The second edition of the MISRA C guidelines

Les Hatton,
Professor of Forensic Software Engineering,
CIS, University of Kingston

Nine years ago, I finally finished a book entitled “Safer C”, (Hatton 1995). The theme of this book was *not* to promote the use of C in high-integrity and safety critical systems, but simply to make the point that uninhibited coding in this language was entirely inappropriate for such systems. However, *if measures were in place to prevent the re-occurrence of very well known fault modes in this language, all the evidence suggested that C was just as capable of producing high reliability systems as any of the other programming languages*. I defended this viewpoint with measurements on a large number of released systems and of course, we should never forget that in spite of a software process which would be deemed chaotic (level 1) by the Capability Maturity Model, the open source community has voluntarily produced one of the most reliable complex applications in history in this language, the Linux kernel. For example, on my office systems, we have now reached 15 server years without any failures on three different kernel releases, a level my Windows systems cannot even approach. We must also of course not forget that at the other end of the scale, some truly lamentable systems have been produced in the same language.

Similar extremes exist in other languages. The very best (0 statically detectable faults per KXLOC (thousand executable lines of source code)) and the very worst (140 faults per KXLOC) I have ever seen came from the same application area in the same programming language, Fortran 77, a language refreshingly free of the dangerously uncharted wastelands of dynamic memory and object irritation which modern computer linguists like to add to languages. Not so refreshingly, the application area was a nuclear reactor simulation program programmed by two unconnected sets of engineers, the one highly knowledgable and the other entirely clueless about such dangers. (These latter engineers told me that this 70,000 line program “had no bugs because they had tested it”. Truly hope beats eternal in the human breast. In essence, they had produced an extremely expensive random number generator.)

As a forensic software engineer, I find these observations really interesting and with these and other experiences, I have become more and more convinced that we do not have a technology problem at all, we have an educational problem, writ very, very large. Indeed, looking at known sources of data for different programming languages, humans tend to screw up at similar rates in all of them. The common denominator in poor systems seems to be that the engineers aren't very good and no amount of process or indeed technological refinement can currently recover from this. If the reader is a little surprised that languages can be internationally standardised with potential fault modes embedded in them, that is simply the way of things. Human fallibility and politics can undermine any efforts to make things truly precise and one of the central roles of forensic engineering is learning how to live with it.

One of the most important educational steps in improving an engineering technology is the recognition of previous failures and the implementation of

avoidance procedures to bypass them in future. In this regard, the older languages like Fortran 77, Ada 83 and C 90 are well-charted territory. There is a very considerable knowledge base of how defects are injected in these languages and the knowledge has been exploited, principally in software tools but equally importantly by public dissemination, to help engineers avoid them. Such *safer subsetting* is an important contribution to more reliable systems. Let us take C for example. The experiments which led to "Safer C" suggested that a residual fault density of around 8 per KXLOC was actually released and later experiments, for example (Pfleeger and Hatton 1997), suggested that at least 3 of these would fail in the life-cycle of a typical program. Now, really good systems typically stay below 1 fault that fails per KXLOC through their entire life-cycle, so we can see straight away that a safer subset is necessary (although certainly not sufficient) to achieve such levels in C. A very similar pattern occurs in Fortran.

So where does MISRA C fit into all of this ? The original MISRA C appeared in 1998, (MISRA 1998) with the title "Guidelines for the use of the C language in vehicle based software". The body responsible was a group of collaborating companies from the automotive industry, (MISRA is the Motor Industry Software Reliability Association), and the spur was the rapidly increasing use of C in embedded systems in general and in automotive systems in particular. MISRA C 1998, consisted of 93 "required rules" and 34 "advisory rules", a total of 127 rules which together represented a significant step forward in encouraging engineers to understand the issues and avoid known defects in the language. (To put this in context for readers not familiar with the nature of such rules, one them (Rule 30) states that uninitialised objects shall not be used, a fault mode afflicting many programming languages). The choice of the phrase "advisory rule" is somewhat oxymoronic but the intent was that "required rules" were mandatory unless a very convincing reason to the contrary was given and "advisory rules" were recommendations only.

MISRA C 1998 had a number of drawbacks unfortunately. The rules themselves are noisy in the sense described by (Hatton 2004) leading to a false positive to real positive ratio of around 60 making it rather difficult to see the wood for the trees. Surprisingly, this is good for a programming standard, most of which concentrate on style rather than substance, but there is certainly room for improvement. Part of this noise is due to rules which in the harsh glow of hindsight, could have been more precisely worded, and part is due to the nature of the rules themselves, some of which are not measurement supported and difficult to police, for example, "code shall not be commented out", Rule 10. In a number of cases, it was not easily possible to determine what the rule actually meant. This led to widespread adoption of *deviation policies*, an officially sanctioned way of ignoring some of the rules, the documentation of which, for a safety-related system, would necessarily form part of the safety case. Deviating *all* of the rules would be frowned upon of course. Another unfortunate side-effect was that "MISRA compliance" was rendered meaningless as different tools claiming to enforce it could and did give radically different answers when given the same code, (Parker 2001).

Nevertheless, this was a commendable effort much better than doing nothing and since 1998, the original MISRA C standard has become very influential and is increasingly widely used in the avionic and medical sectors for example, as well as the original automotive industry. To recognise this growing ubiquity and to address some of the drawbacks highlighted above, the original MISRA committee,

started preparing a second version some three years ago and this version appeared finally after a great deal of hard work in October 2004, (MISRA 2004). Its title "Guidelines for the use of C language in critical systems" reflected this growing ubiquity. My first reaction as an avid watcher of standards bloat was that it is somewhat but not excessively bigger (109 pages compared with 69 pages). MISRA C 2004 now contains 141 rules, caused by removing some of the old rules and adding a larger number of new ones. The rules have been re-numbered and in many cases rewritten and it is hoped that this will address at least in part the known problem areas alluded to above although this has yet to be tested. The committee has not at the time of writing formally defined a succession policy and currently at least, an organisation can offer either version in support of its attempts to adhere to safer subsets. This is probably a wise move as many organisations have invested a significant amount of resources complying with MISRA C 1998 and may want to reflect and await some measurement support before committing to MISRA C 2004.

The MISRA C standard continues to move forward. In the short to medium term, a set of exemplary test cases is being produced by the committee to help vendors and customers understand the scope and enforceability of the new version. Looking further down the line, the committee may have to grasp the nettle that is ISO C99, the latest standardised version of the C language, and like so many ISO languages, standardised to the point of extinction, (although I hasten to add, this is an entirely personal opinion.)

MISRA C 2004 is the latest in a significant effort to produce effective codes of practice for engineers working with C in embedded control systems in which failure is usually very expensive and this continuing initiative and focus should be applauded. You might like to reflect on that as you drive home tonight half a metre from a software controlled bomb embedded in your steering wheel in a car which has very significant amounts of software in the brakes, steering, engine, and everywhere else. As society moves rapidly towards complete dependence on embedded control systems containing large amounts of software, such initiatives will become increasingly important. I wish there were more.

Les Hatton is Professor of Forensic Software Engineering at the University of Kingston and is a director of Oakwood Computing Associates Ltd. His various writings can be downloaded from http://www.leshatton.org/ . He can be contacted at l.hatton@kingston.ac.uk or lesh@leshatton.org.

Hatton L., (1995), "Safer C: developing in high-integrity and safety-critical systems", McGraw-Hill, ISBN 0-07-707640-0
Hatton L., (2004), "Safer language subsets: an overview and a case history: MISRA C", Information and Software Technology, 46, p465-472.
MISRA (1998) "Guidelines for the use of the C language in vehicle based systems", MIRA Ltd., http://www.misra.org.uk/ , ISBN 0-9524156-9-0
MISRA (2004) "Guidelines for the use of the C language in critical systems", MIRA Ltd., http://www.misra.org.uk/ , ISBN 0-9524156-2-3
Parker S and JJ (name unknown) (2001) "A comparison of MISRA C testing tools", http://www.pitechnology.com/downloads/files/MISRA_C_tools.pdf.
Pfleeger S., and Hatton L. (1997) "Do formal methods really work ?", IEEE Computer, January 1997.