Integrating PC-lint into IAR Embedded Workbench

Second Edition

by

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

The Art in Embedded Systems comes through Engineering discipline.

Contents

Integrating PC-lint into IAR Embedded Workbench

The most effective way of doing static analysis is to do it frequently as the code is written from within the code development IDE. This app note will explain how to integrate PC-lint into the IAR Embedded Workbench (EWB) to permit continuous static analysis as the source code is written.

Background

In 1976, before the C language was even complete, Johnson (part of the C and UNIX team with Kernighan, Ritchie & Thompson) had created static analysis and the first lint (Johnson 1979) because programmers were, according to Dennis Ritchie, using "legal but dubious constructs" (Ritchie 1993). A compiler translates syntactically correct source even if it is semantic rubbish. Some simple examples of this include assigning a long to a char and losing 3 bytes of data; falling foul of integer promotion rules, which most people misunderstand; having a local variable inadvertently mask a global variable - the list goes on and on.

Due to the "trust the programmer" ethos around the C language, static source code analysis is required to find all the legal but very dangerous things that can inadvertently get into the binary. Therefore static analysis should be used frequently. Static analysis looks at the source logically, without compiling and running it, hence "static" analysis. Static analysis can find up to 80% of non-functional bugs very quickly. Trying to find these same bugs dynamically within a running system would require extremely long and complex testing.

PC-lint from Gimpel has been around since 1985 and has proved itself over the decades. PC-lint is the best "bang per buck" tool you can get for static analysis. It is a simple command-line tool of the UNIX/dos philosophy and the lack of GUI means it can be integrated into most IDEs and make or build systems.

PC-lint removes any excuse any programmer has for not using static analysis on C or C++ code.

Configuration of the PC-lint files

PC-lint is extremely flexible and the C language has many dialects or variations associated with the many compilers and target architectures in use. Potentially, therefore, there are very many switches and options that

need to be set. In addition, there are configurations for the IDE integration, compiler and project paths.

Fortunately, this is normally a one-off procedure and most of the sets of options have already been set up in pre-written configuration files. These options are in the form of switches for the specific C used in the compiler for a particular target MCU, for the IDE environment and for the various coding standards, such as MISRA-C.

The PC-lint configuration files are ASCII text files and are human readable and editable. The switches will require the PC-lint manual for interpretation, as they are somewhat terse in the best traditions of UNIX. For example these three modifications of message 401

```
e401   /* symbol not previously declared
 static */
+elib (401)
append (401, [MISRA 2004 Rule 1.2])
```

Normally the amount of manual switch setting is minimal. These will normally be mainly for the include paths for the project .c & .h files and system .h files. They do require care and full explicit paths should be used with the path in "". Spaces and non-alphanumeric characters are not recognised by PC-lint and, as with UNIX, characters such as –, + etc act as switches or commands and can have unforeseen effects. Usually getting the include paths set up is the most troublesome task for the first use of PC-lint.

For the IAR EWB there are four configuration files required, three of which are provided and one is generated by the compiler. The three provided are:

1 `iar-co.lnt` This is the generic IAR compiler file which is used for all IAR compilers. This works because IAR uses the EDG Parser and Dinkumware Library across the whole range of compilers. **This file should NOT be modified.**

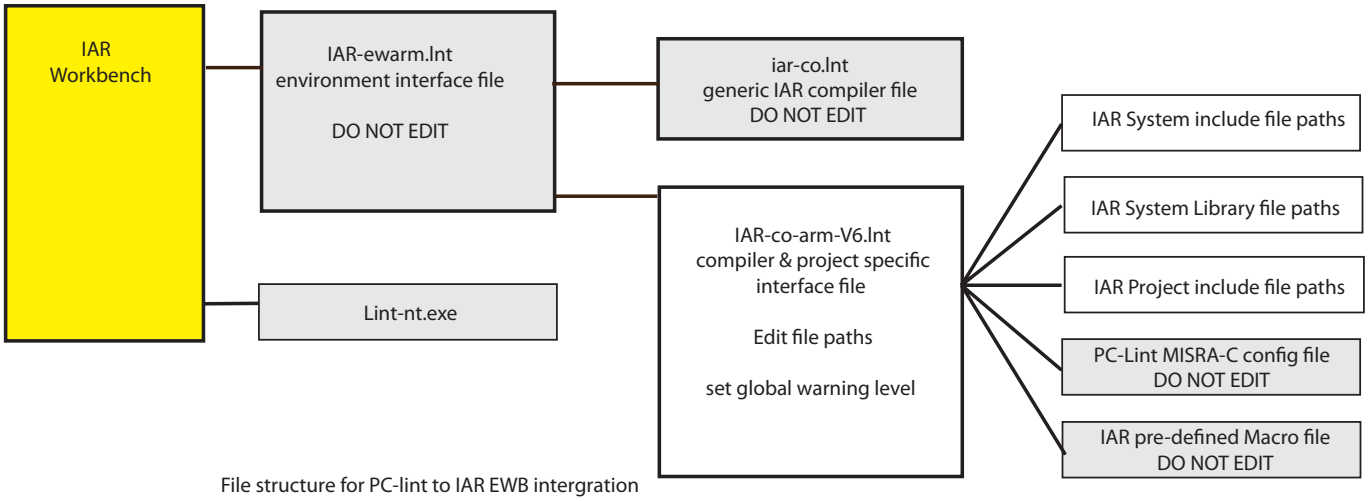
2 `iar-co-***[-v*].lnt` This is the architecture specific IAR compiler file. For example `iar-co-ARM-v6.lnt` is for the IAR V6 ARM compiler. Some architectures have one file, whist others, like ARM, have several (e.g. V4, V5, V6). This file will need editing during set up as it contains all the system and project library paths.

3 `iar-ew***.lnt` This is the EWB environment file e.g. `iar-ewarm.lnt`. This sets-up the integration to the EWB, so PC-lint messages are shown in the output window. It calls the other two compiler files above.
NOTE: this file contains several non-visible ***Control-A*** characters, which get the hyperlinks in the EWB message window. (See the IAR app note *Browse your Application* (Sporrong 2011) on the use of hyperlinks in EWB.) This file should only be looked at in the EWB editor, not a standard text editor. If you do view the file in a text editor do NOT save the file or the ***Control-A***s are likely to disappear. **This file should NOT be modified.**

4 In addition to the three IAR files provided you will need to generate a predefined macros .h file. This file is generated by the compiler, as described below. **This file should NOT be modified.**

If you are using MISRA coding guidelines, then you will need the Gimpel-supplied MISRA-C file (MISRA-C:1998, MISRA-C:2004 or MISRA-C:2012.)

NOTE: Due to the way MISRA-C was developed, there is no approved method, or approved error messages, for static analysers to report MISRA rules violations.



File structure for PC-lint to IAR EWB intergration

Integration into IAR Embedded Workbench

The following will explain how to set up the IAR Embedded Workbench (EWB) with PC-lint to analyse single files or complete projects with multiple files. Both the single file and multiple file set-ups should be completed, as some classes of problems and errors only appear across the whole project if there are multiple files. You should set up for single file analysis first, as the notes below contain the full instructions. The multiple file section has the additional modifications for multiple file analysis you need to apply after setting up single file analysis.

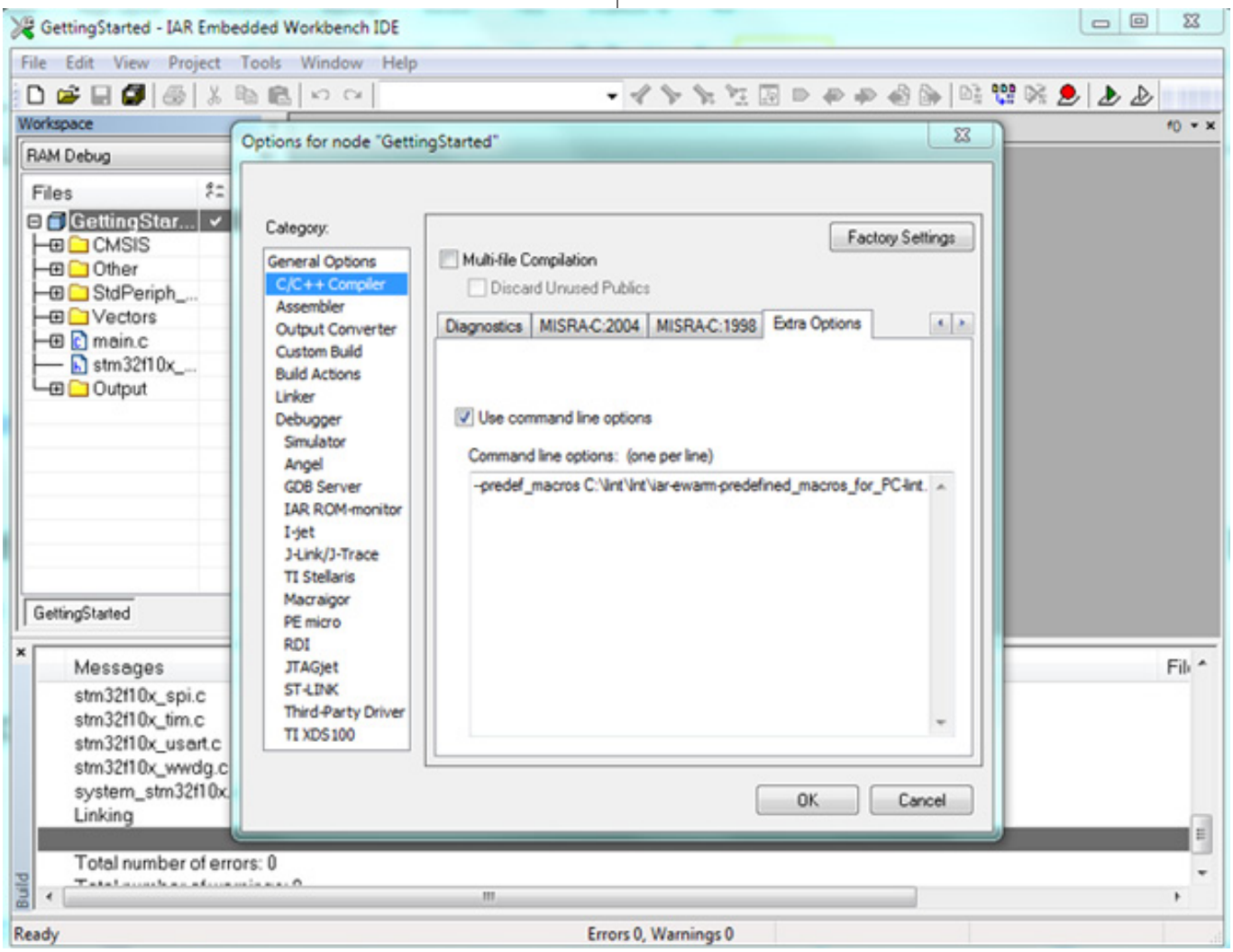
We assume that you have both the IAR Embedded Workbench and PC-lint correctly installed on your computer. These configurations files assume that PC-lint is installed to `C:\lint` with the configuration files in `C:\lint\lnt`. You can install PC-lint to a different location but this will require editing the environment configuration file. Before doing this, please see the note above on ***Control-A*** embedded in the file. The EWB editor is suitable for editing the environment configuration file as it correctly handles the ***Control-A*** characters.

PC-lint set up for Single File Analysis Pre-defined Macro file

The first thing to do is to create the pre-defined macros .h file. Open Embedded Workbench with your project. This assumes you have EWB set up with the classic three-window view: the message window along the bottom and the project window to the left of the editor window.

The first thing to do is to create the pre-defined macros .h file. Open Embedded Workbench with your project. This assumes you have EWB set up with the classic three-window view: the message window along the bottom and the project window to the left of the editor window.

1. Right click on the project root in the file window in EWB and select `Options` in the pop-up menu.
2. In the `Options` dialogue select the category `C/C++ Compiler`



3 In the C/C++ Compiler dialogue you need to select the Extra Options tab. The list of tabs appears to go from Language 1 to List. Next to the List tab you need to use the small right arrow to go to the actual end of the tabs, the last one being Extra Options.

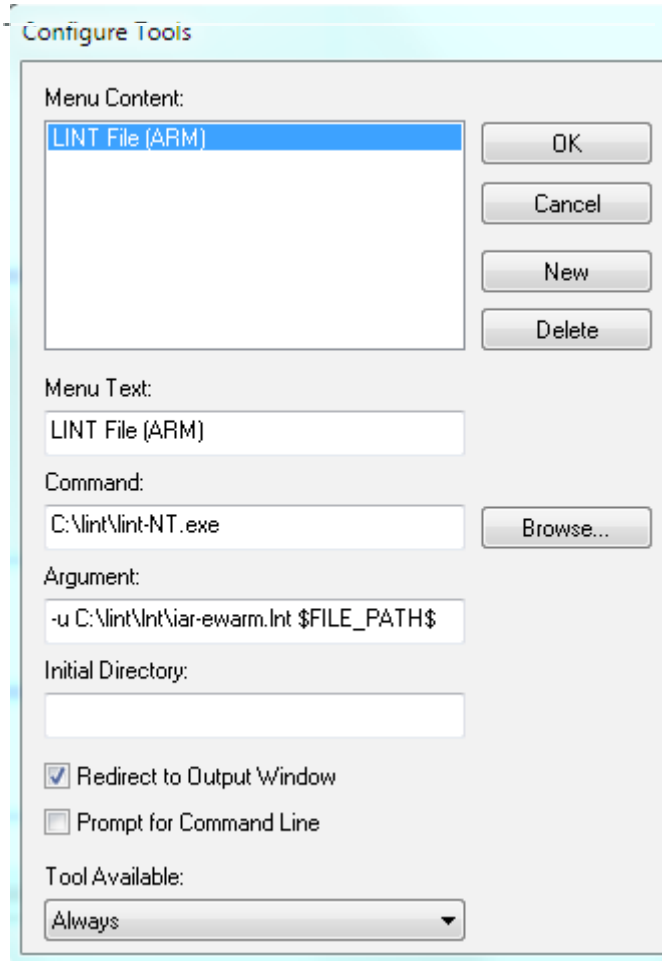
4 In the Extra Options dialogue, enter the line below:

```
--predef_macros C:\lint\lnt\iar-ewarm-predefined_macros_for_PC-lint.h
```

The command is -predef _ macros which outputs all the pre-defined macros to the file following. You need to ensure the directory part of the command is the actual location of PC-lint on your computer. You can name the file anything you like but consistency is important as you may use lint with more than one compiler or project.

5 Do a build all to generate the file. The build all forces a compile of all files whereas a build will only re-compile files that have been changed.

6 Save the pre-defined macros .h file to the lint configuration file directory - in this example to C:\lint\lnt.



7 Having generated the pre-defined macros file go back to the Extra Options dialogue and unclick the Use Command Line Options, otherwise a file will be generated on every compile for one or more files. It is probably best to also remove the line completely from the dialogue, even with the check box de-selected.

Move Configuration files

8 Put the three IAR files iar-co.lnt, iar-co-***[-V*].lnt and iar-ew***.lnt into the PC-lint configuration file directory, C:\lint\lnt.

Set up tools dialogue

9 We now have all the files to start the set-up in the Tools dialogue in EWB and adjust the iar-co***[-v*].lnt file. Doing it this way will mean that the error messages which appear in the EWB message window will give feed back on the settings.

10 In EWB, open the Tools menu and select the Configure Tools option. When the Configure Tools dialogue opens click the New button. Nothing much will appear to change other than New Tool will appear in the top box labelled Menu Content: and the box below Menu Text:.

11 In the box Menu Text: enter what you want to appear on the tools menu as the name for PC-lint - something like Lint File. You should probably also enter the compiler type, as you may have more than one compiler. e.g. ARM, 8051 MSP430 etc. In this example it is Lint File (ARM). You can also have project-specific configurations.

12 In the box Command: you need the first part of the command line that calls PC-lint. The easiest way to do this is to browse to the location of PC-lint, C:\lint in this case. Note that no matter what the version of Windows you are using, NT, XP, Vista or 7, you need lint-NT.exe.

13 The final window is Arguments: which is in three parts: the first is the switch -u, the second is the path to iar-ew***.lnt (in this case iar-ewarm.lnt), which should be C:\lint\lnt\iar-ewarm.lnt and the third part is the IAR internal Macro $FILE _ PATH$ which is the highlighted file in the editor.

14 The box Initial Directory should be left blank. The box Redirect to Output Window should be checked. The box Prompt for Command Line

should be left unchecked and the Tool Available menu set to Always.

15 Confirm all the entries, and then click OK to close the dialogue. Then click on the Tools menu to confirm that Lint File (ARM), or whatever you have called it, is the last entry in the menu.

That completes the tools menu installation.

Configure to include directories and project specific items

16 We now need to adjust the file `iar-co-***[-V*].lnt` to get the correct include paths for the compiler and the project. Open the `iar-co-arm-V6.lnt` file in a text editor.

NOTE:- On a project that already has all its source files and has been compiled, the compiler generates the file `main.pbi.cout` which is placed in the project `debug/obj` directory. This file contains all the library and project paths used. The paths can be cut and pasted in to the `iar-co-***[-V*].lnt` file and will only require a little editing, mainly for `-i` and for quotes.

17 The first section to check is the IAR compiler directories. Please take care of the quotes around the file path as you may need to adjust the part `"Embedded Workbench 6.4"`. The `"\arm\inc\c"` should not need to be changed.

```
//MODIFY to point to your IAR Embedded
Workbench installation –
-i"C:\Program Files (x86)\IAR Systems\
Embedded Workbench 6.4\arm\inc\c"
```

For every directory that should be in the search path, you will need a line like this. The file as supplied should have all the directories required for the IAR installation.

18 The same action will be required for the library directories using the `+Libdir` directive, for example:

```
+libdir("C:\Program Files (x86)\IAR
Systems\Embedded Workbench 6.4\arm\
inc\c")
```

is for the IAR system libraries. At this point you could also add any other third party libraries that are normally used and that are not project specific. This will include at least the directories for the .h files that will be required when passing any .c file in the project.

19 The next section is for the project directories also using the –i directive. The directories should be in "quotes " and there should be one line for each place that has .c or .h files related to the project. This will include any .h files for any libraries not included above.

```
//MODIFY and ADD your project include
  paths here
-i"C:\WORK\iar-arm-st\ST\STM32F10x\
  IAR-STM32-SK\GettingStarted"
```

20 The following sections of the file should NOT need editing. They are specific to the IAR compiler

```
EWARM Size of Scalars
EWARM extended keywords
//MODIFY for problems with internal
  compiler library files: ichooser,
  iutility and xmemory...etc..
-elib(19)       // Usually about stray
  semicolons
-elib(1076)     / Anonymous union
  assumed to be 'static'"
-elibsym(1512)      // Base class
  destructor is non-virtual
-elib(46)           // For bitfields
  with non-int fields
```

21 If you are using MISRA-C (or the Scott Meyers' or Dan Saks' guidelines) you need to place the appropriate file e.g.

```
au-misra1.lnt         for MISRA-C:1998
au-misra2.lnt         for MISRA-C:2004
au-misra3.lnt         for MISRA-C:2012
au-misra-cpp.lnt      for MISRA-C++:
  2008
```

into the `C:/lint/lnt` directory. Then add to `iar-co-***[-V*].lnt`, the line

```
C:\lint\lnt\au-misra1.lnt
```

For additional advice on MISRA-C, Dan Saks' and Scott Meyers' rules, please see the section on *MISRA-C and other Coding Guidelines* (p. 10 – below.)

22 You need to set the global warning level, which determines the severity of errors that generate warning messages. See the section "Using PC-lint" for advice on this setting and on modifying it. To set the default level (w3) the line

```
-w3
```

should be placed after the modifiers mentioned in para 20 and any of the additional files mentioned in

para 21, but before the lines for the extended defines described in para 23, below.

23 The final working line is to include the pre-defined headers we generated earlier.

```
 EWARM extended defines
-header(C:\lint\lnt\iar-ewarm-
   predefined_macros_for_PC-lint.h)
```

Save the `iar-co-***[-V*].lnt` file.

Test the configuration

24 Load a project into EWARM and open a C file to the editor. Click anywhere in the editor window to ensure focus. The "tab" at the top of the edit window should be highlighted. Then go to the tools menu and select `Lint File (ARM)` or whatever you have called the entry.

25 Running the command should give you an output in the compiler output window, similar to the one below. Clicking on the hyperlink (the light blue underlined part, giving file name, line and position of error) will take you to that error in the edit window.

26 The most common errors at this point are messages saying that PC-lint cannot open a file, either one of its own or a header file. This indicates that either the file is missing or you have not correctly set up the `-i` search paths in the compiler specific file `iar-co-***[-V*].lnt`.

27 If the file name etc. is not a hyperlink but plain text you have probably lost the ***Control-A*** characters in `iar-ew***.lnt`. The easiest way of fixing this is to copy the original `iar-ew***.lnt` from the zip file to `c:\lint\lnt` and try again. If that does not work see the IAR app note *Browse Your Application* (Sporrong 2011).

PC-lint set-up for Multiple File Analysis

For multiple file analysis all that is required after setting up the single file analysis is a second entry in the tools menu. The `$FILE _ PATH$` is replaced with `$PROJ _ DIR$\*.c` giving the argument line

```
-u C:\lint\lnt\iar-ewarm.lnt $PROJ_
   DIR$\*.c
```

The project variable `$PROJ _ DIR$` gives the main project directory to which you need to add `\*.C` so that all the C files are picked up.

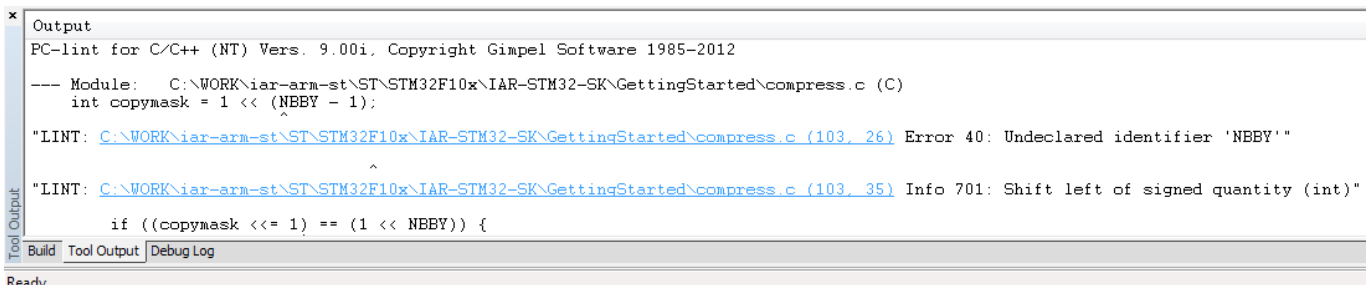
If there are multiple directories in the project you wish to check simultaneously, one option is to place all the directories in a file such as `project-dir.lnt` and call that on the argument line as

```
-u C:\lint\iar-ewarm.lnt  c:\lint\lnt\
   project-dir.lnt
```

Test in the same manner as for the single file set up. Normally the problems are caused by incorrect or missing paths to files.

Using PC-lint: Don't shoot the messenger

PC-lint is very flexible with, as you have discovered an almost infinite combination of switches and options. While, as you have seen, most are pre-configured you can, over time, tune PC-lint to your company coding standard and specific project requirements. Here we have some tips picked up over many years of static analysis using PC-lint. It is worth reading this section, along with the PC-lint manual, and then spending some time configuring PC-lint to set up a policy for static analysis. It will pay dividends in the end and indeed even improve the C language.




```
12
13    // Build Window output format for IAR Embedded Workbench
14    -format="LINT: SOH<SRCREF line=%l file="%f"SOH>%f (%l, %C)SOH</SRCREF SOH> %t %n: %m"
```

General

The first time you use PC-lint, particularly on legacy code you will get large numbers of warnings. Don't be disheartened and most important don't let this stop you using PC-lint. **Don't shoot the messenger!** PC-lint is finding real problems and looking the other way will not solve the problem. It will cause bigger problems in test and debug, if you are lucky. If you are unlucky, these problems will not be found in the test phase, and will go out into the field where they can be infinitely more expensive to fix. Products failing in use can also cause damage to a company and its reputation. For a history of this see *In Search of Stupidity* (Chapman 2003).

Firstly, as with a compiler, an error occurring early in a file can cause hundreds of other warnings though the file. For example a missing include for stdio.h will produce multiple errors every time printf is called. Fix the first error and many more related to it will also disappear. Adjusting the PC-lint switches, carefully, can do the same. So, after what can be a frightening initial pass, the number of errors and warnings in later passes should diminish rapidly.

Warning levels

There are two sets of warning levels, one in the compiler and one in PC-lint. The compiler is a translator that will compile syntactically legal code. The compiler warning level should always be set to its highest warning level and there should be no compiler errors or warnings in the code. The compiler is producing the binary that will actually run on the target. If the compiler issues a warning it means it has a problem, for whatever reason, in producing the binary: the code that generated these warnings should be corrected.

It must be stressed that having a clean compile, on the highest warning setting, does not mean there are no problems in the code. This is NOT a fault of the compiler but the design of the C language, which is very happy for you to shoot yourself in the foot, both feet come to that!

Unlike the compiler, PC-lint is not a translator but an analyser; it looks at semantics as well as syntax and it analyses the source in a very different way to the compiler translating it. PC-lint warning levels can be adjusted, the default is `-w3`, set in file `iar-co-***[-V*].lnt`. However, on legacy code and for the first pass on a new project, you should start at `-w0`. The `-w0` level tests the PC-lint environment, and provides warnings only of fatal errors. These warnings are about things such as not finding files (such as library files and include files). If you find code errors at this point the code will probably not compile.

On new code, after the first pass to check the set up, you should move the error warning level from `-w0` to `-w3`, but for legacy code you should move to `-w1`. This level is where PC-lint generates error messages, some of which will be similar to compiler error messages. These errors should all be fixed by correcting the code. There may be a very few errors that require amending the PC-lint configuration files - things like compiler specific keywords and extensions - but IAR should have done this already with the four configuration files. So think very carefully before amending the PC-lint files. It should be noted that in practice there are no fully ISO C compliant cross compilers for the embedded world - see the comments on `options.lnt` later in this section.

When all the `-w1` errors have been reconciled move on to level `-w2`. Again, PC-lint signals errors and warnings, but these will be much stronger than anything the compiler is likely to pick up. These messages will include semantic problems of, as Dennis Ritchie put it, "legal but dubious constructs" (Ritchie 1993). They will also be downright dangerous constructs.

From error warning level `-w2` move to `-w3`, which is the PC-lint default setting: this level will normally find many things in the source code. Most warnings will involve correcting the code but in some cases you may determine that the code is correct, despite the warnings. In these, few, cases you should suppress the PC-lint message.

Managing messages: options

As mentioned, you may need to suppress or modify some PC-lint messages due to project requirements and deviations on things like MISRA rules or hardware requirements.

There are various ways of doing this. The obvious method is using configuration files. DO NOT EDIT the IAR compiler configuration files, but instead create a new file. The reason is that should/when IAR updates the file you can swap it without having to re-edit it. Traditionally PC-lint user configuration options are

in a file called `options.lnt`, and there is no reason not to use this convention. This should be written in a text editor, using the same syntax as the IAR compiler configuration files, and placed in `c:\lint\lin` with the other PC-lint configuration files and called from the `iar-co-***[-V*].lnt` file just above the `-w3` instruction.

PC-lint switches are read and acted upon in order of processing. So using an options file as the last file called, before the pre-defined macros file and the -w option in iar-co-***[-V*].lnt, will mean these switches override previous switches. Also, should IAR update their files, replacing them will not require editing any files.

There is another method. The source code itself can be instrumented by PC-lint comments, which do not affect the compilation. This modifies messages locally, for just a few lines in a file or a function, rather than globally. There should also be additional comments to explain why the particular messages are being suppressed which should contain a reference to the permitted deviations in the coding standard in use or the deviation in the project specific documentation.

If we want to turn off message 521 and then reinstate it, the comments would be:

```
/*lint    -e521 */
Statements …
/*lint     +e521 */
```

Note: `lint` is lowercase and there is no space between the comment marker and `lint`.

Multiple rules can be suppressed or reset on a single line and `//` style comments can also be used.

MISRA-C and Other Coding Guidelines

MISRA-C can be a difficult subject. Bear in mind, MISRA-C is Engineering Guidance not a Bloody Religion. It is practically impossible to have 100 % MISRA-C adherence with no deviations. More to the point it is undesirable in the same way it is usually undesirable to have 100 % ISO C compliant C code with no extensions or restrictions on cross compilers for embedded systems. Since you will have deviations from the rules, you need to have a MISRA-C compliance matrix. This will document which MISRA-C rules you are checking, where you are checking them and which rules you have determined will have deviations. (An example of a MISRA compliance matrix can be found at http://library.phaedsys.com)

The MISRA-C `lnt` file should be included in `iar-co-***[-V*].lnt` after the IAR compiler specific switches and before the pre-defined macros file the `-w` and `options.lnt` file. There are MISRA-C `lnt` files for each version of MISRA-C, but you should only include one.

Unless you are deviating the majority of MISRA rules, I would suggest that any global modifications are put in the options.lnt file. Local modifications can be placed in the source code in the PC-lint comments, with a comment to document the specific deviation and reference to the project deviation document.

The same applies to other guides such as Scott Meyers and Dan Saks. Since these guides were written things such as the C or C++ standards, the compilers, the target hardware etc have moved on and changed and asynchronously to other things also changing. So first look at the date in the copy of the guide to give you a context. Then use it as a guide not a fixed rule. However this does require that you actually understand what and why the rule is trying to do.

Strong typing in C

One of the big problems with C is that it is not strongly typed. Whilst PC-lint will complain if you put a larger type e.g. an `int` or a `long` into a `char`, due to loss of data and mixing signed and unsigned types, that is only part of the problem. When you have two types with the same underlying type, neither the compiler or PC-lint will complain. This can be remedied, in PC-lint at least with the instruction

```
-strong(flags !typename )
```

This will ensure specified `typedeffs` have strong typing in PC-lint (but not the compiler). There is a whole chapter on this in the PC-lint manual, it is well worth reading. This is a lot easier to implement on new projects than on legacy projects!

Indentation

One seemingly innocuous message, that can be easily overlooked, is a warning on unexpected indentation. This message flags two possible problems, one of which is obvious, the other less so.

The obvious problem is in this case:

```
interlock = ON;
…
if(TRUE == stop)
   flag = ON;
   interlock = OFF;
…

if(OFF == interlock)
   open_doors();
else
   apply_brakes();
```

PC-lint will give a warning when { } is missing in single line `if` statements (also for `do`, while, for and similar constructs) and a second line is added after `flag = ON;`. In this example, the line `interlock = OFF;` should have been part of the `if` block. The author has seen this problem many times, and there are many reported cases, including this example from a rapid transit system.

The other, less obvious, occasion is where this error is apparently a false positive. The code below appears to be correct.

```
if(TRUE == stop)
{
   flag = ON;
   interlock = OFF;
   flag2 = set;
}
if(OFF == interlock)
{
   open_doors();
}else{
   apply_brakes();
}
```

Yet it generates the same "indenting" error after `flag = ON;` This usually caused by a non-visible or printable character or, on occasions, a line that extends off the side of the screen, sometimes by 100's of columns of blank space. This is common where code has been cut and pasted from other sources and the code did not have a CR/LF. Linefeeds, carriage returns and end of file markers differ between many systems and, as they are non-visible characters, you won't see them on screen.

The solution is to manually re-create the line, after deleting the line from, and including, the last visible character on the line above to the first character of the line below, i.e.:

```
ON;
   interlock = OFF;
   fl
```

With code snippets being cut and pasted between DOS/Windows/Linux/Unix/Web and other sources, it is best to listen to PC-lint on this - even when you can't see the problem. You have no idea what the compiler may do with the tokens you can't see.

The last word

DO NOT SHOOT THE MESSENGER. The messages from PC-lint indicate problems that will surface at some point. The sooner these warnings are fixed, the less they cost in time and resources. Remember that time also costs money. The costs rise if test and debug takes so long you miss release dates. The ultimate cost of unreliable software in the market can be the future of a company and your job. Read *In Search of Stupidity* (Chapman 2003)

After the initial PC-lint set-up and use: read the PC-lint manual for ways of improving the static analysis for your project. And finally: Run PC-lint often, far more often than you run the compiler.

References

Chapman, M. R. (2003). **In Search of Stupidity: Over 20 Years of High-Tech Marketing Disasters,** Apress.

Johnson, S. C. (1979) **Lint, a Program Checker.** Unix Programmer's Manual, Seventh Edition 2B,

Ritchie, D. M. (1993). **The Development of the C Language**. ACM: 16.

Sporrong, T. (2011) **Browse your application.** 6

Integrating PC-lint into IAR Embedded Workbench

First edition August 2005
Second Edition January 2013

Phaedrus Systems Library

The Phaedrus SystemsLibrary 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.