Wind River VxWorks Coding Rules Guide

Document ID: VxWorksCodingRulesGuide

Author: Mati Sauks

Version: 1.2

Version Date: 25 March 2020

Status: Approved

Copyright Notice

Trademarks

Corporate Headquarters

Wind River
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.
Toll free (U.S.A.): +1-800-545-WIND
Telephone: +1 510 748 4100
Facsimile: +1 510 749 2010

For additional contact information, see the Wind River website:
http://www.windriver.com

For information on how to contact Customer Support, see:

http://www.windriver.com/support

Revision History			
Date	**Version**	**By**	**Description of Change**
Mar 03, 2018	0.01	Mati Sauks	Created from VxWorks Coding Standard 2014, also addressing JIRA defects VXWCS-1 to VXWCS-44
Jan 14, 2019	0.02	Mati Sauks	Update from review
May 05, 2019	0.03	Mati Sauks	Update from editorial review.
May 07, 2019	1.00	Mati Sauks	Update document to approved
Nov 19, 2019	1.1	Roger Bodén	Add secure coding rules chapter
March 25, 2020	1.2	Roger Bodén	Update secure coding rules chapter w.r.t. Annex K

Table of Contents

List of Tables

1 Introduction

This document defines the VxWorks coding rules whose purpose is to ensure that code written is safe and secure.

These rules will help to reduce coding errors , to avoid error introduction and to facilitate error detection.

1.1 Terminology

This document uses specific terms to indicate the necessity of the given set of rules. The words below have special meanings throughout the document:

- Shall or must – Indicates that the rule must always be followed, unless a deviation is requested and approved.
- Avoid or should – Indicates that the rule is a recommendation for good practice, but can be deviated without approval as needed.

1.2 Deviations

Deviations from rules that must be followed shall be permitted only if **all** the following conditions are met:

1. An architect or technical authority shall approve the deviation.
2. The justifications of the deviations are noted as code comments, preferably at the points of divergence.

1.3 Legacy Code

Some of the code in VxWorks was written before this standard existed, and thus may not comply with some of the rules in this document. The code for a defect fix should follow the style of the original code. All newly written code shall follow the rules in this document.

1.4 Backdoors

A backdoor is an undocumented (secret) mechanism to access a computer system by bypassing normal authentication and/or encryption procedures. VxWorks shall not contain any backdoors.

1.5 Applicable Documents

The table below lists all documents referenced within this document.

Ref.	Title
Ref.1	C++ Core Guidelines – CppCoreGuidelines.md by Bjarne Stroustup and Herb Sutter Version January 3, 2019 and later

Table 1-1 Applicable Documents

2 Acronyms

ANSI	American National Standards Institute
BOOL	Boolean
DMA	Direct Memory Access
EOS	End of string
IEC	International Engineering Consortium
ISO	International Standards Organization
ISR	Interrupt Service Routine
NULL	Zero value
NOOP	No Operation
OS	Operating System
SMP	Symetrical Multi Processing
SQA	Software Quality Assurance
U,UL,ULL	Unsigned, Unsigned Long, Unsigned Long Long

3 C Coding Rules

The following conventions define rules to follow when writing C code.

3.1 Coverity Defect Suppression using Annotation

Coverity is used to detect common errors in C code.

Justification for suppressing a Coverity defect must be provided in the source code (via comments) immediately before the Coverity specific annotation.

Examples:

1) Suppressing a Coverity defect as "false positive"

An annotation marking a Coverity finding as “false positive” must only be used if Coverity is actually wrong. Additional commentary is need to explain why the Coverity error is a false positive.

For example, if a “forward_null” error is detected in some source code line, yet it’s impossible for a NULL pointer to be dereferenced in that source code line, then a Coverity[forward_null : FALSE]” annotation should be used to mark the Coverity finding as a “false positive”. The following specific example from pkgs/os/utils/unix/regex/regcomp.c provides a good description of why this particular NULL pointer dereference is a “false positive”:

```
while (cp < g->must + g->mlen)
    {
    /*
     * Coverity worries that 'scan' (set to 'start'
     * just above) may be NULL here. However, this code
     * is only executed if g->mlen is nonzero, and
     * g->mlen only becomes nonzero if newlen has become
     * (at some point) greater than zero, and start has been
     * initialized from newstart in turn as a pointer offset
     * from the original value of scan in the 'strip'.
     */

    /* coverity[forward_null : FALSE] */
    /* coverity[dereference : FALSE] */

    while (OP(s = *scan++) != OCHAR)
        {
        continue;
        }
#ifdef __RTP__
    clen = wcrtomb (cp, (wchar_t) OPND (s), &mbs);
    assert (clen != (size_t)-1);
#else
    clen=1;
```

```
#endif
    cp += clen;
    }
```

2) Suppressing a Coverity defect as "intentional"

An annotation marking a Coverity finding as “intentional” should be used when Coverity is actually correct, but it was concluded that the code is behaving as intended. For example, a NULL pointer dereference error detected by Coverity could be deemed “intentional” in cases where performance would suffer significantly if the pointer was checked for NULL. In other words, the lack of checking for a NULL pointer was “intentional” to ensure good performance. For example, in the following code:

```
IP_STATIC void ipcom_vxworks_rtp_del_hook
    (
    const RTP_ID rtpId, /* The ID of the RTP */
    const int exitCode  /* The exit code to use */
    )
    {
    WIND_TCB *tcb = taskTcb (taskIdSelf ());
    ip_assert (tcb != IP_NULL);

    /*
     * Coverity raises an issue about the NULL pointer dereference of 'tcb'
     * which strictly speaking is correct. But taskTcb(taskIdSelf) essentially
     * returns the value of 'taskIdCurrent' and it's pretty much impossible
     * for this value to be NULL. Thus the reported issue is marked as
     * intentional.
     */
    /* coverity[forward_null] */

    ipcom_vxworks_task_del_hook (tcb);
    }
```

The reference to the ‘tcb’ automatic variable would be flagged as a NULL pointer dereference. That would be a correct finding since taskTcb() returns a pointer value, and there is no code in the ipcom_vxworks_rtp_del_hook() function that ensures a NULL value is never dereferenced (note the ip_assert() is typically a NOOP in standard builds). However Coverity doesn’t have a systems/OS view of VxWorks, and thus cannot “understand” that taskIdSelf() simply returns the value of ‘taskIdCurrent’ variable (which is a slot in the vxKernelVars[] array for SMP), and taskTcb() simply returns the input parameter. So it’s pretty much impossible for ‘tcb’ to be a NULL pointer, and thus this finding would be categorized as “intentional”.

3.2 Certified Code

The code may contain requirements comments of the form `/* req: <function name>_N_N */` right justified. These should not be touched unless performing certification activities. Leave these comments on the same line, even if the code in the line is removed.

```
int dllCount
    (
    DL_LIST * pList      /* pointer to list descriptor */
    )
    {
    DL_NODE * pNode =  NULL;
    int count = 0;                                          /* req: VX7-8257 */

    pNode = DLL_FIRST (pList);
    while (pNode != NULL)                                   /* req: VX7-8258 */
        {
        count++;
        pNode = DLL_NEXT (pNode);
        }

    return (count);
    }
```

3.3 Standard Library Functions

The following standard library functions shall not be used:

- `setjmp ()`
- `longjmp ()`

3.4 Pointer declarations

Pointer declarations should avoid having more than two levels of pointer nesting. ie.

```
FOO *** ptr;
FOO **** ptr;
```

3.5 Numeric Constants

Use #define to define meaningful names for constants. Numeric constants should not be used in code or declarations (except for obvious uses of small constants like 0 and 1).

Integer constants can be signed or unsigned. Integer constants that are intended to be used in unsigned expressions should be suffixed with U, UL, ULL to produce an unsigned int, unsigned long, or unsigned long long value respectively.

Standard numeric constants should also be used. These can be found for example in ‘limits.h’ and the compiler’s ‘limits.h’.

Do not use octal constants.

3.6 Use Defined Names

Use the names defined by including vxWorks.h wherever possible. In particular, note the following definitions:

- Use TRUE and FALSE for boolean assignment.
- Use EOS for end-of-string tests.
- Use NULL for zero pointer tests.

3.7 Boolean Tests

Do not test non-Booleans as you test a Boolean. This applies to all conditional tests e.g. while, for, if. The tested condition must be a Boolean expression.

For example, where x is an integer:

CORRECT:
```
if (x == 0)
```

INCORRECT:
```
if (! x)
```

Similarly, do not test Booleans as non-Booleans. For example, where libInstalled is declared as BOOL:

CORRECT:
```
if (libInstalled)
```

INCORRECT:
```
if (libInstalled == TRUE)
```

3.8 Passing and Returning Structures

Always pass and return pointers to structures. Structures must never be passed directly.

3.9 Return Status Values

Functions that return status values should return either OK or ERROR (defined by including vxWorks.h). The specific type of error is identified by setting errno. Functions that do not return any values should return void.

3.10 Unused Return Values

The return code of any function (or function pointer) that returns a value should be checked.

However there might be times that it is not practical to test the return code. In cases where the function return value is not checked, comments should be added indicating why the return value is not being checked, and the function call cast to (void). This will exactly describe the developer's intent and make this code easier to maintain.

NOTE: If the code is to be certified, a return value that signifies an error condition must be checked. For cases where the return value does not signify an error condition and it's decided not to check the return value, comments must be added indicating why the return value is not being checked, and the function call must be cast to (void).

In the following examples, funcA returns an int.

CORRECT:

```
/* the return value of funcA () is not needed or used */

(void) funcA ();
return;
```

INCORRECT:

```
funcA ();
return;
```

3.11 Bracing

All single statement conditionals shall be fully braced

CORRECT:

```
if (foo) statement;
if (foo)
    {
    statement;
    }
while (foo) statement;
while (foo)
    {
    statement;
    }
```

INCORRECT:

```
if (foo)
    statement;
while (foo)
    statement;
```

3.12 Reliance on Precedence of Operations

Precedence of operations should not be relied upon; be explicit through the use of parentheses.

CORRECT:

```
(a << 4) | ((b * 5) / 8)
```

INCORRECT:

```
a << 4 | b * 5 / 8
```

3.13 Signed Unsigned Comparisons

Mixing of signed and unsigned variables and constants can lead to unexpected consequences, especially when comparisons are being made. Always select the right type. A signed type indicates that the variable is expected to be negative, otherwise **always** select an unsigned type.

Similarly, numeric constants are by default signed, and require an explicit suffix of U, UL or ULL in order to be unsigned.

CORRECT:

```
uint32_t foo = 2U;
uint32_t bar = 3U;
if (foo < bar)
```

```
int32_t foo = -1;
int32_t bar = 0;
if (foo > bar)
```

INCORRECT:

```
int32_t foo = -2;
uint32_t bar = 2;
if (foo < bar)
```

In this case foo is sign extended then promoted to a uint32_t thus becoming a very large number.

NOTE: a cast of foo to uint32_t does not help. The only correct solution is to verify that foo is not negative before performing the if statement, or part of the if statement.

CORRECT:

```
int32_t foo = -2;
uint32_t bar = 2U;
if (foo < 0)
    {
    …
    }
else if (foo < bar))
    {
    …
    }
```

CORRECT:

```
uint16_t foo = 3U;
uint16_t bar = 2U;
if (foo == bar)
```

INCORRECT:

```
uint16_t foo = USHRT_MAX;
int16_t bar = (int16_t)foo;
```

```
if (foo == bar)
```

In this case foo is not equal bar!!

This is because before the comparison, foo and bar are first promoted to int, during which bar is sign extended.

Please Google: C integer type promotion rules.

or review the ISO/IEC 9899:2011 standards chapter 6.3.1 Arithmetic operands

http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

3.14 Switch/Case

Many times it is appropriate to fall through to the next case in a switch/case construct. If a fall through is appropriate, please document that fact with a comment describing it. This will exactly describe the developer's intent and make this code easier to maintain. The default label requires a statement. Cases without any statements can fall through without comment.

CORRECT:

```
switch (x)
    {
    case 1:
    case 2:
    case 3:
        some statement;

    /* this is an intentional fall through to the next case */
    case 4:
        another statement;
        break;
    ....
    ....
    default:
        break;
}
```

INCORRECT:

```
switch (x)
     {
     case 1:
     case 2:
     case 3:
         some statement;
     case 4:
         another statement;
         break;
    ....
    ....
    default:
        break;
```

```
    }
```

Switch statements must contain a default case as the last case.

CORRECT:

```
switch (initialState)
    {
    …
    case SEM_FULL:
        pSemaphore->semOwner = (WIND_TCB *) SEMB_FULL_STATE;
        break;
    default:
        errno = S_semLib_INVALID_STATE;
        return (ERROR);
}
```

INCORRECT:

```
switch (initialState)
    {
    …
    case SEM_FULL:
        pSemaphore->semOwner = (WIND_TCB *) SEMB_FULL_STATE;
        break;
    }
```

3.15 FUNCPTR Typedefs

To improve code quality and portability, the use of fully typed pointers to functions is now required whenever possible. The use of the FUNCPTR, VOIDFUNCPTR, DBLFUNCPTR, and FLTFUNCPTR type definitions is discouraged. When these types are used, the compiler is not able to perform any type checking of the parameters that are passed through the function pointer. Additionally, some types (such as long long) are not passed correctly from caller to callee if the call occurs through a pointer to a function that is declared using these types.

Pointers to functions should be declared when they are used, using standard C declaration syntax. For example, to declare a global function pointer variable that returns STATUS and accepts a VIRT_ADDR and a PHYS_ADDR parameter, the following syntax is used:

```
STATUS (*_func_foo)
    (
    VIRT_ADDR param1, /* 1st param for _func_foo */
    PHYS_ADDR param2  /* 2nd param for _func_foo */
    );
```

The individual parameters are each listed with a type name and a named parameter. The individual parameters are named so that the meaning of each parameter is clear to the reader:

```
STATUS (*_func_faststrcpy)
    (
    char * pDst, /* destination buffer */
    char * pSrc  /* source buffer */
    );
```

If a pointer to a function is declared within a structure or union, standard C declaration syntax is used within the declaration. Again, the individual parameters to the function are listed, each on their own line:

```
typedef struct fooStruct
    {
    STATUS (*fooFunc)
        (
         VIRT_ADDR startAdr, /* starting address */
         VIRT_ADDR endAdr,     /* ending address */
         uint32_t  options     /* option bits */
        );
    } FOO_STRUCT;
```

If a pointer to a function is needed for a function that accepts a variable argument list, then the variadic form of the function declaration is used:

```
typedef struct barStruct
    {
    STATUS (*printFunc)
        (
        char formatStr, /* formatting instructions */
        ...             /* variadic */
        );
    } BAR_STRUCT;
```

NOTE: At least one typed parameter must be provided prior to the variadic portion of the argument list. This is an ISO C requirement.

In rare circumstances neither the fully typed nor variadic forms of the function declaration can be used. In these situations, the use of one of the FUNCPTR typedefs is allowed.

In most circumstances, it is not necessary to declare an explicit typedef for a pointer to a function, because most function pointer typedefs are used only as part of a structure or union declaration within the same header file that defines the typedef. However, if a type definition for a particular pointer to a function is needed in more than one source file, a typedef can be created to improve readability.

3.16 Remove Compiler Warnings and Errors

The code must compile without errors and warnings (when the default warnings level is in effect).

3.17 Avoid Use of Casts

Each and every cast represents a potential error. It is normal to functionly use a cast to fix warnings reported by the compiler. However, each instance must be examined carefully to ensure that a cast is the appropriate action. Many times the warning indicates an actual error in argument passing that needs to be corrected. Use of a cast overrides the compiler's ability to detect an actual error in data usage that may prove to be significant.

If a cast is used, make sure it is between appropriate types. For example ensure that the cast can never result in unintended information loss (such as casting a long to an int).

3.18 Type Casting

All type casting shall be explicit when converting types. Constants assigned to unsigned variables shall use the suffix U, UL or ULL.

CORRECT:

```
uint8_t  uint8Value = 0U;
uint32_t uint32Value = 3UL;

uint32Value = (uint32_t) uint8Value;

int8_t  int8Value = 0;
int32_t int32Value = 3;

int32Value = (int32_t) int8Value;
```

Type promotion may cause unexpected results when casting between different types and sizes, and must be mitigated in the code if needed

```
int8_t  uint8Value = -3;
uint32_t uint32Value = 3UL;

uint32Value = (uint32_t) int8Value;
```

In this case the int8Value is sign extended before the cast and assignment to uint32Value. uint32Value thus being 0xFFFFFFFD. For example if only the 8 bit quantity is wanted one should use:

```
uint32Value = ((uint32_t) int8Value) & 0xFFUL;
```

3.19 Shift Operations

The right hand operand of a shift must have a value from 0 to 1 less than the width of the type. Do not apply right shift (>>) or left shift (<<) operators to signed operands (the results may be

compiler dependent with respect to the sign bit). Do not use shifts to perform division or multiplication, compilers will translate multiplies and divides to shifts if possible.

CORRECT:
```
a = b * 4;
```

INCORRECT:
```
a = b << 2;
```

CORRECT:
```
uint32_t val;
uint32_t shift;
if (shift < (uint32_t) (sizeof (val) * CHAR_BIT))
    {
    val = val << shift;
    }
```

INCORRECT:
```
uint32_t val;
uint32_t shift;
val = val << shift;
```

3.20 Side Effects

Function parameters shall not have any side effects. The order of parameter evaluation is compiler dependent.

CORRECT:
```
functionA (x * i, y / i, i);
i++;
```

INCORRECT:
```
functionA (x * i, y / i, i++);
```

Do not use side effects in logical expressions:

INCORRECT:
```
if (x && y++)
```

3.21 Logical Expressions

Do not use assignments in logical expressions.

CORRECT:
```
foo = functionA ();
```

```
if (foo == 5) …

if (functionA () == 5) …
```

INCORRECT:

```
if ((foo = functionA ()) == 5) …
```

Comparison in a logical expression shall result in a Boolean comparison

CORRECT:

```
if (ptr == NULL) …
```

INCORRECT:

```
if (ptr) …
```

3.22 Preprocessor Statements

Preprocessor #if and #elif statements shall both have Boolean expressions evaluating to 0 or 1

CORRECT:

```
#if (foo > 0)
```

INCORRECT:

```
#if (foo)
```

3.23 For Loops

"for" loops should use counters specific for the "for" loop. Variables should have a single well-defined purpose.

CORRECT:

```
for (ix = 0; ix < 10; ++ix)
    {
    /* code for loop #1 omitted */
    }
…
for (ix = 5; ix < 25; ++ix)
    {
    /* code for loop #2 omitted */
    }
```

INCORRECT:
```
result = a * SCALE_FACTOR;

for (a = 0; a < 10; ++a)
    {
    /* code omitted */
    }
```
In the above a is used as the loop counter and for calculations outside the for loop.

The loop counter inside the “for” loop should not be modified.

CORRECT:
```
for (ix = 0; ix < stackPages; ++ix)
    {
    MMU_DISABLE_USR_ACCESS ( PAGE_TABLE (stackPtr)->entry[page]);
    mmuTlbInvalidate (stackAddr);
    stackAddr += MMU_PAGE_SIZE;
    ++page;
    }
```

INCORRECT:
```
for (ix = 0; ix < 10; ++ix)
    {
    /* code omitted */

    if (array[ix] == TERM_VALUE)
        {
        ix = 9;
        }
    }
```

The for loop index variable should match the type of the termination variable:

CORRECT:
```
uint32_t end = 64U;
uint32_t ix;
for (ix = 0; ix < end; i++)

int32_t end = 64;
int32_t ix;
for (ix = 0; ix < end; i++)
```

INCORRECT:
```
uint32_t end = 64U;
int32_t ix;

for (ix = 0; ix < end; i++)
```

3.24 Unreachable or Dead Code

There shall not be any unreachable code. This may be in the code flow, or it could be a static function in the file that is not used.

INCORRECT:

```
STATUS valueCheck
    (
    uint32_t value
    {
    if (value == 0U) goto fail;
    goto fail;
    ...
    if (value > 256U) goto fail;
    ...

fail:
    return ERROR;
    }
```

3.25 Recursion

Recursion can only be used if there is a predictable and acceptable depth of recursion.

CORRECT:

```
int sum
    (
    uint_t num
    )
    {
    return (sumInternal (num, 0));
    }

int sumInternal
    (
    uint_t  num,
    uint_t depth
    )
    {
    if (depth > MAX_RECURSE_DEPTH) /* prevent stack overrun */
        {
        return (-1); /* negative value indicates error */
        }

    if (num == 0)
        {
        return num;
        }
    else
```

```
        {
        return (num + sum (num-1, depth));
        }
    }
```

INCORRECT:

```
int sum
    (
    uint_t num
    )
    {
    if (num == 0)
        {
        return n;
        }
    else
        {
        return (n + sum (n-1));
        }
    }
```

3.26 Automatic Variables

Automatic variables shall always be initialized before use.

CORRECT:

```
uint64_t sum
    (
    uint32_t num
    )
    {
    uint32_t ix = 0L;
    uint64_t retValue = 0ULL;

    for (ix = 1; ix <= num; ++ix)
        {
        retValue += (uint64_t) ix;
        }

   return (retValue);
   }

void foo
    (
    uint32_t v,
    uint32_t w
    )
    {
    uint32_t a;
```

```
    a = v * w * 16UL;
    bar (a);
    ….
    }
```

INCORRECT:

```
uint64_t sum
    (
    uint32_t num
    )
    {
    uint32_t ix;
    uint64_t retValue;

    for (ix = 1; ix <= num; ++ix)
        {
        retValue += (uint64_t) ix;
        }

    return (retValue);
   }
```

Sub-block scoped automatic variables are acceptable, with the proviso that the name of an automatic variable shall not eclipse (have the same name as) an automatic variable declared with function scope (or a global variable). Similarly an automatic variable declared with function scope shall not eclipse a global variable.

3.27 Processor and Architecture Specific Code

Processor architecture specific code shall reside in an architecture specific directory, not mixed in with generic code

CORRECT:

```
void workQAdd1
    (
    FUNCPTR func, /* function to invoke */
    int     arg1  /* parameter one to function */
    )
    {
    int key = intCpuLock ();

    JOB *pJob = (JOB *) &pJobPool [workQWriteIx];

    workQWriteIx += 4;   /* advance write index */

    if (workQWriteIx == workQReadIx)
```

```
        {
        workQPanic ();  /* leave interrupts locked */
        }

    intCpuUnlock (key);

    workQIsEmpty = FALSE;      /* we put something in it */

    pJob->function = func;     /* fill in function */
    pJob->arg1 = arg1;   /* fill in argument */
    }
```

INCORRECT:

```
void workQAdd1
    (
    FUNCPTR func, /* function to invoke */
    int     arg1  /* parameter one to function */
    )
    {
    asm volatile ("cli;");

    JOB *pJob = (JOB *) &pJobPool [workQWriteIx];

    workQWriteIx += 4;   /* advance write index */

    if (workQWriteIx == workQReadIx)
        {
        workQPanic ();   /* leave interrupts locked */
        }

    asm volatile ("sti;");

    workQIsEmpty = FALSE;      /* we put something in it */

    pJob->function = func;     /* fill in function */
    pJob->arg1 = arg1;   /* fill in argument */
    }
```

NOTE: Adding `#if(_VX_CPU_FAMILY == _VX_I80X86)` around the assembler doesn not convert this example into being CORRECT.

3.28 Restrictions on keywords

The use of the "goto" and "continue" keywords should be avoided. The use of goto for implementing error handling code is allowed. When using a "goto" the statement must jump forwards, not backwards.

CORRECT:

```
pTblEntry = pNewTcb->pCoprocTbl;

/* perform a restore operation for each context */

while ((pDesc = pTblEntry->pDescriptor) != NULL)
    {
    /* if a NULL context is encountered, skip to the next context */

    if (pTblEntry->pCtx != NULL)
        {
        if (pDesc->pCtxRestoreRtn != NULL)
            {
            (*pDesc->pCtxRestoreRtn) (pTblEntry->pCtx);
            }
        }

    pTblEntry++;
    }
```

Example 2:

```
...
goto taskCreateError1;
...
goto taskCreateError1;
...
goto taskCreateError2;
...
goto taskCreateError3;
...

taskCreateError3:
    (void) semDelete (semId);
taskCreateError2:
    if (_VX_IS_SYS_MODE_UP && (pTcb != NULL))
        {
        (void) taskVarCtl (tid, VX_TASK_CTL_VAR_DELETE,
                           (long *) &windTcbCurrent, &value);
        }
taskCreateError1:
    if (pTls != NULL)
        {
        free (pTls);
        }
    if (pTcb != NULL)
      {
        (void) taskParamCtl (tid, VX_TASK_CTL_UTCB_SET, NULL);
        free (pTcb);
        }
    (void) objDelete ((OBJ_HANDLE) tid, 0);
    return (TASK_ID_NULL);
```

INCORRECT:

```
pTblEntry = pNewTcb->pCoprocTbl;

/* perform a restore operation for each context */

while ((pDesc = pTblEntry->pDescriptor) != NULL)
    {
    /* if a NULL context is encountered, skip to the next context */

    if (pTblEntry->pCtx == NULL)
        {
        pTblEntry++;
        continue;
        }

    if (pDesc->pCtxRestoreRtn != NULL)
        {
        (*pDesc->pCtxRestoreRtn) (pTblEntry->pCtx);
        }

    pTblEntry++;
    }
```

Example 2:

```
loopAgain:
   while (working)
       {
       if (mustLoop) goto loopAgain;
       }
```

3.29 The Ternary (?) Operator

The ternary operator (?) can be used, but it is preferable to use if then else for complex operations.

CORRECT:

```
#define MAX(a, b) (((a) > (b)) ? (a) : (b))
```

INCORRECT:

```
((pVarCmd->value == (foo(x) * VAL)) ? (s = bar(x) / FOO) :
                    (fred = sin(y) * BAR);
```

3.30 The use of the static keyword

All functions and global variables used only in a single compilation unit shall be declared static. Do not export symbols that should not be accessible.

3.31 Complex Expressions

Avoid complex expressions. Complex expressions make it difficult to maintain and to certify.

3.32 Code Complexity Measurement

The McCabe cyclometric complexity metric should have a value of less than 20 per function.

- Cyclomatic Complexity measures the number of independent linear paths through a function
- See http://en.wikipedia.org/wiki/Cyclomatic_complexity for more information
- Coverity Prevent will be used to compute this metric.
- Value of 20 was chosen although various sources classify a metric of 10 as “complex” since OS code tends to contain more complex code than other types of software.

The complexity value can be obtained from the Coverity tool output, or by running the emacs script “pmccabe” or the Linux program “pmccabe”.

The following is an emacs function to return the mccabe cyclometric compleity on the function under point:

```
(defun mccabe-on-defun ()
  "Returns the McCabe cyclomatic complexity on function under point"
  (interactive)
  (save-excursion
    (save-restriction
      (narrow-to-defun)
      (mark-whole-buffer)
      (shell-command-on-region (point-min) (point-max) "pmccabe | awk '{ print
$2 }'"))))
```

You can bind this command to *C-cm* in *c-mode* with this command in my *c-mode* hook

```
(local-set-key "\C-cm" 'mccabe-on-defun)
```

3.33 Parameters of public APIs

Validate all input to ensure that it is in the expected form, does not contain unwanted data or executable characters to ensure that an overflow, improper access or other unexpected behavior cannot occur as a result of consuming the input.

Use exception handling or other programmatic methods to ensure proper handling.

For example, pointer parameters of public APIs must be checked for NULL

CORRECT:

```
STATUS wdStart
    (
    WDOG_ID wdId,               /* watchdog ID */
    _Vx_ticks_t delay,          /* delay count, in ticks */
    FUNCPTR pRoutine,           /* routine to call on time-out */
    _Vx_usr_arg_t parameter     /* parameter with which to call routine */
    )
    {

    if (pRoutine == NULL)
        {
        errnoSet (S_wdLib_INVALID_PARAMETER);
        return (ERROR);
        }

    …

    if (OBJ_VERIFY (wdId, wdClassId) != OK)
        {
        (void) KERNEL_UNLOCK ();
        return (ERROR);
        }

    if (delay == 0)
        {
        errnoSet (S_wdLib_INVALID_PARAMETER);

        (void) KERNEL_UNLOCK ();
        return (ERROR);
        }
```

3.34 Global Variables

All data that is used in a global fashion should be established through “set” and “get” functions unless for documented performance reasons.

- Refers to the programming best practice of “information hiding.” A module’s private data elements are accessed by get/set methods as opposed to direct access
- The VxWorks “funcBind” mechanism where global function pointers of the form _func_xxx are used to decouple components can still be used
- The global variable “errno” should not be accessed directly, instead use the functions errnoSet () and errnoGet ()

3.35 Return value from ISRs

When ISR objects are used, ISRs are dispatched from within an 'isrDispatcher()' function, in order to keep track of metrics about the ISR. The isrDispatcher() function inspects the return code from the ISR to see if the ISR returns OK. If the return code is OK, isrDispatcher() updates a 'serviceCount' field to note that the ISR has been serviced. For any other return code, the 'serviceCount' field is untouched.

The intent of this code within isrDispatcher() is to distinguish between ISRs that were invoked because their underlying hardware required service, and ISRs that were invoked only because they shared a common interrupt line with another device. For example, if both an ethernet device and a serial device share the same physical interrupt line, then the ISRs for each device will be invoked whenever either device asserts its interrupt. Each ISR is expected to analyze the current state of its hardware to determine if the hardware is in fact requesting interrupt service. If the device is requesting service, the ISR should perform whatever device-specific actions are required, and then return OK. If the device is not requesting service, the ISR should return ERROR.

3.36 Avoid alloca() Function

Many compilers support the alloca() function as an extension to the C language. This normally allocates storage from the stack, as any declared variable would. Since the storage area is on the stack, this area does not need to be freed, as the stack is restored upon exiting the function.

While it is normal to see a lot of usage of alloca() in code from Unix and Windows programming models it does not suit VxWorks very well. Both Unix and Windows support automatic stack expansion and stack checking. VxWorks does not. In the embedded world predictable timing and stack usage can be very important. Code for VxWorks should definitely avoid the use of the alloca() function. Allocate the storage directly on the stack, or use malloc() and free() if necessary.

3.37 Void Pointer Arithmetic

The ANSI standard does not allow this since the size of a void item is supposed to be unspecified. Some compilers allows it and assume the data size to be one byte, the same as a ‘char’ data type. The following code example is wrong.

INCORRECT:

```
{
void * pVoid;
pVoid += 1;
pVoid++;
pVoid = pVoid + sizeof(char);
}
```

The examples are all wrong because ANSI pointer arithmetic is based on the size of the object pointed to. In the case of a pointer to a void, the size is undefined.

CORRECT:

```
{
void * pVoid;
pVoid = (char *)pVoid + 1;
}
```

3.38 Use of Volatile and Const Attributes

Proper use of the volatile and const attributes can result in better error detection by the compiler. The use of volatile and const should be used consistently in a library. For example if const is used for one function in a library, it should be used appropriately for all functions in that library.

The volatile keyword is essential for all data elements whose value can be changed by an agent outside of the current thread of execution. This means by a device doing DMA, another task sharing the data, an interrupt function sharing the data, or hardware intervention such as a memory mapped hardware register. Failure to tag a shared data element with volatile can generate intermittent system faults that are very hard to track. It also requires the compiler to access the data atomically through a single instruction cycle where possible, and prevents caching.

The const attribute aids the toolchain in that it indicates that the data item can be placed in a read-only section of the executable. This can improve system footprint and performance.

The const attribute should also be used to indicate to the compiler that a function argument is strictly an input argument and that its value will be unchanged by this function. This helps the compiler to perform error detection and allows it to do better optimization of the code.

3.39 Misuse of the 'Register' Attribute

Do not use the “register” keyword or FAST macro. The keyword is ignored by modern compilers.

3.40 Variadic Functions

Variadic functions are those defined with a variable number of arguments. In old terminology it is varargs support. ISO has standardized a specification for use of variadic functions and the macro implementations for va_list, va_start, va_end, and va_arg. However, because the va_list is an opaque data type that is defined in several different ways, ISO does mandate that if a va_list data type is passed from one function to another, the calling function may not use that data item again after the called function is finished.

Normally when one function is passing a va_list data type to another function it is expected that the called function is the one doing all of the variable argument work. ISO demands that this is the case. For portability purposes, if a va_list data type is passed to a subfunction, the calling function may not reference that item any more. It must call va_end and terminate the item as soon as the subfunction call is completed.

This is necessary because of the variable nature of the actual data type. The implementation may pass the data by value or by reference. To be portable, we must not reference the item after it is passed to another function, in case it was passed by value.

3.41 VxWorks Macros

When building in the VxWorks environment, the keywords `__VXWORKS__` and `__vxworks` will be pre-defined.

The `_WRS_KERNEL` macro will be pre-defined when building for the kernel environment.

3.42 Toolchain Abstraction Macros

The following compiler abstraction macros must be used rather than directly using a tool chain specific #pragma or __attribute__.

3.42.1 _WRS_PACK_ALIGN (n)

Used to specify the packing and alignment attributes for a structure. Packing ensures that no padding will be inserted and that minimum field alignment within the structure is one byte.

The user can specify the assumed alignment for the structure as a whole with the argument n, in bytes. The value n is expected to be a power of two value (1, 2, 4, 8,...). The size of the structure will be a multiple of this value. If the overall structure alignment is 1, the compiler will assume this structure, and any pointer to this structure, can exist on any possible alignment. For an architecture that cannot handle misaligned data transfers, the compiler will be forced to generate code to access each byte separately and then to assemble the data into larger word and longword units.

The macro will be placed after the closing brace of the structure field description and before any variable item declarations, or typedef name declarations.

Always specify fields with explicit widths, for example, UINT8, UINT16, INT32, and so on. Do not use bitfields in a packed structure.

3.42.2 _WRS_ASM ("opcodes")

Used to insert assembly code within a C function declaration. The inserted code must not interact with C variables or try to alter the return value of the function.

The compiler is assumed to not optimize or re-order any specified in-line assembly code.

The use of _WRS_ASM ("") to prevent the compiler from re-ordering C code before the statement with C code that follows the statement, has been deprecated. Instead, please use the VX_CODE_BARRIER () macro for this purpose.

Example (PowerPC):

```
void foo (void)
    {
    functionA (args);
```

```
    _WRS_ASM (" eieio; isync;");
    functionB (args);
    }
```

Example (ARM):

```
/* inline intLock() */

_WRS_INLINE (int  inlineIntLock (void)
{
int key;
_WRS_ASM (
"mrs  r1, cpsr\n\t"
 "and    %0, r1, #(1<<7)\n\t" "orr    r1, r1, #(1<<7)\n\t" "msr    cpsr, r1"
: "=g" (key)
: /* no input */
: "r1");
return key;
} )
```

3.42.3 _WRS_DATA_ALIGN_BYTES(n)

Used in prefix notation to declare an initialized C data element with a special alignment. The argument n is the alignment in power of 2 units (1, 2, 4, 8, 16, ...). This is normally used only with initialized global data elements. Use with care and caution. Overuse of this macro can result in poor memory utilization. If large numbers of variables require special alignment it may be best to declare them in separate sections directly in assembler. The linker loader could then fit them together in an optimal fashion.

3.42.4 _WRS_ALIGNOF (item)

This macro returns the alignment of the specified item, or item type, in byte units. Most structures have alignment values of 4 which is the normal alignment of a longword data value. Data items or types with greater alignment values would return an appropriate alignment value which is expected to be a power of two value: 1, 2, 4, 8, 16, and so on.

3.42.5 _WRS_ALIGN_CHECK (ptr, type)

This macro returns a boolean value, either TRUE or FALSE. A TRUE value indicates that the pointer value is sufficiently aligned to be a valid pointer to the data item or type. The expected implementation is to examine the low order bits of the pointer value to see if it is a proper modulo of the alignment for the given type.

3.42.6 _WRS_UNALIGNED_COPY (pSrc, pDst, size)

This macro is a compiler optimized version of our standard bcopy operation. It moves a data block from the source location to the destination location. This macro allows the compiler to optimize the copy operation based on the data types of the pointers and the size of the block. This macro is

designed to be used in high performance situations and the size of the block is expected to be small. Misuse of the macro for other situations should be avoided.

3.42.7 _WRS_INLINE

This macro is used to identify an inline function declaration. The inline model used is that of a static inline . If the function is referenced in a module, the compiler has two choices. It can inline each reference, or it can insert a subfunction call to a local copy of the function. The choice is entirely made by the compiler and may be influence by the choice of compiler options.

The _WRS_INLINE macro is to be used as a prefix attribute and should precede the definition of the inline function declaration.

```
_WRS_INLINE int intLock(int) {*(0xFFF00100) = 0x1; }
```

3.42.8 _WRS_ABSOLUTE (name,value)

This macro declares the item name as an absolute symbol to appear in the output object module. The value of the symbol is the second argument. Internally, compilers treat the item as a constant integer value.

An absolute symbol's address must be a constant integer value. Host tools can extract absolute symbols from object modules and use that data instead of data from a header file which may or may not be valid for the object module. The main purpose here is ensuring that tools are synchronized with the data in the module, and not data in an external header.

To declare an absolute symbol, use the macro _WRS_ABSOLUTE(name,value). This macro is to be used like a C statement. It will declare the name as an absolute symbol and assign a value to it. It should be followed with a semicolon like a regular statement.

```
_WRS_ABSOLUTE(test1, TEST1ID);
```

To support linking and certain compiler limitations, absolute symbol declarations need to be defined within a function declaration. The special macros _WRS_ABSOLUTE_BEGIN and _WRS_ABSOLUTE_END should be used to surround the definition of absolute symbols with the proper function declaration and assembler code. The absolute begin macro takes an argument that is used to partially form the name of the declared function. This argument is usually appended to the string _absSymbols_. In the following example, the external function that is created is _absSymbols_Common.

```
_WRS_ABSOLUTE_BEGIN(Common)

_WRS_GEN_OFFSET(WIND_TCB,taskId);

_WRS_ABSOLUTE(_vxWorksKey, _VXWORKSKEY);

_WRS_ABSOLUTE_END
```

3.42.9 _WRS_GEN_OFFSET (Struct,field)

This macro is used to generate an absolute symbol which has the value of an offset of a field in a structure.

```
_WRS_ABSOLUTE(_vxWorksKey, _VXWORKSKEY);
```

3.42.10 _WRS_DEPRECATED (“msg”)

This macro is used to issue a warning when the user calls the associated function. The message argument is expected to be a string constant to be displayed when a reference is encountered. The message is printed as a warning. If the API is actually not found, an unresolved global error during the link phase will typically terminate the project build process. The message itself is only a warning.

If the compiler system cannot provide this functionality, the macro will have no effect at all.

For example:

```
int intObsolete (int a)
    _WRS_DEPRECATED("use intNew instead");

int intOld (int oldArg) \
    _WRS_DEPRECATED("pls use intNew");
```

3.42.11 _WRS_LIKELY, _WRS_UNLIKELY

The macros _WRS_LIKELY and _WRS_UNLIKELY assists branch prediction. These macros wrap conditionals and suggest to the compiler the likelihood of one result or another. The following example will suggest to the compiler that the result of the conditional is likely true.

```
if (_WRS_LIKELY (status == OK))
```

3.42.12 _WRS_READ_PREFETCH (ptr), _WRS_WRITE_PREFETCH (ptr)

These macros are used to reduce cache misses by prefetching data into cache prior to access. The amount of data fetched is a single cache line, which is architecture and implementation dependent. Use this with care, as it is possible to damage cache performance in unexpected ways.

3.42.13 _WRS_USAGE_WARNING ("text")

This macro is used to provide a compile time warning when some contruct is used when it should not be. The warning here can appear as a deprecated component, as this macro uses the _WRS_DEPRECATED macro under the covers.

```
extern void spinLockIsrNdInit (spinlockIsrNd_t * spin)
    _WRS_USAGE_WARNING("since system has been configured for " \
    "determinism: VSB option DETERMINISTIC = 'y'");
```

3.43 Complex Macros

Complex function like macros should be avoided. An inline function or preferably a regular function should be used in preference to a macro.

INCORRECT:

```
#define foo(x, y, z) \
    if (x == 4) \
        { \
        dofoo (y, z); \
        } \
    else \
        { \
        dobar (y, z); \
        } \
    dofoobar (x);
```

4 Secure Coding Rules

4.1 Banned functions

Many security vulnerabilities in C are caused by buffer overruns and underruns. Some functions in the traditional C library have an API that is designed in a way that does not help the programmer avoid such mistakes in a good way. For this reason the C11 standard contains updated APIs (in Annex K) for many C library functions. The following subsections provide guidance on what APIs that are banned in new code. Recommendations are also given for what APIs to use instead. It is understood that there is existing code that use the old APIs. The legacy code will be updated to comply with the recommendations in this chapter in an incremental fashion. For example, if a new function that compare two strings needs to be added to an existing file, it is only the new function that must not use strcmp. The new function should call strncmp to compare the strings. Converting other, pre-existing calls to strcmp in the file, is not required. Separate features will be created to convert the existing code base.

Most functions in Annex K return a result, whereas the 'original' C library function does not. In most situations a failed call to an Annex K function is an indication of a bug (invalid arguments), and should never happen. In order to facilitate migrating legacy code to use the Annex K functions, it has been decided to terminate the caller in case an Annex K function detects invalid arguments. This will remove the need to add error handling to legacy code, since a failed call to Annex K functions will not return. The fact that the call will not return will make any bugs highly visible and ensure that they are fixed in a quick manner. When invalid arguments are detected an ED&R fatal event will be injected. If the call is made in RTP space, the RTP will be terminated, and in kernel space the kernel task will be terminated or the system will be rebooted. An ED&R event injection is done using the Annex K constraint handler.

There are situations when invalid arguments are not an indication of a bug. One example is when the argument is received from an untrusted source, e.g. data received over the network or from reading a file. Such data can be controlled by an attacker. Therefore such data cannot be passed to Annex K functions without proper sanity checks. Without such checks, the attacker could craft data in such a way to make the arguments checks fail and in turn make RTPs terminate or the system reboot. This would then constitute an attack vector for denial-of-service attacks.

In the table below there are separate columns for trusted vs untrusted arguments. For some banned functions there are non-Annex K functions that can be used in the processing of untrusted data. For the banned functions that do not have a recommended API for untrusted data, proper sanity checks of the arguments must be performed prior to calling the Annex K functions, to ensure that the arguments are valid.

All new functions that process untrusted data must have fuzz testing implemented.

4.1.1 String Handling Functions

DON'T USE	RECOMMENDED USE (VxWorks), trusted arguments	RECOMMENDED USE (VxWorks), untrusted arguments[1]	RECOMMENDED USE (Hypervisor)
strcpy ()	strncpy_s ()	strlcpy ()	strlcpy ()
strncpy ()	strncpy_s ()	strlcpy ()	strlcpy ()
strcat ()	strncat_s ()	strlcat ()	strlcat ()
strncat ()	strncat_s ()	strlcat ()	strlcat ()
strlen ()	strnlen_s ()	strnlen ()	strnlen ()
strnlen ()	strnlen_s ()		N/A
strcmp ()	strncmp ()	strncmp	strncmp ()
strtok ()	strtok_s ()		strtok_r ()
strtok_r ()	strtok_s ()		N/A
strerror ()	strerror_s ()		strerror_r ()
strerror_r ()	strerror_s ()		N/A
gets ()	fread(stdin,…)	fread(stdin,…)	N/A
gets_s ()	fread(stdin,…)	fread(stdin,…)	
getenv ()	getenv_s ()		N/A

Note that strlen() should not be used to calculate any argument to the functions above. The sizeof operator should be used for statically allocated buffers. In the case of dynamically allocated buffers the size of the buffer should be used. Exceptions can be made for string literals, but in most cases the sizeof operator can be used.

4.1.2 Input Format Conversion Functions

DON'T USE	RECOMMENDED USE (VxWorks), trusted arguments	RECOMMENDED USE (VxWorks), untrusted arguments[2]
scanf ()	scanf_s ()	scanf[3] ()
fscanf ()	fscanf_s ()	fscanf [4]()
sscanf ()	sscanf_s ()	sscanf[5] ()
vsscanf ()	vsscanf_s ()	
vfscanf ()	vfscanf_s ()	
vscanf ()	vscanf_s ()	

[1] If no recommended use exists, proper argument validation must be performed before calling an Annex K function. See above for further details

[2] See 1) above

[3] The maximum field width must be specified in the format string for all arguments of variable size, e.g. %ns where n is the size of the buffer to store the string in.

[4] See 3) above

[5] See 3) above

4.1.3 Formatted Output Conversion Functions

DON'T USE	RECOMMENDED USE (VxWorks), trusted arguments	RECOMMENDED USE (VxWorks), untrusted arguments[6]
`printf ()`	`printf_s ()`	
`fprintf ()`	`fprintf_s ()`	
`sprintf ()`	`snprintf_s ()`	`snprintf ()`
`snprintf ()`	`snprintf_s ()`	`snprintf ()`
`vfprintf ()`	`vfprintf_s ()`	
`vsnprintf ()`	`vsnprintf_s ()`	
`vsprintf ()`	`vsprintf_s ()`	

4.1.4 Time Transformation Functions

DON'T USE	RECOMMENDED USE (VxWorks), trusted arguments	RECOMMENDED USE (VxWorks), untrusted arguments[7]
`asctime ()`	`asctime_s ()`	
`asctime_r ()`	`asctime_s ()`	
`ctime ()`	`ctime_s ()`	
`ctime_r ()`	`ctime_s ()`	
`gmtime ()`	`gmtime_s ()`	
`localtime ()`	`localtime_s ()`	

4.1.5 Memory Modification Functions

DON'T USE	RECOMMENDED USE (VxWorks), trusted arguments	RECOMMENDED USE (VxWorks), untrusted arguments[8]
`memcpy ()`	`memcpy_s ()`	
`memmove ()`	`memmove_s ()`	
`memset ()`	`memset_s ()`	

4.2 Remove Sensitive Information from Memory

Sensitive information is data that should be protected from unauthorized access. Examples of sensitive information are passwords, encryption keys, pre-shared keys and public key-pairs. Once sensitive information is no longer needed, it shall be cleared from memory. If the sensitive information were left in freed memory, the next time that memory is used it may be handed out to some SW with malicious intent. Furthermore, if the information resides longer than necessary in

[6] If no recommended use exists, proper argument validation must be performed before calling an Annex K function. See above for further details

[7] See 6) above

[8] See 6) above

memory, it increases the risk of an attacker exploiting some security vulnerability and getting hold of the sensitive information.

The clearing of the sensitive information should be done by calling memset_s, as this function is guaranteed not to be optimized out by the compiler/linker.

4.3 Prevent User Space from Obtaining Unauthorized Information

Sensitive information should not be inadvertently made available for user space applications. Special care should be taken so that residual information is cleared before memory is made accessible to user space. For example buffers that are reused, or freed and then reallocated, or unmapped and re-mapped should be cleared so that content that was written during the previous use cannot be read by another application.

Pointer values such as kernel object IDs, kernel function pointers, should not be passed to user space. User space should be oblivious to the kernel memory layout.

Register values (integer, floating point or vector) set during the execution of kernel space code or of another application should be cleared. Normally this is taken care of by the kernel context switch, system calls and signal delivery functions.

4.4 Never Trust Information Passed From User Space to Kernel

Information passed from user space to kernel space via system call arguments, shared memory, or messaging must be validated before it is used to perform work in the kernel. Pointers should be validated to ensure that they reference valid application-accessible memory with the proper read and or write privilege. Make sure that, when information passed has multiple indirections (for instance pointer argument to a structure that itself contains pointers or other data that needs validated), validations are performed for all levels.

4.5 Sensitive Information Must be Stored in the VxWorks Vault

Sensitive information that needs to be stored persistently must be stored in the VxWorks Vault. Examples of sensitive information are passwords, encryption keys, pre-shared keys and public key-pairs. Note that only passwords that are needed in clear, e.g. passwords used to log on to other systems, should be stored in the VxWorks Vault. Passwords that are needed to verify login requests should be stored in the user database in the USER_MANAGEMENT layer (which stores the passwords in hashed format).

See the reference entries for secSecretLib and secKeyStoreLib for further details.

4.6 Don’t Provide Hints to Hackers

Error messages that are shown in public interfaces must be reviewed carefully not to reveal information that can be of use for hackers. For instance, an error message regarding failed authentication should simply state “Failed login”, regardless of the reason for the failed login. Having separate error messages for the case of user account not defined and incorrect password would help an attacker to determine what user accounts that are defined in the system.

5 C++ Coding Rules

The C++ code should follow the C coding rules.

In addition, the C++ Core Guidelines document [Ref.1] should be followed when writing any C++ code.

6 Assembler Coding Rules

The following conventions define rules to follow when writing assembler code.

Assembler code should

- not be self-modifying.
- not use calculated jumps or branches.
- have a single entry point.
- minimize multiple return points.
- be simple, avoiding multiple nested conditionals.
- have structured branching in and out of code blocks.
- return to the calling function (in most cases).