Embedded C
–
Traps and Pitfalls

By

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

Keil (UK) Ltd.

As presented at
ESS Conference, Olympia, London
24th May 2000

hitex
DEVELOPMENT TOOLS

Hitex (UK) Ltd.
Warwick University Science Park
Coventry, CV4 7EZ
Tel. +44 (0) 1203 692 066
Fax. +44 (0) 1203 692 131
www.hitex.co.uk
chills@hitex.co.uk

As presented at the
Embedded Systems Show and Conference, Olympia, London
24th May 2000

First version presented at

JAva C & C++ conference
Oxford Union, Oxford UK
Sept 1999
For the Association of C and C++ Users, see www.accu.org

The slides and copies of this paper (and subsequent versions) and the power point slides will be available at www.hitex.co.uk or www.phaedsys.org the authors personal web site.
Chris@phaedsys.org

This paper will be developed further.

Contents

Embedded C - Traps and Pitfalls

1. Introduction

Back in the bad old days, microprocessor programs were developed in assembler and blown into EPROM's such as 2708's, 2716's. With assembler, it was possible to know *exactly* what was in the EPROM. Know both in time (cycles) and byte by byte. Whether it functioned correctly was another matter! There was no possible way of knowing what the program was actually doing as it ran. Ingenious methods such as twiddling port pins were used to tell where the program had got.

The lucky few had ROM-based monitor programs whereby an array of seven segment LED's and a hex keypad allowed assembler to be single stepped and simple execution breakpoints to be set. In some instances, a serial connection to a computer or terminal allowed more flexibility. There were also some very rudimentary software simulators. An elite few, in the richest companies, were blessed with the ultimate tool the In-Circuit Emulator (ICE). However, such was the initial cost and subsequent unreliability of some of these early devices, that the ICE were often ditched in favour of good old monitors. The one-thing emulators did have in common with monitors was a strong assembler-orientation.

SW development systems were, to say the least, were basic. Many things now taken for granted were not possible. Colour syntax highlighting for one.

With great persistence and not inconsiderable brainpower, workable programs were produced. Whilst assembler is often specific to a particular micro many programs required the same services, serial comms for example, for this reason people developed libraries. Re-use was around long before C++!

Embedded systems moved from assembler to C. There were other languages such as PLM, Forth, Mod2 and Pascal but C is by far the most common, most versatile and well known. Apparently, C is used in around 85% of embedded systems with assembler used in around 75%. The assembler would be used in many C systems where accurate timing is needed and for things like drivers. There are fewer assembler only projects these days. Other languages are used but these rarely get over 10% market penetration.

If used properly, C is as robust and safe as any other high level language. That bears repeating: C, ***when used correctly***, is as safe and robust as any other HLL [Hatton].

As a spin off from the wide spread use of C in embedded systems there are many support tools, simulators, monitors and ICE that now support C source level debugging. This makes C an even more efficient way of producing embedded systems. I have been told that the C compiler is the most understood (and heavily tested) software on the planet. This in its favour.

Unfortunately, the History of C works against it. It is seen as a hackers language and has a reputation as a read only language. Part of this is due to the obfuscated C competition to produce the most unreadable and tortuous, but fully legal, C program. I did have C program that as a single (long) line. The main() line with an empty pair of braces{}. It would (with out a word of the text visible in the program) compile and print out to screen the whole of the 12 days of Christmas from the very strange executable parameters!

The other problem is that, just as everyone thinks they can write a book many think they can write a C compiler or debugger. Unfortunately, this means that there are also a lot of poor quality tools out there. Choose wisely.

As an example, I was once asked to set up (for a UK University) a cross compiler. It was a PC hosted Modula2-68K compiler. However, upon trying to install it I ran into problems. The compiler, it turned out, had been written in assembler. The company who produced it had virtually no documentation other than the users manual. There was no test suit or proof of testing. As the programmer had left the company (and the country), they were not able to help at all.

Thus, a cross compiler for safety critical use with a “safe” language (Mod2) was in fact a totally unsafe piece of software. Had I got the compiler up and running the University would have had no idea how unsafe the underlying construction was. It would have been used on several safety critical projects because Mod2 was a safe language.

There are many more C tools out there than Mod 2 ones. Therefore, there are likely to be much more poor quality C tools out there. So, be careful. Make sure of the pedigree of any embedded development tools you buy. As any mechanical, civil, aeronautical or electrical Engineer will tell you:

It is well worth buying good quality tools.

For this paper, I am only interested in the production of good C source code. I am not looking at how you got your design, CASE tools, how the teams were organised or anything of that nature. This is largely because the majority of embedded systems are on the small side and do not warrant CASE tools.

2. C History

The problem with C is its history. I do not propose to re-tell "The K&R Story" [K&R] here however there are some parts pertinent to this paper [Ritchie]. I recommend that people read the paper by Dennis Ritchie [Ritchie] this is available from his web site: http://cm.bell-labs.com/cm/cs/who/dmr/index.htm.

C was developed initially (between 1969 and 1973) to fit in to a space of 8K. Also C was designed in order to write an operating system. Unlike today, operating systems had to take up as little space as possible, to leave room for applications. This makes it idea for embedded systems.

C was developed from B and influenced by soup of several other languages. Interestingly BCPL, from which B was developed used // for comments just as C++ does!

One of the problems with C is that now the majority of people learn C in a Unix or PC environment with plenty of memory, disk space, native debugging tools and the luxury of a screen, and keyboard.

Because C was designed for operating systems it can directly manipulate the hardware and memory addresses (not always in the way expected by the programmer). This can be very dangerous in normal systems let alone embedded ones!

C permits the user to do many "unorthodox" things. A prime example is to declare 2 arrays of 10 items A[10] and B[10]. Then "knowing" that they are placed together in memory use the A reference "for speed" step from A[0] to A[19]. This is the sort of short cut that has got C a bad name.

The syntax of C and its lint with UNIX (famous for its terse commands) means that many programmers try to write C using the shortest and most compact methods possible. This has led to lines like:

```
while (l--) *d++ = *s++;
```

or

```
typedef boll (* func)(M *m);
```

This has given C the reputation for being a write only language and the domain of hackers.

As C was developed when computing was in it's infancy and there were no guidelines for SW engineering. In the early days many techniques were tried that should by now have been buried. Unfortunately, some of them live on.

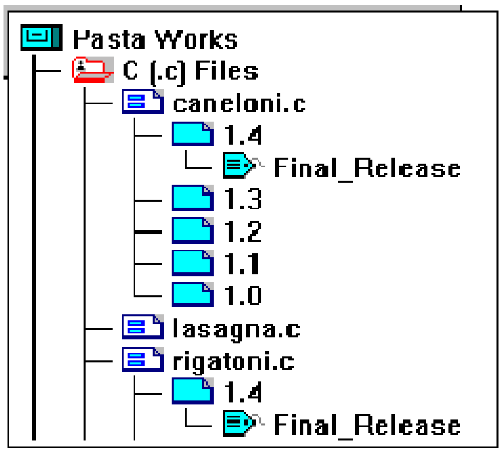
3. SW Engineering with C

As I and others [Hatton][Misra] [COX] have said, *when used properly,* C can be as safe as any other High Level Language. For embedded use there are some additional things one must think about. This paper, in looking at embedded C, will also cover many things that will be of use in general C programming.

As mentioned in the introduction I am only looking at the production of safe, robust C source code. How you got your design, pencil and envelope (50p) or CASE tool (£5,000) is not relevant here. Neither is the design method thought there should have been one.

3.1. Organise

First, organise your files. Both the code files and documentation. Version Control (or Revision Control System, RCS) has been around for many years, yet large number of engineers still do not use it. There are basic RCS/SCCS systems that run on a one machine one user basis up to the systems that can track files across linked networks round the world.

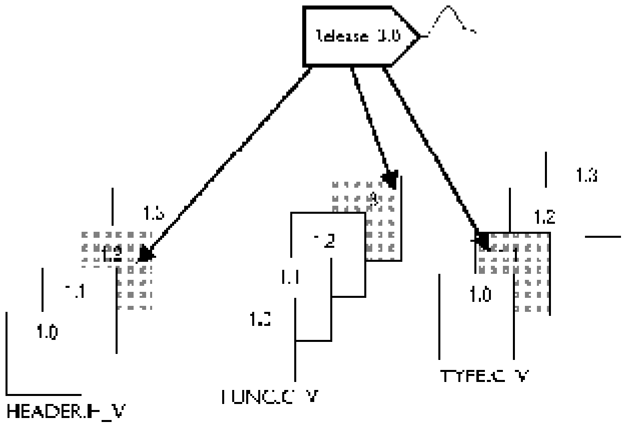


What is VCS? It is basically a database that will hold all the versions of a file. Thus when bugs are fixed or other changes made to a file both the original and the new versions can be stored and retrieved. Most VCS systems permit the labelling of file version and the ability to retrieve all files associated with a label. Usually multiple labels can be assigned to a file. Thus a standard module can be used in several projects.

When linked with a make system it gives the ability to "make Release_1".

VCS means that you never loose a file and can recover any version of a file and therefore create any version of the software.

This can be very useful when major changes are added to a file for the wrong reason and need to be removed.



The other very good use of VCS is that it permits developers to get on with the next version without affecting the current builds. IE one can set up the VCS to let the test team to get the version of the file released for testing. Then there is the choice of fixing the bug in the version the developer is working on or branching the file to give another copy (still tracked by the VCS and linked to the original file). Most VCS systems permit the merging of branches later. This is usually a semi-automatic procedure.

VCS also stops two people accidentally working on copies of the same file. It usually requires several intentional acts to get a second write-able copy out and several more intentional acts to check it back in in-place of the original version pulled out. It is, of course, fully logged and the files can all be recovered.

Many modern C development systems have hooks in them to interface to the VCS systems so that once set up they become transparent to the developers. Due to automatic time and date stamping in VCS systems, no matter how slack you, or some one else gets, you should be able to, this is the part ISO 9000 people like, show a complete audit trail from the day the file was created.

VCS systems cover all sizes of project. I have seen an extreme case where a very large embedded project consisted of several linked systems produced by a couple of hundred engineers across 9 sites in 4 countries. The VCS system (Clearcase from Pure Atria) could not only track all the files but also synchronise all 9 of the databases automatically. This meant that all the developers and testers were always working on the correct (but not necessarily the latest) versions of software. This system was able to cope with two changes of target CPU architecture! All the reusable Sw modules were kept and moved (with their history) to the “new” project.

At the other end I have used a simpler system (PVCS from Intersolve) that I found to be very useful on a single machine (or small networks) running one or several projects. The PVCS suite can also integrate make and bug reporting to full ISO9000 and CMM requirements.

These systems do cost money and take time to set up, but are worth their weight in gold when a customer wants a mod done to a project you last worked on 2 years ago. The other nightmare scenario is where the customer wants a mod and the code has since modified the code for something else….

So, we now have a project where we have organised files where we can get at any version and easily build any version of the system. Incidentally, this also helps with testing as test scripts can be held in the VCS in the same way and any test suite rebuilt. The only thing to watch out for is these systems store deltas of text files. Most let you baseline and start a new set of deltas but for the storage of non-ASCII files they usually make complete copies. This takes up a hell of a lot of disk space so be warned!

What we now need is something in the files we have organised.

Before leaving the RCS completely there is one last point that goes into the next section. The RCS systems can usually insert into the source files things like current version, change log, file names, paths to archive, author etc. In the example shown (PVCS) it is the text between the “$” delimiters. This insertion is automatically done by expanding these keywords. In this example the whole history block after the $log is also added automatically.

This automatic insertion means that a simple template is all that is needed for modules. The developers do not need to complete it, well only the odd line, as the VCS system does it for them.

In the example below the file name, author, revision, and history log are automatically inserted. So even with the tardiest of developers an ISO9000 audit trail is automatic. As the log uses the login name specific to the user it will be obvious who did not correctly complete the header block.

```
/*****************************************************
** $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 $Workfile:   U1CO0001.C  $  *************/
```

3.2. Good C

Now having organised all the files we need “Good C” in them. What is good C?

It must not contain errors
It must perform as expected.
It must be repeatable
It must be readable (so people can see what it is doing)

These seem simple enough and may at initially appear to overlap.

Firstly, the C should not contain any syntactical or semantic errors. This is not always as obvious as one might think. Syntactical errors the compiler picks up but semantic ones can be far more subtle. They can also be a lot more difficult and time consuming to find if left to the test and debug phase to find. These should be found as the source code is written using static analysis, not the compiler. The compiler should always be set to the highest level of warnings.

After the syntactic and semantic errors are removed does the code it do what you expect? It is of no use having a technically correct program that does the wrong thing! This is usually the case of understanding the requirements or quite often things like testing for “greater than” when it should have been “equal or greater than” This can usually only be found by visual inspection (code review) and thorough white box testing. For catching errors during code inspections, the code needs to be readable. This is something I will return too later.

Repeatability is one thing that is often overlooked when testing software. Most software (and embedded in particular) often has to perform the same tasks many times, sometimes for years on end. I have used a program that ran well for a while (3 months). It then crashed but after a reset, it ran again (for about 3 months). It was, under some situations, over writing buffers. Unlike desktop PC’s embedded systems have to be reliable as they do not have a ctrl-alt-del. Also, embedded systems often control machinery. Malfunctions in robots making cars in Japan killed 6 people in one year!

The last point on the list is that if you can’t *easily* read the code you will not be able to check for errors and the correct running of the code. There are two parts to this firstly the code should be using the correct constructs in a safe manner. The best place to start is with the international standards and then the industry specific ones. In this case the international standard is ISO C. (Not ANSI which is a local USA standard). A de-facto standard for c usage is the book “C Traps and Pitfalls” by A Koenig. I will look at the embedded parts of C later. Secondly, to be easily readable the source needs to be uniform in appearance.

You may well (or at least I hope so) agree with the first part of the last paragraph but may disagree with the last line. Many people do not like to be told how to lay out their code. Many see it as an infringement of their civil liberties. However, sw *engineering* is a branch of engineering not a mystical science!

3.2.1. Style

It is easier to count a group of people if they are standing in lines of 5 and blocks of 25 than if they are just standing in a group. Likewise when the source code is laid out is a standard way is it far easier to spot anomalies and errors.

As one of my team one said to me in 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                  M
```

It took me a while to work out what it actually said was:-
ThEoThErTeAmReGuArDsOrUcEc0DeLaYoTaSaNaRtFoRm

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!

The illustration above should have convinced you that a uniform style to a set convention is a good idea.

There are many style guides about. Have a look on the internet or create your own. Which ever you use do so consistently. NOTE:- (and this is important) Style is about *readability*. It is easier to spot mistakes if something is easy to read. Style guides are not about safe code as such or safe subsets of C.

We now have a religious style debate as to where to put the braces. There is no "true faith". Any *system* will do as long as you stick to it! Some of the more common are styles are;

3.2.1.1. (K&R)

```
If( xyz){
        statement
         statement
}
```

3.2.1.2. (Indent)

```
if(xyz)
        {
        statement
        statement
        }
```

3.2.1.3. (Exdented)

```
if(xyz)
{
        statement
        statement
}
```

A fuller description of each is given in appendix A. Personally, the exdented is my preferred style but some of the older debugging tools require the first style.

The only thing I insist upon for braces is that they are used wherever they can be used. This is especially important on things like if clauses for example

```
interlock = OFF;

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

if(ON == interlock)
        open_doors();
else
        apply_breaks();
        sound_alarm();
```

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

I insisted that all code produced with teams I am involved with rigidly adhere to the principal of always using braces were possible. This may sound a bit draconian but I have good reason.

I instigated this rule after three of the 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 error was combined with another similar “non error” to produce a real problem much further away.

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_breaks();
}
sound_alarm();
```

Whereas what was meant was:

```
interlock = OFF;

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

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

This is actually based on a real problem on a rapid transit system in the far east..... written by programmers in the Midlands! It actually made it as far as the test runs. It was only found by accident after a carriage broke down and the test train ran with one fewer carriages than normal.

3.2.1.4. Information blocks and comments

Comments, or the lack of, are one of the most hotly argued things after where to put the braces! Each file or module should have an information block.

```
/*****************************************************
** $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
**
*/
```

This is an example that was used in a project under ISO9000. Some of the significant points are the Analysis reference to tie the source to the design and the history log. This should give the developer all the information on where the file started and who it got to it's current state. The history block in this case is automatically put in by the VCS. Where a VCs is not used it should be manually maintained.

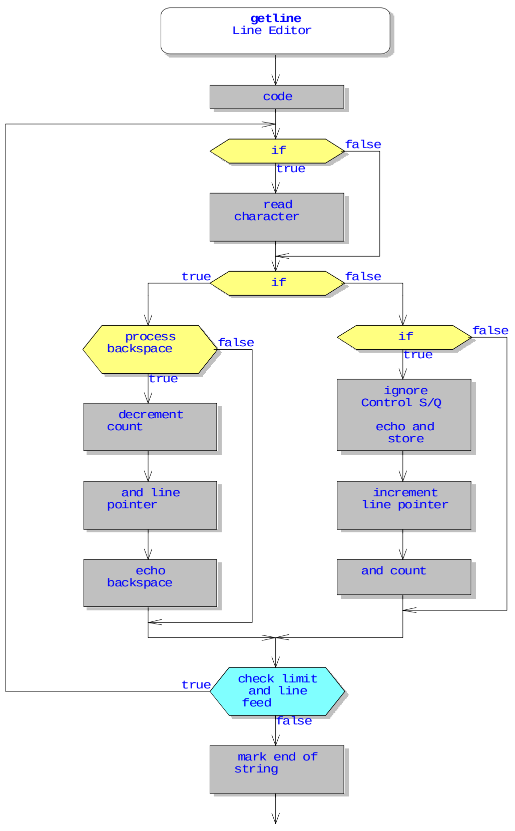
Each function should also have a simple comment block giving the purpose of the function, the input and output parameters. Like the main file information block, where appropriate, the reference to the design or requirements should be included. This may sound like a lot of extra work but I have found that it makes one focus on why the function is there.

```
/******************************************************************************* Convert_One */
/* Name: Convert_one
**
** Purpose: Converts Faranhit to Celsius
**
** Input Parameters
** Return Parameter
**
*/
```

The complexity of the function block can be adjusted from a standard template to suit the function. A simple function to add two numbers and return the answer will need less than a function that manipulates several parameters and outputs to (or in puts from) a peripheral. It is a moot point if the function comment block should or should not contain all the information on the algorithms etc used in it. This will depend on how good your documentation is. Hopefully a reference to the design document is all that is required. The only time I put (almost) as much comment in the code as code was when the source had to be self documenting as it was going in to the public domain without any other supporting documentation.

The one thing I do find useful is at the start of each function to have a single line comment full width on the page with the function name at the right hand end. This makes finding functions easy when scanning listings.

The biggest controversy is where to put comments in the source (if at all). Someone asked the question "where should I put comments in a program" recently (August 99) on one of the C news groups on usenet. The thread attracted an order of magnitude more replies than any other (except the one whether C++ was suitable for embedded use). I scanned the thread but there was no clear winner or consensus. The suggestions went from commenting every line to the idea that well laid out code needs no comments at all.



The answer is somewhere between the two. I use Development Assistant for C. This can take source code and automatically produce metrics and flow charts. The structure of the flow chart comes from the source, the analyser reads the if, switch, do while clauses etc. However the program takes the comments in the code to put in the boxes of the flow chart. I use this program as a guide to where I need comments. The rule of thumb is that comments should be 30% of the file. This figure is only a guide. Comments should be used to make the source readable by *another person* not the original developer.

The only strict rule is that comments should not be nested. As an extension of this code should not be commented out because it could contain comment blocks. Also commented out code is confusing and can be, due to the was C nests comments, not the commented out block the developer thought it was.

All comments should use the /* */ pair not the c++ style of //. This is because not all C compilers support the C++ style even if BCPL did!

3.2.2. Header files.

Header files contain all sorts of things such as defines, macros, and function prototypes that are required in several files. The standard libraries have header files that are usually included. These are included into the source using the #include directive. This is a straight textual insert. What can do wrong? Lots of things

<u>Header files should not be nested.</u>

Nesting hides files. On one project I worked on in one file there were 8 include files. However, these files had nested files. When I unravelled, the nested headers there were over 120 included files. This included 8 files ten times! The interesting point was that when the duplicates were removed the source file would not compile! It appeared that due to some problem no one could be bothered to find three of the files had to be included twice. Once at the start of the includes and once at the end. This was masking a serious problem.

For example something may be defined in one header, turned off in another and as the two get repeatedly included parts of the overall included files will have the define in and others not. This can have some very strange , and almost impossible to find effects.

In order to make sure only one of each header is ever included guards should be used. These guards take the form:

```
#ifndef name
#define name

header file contents

#endif
```

Generally for *name* I use the name of the header file.

An additional tip is to always include header files in the same order. I usually start with system headers at the top.

Incidentally when putting macros into headers use parenthesis enthusiastically. To ensure there are no silly side effects. I.e. the macro is self-contained. Macros should only really be used for constants and function like macros. Using a macro for something like :

```
#define STARTIF  IF(
```

should be avoided

3.2.3. Magic Numbers

Defines & magic numbers. I hope I do not need to tell people not to use magic numbers, #defines or const should be used. The advantage of using const is that it will be visible in debuggers. The disadvantage is that it will actually take up space in ROM or RAM as a (const) variable.

So whilst const is preferable technically, pragmatically #defines may be better in 8 bit systems like the 8051 where RAM is at a premium..

3.2.4. Flow Control

Controlling the flow of a c program has always caused problems. People always tend to take the shortest route. It has already been mentioned that IF clauses should always use the braces {} even when there is only a single statement. This holds true for while statements as well:

```
while(a<count)
        a++ ;
```

is potentially dangerous. It should always be written:

```
        while (a<count)
        {
                a++;
        }
```

3.2.5. If

Something I have found effective in tests for equality if to have the fixed or constant value on the left. The variable on the right. This causes an error if, inadvertently, the test for quality is inadvertently changed to an assignment. For many years, I have writtten

```
        if(constant == variable)
```

This is counter intuitive and does take a while to get used to. However, it does stop many silly errors. Though with a good compiler and rigorous use of lint any errors of this type should be picked up whichever way it is done.

Logic is one of the problems when using if with else. Where if else if is used there must always be a final else clause. This should be done even when the final clause will be empty! A comment should be placed in the final else to say why it is empty.

```
If( Clause)
```

```
{
        statement;
}
elseif(clause)
{
        statement;
}
else
{
        statement;
        /* or comment*/
}
```

This makes it clear why the clause is empty and makes the developer think about the structure of the whole construct.

3.2.6. Switch

Switch statements are completely straightforward, what can go wrong? Lots of things can go wrong! All switch statements must have a default clause. If there is no default action put an error message in there. This saved a lot of grief more than once when (completely unexpectedly) the error message showed up! Break should be used to terminate each case.

```
Switch(variable)
{
        case 1: statement;
                break;

        case 2: statement;
                break;

        case  3:
        case  4: statement;
                break;

        default:
                printf("ERROR!!!\n");
                break;
}
```

Notice the break on the default. Always a good idea just in case the default becomes a case.

3.2.7. Breaking the flow

In a word don’t. Break should only be used at the end of every case and default clause in a switch statement and no where else.

Goto….. This needs no comment as it should never be used. Neither should continue.

Whilst on the subject of jump out of a flow it is a moot point whether there should be a single point of exit from a function. Many say there should only be return. Others say that for readability and sensible flow in a function more than one is better. I have seen cases where the function contained some horribly complex if else if, constructs in order to get one return line whereas it was far cleaner, elegant, shorter and faster with several return statements.

3.2.8. Static linkage

By implication, all functions are externs. However, where a function is only called within the module it is in it can be made static. This has a couple of uses.

Firstly it makes the code safer in that visibility can only be in the module. This means that the function can only be called in the module. When used with static variables declared at file level it makes the variable public to only that file. This rather like the private functions in a c++ class.

Secondly static functions can result in faster code. This is because the compiler knows where the function can be called from. Generally this will be a local, short or relative jump within the file. This will be tighter and faster than a general jump to “somewhere else” out of the file. For embedded use this has the advantage in saving a few bytes per call.

3.2.9. Declaration and initialisation

Variables should be initialised before they are used. In keeping with the general rule of explicit rather than implicit variables should be explicitly initialised as soon as possible. The obvious and most sensible time is when they are declared.

```
static int count = 0;
static signed char  letter =’a’;
```

This ensures that all variables are initialised. Variables should always be declared initialised at the start of a file or function.

3.3.How to prove it is Good C

When “The Gods” [K&R] gave the world C and the world bathed in the brilliance of the language it rather overshadowed another program that “The Gods” had left for the disciples. Those who know it use it religiously. Many, to their cost, leave it by the way side. What is this program? Lint. It was first developed from a C compiler engine in 1979 by Steve Johnson [Johnson] who was one of the original group that worked on C and UNIX.

In his paper on C Ritche [Ritche] says- “To encourage people to pay more attention to the official language rules, to detect legal but suspicious constructs, and to help find interface mismatches undetectable with simple mechanisms for separate compilation, Steve Johnson adapted his Pcc compiler to produce lint.”

Whilst a compiler will very accurately check *syntax* it does not worry about the *semantics* as long as they are legal. Now legal does not mean sensible or safe. A compiler may not care if you want to store part of an int or float in to a char. The fact that it makes no functional sense is irrelevant to the compiler!

To give an example from English language. In a meeting I attended a person (who’s first language was not English) said “I have over seen that.” What he meant was “I have over looked that.” The meaning he gave was I have personally checked that whereas what he meant was “I forgot about it.” Lint will do much the same sort of thing for C.

Going back to the dubious if statements used to illustrate the need for braces:

```
1       interlock = OFF;
2
3       if(TRUE == stop)
4               flag = ON;
5               interlock = ON;
6
7       if(ON == interlock)
8               open_doors();
9       else
10              apply_breaks();
11              sound_alarm();
```

If lint was run over this code it would complain that lines 5 and 11 had incorrect indentation. In fact on PC-Lint it complained:

```
“Warning 539: Did not expect positive indentation from line 5”
“Warning 539: Did not expect positive indentation from line 11”
```

If you refer to the PC-Lint manual it gives an if statement without braces as the example!

Another subtler problem on the same lines, from the lint manual, is:

```
if(….)
        if(….)
                statement
else
        statement
```

The else is in fact part of the second if, not the first. This is where a good style guide is useful so the code is uniform and insisting that braces be used on ***ALL if, do and while clauses.***

Since lint was developed, there have been great strides in static analysis. This is where the analysis of the source code without compiling it or running it. HP has estimated that static analysis and code inspections are 5 times more efficient than white or black box testing. Alcatel have said that static analysis can reduce a project time by up to 30%, primarily from the debugging and fixing stages. The cost of fixing a source code error (other than syntax) rises exponentially the longer it it left and the further down the development it goes. Thus the most cost effective way of fixing sw errors is at source when the engineer is writing the code.

I use the "write and lint" cycle instead of the more common "write and compile" cycle to check the code.

Lint is not the only tool. It was the first one for C and in keeping with it's Unix/C roots is a simple command line program. Other heavyweight static analysers (that are also vastly more expensive and time consuming to set up)

Apart from using lint ***always run the compiler on it's highest level of error checking***.

MISRA-C is a very good set of rules for using C in safety critical embedded situations. PC-Lint has a configuration file to test for as many of the MISRA-C rules as it possible statically. MISRA-C also shows how to construct a conformance chart. I would recommend this to any developer who needs to show "due diligence" or prove that the system has been tested.

3.3.1. Formal Eastern European Writing

There can not be a discussion on safe C, "proper C" without someone bringing in the two methods for producing perfect code. These methods in *theory* are very good. They are usually put forward by theoreticians. Their practical use in real sw engineering is another matter.

Both the methods should, in theory, eradicate many errors and mistakes but, in my view, create more than they solve.

3.3.2. Hungarian Notation

This is a method were the name of a variable conveys information as to the usage and the type of the variable. This method sounds wonderful until a type is changed part way through development. It also does not help readability. There are also several standard notations in use and countless local ones. This breeds confusion.

An example of Hungarian notation from Steve Mconnells Code Complete. A book well worth reading

```
For(ipavariable = paFirstvariable; ipavariable <= paLastvariable; ipavariable)
{

}
```

further examples:

ch	a variable containing a character
ach	array of ch
ich	index to an array of ch
ichMin	indest to first character in array
ppach	pointer to pointer to array of ch

```
mhscrmenu
 m      module level
   h    handle
     scr  to a screen region
        for a menu
```

This is logical BUT I have found that these methods usually cause far more trouble they are worth. There appears to be no general standard. After you learn, one some one changes it. I have seen a project where it was 2 weeks before the team discovered that some of the team, whilst using the same letters were using them to signify different things to the rest of the team!

3.3.3. Formal Methods

Formal methods are a almost a mathematical way of describing a program. They are NOT a programming language though there are some interpreters for at least two of the languages (Z and VDM)

The problem is that there are two interfaces. One takes the specification and turns it into Z or VDM and at the other side the conversion from the formal method to the programming language of your choice.

The interesting thing is that due to the absolute certainty of people in these methods it can cause problems. Folk history has it that a well known CPU vendor used formal methods in its chip design, for the microcode. When several thousands of these chips were produced (at a cost that could bankrupt many companies even now) they were shocked to discover a bug!!!

It transpired that an error was introduced in the translation between the requirements and the formal methods. The formal methods were OK as was the translation to silicon. The problem is that it is generally the mathematicians who like the formal methods.

What do formal methods look like? This is VDM

Max(s:X-set)r:X
Pre $s \neq \{\}$
Post $r \in s \cap \nabla j \in s.r \leq j$

This is a function to find the largest element in a set. I have managed most of it as even with the might of Windows 95 and Word 97 I do not have all the correct symbols!

As I said, wonderful in theory (especially among mathematicians) but a nightmare in practice.

4. Embedded Engineering

Embedded Engineering whilst having many similarities with “ordinary” SW Engineering is different. Different that is to “ordinary” SW Engineering and every other embedded project. Whilst embedded systems share many similar attributes no two are the same.

In general, embedded systems tend to be single task. Albeit that the larger systems may have an operating system and many processes running. They usually have to meet deadlines that are far tighter than general-purpose systems. This is because they often have to react to inputs that are measured in microseconds not seconds or minutes.

Another difference that is crucial is that embedded systems are usually built to a minimum cost with little or no room or even facility for any expansion. That is unlike the desktop PC the resources such as memory are cut to the minimum required for the job and sometimes . This is because, usually, the embedded system is a small part of a larger system. The control system is ancillary to the main function of the system for example a microwave cooker. If additional recourses are added the cost of the product goes up or the profit goes down.

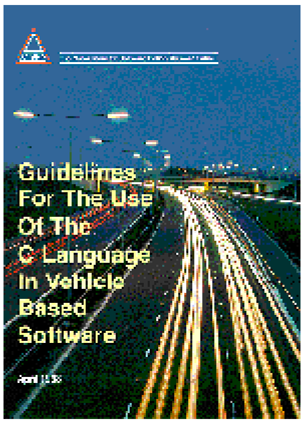
The problem, for the programmer, is that often memory saving techniques have to be used. However the most dangerous problem is memory leaks. I once worked on a comms system where a series of line connections caused a byte of memory to be lost. One of the engineers did some calculations and worked out that the unit would fail in 2 to 4 years of use. The other problem was that depending on usage that the unit had the failure symptoms could be wildly different. The unit had a lifetime of 10 years and many units would be in remote sites.

There is one other very important aspect of embedded systems they tend to be used with mechanical equipment that moves. Whilst not all embedded systems are safety critical many are. The others will cause problems if they fail that are usually more of a problem than having to reboot a PC

5. Embedded C Engineering

Now we have reached the significant part. However, this section will be surprisingly shorter than many people expect. I hope that it will only be a surprise to those who have skipped the previous sections. Good SW Engineering practice and good Embedded Engineering Practice should result in the correct approach for Embedded C! There only a few additional tweaks needed for embedded C.

Just as there are good style guides for C there are useful guides for a ***safe subset*** of C for embedded use. Note: This is NOT a style guide.



The de-facto standard is MISRA-C [MISRA]. This was originally developed for the automotive industry. This industry uses 8, 16 and 32 bit processors from many families. Thus, the guide is suitable for virtually any embedded system which is why it has gained wide spread acceptance across the embedded world. Coupled with a good style guide MISRA-C will help you produce robust and safe C. PC-Lint now has a configuration file to test for many of the rules (it is not possible to test for all 127 rules statically.

A style guide, MISRA-C and PC-Lint between them should let you produce embedded C code that is safe, robust and readable. What more could you ask for? You will find most of the points covered in the following section in MISRA-C and the PC-Lint Manual.

5.1. *SIZE MATTERS !*

As stated previously embedded systems tend to be of fixed size fore the lifetime of the item. In addition, memory and resources cost money. Memory tends to be the most expensive component in an embedded system. Designing in the additional chip, tracking the larger board that will cost more to make all add to the costs. In many cases a single chip MCU is required (due to space and costs) therefore the memory available will be physically fixed. Space is expensive and restricted so make good use of it.

I know of one project where there was a white board where the memory available was displayed down to the last bit! (This was a specialist system where only a specialised purpose built MCU could be used.) There was actually bartering between the developers for the memory.

Memory usage can be decreased by several ways.

Use appropriate sizes of data types. This may sound obvious but it is surprising how many people make incorrect assumptions. I have been told that short is more portable than an int and that a short is always 8 bits. Neither statement is correct.

Typedefs should be used for the size of data the types. Furthermore, all types should be explicit not implicit. By which I mean that there should not be a "char" but either an unsigned char or a signed char. Do you know, off the top of you head, if your compiler defaults to signed or unsigned char?

I use the typedefs shown below. Note I have not used "int" in any of the names. This is because much of the time it is a container for data rather than an integer.

```
typedef unsigned char       U8;
typedef signed    char      S8;

typedef unsigned int        U16;
typedef signed     int      S16;

typedef unsigned long       U32;
typedef signed    long      S32;
```

These typedefs are placed in a short universal header that is included in all files. Short should only be used where it is larger than a char and smaller than an int. For example:

Char	= 8 bits
Short	= 16 bits
Int	= 32 bits
Long	= 64 bit

The use of lint with strong type checking enabled will cause warnings to be issued where one type is assigned to another even if the underlying type for both is an int.

5.1.1. Enumerated Types

Another place where space is wasted is in enumerated types. The ISO standard says that an enum is 16 bits. This can waste a lot of space on 8 bit systems where there are the enum only has a few values. There are several ways round this. Some 8 bit compilers will use a char to hold enums where possible. Another way is to use #defined values. This can also speed things up on 8 bit systems as the define and the 8 bit emun both fit in a char. Manipulating chars is much faster than 16 bit data on an 8 bit system.

The use of lint can improve memory use. When used across all the files in a project lint can pick up globals that can be declared as locals, variables public to a module that can be local and unused variables.

5.1.2. One way of NOT decreasing the code

```
If( get( start +offest1 + calc_offest(start+offset2))
{
code statements;
}
```

This will generate almost the same code as

```
temp1  = start + offset1;
temp2  = start + offset2;
temp2  =  calc_offset(temp2);
temp1  =  temp1 +temp2;
temp1 = get(temp1);

if(TRUE == temp1)
{
code statements;
}
```

In this case, one compiler I tested this on the difference was 4 bytes. Whilst I have said that saving even 4 bytes is a good idea in this case it is not. This difference can depend on how the compiler does temporary variables at the point in question. This depends on the compiler, the architecture of the MCU and what else the program is doing prior the point in question.

The difference in source layout is that using a source level debugger or ICE can stop on each line to the second example and look at the temp values. In addition, each stage of the calculation of temp1 can be checked. With the first example, all that happens is that whole line appears to be executed in a single step. One can of course drop in to assembler to step through but that breaks the flow of debugging and the whole point of a source level debugger.

5.2. Volatile

Volatile is a useful keyword for embedded work. Volatile means to a compiler "this could change without the program touching it" this is essential in embedded programming.)

I have recently seen a very fast memory test routine. It was so fast because the compiler optimised out a variable as it was not being changed between assesses. What it did was a memory test without ever touching the memory!

5.3. Const

Const should be used where something is not going to change. One place it is very useful is in function parameter declarations. For example

```
static U8  func( const U8 name, const U8 number);
```

This will have both the compiler and lint screaming if the function tries to change the values.

Interestingly enough it is possible to have a const volatile. The application can not change the const but it will change in between accesses without the application touching it.

5.4. Register

Register is often used to speed things up. However is usually has the opposite effect! This is because the compiler is very good at shunting things in and out of registers and memory. Also it is only advisory, many compilers ignore it. In those that do not forcing a variable into a register could give the compiler problems sorting the other variables. In the end, you get a slower system.

5.5. Dynamic memory

The dynamic allocation of memory is a dangerous at the best of times as it can lead to memory leaks. In embedded systems, it could be fatal in a very real sense. Much of the dynamic memory allocation routines are not fully specified in the standards and their behaviour (intended or otherwise) is dependable. I have seen a system where a one byte leak in unusual circumstances would cause the system to become unstable after 3 years. The system had a 10-year life.

5.6. Libraries.

There are three types of library. Your libraries and some one else’s library.

Your libraries it goes without saying should be constructed to the highest standards and thoroughly tested.

When it comes to some one else’s they can be divided into those you have source for and those you do not. Where you have the source, you should rigorously lint them, even those supplied with the compiler. Where necessary re-write them. I have come across libraries supplied with compilers in the past that did not survive static analysis! In fact, the produced illegal code under some circumstances. It was discovered by a deep flow static analyser.

5.7. Tuning Libraries

In the case of some libraries, a module may contain a mixed bag of functions. It may well be worth re compiling the libraries leaving out the functions that are not required.

Another thing to look out for is a library with general-purpose functions. For example, the printf function is large and complex. You probably do not need much of the function. It may pay in the end, to write your own print function that only handles the formatting that you need. This will save space, improve speed and you know how they work. It pays to re-do most of the functions that can accept variable numbers of parameters.

5.8. Maths

Maths is one major problem in embedded systems because the maths libraries tend to be large. There are ways around this. For some things like sin and cos a look up table can be used. Whilst this takes up space this is data not code (highly relevant to an 8051 architecture) it can run much faster than a function that actually calculates the sin or cos.

Another way of speeding things up is to revert to fractions. To get 75% of 100 look at as ¾ or 100. Take 100 do an integer divide by 4 and an integer multiply by 3. No floats are needed.

Incidentally never do comparisons with floating point numbers. Rounding errors can play havoc.

The other taboo is playing with the bit fields inside a float. There is no global standard defined. There may be a de-facto standard but it is not worth the risk.

6. Embedded C++

As C++ (and OO) became the in thing in the "normal" programming world so C++ is becoming sought after in the embedded world. This is to some extent fashion. I have seen many people looking for C++ compilers for the 8051 "because C++ is better than C".

People also want to use C++ compilers when writing C because "C++ is a superset of C" This was true many years ago however the two are now distinctly separate languages. Having discussed the matter with members of the UK ISO C and C++ standardisation committees I understand that there are come parts of C and C++ that have the same syntax that mean different things. So, do NOT use a C++ compiler for C.

I believe that C++ is too large for 8 bit systems and indeed many architectures (8051 for example) do not suit the language. C++ is also in its infancy for embedded 16 bit systems. I have found that C++ works in 32 and 64 bit systems. Indeed Embedded C++ (EC++) is being primarily aimed at 32 bit systems and will obviously work on 64 bit systems. I can not see any real incentive to develop EC++ "backwards" in to the smaller CPU.

Of course, compiler vendors would like everyone to go from C to EC++ as this will mean more compiler sales. This leads on to the other problem with C++ that the development tools (and particularly the debug tools) will be very complex and expensive. If you are on a cost conscious project, and who isn't, the buzzwords OO and reuse.

Just as C was a step away from the hardware compared to assembler C++ is a step further away. EC++ will need far more RAM than it's C counterpart. However C++ also creates classes and objects on the fly. The opportunity for memory leaks is very high. EC++ is being developed to counter some of these problems. Things like templates will not be in EC++.

I think that in the next two years C++ will be viable for 16 bit systems upwards. I am still not convinced that this is a good idea in terms of speed, memory resources and deterministic response. However, I would not advocate its use in 8 bit systems even if it does become available.

7. Conclusion

Embedded Engineering is just that. An engineering *discipline*. Like architecture, embedded engineers should use the discipline of proper construction methods and work within the rules. With practice, this will produce robust and safe systems automatically (and quickly).

Once one has got over the learning curve of doing things rigorously, ones mind is free design with flair. Most of the worlds great buildings were designed using standard bricks or frames made from standard girders to stringent (and long) building regulations. Builders and architects who have been shown not to use the correct methods end up in court or no longer able to practice.

As the IEE, BCS, Engineering Council and the government push to raise the status of Engineers in the UK embedded engineers will be expected to come into line with other professions. IE using defined construction methods.

The game is changing and you WILL be judged by it's rules weather you want to play or not.

It does not take much to use Lint, MISRA-C, to use version control (this is required for ISO9000 anyway). The costs for these tools usually repay themselves in the shortening of debugging on the first project. The potential savings are enormous if it saves you having to go to court.

I am assuming that you are of course, using a good compiler and debugger. It is of little use writing good solid embedded C if you then use a doggy compiler or an intrusive ICE. Tools are a whole new ball game explained in ***Microcontroller Debuggers – Their Place In The Microcontroller Application Development Process*** [Hills] Getting the language in to a robust state is one thing. Having the (appropriate) tools of the same quality to support it is another.

Good SW Engineering practice saves you having to thing about much of the trivial time wasting parts of a project and lets you get on with innovative and safe designs.
I recommend that you read MISRA-C and Konigs Traps and Pitfalls.

I shall finish with the last line of the introduction:

It is well worth buying good quality tools.

and add:-

The ART in Embedded Engineering comes though good engineering discipline.

8. Appendix A (style)

I do not intend to go through a complete style guide. These are a few notes to help you decide which of these common styles suit you (if any). Indentation is up to the user as long as it is consistent. That is consistent across a project not per developer.

There are many styles freely available on the Internet. It is not the end of the world if you do not get to use your per scheme.

K&R

This is the original style. However, this has largely been superseded. Especially with the change to ISO C where the parameter types are specified in the function line not below it.

Some people still use this because it is the "correct" way of doing things and site K&R (First edition). C is over 20 years old and even the originators, enlightened as they were, have moved on and learnt more. There is nothing wrong with this style (obviously moving the type declarations) but it is not *the* definitive style.

The only word of warning is that *some* debug tools (usually the older ones) assume this style. They require the opening { to be on the same line as the if, do etc.

```
void change_KandR( from, to)
char *to
char *from
{
	do{
		if('a' == *from ){
			*to ='A';
		}else{
			*to = *from;
		}
		++to;
		++from;
	} while( '\0' != to[-1] );
}
```

Indented Style

This is a later style than K&R. It requires more space on screen and on paper BUT it takes up no more room when compiled. This more open style crept in when screens gained colour, windows and more than 80 columns by 40 lines.

```
void change_case( char * from,  char *to)
{
        do
                {
                if(‘a’ == *from )
                        {
                        *to =’A’;
                }
                else
                        {
                        *to = *from;

                        }
                ++to;
                ++from;
                }
        while( ‘\0’ != to[-1] );
}
```

Exdented Style

```
void change_case( char * from,  char *to)
{
        do
        {
                if(‘a’ == *from )
                {
                        *to =’A’;
                }
                else
                {
                        *to = *from;

                }
                ++to;
                ++from;
        }
        while( ‘\0’ != to[-1] );
}
```

My style

Over the years, I have found I prefer to use Exdented (when nothing else was specified). This is because I find that when doing code reviews it is easier to spot the pairs of braces round a block. I usually join the pairs of braces using coloured pencils.

9. 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.
------------------------------------------------------------------------------*/

#incldue <stdio.h>

void main (void, void)
{
unsigned i;
long fellow;

fellow = 0;

fer (i = 0; i < 1OOO; 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)
```

10. References

Beach, M. Hitex *C51 Primer* 3rd Ed, Hitex UK, 1995, http://www.hitex.co.uk

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

Hatton L, *Safer C*, Mcgraw-Hill(1994)

Hills C A, *Microcontroller Debuggers – Their Place In The Microcontroller Application Development Process* Chris Hills & Mike Beach, Hitex (UK) Ltd. April 1999 http://www.hitex.co.uk & http://www.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

[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.

Kernighan Brian W, The Practice of Programming. Addison Wesley 1999

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

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

MISRA Guidelines For The Use of The C Language in Vehicle Based Software. 1998 From www.misra.org.uk and www.hitex.co.uk

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.htm. This is well worth reading.

hitex
DEVELOPMENT TOOLS

Hitex (UK) Ltd.
Warwick University Science Park
Coventry, CV4 7EZ
Tel. +44 (0) 1203 692 066
Fax. +44 (0) 1203 692 131
www.hitex.co.uk
chills@hitex.co.uk