**Coding rules for the safety domain**

I've talked to a couple of people on MISRA C about the rationale behind the pointer conversion rules, and they've convinced me that rather than looking at particular rules, if you want to design a better rule set, you need to start with:

- overall objectives for the rules,
- specific objectives for the area of the language being considered (in this case pointer conversion)

Then you can produce a set of rules that together aim to address those goals. This is how the MISRA C rules were created – its just unfortunate that the full rationale is buried on a WiKi somewhere.

I need to point out that this is my interpretation of the rationale, so anywhere it falls down, that's my bad, not MISRA's.

The primary requirement for safety code is that it should be unsurprising. Any averagely competent programmer should be able to look at a piece of code and understand what it is going to do, without detailed knowledge of the coding environment (and in particular, specific behaviour of the compiler not covered by the language standard). One unusual property of many safety systems is their longevity. Code may be under maintenance for 15 or more years, and it is quite likely that the programmers performing maintenance were not involved in its original development and may not have previous experience with the particular development environment. In practical terms this means the top level requirements are:

1. any code statement should only have one meaning
2. any code should conform to the KISS principle (not sure how you express this in standardese!)
3. any code should avoid constructs or behaviour that are known to have caused problems in the past

So for the 'one meaning' requirement, this means:

- no undefined behaviour, as by definition you can't know what the code does
- no unspecified behaviour that can lead to an observable change in the program's behaviour. So for example, because the order of evaluation of subexpressions is unspecified, any expression with multiple sub-expressions must exhibit the same behaviour whatever order the subexpressions are evaluated in (i.e. restrictions on side-effects)
- as far as possible remove the influence of implementation defined behaviours, for example using types like int32_t rather than a basic arithmetic type

An example of the KISS principle would be to limit the dependency on operator priority in complex arithmetic expressions. With 42 operators and 16 levels of priority, experimentation has shown that even experienced programmers cannot readily understand complex expressions without bracketing to enforce binding order. Interpretation of KISS includes:

- Rules are permitted to ban some well defined code, if that makes the rule easier to describe and police. However, the intent is not to ban common use constructs unless there is an equally easy replacement,
- Rules should have a minimum of exceptions – easier for both tools and users to handle. In the exceptional cases where what would normally be unacceptable behaviour is needed (such as casting an integer to a pointer to access a memory mapped device), the deviation mechanism allows a justification of the rule violation to be made, whilst still claiming

conformance (an important contractual issue, as frequently compliance is written into a development contract. The deviation process allows what would otherwise be a non-compliance to be justified in a legally defensible manner)
- Rules should be decidable where possible. This tends to mean that rules are more likely to use 'keyhole analysis' rather than 'whole program analysis'

'Known problems' tends to be a bit anecdotal, but flagging if (x = y) … because it may be a typo for if (x == y) … or questioning if (x) ; because it looks like a missing statement seems justified, as these are known to have caused errors in the past.

**For pointer conversion rules**

The undefined behaviours are:

- Conversion of a pointer to an integer type produces a value outside the range that can be represented (6.3.2.3)
- Conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3)
- A pointer is used to call a function whose type is not compatible with the pointed-to type (6.3.2.3)
- A function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function (6.5.2.2)
- A pointer is converted to other than an integer or pointer type (6.5.4)

The implementation defined behaviours are:

- The result of converting a pointer to an integer or vice versa (6.3.2.3)
    - required exception: 0 to pointer, for the NULL pointer

  Though arguably this should be unspecified behaviour rather than implementation defined, as I doubt the documentation for the compiler will tell me what value I'll get for:

```
int x = 0;      printf(¨%d\n¨, (long)(&x) ) ;
```

Known problems:

- Conversions that break the type model – basically any conversion of type *X to *Y (other than to-from void*)

So I'd argue for a set of rules for the safety domain:

- No conversion between a pointer and anything that is not a pointer or integer type (Undefined behaviour)
- No conversion between a pointer and an unrelated pointer, other than void*
- No conversion from void* to a pointer, with the exceptions
    - unless the void* pointer was created by converting a pointer of the same type (the standard guarantees this behaviour)
    - unless the void* is the result of a library function, such as malloc (*1)
- No conversion from pointer to integer (Implementation defined/Unspecified behaviour)
- No conversion from integer to pointer, with the exceptions
    - Conversion of 0 for the NULL pointer

(*1) WG23 recommends that malloc is always called via a macro, to ensure the type of the pointer and the size allocated agree, e.g.

```
#define allocateObject(T)     (T*)malloc(sizeof(T))
#define allocateArray(T, n)   (T*)malloc(sizeof(T) * n)
```

Note that this is against the MISRA philosophy, as it requires a specific coding solution, rather than defining what to avoid and letting the programmer decide how to comply