MISRA C:2012

Part 2:

Implementing

MISRA-C:2012

June 2014

by
Eur Ing ***Chris Hills*** *BSc (Hons),*
C. Eng., MIET, MBCS, FRGS, FRSA

PhaedruS SystemS

The Art in Embedded Systems
comes through Engineering discipline.

Implementing MISRA-C:2012

- MISRA-C:2012, Why it won't save your Project
  - DDC 2013 MISRA Workshop
    - Why MISRA-C
      - On its own
      - When 100% enforced with no deviations
      - Used without static analysis
    - Will NOT save your project
  - Part 1 is Required Reading
    - Available for download
      - in exchange your email address

PhaedruS SystemS

Safely From Conception to Completion
www.phaedsys.com

1 of 143

At the Device Developer Conference 2013 I presented MISRA-C:2012, Why it won't save your project illustrating why, for so many projects, MISRA-C does not help but hinders - usually because it is badly implemented. It is worth reading this paper first; it can be downloaded from http://library. phaedsys. com or from librarian@ phaedsys. com. For the Device Developers Conference 2014 this paper was prepared as "part 2", following on and giving more detail on how to correctly implement MISRA-C in general and MISRA-C:2012 in particular.

Implementing MISRA-C:2012

MISRA-C DISCLAIMER

Adherence to the requirements of this document does not in itself ensure error-free robust software or guarantee portability and reuse.

This disclaimer is in MISRA-C:2004 and, less prominently, in MISRA-C:2012. However it should be printed as a poster on the office wall of the development team. Without care, thought, discipline and careful implementation, nothing is automatic and easy. Even the easy and automatic things need to be thought about and understood before being carefully implemented and properly used.

For all but a trivial program is virtually impossible to prove it is a "zero defect" system. Most embedded systems are far from trivial, so the best you can do is demonstrate that you have minimized the chance of a defect. No one thing can do this and certainly not MISRA-C on its own.

There are no easy answers other than doing it properly. With engineering discipline and a good process things do get easier as effort is applied appropriately and less effort is wasted. As it says on the front of every Phaedrus Systems technical document: ***The Art in Embedded Systems comes through Engineering discipline.***

Implementing MISRA-C:2012

might not save your project…

MISRA-C:2012 is NOT a silver bullet. It is not a magic answer. Nor were MISRA-C:1998 and MISRA-C:2004. The fact that there have been three versions and the MISRA-C team is looking at a possible MISRA-C:202X shows that this continues to be an evolving work. This is partly due to the ISO C standard changing along with the C cross-compilers developing to track it, partly due to a Japanese team translating MISRA-C:2102 into a completely different language system and partly due to static code analysis companies striving for precise definitions. And then there are thousands of users stress-testing MISRA in many vastly different applications, running on systems from 8- to 128-bit.

In fact there are no magic answers unless you live in fairyland or bring your fantasy role-playing games to work. And no, your current project is not a fantasy-role playing game – despite the similarity in places to a Dilbert cartoon!

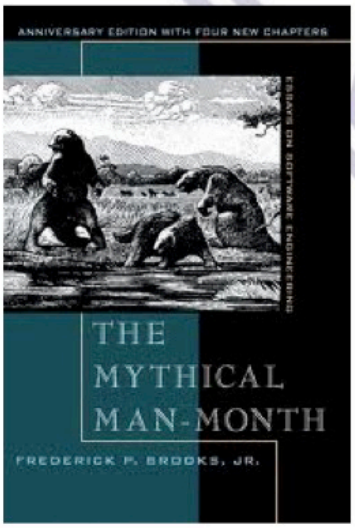
There are far too many who see various tools or methods as The Answer. With all tools and methods it is how the tools, methods and processes are used and how they are used in relation to other tools and the process in general that is important. (For more see Brooks, *The Mythical Man Month* - on the next page.)

There is no one thing that will guarantee error-free, robust code or indeed a robust or error-free system. Embedded software is part of a system that does something physical in the real world. As with most things you have to look at the overall system, which should be greater than the sum of its parts. Requirements, process, tools, integration of tools, specifications, formal design and code reviews etc will all contribute to minimising the occurrence of bugs. And they should make the discovery and rectification of those bugs that do occur much easier and faster. The next few pages look at the elements of a robust development process, that you need in place if you are to get any benefit from implementing MISRA-C

Implementing MISRA-C:2012

Might save your project…

- The Mythical Man Month
  - There is no Silver Bullet
  - 20th anniversary addition
    - Revises data and assumptions
- Project Management
  - Styles change: people don't



PhaedruS SystemS

Safely From Conception to Completion
www.phaedsys.com

The Mythical Man Month is a seminal book on project management. It says that if it takes one man nine months to do something - it does NOT mean that nine men can do it in one month: adding people to a project can even extend the time it takes. As more people are added you need to communicate with them and bring them up to speed. Most importantly you need to ensure that they mean the same things you do when they say something: new people need to learn the local "project language".

Life is more complex than simply dividing people into months but surprisingly it is not that much more complex. In other engineering disciplines most of the rules for team work and project management have been well understood for decades - if not longer. Sadly software engineering degree courses rarely teach project management and finance, and programming is normally taught within computer science, not as an engineering discipline.

Usually it is when a project is running late that more manpower is added. This is far too late: the damage has been done and the additional people are merely fire fighting. At the same time people within the project are trying to ensure that they are not taking the blame, sometimes by trying to make sure that their error(s) look smaller than other people's. Everyone tries to cut corners look after their area and to hell with the rest of them. Adding more people just makes the situation worse.

The answer is to put resources in early so you don't have a fire. To do this means that you need to get the requirements right. Then you will know, accurately, what it is you are building and you will be in a far better position to estimate the resources required for the project.

Mythical Man Month ISBN-13: 978-0201835953

Brooks' web site: http://www. cs. unc. edu/~brooks/

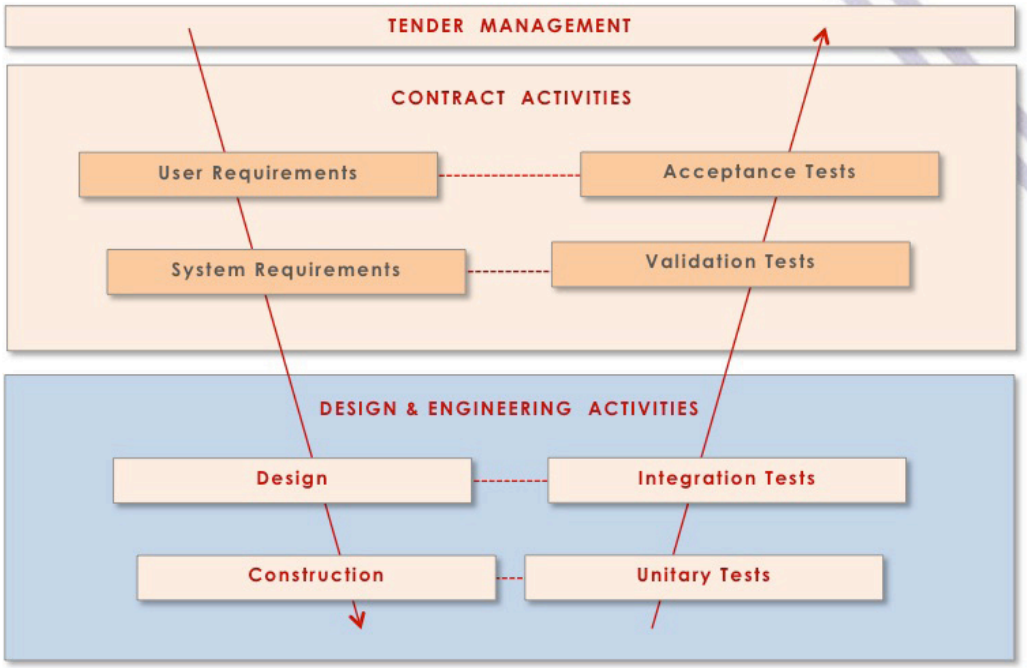
http://en. wikipedia. org/wiki/The_Mythical_Man-Month

http://javatroopers. com/Mythical_Man_Month. html

Chapter 2 http://www. cs. virginia. edu/~evans/greatworks/mythical. pdf

1 hour presentation on MMM and project management in the software domain: http://www. frequency. com/video/frederick-brooks-mythical-man-month/109797838/-/5-9872894

Implementing MISRA-C:2012



The classic V process development model for software and systems works - if used correctly. (That caveat also applies to all processes.) There are many safety-critical systems running today that are saving lives or stopping lives being lost that were developed using the V model. There are many more non-critical systems that just quietly get on with their work that were also developed using the V model.

There are other, equally valid, process models* and the following notes should be applicable to them as well, it is just easier to explain the problems using a V model.

The V model is conceptual and shows information flow through a project. The User Requirements at top left - the start - also provide the Acceptance Tests at the top right - the end. Both of these should be completed before a single line of code is written.

The problem areas in this model (or any model) lie in the interfaces. In this model the gap between Tender Management and the Requirements, the input to the V, is the stage that should convert a fluffy wish list into requirements. Sadly, it more often just passes the fluffy wish list to the designers. They, in turn, produce a design that is either a bit fluffy or uses guesses to fill in the blanks and inconsistencies.

The next interface, the gap between the pink and blue boxes, is the most crucial. The output from the pink requirements phase is usually a paper exercise, involving only the cost of a few expense account lunches or buffets for meetings when talking to the customers. Maybe there are even some visits to the customer.

When you enter the blue section of design and construction you now start to use real time, real effort and in many cases incur real, non-recoverable costs: mistakes can now be measured in money. As well as software, embedded systems also involve a hardware team actually making physical things that cost money. It is far cheaper to double the time in the requirements phase than create an illusion of progress by writing code and making hardware without complete requirements. I have seen 6 months work and a pre-production run of PCBs scrapped due to leaving some decisions to "later".

There is the so called "Spin cycle" in the requirements and specification phase, where proof of concept and other ideas can be run round. Here prototypes are made, algorithms tested, techniques tried out and theories proved. This can be thought of as the Research in R&D. However NONE of this hardware or software should be used in the main development process, other than third-party and other libraries that have already been fully tested and validated. Research is not Development but should inform it.

***Note:** While there may be areas where the Agile approach is valid, the development of complex, particularly safety-critical and high integrity systems, is not one of them.

Implementing MISRA-C:2012

	12 Months	24 Months	36 Months	48 Months
Reusable requirements	10	40	150	300
Reusable Design	10	40	250	500
Reusable source code	15	50	250	600
Formal Design Inspections	**350**	**600**	**1000**	**1500**
Formal Code Inspections	**250**	**600**	**1200**	**1500**
Informal Reviews	**150**	250	300	400
Improved Staff Training	90	200	500	750
Formal Standards	100	115	175	300
Productivity Measurements	150	**450**	**600**	**1000**
Functional Metrics	175	300	450	800

Programming Research has developed this table of Return on Investment (RoI) per 100 (USD/GBP/Yen/Euro) invested in a project. The red numbers highlight the best return over time, and the blue numbers are the second best return. This chart shows that formal design inspections produce the best RoI, followed by formal code inspections. These two score the highest and second highest ROI in all categories, more than all the rest put together. BUT formal code inspections have to assume that the design is right!

Design inspections pay off faster because if you get the design wrong you are wasting time and effort (money) on building the wrong thing in the next stages, with the strong possibility that you may have to scrap both hardware and software. With code inspections the return is higher the further you get from coding: the costs of fixing a coding bug escalate dramatically through the product life-cycle. A bug that would cost 1 (USD/GBP/YEN etc) to fix if found through static code analysis during the coding phase could cost 50,000 (USD/GBP/YEN etc) -or even more - if it escaped into the field.

I have a real world case where that happened. The company in question had turned down an "expensive" static code analysis tool solution costing 20K during the development phase of a multi million GBP project. When the system had been deployed somewhere on the far side of the world, a serious software bug appeared due to some unusual, but legal, use of the system. The company had to fly out an engineer for two weeks to track down the problem. The cost was over 50K before the loss of reputation and possible future sales. The company asked for another demo of the static code analysis tool to see if it would have found the bug. The tool found the "50K bug" in about 15 minutes. The tool also uncovered another 5 problems of similar magnitude that were in the code out in the field waiting for the dice to fall, when they would have caused a real problem affecting machinery and people. The tool found several hundred other minor problems.

MISRA-C:2012

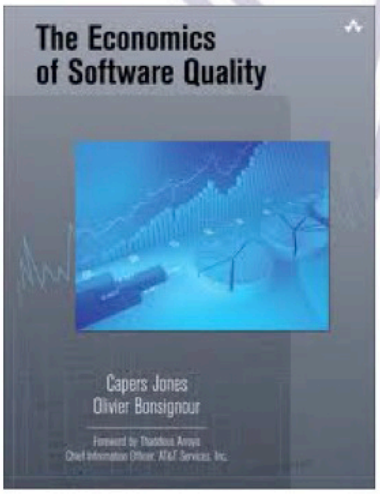
Might save your project…

- SW Quality costs nothing!

High quality schedule = 12.4 months
Avg quality schedule = 13.8 months
Poor quality schedule = 15.6 months

High quality costs = $ 846,636
Avg quality costs = $ 920,256
Poor quality costs = $1,039,889

Serious bugs (high quality) = 22
Serious bugs (avg quality) = 68
Serious bugs (poor quality) = 142



PhaedruS SystemS

Safely From Conception to Completion
www.phaedsys.com

X of 143

That a well defined and robust process saves money is not just theory - it is reality. A lot of it is based on the real world findings of Capers Jones who has been involved in a lot of litigation over software as an expert witness. This, coupled with his research, has contributed to his book, The Economics of Software Quality. Here he has the figures and case histories to confirm that it actually costs less to produce high quality code rather than try and do it on the cheap by cutting corners or making savings.

The table shows three analyses for schedules and costs: high quality, average, and poor quality. All three are 1000 function points in size. Costs are based on $10,000 per month.

The high quality case used static code analysis, inspections, and formal testing.

The average quality case used static code analysis and quasi formal testing.

The poor quality case used only informal testing.

See this Short Video by Capers Jones: http://www. youtube. com/watch?v=zmrqsQxv_yo
Also worth listening to is a Podcast: *Economics of Software Quality - An Interview with Capers Jones*. the Interviewer, Rex Black, is also a well known safety systems expert in his own right)
Part 1: http://www. youtube. com/watch?v=zo8JI9MVxQg
Part 2: http://www. youtube. com/watch?v=FLDgRtzq-Cc
http://sqgne. org/presentations/2011-12/Jones-Sep-2011. pdf

Verification and Validation: that well known double act. Everyone goes on about validation, testing etc., static code analysis, dynamic analysis, unit test etc., and how they can save a lot of time and effort. In fact they are all essential but cannot be used just on their own.

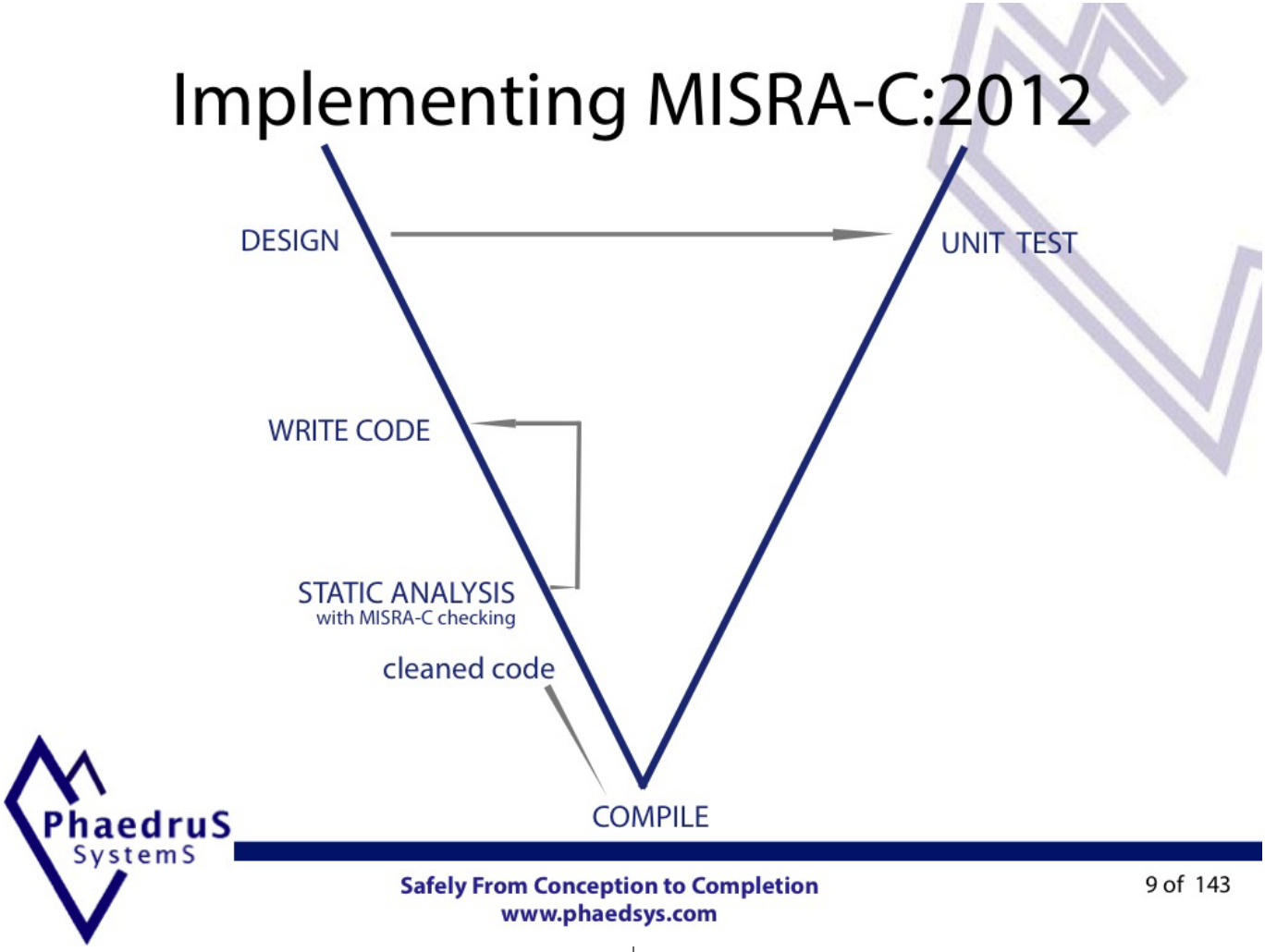
Let's take a quick look. First note that there is a difference between correct code and code that implements the correct functionality. Static code analysis on its own can remove many problems and misuses of the language, BUT it can not prove that the code is functionally correct - that it is doing what the designer wants. Unit test can prove the low level design but it will not find many/any bugs in the code and again nor whether it satisfies the overall system requirements. So you need both static and dynamic analysis in that order.

No matter how good or validated the test tools are, unless you have a solid requirements specification and a reviewed design that relates to the requirements, you don't really know what you are validating. The code may be correct in itself and "work" but it may not be doing what the end user wants.

Verification: Are the requirements correct?

Validation: Static - is the code correct?

Validation: Dynamic - does the unit/system function to the requirements?



Looking at the lower half of the V model in more detail.

The design specifications should have produced the unit test cases which go to the unit test phase. This means that you should have the test cases - unit, integration and system - before you write the source code.

The next step is to write code to implement the design specifications and following the coding standard (which, since we are talking about MISRA-C, is based on the MISRA-C guidelines). Then run the static code analysis and the MISRA-C checker. There is no point in running MISRA-C checking unless you also run static code analysis. As MISRA-C is a subset of the C language, checking for these rules is only a small part of static code analysis which typically finds 100's of problems. Today most static code analysers (if not all of them) also provide MISRA-C checking. Correcting the errors discovered and rechecking will eventually give you clean code.

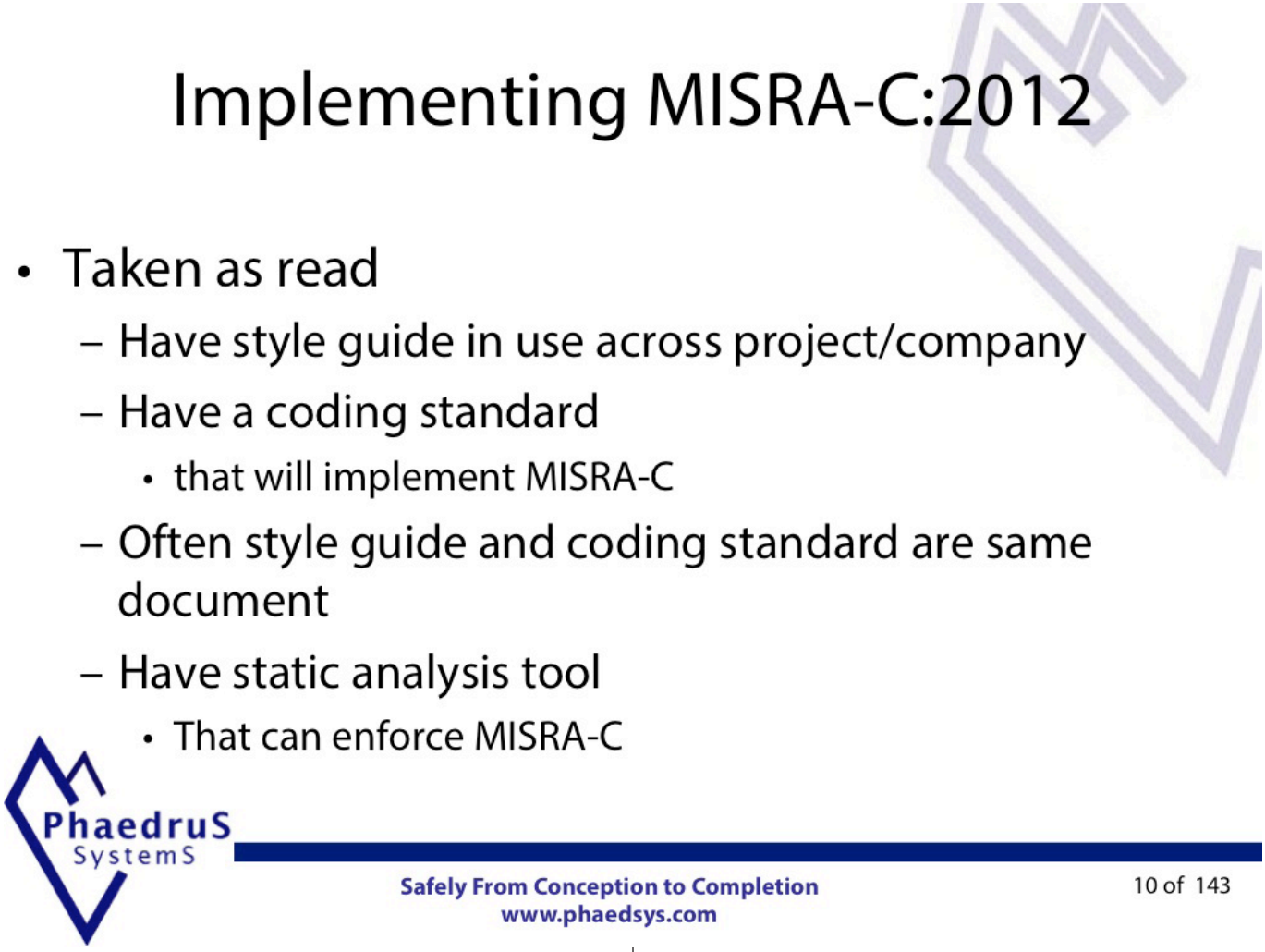
Now compile the code with the compiler set to its highest level of warning. While a properly configured static code analyser should have picked up all the problems it is better to be safe than sorry.

Even though there should not be any problems by this stage, you must resolve ALL compiler errors and warnings.

As I was writing these notes there was a discussion on the MISRA-C & C++ forum on LinkedIn as to whether code should be compiled before or after static code analysis. Opinion was divided: those who know how compilers and static code analysers work pointed out that the static code analyser is an analyser while the compiler is a translator. Whilst many compilers and static code analysers share the same parser (see the customer list at http://www. edg. com/) - this is only the front end parser. After that compilers and static code analysers differ in what they do and how they do it. A static code analyser will pick up incorrect syntax as easily as the compiler but it does a lot more besides. So before fixing the code you want the full picture.

It helps if the static code analysis tool is integrated into the programmer's IDE, then it can be called as frequently and as easily as the compiler. Also it should be configured to analyse either the current file or a group of files.

NOTE: If you unit test before static code analysis you will prove nothing. When you statically test, the changes you make in the code after you have found the bugs will render all the unit tests invalid. So it is a complete waste of time.



In order to implement MISRA-C you will need a company coding standard. It should contain both a style guide (for layout) and local coding standards, as well as the MISRA-C rules. This coding standard will include things like:

The file and function information blocks.

Naming conventions.

Where the {} are placed.

How many spaces in a tab (Tabs should be converted to spaces and most editors will do this automatically).

Ideally there should be one coding standard per language across the company. (Desktop & PC applications will need a different coding standard to that used for embedded systems – but today C is rarely used on desktop applications.)

If you don't have a style guide there are many on the internet. To be honest, it does not matter which one you pick as a basis just as long as there is consistency across the whole project, if not the whole company.

Note: this standard is for your own code. Third party libraries that are bought in will have their own coding standard. However code supplied by people and companies working for your company should adhere to your coding standard. I once came across a case where a contractor had his own style and refused to budge. The answer should be, "Use a different contractor." In reality, for many contractors, if it is a case of "Conform or not get paid" they usually conform.

The other point that is assumed is that you have a static code analyser that enforces MISRA-C. The original MISRA-C was strongly based on the work of a static code tool company. That company, along with another, has been part of the MISRA-C working group since the start. For this reason the MISRA-C guides have stopped short of requiring static code analysis, lest anyone claim that there is a commercial motive. However, for the C language, all the studies over the last 38 years have shown that static code analysis is very cost effective and is the most effective tool at removing non-functional problems. It is certainly the fastest way of uncovering and highlighting problems.

MISRA-C is a language subset that is well suited to being enforced as part of static code analysis: anyone trying to enforce MISRA-C without a static code analyser is really missing the majority of the benefits of using MISRA-C.

Implementing MISRA-C:2012

"To encourage people to pay more attention to the official language rules,

to detect legal but suspicious constructs,

and to help find interface mismatches undetectable with simple mechanisms for separate compilation, Steve Johnson adapted his Pcc compiler to produce lint."

Dennis Ritche.

ACM journal 1993

B 9/9/41 D 9/10/11

When the first static code analyser for C (lint)* was made it was to detect legal but suspicious constructs. **A LOT of LEGAL C is DANGEROUS** according to Denis Ritchie, writing in 1993 about the first lint program that was constructed in 1976. So, even before the first language reference for C (K&R) in 1978, and over a decade before ISO C, there were problems with C being misused. Even then the compiler told you very little about the quality of the code.

Also programmers like to try and prove how clever they are with C. Brian Kernighan said, "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." This comment by Kernighan suggests that your cleverest and smartest programmers should be carrying out test and debug: that is an interesting proposal to put to a development team!

Using a lint program, or what we would now call static code analysis, was intended to be part of the standard C compiler chain from the beginning and it certainly was on UNIX. But for some reason it never survived the leap to the PC development platforms. Many of us, with a UNIX background, did use lint in the 80's but most developers never started the habit and it seems universities never pushed it. The culture of "it compiles - it must be OK" started to prevail.

The original lint static code analysers have developed into, at the high end, very powerful code analysers that can enforce local coding standards as well as rigorously analyse code with configurations for many dialects of C. Even at the entry level the static code analysers are more advanced than the compilers for code analysis. Which is not the same as code translation the primary, but not the only, purpose of a compiler. (In the embedded world most compilers have extensions for the hardware architecture, specific IO and registers.) Check the pedigree of any static code analyser you intend to use. Some of the free tools such as splint have not been maintained in years or have support for cross-compilers and have not really been fully tested.

Many studies show static code analysis works and SAVES TIME AND MONEY. Most static code analysis tools pay for themselves the first time they are run by finding simple bugs that, if they escaped into the wild, could cost several times the cost of the static code analysers. (And that is true not just for safety-related projects.)

*The father of static code analysis: http://en. wikipedia. org/wiki/Stephen_C. _Johnson

Implementing MISRA-C:2012

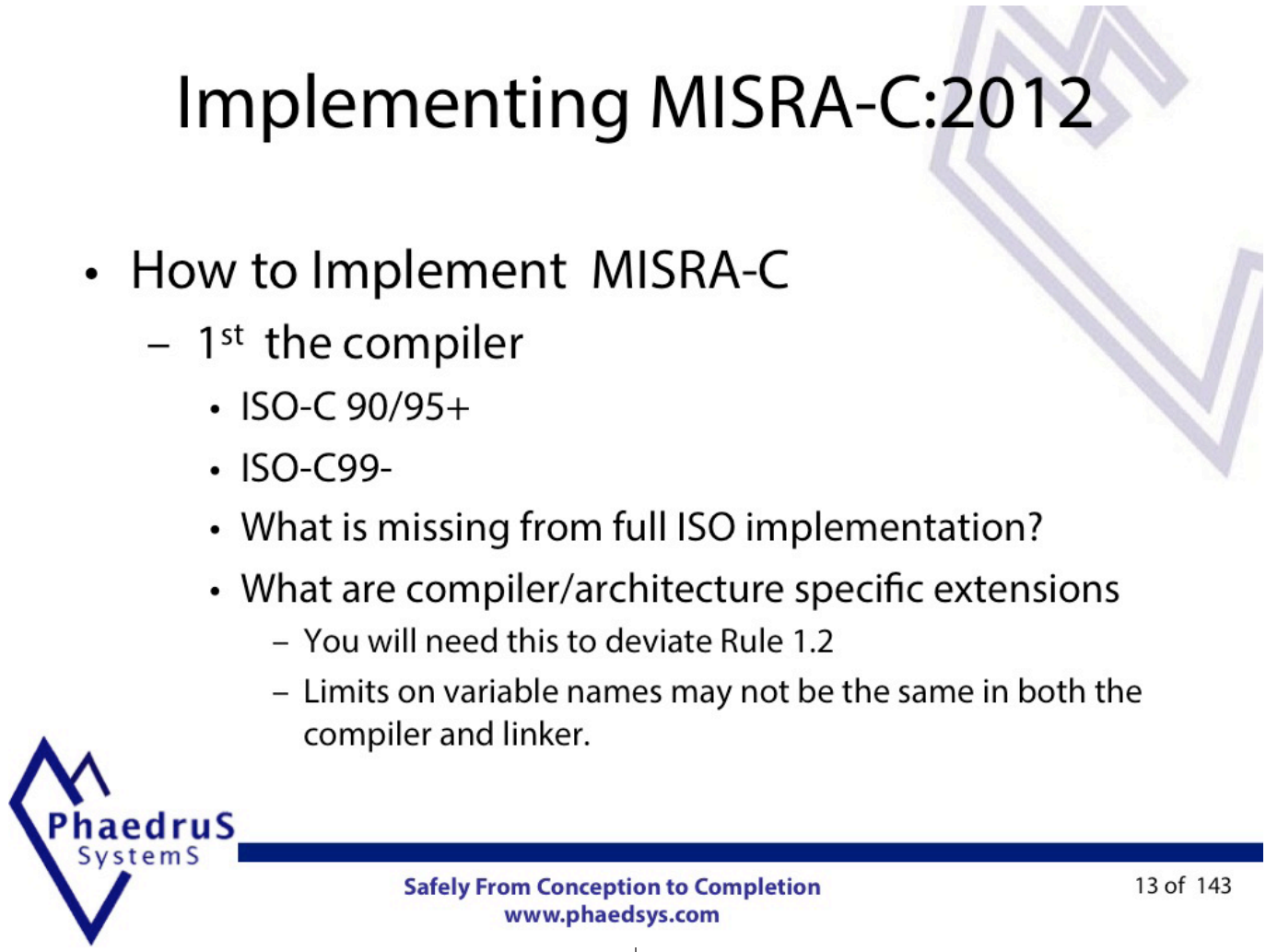
- MISRA-C does NOT find bugs
  - MISRA-C restricts the language to remove dangerous and miss-used constructs.
  - Code that whilst syntactically correct
    - May not do what was expected
  - Multiple un chucked thongs can cause problems that are herd to sea or fine.

It is important to realise that MISRA-C does not find bugs as such. In C there are legal constructs that are commonly misunderstood and misused by programmers, producing code that, while appearing to function, does not do what the programmer expects it to do. The code may appear to work in a narrow range of cases but often, when combined with other, similar, constructs causes problem symptoms somewhere else. These symptoms may appear to be un-related in the code (and in time).

There are many studies that show 30-40% of project time is spent on avoidable rework and bug hunting. The point of MISRA-C is to restrict the use of these constructs so that the code will do exactly what is expected, in the way expected, with no unwanted side effects.

The C language has virtually doubled in size on each iteration (K&R 1, C90, C99, C11) so there are very few, not even the language lawyers, who have a solid understanding of the whole language. Also the "undefined and implementation defined" aspects that are described in Annex G of ISO 9899 (C99), have grown in a similar fashion, giving compilers, and cross compilers in particular, a lot of latitude.

MISRA-C seeks to clarify and minimise these variables, especially as the vast majority of programmers have never seen, let alone read, the ISO C standard. NOTE K&R 2 is 25 years and several generations out of date. It is a nice historical book but not a modern C language reference.



So how do we implement MISRA-C? First read the paper "MISRA-C: Why it won't save your project" and make sure you have a good process, solid project requirements,* a good design process incorporating a formal design review that validates the requirements. Without this in place there is no real point in continuing.

Now look at your tools: starting with the compiler

You need to know which version of C the compiler thinks it is implementing. "ANSI-C" is not an answer, come to that neither is "ISO C". You need to know precisely which version of ISO C and, as importantly, where your compiler differs from ISO C. Fortunately this should be in the compiler manuals, so READ THEM. You will need to quantify what the compiler does with the "implementation defined" things and for cross compilers the extensions of the language for the target architecture. This will all need to be documented in the MISRA deviation document as you WILL be deviating MISRA Rule 1. 2. You cannot implement MISRA-C:2012 without deviating at least this rule. There is more on deviation later.

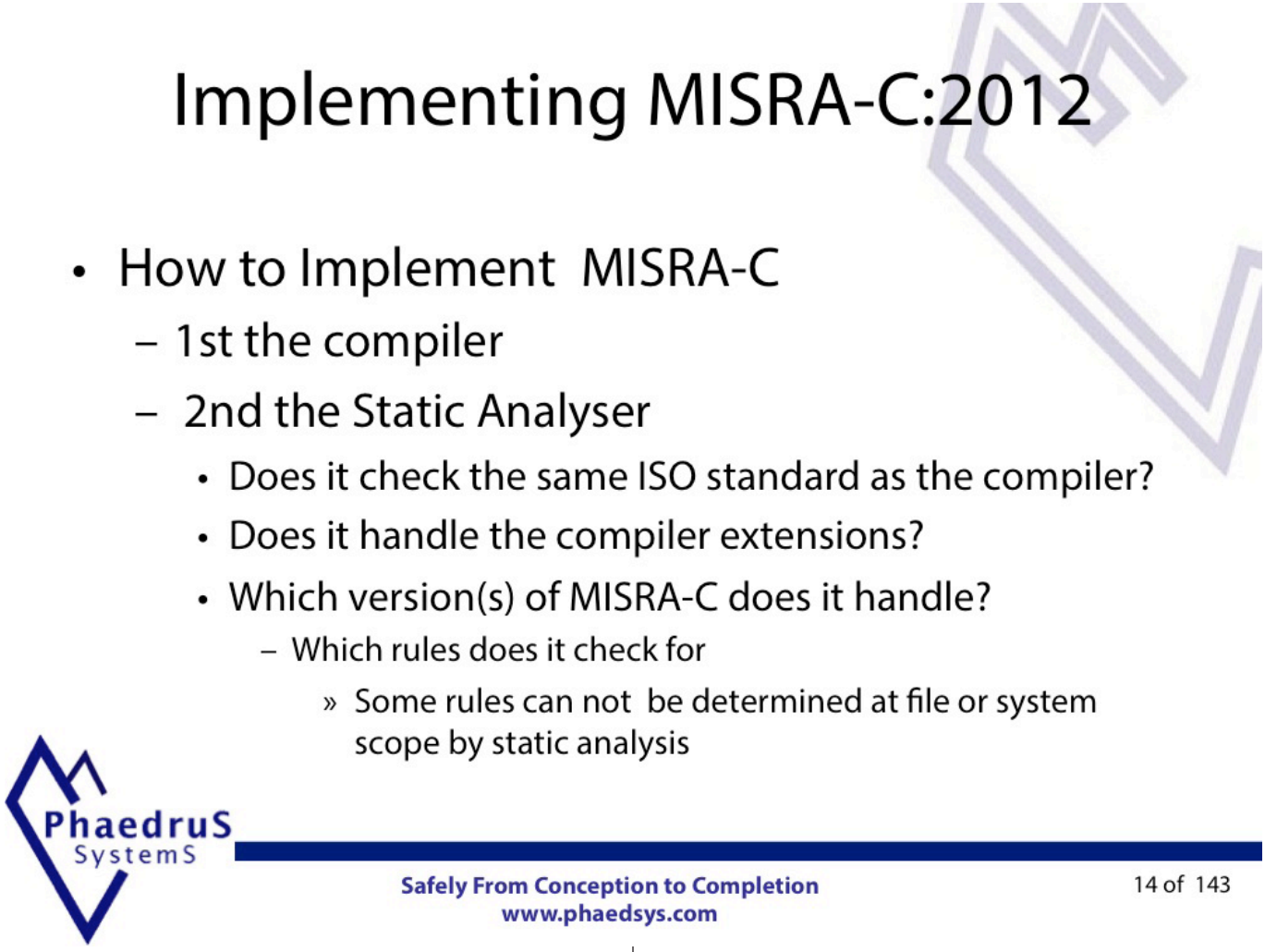
As a salutary note, some years ago on a project before MISRA-C we had done a loose version of this in our process. We noted that the compiler had a limit, requiring variables to be distinct in the first 32 characters. We neglected to note that the limit on the linker was 31 characters. This made a difference on one pair of variables A and B. This did not appear until late in the testing, when the symptoms had us chasing a hardware fault.

So you really do need to check the specification of the compiler changes in some detail. A day spent doing this will save many days of chasing your tail later.

Finally SET THE COMPILER WARNING LEVEL TO MAXIMUM and investigate/remove ALL Warnings in compiled code. No matter how theoretically correct the code is to ISO C and MISRA-C, it is the compiler that is producing the binary. If there is a warning it must be investigated. I am sure this point does not even need mentioning for compiler errors.

Since a compiler does not do the same job as the static code analyser, we need to look at that next.

*See the paper "Requirements are Required" in the library (http://library. phaedsys. com)



Having determined what the compiler is doing you need to look at the static code analyser. NOTE: there is no point in implementing MISRA-C unless MISRA-C conformance testing is part of the static code analysis phase.

Firstly you need to know the version of ISO C your compiler is working to and ensure that your static code analyser supports the same one.

Secondly, and this is almost more important, can the static code analyser handle the language extensions and non-standard keywords the compiler uses? Some static code analyser tools such as PC-lint have many (over 80) standard configuration files in the tool. Others, like Programming Research's QA•C, take a different approach and have a compiler personality generator tool that will generate a configuration file for your compiler. Without doing this your static code analyser will throw up hundreds of false positives. It is impossible to stress too highly the importance of correctly configuring the static code analyser for the compiler.

Once you have a static code analyser that is configured for the compiler you need to ensure that it will test for the same version of MISRA-C that you will be using in the code. It is no use writing code to MISRA-C:2012 if the analyser is checking against MISRA-C:2004.

(When you want to look at moving code from an older MISRA-C to MISRA-C:2012 there is a case to be made for running the code through a checker running to MISRA-C:2012, however.)

Just as you had to look at the compiler extensions and implementation defined things, you need to look at which MISRA-C rules the static code analyser enforces: not all of MISRA-C is enforceable with static code analysis. Also some rules are only enforceable across the whole project, whereas others are enforceable at file level.

Implementing MISRA-C:2012

- How to Implement MISRA-C
  - 1st the compiler
  - 2nd the Static Analyser
  - 3rd Compliance matrix
    - One compliance matrix for all tools in the chain.

Having configured the compiler and the static code analyser to the same version of ISO C, and, for the static code analyser, the correct version of MISRA-C the next step is to produce a compliance matrix.

The compliance matrix has been a part of MISRA-C since 1998, and, just like static code analysis has been a part of C since 1976, it has also largely been ignored. However, it is a corner stone of implementing MISRA-C as it shows what is checked where. More importantly it shows what is not checked automatically by a tool. You will need one compliance matrix per project, as you will need to show that you have covered ALL the MISRA-C rules in one way or another.

Implementing MISRA-C:2012

- Compliance matrix
  - List the rules and show where you
    - Check them
      - With a tool or manually.
    - Deviate them (at project scope)

Guideline	Compiler 1	Checker	Manual Review	Project Deviation	Rule Checked
Dir 1.1			Procedure x		*
Dir 2.1	no errors				*
…					
Rule 4.1		message 38			*
Rule 4.2		warning 97			*
Rule 5.1	warning 347				*
Rule 5.2				R_00102	*

PhaedruS SystemS

Safely From Conception to Completion
www.phaedsys.com

16 of 143

Other than the colours changing in MISRA-C:2012 the diagram has not changed since MISRA-C1 in June 1998! (Although the illustration shown here is a modification on the one in the MISRA-C publication.) The table lists ALL the rules and directives, indicating at what stage they are checked. Some rules may be checked at more than one stage and so appear in more than one column: for example, there may be an overlap between the compiler and the static code analyser. However the main MISRA-C checker should be the static code analyser. The table also has a column for a manual review. Static code analysis and MISRA-C do not negate the need for a formal code review, and indeed some of the directives require that one be done.

The column for project deviation is an addition to the MISRA table. This should be used to list all the rules that will be deviated at project level and then there will be a complete column of check marks in the final, Rule Checked, column. This is because you may be deviating some rules that are not automatically checked by the compiler or the checker. Without this column it is difficult to establish compliance, or rather a positive non-compliance, for that rule in a manual check.

We said earlier (page 6) that a formal code review is on a par with a formal design review, for giving the best Return on Investment (ROI) in a software project. However, assuming you have also used a uniform style across the whole project, as well as carrying out static code analysis and MISRA-C checking prior to the code review, it can concentrate on adherence to the design, the correct use of the algorithms and things like that. You won't get bogged down in the detail of the syntax or have to do mental mind-flips to try and read code that is laid out in a different way to the last file.

A compliance matrix is easy to build in a spread sheet, word processor or even formal requirements management software. It does not matter how you build your compliance matrix BUT YOU NEED ONE.

Implementing MISRA-C:2012

- How to Implement MISRA-C
  - 1st the compiler
  - 2nd the Static Analyser
  - 3rd Compliance matrix
  - 4th Deviation document

Now we come to the difficult part: the Deviation Document. You WILL need to deviate some of the MISRA rules: that is, you will need to decide to or not to use a rule. Trust me - not deviating will cause far more pain and will land you in a lot more trouble in the short, medium and long term than careful and thoughtful deviation. Having accepted that fact, the questions are what rules to deviate, when and why? All the rules in the MISRA-C guide are there to stop some form of misuse or another. Some rules seem to contradict others. Choosing which is relevant depends what you are trying to do and why. All the deviations should be recorded, so a deviation document is essential and it needs to be done very carefully.

Implementing MISRA-C:2012

Deviations

MISRA-C

is

Engineering Guidance

Not a

Bloody Religion

MISRA-C:2012 is NOT just a tick box. You have to read, understand and apply "sensibly". It is not a religion to be followed blindly: it is engineering guidance but you have to be able to justify your decisions. The good news with MISRA-C:2012 is that the rules are explained in more detail than in previous versions. What the team call the headline rules are shorter, but they are not stand alone: you have to read the rest of the rule. The next section, the Amplification, follows on from the headline rule to explain what the rule does. Then the Rationale explains why the rule is there and what the drafting team was thinking. You have to read all the sections carefully,

Some things are "banned" because usually they are known to cause a problem. One example is unions. However unions are required in a few special cases such as packing and unpacking message structures in communication streams. Other things, such as GOTO, are things that are often misused.

GOTO is not bad in itself but the sort of programmer who uses a GOTO as a first option generally has spaghetti code with very poor structure*. The alternative to GOTO, sadly often seen where deviations are not permitted, can be horribly complex and often faulty using nested if else, switch and loop constructs that are very inefficient, not at all elegant and are difficult to fully test and debug. You want a clean, elegant solution that is easily readable, testable and maintainable. When GOTO is the best solution then you can deviate Rules 15.1-15.4 and use it. But remember, you will have to take responsibility for the deviation.

Thus deviations will be required. But before deciding to deviate a MISRA-C rule, make sure that you have read all of the text for that rule, as in MISRA-C:2012 there are some permitted exceptions for some rules. So if you deviate an exception it is going to scream that you haven't read the rules.

Implementing MISRA-C:2012

Deviations

- Claiming MISRA C compliance
  - Only for a project not a company
    - Deviations specific for a product
      - (architecture, compiler, processor)
    - Read and understand the rules.

There are now notes in MISRA-C:2012 on how to claim MISRA-C Compliance for a project: not for a company, only for an individual project. You MUST have a completed compliance matrix and deviation document. They must match each other and match the configuration of the MISRA-C checking tools, including a static code analyser. Theoretically you could claim MISRA-C compliance without a static code analyser, but it would take so much time and manpower that is it not a commercial option.

You must of course adhere to the Mandatory rules if you are working to MISRA-C:2012. (Currently there are 10 of them.)

Do make it clear which MISRA-C you working to. Is it 98, 04 or 12? Of course, with legacy code and third party libraries, you may be claiming compliance to more than one MISRA-C on a project.

Remember you may have to produce both the compliance document and the deviation documents to substantiate your claims. So your decisions for the deviations had better be sound.

Of course you will need complete traceability between the requirements and the source code. Either that or have written a fascinating deviation as to why there is not!

Implementing MISRA-C:2012

Deviations

- READ THE BOOK
  - Understand the rules
  - Which rules are appropriate and which are not?
- Go on a MISRA-C course
  - It will help to get an outside, experienced, and independent view.

PhaedruS SystemS

Safely From Conception to Completion
www.phaedsys.com

20 of 143

I can't stress this enough: you must READ MISRA-C and that means all of it. I have used a slide asking what the Kama Sutra had in common with MISRA-C. The answer is that the Kama Sutra has seven parts but the only one part anyone, outside academia, has ever heard of, is the one about sex. Similarly with the previous versions of MISRA-C there were seven sections but most people only read the section containing the rules. Now, more than ever, you have to read the whole of the MISRA-C document. The headline rules no longer work on their own. As a bare minimum, you have to read the amplification and the exceptions. It is also valuable to read the rationale, which helps to explain the rule. However, to understand how to use the rules, create a compliance matrix and deviation document there are other chapters to read. You are going to have to read 80% of it so you may as well read the rest and actually understand the whole document properly.

Go on a MISRA-C course? Over the last 30 years I have come to realise that very few people fully understand C and would bet that none of them are in your company. This conclusion comes from having spent over 15 years on the BSI/ISO C working group (with 4 years as convenor) and over a decade on the MISRA-C team. Not to mention some 15 years doing tech support for compilers. The C language has expanded from the small "K&R" book in the 1970s through three major iterations of an ISO standard, which most programmers have never seen let alone read. Also there are the "undefined and implementation defined" elements in Annex G: unless your programmers know that by heart and how it is implemented in your specific tools, they don't know C.

Phaedrus Systems recommends one particular training company that specialises in training for embedded/real time programmers. Their entry level summary training for MISRA-C takes 8 hours. The full course is 4 days. It is well worth sending at least one of the team on a course like that. It will help explain MISRA-C and using it safely with C. More to the point, it helps highlight many of the dangerous parts of C that don't behave in the way most programmers expect. Armed with a course like that you will find it far easier to implement MISRA-C and the team will be turning out far more robust and reliable code that actually does what you think it is going to do. It also gives you an independent external sanity check on your thinking.

Note: Members of MISRA-C team cannot give any advice on which rules to implement or ignore. More to the point, most of the MISRA-C working group are fully employed by companies and can't do freelance external consultancy. Of the two that are not, one has retired outside the UK, so you are going to have to work it out for yourselves. You can ask for an official answer on the MISRA-C forum (http://www. misra. org. uk/forum/). Be warned that answers usually take a couple of weeks rather than days. An alternative is to join the MISRA-C & C++ forum on LinkedIn which, while unofficial, has a large number of the MISRA-C team on it.

Implementing MISRA-C:2012

Deviations

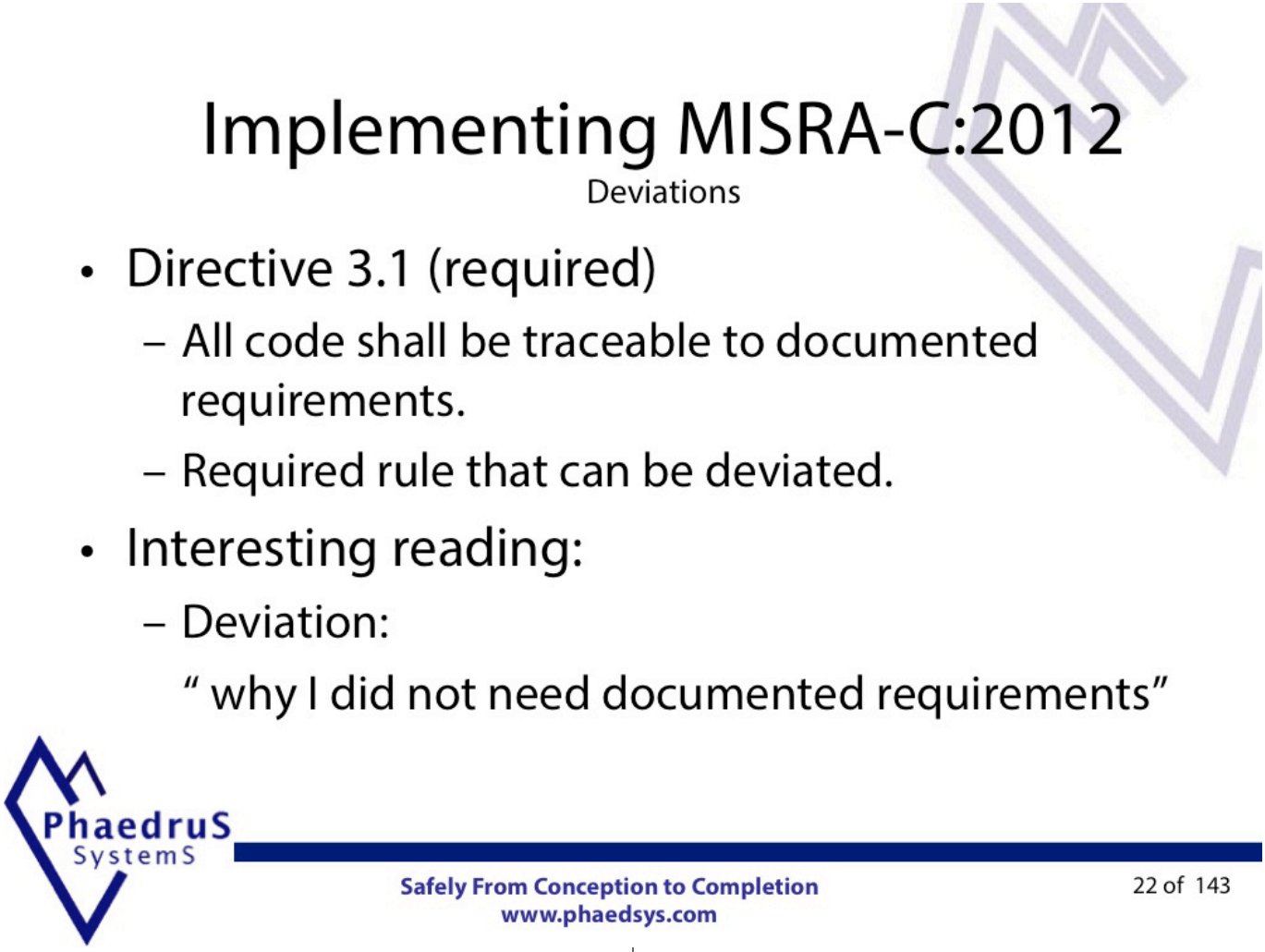
- 10 Mandatory rules
  - No MISRA C compliance without them
    - Started with 30(ish) mandatory
  - Required and Advisory can be deviated.
  - Do you need formal MISRA-C Compliance?
    - If yes you will need to be able to demonstrate it.
      - Compliance matrix and deviation document

In MISRA-C:2012 there are Mandatory, Required and Advisory rules.

There are 10 Mandatory rules: that is rules that cannot be deviated. The MISRA-C Team originally considered about 30 mandatory rules but these were thinned out as people kept finding legitimate reasons for deviation. So there are only 10 out of 159 rules and directives (7%) that the team, and a large number of reviewers, could find no legitimate reason to deviate and are true 100% of the time. This shows clearly that there are not many things that are universally true for C because of architectures, extensions and restrictions and also the nature of the project. As was said earlier, unions are banned but it was expected that those using communication streams will deviate that rule.

If you do not need formal MISRA-C compliance and you are not going to say, even informally, you are MISRA-C compliant then you could deviate the mandatory rules, but you will still need a deviation for them to justify it.

Required rules require individual deviations. MISRA-C:2012 does say that Advisory rules can be deviated without a formal deviation. However it is recommended that you deviate Advisory rules in the same way as you would for Required rules. Then should anyone ever audit your code it will not be held against you. Simply not bothering with deviations for Advisory rules may be permitted technically but the auditors, lawyers, customers, jury etc will look at it and know it is not really right.



The ultimate MISRA-C:2012 rule is Directive 3. 1, "All code shall be traceable to documented requirements." As it is a Required, not Mandatory, rule you can deviate this directive. But, in order to do so, you have to show why you do not need documented requirements or be able to trace them from the code. So you need to come up with a good reason why you wrote code that you did not have proper requirements to write. This rule is a game changer as it puts responsibility on to the people enforcing MISRA-C in the company. If they don't deviate MISRA-C then you need full sets of requirements and code traceability. So you can't start writing code unless the requirements are complete (and someone has signed for them). If someone has taken responsibility and signed for the deviation of 3.1 then you can start writing code without full requirements - that should get a few people thinking!

During the Device Developer Conference 2014 it was suggested that during R&D you could be writing MISRA-C compliant code but not have any requirements. But R&D has two phases, Research and then Development. In Research you play around with ideas, techniques, algorithms etc and here you may not have a full set of requirements. You will, as a matter of course, be using the company coding standard and MISRA-C as even in Research you need the code to do what you think it should be doing. (Also, over time, the programmers will naturally tend to write to the company & MISRA-C standards.)

When you get to the Development phase you should have full requirements, which will partly have come from the Research phase. So when you are making things that will never be released to anyone else you might deviate Directive 3. 1. Otherwise no one has found any legitimate reason since we wrote the directive.

Implementing MISRA-C:2012

Deviations

- Deviation document
  - MISRA C now has Deviation Guidance

•Example deviation record

Project	F10_BCM	Deviation ID	R_00102
MISRA C Ref	Rule 10.6	Status	Approved
Source	Tool: MMMC	Scope	Project
Raised by	E C Unwin	Approved by	D B Stevens
	Signature		Signature
Position	Software Team Leader	Position	Engineering Director
Date	27-Jul-2012	Date	12-Aug-2012

As a lot of people wanted an "approved" deviation the MISRA team wrote a couple of pages on deviation, with an example. The example (shown above) is just a general example. You should modify both the diagram and the suggested methods to fit your processes. The person who wrote the guidance in MISRA-C:2012 worked for a large company in a specific industry and it shows. So use it as an editable template not a rigid form. It really does not matter what the form is or looks like, it is the function that is important. Also regard the rest of the text on deviations as guidance only.

When thinking about the deviation document there are several things that you should bear in mind.

- Each deviation must have a unique reference or identifier.
  - You may want to have the identifier also identify the project which may be a product or a range of products or a library used in many projects or products.
- Each deviation should have a headline explanation which can be used as a standard reference for the deviation.
- There should be a "raised by" and "approved by" each with name as well as a position.
  - The important part is the position as much as the signature. In some companies one person may hold more that one position at some time. In some companies one person might hold all the positions! This does not mean you only need one set of boxes. You should design a process with all the positions and use the positions, even if sometimes it is the same person holding more than one position. It may not always be the case and you don't want to redesign the form because of it.
- There should be a description of the scope of the deviation- whole project, specific module, specific file etc. This is discussed further, below.
  - You may want to use a code letter to indicate the scope in the Deviation ID.
  - For scope there should be for the name of the function, file or module.

Implementing MISRA-C:2012

Deviations

- Some project scope
- Some module scope
- Some file scope
- Some function scope
- Some block or only a few lines

Deviations should be for the minimum area possible. Some will be at project scope and these should go on the compliance matrix as well as the deviation document. Other deviations may be for a specific module. For example the rules on unions might be deviated in communications drivers for packing/unpacking messages. Still others may be deviated where you are interfacing to a third-party maths library. Whilst a module might be several C files, you could have rules that only need to be deviated on a single file in a module when it would be unwise, or simply not needed, to deviate them in other files.

Continuing this theme, you may want to deviate only in a single function in a file, where, for example, a function is doing something specialist that requires a deviation but for the rest of the functions in that file the deviation could be dangerous. Given that some functions can be quite large it follows that you may also want to deviate only for a block in a function.

There is a good reason for being this pedantic. The vast majority of static code analysis tools can use comments in the code to turn rules on and off. For example with PC-lint `/*lint -e(413)*/` will turn off message 413 for the expression following. `-efunc` will suspend a rule(s) for a function. There should also be an additional comment with these in line suppressions giving the deviation reference and a brief "one line" explanation. This should be the official one line explanation that the previous page said should form part of the deviation.

Implementing MISRA-C:2012

Deviations

- Use static & MISRA analyser configuration
  - Minimise size/scope of deviation
- Deviations should be commented in source
  - This should include deviation reference number
  - Brief comment as to WHY deviation

The static code analyser and MISRA checker should be configured to handle your deviations automatically. You don't want thousands of analysis messages for rules you are deviating. However you want to suppress the messages only in the places where you actually intended to deviate. It is no use suppressing the messages where you did not want to deviate.

To achieve this you will probably need to instrument the code with the specific style of comments the static code analysis tool requires e.g.:

```
/*lint rule suppression */
```

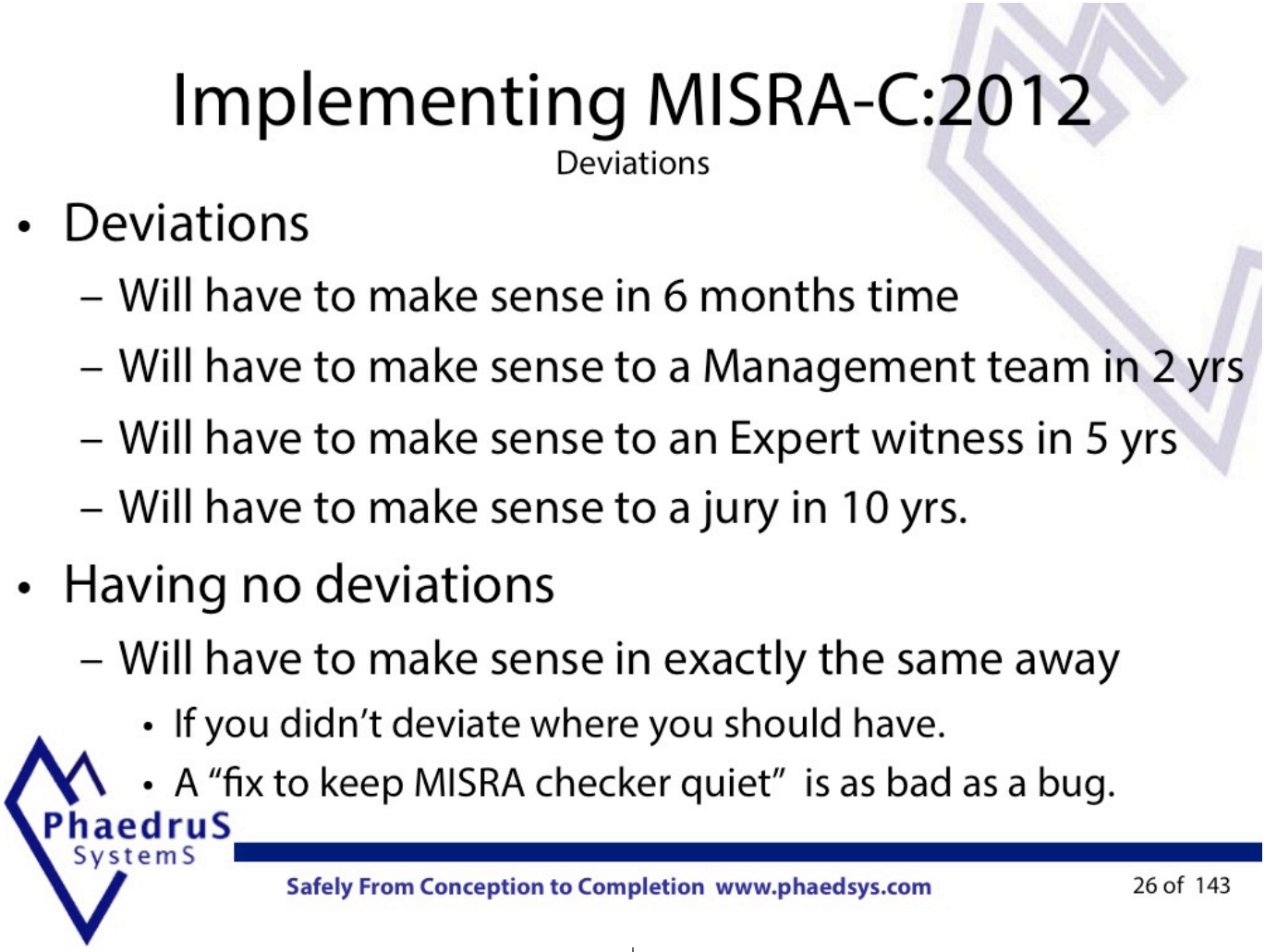
Your file and function information blocks could include these specialised comments with the deviations for the file and the functions. The positioning of these comments may vary from tool to tool.

There is an open discussion as to where the deviations should be placed. Should all the deviations be grouped at the top of the file in one place? For example:

```
// File deviations
…. followed by
//Function-name deviations
…
//Function name deviations
```

Or should the deviations for each function be in the information block for that function?

In all cases you need the deviation ID reference and the one line "headline" reason for the deviation. Anyone reading the code will not only see the reference to the full deviation but from the headline understand why it is there.



Whether a deviation will make sense to you in six months time or to other people at any time, is a hot topic at the moment (2014). This is because several high profile court cases involved software that deviated from MISRA-C. The problem is often that deviations are done to solve an immediate problem. Most deviations will be raised during development, not before. Only some things (e.g. compiler extensions) can be deviated while you are setting up the tool environment.

YOU have to justify the deviation. Someone will have to sign for it and take responsibility. The words "cool", "neat", "radical" do not appear in ISO Standards or in most court cases other than from the defendants. It is no use making up a deviation that, like the code, makes sense now but you will not be sure about later. Will it make sense to you and the team in 6 months time when you are doing a review?

Would it make sense to the management in a review following a customer having a problem and the bug is in that area of code? A lot of management is not technical, so is the Deviation in plain English that makes sense to someone other than a fellow programmer? It does make sense to ensure the deviation is clear.

In many cases where an expert witness has looked at the code and documentation, it is because something has gone wrong. Humour is usually in very short supply. What is a clever technical funny line can come across as unprofessional. Expert witnesses & auditors who as people may be signed up members of the Monty Python Fan Club have to be professional and un-funny in their work.

Finally whilst you are not expecting it, you should think, "Will my deviations make sense to a non-technical lawyer working in my defence?" The sub-sub contractors working for the OEM that supplied Toyota in 2001-2004 were not expecting to have their code scrutinised in a major legal battle over a decade later. The deviation must not be a hook the prosecution lawyers can use. Can the deviation be understood by a jury made up of Joe the Plumber, Aunt Flossie, your Mum and the lads you drink with down at the sports bar?

Just as important as deviating is not deviating: if you don't deviate but do some "clever code" to "keep the MISRA-Checker quiet" you are likely to have that picked up by an auditor, expert witness or lawyer. This shows "bad practice" and "sloppy procedures" and a "bad attitude" in the development team. Even if the jury is told to disregard the comments from the lawyer the idea will have been planted in their heads.

Implementing MISRA-C:2012

Deviations

Blind adherence to the letter without understanding is pointless. Anyone who stipulates 100% MISRA-C coverage with no deviations does not understand that they are asking for.

In my opinion they should be taken out and… Well…

just taken out.

Chris Hills, Member of MISRA C Working Group
MISRA Matters Column in MTE June 2012

Deviations are something the MISRA-C team has been regularly and frequently asked about. Questions fall into two groups. One is, "How do I deviate?" which I will cover next. The others come from those who are told, "100% MISRA-C with no deviations" [TICK]. This mandate usually (actually always) from people who don't understand what MISRA-C is or how to implement it.

As mentioned there are only 7% of the MISRA-C rules that are Mandatory. That is rules that are applicable 100% of the time. Therefore we hope that 99. 9999% of MISRA-C users will deviate the appropriate rules. MISRA-C can be counter productive when some manager demands 100% compliance without realising he is dangerously handicapping the project. The team fights with the standard resort to all sorts of time consuming and, in some cases, dangerous tricks to get round the warnings from the code analyser. The team is spending a lot of time getting hideous and less efficient code.

A4 size copies of this slide are available, signed, for your manager's office wall!

Deviations WILL be required. Just as the rain must fall but too much is a flood.

Implementing MISRA-C:2012

Deviations

- MISRA-C-ADC
  - Approved Deviation Compliance for MISRA-C:2004
    - Effectively 1&1/2 pages
  - There will be an ADC for MISRA-C:2012
    - Much larger document(s)
    - But not in 201* , don't hold your breath.
- For Deviation guidance you are on your own
  - You will have to take responsibility for them

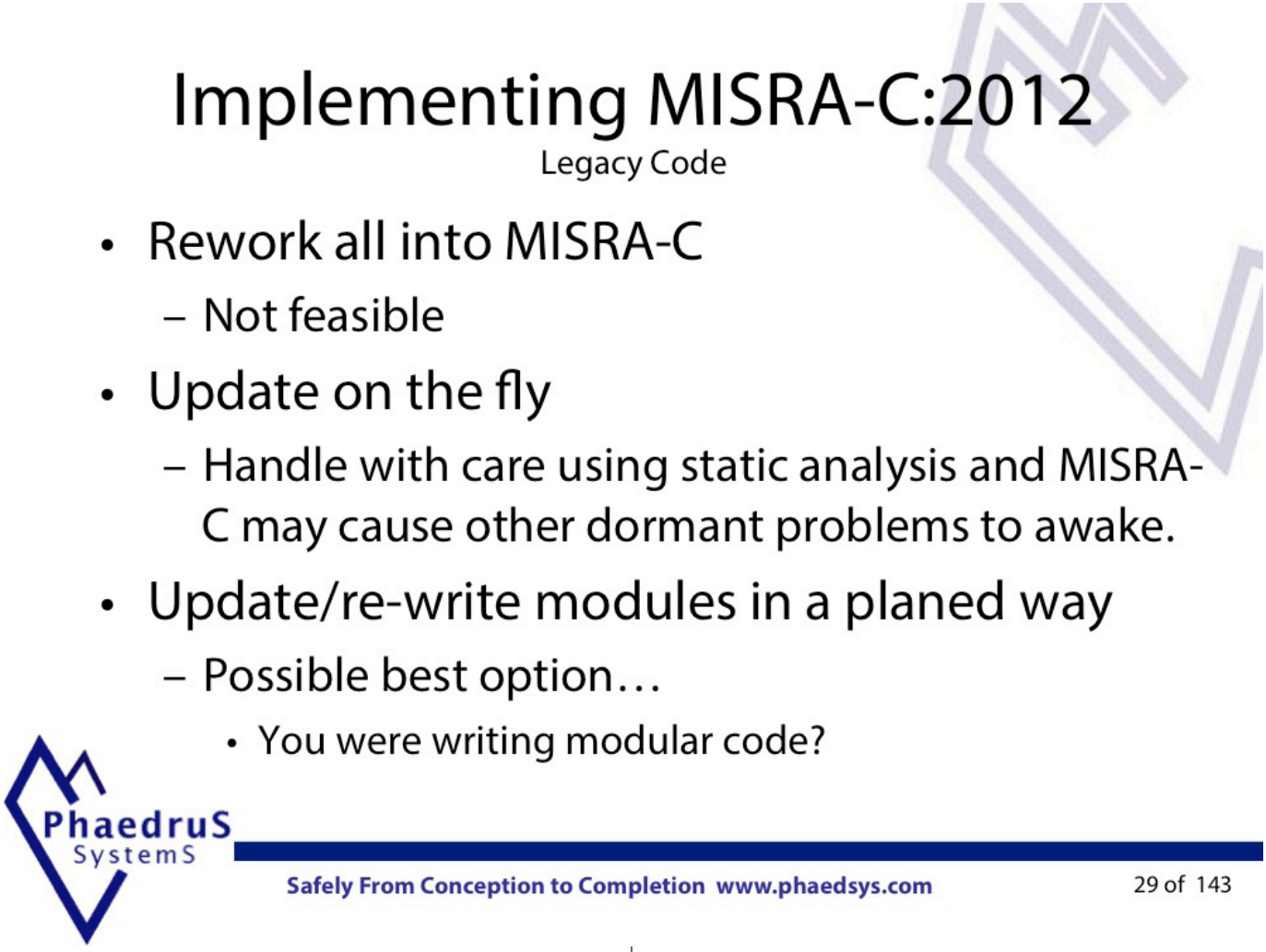
PhaedruS SystemS

Safely From Conception to Completion
www.phaedsys.com

28 of 143

The MISRA-C team has produced an Approved Deviation Compliance for MISRA-C:2004 However this is effectively only 1 ½ pages of text when you remove the title and admin pages. It was largely put out for non-technical reasons to help one particular industry in a particular country. It contains less advice than this presentation!

As of the Summer of 2014 the MISRA-C team is working on a new Approved Deviation system. Bearing in mind progress on MISRA-C documents over the last 15 years I do not expect to see the new MISRA Approved Deviation Compliance before 2016.



All your new code is sorted. But you still have legacy code. This falls into several groups.

- Old projects that are mothballed bar the occasional minor bug fix.
- Old projects that are updated with new features every now and again.
- Current projects that are in their nth revision and are not MISRA-C compliant or are possibly compliant to an older version of MISRA-C.
- New projects that contain local library files or code re-used from older projects.

Deciding what to do and when to do it is a matter for judgment calls: however someone needs to take responsibility for the strategy AND have the authority to implement it. In a perfect world you would stop all development in the company and update ALL the source code to MISRA-C:2014. Actually that is not true: In a perfect world you would be using Modula2 or Oberon in place of C. However neither Oberon nor stopping the company to update all the code is going to be feasible. More practical is to update files as you go along, usually as they are modified. This is the solution most jump at, but it is also fraught with danger. MISRA-C restricts the C language and of course you will be using a static code analyser, even if you didn't when the code was first written. This will tighten up, and possibly modify, the behaviour of the code to the specified behaviour. But the system may only be producing the correct output because of other "loose" code and making a function or file correct with static code analysis and MISRA-C may cause latent bugs and problems in other areas to appear. This is why it is sometimes said that static code analysis causes more problems than it solves. In reality it doesn't: it removes some problems and makes others more obvious.

The best and safest answer is to update and re-write modules. Now I don't propose to get into a long discussion on what a module is - you can go to the internet forums for that. In this context a module is a functional, self contained group. It may be a file or collection of files. The interfaces to a module should be well understood. Therefore the code contained within should be updatable without affecting code elsewhere in other modules. At this point you should use static code analysis first then update to MISRA-C compliance a file at a time, on a copy of the code in parallel to the main development.

This approach does assume that you were writing modular code in the first place. If you were not then you have more problems and it may not be possible to easily and safely update the code to MISRA-C compliant code.

Updating between versions of MISRA-C. As your code will already have been statically checked and MISRA-C:1998 or MISRA-C:2004 checked, all you need to do, after having produced a compliance matrix and deviation document for the version you are moving to, is to run the new MISRA-C:2012 checker over the code, file by file, and adjust the code according to the warnings.

Implementing MISRA-C:2012

Legacy Code

- Team of 6, module in large UNIX system
- Linting code 20 min a day for 6 months
- Cleaned up module.
- Things stopped working
- Data and calls coming in from an other module faulty. Masked by complimentary errors in our module.

To provide a perspective on cleaning legacy code, some years ago I was one of a team of six working on version five of a large UNIX project, alongside eight other teams of four to eight people. Five of us were new, so we looked at the code to familiarise ourselves with it. We noticed many anomalies. Therefore we spent 20 minutes a day running lint over our code. We did this for six months until we had removed all the lint warnings for the old code.

We found we had removed a lot of redundant code, a lot of dead code, unravelled the 10 include files to discover we had over 120 nested include files. Many of them were included more than once, without duplication guards. When we tried to rationalise them, things stopped working. It appeared that due to the dependencies some things that were being declared as one thing in a header file were used in another "include" file and were then re-declared in another header file that was included in some more header files before the first header file was included, changing the item back to the original declaration. Are you keeping up? For example "colour" was declared as "red" then "blue" for a while, then "red" again. This happened more than once. We had to re write some of the code with "colour1" and "colour2" and it took some time to work out if a particular use of "colour" had to be 1 or 2.

We also found that after tightening up the module and all the interfaces our module suddenly stopped working. We tracked it back to an input from another module. However on inspection it turned out the parameter was passed through that module, without any checking at all, from yet another module. Had the module passing to ours done the appropriate checking it should have flagged an error on receiving the data. We had to red flag that, so that they started to implement checking. We also had to issue a fault report to the originating module.

These kinds of problems are not uncommon when you start to clean up code. However you will then have far more reliable and robust code with far fewer surprises. More to the point you will have removed the problems before the customers find them and they cost a lot more to sort.

Implementing MISRA-C:2012

Legacy Code

- "don't fix what isn't broke"
  - See Toyota-Brookout transcripts
  - Old code will be judged on best practice at time of shipping
- "don't fix what isn't broke"
  - Arianne 5 Rocket used modules from Arianne4
    - Arianne 5 failed at 29 seconds from launch

PhaedruS SystemS

Safely From Conception to Completion www.phaedsys.com 31 of 143

There is a tendency to say: Don't fix what ain't broke. The problem here is that in 5, 10, 15 or even 20 years' time your code, or code you are now responsible for, might end up in court. You might say in 10 years' time I won't be here. However as the current team shipping code you are going to be seen as responsible for what you ship, even the legacy code from 10 years ago that you did not write. You did test it, didn't you?

The code will be judged on the standards at the time the code was shipped: which is good. So in 2024 code shipped today will be judged on today's (2014) best practices - i.e. full static code analysis, full unit testing and MISRA-C:2012. Of course, if the project is an on-going (and properly documented) MISRA-C:2004 project and you are carrying out full static code analysis and full unit testing, you may be able to get away with it for a while longer. However new projects will have to be MISRA-C:2012 compliant.

The Ariane 5 rocket crashed because they reused code modules from Ariane 4. These modules were fully tested against the Ariane 4 flight plan, but not against the full Ariane 5 flight plan. So if you reuse code you may need to fully test against today's standards.

In both this case and that of Toyota, the failure to spend a couple of tens of thousands cost many billions. There are many more cases where not re-testing and bringing code up to current standards has cost far more than it appeared to save, but these mostly go un-noticed as, unlike Ariane and Toyota, they are not in the public domain.

Implementing MISRA-C:2012

3rd party Libraries

- MISRA-C compliant 3rd party libraries
- You will need their
  - Deviation document
  - Compliance document
- If they use the same static analyser/MISRA-C checker
  - You could re-rest with the same configuration they used
    - They may have different deviations

Legacy code, of course, includes any company libraries where you have the source.

NOTE: It is always a good idea to periodically look at company library code. You want library code that you are reusing often to be of the highest quality, robust, reliable but also compact and efficient. You can tune it and ensure it is well tested and documented to your requirements. If you are buying-in, or downloading from the internet, third-party code, this is a different matter. If you pay for support for third-party libraries you will be getting maintenance releases and updates.

There is no such thing as MISRA-C certification. When MISRA-C first started in the mid 1990s it was a local guide for two UK automotive companies who were both using the same static code analysis tools. Since then MISRA-C has gone global and many other tools vendors asynchronously started to support MISRA-C in their own way. These tools may use different but equally valid techniques, methods and algorithms. Therefore the way in which they detect errors and the way they flag them may vary from tool to tool. Usually they are different not wrong. For example Pale Black == Dark White == Mid Grey (or Dusk as it is called by the marketing team).

There are vendors of third-party code such as RTOS, OS, graphics, communications stacks such as USB, CAN, TCP/IP, Modbus etc and middleware such as databases, that claim MISRA-C compliance. If you buy this code you should ask for the deviation documents, showing which rules have been deviated and why. The vendor's compliance matrix will show where they check each rule and the name of the static code analysis tool they used for MISRA-C checking.

Depending on what your project is you may take this at face value and enter it into your project documentation.

Implementing MISRA-C:2012

3rd party libraries

- Non-MISRA-C compliant 3rd party libraries
  - May have to deviate whole library
  - May have to do wrappers for include files.
  - Might suggest that library vendor produce MISRA-C compliant include files.
  - Worst case may have to change libraries

For non-MISRA-C compliant libraries there are several options. Obviously you re-write the whole library as fully MISRA-C compliant. Ok - that is unlikely to happen unless it is a small, unsupported/obsolete library which you want to continue using.

Where it is a large and supported library the option is to deviate the library. If possible write MISRA-C compliant wrappers for the include files which do the appropriate range and error checking, const parameters etc. I would suggest talking to the vendor and applying pressure for them to produce the MISRA-Compliant include files. Or do a deal for several years' free maintenance if your team does it for them.

Then of course you go on to apply pressure for them to make the rest of the library MISRA-C compliant. If you only have the object or binary versions of the library you will not know what the source looks like but most libraries have a source code version. Projects for critical systems, which are likely to require MISRA-C compliance, will normally buy source code versions. In this case the vendor has more of an incentive to make their library MISRA-C compliant, as otherwise they will be excluded from many projects.

You can expect over the next few years that more libraries and middleware will become MISRA-C compliant. Until then deviating the library may be the only option.

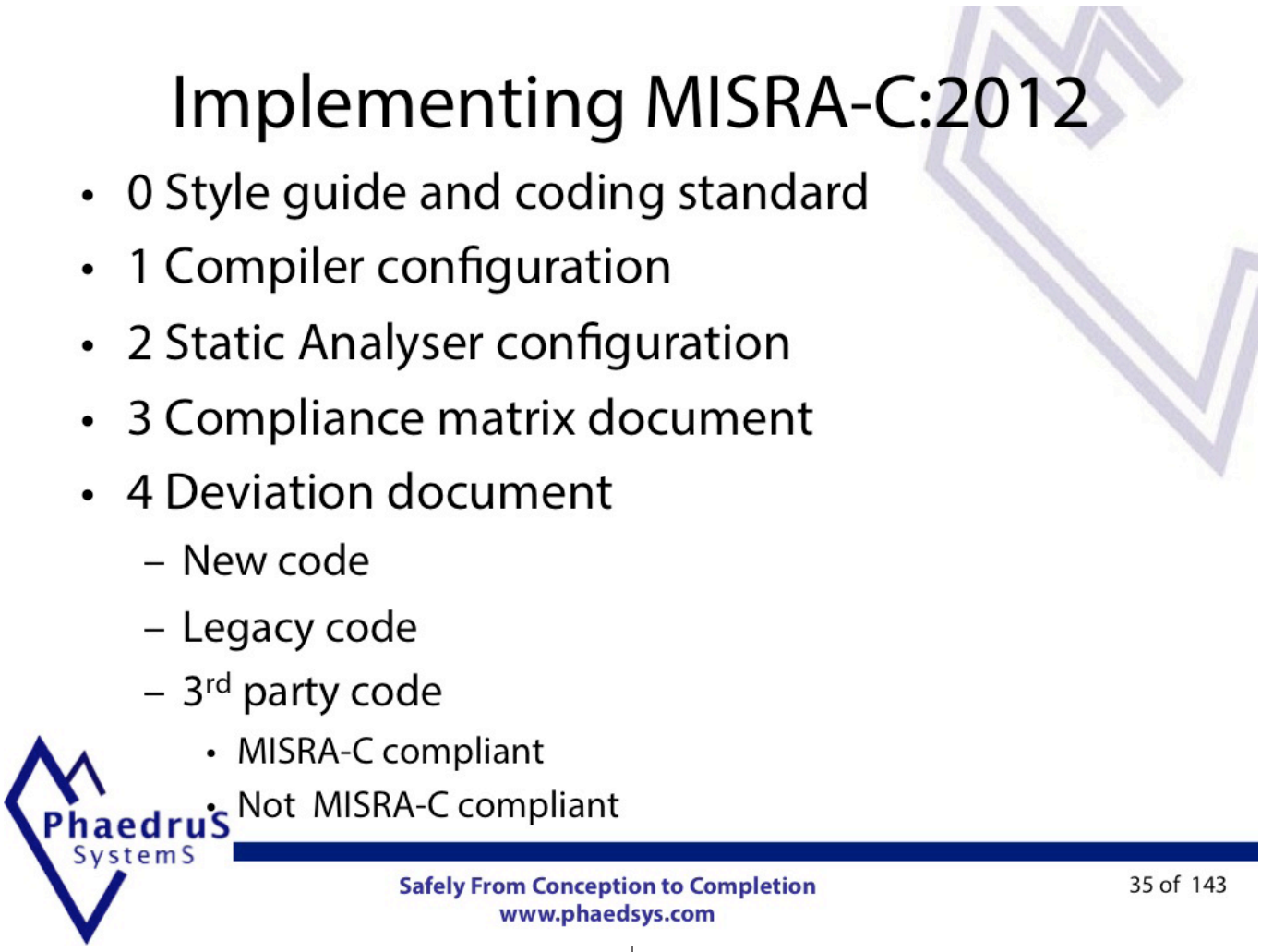
Implementing MISRA-C:2012

standard libraries

- Compiler Standard Libraries
  - ISO C specified.
  - Not going to be MISRA-C compatible any time soon
  - Not going to break existing code base
  - Possibility of MISRA-C wrappers for include files
    - But don't hold your breath (shout at compiler companies)
    - MISRA-C bans a lot of standard library anyway.

The [ISO] Standard Library that usually comes with the compiler is specified in ISO 9899. It is not going to change to be MISRA-C compliant any time soon (read decades). This is partly because ISO C working groups don't like breaking legacy code and partly because if they do make the change, it will take a very long time to do and they have many other, higher priority, things to do.

The solution can be, as with legacy code and other third-party libraries, to write MISRA-C wrappers for the include files you do use. Bearing in mind of course that MISRA-C bans the use of much of the standard library anyway. The answer could be, as suggested previously, to put pressure on your compiler vendors to provide a standard library with a set of MISRA-C:2012 compliant wrappers. As many compilers use the http://www.dinkumware.com standard libraries as a basis it might be easier to do than you think.



Check list for implementing MISRA-C:2012

- Style guide and coding standard.
  - This is a must have. Without uniformly written code you are just wasting your time. Your style-guide/coding-standard will have more in it than just the MISRA rules.
- Configure the compiler.
  - Note, understand (and document) where it has differences from and extensions to ISO C.
- Set the compiler warning level to maximum.
  - Investigate/remove all warnings in compiled code. No matter how theoretically correct the code is to ISO C and MISRA-C it is the compiler that is producing the binary. If there is a warning it must be investigated. I am sure this point does not even need mentioning for compiler errors.
- Configure the static code analyser.
  - It needs to match and support the compiler and target MCU, including noting and documenting the support for MISRA-C, ensuring that the static code analyser supports both the same ISO C as the compiler and the same version of MISRA-C that you are implementing.
- Produce the Compliance Matrix.
  - Having configured the compiler, static code analyser and MISRA-C checker, you are now in a position to produce the compliance matrix, though you may not be able to complete the project deviation column until after the next step.
- Produce a Deviation Document.
  - With the tools set up and a Compliance Matrix complete, you can produce your deviation document. This will take some time and require information from the previous steps as well as an understanding of the project which will come from the design. You will probably need several sections in the deviation document to cover the different sets of deviations for new code, legacy code and third-party party code. Where "MISRA-Compliant" code is used it should come with its own documentation including a compliance matrix and deviation guide.

Implementing MISRA-C:2012

- For obvious reasons I can't tell you which rules to deviate other than
  - Rule 1.1
  - Rule 1.2
  - Rule 1.3
- You will have to read MISRA-C:2012 to see why

I cannot tell you which rules to deviate or why. YOU have to take responsibility for your own deviations (or not). Not deviating can be as much of a crime as deviation depending on the rule and the situation. It is not possible to have 100% MISRA-C compliance with no deviation. You have to deviate Rules 1. 1, 1. 2 and 1. 3. If you do not deviate these three rules (at least) there is something seriously wrong.

One of the major changes in MISRA-C:2012 compared to the previous editions is that Headline rules don't work on their own. The Headline rules were intentionally kept short and are augmented by the Amplification, Rationale and any Exceptions. Therefore the rule and all the supporting text has to be read and understood in order to implement or deviate a rule. This was intentionally done to keep some rules from ending up a paragraph long. We wanted things readable.

There is no substitute for having a copy of MISRA-C if you are intending to work to MISRA-C. Actually it is impossible to implement MISRA-C:2012 without reading and understanding the whole document which is why I recommend at least one of the team attends a good MISRA-C course to gain an in-depth understanding of not only MISRA-C but the widely misunderstood idiosyncrasies of the C language.

Phaedrus Systems

Safety Critical and High Reliability Embedded Systems Tools

Implementing MISRA-C:2012

Any Questions?

The final notes from Part One from Device Developer 2013 still hold true: MISRA-C might save your project as part of a properly implemented system. On its own it is just one more tool in the box. Like any other tool it can do more harm than good if misused.

If you have any questions there are several places you can go for help:

For Authoritative and definitive statements from the MISRA-C working Group got to www. misra-c. com/ forum

For general discussion on MISRA-C and C++ there is the LinkedIn forum "MISRA-C and C++". This is where most of the MISRA-C team hang out.

Otherwise for general MISRA-C information, static code analysis, general software engineering and project control information, contact Phaedrus Systems MISRA@ Phaedsys. com

Implementing MISRA-C:2012

Deviations

Blind adherence to the letter without understanding is pointless. Anyone who stipulates 100% MISRA-C coverage with no deviations does not understand that they are asking for.

In my opinion they should be taken out and… Well…

just taken out.

Chris Hills, Member of MISRA C Working Group

MISRA Matters Column in MTE June 2012

MISRA C:2012

Implementing MISRA-C:2012

May 2014

First edition May 2014

Phaedrus Systems Library

The Phaedrus Systems Library is a collection of useful technical documents on development. This includes project management, integrating tools like PC-lint to IDE's, the use of debuggers, coding tricks and tips. The Library also includes the QuEST series.

Copies of this paper (and subsequent versions) with the associated files, will be available with other members of the Library, at:

http://library.phaedsys.com

PhaedruS SystemS

The Art in Embedded Systems comes through Engineering discipline.