# How to wield PC Lint

## Abstract

Using PC Lint has become standard in many development efforts using C or C++. Several project leads I know insist on its usage, since they know the potential benefits. However, having been intimate to many C/C++ projects in the last several years, the sophistication of Lint usage differs greatly. In many projects, he PC Lint user's manual is rarely taken to hand, the plethora of options available are mostly unknown, and several error messages are not well understood.
The short chapter "Living with Lint" in the user's manual doesn't really solve the problem if the user isn't sufficiently literate with lint options.
So, over the years I have collected some "best practices", to be presented here. On many occasions I have been pushed into projects without any previous Lint contact. Digging through the massive amounts of messages isn't easy, but do-able if you take a structured approach.
Finally, I also give some hints as to how to use the various Lint options to their fullest.
Yes, everything written here can also be found in the manual. But the information is scattered. Here I present you my roadmap, single-stepping through the various acts of cleaning your code.
One more remark: To limit the scope and size of this document somewhat, I'll concentrate on pure C and leave out a discussion for C++. Parts of the document are generally applicable, and strategies will be the same, but especially a discussion of specific error-types leaves out all C++ discussions.

## Structuring Lint options

One of the first pitfalls lies in the way to utilise the various Lint options. Especially when you encounter a project where PC Lint has been used all along, you will see code littered with so called "Lint Comments", special C/C++ comments intended for PC Lint, containing Lint options.
Now, PC Lint has several places where it reads its currently valid options:

- From the command line
- From special Lint option files (usually called **.lnt*)
- From within the source code modules, in Lint comments

OK, you may also create a special include file, containing definitions relevant only to Lint, and use a special option `-header(…)` to indicate this to the program. Until today, I have never felt the need, but your style may be different.
So, most installed projects using PC Lint I have seen use some options on the command line, and then litter their code with lint comments. This will work as long as you keep the Lint comments in sync with your code. And be honest: You always do that, don't you?

### *Using option files*

Here's my way. I use three separate option files for PC Lint, called *loptions.lnt*, *project.lnt*, and *temp.lnt*. So my command line looks as follows:

```
lint-nt -b loptions.lnt project.lnt temp.lnt <sourcefiles>
```

The `-b` option only works when given on the command line, therefore I need it there, then the three option files, after which come the source files. I try to avoid unit-checkout ('linting' single files in isolation) because of the reduced error output, but if you wish, use `-u` for that.
And if you're into Lint-Object-Modules, you'll probably not need this introduction in the first place.

### *loptions.lnt*

The file *loptions.lnt* has been more or less the same for me since I adopted this structure. It contains just a few options, and those only of a general nature:

- Message presentation: How shall Lint messages be formatted
- Error output: What value shall Lint return to the operating system after processing

- How to handle libraries: Shall we lint them together with the code, or better ignore them
- How many passes do we need
- Directory structure: Where are the rest of the configuration files to be found (*project.lnt*, *temp.lnt*)

That's it. For being so general, I can reuse the file for any new project.

## *project.lnt*

Here's where the action is. Whatever your project needs as far as project-global Lint options are concerned goes into this file: Tab-size, library headers, project conventions (is `char` signed or unsigned? Are trigraphs allowed? Do we allow early modifiers? May we use enumerations and integers interchangeably?)
This is the file we'll be discussing most, but not here.

## *temp.lnt*

This file is empty. Or at least it should be. In any version control system it shall be checked-in containing just comments; but in my workspace, it may contain many error inhibition options. This file I use to reduce the massive number of error messages to a number I can handle. And as soon as I have solved most issues, I reduce the contents of this file, and work on the next chunk.

Is it possible to use less option files? Yes, it is. If you want, you may put all three parts into one file. Having this file split in 'responsibilities' makes sense, but putting it all in one file doesn't detriment your possibilities.
But having them in three files enhances reusability.
Is it useful to have more than three option files? This depends on your project structure. If you work in a department developing for six different processor platforms (e.g. x86, IA64, ARM, etc), using four different compilers (Microsoft 6,x, Microsoft .NET, GNU GCC 3.x/4.x, etc.), it may make sense for you to have one global *options.lnt*, one compiler-specific *comp.lnt*, one platform-specific *platform.lnt* and one project-specific *project.lnt*, as well as *temp.lnt*. Look at your environment, and make an informed decision, based on reusability.

## Where to put compiler specifics

I guess this is a matter of taste. In my environment, the compiler-specific options (like the size of '`int`', '`long long`', '`short`', alignment, specific macros the compiler defines for you, etc.) are put in a special file and included in *project.lnt*. You may also add a fourth file to the command line (before *project.lnt*), or create a wrapper file, say *wrap.lnt*, including all three or four, reducing the length of your command line. As long as the ordering is kept intact (offering the possibility of overriding more global settings at every subsequent level), and each file discussed is physically separate for reuse-reasons, the form of execution is of no importance.
Be aware that Gimpel offers a whole slew of compiler specific option files, called *co-<comp-abbr>.lnt*, installed with the product or offered on their website. Be sure to check there regularly, since these files get updated occasionally. They present a fair starting point. However, being into embedded development, I have regularly had a compiler not supported "out-of-the-box", so I wrote such a support file by myself. You may take the offered option files as samples, of course.

## Accommodating individual requirements

Especially the error formatting requirements differ from developer to developer. At least in teams where the environment is not a matter of dictatorship, every code editor is a little different. And since developers are lazy by nature (at least I am), they will want to handle Lint messages like compiler errors/warnings, and be able to jump wherever is pointed by pressing a key.
So, in some projects you'll have the need to override some settings on an individual basis, without compromising your Makefile-integrity. You'll have to find your own way, but if I have control over the Makefiles, I usually add an option like "if you find a file called *individual.lnt* at such-and-such position in the file system, add that filename between *temp.lnt* and the

source files in the command line for PC Lint of the Makefile". Since the Lint option files do not have any possibility for conditionally including options, this seems a viable way to go.

## Source filenames on the command line

Those of you working with UNIX (or Cygwin, or…) are lucky for having a very long command line (usually 4096 characters or larger). Those of us working with Windows (or even MS-DOS, yes, they're still among us!) usually have to stay below 256 or even 128 characters per command. So, unless you're into very small projects, the command line will be too small (sometimes even on UNIX!) for listing all relevant source files on one command line.
I usually direct my Makefile to create or update a file called *files.lnt* with all the C/C++ source file names making up the final executable. This would be the fourth (or fifth) filename to be included on the Lint command line.

## Where to send the output

One more issue has to be addressed before we can present the full command line. Lint is a tool, apparently made by programmers for programmers. It doesn't offer a flashy GUI, where you can point-and-click through your messages. It used to be a DOS program (I do not know, whether a UNIX variant has been available in those days), and as far as I know it will still run on MS-DOS in the current version. So it was and is restricted to two output channels, *stdout* and *stderr* (by default, your screen is targeted). The screen is not a very efficient recipient of information, especially when the amount is as large as Lint will happily make it. Lint-runs to the screen easily take 20 times as long compared to redirecting the output to a file, when running a command line under Windows (*CMD.EXE*).
Most command shells, including the Windows DOS-box, offer redirection possibilities. If you like, use those. However, command shells are all different, both in behaviour and in syntax. Therefore, I use the option offered by Lint: `-os(lintout.out)`.
Where to put this option? *loptions.lnt* would present itself, since the (single) output filename will probably remain the same over projects. But if the project (-manager) dictates such a name, *projects.lnt* would be perfect. *temp.lnt* shall remain empty.
I put this option on the command line, for one reason: When integrating Lint in your Makefile, you'll probably use a make-target called "lint" or "lint_all". If, as far as the make program is concerned, the error file can be viewed as another make-target, make can decide for itself, based on file timestamps, whether a new Lint run is necessary. In that case, the filename will be integrated in the Makefile, so it would be natural to be able to include it on the Lint command line. Even name changes between projects can easily be accommodated this way.
So, the command line to be used would look like this:

```
lint-nt -b -os(lintout.out) loptions.lnt project.lnt temp.lnt
individual.lnt … files.lnt
```

(The dieresis '…' indicates that this command line is just one, broken in two for layout reasons).
One word of warning: The `-os` option needs to be given before the first output-generating option is presented on the command line (directly or indirectly), but, especially in a Makefile, you would not want one of the options files to override the value presented on the command line. Therefore, the position of the `-os` option has been chosen for a reason.
See also the use of `+flm` in *loptions.lnt*.

# *My global options*

The file *loptions.lnt* hasn't got much to do with the task at hand, but just to get a starting point I'll present the options I usually use, with a few remarks.

```
-wlib(1)     // Only report errors from the libraries

// Output: One line, file info always, use full path names
-hF1
+ffn
-"format=%(\q%f\q %l %C%) %t %n: %m"

// Do not break lines
-width(0,0)

// And make sure no foreign includes change the format
// or the output option
+flm

// Warn when trigraphs are used
-ftg

// Include files within library includes without full path
+fdi

// Don't stop make because of warnings
-zero(99)

// Make sure to make two passes (for better error messages)
-passes(2)

// Include directory where other option files *.lnt are located
-iC:/Lint90/lnt

// Don't let -e<nnnn> options bleed to other source files
-restore_at_end

// Produce a summary of all produced messages
-summary()
```

Let's discuss these one by one. Details on the various options can be found in the Lint user's manual, so I will not cover everything here.

- `-wlib(1)` reduces the flood of errors and warnings from libraries I have no control over. Since Lint is very powerful in respect to determining what is and what is not a library, this is just a global precaution.
  Start with '1' (one) for a good reason: Lint will report any errors it finds, no warnings, info's or notes. But the errors, if any, will alert you to some very valuable findings, such as missing types, incomplete definitions or declarations and the like. This will alert you to some configuration problems you might experience, like (compiler-) predefined macros that Lint knows nothing about, or wrong ordering of include files, making Lint find the wrong include file first.
  As soon as your project is in good shape, use *`temp.lnt`* to experiment with different settings (2, or even 3). If you're confident about the new settings, incorporate them in *`project.lnt`* (don't use *`loptions.lnt`*, since you wouldn't want a new project to start out at warning level 3 for the libraries!).
- `-hF1`, `+ffn` formats the Lint messages a certain way. I don't need any source code context in my warning messages if I can jump to the exact location in my editor by simply pressing a key; I don't need an indicator as to in which column Lint recognises a potential problem, if my editor can not only jump to the indicated line, but also to the indicated column (note the difference between %C and %c, counting columns starting at zero or one, depending on your editor's requirements!); I need the full pathname indicated in the message, so my editor can find the file no matter what its current directory happens to be.
- `-"format=%(\q%f\q %l %C%) %t %n: %m"` this is just the format I happen to like. Note the quotes; they are needed because the option string contains spaces. And note the '`\q`' around the filename '`%f`', they make Lint emit quotes around the filenames in all messages it produces, to allow source paths also containing spaces.
- `-width(0,0)` For better processing I like one line per message, no matter how long that line gets (in C++ lines can get very long indeed!).

- `+flm` locks-in the so given message format. It is up to you whether or not to use such an option, but I have seen different message format options located in Gimpel-provided compiler option files (like *co-msc90.lnt*). In addition, this option locks-in the `-os` option for indicating output redirection, so you'd better place that option (on the command line, if you will) before this one.
- `-ftg` disables tri-graphs, or better: It makes Lint warn you when tri-graphs are used in your code. I see no need to use tri-graphs in my code, and I would like Lint to warn me when a character sequence is used which is interpreted differently from what I have written. YMMV.
- `+fdi` depending on how projects are split into directories, and how the compiler includes like *stdio.h* are structured, include files refer to sibling includes without specifying a path. So, by default, I'd like to adopt that convention. If need be, I can override the setting in the compiler- or project-specific options. On UNIX (FlexeLint), this is the default.
- `-zero(99)` when starting with a new Lint project I definitely want this option gone (using `-zero` in *project.lnt*), otherwise my Makefile will always terminate prematurely. However, for stable projects, regularly linted, it makes sense to abort make when "real" errors are found using Lint (probably your compiler would also complain).
- `-passes(2)` again, when starting with a new Lint project I'd want this option reduced to one pass only. Two or more passes only makes sense when the project is at least moderately cleaned. In general, however, two or more passes makes sense, since Lint can perform better quasi-dynamic checks using value-propagation techniques.
- `-iC:/Lint90/lnt` because I use my own copy of PC Lint on my laptop, where Lint is always installed in the same directory `;-)`, I indicate this directory here. Your environment might benefit more from such an option in *project.lnt*, and it will probably use a different pathname. Note the use of forward slashes in the directory name. Especially when using relative pathnames, option files can be shared between Flexe-Lint ("PC Lint for Unix") and PC Lint, between UNIX and Windows.
- `-restore_at_end` provides a kind of sandbox concept for each source file when you present multiple files to Lint in one run. Without this option, Lint comments like `/*lint -e8?? -w1 */` will be valid not only for the file containing this comment, but also for all subsequent files. It was added at my request in version 8.00v, and now officially part of version 9.0.
- `-summary()` provides a handy way to look for certain types of warnings. Lint will produce one line indicating the type of warning for each type generated. Especially with really large output, looking at the summary may be faster than searching the whole file.

# An existing project without previous Lint exposure

We have a basic *1options.lnt*, an empty *project.lnt* (possibly only including the correct compiler-options file), and an empty *temp.lnt*. Now let's get down to work.

## *Project defines*

Most projects specify at least one or two macros on the compiler command line, if only something like `-DTARGET=RELEASE`. Such options (usually contained in Makefiles or command scripts) have to be provided for Lint also.
If you're using a Makefile to start Lint, provide the same options in the same way as for the compiler. But if you want you may also specify them in the *project.lnt* file using the same syntax. If there are more macros defined like this, provide them all, or use common sense to differentiate which ones are really necessary.
One issue you might encounter: Most compilers offer an initial set of predefined macros. Unless you are lucky to find a Gimpel-provided *co-<name>.lnt* file containing the settings for exactly your compiler, you will have to provide these to Lint. Since Lint caters for all known compilers and then some, Lint cannot build-in a set of predefined macros. But if the include-files provided with your compiler (and believe me, Lint had better use exactly those!) assume the definition of those built-ins, you will have to educate Lint. Exactly how, is up to you: Browse the manual, ask some colleagues, or, new for Lint 9.0, use the `-scavenge` option. The

`-scavenge` option has been added to Lint to aid in the process of finding out the implicit compiler defines. The process to do so is a multi-step process that may need repetitions whenever the compiler options are changed. The details are described in the manual, so I will simply point you there.
And if you are lucky, your front-end is called 'EDG' version 3.9 or later (compilers like Intel, IAR, Green Hills, TI, Comeau and others) you may look for an option `--list_macros` which enables you to list all predefined macros. If you use the GNU compiler tool chain, the option `-dM` may do the same. Check the respective manual.
One hint: Until you have cleaned out most of the messages, stick to only one build-type for Lint (`DEBUG` or `RELEASE`), best is to use the same configuration as is regularly used for development, usually `DEBUG`. Later in the project you may choose to perform Lint for some more or all of the configurations.

## *Include directories*

If we now run Lint, it will probably[1] abort because of include-files it cannot find. So, let it run, examine the last few lines of output, and locate the include-file in your source directory or in your compiler's library includes. Then add the option `-i<directory>` to your options. Don't even look at the many warning messages; they will only later become relevant.
If you use a specific file for compiler options, be sure to add the corresponding include options to that file, if the path belongs to the compiler includes; otherwise, add it to the ***project.lnt*** file.
For details as to how Lint searches for and finds include-files – including the use of the environment variable `INCLUDE` – refer to the Lint User's Manual. But one issue is still relevant here. Sometimes you will get error messages from Lint concerning include files it cannot find, but then you cannot see why. Consider this scenario:
If your C file includes an file in another directory, like `#include "abc/def.h"`, and the directory ***abc*** is not in the regular 'include-path', you may run into trouble if, say, ***def.h*** includes ***gkl.h***, located in the same directory ***abc***, without indicating the full path, `#include "gkl.h"` instead of `#include "abc/gkl.h"`. If you've taken my ***loptions.lnt*** file without adaptations, all's fine. If not, please check if you have included the option `+fdi`[2] at least in ***project.lnt***, since that option teaches Lint to look in the including file's directory for further includes.
Repeat this procedure until Lint completes its full run. In that case, the last few lines of output do not indicate a missing include file. Verify that all include paths are found by searching backwards for the string "Global Wrap-up". If it is found, Lint has produced a full output, and we're set to start analysing.

## *Library includes*

OK, if you really have library include files (from your compiler, or from bought libraries like operating system, network management or math packages), be sure to list them as such. Use the option `+libdir(<directory name>)` if all the include files in that directory are to be considered library headers, or use `+libh(<filename>)` for individual files. Especially when starting out with Lint, run the program immediately afterwards and check that the appropriate messages have effectively disappeared; you wouldn't be the first to use wrong path- or filenames and not notice until far too late.
Most such options belong in the file ***project.lnt***; some might belong in your compiler options file, if you're using such a file.

## *Layout and indentation*

My personal opinion is never to use tabs for indenting blocks of code. Modern development editors can be configured that way, at least for source files. But usually 'some colleagues' can't get their settings right, so tabs seem to be a fact of living.
If, in going through your messages, you encounter many messages concerning an unexpected indentation (search for the keyword "indentation"), there may exist a mismatch

[1] If you find Lint stopping because of a `#error` directive, find the reason (usually a missing or imprecise macro definition) and remedy that.
[2] This option is active by default for Unix/FlexeLint installations.

between your ideas of a tab's size calculated in spaces, and Lint's. In such a case, check your coding guidelines, and provide the option `-t2` in your ***project.lnt*** options file (replace the '2' with the number of spaces equivalent to a tab in your project). Rerun Lint, and check the results (not the details, just the total number of messages as compared to the previous situation); this will give you a very rough idea as to the amount of tabs still in use, and it will flag the layout inconsistencies based on correct assumptions.
One other layout related issue you might encounter concerns the `typedef` names of structures/unions/enums: If they are not on the same line as the closing brace, consider changing the layout (recommended) or switch off message 659 by using option `-e659` in your project options file.

## Signed or unsigned characters

Never mind the messages, neither by number nor by content. Usually, characters are seen as unsigned. Now, since the standard doesn't require one or the other, most compilers as well as Lint can handle all three cases: "Characters are signed", "characters are unsigned", and "characters are neither signed nor unsigned". In the last case, `char`, `unsigned char`, and `signed char` are all three considered different types. Be sure to let Lint know which of the three you intend for your project, for instance by specifying `+fcu` in your project options file. Especially when specifying `-fcu`, be sure to specify so after including your compiler's options, since many compiler's option files as provided by Gimpel specify `+fcu`, possibly overruling your project's setting. On using C++, consider the `+fdc`-option to distinguish all three forms (overloading!).
If you never use the 'plain' type `char`, and you only explicitly use either `unsigned char` or `signed char`, this will not make any difference to you. Enable warning `+e971` if you want to find all cases where `char` has been used without `signed` or `unsigned`.

## Using enumerations

Strictly taken, enumerations are separate types with a limited number of valid values. The C standard, however, doesn't enforce this separation, and many C compilers consider enums to be equivalent to integers. If you want PC Lint to adopt this 'loose integer model', specify the option `+fie` in your project option file. In that case, enums can be assigned to integers and vice-versa; you may loop through a set of enum values, etc. But you may also assign basically invalid values to enum-type variables without Lint complaining about it, so consider your options carefully.

## Using bit fields

If you're into embedded development like me, your code probably has bit fields all over. A portable use of bit fields uses `unsigned` to declare the bit fields, like

```
struct B { unsigned a:4; };
```

The standard will also allow `signed` or `int` for declarations, considering the bit field to be signed. However, some compilers and/or developers will still treat the bit fields as unsigned. Specify the option `+fbu` if your environment is like that.
Some environments are even more relaxed, allowing type declarations like `char`, `long` or even enumerations to declare bit fields:

```
typedef enum { false, true } eBool;
typedef struct BF
{
  char  ec:2;
  long  el:2;
  eBool eb:1;
} BF_t;
```

For PC Lint to accept this as a valid bit field declaration, you'll have to switch off message 46 (to be able to use other types than signed or unsigned integers) using option `-e46`, and if using enumerations to declare bit fields is common in your source code, also specify option `+feb` in your project option file.

## *Pre-processor expressions*

Some C compilers allow the use of enums and constant variables in pre-processor expressions, like:

```
const int cVar = 5;
#if cVar < 6
  ...
#endif
```

If you want PC Lint to allow such (non-standard) use of the pre-processor, include the option `+fep` in your compiler options file or in *`project.lnt`*, whichever you prefer.

## *Inline assembler*

The world of embedded systems uses inline assembler frequently, and PC Lint is able to handle most variants I have seen to date. Let's consider three common variants.
If inline assembler instructions are framed with pragmas like in

```
#pragma asm
  mov r1,r2
#pragma endasm
```

For these assembler instructions to be ignored by Lint you need to switch off checking using the option `+pragma(asm,off)`, and re-enable checking using the option `+pragma(endasm,on)`.
For most PC-based compilers it might be sufficient to enable the Lint-specific special keyword `asm` by providing the option `+rw(asm)`. However, if an inline assembler routine is fitted with a C-like prototype, this will not be sufficient. Consider:

```
asm int iGetCurrentSp(void) { mov sp,r10 }
```

Here we need Lint to ignore the function body. For that we enable not only the Lint-keyword `asm`, but also the 'gobbler' keyword `_ignore_init`. Specify the option `+rw(asm,_ignore_init)`, or just `_ignore_init` if you already enabled `asm` before. Define a Lint-specific macro replacement using the option `-dasm=_ignore_init`. Lint will ignore the 'initialisation', which in case of a function definition means the function body.
And, finally, consider the variant `__asm("mov sp,r10");` embedded within normal C code. Since it clearly cannot be considered an initialisation, not even in the broader sense like before, we'll again need a new construct. For this to work, specify another 'gobbler' keyword `_to_brackets` using the option `+rw(_to_brackets)`, and let Lint know to ignore the brackets (everything in between included) after `__asm` using the macro replacement `-d__asm=_to_brackets`.
I've seen all three variants supported in one compiler, and, thanks to various external libraries, also used simultaneously.
If your environment supports and uses different variants, you can experiment with the techniques described here. Also consider the existence of even more 'gobbler' keywords (to be enabled with the `+rw(...)` option first) like `_gobble`, `_to_semi`, and `_to_eol`.

## *A first run*

If you're in a situation similar to mine, you need to produce first results quickly. So, run Lint on the sources, and be sure to redirect the output to a file. I've seen initial Lint-runs take more than an hour on a moderately complex project (less than 400 source files) when the output scrolls through a DOS-window on a fast Windows XP machine. With redirected output, the process took only a few minutes, producing more than 100,000 lines of output.
How to handle a file with thousands of lines of messages? It is totally impractical to go through it line-by-line, that would take a year. So use a decent editor capable of handling large files efficiently, and use the search feature.
From my experience, there are a few searches that will get you up to speed and find the most prominent mistakes quickly. Of course, these string searches could also be replaced by searches for specific error numbers. However, PC Lint evolves and produces new error messages in most every new version. Using the words instead of the unique numbers enlarges the chance to also catch new errors.
Please be aware: The following search ideas are intended as a guideline. Feel free to vary your searches as you think fit. Also, some searches may overlap, so sometimes they might direct you to the same messages.

Important: Never disable errors with a number below 400, unless the manual says you can. Such errors usually have a real cause (configuration problems?) and need to be resolved, not suppressed. If you suppress them, changes are that subsequent errors appear because of these errors, but you'll not be able to find the root cause.

## Memory corruption and other quickly found errors

The most obvious serious errors Lint can find are broadly related to memory corruption. The keywords we look for are "division by 0", "out-of-bounds", "beyond array", "data overrun", "null pointer", "suspicious use of", "memory leak", and "custodial pointer". Be sure to switch off case-sensitive and whole-words-only searching.
This way we're looking for presumed access outside of array boundaries, null pointer access, `malloc`/`free` errors and memory leaks and a few curious constructs. All messages thus found indicate pieces of code worth looking at in some detail. Again, do not dismiss a PC Lint observation too lightly!
The messages thus found may indicate a real error, or they may indicate a misinterpretation by PC Lint. At this stage it is not useful to do more than just glance over the area indicated. If the source code has never been submitted to Lint analysis, chances are quite good to immediately find some real problems in your source code. Report or correct these immediately, and continue looking. Do not add Lint comments or specific options for all those locations where you cannot find a real error, and simply want to silence Lint. We're not that far along.

# *The next steps*

From here onwards, we have two possibilities: Start cleaning the header files, or concentrate on the implementation files. Since I'll discuss the cleaning of the header files extensively, let me briefly explain the other option.
Ignoring messages from all include files is quite easy. Using the option `+libh(*.h)` effectively all include files are declared library header files, and because of the `-wlib(1)` option they are largely ignored. Their contents are not ignored, however, and that presents one of the potential problems with this way of working. It is quite possible to find messages for problematic points in the C files originating from an ignored problem in some include file. That is the reason I personally start working on the header files first.

## Classifying messages

Take a look in your output file. For every message you'll have to decide – at some point in time – whether you want to change the code to conform to Lint's expectations, or if you want to tell Lint to ignore this issue. If you want to change the code, go ahead and do it. Since I do not know a thing about your code, you're on your own there.
I'll here concentrate on telling Lint that everything is OK, while retaining as much error detection capability as possible.
Silencing Lint is easy. I've seen a developer starting all his C-files with the option `-e*`, effectively disabling Lint for all his modules and include files. Until I found his Lint-comments, he was highly thought of. The difficulty lies in retaining Lint's capabilities to the fullest possible.
To illustrate the advantages and disadvantages of some method, I'll discuss a few common situations in detail.

## Macros TRUE and FALSE

Many environments using C89 have defined macros called `TRUE` and `FALSE`. `FALSE` is defined as 0 (zero), and presents no problem. If `TRUE` is defined as "`!FALSE`", Lint will complain because of a 'constant value boolean' (message number 506), which is basically correct but still intended here. And since these macros tend to be used quite often (including other variants and macros), a plethora of messages is produced.
Several options exist:

- Disable message 506 globally by adding the option `-e506` in your project options file. This alternative has the grave disadvantage that relevant messages from other contexts are suppressed as well.

- Use a Lint-comment within the include file where the macros are defined, disabling message 506 after the definition, like `/*lint -e506 macro TRUE, intentional */`. OK, this is a little better, but not much. Potentially many files will include this header file, and the slight advantage over the previous option is almost gone.
- Use single-line suppression like `/*lint !e506 macro TRUE */` everywhere the macro is used. Now this works almost perfectly (except for similar errors in the same line), but it is extremely tedious to do, and you won't remember it at all times.
- Use the single-line suppression comment within the definition of the macro, like in:

```
#define TRUE !FALSE /*lint !e506 */
```

  This would seem to work, since Lint doesn't quite discard all comments, only the non-Lint comments. However, this sort of comment is not allowed (cf. User's manual, section 5.1, "Options within Macros").
- The way to get the previous (invalid) alternative working is to use the option –save in a Lint comment:

```
#define TRUE /*lint -save -e506 */ !FALSE /*lint -restore */
```

  This quickly becomes unreadable, but it will work.
- Equivalent, but easier to read in source code, would be to use the option `-emacro(506,TRUE)`. This option has to precede the macro definition, so place it in a Lint comment before defining the macro, or preferably in your project options file.

Clearly the last alternative has my personal preference: Used either just before the definition, or even in your project options file, the effect is to only suppress the irrelevant message in those cases where it really is unnecessary. And it keeps your code readable.
Another issue with macros like `TRUE`/`FALSE` or `VALID`/`INVALID` arises when MISRA[3]-rules are a requirement. The specific rule (rule 49 in the 1998 standard, rule 13.2 in the 2004 standard) reads "Tests of a value against zero should be made explicit, unless the operand is effectively Boolean." For details, refer to the MISRA specification.
This requirement has implications for our macros, since we are not allowed anymore to specify like this:

```
int fIsValueValid(int iValue);
...
if (fIsValueValid(25))
{
  ...
}
```

We are required to specify "`if (TRUE == fIsValueValid(25))`" instead, and if `TRUE` is defined as "`!FALSE`", message 731 "Boolean argument to equal/not equal" is produced.
In this case, specifying the option `-emacro(731,TRUE)` is not sufficient (refer to the location where the virtual `-restore` option would be placed), since the complete expression is the source for this message.
If you still need to work like this, consider using the form `--emacro((731),TRUE)`. This will place the message inhibition in the syntax tree and cover the entire expression in which the macro is used. For some messages (not message number 731 as in this example) you may even need to consider using the form, `--emacro({731},TRUE)`. For details, see the Lint documentation.
Important is: Start with the simplest inhibition options, and only use the more powerful ones when really necessary. Otherwise, your messages will disappear, but other (unaddressed) messages may also be suppressed. Let 'good enough' be sufficient!

## Using assert and derivate functions

Using the function `assert` to validate assumptions at least in debug code makes absolute sense. If the condition presented to the `assert` macro is false, `assert` will call the function `abort`. The macro, however, is part of the Standard C Library and defined in ***assert.h*** as offered by the compiler tool chain. And even if you never include the library include ***assert.h***,

[3] MISRA: Motor-Industry standard for programming in C; increasingly relevant for embedded development in the automotive industry.

MISRA (and common sense) prohibit the redefinition of anything defined in the Standard C Library[4].
Embedded systems therefore frequently create their own assert facility, usually without any reference to the Standard Library.
Now back to PC Lint. PC Lint knows the semantics of the `assert` macro: That the argument is checked for true/false, and if the condition is false, the `assert` macro (sometimes a function) will not return to the caller. It will also use the knowledge about the truth of the condition to enhance its knowledge about the values of variables. Similar knowledge is available on the `exit` and `abort` functions.
Let's have an example:

```
#define MyAssert(cond) { if (!(cond)) MyAbort(); }

extern void MyAbort(void);

int LocalArray[10] = {0};

int main (void)
{
  unsigned int uLoop;

  for (uLoop = 0; uLoop < 10; uLoop += 2)
  {
    MyAssert(uLoop < 9);
    LocalArray[uLoop] += LocalArray[uLoop + 1];
  }
  return 0;
}
```

Since the loop counter is incremented in steps of 2, the largest value for `uLoop` will be 8, so `LocalArray[uLoop + 1]` will be a valid array access. PC Lint will complain, though, since the current version (tested with version 8.00t) does not realize that incrementing will stop after 8. The `MyAssert` statement would be the way to tell PC Lint, but PC Lint doesn't know about the special properties of `MyAbort`. Try the PC Lint command:

```
lint-nt -u LintTest.c
```

This will produce message 661:
`Possible access of out-of-bounds pointer (1 beyond end of data) by operator '['`
But if we tell PC Lint that `MyAbort` is like `exit` in that it will not return to the caller, PC Lint will discover that the for-loop body cannot be executed with `uLoop` equal to 9, accessing the out-of-bounds pointer, and the message will disappear.

```
lint-nt -function(exit,MyAbort) -u LintTest.c
```

Yes, this solution has been described in the manual, refer to chapter 10.2.1. However, there are more possibilities than described in the chapter; the implications of defining everything yourself are not clear to some developers; and finally, this article is much shorter than the PC Lint manual.

# A final few words

Would you use a blunt knife, unless you need a screwdriver? Not likely. So, sharpen your knife! Patch Lint to the most recent patch level (9.0b at the time of this writing, January 2009, or if you are still using version 8, the patch level is 8.00x; consider upgrading to 9.0, it's more than worth it!). Patching is free for everyone within a major version: a license for version 8.00 gives you every patch up to 8.00x, independently from your current patch level. And the same is true for version 9.0.
And if you really use version 7.50 (some are among us still), at least use version 7.50ad, that patches page is yet available, and seriously consider buying an upgrade to version 9.0b or later.
If you need support with your endeavours, don't email me directly, but go to http://www.gimpel.com/Discussion.cfm, a forum for peer-support. Yes, I hang around there quite frequently, but I'm not the only one, and I might have an assignment keeping me away from the Internet for a while. I will also refer any direct questions to that forum.

[4] MISRA-C: 2004 Rule 20.1: Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined. `assert` is specifically mentioned.

And yes, I also offer consultancy, so if you need someone to do this for you, please _do_ email me, I'll be glad to make you an offer, on-site or off-site, perfectly tailored to your needs. But then, this document is not an advertisement, so I'll leave it at that.

# Postscript

I have the hope that this document will prove useful to many developers with access to PC Lint as a tool, but no real experience in wielding it. And maybe even some experienced user can find something new, as I constantly do.
This article has been intended to augment the chapter "Living with Lint" in the PC Lint manual, but it cannot replace the full manual. If you will, try the methods I have outlined, read relevant parts of the manual, and organize your options in a way useful to you. And before using Lint comments in your source code to eliminate individual messages or groups of messages, ask yourself if there could be a way using just the configuration files, not touching the code proper.
If new issues arise, or questions to certain problematic areas accumulate, I may produce newer versions of this article. Comments and criticism are always welcome.
Good luck linting!