	Change	Date	Author
1.0	Created	30.06.1999	Gerald Harris
2.0	Upgraded for CPP	20.11.1999	Dirk Schönberg
2.1	Correction of Rules	24.11.1999	Dirk Schönberg
2.9	Corrected and expanded the rules, examples and explanations added, structure and layout changed.	25.04.2000	Dirk Schönberg
3.0	Corrected and expanded the rules, examples and explanations added	10.05.2000	Dirk Schönberg
4.0	Revision after Conference on Coding Styles.	12.02.2003	Arne Haeckel
4.1	Translation from German to English.	21.02.2003	Guido Köhler
4.2	Changes after Review of Gerald Harris & Thomas Hermann	07.03.2003	Arne Haeckel
4.3	Added MISRA, Scott Meyers rules. Updated Organization of Identifiers	07.03.2003	Arne Haeckel
4.4	Results of Review; NN01 removed	21.03.2003	Arne Haeckel
4.5	Stylistic changes, changed: AD02, AD03, VN02, NV01, new: B002, renamed: VN01 to P001	28.04.2003	Arne Haeckel
4.6	changed: NV01, new: NV03	05.05.2003	Arne Haeckel
5.0	New public release 5.0 (see section 7.3)	18.03.2004	Andreas Ludwig
5.1	New public release 5.1 (see section 7.2)	27.07.2005	Andreas Ludwig

7.2 Changes since Release 5.0

New publication style.

Clarified instructions for existing code (2.2.1): Code designed for broad reuse should always be updated to the current coding styles.

Wording adapted to match guideline for requirements (see 2.4).

Clarified instructions for tailoring (2.5): Not departments but product lines may tailor rules.

Definition of non-standard terminology added as chapter 1.

Updated examples to comply with Mocca V2.

HB02: Rules for include paths added. Allowances made for header inclusion for test programs and generated code.

HB04: Rule now properly refers to definitions, not declarations. In comments: Inner classes are admissible.

HB08: Recommended indentation of switch contents. Recommended indentation size 3 blanks.

HB09: Added: No braces required after case.

HB10: Now defines requirements for code documentation. Recommended use of todo tag.

HB12: Former contents of rule HB10 now merged with rule HB12 which had the same scope; there are no contradictions to the 5.0 rules.

HB13: Recommended component documentation filename is „package.txt“.

HB14: Recommendation of a package prefix C++ units removed, 14.1 only refers to C units.

HB15: Removed suffix requirements for preprocessed files.

HB16: Naming of template parameters.

HB20: Former contents of rule HB21 now merged with rule 20 which had a similar topic. Added recommended use of #if instead of #ifdef.

HB21: Now defines requirements for pragmas which has been a note before.

HB29: Removed duplicates with HB33. Changed rule for virtual destructors: only classes with virtual methods need those.

HB30: Using is prohibited for global namespace.

HB36: Incorporates former HB37, collects all rules to avoid code bloat.

HB37: Removed, incorporated into HB36.

MISRA: Incorporated new version of MISRA-C:2004 rules. The rules received a new numbering scheme and some rules have been rescinded. Some exclusions for H/B code and a compliance table are defined.

E02: Excluded for embedded systems projects.

7.3 Changes since Release 4.6

All rules from the previous release have been revamped.

Naming rules have been simplified.

Guidelines for code documentation using DoxyGen have been added.

A guideline for the handling of header files for the common build process has been added.

Many rules have improved descriptions and better examples.

The rules are now much less dependent on a concrete framework and can be tailored to accommodate special project needs.

Some rules dealing with several aspects of one topic have been merged into more concise rules (e.g. concerning naming conventions).

Duplicates of H/B rules implementing MISRA rules have been removed.

Some exceptions for the application of MISRA rules have been defined.

8 Bibliography

1. Programming in C++, Rules and Recommendations, Copyright © 1990-1992 by Ellementel Telecommunication Systems Laboratories, Box 1505, 125 25 Älvsjö, Sweden. www.doc.ic.ac.uk/lab/cplus/c++.rules
2. Effective C++. 50 Specific Ways to Improve your Programs and Designs. By Scott Meyers, Addison Wesley professional (also available in German)
3. More Effective C++. 35 New Ways To Improve Your Programs and Designs. By Scott Meyers, Addison Wesley professional (also available in German)
4. MISRA The Motor Industry Software Reliability Association, Guidelines For The Use Of The C Language In Vehicle Based Software. www.misra.org.uk
5. Advanced C++ Programming Styles and Idioms. By James O. Coplien. Addison-Wesley, ISBN 0201548550.
6. Thinking in C++. By Bruce Eckel. Prentice Hall.
www.mindview.net/Books/TICPP/ThinkingInCPP2e.html