MISRA C Technical Clarification – 25th July 2000

Introduction

This document clarifies issues raised on the **interpretation** of the MISRA document "Guidelines For The Use Of The C Language In Vehicle Based Software" published in April 1998.

This document is based on the questions raised in the above document on the MISRA-C on-line discussion group.

This document is limited to clarifying the intention of the rules in sufficient detail to allow consistent static enforcement of the guidelines, and therefore aid the definition of a definitive standard. We have attempted to not extend the definition of MISRA C in this document. Many useful issues have been made, and these will be addressed in future documents, primarily to deal with the impact of C99 as the issues become know.

The authors would like to thank all those who participated in these discussions. While the authors of MISRA C have carefully studied the comments made, we have reserved the right to retain our intention of the rules. We acknowledge that some participates disagreed with, or questioned the value of some of the rules.

The author acknowledges and thanks the following for their help in developing this document.

Stephen Parker
Derek Jones
Les Hatton
David Ward
Paul Edwards

Author
Gavin McCall

Discussion on individual rules

Rule 1

The guidelines were written based on ISO9899:1990. No consideration is made in either the guidelines, or in this clarification of the ISO9899:1999 version of the standard. No claim is made to the suitability of these guidelines with respect to this new version of the standard. Any reference in this document to 'Standard C' refers to the older standard.

The guidelines address 'freestanding implementation'.
As such, only the following library usage can be made, <stdarg.h>, <stddefs.h>, <limits.h>, <float.h> and <iso646.h>. Use of other library files requires deviations. Rules 114 through 127 should be applied as appropriate.

This rule provides a means to normalise all applications to the restrictions of a conforming environment. Therefore, the use of any extra features of a particular compiler can be considered and recorded via the deviation mechanism.
A product's software should ideally be claimed to be MISRA C compliant. In reality, this is more likely to be a claim that this product's software is MISRA C compliant subject to these specific and justified deviations. This rule provides a mechanism to document via deviations the necessary real-world implementation issues.

Rule 3

Assembly code and C code should not be mixed. Real-time chronometric, size and other issues may require the use of assembly code. Where such a need is required, the mixing of the languages should be via a well-defined interface. Consideration to robust means of parameter passing should be given, and the facilities of the tools for address resolution should be used in preference to manual techniques. The rule states a means that may achieve this, but other techniques may be applicable.
Additionally, the encapsulation of non-standard features is recommended. This allows the identification of machine-independent and machine dependent code. The latter would need considerable modification if moved to new hardware.
The use of inline assembler for enable and disable interrupt is an example were no data is passed, and the assembler code structure is linear code. Therefore the simplicity can be used to justify this method over the alternative to invoke a function call to achieve the same effect. This is made not withstanding the fact that the enabling and disabling of interrupts is an exceptional action, and its use should be limited to a well-designed interrupt system. The authors in no way advocate the general and blasé use of enable and disable interrupts across the application code, their use should be carefully designed, coded, and tested.
The use of in-line assembler violates rule 1, and must be covered by a deviation.

Rule 6

Character-constants and string-literals shall only contain characters that map to a defined subset of ISO 10646-1. This is because characters not covered by the standard may not be portable between implementations.

Rule 7

The authors of the guidelines did not consider Technical Corrigenda 2 or Amendment 1. Given that ISO9899:1999 has now been issued, it is not intended to consider these omitted documents. Instead, MISRA will consider the new version of the standard at a time in the near future.
The direct presence of defined trigraphs should be detected and reported. A MISRA C compliant static analysis tool will perform a test for defined trigraphs. The text provides a means to perform a more general check for all '??' sequences, using standard text searching tools. This rule is not intended to forbid '??' sequences which are not defined trigraphs, and such sequences should not be reported.
The text advice to use a compiler switch to disallow the use of trigraphs is withdrawn. The compiler should be operated in its strict ISO mode.

Rule 9

Standard C provides no concept of nested comments. A second successive opening comment marker is simply ignored

as part of the comment. We are effectively saying that nested comments *should* be recognised by the tool and then flagged as an error, because this may indicate a programming error. Consider the following code fragment.

```
/* some comment, closing comment accidentally omitted

<<New Page>>

Perform_Critical_Safety_Function(X);

/* the execution of the above line ensures safe function */
```

Reviewing the page containing the call to the function, the assumption that it is executed code, and not a comment is unclear. This rule allows a means to flag such cases.
Further to this, static analysis tools should provide visualisation of the code to provide a means to highlight code, comments, and conditionally ignored code. Colour may be an appropriate means to achieve this.

Rule 10

MISRA C does not prohibit the use of conditional compilation as a means to perform well-planned configuration management. If conditional compilation is used, then compliance is required for specific combinations of build options. Each build option configuration should be conformant with the rules of the document. (Tools should make it very clear which sections of "code" were not checked.)
During Test phases, exactly the same configuration issues exist has to which version as been tested.

Rule 12

The exception highlighted is normative. The members of either structures or unions may be reused within other structures. This is part of the rule, and tools should not flag such occurrences.

The example below is considered a violation of this rule.

```
struct {SI_16 name; } record;
SI_16 name;

record.name = 17;
name = 17;
```

record.name and name could accidentally be used in place of each other.

Rule 13

This rule intends that a set of typedefs should be used once to define a set of minimal length specific types using typedefs in terms of the basic types. All subsequent type references (definitions or casts) should use these typedef definitions.

Example 1:

```
typedef int int_16a;

#define int int_16a

int foo_bar;
```

The #define creates a macro with the same name as a keyword. Overloading the keyword in this way is confusing, and is against the intention of this rule.

Example 2:

```
#define int_16b int
```

```
int_16b foo_bar;
```

The rule states that the types will be defined using typedef statement. This example uses macro substitution, and therefore breaks the rule.

Compliance with this rule may not prevent all size related portability issues. Default promotions and implicit casting need to be evaluated for each implementation. Explicit casting in one implementation may cause redundant casting in a different application.

Rule 15

This rule is not intended to be statically enforceable. This is a compiler selection issue, and not a source code issue.

Rule 16

This rule is to prevent the direct manipulation of the bits of the representation by the application programmer. Use the library functions and operators supplied by the compiler vendor for the manipulation of floating-point numbers.

Rule 17

Typedef names shall not be reused as either other typedef names or for any other purpose. This rule is limited to all the files within the system for which we have C source, for a given project.

Rule 18

This advisory rule is related to Rule 43. Numeric constants have a type, and by using the appropriate suffix, type and sign changes can be avoided. The recommendation concerning the use of L in preference to l is not considered normative, and is not part of the rule.
The most useful application of the rule is when using unsigned types and constants as shown below.

```
unsigned int x = 27u;
unsigned int y = 27; /* static analysis can determine that this signed constant
can be converted to an unsigned constant with no loss of value – no warning
required. */

x = x + 25u;  /* unsigned expression */
y = y + 25;   /* y is converted to signed, added to 25, and the result cast to
unsigned. – y may be a signed value which is changed by the cast – warning
required. */
```

27 is a signed constant, and an implicit cast is required to covert this to unsigned.

Rule 21

The terms outer and inner scope are defined as follows. Identifiers that have file scope can be considered as having the outermost scope. Identifiers that have block scope have a more inner scope. Successive, nested blocks, introduce more inner scopes.

The rule is only to disallow the case where a second inner definition hides an outer definition. If the second definition does not hide the first definition, then this is not a violation of this rule.

Rule 22

The term function scope is the block scope of a function definition. This rule was intended to reduce definitions at file scope, and to encourage definitions at the ‘outermost block scope’ of functions. It is not intended to forbid definitions in inner scopes of functions. The presence of a static file scope definition which is only used within one function could

be moved the function scope of the function which uses it.

Rule 24

The code example in the guidelines is incorrect. The following code example is correct.

```
static UI_8 x;

void f(void)
{
extern UI_8 x; /* matches against file scope and therefore has static
storage class */

    {
    extern UI_8 x; /* matches against the block scope declaration
                    * which is hiding the file scope declaration.
                    * So it does not have the same linkage as the
                    * prior declaration.
                    * Has external linkage.
                    */
```

Rule 27

Each external object should only be declared in a single header file. The external objects across a project can be declared in one or more header files. This rule is not concerned with the number of header files, but with the number of declarations of any one given external object.

Rule 29

The following example violates this rule.

```
/*
 * (c) Copyright 1999 Knowledge Software Ltd
 *
 * www.knosof.co.uk
 *
 * Permission is given to use and distribute this source
 * code provided this copyright notice is retained.  This
 * copright notice must also be retained in any derived works.
 *
 *
 * KNOWLEDGE SOFTWARE DISCLAIMS ALL WARRANTIES WITH REGARD TO
 * THIS CODE.  IN NO EVENT WILL KNOWLEDGE SOFTWARE BE LIABLE FOR
 * ANY DAMAGES, LOST PROFITS, OR LOST SAVINGS.
 */

/*
 * misra29.c
 *
 * 27 Jun 99
 */


#include "misra.h"


struct I_TAG { MY_INT i; };

/*
```

```
 * As well as violating a MISRA rule the following
 * is not conforming C either.
 */
struct I_TAG a = { "abc" }; /* complain */
```

Rule 35

There shall be no assignments in conditional expressions of control statements.

Rule 36

There shall be no bitwise logical operators (&, |, or ~) in conditional expressions of control statements.

Rule 40

If the *sizeof* operator is used on an expression, then under the incorrect assumption that the expression will be executed, it should not contain any side effects.

```
int i, j;

j = sizeof(i = 1234);

/* j is set to the sizeof the type of i which is an int */
/* i is not set to 1234.                                */
```

Rule 43

Consider the following code examples.

```
typedef unsigned char U8;
typedef unsigned int  U16;

int main(void) {

   U8 i, j, k; /* unsigned 8 bit significance, implemented as unsigned char */
   U16 m;      /* unsigned 16 bit significance, implemented as unsigned int */

   m = 0u;

   i = 255u;        /* Note  1 */
   i = 257u;        /* Note  2 */
   i = m;           /* Note  3 */

   j = (U8) 255u;   /* Note  4 */
   j = (U8) 257u;   /* Note  5 */
   j = (U8) m;      /* Note  6 */

   if ( m < 256u )
   {
      k = m;       /* cast not necessary due to clip */ /* Note 7 */
   }
   else
   {
      k = 255u;     /* Note 1 */
   }
}
```

Note 1: Constant expression, which is not changed, no warning.
Note 2: Constant expression, which is implicitly changed, WARNING REQUIRED
Note 3: Non-constant expression, implicit cast, WARNING REQUIRED
Note 4: See Note 1. This is an unnecessary, but strictly not redundant cast.
Note 5: Constant expression, which is explicitly changed, WARNING REQUIRED
Note 6: Non-constant expression, which is explicitly cast, no warning.
Note 7: Non-constant expression, which is implicitly, cast, WARNING REQUIRED, except if flow analysis is performed, it can be statically determined for this example that this cannot be reached. This warning is accepted as being a false warning in this case. The purpose of the warning is to prompt review.

In Case 6, the presence of the (U8) is sufficient to indicate to the reader of the code that explicit casting is occurring.

The cases of the assignment-operators *= /= %= += -= >>= &= ^= and |= are addressed below.

```
U8 u; /* or any type smaller than |int|. */

u += 8u; /* Note 8 */
```

Under the "as-if" rule could be described by the following.

```
u = (u + 8u); /* therefore see Note 3 */
```

This would need to be written as the following to meet this rule.

```
u = (U8) (u + 8u); /* see Note 6 */
```

In many cases, the compiler used may use byte-based arithmetic for the actual implementation, so the "as-if" is not the actual implementation.

Note 8: The use of assignment-operators is a special case. This is considered to be an expression in the type of unsigned integer. Care should be taken to prevent unwanted modulo arithmetic. Assignment-operators with a lhs of a type smaller than int will always have a rhs of a type of the size of int, and will also have the potential for the value being changed due to modulo arithmetic. All non-constant arithmetic expressions (including those in a single type) have the potential to overflow, and therefore no special case will be made for the assignment-operator. Explicit casting cannot be used, and it is not the intension of MISRA to require a warning message for every such occurrence.

Rule 44

```
double f;

/* Intention is to set f to the value 0.3333333 */
f = (double) 1.0/3.0;        /* Warning - redundant cast */
f = 1 / 3;                   /* Wrong, f = 0.0 */
f = (double) 1 / 3;          /* Correct */
f = 1 / (double) 3;          /* Correct – cast is better on first term */
f = (double) 1 / (double) 3; /* Warning – 2nd cast is an unnecessary cast, the
term will be balanced */
```

Redundant casts shall be identified. A redundant cast is casting an expression to a type that it already is.
Unnecessary casts should also be identified. An unnecessary cast is performing a type change that would occur due to operator balancing.

Rule 45

The use of the phrase "cast" is to include both casting and conversion of pointers.

Consider the following code.

```
short * short_ptr;  /* short is 16 bits aligned*/
```

```
int * int_ptr;   /* int is 32 bits aligned */

int_ptr = (int *) short_ptr;
```

The pointer short_ptr may be pointing to a parameter that cannot be converted to an int aligned parameter. This is a specific example of how changing pointer types can cause problems.

Rule 46

In the example,

```
i = 0;

j = func( i++, i++);
```

This can result in either j = func (0, 1);, or j = func (1 ,0);, depending on the order of evaluation of the parameters of the function call. The call of the function is a sequence point, but the evaluation of both parameters occurs before this sequence point.

Rule 47

When an expression contains several operators, and these operators have different levels of precedence, parentheses should be used to distinguish the order of precedence. Therefore when operators have equal precedence, parentheses is not required - order of evaluation is left to right (except assignment).

```
x = a + b - c + d; /* equal precedence */

x = ((a + b) - c) + d; /* This is cluttered and not required, + has equal
precedence, and evaluation is from left to right. */

x = a + b – (c + d); /* equal precedence */

x = (a + b) – (c + d); /* This is not required, (a + b) is excessive. */
```

Rule 48

In the given example, 3/2 = 1 = 1.000 may be the intended function.

Consider the following mixed precision expressions.

```
int main(void)
{
   int i_three = 3;
   int i_two = 2;
   float f_two = 2.0f;
   float f_answer1;
   float f_answer2;
   float f_answer3;

   f_answer1 = i_three / i_two * f_two;

   f_answer2 = (float) i_three / i_two * f_two;

   f_answer3 = f_two * i_three / I_two;

   return((int) f_answer1);
}
```

f_answer1 = 3 / 2 * 2.0; integer 3 divided by integer 2 is integer 1.
f_answer1 = 1 * 2.0; integer 1 is balanced with float 2.0 to type float. Float 1.0 multiplied by float 2.0 is float 2.0.

f_answer2 = (float) 3 / 2 * 2.0; integer 3 is cast to float 3.0
f_answer2 = 3.0 / 2 * 2.0; integer 2 is balanced with float 3.0 to type float. 3.0 / 2.0 = 1.5.
f_answer2 = 1.5 * 2.0; float 1.5 multiplied by float 2.0 is float 3.0.

f_answer3 = 2.0 * 3 / 2 = 6.0 / 2 = 3.0.

In a mixed precision expression, all operators shall operate on the same data type. Either cast or locate this data type for the first item in the list. Use separate statements, each with a separate precision if different precisions are required.

```
int t_int = 3/2;
float f_answer = f_two * t_int;
```

Rule 49

The results of the relational operators (< > <= >=), equality operators (== !=), and the logical operators (&& ||) produce 'effectively Boolean' results. These results do not need to be compared against not zero.
If the following is written, a is of type int;

```
if (a = 2) ..
```

The tool should complain, since the correct form should have been either

```
if ((a = 2) != 0).. /* Assignment was required, or */
if (a == 2)..       /* A Boolean expression */
```

If the following is written, a is of type int;

```
if (a)..
```

This should always be written as

```
if (a != 0)..
```

The !=0 will be removed by the compiler, since it makes no difference to the expression.

Rule 50

A test of the form if (! ((x < 0.0) || (x > 0.0))) for if (x == 0.0) breaks the intent of this rule. It is unlikely that a tool can detect such violations.

Rule 52

The following code contains two examples of unreachable code. This first is due to a constant expression in an if statement. The second example is a line of code following the break statement with the switch statement. The variables i, j and tlight can become corrupted, and therefore code taken based on unexpected values can be reached. Any non-constant expression should be considered subject to corruption.

```
int i = 1;
extern int j; /* set to 1 */
enum colour {red , amber, green};
enum colour tlight = red;

if (1)
{
   /* reachable code */
}
else
{
   /* unreachable code */
}

if (i != 0)
{
   /* reachable code */
```

```
   }
   else
   {
      /* unreachable code, i was set to 1 in the declaration of i, it is not
volatile, and data flow analysis shows that no lines have changes i since it was
set. */
   }

   if (j != 0)
   {
      /* reachable code, j can be changed by external code. */
   }
   else
   {
      /* reachable code */
   }

   switch(tlight)
   {
      case red   : break;
                   tlight = green; /* this line is unreachable */
                   break;
      case amber : break;
      case green : break;
      default: /* here goes reachable, defensive code to correct in
                  the event of corruption of tlight.  */
                   break;
   }
```

Consider the code fragment below.

```
   unsigned int i;

   /* ... */

   switch (i & 3u)
   {
      case 0: /* something */ break;
      case 1: /* something */ break;
      case 2: /* something */ break;
      case 3: /* something */ break;
      default:
         /* required by rule 62, suitable comment here */
         break;
   }
```

In this example, the variable i can take any integer value prior to the switch statement. The statement i & 3u performs the operation `i mod 4`, which results in values between 0 and 3. But in most cases, this would mean performing the mod 4 on a garbage value, and treating the result as data. If i is supposed to take values between 0 and 3, the mod operation only masks any error. Perform the switch on i and allow the default clause to detect values outside of the correct range. The code example above uses a very poor coding style. The following is better, using the default clause to perform corrective action.

```
   unsigned int i;

   /* ... */

   switch (i)
   {
      case 0: /* something */ break;
      case 1: /* something */ break;
      case 2: /* something */ break;
```

```
      case 3: /* something */ break;
      default:
         i = ..;
         /* i should be in the range 0..3, set i back to this range, and perform
corrective action as required.*/
         break;
   }
```

Alternatively, see rule 62 for the case where all values of i have meaning.

Rule 53

All non-null statements shall either have a side–effect or change the program flow. The goto, continue, break and return statements have an effect in terms of changing program flow.

Rule 54

Computers and tools are not confused by the location of the semi-colon in a line of text. This is strictly a human interpretation error. This rule should therefore be applied to the C source before any translation.
A null-statement should appear on a physical line by itself. There should only be space or tab characters on that line.

```
/* comment */ \
      ;
```

The above case of preceding the null-statement with a continuation character is not a violation, since the null-statement is on a physical line by itself.
The purpose and intent of the continuation character in this situation is of no value, and this may be addressed by subsequent rules.

```
#define NOTHING

if ( <condition> )
{
   NOTHING;
}
```

This is an example of a macro to deliberately contain no text.
If after translation phase seven, a null statement is found, then it shall correspond to a line containing no text other than space and tab in the source code. Therefore the above breaks this rule. The following is required.

```
#define NOTHING

if ( <condition> )
{
   NOTHING
   ;
}
```

This would only be of use if NOTHING could in fact be a valid statement in another build configuration. In that case NOTHING is set to a valid statement. Such practice would not be recommended.

Rule 59

This rule should also be applied to switch statements.

Rule 61

A non-empty case clause is the sequence of statements, including the labelled statement and terminating at the next

case label, default label, or the end of the switch statement."
An empty case clause is the label followed immediately by either another label, or by the default label.

Rule 62

It is required that the default clause be the final clause in the switch statement. The code example in rule 52 can be modified to the following.

```
unsigned int i; /* assume int is 16 bit, 0..65535 */

/* ... */

/* all values of i have meaning, i cannot take an illegal value */
switch (i & 3u)
{
   case 0:  /* handle (i mod 4 == 0) = 0, 4,  8,..,65534), something */
   break;
   case 1:  /* handle (i mod 4 == 1) = 1, 5,  9,..,65533), something */
   break;
   case 2:  /* handle (i mod 4 == 2) = 2, 6, 10,..,65534), something */
   break;
   default: /* handle (i mod 4 == 3) = 3, 7, 11,..,65535), for rule 62 */
   break;
}
```

This example satisfies both rules 52 and 62.

Rule 63

A switch statement should not be used for cases of only two cases. (This would be a case label and a default label.) Such a switch statement can be describe using an if then else statement which is the usual form.

Rule 65

The problem is that people (might) forget that floats are inexact representations of numbers. Particularly if exact value is then tested – violation of another rule.

```
int carry_on;
float a_floating_obj;

/* later in the program */

while (carry_on != 0)
{
   if (a_floating_obj > 1.2)
   {
      carry_on = 0;
   }
   /* etc */
```

In this example, carry_on is the loop control variable. The setting of the loop control variable may be based on a well-formed expression involving floating types.

Rule 66

'Loop counters' and 'iteration counting' are effectively the same thing. 'Concerned with loop control' is a larger category consisting of:

1. 'Loop counters'
2. Flags

(The rule uses the term 'other loop control variables' – we think this is only the use of flags).

Consider the following functions.

```
int f1(int x)
{
   int i;
   int j;

   j = 0;

   for ( i = 0; i < x; i++)
   {
      j = 2*j + i*x;
      printf("i=%d, j=%d\n", i, j);
   }
   return(j);
}

int f2(int x)
{
   int i;
   int j;

   for ( i = 0, j = 0; i < x; j = 2*j + i*x, printf("i=%d, j=%d\n", i, j),i++)
   {
      ;
   }
   return(j);
}

int f3(int x)
{
   int i;
   int j;

   i = 0;
   j = 0;

   for (; i < x;)
   {
      j = 2*j + i*x;
      printf("i=%d, j=%d\n", i, j);
      i++; /* breaks rule 67 */
   }
   return(j);
}
```

Function f1 is considered the correct use of a for statement. The loop control variable i is initialised, tested, and incremented in the for loop.
Function f2 shows additional statements than just loop control being performed within the loop control structure. This extreme example moves the entire function into the for statement. This coding practice is not recommended.
Function f3 shows the loop control variable being initialised and incremented by code outside of the control statement.

Rule 67

This rule is subjective and may not be statically checkable in this form. See example in function f3 in clarification of rule 66.

Rule 72

We did intend this rule to be applied within the user written code. Library header files may require a deviation. The standard describes compatible types, for which conversions are made silently. Since these silent changes may result in loss of precision and/or range, MISRA require that "compatible" is not sufficient.

Rule 82

We intended the functions single exit point to be the textual end of the function.

Rule 83

Functions with non-void return type shall not terminate by reaching the implicit return when a function terminating brace is reached. It shall have an explicit return statement before this point.

Rule 90

By function-like macro the standard definition in section 6.8.3 of the C standard. Wider issues exit, which are not discussed in the release.

Rule 93

This rule as currently worded is not statically checkable.

Rule 101

An array index is not considered to be pointer arithmetic.

Rule 102

Structure member de-referencing, high-dimensioned arrays, and linked lists, are all examples of situations where pointer indirection of greater than 2 does not necessarily lead to poor readability of the code (usually because there is a special mechanism involved for visualising the pointer indirection).

Strictly these would require a deviation to the rule, but it shouldn't be difficult to justify the deviation on the grounds of good readability as described above.

Rule 104

Function pointers should not be const qualified. Given a function f, we can refer to the address of f by &f. We would not expect to find arithmetic expressions involving the address of functions. Simple assignments of function addresses are permitted.

Rule 107

The example given in the guidelines text highlights the problem when the pointer is returned from a function. The example below shows the case when a pointer is passed to a function. Functions present a special case, which takes two forms. In the code example below, the called function de-referencing p cannot determine that p was set to a valid value by the calling function. (In this simple example, we can, but in the general case this would not be possible.) Alternatively, in the case of the called function returning a pointer, the calling function cannot be sure of the validity of the returned pointer. Within a function, the correctness of the pointer can be determined by how it is initialised and/or

set, or by a single validation test before use (assuming that the pointer is not volatile).

The text in the guidelines is an example of when de-referencing of pointers is required, and is not the only case.

It is recognised that this rule requires judgement, and therefore cannot be determined by static analysis. Therefore this rule should be implemented by another means such as a combination of software design rules, code inspections and/or software testing. Static analysis can be used to highlight occurrences of de-referencing, but care should be taken to minimise noise if these are highlighted.

It is recognised that this is a required rule. MISRA C is not limited to rules that can be implemented by static analysis alone.

```
/*
 * (c) Copyright 1999 Knowledge Software Ltd
 *
 * www.knosof.co.uk
 *
 * Permission is given to use and distribute this source
 * code provided this copyright notice is retained.  This
 * copyright notice must also be retained in any derived works.
 *
 *
 * KNOWLEDGE SOFTWARE DISCLAIMS ALL WARRANTIES WITH REGARD TO
 * THIS CODE.  IN NO EVENT WILL KNOWLEDGE SOFTWARE BE LIABLE FOR
 * ANY DAMAGES, LOST PROFITS, OR LOST SAVINGS.
 */

/*
 * MISRA107.c
 *
 *  1 Oct 99
 */

#include <stdlib.h>

#include "misra.h"

void g(MY_INT *p)
{
if (ei == 11)
   {
/*
 * No checking that p might be NULL
 */
   si=(*p); /* complain */
   }

if (p != NULL)
   {
   si=(*p); /* OK */
   }
}

void f(void)
{
MY_INT i = (MY_INT)*(MY_INT *)NULL; /* complain, also violates rule 45 */

g(NULL);
}
```

Rule 113

The text “It is recommended that structures are declared specifically to hold the sets of bit fields, and do not include any other data within the same structure.” is not normative, and is not considered part of this rule.

Rule 116

The authors are driving at the point that we should be in complete control of the code comprising the project for embedded systems work.

Rule 117

When passing values to library functions, beware of the behaviour of the function to out of range values. Either perform pre-checking or post-checking to protect for this condition.

Rule 118

Calloc, malloc, realloc and free are explicitly banned, and the use of these functions can be statically determined. The text highlights other means by which dynamic storage may be attempted and these other means are sufficiently general to prevent static detection.

Rule 121

Additionally localconv() should not be used.

Rule 127

The supporting text is an example of the supporting actions when a deviation is required.