

# The Tile Hill Style Guide

—

## C Style Guide for embedded systems

By

Chris Hills

First Edition 1.3  
10 April 2006

Part of the [QuEST](#) series:- [QA2](#)



[chills@phaedrus.org](mailto:chills@phaedrus.org)  
[QuEST.phaedrus.org/](http://QuEST.phaedrus.org/)



# The Tile Hill Style Guide

—

## C Style Guide for embedded systems

First Edition 1.3

10 April 2006

By

Eur. Ing. **Chris Hills** BSc, C. Eng., MIEE, MIEEE, FRGS

Copyright Chris A Hills 2001–2006

The right of Chris A Hills to be identified as the author of this work has been asserted by him in accordance with the Copyright, Designs and Patents Act 1988

The slides and copies of this paper (and subsequent versions) the templates, any source code and the power point slides will be available at

<http://quest.phaedsys.org/>,  
[chills@phaedsys.org](mailto:chills@phaedsys.org)

The *ART* in Embedded Engineering  
comes through  
Engineering discipline.

### Quality Embedded Software Techniques

**QuEST** is a series of papers based around the theme of Quality embedded systems. Not for a specific industry or type of work but for all embedded C. It is usually faster, more efficient and surprisingly a lot more fun when things work well.

### QuEST Series

<b>QuEST 0</b>	Introduction & SCIL-Level
<b>QuEST 1</b>	Embedded C Traps and Pitfalls
<b>QuEST 2</b>	Embedded Debuggers
<b>QuEST 3</b>	Advanced Embedded Testing For Fun

### Additional Information

<b>QA1</b>	SCIL-Level
<b>QA2</b>	Tile Hill Style Guide
<b>QA3</b>	QuEST-C
<b>QA4</b>	PC-Lint MISRA-C Compliance Matrix



## Contents

<b>1.0</b>	<b><i>Introduction</i></b> .....	<b>7</b>
2.1	The Tile Hill Style Guide– History.....	7
2.2	Availability and templates .....	8
2.3	Errors and omissions .....	8
<b>2.0</b>	<b><i>The Style Guide– Rational</i></b> .....	<b>9</b>
2.4	General notes .....	9
<b>3.0</b>	<b><i>Version control and other tools</i></b> .....	<b>14</b>
2.5	Lint & static analysis .....	15
2.6	DA–C.....	16
<b>4.0</b>	<b><i>The Tile Hill Embedded C Style Guide</i></b> .....	<b>19</b>
<b>2.7</b>	<b>Files</b> .....	<b>19</b>
2.7.1	File naming .....	19
2.7.2	File names should be kept short.....	19
2.7.3	Header file names .....	19
2.7.4	Source files should be ASCII chars only.....	19
2.7.5	Directory Names.....	19
<b>2.8</b>	<b>File name extensions</b> .....	<b>20</b>
2.8.1	C/C++ File extensions .....	20
2.8.2	Assembly files .....	20
2.8.3	Miscellaneous files. ....	20
<b>2.9</b>	<b>Variable names</b> .....	<b>20</b>
<b>2.10</b>	<b>White Space</b> .....	<b>21</b>
<b>2.11</b>	<b>Templates for files</b> .....	<b>21</b>
2.11.1	Source File Template .....	21
4.3.1.1.	File Information Block.....	21
4.3.1.2.	Other information .....	21
2.11.1.1.1	Include.....	22
2.11.1.1.2	Order of includes .....	22
2.11.1.1.3	Absolute paths may not be used .....	22
2.11.1.1.4	Define .....	22
2.11.1.1.5	Global .....	23
2.11.1.1.6	Prototypes .....	23
4.3.1.3.	File end of file Marker.....	24
2.11.2	Templates for functions .....	26
2.11.3	Order of functions.....	26
4.3.1.4.	Main function.....	27
2.11.4	Templates for Header files.....	27
<b>2.12</b>	<b>Braces and the One True Faith</b> .....	<b>28</b>
2.12.1	IF .....	28
2.12.2	For .....	29
2.12.3	While.....	29
2.12.4	Do.....	29
2.12.5	Nested .....	30
<b>2.13</b>	<b>Constants, Defines and Types</b> .....	<b>31</b>
2.13.1	Long constants.....	31
2.13.2	Hex constants .....	31
2.13.3	Octal Constants.....	31
2.13.4	Const .....	31

2.13.5 Defines .....	32
2.13.6 Types .....	32
2.13.7 Macros .....	33
4.3.1.5. Macros shall be balanced .....	33
4.3.1.6. Macro shall not be undefined .....	33
4.3.1.7. Functions shall be used where possible .....	33
4.3.1.8. Structures and Unions .....	33
<b>2.14 Flow Control .....</b>	<b>35</b>
2.14.1 Function calls in control clauses .....	35
2.14.2 Null Statements .....	35
<b>2.15 Comments .....</b>	<b>35</b>
2.15.1 Nested comments .....	36
2.15.2 Commenting out code .....	36
2.15.3 Comment density .....	36
<b>5.0 Source Code Templates .....</b>	<b>37</b>
<b>2.16 Source Safe Templates .....</b>	<b>38</b>
<b>2.17 MKS Source Integrity Templates .....</b>	<b>40</b>
<b>2.18 PVCS Templates .....</b>	<b>42</b>
<b>2.19 Clear CASE Templates .....</b>	<b>44</b>
<b>2.20 Component Software's RCS .....</b>	<b>46</b>
<b>6.0 Appendix B Lint and Example Program .....</b>	<b>49</b>
<b>7.0 References .....</b>	<b>53</b>
<b>8.0 Standards .....</b>	<b>57</b>

# The Tile Hill Style Guide

-

## C Style Guide for embedded systems

### 1.0 Introduction

When writing C there are two types of guide available and programmers often confuse them. Firstly the style guide which will be described here. Style dictates source code layout and should ensure a uniform and more readable format. A uniform style makes errors and non-conformance stand out. It is this word that causes the problems "conformance".... Free thinking programmers hate it. However for Engineers it is a cornerstone. This Guide is for Software *Engineers*. Many in embedded engineering come from a disciplined electronic engineering background and are used to working to strict regulations for many things. They understand that:-

The ART in Embedded Engineering  
comes through  
engineering discipline.

This style guide is not about safe use of C as such. The other type of guide covers the safe use of C, normally via a [safer] subset of C. This will restrict which parts of C can and cannot be used. In some cases it lays down very strict lines about how a particular feature is to be used. This type of guide usually has less resistance to being used because there is a clear reason for the rule. Style guides are just that. They have no solid engineering reason to do it this way instead of another way. The problem occurs when everyone has his or her own system....

For the safe use of C there are many, often narrow industry specific, guides available. However, there is one which is generic embedded and has escaped into widespread use. The MISRA-C guide (or to give it its full title:- The Motor Industry Software Reliability Association Guidelines For The Use Of The C Language In Vehicle Based Software). MISRA-C is available from MIRA (and also from Phaedrus Systems). MISRA-C is a very readable set of rules that make sense for most C programming, embedded or not.

### 2.1 The Tile Hill Style Guide- History

The Tile Hill style guide has been produced not because it is "The Best" or "Only Way" but because every time I mentioned style guides we were asked if we had one people could use.

When we told people to search the net, look in libraries etc they usually said, "Can you send us the one you use?" Well, you asked for it! This is it, the Style Guide I use at Phaedrus Systems.

It has been developed over the last 12 years of software Engineering in several small, medium and large companies most working to or at ISO9000. It has been used on

projects as diverse as an 8051 smart card through to a multiprocessor communications system that was based on Sparc, PowerPC and 68040 over time and also had some i486 content....

To get the folk law out of the way first: It is called the Tile Hill guide because at the time of it's first publication to a wider audience (I.E. Not to a project I was working on) I was working in Coventry close to Tile Hill that I had to drive through twice a day. Also the title is a play on the legendary "**Indian Hill Recommended C Style and Coding Standards**" from AT&T Bell labs. My copy is Version 6 dated 1990.



Indian Hill was one of the famous Bell Labs. Tile Hill has no such distinction.

The photo shows the **Tile Hill Newsagents** who do a fine range of newspapers, magazines, greetings cards, sweets and soft drinks. . They do not (as far as I know) have any involvement in software or embedded

Engineering

**Note:-** I do not claim that the Tile Hill Guide is better or replaces the Indian Hill document. I just produced my own guide to the style I use because people asked for one. However whilst the Indian Hill guide is generic the Tile Hill Guide is aimed at embedded systems. I am not sure how much difference that makes if any.

## 2.2 Availability and templates

The Tile Hill Guide is freely available in electronic form, with all the source code templates from <http://quest.phaedsys.org/> At the time of writing the Indian Hill guide was available from at least ten sites. Do a search on "Indian Hill Labs". Though all the ones I found were dated 1990.

Jack Ganssel also has an embedded project guide available from <http://www.ganssle.com/>.

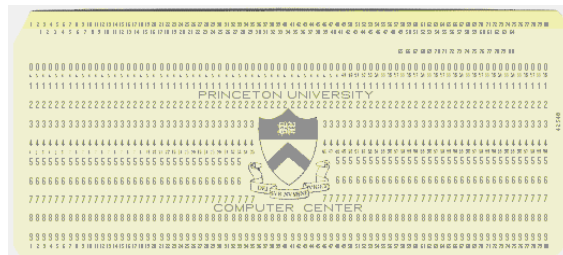
## 2.3 Errors and omissions

If you find any errors or omissions, have any constructive criticisms, or want to see something added contact the author at: - <mailto:chills@phaedsys.org> New versions will be posted to <http://quest.phaedsys.org/>

## 2.0 The Style Guide- Rational

### 2.4 General notes

The idea behind the style guide and the reason they are misused is free will. Many years ago (1950's to 1960's) computer programs were written on punched cards with 80 columns. The layout of each line was fixed as was whole sections of the program..



This card is from Douglas W. Jones A History of punched cards [Jones], which provides a fascinating history to early computer systems.

Computer languages developed from punched cards to initially typewriter like terminals and then onto VDU screens. Computer languages continued over from punch cards and required that commands start in certain columns and in a particular form COBOL or FORTRAN for example.

```
MMAX = 1
  90 IF (MMAX-N) 100, 130, 130
100 ISTEP = 2*MMAX
    DO 120 M=1,MMAX
      THETA = PI*FLOAT( ISI*( M-1 ) )/FLOAT(MMAX)
      W = CMPLX( COS( THETA ) ,SIN( THETA ) )
      DO 110 I=M,N,ISTEP
        J = I + MMAX
        TEMP = W*DATA(J)
        DATA(J) = DATA(I) - TEMP
        DATA(I) = DATA(I) + TEMP
110   CONTINUE
120   CONTINUE
      MMAX = ISTEP
      GO TO 90
130 IF (ISI) 160, 140, 140
```

Example of Fortran.

As computers developed so the languages developed away from the punched card and there was more freedom of layout.

C was developed in the 1970's when VDU screens were common and new systems did not use punched cards. Therefore C was developed as a free format language. There is no fixed layout as regards columns. (There are a few exceptions to do with the pre-processor)

One white space has the same weight as 50 white spaces... This assumes that you are using black on white. For those of you using DOS screens it is black spaces (in the case of Joe it is lilac spaces). Effectively, this means that the programmer is free to layout the source code as he or she sees fit. What is more there is no technical reason why the programmer has to keep their style constant. Even from line to line.

Many programmers will complain that they can use their own style and anything else imposed is an infringement of Civil Liberties and the end of Civilisation As We Know

It! Most Project managers have too many other things to worry about. As long as the code compiles (reasonably<sup>1</sup>) clean they are happy.

The problem comes when you have several programmers working on one project or worse still several programmers on the project are changed.... The original "geniuses" move on and someone else has to do the maintenance. This is when you end up with many conflicting styles. Often within the same source file! This is where C learnt its reputation as being a *read only* language.

As the army discovered a long time ago it is easier to count and control a group of people if they are standing in lines and sets of lines than if they are just standing in a group. As an example (do this in private or you will thought of as very sad ☹, tip a box of paper clips on to your desk. Try and count them without moving them. Not easy is it. You could could them by moving them from one pile to another. However if you loose count you have to start again. Now try putting them it to rows of 5 with 5 rows to a block. Then all you have to do is count the blocks. It is also very easy to see if there is a paper clip missing in any row or block. Likewise when the source code is laid out in a standard way is it far easier to spot anomalies and errors.

I worked on one fun project involving six nationalities on three continents. One of my team sent me an email after doing a review on another teams code:- (N.B. time how long it takes to read what this says)

```
th   eo           t
her   tea       mRe           Gua
R     d         so         ru  c
E           Co   DeLay
O           T     A           Sana
Rtf                               or
```

It took me a while to work out what it actually said was:-  
ThEoThErTeAmReGuArDsOrUcEc0DeLaYouTaSaNaRtFoRm

Sorry, I meant "**theo tert eamr egua rdso ruce code layo tasa nart form**" or to restrict my civil liberties and stifle my creative spirit; "The other team regard source code layout as an art form". I am sure that you instantly spotted the '0' (zero) in place of the O and the misspellings. In fact now I come to look at it, the first 3 versions have different errors but I am sure that was obvious to you!

That illustration should have convinced you, even if it is only the time saving on reading it, that a uniform style to a set convention is a good idea. It makes the source easily readable to all with the likelihood of fewer errors.

Another far more dangerous example of code layout causing problems is one that actually came originally from a problem in a rapid transit system in a major city.

```
interlock = OFF;

if(TRUE == stop)
    flag = ON;
    interlock = ON;

if(ON == interlock)
```

---

<sup>1</sup> Reasonably? For some this is only "some" warning, others turn off the compiler warnings and only look at errors....

```
        open_doors();
else
    apply_brakes();
    sound_alarm();
```

What it should do is only open the doors *if* stopped *else* apply brakes and sound alarm but not open doors. This code will in fact, obviously, always open the doors and sound the alarm but not apply the brakes.

I insisted that all code produced with teams I am involved with rigidly adhere to the principal of always using braces where possible, even on one line while or if statements as above. This may sound a bit draconian despite the example above but I have good reason.

I instigated this rule after three of my team spent two days trying to trace a bug caused by a two line *if* statement where only one line was actually inside the *if*. It caused an error some distance from the *if* statement and was not immediately linked to the problem. When the *if* statement was considered all three engineers glancing at it saw a correct *if* statement and mentally put braces round the two statements. The mind saw what it thought should be there. The human mind is good at filling in the blanks with what it thinks is right. Ask any married person!

In this case the error was combined with another similar “non error” to produce a real problem much further away. The reason for c style guides is to make the code uniform so reducing the silly errors like this. There is no right or wrong style, as long as it is consistent, but some have a tendency to reduce errors more than others.

The previous code example (according to my pedantic formatting) is actually the following:

```
interlock = OFF;

if(TRUE == stop)
{
    flag = ON;
}
interlock = ON;

if(ON == interlock)
{
    open_doors();
}
else
{
    apply_brakes();
}
sound_alarm();
```

Whereas what was meant was:

```
interlock = OFF;

if(TRUE == stop)
{
    flag = ON;
interlock = ON;
}
```

```
if(ON == interlock)
{
    open_doors();
}
else
{
    apply_brakes();
    sound_alarm();
}
```

By pedantically putting the braces in (taking a couple of seconds) we could have saved about 5 man-days of effort!

As mentioned previously this problem was actually based on a real problem on a rapid transit system in the Far East though programmers in the Midlands wrote the code! It actually made it as far as the test runs. The bug was only found by accident after another "freak" bug triggered the events for this one to become visible.

This should convince you that a uniform style is a good idea.... Well your troubles have just begun! Which style of layout to use? Everyone has their favourite and can see no reason why they should change. It is one of the most contentious parts of any style guide.

There are several recognised styles. All are equally good and bad. In personal order of preference from top to bottom with my favourite at the top.

```
if(style 1 Exdent)
{
    statement;
statement;
}
```

```
if(style 2 Indent)
{
    statement;
statement;
}
```

```
if(style 3 K&R){
statement;
statement;
}
```

```
if(style 4)
{statement;
    statement;
}
```

As a final thought: Your source code could land you in prison! Currently you have to show "due diligence" in anything you produce. Thus if there is a court case over something to do with a system containing your software you may be called upon to show that you exercised "Due Diligence" in constructing the software. The claimants are likely to have experts who will explain what an ideal software development process should be.

A uniform style in programming along with a coding standard has long been recognised as "Best Practice" in the software industry. You can be sure that the insurance company for the claimants will bring it up somewhere. They will ask "an you demonstrate that you took due diligence in constructing this software?"

Of course as a director or manager you may have more pressing problems if the Corporate Manslaughter Bill goes through. The stakes are getting higher. Given that software produced to a consistent style using a sensible coding standard using the correct support tools has consistently been shown to get to market faster with fewer bugs there is no reason (legal or commercial) not to do it.

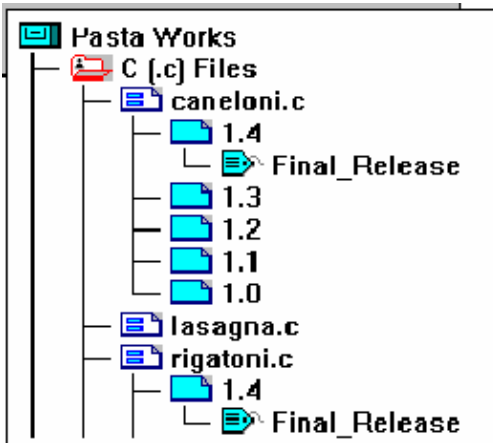
There is a fuller discussion of this topic in Embedded C: Traps and Pitfalls at [www.phaedsys.org](http://www.phaedsys.org) and the Home office documents are available at:

<http://www.homeoffice.gov.uk/consult/invmans.htm>

### 3.0 Version control and other tools

Version Control is a much under used method for getting bug free code. For a full explanation see *Embedded C Traps and Pitfalls [Hills]*. In simple terms **version control** means keeping track of *every version* of a source file. This seems a little daunting and more likely to introduce more errors. However if implemented properly it can drastically reduce errors and save a lot of time.

There are many ways of doing version control. The simplest is to archive directories of source code. The next step is one of the many version control programs. These are a form of database program. These programs make it possible to retrieve any version of the source code either by specific version or by using labels whole builds can be retrieved at the click of a mouse.



This can save much time hunting around for old versions and is also extremely useful when a new set of mods go wrong and you have to go back to a known good version.

One of the more useful features of most commercial VCS packages is "keywords" and "keyword expansion". Key words can automate much of the documentation in source files. Essentially when a VCS keyword is placed in a source file the VCS program will expand the keyword with specific information such as date, time, file-name user name and even file history.

For example the following file header block

```
/*  
/*****  
** $Workfile: $  
** Name: Application Block  
** Copyright :Keil (uk) 1999  
** $Author: $  
** $Revision: $  
**  
** Analysis reference:123/ab/45678/001 5.6  
** Input Parameters: NONE  
** Output Parameters: NONE  
**  
** $Log: $  
**  
** End of history  
**/  
*/
```

will automatically have the keywords expanded when the file is put in (I.E. a version saved) and then taken back out to work on again.

```
/*  
/*****
```

```

** $Workfile: U1CO0001.C $
** Name: Application Block
** Copyright :Keil (uk) 1999
** $Author: Chris Hills$
** $Revision: 1.1 $
**
** Analysis reference:123/ab/45678/001 5.6
** Input Parameters: NONE
** Output Parameters: NONE
**
** $Log: C:/ENG/KOS2/A2C001.C_V $
**
** Rev 1.1 06 May 1998 16:48:28 HILLS_CA
** Issued for review
**
** Rev 1.0 01 Apr 1998 13:09:02 HILLS_CA
** initial version
**
** End of history
*/

```

The good news is that the users cannot alter the comments above in the key word text. For example if I changed the history section the next time the file is pulled out the VCS will recreate the original text. The VCS administrator can change the history comments in the VCS but it is far easier to do it right the first time!

Incidentally one thing you should remember... it is not only the source code that needs version control. Test scripts, programs, make files and even the tools may need version control

There are several free version control programs about principally RCS, which is part of most Unix implementations. I have a PD DOS version. However a well documented (and used) manual version you do yourself, I.E. history logs, archive directories and back up to CD ROMs etc may be all you need.

It should be noted that a well-organised file system will save a lot of time and effort and makes a lot of sense in pure commercial terms never mind the engineering arguments. Especially if you build up a library of reusable components and use them in many versions of software.

Note:- Version Control Systems (VCS) are also known as Revision Control Systems (RCS). It is also sometimes referred to as Configuration Management System. In all cases System is often replaced with the word Software when talking about the control system software ... Sometimes it is referred to as Software VCS (SVCS). The programs are often called VCS, RCS, SCCS depending where you first started to use them.

## 2.5 Lint & static analysis

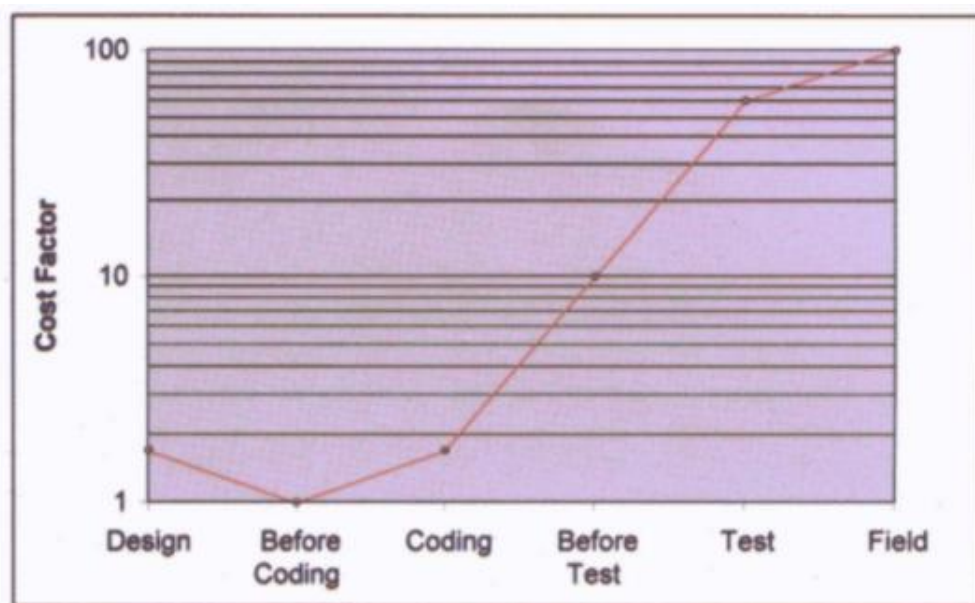
Lint is a static analysis tool. It is like a spelling and grammar checker for C. It will pick up basic indentation, globals that could be local, misuse of pointers, loss of precision and many other dubious uses of C. Static analysis will analyse source code without a compiler. This means that it can check in complete and un-compliable source.

Lint was the first and has been around since 1979 it is a command line tool that was intended to be run as part of the compiler chain in the make file. These days it will integrate into most IDE's.

It should be stressed that static analysis is recommended by virtually every C and C++ style and coding standards since the original Lint was created by the same team that put C and Unix together in 1974. Denise Ritchie in his paper: *The Development of the C Language* says:

Although the first edition of K&R described most of the rules that brought C's type structure to its present form, many programs written in the older, more relaxed style persisted, and so did compilers that tolerated it. To encourage people to pay more attention to the official language rules, to detect legal but suspicious constructions, and to help find interface mismatches undetectable with simple mechanisms for separate compilation, Steve Johnson adapted his pcc compiler to produce lint [Johnson 79b], which scanned a set of files and remarked on dubious constructions.

It is not a case of if you should use static analysis but which tool you will use!



Why use static analysis? The graph below should give some indication as to the cost of finding bugs. The closer to the point where they start the better.

There are many tools (starting with Lint) going up to 10,000 £ or \$ with extensive rules and templates many that can be made to test for house coding standards and styles. See Appendix B.

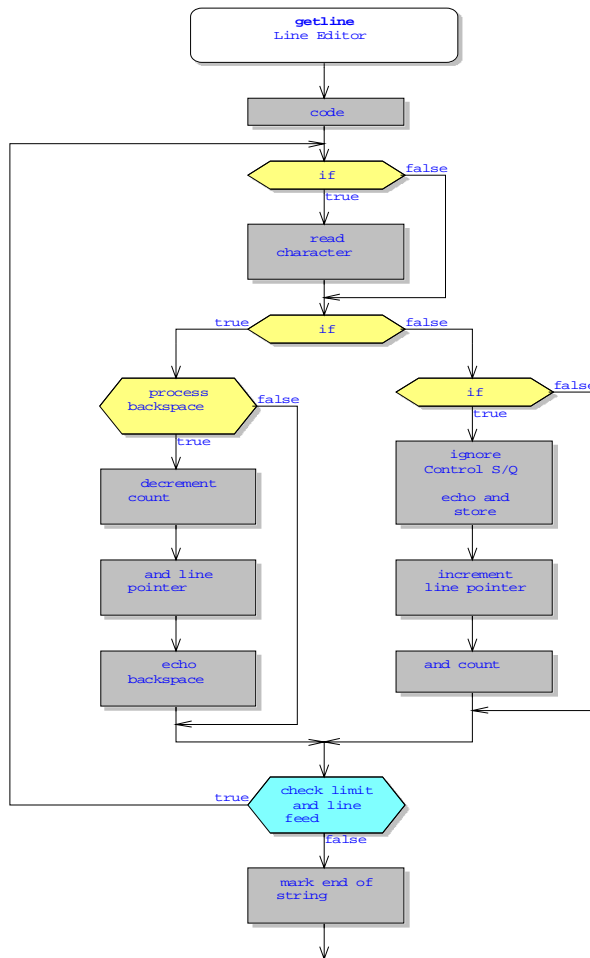
The simplest and easiest static analyser to use (and also the least expensive) is Lint. There are several free and public domain ones about. The first commercial one is PC-Lint at just over £100, which comes complete with configuration files for the majority of the worlds embedded C compilers. Which saves hours of setting up.

After that the sky is the limit which is often not the case for some embedded aerospace projects. Generally the more expensive the tool the more it will do, the deeper it will hunt and the better the graphical user interface.

## 2.6 DA-C

The program Development Assistant for C (DAC) is a wonderful tool where there is a lot of legacy code that needs to be brought up to standard. It will produce, from the

source code, complete, commented, function flowcharts and program structure charts, as well as data structure charts all of which can be cut and pasted into your documentation.



Whilst not part of a style guide it can be very useful in re-documenting legacy projects where the original documents and design cannot be found. It too has a static analyser but it is not as powerful as lint. This makes it ideal for a first pass over legacy code before documenting.

One of my tests for commenting is to run DAC over the code and see if the flow charts make sense. The boxes are automatically completed using the comments in the code. Where there are no comments the word "CODE" is inserted. If the automatically generated flow chart does not make sense then the code had better be well constructed using defines and intuitive variable names.

As of summer 2002 DAC is automatically able to create documentation from the source code to IEEE 1016. It generate structure charts, tables of parameters and module/function information all with section

numbering, contents and title page. The areas you have to complete contain instructions in a different colour.



## 4.0 The Tile Hill Embedded C Style Guide

This *style* guide should be read in conjunction with a suitable coding standard such as MISRA-C

This style guide should be used as part of a unified software development system. See Embedded C: Traps and Pitfalls [Hills] and the SCIL-Level Guide

A style guide is there to make the source readable. Coding standards make the code safer. Other tools make the project safer

This guide should be used for ALL software on the project. Exceptions are standard third party libraries. Where third party source is developed for the project it should comply with this guide and all of the project standards.

### 2.7 Files

#### 2.7.1 File naming

**File names shall not contain spaces.** Whilst most host operating systems can now handle spaces in file names many of the tools whose roots are in DOS do not. Also the tools may handle spaces and other characters differently. The same holds true for directory paths: **spaces should not be used in directory names.**

File names should not be used as part of a version control system. For example A123001.c for version 1 and A123002.c for version 2 etc  
If version control is required a suitable program should be used.

#### 2.7.2 File names should be kept short

The eight. three configuration for files is now a restriction of the tools rather than host operating systems. However as many tools still work to this format it is advisable to stay with it. In any event file names should be less that 16 alphanumeric ASCII characters.

#### 2.7.3 Header file names

Header file names for header files containing function prototypes or defines for a particular source file should reflect the name of the source file they belong to. Header files should not use the name of any known standard library header file regardless of whether the library header files are being used or not.

#### 2.7.4 Source files should be ASCII chars only

Source files should only contain printable ASCII characters.

This means that source files cannot be written in a word processor as a \*.doc file. The only permitted exception is the EOF (End Of File) marker and the carriage return/line feed marker. Note this can cause problems if moving between operating systems such as VMS, Windows, MAC and Unix.

#### 2.7.5 Directory Names

Directory names shall not contain spaces or non-alphanumeric characters, the exception being the delimiter. Whilst some operating systems (ie Unix) clearly mandate and uniformly handle spaces and long directory names others (ie MS Windows) do not.

Directory names should be of the style

Z:/name/nametwo/something

And not this sort of thing

Z:/abc - project one/version 123.5 - debug friday

Confirm any path lengths of all tools prior to starting a project. As a guide 256 is the maximum length of a full directory path.

## 2.8 File name extensions

### 2.8.1 C/C++ File extensions

C/C++ File extensions shall follow the table below except where tools used require different extensions. Extensions should be up to three alpha characters.

C source files	*.c
C header files	*.h
C++ source files	*.cpp
C++ header files	*.hpp

### 2.8.2 Assembly files

Assembly files may have to use extensions required by the tool chain. Where this is not the case the following shall apply

*.a51	
251	*.a51
166	*.a66
Z80	*.a80
X86	*.a86

Any other family should follow a similar pattern

### 2.8.3 Miscellaneous files.

File extensions shall follow the table below except where tools used require different extensions. Extensions should be up to three alpha characters.

Object files	*.obj
Libraries	*.lib
OMF files	*.omf
Elf files	*.elf
List files	*.lst
Map files	*.map
Make files	*.mak
Batch files	*.bat
Intel Hex files	*.hex
S Rec files	*.mot

## 2.9 Variable names

## 2.10 White Space

## 2.11 Templates for files

All source code is held in a number of files. These files should be laid out in a standard manner. The easiest way of doing this is to use templates for the file as a whole, the functions within it and the header files.

Much of the work involved can be automated using Revision Control Software (RCS). The version shown in the text is a manual version. The appendices contain versions of the templates for MS Source Safe, PVCS Version Manager, MKS Source Integrity and Clear Case. These templates will also be available from <http://www.phaedsys.org>

### 2.11.1 Source File Template

This should be a standard template used for all source code developed on the project. It is not required to re-engineer any third party source such as libraries. Any third part source developed for the project should use the templates.

#### 4.3.1.1. File Information Block

These files should have an information block that conveys information to the reader. This information should be:

- The name of the file.
- The name of the application or project \*
- The initial author \*
- The copyright
- The analysis or design reference
- The current revision\*
- The change history.\*

The items marked \* can usually be supplied automatically by the VCS . The copyright block will usually be insisted upon and supplied by the company. It is usually the only part of the software process they take any note of!

#### 4.3.1.2. Other information

The source file will contain logical sections. Functions are covered later. The file will start with the File Information block as detailed above.

Following the FIB shall be the header files. The section shall start with a single line comment using \* across the whole page. On the right hand end of the comment then the name of the section shall be placed with the section name at the right hand end of the comment. The case may be upper, lower or title case but must be consistent across the project.

```
/****** Section Name */
```

The comment shall be full width of the standard listing paper when printed. This may be modified to the maximum width before any wrapping or loss of text on screen or printer. In any even all the lines must be uniform. The sections shall be in the order described in the following sections.

#### **2.11.1.1.1 Include**

```
/****** Include */
```

On the far right shall be the word "Include" to assist with locating the section in code reviews.

The header files shall be included in the following order

```
#define CO0001_M
#include <system.h>
#include <third-party-sw.h>
#include <project.h>
#include "local.h"
#include "ownheader.h"
```

The line #define CO0001\_M is used with the files own header file. This permits the correct definition of local and external defines, prototypes etc. see section on header file templates.

Note it is not permitted to include a \*.c file at any time.

#### **2.11.1.1.2 Order of includes**

The order of system, third party, project, local and own-header is because in some cases the local or project headers may use things in the system or third party libraries.

It has been suggested that local headers should be first to prove that they are stand alone and complete. However this falls over where typedefs for standard types are used. Also the local files may require things defined in third party libraries (buffer sizes , arrays or structures for example) which may in turn require the standard libraries first.

The only other way round this is to have nested includes which is an even greater risk and should be avoided at all costs

#### **2.11.1.1.3 Absolute paths may not be used**

In some third party libraries that install to include/libX where include is the standard compiler library directory the following is permitted

```
#include <../libX/third-party-sw.h>
```

#### **2.11.1.1.4 Define**

The defines section shall be marked as follows with the word "defines" at the right hand end of the comment:-

```

/***** Defines */

#define     NAME  (value)
#define     NAME1 (value)
#define     NAME2 (value)

```

Defines should be spaced with tabs into columns. This enhances readability.

The defined NAME should be UPPERCASE and the defined (*value*) should be in parenthesis to avoid any side effects.

#### 2.11.1.1.5 Global

The Globals section shall be marked as follows with the word "Globals" at the right hand end of the comment

```

/***** Globals */

uint8_t     count      = 0;
int8_t      flag_one   = 0;

uint16_t    motor_one  = 0;
int16_t     error_rate = 1;

```

As with defines the declarations should be tabbed into columns. This enhances readability.

There should only be one declaration per line.

Types should be grouped logically, I.E. 8 bit, then 16 bit etc or by use. All the globals relating to a particular function.

#### 2.11.1.1.6 Prototypes

The Prototypes section shall be marked as follows with the word "Prototypes" at the right hand end of the comment.

```

/***** Prototypes */

static void function(uint8_t variable);

```

Only static or local function prototypes shall be permitted in this section. Externally visible prototypes shall be in a header. Therefore all prototypes in this section shall start with the C keyword "static"

The use of the static keyword tells the compiler that the function will only be called by other functions in this file. Therefore many compilers are able to use shorter, faster jumps of calls for this function. Some optimising compilers will even "move" the functions around in order to get the most efficient use of jumps to the function. This results in faster and more compact code.

The other reason for the use of static is that it effectively hides the function from the rest of the program.

##### 2.11.1.1.6.1 Function prototypes shall be fully specified.

That is the parameters and return to a function shall have both a name and a type. That is the sign and the magnitude shall be given. For example:-

```
static void function(uint8_t variable);
```

The parameter is an unsigned char called "variable"

#### 4.3.1.3. File end of file Marker

Each file shall end with a textual End Of File marker or comment giving the name of the source file. NOTE this is NOT the same as an EOF token which is a non-printable ASCII character that is used to make the end of the file for the editor.

Using Version control software it is possible, as in the example, to get the VCS to put the file name in automatically

This form of end comment is useful when looking at printout to know you have the last sheet. Some projects also insert version number as well.

```
/*  
***** End of $Workfile: CO0001.C $ *****  
*/
```

Example Template

```

/*****
** $Workfile: CO0001.C $
** Name: Application Block
** Copyright :Phaedrus Systems 1999
** $Author: Chris Hills$
** $Revision: 1.1 $
**
** Analysis reference:123/ab/45678/001 5.6
**
**
** $Log: C:/ENG/caos/CO0001.C_V $
**
** Rev 1.1 06 May 1998 16:48:28 HILLS_CA
** Issued for review
**
** Rev 1.0 01 Apr 1998 13:09:02 HILLS_CA
** initial version
**
*/

/***** Include */
#define CO0001_M

#include < system.h>
#include < thirdpartysw.h>
#include " project.h"

/***** Defines */

#define name value
#define name value
#define name value

/***** Globals */

/***** Prototypes */

static void function(void);

/***** Main */

void main(void)
{

}

/*****/
/***** End of $Workfile: CO0001.C $*****/
/*****/
```

### 2.11.2 Templates for functions

The file template should give the developer all the information on where the file started and how it got to its current state. The VCS in this case automatically puts in the history block. Where a VCs is not used it should be manually maintained. This should include any additional functions added (or removed) after the initial build.

Each function should also have a simple comment block giving the purpose of the function, the input and output parameters. Like the main file information block, where appropriate, the reference to the design or requirements should be included.

Only in exceptional cases should the function be fully documented (e.g. the description of algorithms) in the function template. This information should be in the documentation.

The function template shall start with a full width comment of the same length of the other comments in use. The function name shall be inserted

```
/* ***** Convert_One */
/* Name: Convert_one
**
** Purpose: Converts Fahrenheit to Celsius
**
** Design reference: abd_123-abc
**
** Input Parameters: Temperature in Fahrenheit
** Permitted range: -20 to 200
**
** Return Parameter: Temperature in Centigrade.
**
*/

uint_8 convert_one( Unit_8)
{
    .....
```

Where applicable ranges should be given, In this case the input Fahrenheit range permitted and the output range supplied.<sup>2</sup>

Main is, as always, a special case. In a self-hosted system main will not have any parameters or return values. It's function is clear therefore the second descriptive comment block is not required.

### 2.11.3 Order of functions

The functions shall be in top down order. In all source files the functions externally callable shall be at the top of the file. The most significant external function at the top of the file.

The functions shall then follow in the order in which they are called in main. Static functions shall be last. Where possible all functions shall be lower in the file than the function that called it.

---

<sup>2</sup> Had this simple rule been applied to a project a multimillion-pound rocket might have flown a bit longer than 29 seconds!

#### 4.3.1.4. Main function

There shall be a function called *main* which shall be the first C function that is run after the start-up assembler. Main shall be the first function in its source file but does not require a prototype.

In self-hosted environments *main* shall have the prototype:

```
void main(void)
```

Note: In strict ISO C in self-hosted environments "main" is not required to be called "main". It can technically be called anything you like. However 99% of the worlds debugging tools look for "main".

#### 2.11.4 Templates for Header files

Header files shall contain function prototypes for functions that are not static and must be visible outside their home source file.

The template shall start with a full width single line comment of \*'s

The File information block shall contain the following information:-

```
Name of the file*  
Copyright  
Author or team *  
Revision or version number *  
Reference to design or analysis document  
History log*
```

The items marked \* should be available automatically from the VCS.

One item that can also be added is the target CPU or compiler system that was used, also the intended crystal speed. This is more relevant where the header file is to be used when its source file is only available as an object file, as in the case of a library function.

The end of the file shall have a single line EOF comment that includes the name of the file.

NOTES:-

This header files contains

```
#ifndef CO0001_H  
#define CO0001_H
```

```
#endif /*end guard */
```

This permits only one inclusion of the header file and stops re-definitions. This should be part of your coding standard. The other set of defines

```
ifdef CO0001_M
```

```

    #define CO0001_SCOPE_ /**/
#else
    #define CO0001_SCOPE_ extern
#endif

CO0001_SCOPE_ void Application_Block1(void);
CO0001_SCOPE_ void Application_Block2(void);
CO0001_SCOPE_ void Application_Block3(void);

```

Are used with a #define in the files home source file #define CO0001\_M that is above the include section to permit the correct declaration of externs

```

/*****
/*
** $Workfile: CO0001.H $
**
** Copyright Phaedrus Systems Ltd
** Author: Chris Hills
**
** $Revision: 1.1 $
**
** Analysis reference: ab2-123-1.2.4
** Compiler : Keil C51
** Target : 8051/Fa
**
** $Log: G:/SHARED/ENG/caos/CO0001.H_v $
**
** Rev 1.0 01 Apr 1998 13:19:28 HILLS_CA
** Initial Version
*/

#ifndef CO0001_H
#define CO0001_H

#ifdef CO0001_M
    #define _SCOPE_ /**/
#else
    #define CO0001_SCOPE_ extern
#endif

CO0001_SCOPE_ void Application_Block1(void);

/***** END of $Workfile: CO0001.H $ *****/

```

## 2.12 Braces and the One True Faith

There are several styles for the placement of braces, K&R, indent and extent. **The rule is consistency. The preferred method shown here is extent.**

### 2.12.1 IF

```

if(clause)
{
    statement;
statement;
}

```

### 2.12.2 For

```
for (init; until; step)
{
statement;
statement;
}
```

### 2.12.3 While

```
while( clause)
{
    statement;
statement;
}
```

This construct may not execute the statements it contains depending how the while clause evaluates.

### 2.12.4 Do

```
do
{
    statement;
statement;
}(while)
```

This constructs ensures that in all cases the statements contained are evaluated at least once.

### 2.12.5 Nested

Nested if, whiles etc should be as follows

```
if(clause)
{
while( clause)
{
        statement;
statement;
}/*end while*/
        statement;
statement;
}/*end if*/
```

NOTE some old libraries and some tools may require K&R style

NOTE the use of `/*end if */` and `/*end while */` These are useful where there are nested sets of braces. In multiple nested ifs and while's this should be extended to more descriptive markers such as:

```
/* end while count */

or

/* end if flag true */
```

## 2.13 Constants, Defines and Types

Magic numbers or literal numbers in C shall not be permitted. Defines or Const shall be used. There is a subtle difference between a const and a define. A define is a textual replacement, for example:

```
#define TRUE 1
```

Wherever TRUE is found in the source (but not in comments) it will be physically replaced by a 1. With a const a type is also required e.g.

```
const uint8_t True = 1;
```

NOTE:- a const is NOT a define and therefore should not be all uppercase.

The difference is that the const takes up physical space but when debugging it will show up as "TRUE " (with type information) where as the define will show up as 1 with no other information. In an idea world consts would be used but in embedded systems memory is usually at a premium.

Note this should not be taken to extremes.

Where MAX\_STRING = 250 the following shall be permitted:-

```
uint8_t output_buffer[MAX_STRING+1]
```

Where an additional byte is required for the null terminator. However the following and all similar are prohibited:

```
#define ONE (1)
const uint8_t ONE = 1;
```

Exceptions would be for S-Boxes in DES routines.

### 2.13.1 Long constants

Long constants shall be written using a capital L suffix NOT a lower case l as a lower case l can be confused with a 1. Thus 121212 would be written:

```
#define NAME (121212L)
```

### 2.13.2 Hex constants

Hexadecimal constants shall be used for all bit masks and where bit patterns are important. They shall be written using a lower case x and the hexadecimal letters (A, B, C ,D, E, F) in upper case. Thus 0f would be written :

```
#define NAME (0x0F)
```

### 2.13.3 Octal Constants

Octal constants should not be used. The only logical exception is the octal zero, which is "0".

### 2.13.4 Const

A const shall be used on any values that shall not change during the lifetime of the program.

This will include parameters passed to functions where the parameter will not be changed in the function e.g.

```
uint8_t function( const uint8_t temperature );
```

### 2.13.5 Defines

Define shall be used on any values that shall not change during the lifetime of the program.

Defines shall be in UPPER-CASE and shall not be permitted to start with an underscore.

```
#define TRUE (1)
```

See also MACRO's in 4.5.7

### 2.13.6 Types

Types should be completely specified. This includes signed and unsigned types. Types should also have their size fully specified.

Type	Minimum # Bits	No Smaller Than
char	8	
short	16	char
int	16	short
long	32	int
float	24	
doubl	38	float
e	14	
any *	15	any *
char *	15	any *
void *		

Some machines have more than one possible size for a given type. The size you get can depend both on the compiler and on various compile-time flags. The following table shows "safe" type sizes on the majority of systems. Unsigned numbers are the same bit size as signed numbers.

Unless a short is both larger than a char and smaller than an int, short should not be used. I.E. short may be used where

```
char    == 8 bits
short  == 16 bits
int    == 32 bits
long   == 64 bits
```

In order to ensure portability and that the correct types are used ISO C defines the following types definitions to be used

```
char_t   for characters

int8_t   uint8_t
int16_t  uint16_t
int32_t  uint32_t
int64_t  uint64_t

float32_t
float64_t
float128_t
```

These should be used in a header file as follow (example for an 8 bit architecture).

```
typedef signed      char   int8_t;
typedef unsigned   char   uint8_t;
typedef signed     int     int16_t;
typedef unsigned   int     uint16_t;
typedef signed     long    int32_t;
typedef unsigned   long    uint32_t;
```

Note the use of tabs to keep the typedefs in columns.

### 2.13.7 Macros

In this guide a macro is treated differently to a defined constant for example

```
#define TRUE 1
```

A macro contains more than a single definition they also have the following misleading properties:.

- what you see on the page is not what the compiler sees
- they do not respect scope, and know nothing of the language
- errors appear at the point of usage not the point of definition

MACROS as with defines shall be ALL UPERCASE All other names must contain at least one lower case letter. No names should start with an underscore to avoid possible clashes with library code.

#### 4.3.1.5. Macros shall be balanced

Macros shall be balanced as regard parenthesis

```
#define TRUE (1)
```

and shall NOT be of the form

```
#define IF if(
```

```
#define WHILE while(
```

#### 4.3.1.6. Macro shall not be undefined

Macros shall not be undefined or redefined within a file.

#### 4.3.1.7. Functions shall be used where possible

C functions should be used where possible rather than complex macros

#### 4.3.1.8. Structures and Unions

Structures shall not be declared in a nested form.

```
struct name
{
    uint8_t size;
    uint8_t number;
    int16_t speed;
    int16_t direction;
    struct part
    {
        uint8_t rate1;
        uint8_t rate2;
        int16_t climb1;
        int16_t climb2;
    }parts;
};
```

Each structure should be declared separately as follows:

```
struct part
{
    uint8_t rate1;
    uint8_t rate2;
    int16_t climb1;
    int16_t climb2;
};
```

```
struct name
{
    uint8_t size;
    uint8_t number;
    int16_t speed;
    int16_t direction;
    struct part parts;
};
```

Only one declaration per line.

## 2.14 Flow Control

### 2.14.1 Function calls in control clauses

Function calls should not be used in flow control statements. For example:-

```
if( printf("hello world\n"))
{
    statements;
}

while( Get_result() )
{
    statements;
}
```

This is partly because of side effects, partly for debugging and also readability.

### 2.14.2 Null Statements

Null control statements shall be clearly written as shown below using braces to hold the empty or null line with a comment.

```
while( TRUE == Ri)
{
    ; /* Empty body Loop */
}
```

Some tools may require the following construct as they do not permit a comment after a ";" on an otherwise empty line

```
while( TRUE == Ri)
{
    ;
    /* Empty body Loop */
}
```

Note: Initially "NULL Loop" had been used in the comment but as NULL is already used in C it is less confusing to use "Empty Body".

## 2.15 Comments

Comments shall follow the style

```
/* single line comment */

/*
** multiple
** Line
** Comment
*/
```

The comment system // should not be used unless it is documented that ALL tools in the system will accept this method. Many do not.

In which case the system

```
// single line comment
```

```
//  
// multiple  
// Line  
// Comment  
//
```

Shall be used.

It shall not be permitted to mix both types of comment markers in a project. This excludes third party libraries. However where possible 3<sup>rd</sup> party libraries should conform to the guide.

### **2.15.1 Nested comments**

Comments shall not be nested.

### **2.15.2 Commenting out code**

Code shall not be commented out. Dead code that is commented out is dangerous as it is very easy to make it live again.

### **2.15.3 Comment density**

This is extremely difficult to mandate. The phrase often seen in coding standards is "Comments shall be meaningful" which is pointless and no help at all.

The current rule of thumb is 30% of the file by volume. However some code written well using Consts and defines may need few comments.

Other code may need many comments. What is obvious to the programmer may not be obvious to another programmer (or even the original programmer 6 months later).

The requirements for comments can only be found through code review. There is no magic formula or automatic answer. A metrics tool [see DAC] can tell you if you have more or less than your arbitrary level eg 30% but no tool can tell you if the comments make sense or if more (or less) are needed.

A manual code review is the only answer

## 5.0 Source Code Templates

There follows a set of file templates for several popular version control systems. These can be used as they are or modified to suit your own requirements. Their inclusion here implies no recommendation for or against. They are just the ones I have come across.

Microsoft Source Safe. PVCS from Merant/Intersolve, source Integrity from MKS and RCS from Component Software all use keywords in a similar way.

Rational's Clear Case does not have key words. This is due to the way the system works. Clear Case is designed to be used over distributed databases and networks. This permits safe developments over several separate sites (that do not even need to be on the same continent). I have used this system with a development team running into three figures on 10 sites on three continents. It was an embedded Unix project.

I have used all of the VCS listed here are one time or another. Each has it's own strong points and much will come down to how you get on with the interface. I would urge you to look at the levels of support and the costs. Also check for compatibility with the other tools you wish to use.

Source Integrity is available from [www.MKS.com](http://www.MKS.com)

Source Safe is available from Microsoft

PVCS is available from [www.merant.com](http://www.merant.com)

ClearCase is available from [www.rational.com](http://www.rational.com)

CS-RCS is available from [:-http://www.componentsoftware.com/](http://www.componentsoftware.com/) and, at the time of writing, is free for single users.

## 2.16 Source Safe Templates

```
/*
*****
** $Workfile: $
** Name:
** Copyright :Phaedrus Systems 1999
** $Author: $
** $Revision: $
**
** Analysis reference:
**
** Compiler :
** Target      :
**
** $Log: $
**
*/

/* ***** Include */
#define C00001_M

#include < system.h>
#include < thirdpartysw.h>
#include " project.h"

/* ***** Defines */

#define name value
#define name value
#define name value

/* ***** Globals */

/* ***** Prototypes */

static void function(void);

/* ***** Main */

void main(void)
{

}

/* *****
***** End of $Workfile: $*****
*****
*/
```

```

/*****/
/*
** $Workfile: $
**
** Copyright: Ltd
** $Author: $
**
** $Revision: $
**
** Analysis reference:
** Compiler :
** Target      :
**
** $Log: $
**
*/

#ifndef CO0001_H
#define CO0001_H

#ifdef CO0001_M
#define CO0001_SCOPE_ /**/
#else
#define CO0001_SCOPE_ extern
#endif

CO0001_SCOPE_    function_name();

#endif /*end guard */

/***** END of $Workfile: $ *****/

/***** */
/* Name:
**
** Purpose:
**
** Design reference:
**
** Input Parameters:
** Permitted range:
**
** Return Parameter:.
**
*/

```

## 2.17 MKS Source Integrity Templates

```
/*
** $RCSfile$
** Name:
** Copyright :Phaedrus Systems 1999
** $Author$
** $Revision$
** $ProjectName$
** Analysis reference:
**
** Compiler :
** Target    :
**
** $Log$
**
*/

/* Include */
#define C00001_M

#include < system.h>
#include < thirdpartysw.h>
#include " project.h"

/* Defines */

#define name value
#define name value
#define name value

/* Globals */

/* Prototypes */

static void function(void);

/* Main */

void main(void)
{

}

/*
** End of $RCSfile$
*/
```

```
/*
*****
*/
/*
** $RCSfile$
** $ProjectName$
** Copyright: Ltd
** $Author$
**
** $Revision: $
**
** Analysis reference:
** Compiler :
** Target      :
**
** $Log$
**
*/

#ifndef CO0001_H
#define CO0001_H

#ifdef CO0001_M
#define CO0001_SCOPE_ /**/
#else
#define CO0001_SCOPE_ extern
#endif

CO0001_SCOPE_    function_name();

#endif /*end guard */

***** END of $RCSfile$ *****

```

## 2.18 PVCS Templates

```
/*
*****
** $Workfile: $
** Name:
** Copyright :Phaedrus Systems 1999
** $Author: $
** $Revision: $
**
** Analysis reference:
**
** Compiler :
** Target      :
**
** $Log: $
**
*/

/****** Include */
#define CO0001_M

#include < system.h>
#include < thirdpartysw.h>
#include " project.h"

/****** Defines */

#define name value
#define name value
#define name value

/****** Globals */

/****** Prototypes */

static void function(void);

/****** Main */

void main(void)
{

}

/******
***** End of $Workfile: $*****
*****
*/
```

```

/*****/
/*
** $Workfile: $
**
** Copyright: Ltd
** $Author: $
**
** $Revision: $
**
** Analysis reference:
** Compiler :
** Target      :
**
** $Log: $
**
*/

#ifndef CO0001_H
#define CO0001_H

#ifdef CO0001_M
#define CO0001_SCOPE_ /**/
#else
#define CO0001_SCOPE_ extern
#endif

CO0001_SCOPE_    function_name();

#endif /*end guard */

/***** END of $Workfile: $ *****/

/***** */
/* Name:
**
** Purpose:
**
** Design reference:
**
** Input Parameters:
** Permitted range:
**
** Return Parameter:.
**
*/

```

## 2.19 Clear CASE Templates

```
/*
*****
** Workfile:
** Name:
** Copyright :
** Author:
** Revision:
**
** Analysis reference:
**
** Compiler :
** Target      :
**
**
**
*/

/****** Include */
#define C00001_M

#include <system.h>
#include <3rd party.h>
#include "project type defs"
#include "own module.h"

/****** Defines */

#define name value
#define name value
#define name value

/****** Globals */

/****** Prototypes */

static void function(void);

/****** Main */

void main(void)
{

}

/******
***** End of Workfile: *****/
*****/
```

```
/*
*****
*/
/*
** Workfile:
**
** Copyright: Ltd
** Author:
**
** Revision:
**
** Analysis reference:
** Compiler :
** Target      :
**
** $Log: $
**
*/
```

```
#ifndef CO0001_H
#define CO0001_H
```

```
ifdef CO0001_M
    #define CO0001_SCOPE_ /**/
#else
    #define CO0001_SCOPE_ extern
#endif
```

```
CO0001_SCOPE_    function_name();
```

```
#endif /*end guard */
```

```
***** END of Workfile: *****
```

```
***** */
/* Name:
**
** Purpose:
**
** Design reference:
**
** Input Parameters:
** Permitted range:
**
** Return Parameter:
**
*/
```

## 2.20 Component Software's RCS

```

//*****
//
//   $RCSfile:$
//   Copyright
//   $Author:$
//   $Revision:$
//   $Date:$
//
//   Specification Document :
//   Compiler:
//   Target
//
//   $Log:$
//
//***** Include
#define UNIQUE_FILE_ID

#include <system.h>
#include <3rd party.h>
#include "project type defs"
#include "own module.h"

//***** Defines

//***** Globals

//***** Prototypes
static void function(void);

//***** Main
void main(void)
{
    //Null statement
}

//*****
/
/           End of $RCSfile:$
//*****

//*****
```





## 6.0 Appendix B Lint and Example Program

The following program, BADCODE.C, is one of the example programs provided with our evaluation kits. This program has a lot of errors and is intended to demonstrate the error detecting and correcting capabilities of our tools.

Following are listings of the example program, output from the C51 compiler, and output from PC-Lint. The C51 Compiler detects and reports 12 errors and warnings while PC-Lint detects and reports 26 errors and warnings.

As you can see, the quantity and quality of the error messages reported by PC-Lint is greater than that reported by the C compiler.

```
/*-----
```

```
BADCODE.C
```

```
Copyright 1995 KEIL Software, Inc.
```

```
This source file is full of errors. You can use uVision to compile and  
correct errors in this file.
```

```
-----*/
```

```
#include <stdio.h>
```

```
void main (void, void)
```

```
{  
  unsigned i;  
  long fellow;
```

```
  fellow = 0;
```

```
  fer (i = 0; i < 1000; i++)
```

```
  {  
    printf ("I is %u\n", i);
```

```
    fellow += i;  
    printf ("Fellow = %ld\n", fellow);  
    printf ("End of loop\n")  
  }  
}
```

## C51 Output

When compiled with the C51 compiler, the BADCODE program generates the following errors and warnings:

```
MS-DOS C51 COMPILER V5.02
Copyright (c) 1995 KEIL SOFTWARE, INC. All rights reserved.
*** ERROR 315 IN LINE 10 OF BADCODE.C: unknown #directive 'incldue'
*** ERROR 159 IN LINE 12 OF BADCODE.C: 'typelist': type follows void
*** WARNING 206 IN LINE 19 OF BADCODE.C: 'fer': missing function-prototype
*** ERROR 267 IN LINE 19 OF BADCODE.C: 'fer': requires ANSI-style prototype
*** ERROR 141 IN LINE 19 OF BADCODE.C: syntax error near ';'
*** ERROR 141 IN LINE 19 OF BADCODE.C: syntax error near 'OOO'
*** ERROR 202 IN LINE 19 OF BADCODE.C: 'OOO': undefined identifier
*** ERROR 141 IN LINE 19 OF BADCODE.C: syntax error near ')'
*** WARNING 206 IN LINE 21 OF BADCODE.C: 'printf': missing function-prototype
*** ERROR 103 IN LINE 24 OF BADCODE.C: '<string>': unclosed string
*** ERROR 305 IN LINE 24 OF BADCODE.C: unterminated string/char const
*** ERROR 141 IN LINE 25 OF BADCODE.C: syntax error near 'printf'
```

C51 COMPILATION COMPLETE. 2 WARNING(S), 10 ERROR(S)

## PC-Lint Output

When the same code is parsed by PC-Lint, the BADCODE program generates the following errors and warnings:

```
--- Module: badcode.c
badcode.c 10 Error 16: Unrecognized name
badcode.c 10 Error 10: Expecting end of line
badcode.c 12 Error 66: Bad type
badcode.c 12 Error 66: Bad type
badcode.c 19 Info 718: fer undeclared, assumed to return int
badcode.c 19 Info 746: call to fer not made in the presence of a prototype
badcode.c 19 Error 10: Expecting ','
badcode.c 19 Error 26: Expected an expression, found ';'
badcode.c 19 Warning 522: Expected void type, assignment, increment or
decrement
badcode.c 19 Error 10: Expecting ';'
badcode.c 19 Error 10: Expecting ';'
badcode.c 21 Info 718: printf undeclared, assumed to return int
badcode.c 21 Info 746: call to printf not made in the presence of a prototype
badcode.c 23 Info 737: Loss of sign in promotion from long to unsigned long
badcode.c 23 Info 713: Loss of precision (assignment) (unsigned long to long)
badcode.c 24 Error 2: Unclosed Quote
badcode.c 25 Error 10: Expecting ','
badcode.c 26 Error 10: Expecting ','
badcode.c 26 Error 26: Expected an expression, found '}'
badcode.c 26 Warning 559: Size of argument no. 2 inconsistent with format
badcode.c 26 Warning 516: printf has arg. type conflict (arg. no. 2 -- pointer vs.
unsigned int) with line 21
badcode.c 27 Warning 550: fellow (line 15) not accessed

--- Global Wrap-up
Warning 526: printf (line 21, file badcode.c) not defined
Warning 628: no argument information provided for function printf (line 21, file
badcode.c)
Warning 526: fer (line 19, file badcode.c) not defined
Warning 628: no argument information provided for function fer (line 19, file
badcode.c)
```



## 7.0 References

This is the full set of references used across the whole **QuEST** series. Not all the references are referred to in all of the QuEST papers. Most of these books have been reviewed by the ACCU. the reviews for these books and about 3000 others are on <http://www.accu.org/bookreviews/public/>

**Ball** , Stuart. Debugging Embedded Microprocessor Systems, Newnes, 1998, ISBN 0-7506-9990-6

**Baumgartner** J Emulation Techniques, Hitex De internal paper, may 2001

**Barr**, Michael. Programming Embedded Systems in C and C++. O'Rilly, 1999, ISBN1-56592-354-5

**Beach M**, Embedding Software Quality Part 1, Hitex UK Available from [www.Hitex.co.uk](http://www.Hitex.co.uk)

**Berger**, Arnold. Embedded Systems Design: An introduction to Processes, Tools and Techniques. CMP Books, 2002, ISBN 1-57820-073-3

**Black**, Rex. Managing the Testing Process (2<sup>nd</sup> ed), Wiley, 2002, ISBN 0-471-22398-0

**Brooks**, Fred. The Mythical Man Month: Essays On Software Engineering, Anniversary Edition. Addison Wesley, 1995 ISBN 0-201-83595-9

**Brown** John, Embedded Systems Programming In C and Assembly, VNR, 1994, ISBN 0-442-01817-7

**Buchner F** Embedding Software Quality Part 1, Hitex DE Available from [www.Hitex.co.uk](http://www.Hitex.co.uk)

**Buchner F** The Classification Tree Method, Internal paper: Hitex DE, 2002

**Buchner F** The Tessa article for the ESC II Brochure Hitex DE, 2002

**Burden**, Paul. Perilous Promotions and Crazy Conversions in C, PR Ltd, MISRA-C Conference 2002. <http://www.programmingresearch.com/>

**Burns & Wellings** Real-Time Systems and Their Programming Languages, Addison Wesley, 1989, ISBN 0-201-17529-0

**Chen Poon & Tse**, Classification-tree restructuring methodologies: a new perspective IEE Proceedings Software, Vol 149 no 2 April 2002 pp 65-74

**Clements Alan**, 68000 Family Assembly Language Pub PWS 1994

**Computer Weekly** RAF JUSTICE :How the Royal Air Force blamed two dead pilots and covered up problems with the Chinook's computer system FADEC Computer Weekly 1997

**Cooling J**, Real-Time Software Systems ITC Press 1997 ISBN 1-85032-274-0

**Cooling J**. Software Design for Real time Systems ITC Press 1991 1-85032-279-1

**COX B**, Software ICs and Objective C, Interactive Programming Environments, McGraw Hill, 1984

**Douglas** BP Doing Hard Time, Developing Real-Time Systems with UML, Addison Wesley, 1999, ISBN0-201-49837-5

**Edwards**, Keith. Real-Time Structured Methods: Systems Analysis, Wiley, 1993, ISBN 0-471-93415-1

**Fertuck**, L, Systems Analysis and Design with CASE tools Pub WCB 1992

**Gerham, Moote & Cylaix**, Real-Time Programming: A Guide to 32-bit Embedded Development, Addison Wesley, 1998, ISBN0-201-540-0

**Hatton** Les, Safer C: Developing Software for High-integrity and Safety Critical Systems, McGraw-Hill(1994) ISBN 0-07-707640-0

**Hills** C A, Embedded C: Traps and Pitfalls Chris Hills, Phaedrus Systems, September 1999, [quest.phaedsys.org/](http://quest.phaedsys.org/)

**Hills** C A, *Embedded Debuggers* -Chris Hills & Mike Beach, Hitex (UK) Ltd. April 1999 <http://www.hitex.co.uk> & [quest.phaedsys.org](http://quest.phaedsys.org/)

**Hills** C A, Tile Hill Style Guide Chris Hills, Phaedrus Systems, 2001, [quest.phaedsys.org/](http://quest.phaedsys.org/)

**Hills** CA & Beach M, Hitex, **SCIL-Level** A paper project managers, team leaders and Engineers on the classification of embedded projects and tools. Useful for getting accountants to spend money Download from [www.scil-level.org](http://www.scil-level.org)

**Hills** CA & **Beach**, M. Hitex *C51 Primer* 3<sup>rd</sup> Ed, Hitex UK, 2006, Draft 3.5 is on <http://quest.phaedsys.org/>)

**HMHO** Home Office Reforming the Law on Involuntary Manslaughter : The governments Proposals [www.homeoffice.gov.uk/consult/lcbill.pdf](http://www.homeoffice.gov.uk/consult/lcbill.pdf)

**Johnson** S. C. Johnson, 'Lint, a Program Checker,' in *Unix Programmer's Manual*, Seventh Edition, Vol. 2B, M. D. McIlroy and B. W. Kernighan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.

**Jones** A History of punched cards. Douglas W. Jones Associate Professor of Computer Science at the University of Iowa. <http://www.cs.uiowa.edu/~jones/cards/index.html> see also <http://www.cwi.nl/~dik/english/codes/punched.html>

**Jones**, Derek. The 7+/- 2 Urban Legend. MISRA-C Conference 2002. <http://www.knosof.co.uk/>

**Kaner**, Bach & Pettichord, Lessons Learned in Software Testing, A Context Driven Approach. , Wiley, 2002 ISBN 0-471-08112-4

**Kernighan** Brian W & Pike , The Practice of Programming. Addison Wesley 1999 ISBN 0-201-61586-X

**Kerzner**, Harold. Project Management: A Systems Approach to Planning, Scheduling, and Controlling. (7<sup>th</sup> ed) Wiley, 2001. ISBN 0-471-39342-8

**Koenig** A *C Traps and Pitfalls*, Addison Wesley, 1989

**K&R** *The C programming Language* 2<sup>nd</sup> Ed., Prentice-Hall, 1988

**Lyons**. JL, Ariane 5: Flight 501 Failure. Report by the Enquiry Board , Ariane, 1996

**Maric B**, How to Misuse Code Coverage. Reliable Software Technologies, 1997. [www.testing.com](http://www.testing.com)

**Maguire**, Steve. Writing Solid Code, Microsoft Press, 1993, ISBN1-55615-551-4

**McConnell** Steve, Code Complete, A handbook of Practical Software Construction. Microsoft Press, 1993, ISBN 1-55615-484-4

**MISRA** Guidelines For The Use of The C Language in Vehicle Based Software. 1998 From <http://www.misra.org.uk/> and <http://www.phaedsys.org/>

**Morton**, Stephen. Defining a "Safe Code" Development Process, Applied Dynamics International, 2001

**Murphy**, Nial. Front Panel: Designing Software for Embedded User Interfaces, R&D Books 1998 ISBN 0-87930-528-2

**Pressman** Software Engineering A Practitioners Approach. 3<sup>rd</sup> Ed McGrawHill 1992 ISBN 0-07-050814-3

**PRQA** Programming Research QA-C static analysis tool. [www.programmingresearch.com](http://www.programmingresearch.com)

**Randel**, Brian. The Origins of Digital Computers, Springer Verlag 1973

**Ritchie** D. M. *The Development of the C Language* Bell Labs/Lucent Technologies Murray Hill, NJ 07974 USA 1993 available from his web site <http://cm.bell-labs.com/cm/cs/who/dmr/index.html> This is well worth reading.

**Simon**, David, An Embedded Software Primer, Addison Wesley, 1999, ISBN 0-201-61569

**Selis, Gullekson & Ward**. Real-Time Object-Orientated Modeling, Wiley, 1994, ISBN 0-417-59917-4

**Sutter** Ed. Embedded Systems: Firmware Demystified, CMP Books, 2002 ISBN 1-57820-09907

**Vahid & Givargis** Embedded System Design: A Unified Hardware/Software Introduction, Wiley, 2002, ISBN 0-471-38678-2

**Van Vilet** Software Engineering Principals and Practice Pub Wiley 1993 ISBN 0-471-93611-1

**Watkins**, John. A Guide To Evaluating Software Testing Tools (V3) Rational Ltd 2001

**Watson & McCabe**, Structured Testing: A testing Methodology Using the Cyclomatic Complexity Model

**Webster**, Bruce. The Art of Ware, Sun Tzu's Classic Work Reinterpreted, M&T Books, 1995 ISBN 1-55851-396-5

**Whitehead**, Richard. Leading A Software Development Team: A Developers Guide to Successfully Leading People and Projects, Addison Wesley, 2001 ISBN 0-201-67526-9

**Wilson**, Graham.. Embedded Systems & Computer Architecture, Newnes, 2002, ISBN 0-7506-5064-8



## 8.0 Standards

This is the full set of standards used across the whole [QuEST](#) series. These are Standards as issued by recognised national or international Standards bodies. Note due to the authors position in the Standards Process some of the documents referred to are Committee Drafts or documents that are amendments to standards that may not have been made publicly available by the time this is read.

### ISO

9899:1990 Programming Languages - C  
9899:1999 Programming Languages - C  
9899:-1999 TC1 Programming Languages-C Technical Corrigendum 1  
9945 Portable Operating System Interface (POSIX)  
9945-1 Base Definitions  
9945-2 System Interfaces  
9945-3 Shell and Utilities  
9945-4 Rational

12207:1995 Information Technology- Software Life Cycle Processes

14764:1999 Information Technology - Software Maintenance

14882:1989 Programming Languages - C++

15288:2002 Systems Engineering - System Lifecycle Processes

JTC1/SC7 N2683 Systems Engineering Guide for ISO/IEC 15288

WDTR 18037.1 Programming languages, their environments and system software interfaces —Extensions for the programming language C to support embedded Processors

### IEC

61508 :FCD Functional Safety or Electrical/Electronic/Programmable Electronic Safety -Relegated Systems

- Part 1 General Requirements
- Part 2 Requirements for Electrical/Electronic/Programmable Electronic Safety -Relegated Systems
- Part 3 Software Requirements
- Part 4 Definitions and Abbreviations
- Part 5 Examples of methods for the determination of SIL
- Part 6 Guidelines for the application of parts 2 and 3
- Part 7 Over View of Technical Measures

ISO/IEC JTC 1 N6981 Functional Safety and IEC61508: A basic Guide.

## IEEE

You may be wondering where ANSI C is... ANSI C became ISO C 9899:1990 and ISO 9899 has been the International standard ever since. See "A Standard History of C" in Embedded C Traps and Pitfalls

1016-1998 Recommended Practice for Software Design Descriptions

5001:1999 The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface

## NASA

<http://sel.gsfc.nasa.gov/website/documents/online-doc.htm>

SEL-81-305 Recommended Approach to Software Development Rev 3

SEL-84-101 Manager's Handbook for Software Development Rev 1

SEL-93-002 Cost And Schedule Estimation Study Report

SEL-94-003 C Style Guide August 1994 Goddard Space Flight Centre

SEL-94-005 An Overview Of The Software Engineering Laboratory

SEL-94-102 Software Measurement Guidebook Revision 1

SEL-95-102 Software Process Improvement Guidebook Revision 1

SEL-98-001 COTS Study Phase 1 Initial Characterization Study Report

## OSEK

Network Management Concept and Application Programming Interface  
Version 2.50 31st of May 1998

Operating System Version 2.1 revision 1 13. November 2000

**OIL: OSEK Implementation Language** Version 2.2 July 27th, 2000

Communication Version 2.2.2 18th December 2000

## BCS

Standard For Software Component Testing Draft 3.3 1997

## MOD

## Defence

## Standards

Def-Stan 00-13 REQUIREMENTS FOR THE ACHIEVEMENT OF TESTABILITY IN ELECTRONIC AND ALLIED EQUIPMENT

Def-Stan 00-17 MODULAR APPROACH TO SOFTWARE CONSTRUCTION, OPERATION AND TEST-MASCOT

Def-Stan 00-31 (obsolete) THE DEVELOPMENT OF SAFETY CRITICAL SOFTWARE FOR AIRBORNE SYSTEMS

Def-Stan 00-42 part 2 RELIABILITY AND MAINTAINABILITY ASSURANCE GUIDES PART 2: SOFTWARE

Def-Stan 00-54 Part 1 REQUIREMENTS FOR SAFETY RELATED ELECTRONIC HARDWARE IN DEFENCE EQUIPMENT PART 1: REQUIREMENTS

Def-Stan 00-54 part 2 REQUIREMENTS FOR SAFETY RELATED ELECTRONIC HARDWARE IN DEFENCE EQUIPMENT PART 2: GUIDANCE

Def-Stan 00-55 Part 1 REQUIREMENTS FOR SAFETY RELATED SOFTWARE IN DEFENCE EQUIPMENT PART 1: REQUIREMENTS

Def-Stan 00-55 Part 2 REQUIREMENTS FOR SAFETY RELATED SOFTWARE IN DEFENCE EQUIPMENT PART 2: GUIDANCE

Def-Stan 00-56 Part 1 SAFETY MANAGEMENT REQUIREMENTS FOR DEFENCE SYSTEMS PART 1: REQUIREMENTS

Def-Stan 00-56 part 2 SAFETY MANAGEMENT REQUIREMENTS FOR DEFENCE SYSTEMS PART 2: GUIDANCE

Def-Stan 00-58 part 1 HAZOP Studies on Systems Containing Programmable Electronics Part 1 Requirements

Def-Stan 00-58 part 2 HAZOP Studies on Systems Containing Programmable Electronics Part 2 General Application Guidance

**QuEST Series** (see <http://QuEST.phaedsys.org>)

## 2.20.1.1.1.1.1.1 **QuEST 0** Design and Documentation for Embedded Systems

**QuEST 1** Embedded C Traps and Pitfalls

## 2.20.1.1.1.1.1.2 **QuEST 2** Embedded Debuggers

**QuEST 3** Advanced Embedded Testing For Fun

**QuEST 4** C51 Primer

**QA1** SCIL-Level

**QA2** Tile Hill Embedded C Style Guide

**QA3** QuEST-C

**QA4** PC-Lint & DAC MISRA-C Compliance Matrix



[chills@phaedsys.org](mailto:chills@phaedsys.org)  
[QuEST.phaedsys.org/](http://QuEST.phaedsys.org/)