JBLopen
Embedded Software Insight

MISRA-C 2025 Guideline Compliance Summary

wp0001 Version 1

April 30, 2026

Contents

1 | Introduction

This document contains JBLopen's MISRA-C 2025 Guideline Compliance Summary (GCS) including the compliance matrix and related deviations and remarks. The compliance matrix, along with a short explanation of its structure, can be found in the Compliance Matrix section.

Each deviation from a MISRA rule or directive is appropriately justified through a corresponding deviation record which can be found in the Deviations section. In line with the *MISRA Compliance:2020 Achieving compliance with MISRA Coding Guidelines* document, each deviation record contains the reason for the deviation, as well as a complete justification and description of the context where such deviation can be found.

In the absence of any deviation, additional remarks are sometimes provided to clarify how the compliance to a given rule or directive is implemented. These remarks can be found in the Notes section.

Please note that this document applies to all JBLopen's code unless stated otherwise. **All code covered by this document adheres to the C99 C Standard**. Some debugging or other non-critical optional modules may not be covered by the MISRA-C coding standard. Please refer to the relevant JBLopen's product User Manual or other project-specific documentation for more precisions on how the MISRA coding standard applies in each particular case.

2 | Compliance Matrix

JBLopen's MISRA-C 2025 compliance matrix can be found in Table 2. For a short explanation of the various elements contained in the matrix, please refer to Table 1.

The MISRA compliance matrix contains all directives and rules listed in MISRA-C 2025. Some rules, however, are grayed out which can mean one of three things:

- the rule is obsolete, that is, it used to appear in previous versions of the MISRA coding standard but was either removed, or merged with another rule, or simply renumbered;
- the rule only applies to a version of the C Standard posterior to C99 and, therefore, does not apply to JBLopen's code which adheres to C99;
- the rule does not apply because it refers to a language construct or standard library interface that is never used.

The reason why a particular rule is grayed out is indicated in the 'Note/Deviation' column.

Column	Description
Directive/Rule	Contains the rule or directive number, where 'D' stands for directive and 'R' for rule.
Category	Indicates whether the directive or rule is advisory (A), required (R) or mandatory (M).
Guideline	Contains a brief description of the rule or directive as it appears in the *MISRA C:2025 Guidelines for the use of the C language in critical systems* document.
Compliance	States the level of compliance for that rule or directive. There are three possible levels of compliance: 'Yes', 'Yes / Deviation' or 'No'. Whenever the rule or directive is not fully complied with, a link to a deviation record appears in the *Note/Deviation* column.
Note/Deviation	Contains a clickable link to a note (📝) or deviation record (⚠) for that particular rule or directive. Can also contain a short note. Notes and deviation records also have a clickable link (⮌) back to the corresponding line in the compliance matrix.

Table 1 – Elements and structure of the compliance matrix

Table 2 – MISRA Compliance Matrix

Directive / Rule	Category	Guideline	Compliance	Note / Deviation
D1.1	R	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood.	Yes	
D1.2	A	The use of language extensions should be minimized.	Yes	
D2.1	R	All source files shall compile without any compilation errors.	Yes	
D3.1	R	All code shall be traceable to documented requirements.	Yes	✎
D4.1	R	Run-time failures shall be minimized.	Yes	✎
D4.2	A	All usage of assembly language should be documented.	Yes	✎
D4.3	R	Assembly language shall be encapsulated and isolated.	Yes	✎
D4.4	A	Sections of code should not be ”commented out”.	Yes	
D4.5	A	Identifiers in the same name space with overlapping visibility should be typographically unambiguous.	Yes	
D4.6	A	*typedefs* that indicate size and signedness should be used in place of the basic numerical types.	Yes / Deviation	⚠
D4.7	R	If a function returns error information, then that error information shall be tested.	Yes	
D4.8	A	If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden.	Yes / Deviation	⚠
D4.9	A	A function should be used in preference to a *function-like* macro where they are interchangeable.	Yes	
D4.10	R	Precautions shall be taken in order to prevent the contents of a *header file* being included more than once.	Yes	✎
D4.11	R	The validity of values passed to library functions shall be checked.	Yes	
D4.12	R	Dynamic memory allocation shall not be used.	Yes	
D4.13	A	Functions which are designed to provide operations on a resource should be called in an appropriate sequence.	Yes	
D4.14	R	The validity of values received from external sources shall be checked.	Yes	
D4.15	R	Evaluation of floating-point expressions shall not lead to the undetected generation of infinities and NaNs.	Yes	
D5.1	R	There shall be no data races between threads.	—	C11 only
D5.2	R	There shall be no dynamic thread creation.	—	C11 only
D5.3	R	There shall be no deadlocks between threads.	—	C11 only
R1.1	A	The program shall contain no violations of the standard C syntax and *constraints*, and shall not exceed the implementation’s translation limits.	Yes	
R1.2	A	Language extensions should not be used.	Yes / Deviation	⚠
R1.3	R	There shall be no occurrence of undefined or critical unspecified behaviour.	Yes	
R1.4	R	Emergent language features shall not be used.	—	C11 only
R1.5	R	Obsolescent language features shall not be used.	Yes	

...continued from previous page

Directive / Rule	Category	Guideline	Compliance	Note / Deviation
R2.1	R	A project shall not contain *unreachable code*.	Yes / Deviation	⚠
R2.2	R	A project shall not contain *dead code*.	Yes	
R2.3	A	A project should not contain unused type declarations.	Yes	
R2.4	A	A project should not contain unused tag declarations.	Yes	
R2.5	A	A project should not contain unused macro definitions.	Yes	
R2.6	A	A project should not contain unused label declarations.	Yes	
R2.7	A	A function should not contain unused parameters.	Yes / Deviation	⚠
R2.8	A	A project should not contain unused object definitions.	Yes	
R3.1	R	The character sequences /* and // shall not be used within a comment.	Yes	
R3.2	R	Line-splicing shall not be used in // comments.	Yes	
R4.1	R	Octal and hexadecimal escape sequences shall be terminated.	Yes	
R4.2	A	Trigraphs should not be used.	Yes	
R5.1	R	*External identifiers* shall be distinct.	Yes	
R5.2	R	Identifiers declared in the same scope and name space shall be distinct.	Yes	
R5.3	R	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.	Yes	
R5.4	R	*Macro identifiers* shall be distinct.	Yes	
R5.5	R	Identifiers shall be distinct from macro names.	Yes	
R5.6	R	A *typedef* name shall be a unique identifier.	Yes	
R5.7	R	A tag name shall be a unique identifier.	Yes	
R5.8	R	Identifiers that define objects or functions with external linkage shall be unique.	Yes	
R5.9	A	Identifiers that define objects or functions with internal linkage should be unique.	Yes	
R5.10	R	A reserved identifier or reserved macro name shall not be declared.	Yes	
R6.1	R	Bit-fields shall only be declared with an appropriate type.	Yes	
R6.2	R	Single-bit named bit-fields shall not be of a signed type.	Yes	
R6.3	R	A bit-field shall not be declared as a member of a union.	Yes	
R7.1	R	Octal constants shall not be used.	Yes	
R7.2	R	A ”u” or ”U” suffix shall be applied to all integer constants that are represented in an unsigned type.	Yes	
R7.3	R	The lowercase character ”l” shall not be used in a literal suffix.	Yes	
R7.4	R	A string literal shall not be *assigned* to an object unless the objects type is ”pointer to *const*-qualified *char*.	Yes	
R7.5	M	The argument of an integer constant macro shall have an appropriate form.	Yes	

...continued from previous page

Directive / Rule	Category	Guideline	Compliance	Note / Deviation
R7.6	R	The small integer variants of the minimum-width integer constant macros shall not be used.	Yes	
R8.1	R	Types shall be explicitly specified.	Yes	
R8.2	R	Function types shall be in *prototype form* with named parameters.	Yes	
R8.3	R	All declarations of an object or function shall use the same names and type qualifiers.	Yes	
R8.4	R	A compatible declaration shall be visible when an object or function with external linkage is defined.	Yes	
R8.5	R	An external object or function shall be declared once in one and only one file.	Yes	
R8.6	R	An identifier with external linkage shall have exactly one external definition.	Yes	
R8.7	A	Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.	Yes	
R8.8	R	The *static* storage class specifier shall be used in all declarations of objects and functions that have internal linkage.	Yes	
R8.9	A	An object should be defined at block scope if its identifier only appears in a single function.	Yes	
R8.10	R	An *inline function* shall be declared with the static storage class.	Yes	
R8.11	A	When an array with external linkage is declared, its size should be explicitly specified.	Yes	
R8.12	R	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.	Yes	
R8.13	A	A pointer should point to a *const*-qualified type whenever possible.	Yes	
R8.14	R	The *restrict* type qualifier shall not be used.	Yes	
R8.15	R	All declarations of an object with an explicit *alignment specification* shall specify the same *alignment*.	—	C11 only
R8.16	A	The *alignment specification* of zero should not appear in an object declaration.	—	C11 only
R8.17	A	At most one explicit *alignment specifier* should appear in an object declaration.	—	C11 only
R8.18	R	There shall be no tentative definitions in a *header file*.	Yes	
R8.19	A	There should be no external declarations in a *source file*.	Yes	
R9.1	M	The value of an object with automatic storage duration shall not be read before it has been set.	Yes	
R9.2	R	The initializer for an aggregate or union shall be enclosed in braces.	Yes	
R9.3	R	Arrays shall not be partially initialized.	Yes	
R9.4	R	An element of an object shall not be initialized more than once.	Yes	
R9.5	R	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.	Yes	

...continued from previous page

Directive / Rule	Category	Guideline	Compliance	Note / Deviation
R9.6	R	An initializer using chained designators shall not contain initializers without designators.	Yes	
R9.7	M	Atomic objects shall be appropriately initialized before being accessed.	—	C11 only
R10.1	R	Operands shall not be of an inappropriate *essential type*.	Yes	
R10.2	R	Expressions of *essentially character* type shall not be used inappropriately in addition and subtraction operations.	Yes	
R10.3	R	The value of an expression shall not be assigned to an object with a narrower *essential type* or of a different *essential type category*.	Yes	
R10.4	R	Both operands of an operator in which the *usual arithmetic conversions* are performed shall have the same *essential type category*.	Yes	
R10.5	A	The value of an expression should not be cast to an inappropriate *essential type*.	Yes	
R10.6	R	The value of a *composite expression* shall not be assigned to an object with wider *essential type*.	Yes	
R10.7	R	If a *composite expression* is used as one operand of an operator in which the *usual arithmetic conversions* are performed then the other operand shall not have *wider essential type*.	Yes	
R10.8	R	The value of a *composite expression* shall not be cast to a *different essential type category* or a wider *essential type*.	Yes	
R11.1	R	Conversions shall not be performed between a pointer to a function and any other type.	Yes	
R11.2	R	Conversions shall not be performed between a pointer to an incomplete type and any other type.	Yes	
R11.3	R	A conversion shall not be performed between a pointer to object type and a pointer to a different object type.	Yes	
R11.4	A	A cast shall not be performed between a pointer to object and an integer type.	Yes	
R11.5	A	A conversion should not be performed from pointer to *void* into pointer to object.	No	⚠
R11.6	R	A cast shall not be performed between pointer to *void* and an arithmetic type.	Yes	
R11.7	R	A cast shall not be performed between pointer to object and a non-integer arithmetic type.	—	Merged with Rule 11.4
R11.8	R	A cast shall not remove any *const* or *volatile* or _*Atomic* qualification from the type pointed to by a pointer.	Yes / Deviation	⚠
R11.9	R	The macro `NULL` shall be the only permitted form of integer *null pointer constant*.	Yes	
R11.10	R	The _*Atomic* qualifier shall not be applied to the incomplete type *void*.	—	C11 only
R11.11	R	Pointers shall not be implicitly compared to `NULL`.	Yes	
R12.1	A	The precedence of operators within expressions should be made explicit.	Yes	

...continued from previous page

Directive / Rule	Category	Guideline	Compliance	Note / Deviation
R12.2	R	The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the *essential type* of the left hand operand.	Yes	
R12.3	A	The comma operator should not be used.	Yes	
R12.4	A	Evaluation of *constant expressions* should not lead to unsigned integer wrap-around.	Yes	
R12.5	M	The *sizeof* operator shall not have an operand which is a function parameter declared as ”array of type”.	Yes	
R12.6	R	Structure and union members of atomic objects shall not be directly accessed.	—	C11 only
R13.1	R	*Initializer lists* shall not contain *persistent side effects*.	Yes	
R13.2	R	The value of an expression and its *persistent side effects* shall be the same under all permitted evaluation orders and shall be independent from thread interleaving.	Yes	
R13.3	A	A *full expression* containing an increment (++) or decrement (–) operator should have no other potential *side effects* other than that caused by the increment or decrement operator.	Yes	
R13.4	A	The result of an assignment operator should not be *used*.	Yes	
R13.5	R	The right hand operand of a logical && or || operator shall not contain *persistent side effects*.	Yes	
R13.6	M	The operand of the *sizeof* operator shall not contain any expression which has potential *side effects*.	Yes	
R14.1	R	A *loop counter* shall not have *essentially floating* type.	Yes	
R14.2	R	A *for* loop shall be well-formed.	Yes	
R14.3	R	Controlling expressions shall not be invariant.	Yes	⚠
R14.4	R	The controlling expression of an *if* statement and the controlling expression of an *iteration-statement* shall have *essentially Boolean* type.	Yes	
R15.1	A	The *goto* statement should not be used.	Yes	
R15.2	R	The *goto* statement shall jump to a label declared later in the same function.	Yes	
R15.3	R	Any label referenced by a *goto* statement shall be declared in the same block, or in any block enclosing the *goto* statement.	Yes	
R15.4	A	There should be no more than one *break* or *goto* statement used to terminate an iteration statement.	No	⚠
R15.5	A	A function should have a single point of exit at the end.	No	⚠
R15.6	R	The body of an *iteration-statement* or a *selection-statement* shall be a *compound-statement*.	Yes	
R15.7	R	All *if...else if* constructs shall be terminated with an *else* statement.	Yes	
R16.1	R	All *switch* statements shall be well-formed.	Yes	
R16.2	R	A *switch label* shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement.	Yes	
R16.3	R	An unconditional *break* statement shall terminate every *switch-clause*.	Yes	

...continued from previous page

Directive / Rule	Category	Guideline	Compliance	Note / Deviation
R16.4	R	Every *switch* statement shall have a *default* label.	Yes	
R16.5	R	A *default* label shall appear as either the first or the last *switch label* of a *switch* statement.	Yes	
R16.6	R	Every *switch* statement shall have at least two *switch-clauses*.	Yes	
R16.7	R	A *switch-expression* shall not have *essentially Boolean type*.	Yes	
R17.1	R	The features of `<stdarg.h>` shall not be used.	Yes	
R17.2	R	Functions shall not call themselves, either directly or indirectly.	Yes	
R17.3	M	A function shall not be declared implicitly.	Yes	
R17.4	M	All exit paths from a function with non-*void* return type shall have an explicit *return* statement with an expression.	Yes	
R17.5	A	The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.	Yes	
R17.6	M	The declaration of an array parameter shall not contain the *static* keyword between the `[]`.	Yes	
R17.7	R	The value returned by a function having non-*void* return type shall be *used*.	Yes	
R17.8	A	A function parameter should not be modified.	Yes	
R17.9	M	A function declared with a *_Noreturn* function specifier shall .	—	C11 only
R17.10	R	A function declared with a *_Noreturn* function specifier shall have *void* return type.	—	C11 only
R17.11	A	A function that never returns should be declared with a *_Noreturn* function specifier.	—	C11 only
R17.12	A	A function identifier should only be used with either a preceding &, or with a parenthesized parameter list.	Yes	
R17.13	R	A *function type* shall not be type qualified.	Yes	
R18.1	R	A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.	Yes	
R18.2	R	Subtraction between pointers shall only be applied to pointers that address elements of the same array.	Yes	
R18.3	R	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object.	Yes	
R18.4	A	The +, −, += and −= operators should not be applied to an expression of pointer type.	No	⚠
R18.5	A	Declarations should contain no more than two levels of pointer nesting.	Yes	
R18.6	R	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.	Yes	
R18.7	R	Flexible array members shall not be declared.	Yes	
R18.8	R	Variable-length array types shall not be used.	Yes	
R18.9	R	An object with *temporary lifetime* shall not undergo array-to-pointer conversion.	Yes	

...continued from previous page

Directive / Rule	Category	Guideline	Compliance	Note / Deviation
R18.10	M	Pointers to variably-modified array types shall not be used.	Yes	
R19.1	M	An object shall not be assigned or copied to an overlapping object.	Yes	
R19.2	A	The *union* keyword should not be used.	Yes	
R19.3	R	A union member shall not be read unless it has been previously set.	Yes	
R20.1	A	*#include* directive should only be preceded by preprocessor directives or comments.	Yes / Deviation	⚠
R20.2	R	The `'`, `"` or `\` characters and the `/*` or `//` character sequences shall not occur in a *header file* name.	Yes	
R20.3	R	The *#include* directive shall be followed by either a `<filename>` or “`filename`” sequence.	Yes	
R20.4	R	A macro shall not be defined with the same name as a keyword.	Yes	
R20.5	A	*#undef* should not be used.	Yes	
R20.6	R	Tokens that look like a preprocessing directive shall not occur within a macro argument.	Yes	
R20.7	R	Expressions resulting from the expansion of macro parameters shall be appropriately delimited.	Yes	
R20.8	R	The controlling expression of a *#if* or *#elif* preprocessing directive shall evaluate to 0 or 1.	Yes	
R20.9	R	All identifiers used in the controlling expression of *#if* or *#elif* preprocessing directives shall be *#define*’d before evaluation.	Yes	
R20.10	A	The `#` and `##` preprocessor operators should not be used.	Yes	
R20.11	R	A macro parameter immediately following a `#` operator shall not immediately be followed by a `##` operator.	Yes	
R20.12	R	A macro parameter used as an operand to the `#` or `##` operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.	Yes	
R20.13	R	A line whose first token is `#` shall be a valid preprocessing directive.	Yes	
R20.14	R	All *#else*, *#elif* and *#endif* preprocessor directives shall reside in the same file as the *#if*, *#ifdef* or *#ifndef* directive to which they are related.	Yes	
R20.15	R	*#define* and *#undef* shall not be used on a reserved identifier or reserved macro name.	Yes	
R21.1	R	*#define* and *#undef* shall not be used on a reserved identifier or reserved macro name.	—	Renumbered as R20.15
R21.2	R	A reserved identifier or macro name shall not be declared.	—	Renumbered as R5.10
R21.3	R	The memory allocation and deallocation functions of `stdlib.h` shall not be used.	Yes	
R21.4	R	The standard *header file* `setjmp.h` shall not be used.	Yes	
R21.5	R	The standard *header file* `signal.h` shall not be used.	Yes	
R21.6	R	The Standard Library input/output functions shall not be used.	Yes	

...continued from previous page

Directive / Rule	Category	Guideline	Compliance	Note / Deviation
R21.7	R	The `atof`, `atoi`, `atol` and `atoll` functions of `stdlib.h` shall not be used.	Yes	
R21.8	R	The library functions *abort*, *exit* and *system* of `stdlib.h` shall not be used.	Yes	
R21.9	R	The library functions *bsearch* and *qsort* of `stdlib.h` shall not be used.	Yes	
R21.10	R	The standard library time and date functions shall not be used.	Yes	
R21.11	R	The standard *header file* `tgmath.h` shall not be used.	Yes	
R21.12	A	The exception handling features of `fenv.h` should not be used.	Yes	
R21.13	M	Any value passed to a function in `ctype.h` shall be representable as an *unsigned char* or be the value `EOF`.	Yes	
R21.14	R	The Standard Library function *memcmp* shall not be used to compare null terminated strings.	Yes	
R21.15	R	The pointer arguments to the Standard Library functions *memcpy*, *memmove* and *memcmp* shall be pointers to qualified or unqualified versions of compatible types.	Yes	
R21.16	R	The pointer arguments to the Standard Library function *memcmp* shall point to either a pointer type, an *essentially signed* type, an *essentially unsigned type*, an *essentially Boolean type* or an *essentially enum* type.	Yes	
R21.17	M	Use of the string handling functions from `string.h` shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters.	Yes	
R21.18	M	The `size_t` argument passed to any function in `string.h` shall have an appropriate value.	Yes	
R21.19	M	The pointers returned by the Standard Library functions *localeconv*, *getenv*, *setlocale* or, *strerror* shall only be used as if they have pointer to const-qualified type.	Yes	
R21.20	M	The pointer returned by the Standard Library functions *asctime*, *ctime*, *gmtime*, *localtime*, *localeconv*, *getenv*, *setlocale* or *strerror* shall not be used following a subsequent call to the same function.	—	N/A
R21.21	R	The Standard Library function *system* of `<stdlib.h>` shall not be used.	Yes	
R21.22	M	All operand arguments to any *type-generic macros* declared in `<tgmath.h>` shall have an appropriate *essential type*.	—	N/A
R21.23	R	All operand arguments to any multi-argument *type-generic macros* declared in `<tgmath.h>` shall have the same standard type.	—	N/A
R21.24	R	The random number generator functions of *stdlib.h* shall not be used.	Yes	
R21.25	R	All memory synchronization operations shall be executed in sequentially consistent order.	—	C11 only
R21.26	R	The Standard Library function *mtx_timedlock()* shall only be invoked on mutex objects of appropriate mutex type.	—	C11 only
R22.1	R	All resources obtained dynamically by means of Standard Library functions shall be explicitly released.	—	N/A

...continued from previous page

Directive / Rule	Category	Guideline	Compliance	Note / Deviation
R22.2	M	A block of memory shall only be freed if it was allocated by means of a Standard Library function.	—	N/A
R22.3	R	The same file shall not be open for read and write access at the same time on different streams.	—	N/A
R22.4	M	There shall be no attempt to write to a stream which has been opened as read-only.	—	N/A
R22.5	M	A pointer to a `FILE` object shall not be dereferenced.	—	N/A
R22.6	M	The value of a pointer to a `FILE` shall not be used after the associated stream has been closed.	—	N/A
R22.7	R	The macro `EOF` shall only be compared with the unmodified return value from any Standard Library function capable of returning `EOF`.	—	N/A
R22.8	R	The value of `errno` shall be set to zero prior to a call to an *errno-setting-function*.	—	N/A
R22.9	R	The value of `errno` shall be tested against zero after calling an *errno-setting-function*.	—	N/A
R22.10	R	The value of `errno` shall only be tested when the last function to be called was an *errno-setting-function*.	—	N/A
R22.11	R	A thread that was previously either joined or detached shall not be subsequently joined nor detached.	—	C11 only
R22.12	M	Thread objects, thread synchronization objects, and thread-specific storage pointers shall only be accessed by the appropriate Standard Library functions.	—	C11 only
R22.13	R	Thread objects, thread synchronization objects and thread-specific storage pointers shall have appropriate storage duration.	—	C11 only
R22.14	M	Thread synchronization objects shall be initialized before being accessed.	—	C11 only
R22.15	R	Thread synchronization objects and thread-specific storage pointers shall not be destroyed until after all threads accessing them have terminated.	—	C11 only
R22.16	R	All mutex objects locked by a thread shall be explicitly unlocked by the same thread.	—	C11 only
R22.17	R	No thread shall unlock a mutex or call *cnd_wait()* or *cnd_timedwait()* for a mutex it has not locked before.	—	C11 only
R22.18	R	Non-recursive mutexes shall not be recursively locked.	—	C11 only
R22.19	R	A condition variable shall be associated with at most one mutex object.	—	C11 only
R22.20	M	Thread-specific storage pointers shall be created before being accessed.	—	C11 only
R23.1	A	A generic selection should only be expanded from a macro.	—	C11 only
R23.2	R	A generic selection that is not expanded from a macro shall not contain potential *side effects* in the controlling expression.	—	C11 only
R23.3	A	A generic selection should contain at least one non-default association.	—	C11 only
R23.4	R	A generic association shall list an appropriate type.	—	C11 only
R23.5	A	A generic selection should not depend on implicit pointer type conversion.	—	C11 only

...continued from previous page

Directive / Rule	Category	Guideline	Compliance	Note / Deviation
R23.6	R	The controlling expression of a generic selection shall have an *essential type* that matches its standard type.	—	C11 only
R23.7	A	A generic selection that is expanded from a macro should evaluate its argument only once.	—	C11 only
R23.8	R	A default association shall appear as either the first or the last association of a generic selection.	—	C11 only

3 | Notes

Directive 3.1 Note

Guideline ⮌ All code shall be traceable to documented requirements.

Note The behavior of all public interfaces is documented in the corresponding header files and, in the case of JBLopen's products, reproduced in the accompanying API Reference Manual. Internal functions and algorithms are described in the relevant design documents produced as part of JBLopen's software development practices.

Directive 4.1 Note

Guideline ⮌ Run-time failures shall be minimized.

Note JBLopen's code contains extensive validation of function arguments and internal state consistency that can be enabled or disabled at compilation time. Those checks are designed to be usable in production for safety and mission critical projects and programmers are encouraged to keep them enabled at all time.

Directive 4.2 Note

Guideline ⮌ All usage of assembly language should be documented.

Note As a general rule, the use of assembly language is limited to platform porting code which is clearly documented and identified as such.

Directive 4.3 Note

Guideline ⮌ Assembly language shall be encapsulated and isolated.

Note In addition to being constrained to clearly idenified porting code, assembly language is either encapsulated in macros or functions.

Directive 4.10 Note

Guideline ⮌ Precautions shall be taken in order to prevent the contents of a header file being included more than once.

Note Header include guards are present in all header files.

4 | Deviations

Directive 4.6 Deviation

Guideline ⮌	*typedefs* that indicate size and signedness should be used in place of the basic numerical types.
Category	Advisory
Reason for deviation	Code quality (usability, portability)
Description	JBLopen's code makes extensive use of C99 fixed width data types from `stdint.h`. There are two general exceptions to this rule. The first exception is that most functions (both internal functions and public API) return a `int`-typed value to indicate an error condition. The second exception covers the case where a generic API function accepts an implementation-specific enumerated type (*enum*) as an input.

In the first case, it makes more sense to use the basic *int* type since returned codes are essentially enumerated types. In other words, their numeric value should not be interpreted, so their width should not be specified either. Returned codes are guaranteed to be equal to or greater than 0 and fit the current architecture `int` data type. This is easy to enforce since all possible return codes are compile-time constants. Given that the risks usually associated with variable-width types is mitigated in this case, a deviation from Directive 4.6 seems justified.

In the second case, the use of an enumerated type to internally represent implementation-specific constants eases readability and debugging. However, such implementation-specific type cannot be used in a generic interface function prototype. In that case, the generic function uses an `int`-typed parameter in lieu of the enumerated type. The enumerated type is given as an argument upon invoking the generic function, typically with a cast to `int` to prevent compilation warnings. This is illustrated in Listing 1.

```
/*******************
 * In bp_zynq_int.h
 *******************/
typedef enum bp_zynq_int {
    BP_ZYNQ_INT_TIMER = 27,
    BP_ZYNQ_INT_NFIQ = 28,
    BP_ZYNQ_INT_PRIV_TIMER = 29,
    BP_ZYNQ_INT_PRIV_WDOG = 30,
    BP_ZYNQ_INT_NIRQ = 29,
```

```
    ...
} bp_zynq_int_t;

/**************
 * In bp_int.h
 **************/
int bp_int_src_en(int id);

/***********
 * In app.c
 ***********/
rtn = bp_int_src_en((int)BP_ZYNQ_INT_TIMER);
if (rtn != RTNC_SUCCESS) {
    /* Error handling. */
}
```

Listing 1 – An int-typed parameter is used in lieu of an implementation-specific enumerated type.

Directive 4.8 Deviation

Guideline ⮌	If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden.
Category	Advisory
Reason for deviation	Code quality (Portability)
Description	Multiple implementations of a same interface are sometimes obtained by extending a generic object. For instance, multiple UART driver implementations can build upon the same generic UART structure. All implementations can access the generic object, but it is possible that some do not.

Rule 1.2 Deviation

Guideline ⮌	Language extensions should not be used.
Category	Advisory
Reason for deviation	Code quality (Portability)
Description	Usage of language extensions is limited to porting code where platform support requires language extensions for assembly and linking directives. Usage of extensions are selected in order to balance reliability, performance and maintainability of the code. Language extensions can also be used in integration code, such as kernel ports and manufacturer SDK integration, to maintain compatibility with those third party products.

Rule 2.1 Deviation

Guideline ⮌	A project shall not contain unreachable code.
Category	Advisory
Reason for deviation	Code quality (Reusability, Fault Tolerance)
Permits	R-2.1.A.1 / R-2.1.B.1
Description	Unreachable code is allowed as the result of defensive programming measures or as the result of particular build configurations.

The use of extensive argument and assertion checks cause some of those checks to become redundant in certain situations. Listing 2 shows an example of such a situation. In this scenario, the *default* switch case is unreachable but remains useful as a line of defense against a possible future programming error.

```
enum enum_type {
    ENUM_TYPE_A,
    ENUM_TYPE_B,
    ENUM_TYPE_C,
    ENUM_TYPE_INVALID
};

int afunc(enum_type_t e)
{
    if (e >= ENUM_TYPE_INVALID) {
        return (RTNC_FATAL);
    }

    switch(e) {
        case ENUM_TYPE_A:
            /* Handle type A. */
            break;
        case ENUM_TYPE_B:
            /* Handle type B. */
            break;
        case ENUM_TYPE_C:
            /* Handle type C.*/
            break;
        default:
            return (RTNC_FATAL); /* Unreachable. */
    }
    return (RTNC_SUCCESS);
}
```

Listing 2 – Unreachable code resulting from defensive programming measure

Rule 2.7 Deviation

Guideline ⮌ There should be no unused parameters in functions.

Category Advisory

Reason for deviation Code quality (Portability)

Description Unused parameters may arise in one specific implementation of a generic interface. In this case, unused parameters should be voided to indicate that they are intentionally ignored.

Listing 3 shows one example of unused parameters in a specific implementation of the generic `tsfs_media_drv_trim_t` interface. Since raw nand flash does not support trimming, the function does nothing so all its parameters are voided.

```
int tsfs_nand_media_adapter_trim(void *p_drv_data, uint64_t addr, uint64_t len)
{
    (void)p_drv_data;
    (void)addr;
    (void)len;

    return (RTNC_SUCCESS);
}
```

Listing 3 – Unsused parameters in an specific implementation of a generic interface

Rule 11.5 Deviation

Guideline ⮌ A conversion should not be performed from pointer to *void* into pointer to object.

Category Advisory

Reason for deviation Code quality (Portability)

Description Cast from *void pointers* are used in callback functions, ISRs and whenever implementation-specific data is provided through a higher level abstraction.

Rule 11.8 Deviation

Guideline ⮌	A cast shall not remove any *const* or *volatile* or *_Atomic* qualification from the type pointed to by a pointer.
Category	Required
Reason for deviation	Code quality (Maintainability)
Description	A function with a *const* pointer parameter returning a modifified (e.g. incremented) version of that pointer may need to cast out the *const* qualifier in order to preserve *const*-correctness (see rule 8.13).

Listing 4 is one example of such a function where removing the *const* qualifier from the returned value is necessary. One alternative, to preserve the *const*-correctness, would be to return a *const* pointer, but in that case the caller would be responsible for performing the cast, which is arguably worst than performing the cast only once inside the function. Notice how the cast is performed to avoid triggering a `cast-qual` warning.

```
char *tsfs_last_path_component_get(const char *p_path)
{
    const char *p_cur_char;

    p_cur_char = p_path;

    ...

    // Cast to uintptr_t required to avoid triggering -Wcast-qual
    return ((char *)(uintptr_t)p_cur_char);
}
```

Listing 4 – Removing the const qualifier is the best option in this case.

Rule 14.3 Deviation

Guideline ⮌	Controlling expressions shall not be invariant.
Category	Required
Reason for deviation	Code quality (Reliability)
Description	Argument checks and assertion checks are allowed to create an invariant *if* clause.

Rule 15.4 Deviation

Guideline ⮌	There should be no more than one *break* or *goto* statement used to terminate an iteration statement.
Category	Advisory
Reason for deviation	Code quality (Maintainability)
Description	In line with this rule, *goto* statements are never used to terminate an iteration statement. However, the use of multiple *break* statements is permitted if it enables a simpler and more intuitive code structure.

Rule 15.5 Deviation

Guideline ⮌	A function should have a single point of exit at the end.
Category	Advisory
Reason for deviation	Code quality (maintainability).
Description	To avoid obscuring program flow with long error return paths or numerous *goto* statements, functions are allowed to return immediately upon error.

Although the use of the *goto* statement is allowed for error handling (see Rule 15.4 Deviation), a systematic use of this method can quickly become confusing for the programmer and increase the chance of error. The early exit approach simplifies debugging since the cause of an error can be determined more readily. Most importantly, it prevents the cluttering of normal flow control with constant evaluation of error conditions. Outside of error handling, the of multiple return statements is discouraged.

Rule 18.4 Deviation

Guideline ⮌	The +, −, += and −= operators should not be applied to an expression of pointer type.
Category	Advisory
Reason for deviation	Code quality (Analysability)
Description	In some cases, pointer arithmetics is considered more natural than the array subscript equivalent. When this is the case, checks and assertions are added to make sure that the resulting pointer is within bounds.

Rule 20.1 Deviation

Guideline ⮌	*#include* directive should only be preceded by preprocessor directives or comments.
Category	Advisory
Reason for deviation	Code quality (Maintainability)
Description	In some cases, instrospection code used for testing purposes must have access to internal structures which are hidden from other translation units. In this case, the introspection code can be *#include*'d at the end of the translation unit (after all functions and variables declarations) therefore violating rule 20.1.

5 | Document Revision History

Version	Release Date	Description of Documentation Revision
1	2026-04-30	Initial release.

Table 3 – Document Revision History