On-Target Stress Testing of C Cross Compilers

First Edition

by

Mrs ***Olwen Morgan*** *CITP, MBCS*

and

Eur Ing ***Chris Hills*** *BSc , C. Eng., MIET, MBCS, FRGS, FRSA*

PhaedruS SystemS

The Art in Embedded Systems comes through Engineering discipline.

Contents

On-Target Stress Testing of C Cross Compilers1

Compiler Stress Testing for Safety-Critical Projects

Stress testing of compilers, alongside traditional test suite compiler testing, can form a valuable element of a safety case for critical projects at IEC 61508 SIL3/4 or equivalent integrity levels.

1 Introduction

Many software engineers will have some idea of what compiler test suites are but fewer will be familiar with the use of compiler stress test tools. The purpose of this paper is to describe how such tools work, what they test, what they do not test, and how to assess whether stress testing is actually needed for your project.

2 What is a compiler stress tester?

In general usage the term "stress testing" means exercising a system under test beyond the conditions expected in normal operation. Its purpose is to assess how robust the system is under abnormal loads or operating conditions. Compiler stress testing is essentially the same but stresses the compiler by making it compile correct but syntactically convoluted code, i.e. code that a human programmer would be unlikely to write outside the Obfuscated C Competition (see http://www.ioccc.org). In fact since 1984 the International Obfuscated C Code Contest has often been used as an informal source of stress testing programs for compilers.

Formal compiler stress testing programs are, however, commonly pseudo-randomly generated by a stress test generator tool rather than a human mind.

A major landmark in stress testing was the development of Wichmann's stress test generator for Ada [WiDa, 1989]. Wichmann and Davies described various concepts of stress testing that have been adopted by subsequent tools. Stress testing is not just a completely random unstructured collection of programs that *"some one thought would make a good test"*. (Though many compiler companies will use some of the Obfuscated C Competition programs and other assorted programs ***in addition to*** a formal Compiler Validation Suite, Regression Test Suite and Stress Testing.)

The main formal stress tester for C used today is the Csmith stress test generator, developed at the University of Utah by John Regehr and his team, which applies the Wichmann-Davies principles to the testing of C compilers [XYER, 2011]. Csmith is widely used and is freely downloadable from the University of Utah web site https://embed.cs.utah.edu/csmith.

However there is a lot more to testing a compiler than downloading the stress test suite and running it. As Deep Thought once said: "42 ***is*** the answer but you never really understood the question…" In fact the Csmith web site contains the following warnings:

We strongly request that you:

- *Acquire a sophisticated understanding of the C standard before reporting any bug -- this may take months*
- *Read this entire document before reporting a compiler bug*
- *Always check if an issue is known before reporting it*
- *Understand and conform to whatever additional local bug-reporting conventions apply to the compiler you are testing*
- *Listen to feedback from compiler developers and other members of the community*

Failure to heed these instructions will cause you to:

- *Waste developers' time*
- *Probably be publicly flamed*
- *Definitely be ignored in the future*

This is quite apart from wasting your own time. When they say "A sophisticated understanding of the C standard" this means you need an *official* copy of the ISO C Standard (not one of the many drafts wandering the internet which may be "almost complete") and have not just read it but fully understood it, and in particular ISO 9899:1990 Annex J, for obvious reasons. Reading K&R's C book and having a few years developing experience doesn't cut it. Compiler testing is not the same as application testing.

3 Stress testing vs. validation suites

Pseudo-random stress testing differs in several ways from testing using a Compiler Validation Suite (CVS). A CVS is a fixed set of programs designed primarily to demonstrate that the compiler complies with the relevant language standard. Most programs in such suites are conforming programs. Deviance tests may be included to see whether the compiler exhibits certain common types of fault but such tests are never conforming programs. Each is preceded by a pre-test to check that a program the same as the conforming program except for a single nonconforming construct does actually compile and run successfully. A CVS may also include tests of implementation-defined, unspecified and undefined aspects of the implementation. It is a characteristic of fixed validation suites that they exercise the front end of a compiler more strenuously than the back end.

Generally the front end of a compiler is the language parser - that is the Lexical Analyser, Syntax Analyser, and Semantic Analyser.

Generally the Back End is the Intermediate Code Generator, Machine Independent Code Improver, Target Code Generator and Machine Dependant Optimiser. Some optimisations, or ranges of optimisations can be pre-selected by compiler flags. Other optimisations or ways of converting certain constructs are often possible. They can often depend on the target architecture and the methods or algorithms used by the compiler. A mismatch between the front and back ends of a compiler can cause problems and both need to be tested. This is why as of 2015/6 the authors are seeing more requests for compiler validation on the target processor.

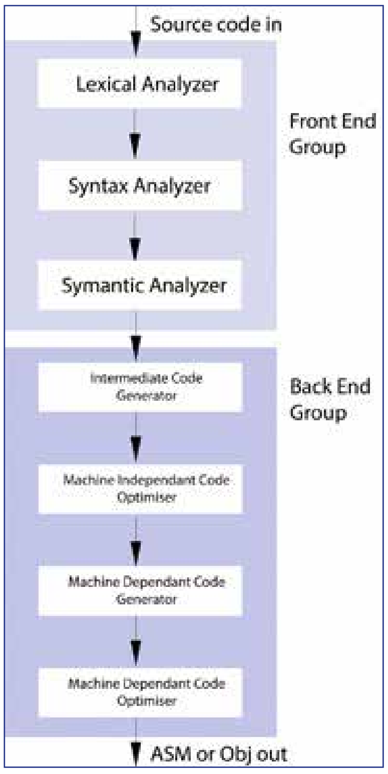
Source code in
Lexical Analyzer
Front End Group
Syntax Analyzer
Symantic Analyzer
Intermediate Code Generator
Back End Group
Machine Independant Code Optimiser
Machine Dependant Code Generator
Machine Dependant Code Optimiser
ASM or Obj out

Fig. 1

Pseudo-random stress testing does not rely on fixed sets of programs, though tests that have revealed errors may subsequently be preserved and re-used for regression testing purposes. What a stress test generator does is to generate a correct program by pseudo-random traversal of the language grammar. This creates test programs that are quite unlike those that a human programmer would, or should, normally write. They are also quite different from those that would be created by a modelling tool with an associated code generation package, for example MATLAB/SIMULINK.

Experience has shown that such programs exercise the back-end of a compiler more strenuously than the front end. This is because they throw up quite convoluted and unusual combinations of operators that would not typically form part of any logical test pattern unless some form of 100% collocation coverage was being undertaken. Practically speaking, this is virtually impossible owing to the vast number of possible combinations.

4 How a stress test generator works

To understand pseudo-random stress testing properly you need to understand how pseudo-random stress tests are generated. Consider the (highly simplified) program in Fig. 2.

This seemingly simple function is a lot more complex than it appears because FIXED and GENERATED are defined as macros. The stress tester first randomly selects an integer value for FIXED, say, 7. It then creates as the expansion of GENERATED an expression that should evaluate to 7. This expression is, however, generated by a random traversal of the C syntax for expression and can

```
#define FIXED <inserted by stress test generator>
#define GENERATED <inserted by stress test generator>

#include <limits.h>
#include <math.h>
#include <stdio.h>

int main(void)
{
  int f = FIXED;
  int g = GENERATED;

  printf("EXPECTED RESULT = %i \n", f);
  printf("ACTUAL RESULT   = %i \n", g);

  if (f != g)
  {
   printf("FAIL\n");
  }
  else
  {
   printf("PASS\n")
  }
  return 0;
}
```

Fig. 2

be quite convoluted.

The brief example in Fig. 3 shows a simple possible instance of pseudo-random expression generation.

```
7 → expression1 + expression2          // random form selected
expression1 →  11                      // random value selected
expression2 → -4                       // to sum to 7

11 → expression3 * expression4         // random form selected
expression3 →  2                       // random value selected
expression4 → 5.5                      // to give product 11

-4 → expression5 / expression6         // random form selected
expression5 →  -16.0                   // random value selected
expression6 →  4                       // to give quotient 4

-16.0 → sqrt(expression7)              // random form selected
expression7 →  256                     // to give 16 as root

4 → expression8 << expression9         // random form selected
expression8 → 1                        // random value selected
expression9  →  2                      // to give 4 on shifting
```

Fig 3

Where numbers are randomly expanded, the tool selects a relevant form of expression at random from the C language syntax. Once the form of expression has been selected, values are allocated to its sub-expressions so as to preserve the value of the number for which the expression is being generated. These steps are repeated cyclically, at each stage ensuring that when the whole randomly generated expression is evaluated, it will be equal to the fixed value (here 7) originally selected at

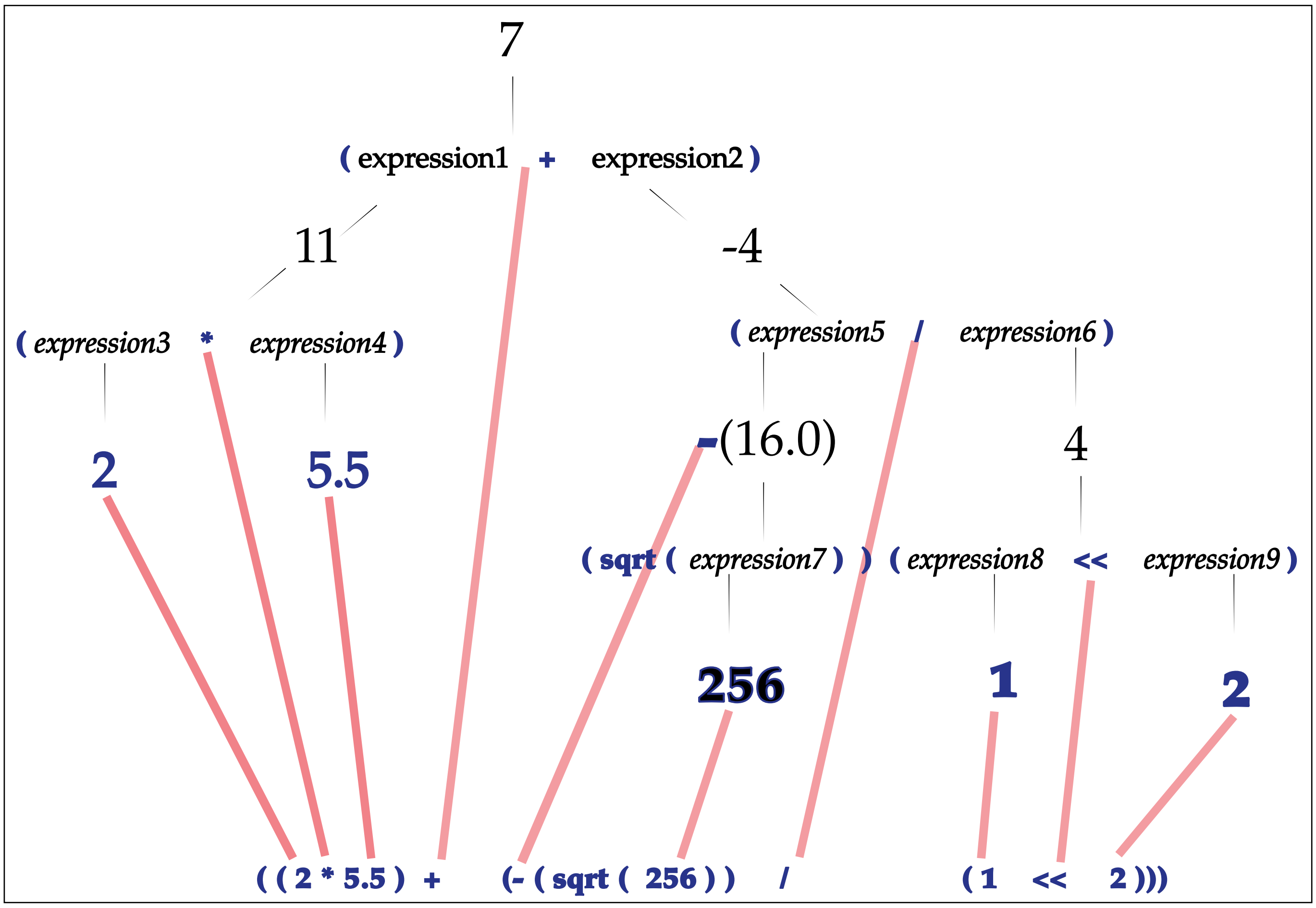
7
(expression1 + expression2)
11
-4
(expression3 * expression4)
(expression5 / expression6)
2
5.5
-(16.0)
4
(sqrt (expression7))
(expression8 << expression9)
256
1
2
((2 * 5.5) + (- (sqrt (256)) / (1 << 2)))

Fig 4

random. Notice that the tool determines the values of expression according to mathematical rules, and need not insert explicit type conversion if a conforming compiler should produce the correct result by implicit conversion. The process can be depicted graphically in the form of a parse tree as shown in Fig.4. The example shown here is simplified to show the general idea. In practice, a stress test generator will generate much larger and much more varied forms of expression involving the entire range of operators. It will also introduce variables as operands and these may be in any kind of storage chosen at random. In addition, randomly generated constructs are not limited to expressions. Complex control structures can also be generated which may exceed all recommended limitations on control flow complexity. The result is something that neither a human programmer nor a code generator for a modelling tool is ever likely to write.

Since the test generation process is random and the total number of possible combinations is immense it is highly likely that forms of expression will be generated that were never included in the compiler developer's original testing, particularly when, as with some compilers, the front end parser and back-end come from two different sources to form the compiler suite. Front ends tend to be standard e.g. the EDG parser or the GCC front end but are married to many different back ends. A crucial part is how the intermediate code system is devised and implemented.

Since the generated programs are always made syntactically and semantically correct, they almost invariably compile without error messages but quite often they can include code generation errors. Sometimes these errors have been long dormant in the compiler. It is quite common for random stress test programs to find previously undiscovered errors in very widely used and generally reliable compilers, simply because that combination of operators and expressions has never been used before.

5 Program pruning

Where a generated program is run and outputs a FAIL result, it will generally be a single localised construct in the text that actually elicits the error. Consequently, the test can be simplified by traversing the parse tree of the randomly generated expression to find exactly which sub-expression causes the error. The tool can then generate a program containing only that sub-expression, creating a small test that can be used for regression testing purposes.

This pruning process has to be automated. Raw random stress test programs are typically so convoluted that they are hard for humans to read accurately, let alone for them to examine to locate errors manually. Test program pruning routines are an essential part of any industrial-strength stress test generator.

6 Using a stress testing tool

The procedure for using a stress test generator is straightforward. You generate a predetermined number of test programs and run them. Those that do not demonstrate compiler errors are put in one group. Those that do find errors are then reduced to their essentials by program pruning and re-run to show that they still elicit the error. These errors need to be investigated to ensure they really are errors and are not due to architecture, compiler or MCU limitations. (See the comments in section 2, above from the Csmith web site on error reporting.) The pruned tests are then put in another group for re-use in future tests of the same compiler or tests of different compilers and form part of the Regression Test Suite. Used in this way a stress-testing tool typically finds bugs faster than a population of users. Thus stress tests greatly strengthen compiler regression testing.

7 When do you need stress testing?

Not all projects will need to undertake compiler stress testing. It may, however, be required for safety related projects at SIL3/4 or ASIL C/D as discussed below:

Code generated from modelling tools

Increasingly system-modelling tools can generate code automatically for a verified model. Such code generators should be used with extreme caution as experience has shown that the quality of the generated code is often poor. One particularly well-known model-to-code tool at one time produced code that was littered with violations of MISRA C coding rules, even the MISRA Auto Code rules. These modelling tools build the source by using templates and putting together blocks of code, which can produce code that no human would construct and probably not in a way that any tester would use to test the compiler. Whilst the blocks and templates themselves may be sound, some combinations may not be. There are no third-party test suites for modelling tools in the same way there are for other translators like compilers. And it is not uncommon for concerns over code quality to cause safety authorities to frown upon the use of such tools.

Manual translation from model-to-code does not necessarily remove this problem. For example, it is relatively straightforward to produce good C code by manual translation of a system model given in, say, Coloured Petri Nets [JeKr, 2009]. On the other hand this will involve translating from a dialect of the functional language ML into the imperative language C. The resulting coding style, essentially following a single-assignment pattern, is not common in handwritten C and may well collide with compiler code generation errors purely on that account.

As an example consider the coloured Petri net in Fig. 5, which depicts part of the code of an instrument that computes liquid density.

The C code that represents this might be:

```
static volatile double P1_mass;
static volatile double P2_volume;
static volatile double P3_density;
…
double T1_Compute_density (double x,
   double y)
{
   double x_in = x;
   double y_in = y;

   double ret_out = x/y;

   return (ret_out);
}
…
P3_density = T1_Compute_density (P1_
   mass, P2_volume);
```

The simple way to write this would be

P3 _ density = P1 _ mass/P2 _ volume

but in translating Petri nets into C we would probably follow a convention that transitions are represented

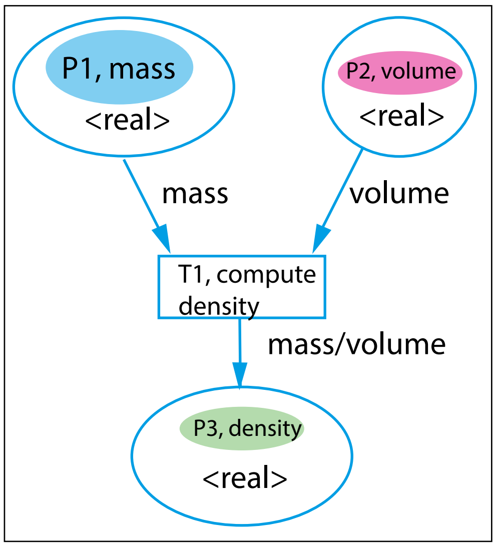

Fig. 5

by functions and not inline code and an optimiser might well eliminate the function-call (and indeed the function) entirely. The problem here is that by generating C code from the net that is a systematic translation of code in a functional language, we might present the optimiser with far more optimisation opportunities than handwritten code might give it. This, in turn, risks colliding with parts of the optimiser that are less often exercised and might turn out to be buggy.

Whenever code is constructed by translation from a system model, whether automatically or manually, it is prudent to perform stress testing of the compiler in case the code elicits dormant code generation errors in the compiler.

Unusual combinations of compile-time switches

Modern C compilers tend to have large numbers of compile-time switches. Even old C compilers for some MCU's have many combinations of switches that have only been tested in the "common" configurations even after many years of use.

It is impossible for the compiler developer to test under all possible switch combinations especially with all possible combinations of C language constructs and variations of target hardware. For the humble 8051 family there were around 1000 variants. For the ARM Cortex M there is at least an order of magnitude more variants and the number is growing daily. Thus whilst some popular compilers like the Keil 8051 compiler can claim "proven in use" with about 80% of the market; the compiler might never have been used, never mind tested, with any particular MCU and switch settings until you do it. And that is for a compiler binary issued from a single source. For GCC compilers where there is no single binary source point, the situation is even more fraught with uncertainty.

Code compiled with untested switch combinations may be more likely to elicit code generation errors than code tested under switches covered by the developer's testing. In such cases it is prudent to undertake stress testing with the switch combinations used by the project to provide some measure of assurance that the compiler behaves properly in those circumstances.

```
-c99 -c --cpu Cortex-M3 -D__
EVAL -D__MICROLIB -g -O3
--apcs=interwork --apcs /ropi/
rwpi --split_ldm --split_
sections --strict --enum_is_int
--signed_chars -DSTM32F10X_MD
-o ".\Obj\*.o" --omf_browse ".\
Obj\*.crf" --depend ".\Obj\*.d"
```

Example compiler switches

Compiler optimisations

One particular kind of compile-time switch warrants special mention. It is usually recommended not to use C compiler optimisations in critical systems. The problem is that a C compiler will often apply optimisations by default and the switches may switch them off rather than on. The degree of control that this switching-off gives varies from compiler to compiler. Some compilers by design produce more efficient and compact object code than others - not all compilers work in the same way. Variation in how these various switches work in combinations with the algorithms in the compiler and with the combinations of the C source can produce quite unexpected results.

C programs are particularly vulnerable to being incorrectly optimised as a result of the lack of restriction in the language compared with, say, Ada. An example of unexpected results was found using a compiler that had an option switch of "faster" or "smaller" code. With several modules the "faster" switch actually gave smaller object code than the "smaller" switch!

Nevertheless there are occasions on which the use of optimisations is necessary. For example, for a large air traffic control system optimisation was used to meet performance requirements. In other cases aggressive optimisation has been used in order to fit additional features into a fixed memory, as found in most embedded systems. In the real world optimisations cannot always be avoided.

The problem with such optimisations is that there are often more bugs in generating optimised code than un-optimised. This, of course, is where stress testing comes into its own. Whilst some of the better CVSs do have an element of back end testing, most don't. Those that do test the back end may do it only partially. Randomly generated stress test programs often find compiler bugs in back end code optimisers. The value to the developer is that this highlights usages that should be avoided or at least be subject to increased testing rigour or special workarounds. Fixed validation suites cannot match the capability of stress tests in this area.

Compilers not tested on-target

Arguably the best reason for undertaking compiler stress testing is that compiler developers actually perform very few of their compiler tests on real targets. The reason for this is simply that it takes too long, because on-target test times are dominated by the times taken to upload object code to a target and download results from it. In the early days one compiler validation failed because the number of write, read, erase cycles killed the flash memory: fortunately flash memory has moved on a long way since then! (Though even in 2017 flash on test boards can fail after it has been used for only a few full test runs.

There have been cases where in order to do the compiler validation tests the memory in the target simulator was expanded beyond that of the real hardware. This is because some of the test modules would not fit in the memory space of the actual hardware. It also meant that the tests could not be run on the real target hardware.

One particular silicon vendor provides a compiler that is very strenuously tested on-host using simulators for the target processor cores and every available fixed test suite and stress test generation tool plus a large volume of accumulated regression tests. This particular compiler is quite possibly the most thoroughly tested in the industry, yet its generated code may still not be tested for the specific target processor silicon variant used by a particular project. Significant numbers of the vendor's regression tests started life as stress test programs that had found on-target errors previously missed by on-host testing. Compiler testing is not a simple task

8 Combined testing

For most critical projects it may be advisable to perform compiler testing using both a fixed validation suite and pseudo-random stress tests. At least one TÜV has required this for a SIL3 project running on a safety-rated dual-core lockstep microcontroller. In such cases a basic validation test suite augmented by random stress tests provides a level of compiler assurance commensurate with the criticality of the project.

It is reasonable to expect increasing requirements for this combined form of testing for critical projects across the industry sectors and indeed since 2015 the authors have seen this beginning to happen.

9 Conclusion

Stress testing is a valuable complement to the use of fixed validation suites in critical projects, by revealing long dormant compiler code generator errors that could lead to operational failures of critical products. While this is normally thought of as referring to safety-critical applications, you should also consider a critical product one that could cause problems for you or your company if it should be faulty in operation.

Stress testing of compilers can form a valuable element, alongside traditional test suite compiler testing, of a safety case for critical projects at IEC 61508 SIL3/4 or equivalent integrity levels. The Csmith stress test generation tool is available free of charge and is suitable for use with all embedded applications. Nevertheless, we would argue that using a stress test generator tool is a specialised task that should be undertaken only by suitably qualified staff, as it requires technical control to accredited test laboratory standards. Also it should only be used as part of a coherent Compiler Test Plan that involves formal CVSs, Regression Test Suites and other testing. Phaedrus Systems consultants are qualified to do this kind of testing and can advise on all aspects of compiler validation for high-integrity developments.

It should be noted that there is more that can be done. Even with a tested compiler, the project source code should be written to conform to a coding standard, a language subset, such as MISRA C, and be subjected to static analysis. (This is mandatory for IEC 61508-7 (Functional Safety)). This approach will reduce problems by avoiding the areas of the C language that programmers often abuse or have problems with and can keep you out of the darker areas of the compiler – today most compiler companies test against MISRA C.

References

[JeKr, 2009]	Jensen, K., and Kristensen, L. M., *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*, Springer, July 2009, ISBN-10: 3642002838
[WiDa, 1989]	Wichmann, B. A. and Davies, M., *Experience with a compiler testing tool*, NPL Report DITC 139/89, National Physical Laboratory, UK, March 1989
[XYER, 2011]	Xuejun Yang, Yang Chen, Eric Eide, John Regehr, *Finding and understanding bugs in C compilers,* ACM SIGPLAN Notice

On-Target Stress Testing of C Cross Compilers

First edition September 2017

Phaedrus Systems Library

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

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

http://library.phaedsys.com

PhaedruS
SystemS

*The Art in Embedded Systems
comes through Engineering discipline.*