Image courtesy of digitalart at FreeDigitalPhotos.net

Chris Hills, CTO of Phaedrus Systems and a member of the MISRA C working group discusses the new MISRA C:2012 guidelines but also looks at why, on their own, they are not that useful.

The C programming language is incredibly popular and widely used on many projects ranging from the simple to complex, safety-critical systems. But the way C has evolved means that it is perfectly possible to write C that is legal, that compiles but is unpredictable in execution. This is clearly not acceptable for safety-critical systems, so in the 1990s automotive companies Rover and Ford separately developed their own sub-sets of C. Under the auspices of the MISRA Steering Group, these were merged to become Guidelines for the Use of the C Language in Vehicle Based Software or MISRA C:1998.

As these guidelines began to be used in real-life projects, it was clear that there were deficiencies and also that the guidelines were applicable way beyond the automotive market. The next version, MISRA C:2004, was called Guidelines for the Use of the C Language in Critical Systems. (Its friends sometimes call it C2.). These guidelines are now very widely used, either in themselves or as the basis of company specific guidelines.

When C2 was published it was specifically aimed at C90/95 (ISO-C 9899:1990 +A1, TC's 1, 2 & 3). (Although C99 had been published it was not widely implemented.) The working group on C (of which I am a member) continued to meet. It produced what we called a Technical Corrigendum and then began to look at the changes needed in the light of user experience and the greater availability of C99.

As the user experience began to feed back, we got some surprises, some good and some not so good. The good news was the wide take-up of C2, and the growth of MISRA C checking tools, normally as part of static code analysis. The bad news was that '100% MISRA C compliance' had become one of the boxes to tick in any audit process, demonstrating a complete lack of understanding of what MISRA C is about.

The give-away is in the word Guidelines in the title. MISRA does not set out a set of prescriptive rules: instead it provides a set of building blocks to create a coding standard for a specific set of circumstances. To allow this to happen, there has always been the option to deviate a rule: that is, effectively to ignore it, providing the project documentation records why that

rule has been deviated. With C2 it was legally possible to have a coding standard that was technically MISRA compliant but in which every rule had been deviated. (The deviation document would be large and make very interesting reading!)

Compilers were adding MISRA checking – which implied that MISRA compliance had not been checked by static code analysis before compilation. And there was other feed back that suggested that while we thought we knew what we were saying when we drew up a rule for C2, users found it ambiguous or, perhaps what was worse, did not understand why it existed.

A new approach

So for C3 (MISRA C:2012) we took a new approach. Firstly the headline rules were re-written, to be short and succinct. In some cases this meant splitting an old rule into two or more new rules. Then the rule has an optional amplification, a rationale and, again optionally, exceptions. With the amplification, or clarification, we can explain a short rule, rather than having a 5 line headline rule. The rationale gives the whys and wherefores of the rule followed, optionally by any exceptions. The exceptions again help to keep the headline rule short. Hopefully, this will ensure the whole rule, including the rationale is read. (In the past people seem to have read only the headline.)

With MISRA C:2012 it is no longer possible to deviate every rule. In addition to Required and Advisory rules there are now Mandatory rules, which must be adhered to claim any sort of MISRA C compliance. We started with about 30 Mandatory rules but after many discussions ended up with about 10 of them, as there are very few rules that will universally apply to all C developments. (This is an important point as it means that for projects using MISRA C, the vast majority, if not all of them, will have to deviate some rules.)

Recognising this, C3 now has an appendix on deviations, how to record them and how to link them to a compliance matrix. The compliance matrix lists all the rules and whether they are checked by the compiler, static analyser or manual review. (Yes, you still need a manual review, even after static analysis, and yes, you do need static analysis.)

One reason for going beyond just using static analysis is that there is a new factor for the rules: Decidability. A Decidable rule can be checked as a yes or no – has this rule been broken or obeyed for file or system scope. A rule may not be decidable at all. About 80% of the rules are Decidable, but this leaves 20% where a checker cannot say if the rule was broken. In addition there are also 16 directives. These are 'rules' where compliance cannot be determined from just the source code. For example: "All source files shall compile without any compilation errors." This, you might think, should be a given, but some compilers will produce an executable despite having generated 'error' messages as opposed to 'warnings'. Another directive is: "All code shall be traceable to documented requirements." Requirements? Did someone mention documentation and requirements?

And this neatly brings us to the real issue: MISRA C's value only appears when it is part of a disciplined and structured development process. Just saying 'use MISRA C' is not just pointless but can be counter-productive, damaging the morale of the development team and producing second rate code.

Specify first

People working to develop products that meet safety standards such as ISO 26262 or IEC 61508 have become used to a development process designed to make it possible to track a system requirement from its initial statement in a requirements specification to its implementation in the final system. But this level of knowledge should not be limited to safety-critical projects. What we are looking at here is using tools in a process to change developing software from a hand-craft on a par with knitting socks or making sandals out of old car tyres, to an engineering process that can, through monitoring and recording, be repeated and improved: a process that delivers a quality end-product, repeatedly and within time and budget.

In much of software development 'bugs' are regarded as a nuisance – something you swat through testing. But for years now it has been well-documented that the later in development you identify the bugs, the more resource you need to swat them and the higher the cost, particularly in delaying the delivery of the final product.

So how do you go about implementing a development process? And what do you need to do it? The process described here is the V method. This is often regarded as old fashioned by exponents of techniques such as agile programming, but it is at the heart of quality standards like ISO 26262.

You start by defining your requirements; what are you trying to achieve here. These requirements

Image courtesy of ddpavumba at FreeDigitalPhotos.net

definitions are often massive word processed documents that are stored away once this stage is passed. There are tools that not only help you capture these requirements, but make them available throughout the product life-cycle, allowing you to refine them in the light of later progress. The requirements can also include meeting a specific standard, such as 61508, and again there are tools that can help you identify which parts of the standard are relevant for your project.

Requirements need to be turned into a specification of a product that will satisfy those requirements. There are tools to do this and that also help you to determine that the code you are generating meets the specification. At last you can start coding and, at last, MISRA C becomes relevant.

As coding progresses the code should be run through static code analysis tools. These range in sophistication from fairly simple checkers to sophisticated analysers that identify a whole range of problems throughout the code base – and also check for MISRA C compliance. After static code analysis there is still a role for manual code review. If you like, once the static code analysis has determined whether or not the code is performing correctly, code review can concentrate on whether the code is actually doing what the specification wants it to do.

All compilers are not created equal, and the choice of compiler may also be influenced by whether the project is working to a safety standard or not. If it is, there are compilers validated to those standards.

For unit testing the code, there are tools that have specifically been developed for the safety standards and which are designed to automate the testing against your specification.

Producing the system requires middle-ware and there is a whole range of application dependent middle-ware products available. Things like file systems, software stacks for communications, data-bases and RTOSs can all be found that have been developed to high standards and some are even MISRA C compliant.

Finally there is system test and customer testing, all again capable of automation. In the classic V model, actions after coding feed back in to the earlier stages, both to produce revisions to meet problems found and to ensure that the system and the requirements/specification and the actual project remain in step.

All the way through this section I have kept emphasising tools. Traditionally software has been the poor relation of hardware in investing in tools, but with software becoming not just important but the main embodiment of the end product, management is beginning to understand that a relatively small investment in tools will give enormous returns in terms of speed and quality of product development.

And back to MISRA C. The code that is the heart of embedded products is not just important, it is often the product itself, with the hardware acting only as a delivery platform. MISRA C helps produce safe and reliable code. But it is only when a disciplined development process ensures that the code meets specification and requirements that it truly begins to play its full role.

More from Phaedrus Systems