Compiler Validation 7

Testing Code Generators: A Dark Art

First Edition
by
Mrs Olwen Morgan CITP, MBCS
and
Eur Ing Chris Hills BSc , C. Eng., MIET, FRGS, FRSA

PhaedruS SystemS

The Art in Embedded Systems comes through Engineering discipline.

Contents

Testing Code Generators: A Dark Art

Compilers conceptually have a front end and a back end. The back end is the code generator that takes intermediate code file and produces, object code.

1 Introduction

Compilers conceptually have a front end and a back end. The front end is the part that parses the source language into tokens and builds a syntax tree, which may be saved as an intermediate code file. The back end is the code generator that takes intermediate code file and produces either assembler or, more commonly these days, object code ready for linking.

In practice, however, except for a few generic compilers for older architectures, the front-end/back-end split is no longer the same. (See our paper: "What is a compiler?") for more detail on this.) Modern compilers are multi pass and start optimising for specific targets from just after the parser in what was the front end. Also, modern linkers will play a large part in optimisation. For some compiler-MCU combinations more optimisation is done in the linker than the compiler.

Today almost all compilers perform optimisations. There are many ways of translating source to binary. A lot of it will depend on the design of the compiler, the design of the intermediate language and the target architecture. The optimisation switches on a compiler merely adjust the window of optimisation a little. You generally can't "turn off optimisation" as such.

In our paper on choosing a test suite [C Compiler Validation: Choosing a Validation Suite] we saw that fixed test suites are good at testing compiler front ends, that is the handling of syntax and static semantics, but not so good at testing back ends or code generation. There are several reasons for this, which we have mentioned in the other papers in this series but will discuss more fully in this paper.

The difficulties in testing compiler back ends should not be underestimated. Currently, in 2020, there are no entirely satisfactory ways to perform comprehensive testing of C and C++ compiler back ends or code generators. It is important for developers writing critical software to understand why this is and what can be done. It also goes some way to explain why compiler validation on target is gradually becoming more commonly required - having a validated compiler does not remove the need for testing any binary produced by it.

2 Testing code generators

When writing a fixed compiler test suite, the starting point is: **The Language Standard**. For the overwhelming majority of embedded systems the relevant standards are the ISO C 9899 and/or ISO C++14882 standards. A good test suite will contain at least one test for each requirement of the language standard and will thus achieve 100% requirements coverage with respect to the standard and hence 100% coverage of the syntax. This is why fixed test suites are good at exercising front ends - they pretty strenuous testing by any measure.

The situation with back-end tests is not so fortuitous. First, a typical language standard says, at least implicitly, what the compiler must do but leaves implementation details to the implementer. The authors have served on the ISO language standards bodies for C and C++ for many years (decades) where most of the working group members have little interest in, nor experience, of bare-metal embedded hardware. Indeed, some parts of the ISO C standard have been implemented on relatedly few embedded target processors.

In addition there has been, certainly from 1990 to 2020, a convention in the ISO C working group not to break existing implementations. ***NOTE in 2020 there was a suggestion by several major companies that C++ should only support 64 bit and above going forward.*** Therefore the C standard explicitly designates some constructs and behaviours as implementation-defined, undefined or unspecified, leaving the implementer to make relevant design decisions. For example the order of expression evaluation can, subject to operator precedence, be evaluated left to right, right to left or middle out or any combination.

Where the implementer clearly describes the nature of such items, it is usually possible to write tests for them. For code generators, however, the requirements imposed on implementers are too general to form the basis of a test coverage domain. This is especially true of optimisations and effectively closes the door to robust, traceable testing. If you have no coverage domain to cover, then you can't even measure coverage. The difficulties are best shown by examples.

2.1 First example: assigning to volatile variables

Variables declared volatile are a well-known problem area. One paper [EEJR2008] on their use in embedded C code suggested that cross compilers did not implement volatile correctly. This is somewhat misconceived as the paper really only tested a dozen or so variants of the GCC compiler which at the time did not correctly handle volatile – while at the same time the leading commercial compilers did correctly handle volatile. However the myth still remains in some quarters.

Consider the following function that sends an octet via an external interface:

```
volatile unsigned char octet = 0u;
volatile unsigned char cntrl = OFF;
// sets I/O interface to inoperative
// assumes OFF defined by a #define

void SendOctet1 (unsigned char uc)
{
    octet = uc;
    cntrl = SEND;
// where SEND is a bit pattern used
for I/O control
    cntrl = OFF;

    return;
}
```

In this kind of coding pattern (nostalgically typical of UARTs), uc and SEND are put into locations declared static that will be mapped, respectively, to an I/O data register and the corresponding I/O control register. The mapping may be done in the linker or may be visible in the program using extension constructs that permit mapping of variables to absolute addresses. The keyword volatile is here an indication to the compiler that it should not optimise away the initialisation.

Unfortunately the standard does not prohibit the compiler from changing the order of static initialisation. A sensible optimiser would perform them in the order written but the C standard does not mandate this. Typically commercial embedded C compilers do but compilers for desktops don't but neither can be depended on to do it the typical way. Also, it is very easy for implementers to make mistakes in deciding whether an optimisation can be safely attempted, especially in highly optimising compilers that perform different kinds of optimisation one after the other. To get around this problem, a cautious developer might rewrite the above function as:

```
void SendOctet2 (unsigned char uc)
{
return
((void)(octet = uc,cntrl = SEND,
 cntrl = OFF));
}
```

where the C standard's semantics for the comma operator explicitly prescribe the order of assignment. To test whether the compiler changes the order of evaluation in SendOctet1, we would have to compare the results of both SendOctet1 and SendOctet2. This means that a properly veridical test must be a differential test. It must compare the results obtained from two different functions that are devised so as to prevent optimisation in one case but permit it in the other.

Moreover, just one differential test would not be enough here. It may be that the compiler takes notice of the **`volatile`** key word and simply leaves alone anything that writes to or reads the relevant variables. Hence one might have to write two differential tests, one with the locations declared **`volatile`** and one without. Yet in this case you could not necessarily conclude much from a test in which the optimiser does the right thing. It might do the right thing in one context yet fail in another. Generally in testing optimisations, the only helpful results are those that show the optimisers behaviour to be incorrect in particular instances.

2.2 Second example: common sub-expressions and recursion removal

When testing optimisations, not only is there no traceable coverage domain but the tests themselves cannot always be functional tests. This may seem counter-intuitive but an example clearly demonstrates the problem. Consider the following C functions that calculate the greatest common divisor of two integers (granted you wouldn't use a recursive program in a critical setting but it serves here to show the effects of interacting optimisations).

```
  int gcd1 (int p,int q)          / /
assuming q <= p
  {
      return ( p%q == 0 ? q : gcd1(q,
p%q));
  }
```

Here we see that **`p%q`** is evaluated twice. A compiler that performs common sub-expression elimination might notice this and avoid it by introducing a temporary variable so that the generated code would be as if the programmer had written:

```
  int gcd2 (int p,int q)          / /
assuming q <= p
  {
      int temp = p%q;

      return (temp == 0 ? q : gcd2 (q,
temp));
  }
```

The question then arises of how you test the part of the optimiser that performs common sub-expression elimination. Clearly no black box test will suffice because, if the optimiser is performing correctly, you would expect the same output in both cases. But you would also expect the same if the compiler were not performing the optimisation. The kind of test you need here is to execute the function, say, ten million times while timing it using either a simulator or debugger with trace. If the run times are very nearly the same, then you can reasonably infer that no optimisation has been performed but you would expect the second function to run faster if it had.

To complicate things further, the compiler might also perform an optimisation that replaces tail recursion with iteration and might do so in addition to removing common sub-expressions. In this case the compiler might generate object code as if the original source were:

```
int gcd3 (int p,int q)
{
    int tempq q;
    int tempmod = p%q;

    while ( tempmod != 0)
    {
          tempq  = tempmod;
          tempmod = tempq%tempmod;
    }

    return tempmod;
}
```

Now the question arises of how to distinguish the following four cases:

1. no optimisation
2. common sub-expression elimination,

3. tail recursion elimination,

4. both common sub-expression elimination and tail recursion elimination.

Here full differential testing requires an additional gcd function (call it gcd4) in which iteration is used instead of recursion and there is a redundant sub-expression evaluation. Six tests are required each one comparing one of the six possible pairs of functions. Even then it one might have to specify some kind of tolerance range comparison of timings in order to get a useful verdict because the hand-optimised versions of the functions might not be exactly equivalents of what the optimiser itself produces. Moreover those six cases would be for just two interacting optimisations. If register optimisation were possible, then one might need to have eight different versions of gcd and 28 different pairwise comparisons. Combinatorial explosion looms already.

All this potential complication arises because the C standard imposes no requirement on an implementer to document what optimisations are performed or whether any are performed at all. The closest the C standard gets to prescriptiveness here is text like the following from clause 5.1.2.3 of the ISO 9899:1999 C standard:

"In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object)."

Unfortunately the standard does not actually define what it means for a side effect to be "needed". Thus the standard permits optimisations but it leaves the implementer free to decide when optimisations are performed without any precise provision as to when he shouldn't do them. Worse still is that the standard imposes no requirement that all possible optimisations of a particular kind should be performed. This allows the implementer to omit optimisations at will (or whimsy for that matter). It is this lack of prescriptiveness that leads to the need for potentially large numbers of differential tests in order to give a basis for reliable conclusions to be drawn from the results of testing.

2.3 Third Example: Idle loops

A common coding pattern in embedded systems is the time-triggered cycle in which a clock interrupt occurs at regular intervals and sets off a sequence of actions that is completed before the next interrupt occurs. When the actions for a given cycle are completed, control passes to an idle loop which goes around doing nothing waiting for the next clock interrupt. To achieve this in C, some C coding standards recommend the **for** construct:

```
  for ( ; ; );
```

The problem here is that an incautious optimiser might remove the loop entirely judging it to produce no side effects at all. (Some early FORTRAN optimising compilers did this, much to the initial confusion of the people who were testing them.) A possible solution is to write the loop as:

```
  volatile static int mod   = 7;
  volatile static int cycval = 1;
  ...
  TEST:  if( cycval == 3 ) goto: ERROR;
// cycval is never 3
      cycval = (cycval + cycval) % mod;
      goto TEST;
// infinite loop

  E     R     R     O     R     :
return((void)(exit(EXIT_FAILURE));
  }
```

where the final brace is the closing brace of the main function.

(*Note that the loop is coded without braces so as to avoid cause for use of the stack, otherwise stack overflow could occur after just a few cycles since, in an incautious implementation, the interrupt might simply transfer control leaving the stack unchanged. In other contexts, both the unbraced loop and the* **goto** *statement would probably be prohibited by a coding standard.*)

Here the loop runs **cycva**l through the sequence 1, 2, 4, 1, 2, 4, 1, 2, etc. Since 1 is a quadratic residue modulo 7, **cycval** can never attain any of the non-quadratic residue values 3, 5, or 6, so the loop will never terminate. More importantly, even though an optimiser might erroneously ignore the **volatile** keyword, it has to perform reasoning in elementary number theory to recognise that this is an infinite loop. The author has never known such a loop to be optimised away and infers that few C compilers even attempt such analysis.

To test whether the loop is optimised away, we would have to detect the failure exit and this would necessarily rely on highly implementation-dependent aspects of the compiler. One could not necessarily rely even on a single differential test to deliver the required test verdict.

2.3 Fourth Example: Idle loops

Defensive coding is a problem. It is code that is designed to catch exceptions, unexpected errors and is generally infeasible and cannot be executed under normal operation. This is different to dead or unreachable code. Most safety and security coding standards require defensive code and at the same time do not permit dead or unreachable code. This can be reconciled on paper with local rules. The bigger problem is that the ISO C standard does permit the removal or optimisation of un-used, dead or unreachable code. For example: hat, they will be different to the published standard.

```
   enum (red,amber,green) colour;

  switch (colour)
  {
  red:     break;
  amber:   break;
  green:   break;
  default: ;/* ERROR!*/
  }
```

When we ran tests on this we discovered that ***most*** embedded cross compilers do not remove the default clause. This is because their authors know that in embedded systems there is hardware that may or may not change things outside the knowledge of the software. The notable exception is GCC which, in this instance, sticks to the ISO-C requirement and silently removes the default.

It is not just the **default** in **switch** constructs but other similar defensive coding constructs that can be silently removed. The problem is as with opening the fridge door to see if the light is on, almost any test that involves instrumenting the code will give a legitimate path and the default will not be removed. Therefore testing for this sort of construct has to be done very carefully and on the ***normal production binary,*** not on test code.

2.5 Where does this leave us?

The foregoing three examples barely scratch the surface of many difficulties in testing code generators with interacting optimisations. Yet they already show us just some of the difficulties of testing:

- we need to perform differential testing.
- we need to observe non-functional aspects of behaviour.
- we need quite contrived tests to enable us to compare the effects of optimisation vs. no optimisation.
- the possibilities are not just optimisation vs. no optimisation; we have to consider the possible ways in which faulty and interacting optimisation may affect tests.
- unavoidable reliance on implementation-dependent aspects of the compiler means that it may not be possible to make differential tests reliably veridical; for any given aspect of optimiser behaviour, several sets of differential tests may be needed to cover the range of implementations expected in practice.
- there is little in the C standard to which we can make tests traceable or that can serve as a widely agreed basis for defining systematic coverage domains; syntax rule coverage is nowhere near strong enough to cover the different ways in which an optimiser may behave.

This may make us wonder whether attempting to test code generators and optimisers is worthwhile at all. It turns out that it is but that inherent technical limitations constrain what can be achieved in practice.

3 So how do we test code generators?

The discussion in Section 2 makes it sound as though the prospects for testing optimising code generators are uniformly poor if not impossible in any meaningful way. Fortunately this is not the case and we now outline areas in which useful tests can be performed to give a meaningful result that when coupled with the front end testing using appropriate fixed test suites will give a high overall confidence in the results of testing.

3.1 Can fixed test suites do useful code generator testing at all?

From section 2 we can see that it is not easy to devise a fixed test suite to exercise an optimising code generator either strenuously or extensively. Nevertheless, some kinds of behaviour such as register allocation and in particular the handling of arithmetic at boundary values, can be reasonably exercised by a fixed test suite. As of 2020 The SolidSands SuperTest test suite contains comprehensive tests – in its the depth suite - that do precisely that. This does help considerably as this is a common problem area. Nevertheless, while significant, this covers only a small part of what an aggressively optimising compiler might attempt.

3.2 Is there any hope for systematic coverage strategies?

The near absence of prescriptive requirements (other than for arithmetic) in the C standard leaves a compiler tester with little basis for determining suitable coverage domains for code generator testing. Here, of course, we mean and independent third-party tester who has no privileged access to technical information on the compiler's internal design. Note this is the design not the source code. Compiler design is closer to discreet mathematics than programming.

Compiler developers are in a better position but even they are hampered by the sheer complexity of code generation and optimisation and the consequent size of even theoretically possible coverage domains. Given the number of compiler switches and the number of possible patterns of source code construct usage, the number of combinations is infinite for all practical purposes. This remains the case even for C subset compiler targeted on RISC CPUs with limited instruction sets. In practice, therefore, it is fair to say that the scope for using systematic test coverage strategies remains limited.

3.3 Where does that leaves us with practical testing options?

With limited areas for using fixed test suites and combinatorial explosion arising from interacting optimisations, there is in currently only one practical option for strenuous exercise of code generators in independent third-party testing. That option is to use pseudo-randomly generated stress test programs. In adopting this kind of test strategy, we are effectively abandoning any attempt at getting systematically justified positive conclusions from testing.

If we cannot get fine-grained veridical testing with fixed test suites, we simply generate correct but convoluted programs in an attempt to trip up the compiler's back end. Early steps in this field were taken by Brian Wichmann, and his colleagues, at the National Physical Laboratory in the UK.[WiDa1989]. They wrote a pseudorandom stress test generator for Ada compilers. Wichmann's approach was subsequently applied to C compilers by John Regehr and his colleagues at the University of Utah, who produced the now well-known CSmith stress test generator. [XYER2011].

3.4 How do we exploit stress testing?

Ideally we would like to have fixed test suites that provide good tests of optimising code generators but just a few examples suffice to show us that such test suites may be unmanageable in both size and complexity. This leaves us with pseudo-random stress testing as the only practical alternative. It also means that the objectives of testing change.

Experience shows that stress testing is very good at finding obscure errors in code generators but it does not provide us with systematic coverage domains. Since, however, little testing of code generators can be made traceable to the C standard, this is not of pressing concern and we may as well abandon it as a desideratum. Hence stress testing is not well suited to positive validation and will not increase our confidence that the compiler under test actually performs as it should. What it will do, however, is go all out to break the compiler.

Of course, one may then ask what use it is to a developer if a compiler passes fixed validations tests with flying colours and then collapses ignominiously in stress tests. The answer is that stress testing is complementary to testing using fixed validation suites. We run a fixed validation suite as a screen for obvious errors. We use stress testing to give ourselves a sporting chance of identifying and avoiding areas of code generation that may contain infrequently encountered bugs.

4 Practical independent testing

If, for a given project, an assessment body requires independent, third-party testing of a compiler, then this should certainly include use of an extensive fixed test suite. If use of compiler optimisations is unavoidable, then the testing should ideally also include targeted use of pseudo-random stress tests. Done properly, validation plus stress testing should provide the developer with the following documents:

- a validation report based on testing with one or more fixed test suites, and
- a report on the results of stress testing.

These reports have to be separate because they are doing different kinds of testing. The expectation is that the fixed test suite will demonstrate compliance with the language standard under the options the developer is using for the project. It would also be surprising if stress testing did not throw up several errors. The report on validation with the fixed suite should therefore (assuming sufficient tests have been passed) certify that testing under the nominated options has produced results such that the compiler is deemed to comply with the language standard. No such conclusion, however, is likely to be appropriate for the report on stress testing. Generally there will have to be some recommended measures for avoiding the areas of the compiler that exhibit bugs. This necessitates careful planning of testing. In particular it is advisable to do stress testing before testing with a fixed validation suite.

4.1 Finding suitable compiler options

.

It is easy to see why stress testing, when done at all, should be done first. Initially one stress tests the compiler under the options intended to be used for the application. Then if stress testing finds error, the chosen options may be changed to see whether different options avoid the error. If such options can be found, then fixed testing can be done under those options with the result that both fixed and stress testing will, typically, show the compiler to behave reasonably under those options. This way the developer gets a third-party test report that not only demonstrates compliance under a fixed test suite but also shows that the developer has taken steps to avoid compiler errors that a fixed test suite cannot find.

4.2 When suitable options cannot be found

A pseudorandom stress test contains very convoluted code intended to push the compiler to and beyond its limits. Typically, it is only a small part of the test that actually elicits the observed error. Therefore, a usable stress test generator must also offer facilities to prune a test that has found an error. The process of pruning has been outlined in an earlier paper in this series: ***On-Target Stress Testing of C Cross Compilers*** Here we need to consider what to do when the code that causes the error has been isolated.

It may be that there is no set of compiler options that eliminates the error. In this case the developer has to find a workaround to avoid the error. At this point the origin of the error has to be tracked down and the problem should be referred to the compiler developer's technical support. What the compiler user then needs from them is details of any available workarounds, which could be anything from avoiding particular source language constructs to, in extremis, scanning generated code to find signatures of erroneous compilation - an expedient that is far from unknown. Once a robust workaround has been found, this may be documented and included in the stress testing report.

4.3 When suitable options cannot be found

The worst case in stress testing is when it finds an error for which there is no viable workaround and the compiler developer cannot correct the problem quickly. If this happens, it's time to think about switching to a different compiler. Though that may be a nuisance, it is probably better than persevering with a demonstrably flawed compiler. Independent testing will then have done its job by identifying that a proposed compiler was not fit for the originally envisaged purpose.

This situation has occurred in the wild where a compiler was written to the MCU specification but the initial shipments of the MCU did not meet its own specification and the compiled code did not work. There was then a discussion between the compiler company, the chip-maker and a major customer. In that case the compiler, for that customer, was patched and the other early release MCU customers notified. The MCU hardware was corrected and the original version of the compiler worked for all other non-early release. Such circumstances are actually more common than compiler users might suppose.

4.4 What an independent assessor may look for.

In the current climate of 2021, independent assessors are most likely to look only for validation testing with a fixed test suite – but this is changing with the availability of safety-rated dual-core lockstep micro-controllers. When robust hardware self-checking is available, the area at greatest risk of common-mode failure is the software itself and cautious independent assessors may look not only for traditional testing with fixed validation suites but also the wider testing that a stress test generation tool provides. This, however, remains relatively new ground for independent assessors. In the immediate short term, they are likely to decide on a case-by-case basis what an appropriate testing regime is and what, if stress testing is used, constitutes a suitable response to a stress test failure.

5 Conclusion

5.1 In the short term

In the present state of the art, there are no entirely satisfactory ways of performing independent, strenuous, and traceable testing of C compiler code generators - particularly those of aggressively optimising compilers. This situation is not likely to change with more powerful test hosts nor with more capable micro-controllers.

Ultimately the problem lies in language standards, most of which state hardly any requirements for how back ends in general and optimisers in particular should perform. This problem is almost certainly going to persist until it becomes the norm to give mathematically formal semantics in language standards and to specify in mathematical terms what invariants optimisers should preserve. For the moment, independent testers are stuck with one static test suite (SuperTest) that does address some of the problems and a pseudo-random stress tester (Csmith) that aggressively stress code generators. That combination gives the least bad option for back-end testing. It's not ideal for developers of the most critical systems but can at least help to steer them clear of some aspects of negligence.

Regardless of the compiler testing regime used, there still has to be testing of the binary actually produced for a project. This kind of testing is, thankfully, well supported by a wide range of commercial software tools that can perform stringent dynamic analysis of testing both at host/source and target/object level. This includes tools like stack analysers and timing tools. Also In Circuit Debuggers coupled with unit testing tools that can run functional tests on the target.

5.2 Future prospects

It is not all bad news, however. Research to develop verifying compilers is under way in both academic and industrial laboratories, for example the CompCert compiler from AbsInt [CompCert]. Once it is demonstrated that industrial-strength verified compilers for C are feasible, the landscape could change dramatically. Such compilers would make it significantly easier to develop traceable tests for code generators and optimisers and weaken the current dependence on pseudo-random methods.

On the other hand, the record of C's development to date does not exactly give cause for optimism. If the C standard continues to give wide discretion for optimisation without prescribing soundly based semantics, it will be very hard to develop robust, traceable back-end tests. The choice of languages and compilers that developers might then have will be down to what regulatory assessors deem acceptable. Nothing stays still for very long in systems engineering and in the long run a programming language whose standard eschews precisely specified requirements is not necessarily guaranteed continuing popularity.

5.3 The pragmatist's approach

Technical constraints in a field are rarely a good reason to ignore it. Though currently stress testing is limited in its traceability, it remains an exceptionally effective means of identifying flaws in compiler code generators. As such it is a well-proven adjunct to conventional validation testing. It is far better to know about compiler bugs that to use a compiler in total ignorance of them.

Owing to their complexity and their large numbers of possible options, compilers will continue to contain bugs. For the foreseeable future those bugs will continue to congregate in code generators and optimisers. When a regulatory assessor requires it, stress testing has a well-earned place in the compiler tester's toolkit. Like any tool, it needs to used by some one with a solid understanding of compilers and stress testing in a laboratory environment. Doing this properly is not a job for the average software engineer. Also some level of separation from the production project will be required. An independent compiler tester will know this and will be able to advise developers accordingly

References

[CompCert] http://www.absint.com/compcert/index.htm.

[EEJR2008] Volatiles Are Miscompiled, and What to Do about It, Eric Eide & John Regehr University of Utah, School of Computing, 2008

[WiDa1989] Wichmann, B. A. and Davies, M, Experience with a compiler testing tool, Report DITC 139/89, National Physical Laboratory, UK, March 1989.

[XYER2011] Xuejun Yang, Yang Chen, Eric Eide, John Regehr, Finding and understanding bugs in C compilers, ACM SIGPLAN Notices, Vol 46, Iss. 6, June 2011.

The following that were referenced above are part of the Compiler Validation series of documents and can be found here
http://www.phaedsys.com/standards/compilervalidation/index.html
C Compiler Validation for Embedded Targets

Repeatability and Reproducibility: Why testers sweat the

Testing Code Generators: A Dark Art

Contact the authors at:
info@phaedsys.com

Phaedrus Systems Library

The Phaedrus Systems Library is a collection of useful technical documents on development. This includes project management, requirements management, design methods, integrating tools 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.