C Compiler Validation: Choosing a Validation Suite

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

C Compiler Validation: Choosing a Validation Suite

Tools for Validating Compilers for Use in Safety-Critical Projects

Compiler validation requires specialist tools. These notes are deigned to help you choose the most appropriate tools for your project and aid you in deciding whether to carry out the validation in-house or by using a third party service..

1. Introduction

As assessors such as the TÜVs and others are beginning to require C compiler validation for critical projects, with some requiring on-target validation, choosing a validation suite is becoming an issue. This paper reviews the main candidates, examining their advantages and disadvantages.

2. Types of Test Suite for C

A C Compiler Validation Suite (CVS) is a set of test programs specifically designed to test a compiler against the requirements of a relevant C language standard. Since 1990, successive versions of ISO/IEC 9899 Information technology --Programming languages --C (from now on called ISO C) have been the C standard, with National Bodies such as BSI, ANSI, DIN etc issuing their own ISO-licensed local editions.

In the 1980's there was no C standard. The K&R books were often regarded as one, but were intended as tutorials. There was, of course, compiler documentation, but compiler vendors added their own extensions and interpretations to C. In the cross compiler market for embedded systems, the MCU architectures sometimes required restrictions and interesting adaptations that gave rise to many of the undefined, unspecified and implementation-defined parts of C. In 1989 the American National Standards Institute (ANSI) produced a standard for C (C89), which was ratified the following year by the International Standards Organisation (ISO) (C90), which continues to maintain the standard, having issued new versions in 1999 and 2011.

Currently (2017) only three validation suites in use were specifically designed for the purpose of validating C compilers against the C standard. They are the suites developed by Plum Hall, Perennial and Solid Sands (formerly the ACE suite). All three were initially developed in the mid 1980's as the C89/C90 was under development. Members of all three CVS teams were members of the ISO and ANSI working group developing the C standard.

The well-known GNU test suite, though containing many useful test programs, is not itself primarily designed to be a validation suite that tests compilers against C language standards. The equally well-known Csmith test system is a tool that pseudo-randomly generates "stress test" programs. We will look at all these two in more detail later.

In addition to the three CVSs there are many other smaller test suites and benchmarks that exercise specific things such as maths and other particular patterns of computation.. Benchmarks normally measure performance, often a very specific and narrow aspect of performance and are not used for compiler validation but will be used by compiler developers to assess performance. Marketing departments like benchmarks but unless your application makes extensive use of the particular thing that a benchmark tests, they are not particularly relevant. We cover benchmarks more fully in a separate document.

3. Hosted vs. Freestanding Implementations

The ISO C language standards distinguish between "hosted" and "freestanding" implementations. What constitutes "hosted" and what "freestanding" is somewhat blurred as we move from bare metal to simple and complex schedulers through micro-kernel systems and RTOS on to full blown, non real time, OS like OSX, UNIX, Windows and Linux. We know what the two ends are but where, precisely, the line should be drawn in the middle, has been, and still is, hotly debated.

The ISO C standard says a hosted implementation must provide all of the standard libraries defined in the standard. A freestanding application need provide only <float.h>, <iso646.h>, <limits.h>, <stdarg.h>, <stdbool.h>, <stddef.h>, and <stdint.h> and need not provide support for complex numbers. The notion of a freestanding implementation was originally put forward to allow for cross-compilers for embedded targets, which in the 1980s were often restricted-architecture 8 bit MCUs. These did not require many of the standard library facilities which would not fit into their functionality. Many did not implement floating point at all. It was not uncommon to have an "integer compiler" where floating point was either not available or was implemented in a separate more expensive "floating point compiler" as many embedded systems used integers or fixed point arithmetic. (https://en.wikipedia.org/wiki/Fixed-point_arithmetic)

Note: In the 1980's the desktop PC was a 16 bit system with an optional and separate maths co-processor.

In practice, as embedded world targets have moved from 8 bit to 16 and 32 bit targets, many cross compiler implementers now provide some, most or all of the other standard libraries that are required in hosted implementations – in particular many implementations provide a cut-down version of the <stdio.h> and other libraries, albeit often in forms with reduced functionality.

One test suite (Perennial) comes in two versions, one for hosted and one for freestanding implementations. The freestanding version covers the language and only those libraries that the ISO C standard requires in freestanding implementations. For those critical systems that deliberately minimise reliance on library functions, that is all that may be required and they can use the freestanding version of the Perennial test suite. Where any embedded system does use cut-down versions of other standard libraries, testing using a subset of the tests for a hosted implementation may be more appropriate.

This is why you need to check exactly what your compiler does and does not do, where it does and does not adhere to the ISO C standard, and come to that, which version of the ISO standard. There is more on this in our paper "Repeatability and Reproducibility: Why testers sweat the details."

4. One Test Per Program and Many Tests Per Program

When running compiler tests, one must ensure that a failed test does not put the environment into a state in which subsequent tests could fail spuriously and produce false negative results (or false positive results.) A simple way to do this is to write the test suite so that it contains exactly One Test Per Program (OTPP). At the end of each test the system is returned to a known state.

OTPP is also valuable for on-target testing when memory limitations mean that large Many Tests Per Program (MTPP) modules cannot fit in the memory – instead testing is carried out by a large number of OTPP files.

But OTPP can also be inconvenient for on-target testing – the total time to run the suite is dominated by the time taken to download the programs to the target and, after running and reporting results, clearing down the target for the next test. Another major problem, given the large number of tests, is the number of write/rewrite cycles the flash memory can handle. On-target compiler tests have failed because the flash memory on the target board has died. Flash memory erase/write cycles have improved a lot but even now (2017) compiler validation tests can still cause flash memory to fail by exceeding their write/rewrite limits.

In practice some CVSs, for example, Solid Sands, have MTPP with tests running to millions in total but constructed in a way that makes the suite usable for on-target testing. The Solid Sands CVS provides means to isolate failed tests to ensure that they do not affect the running of subsequent tests.

MTPP is not ideal but for strenuous testing on development kits, it is currently one of only two practical ways to run very large numbers of tests in an acceptable period. An alternative is to test simultaneously on several target development boards, but this is only reliable when all the development boards are identical and using the same silicon revision for the MCU and other parts.

When a test suite is designed to be MTPP, the suite developer should ideally provide means to select library test programs on a case-by case basis, according to what libraries the compiler under test (CUT) provides in the embedded environment. Therefore an ideal CVS is somewhere between an OTPP and MTPP with the ability to recover from individual test failures and continue testing.

5. Plum Hall CVS

Plum Hall, in 1986, produced a commercial C Validation Suite(CVS). It contains both conformance tests and deviance tests and has been widely used over the last 30 years. It has over 56,000 lines of code and thousands of tests. The test suite is intended primarily for testing hosted C implementations and is of MTPP design. This has led some C cross compiler developers to use it in modified and/or reduced forms to make it more convenient for testing freestanding implementations. The Plum Hall suite continues to be used by compiler developers but has been comparatively little used by embedded systems developers, having apparently lost ground in the market when the Perennial test suite was approved for use in testing compilers for US government projects.

6. Perennial CVSA

The Perennial C Validation Suite(s) (CVSA), first produced in 1986 are of OTPP design and come in two versions, for hosted and freestanding implementations respectively. For projects using compilers that use standard libraries other than those mandated for freestanding implementations, it is best to use a subset of the tests for hosted environments provided in the full-hosted implementation test suite. Depending on the libraries tested and on the target microcontroller, running such tests on-target can take several days which can be a significant disadvantage for some projects.

Currently CVSA provides tests for behaviour covered in all versions of ISO C:9899 from 1990, K&R C, and Technical Reports:

18037 - Extensions to the C Language, support for embedded processors.

19769 - Extensions to the C Language, support for additional character types.

24731 - Extensions to the C Language, Specification for Secure C Library Functions.

24732 - Extensions to the C Language to support Decimal Floating Point Arithmetic.

There are over 69,000 tests in the full suite for hosted implementations. The freestanding test suite has far fewer tests and exercises only those aspects of behaviour required in a freestanding application.

The Perennial test suite is more systematically designed than the Plum Hall suite but has features that some specialists have seen as flaws. The main example of this is that each test program contains code that presents an API to external test control program in order to facilitate management of test runs. Such modifications to the code are not actually necessary since it is not hard to control test runs by using the facilities of external test driver programs and the test host's operating system. Ideally a test should contain only those language constructs that suffice to test a single language feature.

Given that a test driver API is not technically needed, it is hard to see why Perennial's suite uses one, since it requires yet more configuration parameters to get right and provides greater scope for unintended functioning of test programs. Aside from those reservations, however, the Perennial test suite was the first C test suite to be designed for certification-quality validation and provides a good exercise of any C compiler.

7. Solid Sands SuperTest

The Solid Sands SuperTest suite started life as the ACE (Associated Computer Experts bv, Netherlands founded in 1975) test suite in 1984 some two years before Perennial and Plum Hall but on the European side of the Atlantic. The test suite became Solid Sands SuperTest CVS in 2014 and contains more tests than the Perennial and Plum Hall test suites combined (around 800,000 test programs). The suite is of MTPP design but is suitable for on-target testing. Progressive tests probe the functionality of:

- **in**put file character mappings
- **pre**-processing
- **le**xical, syntactic and semantic analysis
- **opt**imisation, code generation
- **li**nking
- **ex**ecution

There are tests for C language operations for different operand types over their permitted ranges of values. Tests are also included for behaviour that the C standard states to be “implementation-defined”. The so-called “depth tests” do extensive boundary-value based testing for different operators, operands, operand types, storage classes and operand value. There is also a regression-oriented group of tests that test for the presence of bugs that have been noted in widely used implementations. Where relevant there are both conformance and deviance tests.

Subsets of the suite can be selected specifically to exercise the requirements of all versions of ISO C from 1990, and even K&R C, also

- IEC 60559:1989 (Floating Point Arithmetic)
- Relevant aspects of C++ as defined in ISO 14882
- C as used for digital signal processing
- C as used on embedded targets

Over the 800,000+ test programs, there are around three million individual tests, making SuperTest the largest C validation suite currently available.

With such a large number of tests, MTPP design is the only practical option for design of the suite to make on-target validation feasible but this still takes some time to run and varies depending on the chosen target and the technical characteristics of the debugger protocols used.

8. Csmith and Obfuscated C

While we have noted that Csmith and the GNU test suites are not purpose-designed standards-based validation suites, they do still have a useful role in compiler testing of which embedded software developers should be aware.

Csmith, as noted in our paper On-Target Stress Testing of C Cross-Compilers, is a pseudo-random program generator or stress tester, that produces a large number of pseudo-randomly generated test cases. The C language is very flexible and legal constructs can be formed in a vast number of combinations. As it is unrealistic to test every possible combination of constructs, an alternative is a random selection of programs that a developer is would be unlikely to write generated by the stress tester.

Csmith is particularly useful for use in conjunction with fixed test suites that do not attempt many tests to specifically stress compiler code generators. The authors recommend running Csmith in addition to a good CVS. Csmith alone will not validate the compiler.

The Solid Sands test suite has a larger number of tests including those designed to test the back end code generators. Therefore Csmith stress testing in conjunction with SuperTest is a very strong combination.

An additional source of stress testing to be used in addition to and not in place of Csmith is the code submitted to the International Obfuscated C Code Contest. Most compiler developers will run the winning code for each year against their compilers, but this will depend on the version of C the winning obfuscated C code is written for and the version the CUT conforms to. However that is one small set of tests - a stress tester can produce many more both quickly and efficiently.

Csmith can, rather counter-intuitively, save sets of the pseudo-random tests for running again as a regression test set. This is useful for compiler developers where a problem is found with the compiler and the corrections to the compiler need to be tested.

9. GNU Test Suites

The GNU C test suites were originally developed for specialised purposes in testing various elements of the GNU Compiler Collection. These tests are useful for C compiler developers, and several use them but they are not particularly useful for users or developers seeking a C validation test, particularly when testing on-target. The tools are not structured as a test suite. As they are not grouped to follow the ISO C structure, they make it difficult to characterise test coverage, nor show what it does, and do not carry out a structured test approach. Validators using the GNU test suites will still need a full CVS and the Csmith stress tester.

10. Other test suites

There are other test suites available though none as comprehensive or widely used as the three previously discussed, and they should only be used in addition to one of the formal CVS's and Csmith. These suites often were created to test a specific aspect on a particular project. For example the Paranoia Test suite (http://www.leshatton.org/index_BE.html) was built to test system arithmetic and Nullstone (http://www.nullstone.com/htmls/brief.htm) for testing specific areas of compiler optimisation. Since anyone, from highly respected software engineers and scientists to undergraduate students, can write some software and call it a "test suite", it doesn't mean they are of equal quality or indeed that they even do what they say they do. One test suite proudly proclaims over 6,500 tests. As noted, others are an order of magnitude, or more, greater.

You do need some sort of provenance for a test suite. The authors of the three CVS we have looked at were on the ANSI/ISO C working group and helped shape the language. They wrote the test suites to prove the language they were also standardising. Some of the other test suites do have good provenance but are generally much smaller, for example Paranoia mentioned above. It is not to say that these other test suites are not useful: often they are but just not in isolation.

11. Proven in Use.

While a formal compiler validation is now technically preferablel, some still refer to "proven in use" justifications for compilers. However, "proven in use" is in reality not really worth anything, as all the compilers that could legitimately claim "proven in use" are also those that are rigorously tested with formal test suites, stress testers and other tools. For example compilers from IAR, ARM Keil and Green Hills who formally test their compilers with formal CVS have many thousands of customers using *the same tested binary*. Many are using these identical compilers on critical projects where they are again formally validated on the project. While these compilers have enough users *of the same binary* to claim: "proven in use" they have already been formally validated by the compiler developers.

On the other hand, while GCC is used by thousands of users it is wrong to call it "proven in use" since, with its diverse back ends, GCC is many different compilers built from different components by many different people. In theory 10,000 users could have 10,000 different compilers, unlike the commercial companies, where a single binary is validated and distributed to 10,000 users. While it is possible to validate a GCC compiler, the validation will only apply to identical copies, not any other builds.

Thus arises the paradox that "proven in use" is not a useful metric: As far as the authors know, all the compilers that can legitimately claim ″proven in use" are also formally validated anyway.

12. Test suites, benchmarks and compiler developers

Of course you can never have enough compiler tests. Commercial compiler developers tend to use one, or sometimes two, of the three test suites mentioned above *and* Csmith *and* often the obfuscated C code *and* additional in house test suites *and* benchmarks. (See our paper on Compiler Benchmarks.) As previously discussed while benchmarks are not a sign of correctness nor overall performance, they do test the efficiency of specific operations, notably floating point operations. These programs (or similar sets) are run in a formal way with all results logged.

The in house test suites tend to be large bodies of source code that previously compiled correctly, or with known problems. These are of value for cross-compilers for embedded targets, where they test compiler or architecture-specific extensions to ISO–C, or the "unspecified, undefined and implementation-specific" parts of ISO C. In addition there are normally regression test suites to confirm previous problems have not recurred. There are also test suites for arithmetic and maths functions (some test suites like Solid Sands also include these) and other in house test sets.

The table shows the additional test software used, by a well-known commercial compiler company. Most other commercial compiler developers have a similar set of test programs They, between them, test most areas of common compiler usage and some specific areas of

Name	Function
blowfish	Blowfish algorithm
bt_stack	Embedded protocol stack
car_navig	Customer application
decrypt	Functions to descramble encrypted multimedia content
dyn_array	Dynamic array allocation
embos_test	Real time operating system
float	Part of customer application, battery charger. Floating point calculations for capacity, diffusion, service hours etc
floattest	Synthesized test of float and integer arithmetic's
generator_controller	Customer application; marine generator controller
gsm_efr	GSM Enhanced Full Rate (EFR) coder/decoder
mars	encryption algorithm
math	Misc. math routines
microwave_sensor	Customer application; microwave sensor for measurement of moisture in bins
mix	DVI ADPCM coder/decoder and Patricia trie (from retrieval) implementation
modeit	Internet application with TCP/IP stack
reed_solomon_ decoder	Reed-Solomon decoder
regexp	Regular expression scanner. Uses search string with special characters to match patterns of text
rijndael	encryption algorithm
serpent	encryption algorithm
sha	secure hash algorithm
s*****meter	Customer application, S*****meter. Device that determines how well the lungs receive, hold, and utilize air, to monitor a lung disease
susan	Image processing to detect the position of edges/ corners for the guidance of unmanned vehicles
temp_display	Customer application; temperature logger

interest to particular industries.

There are a lot of cryptographic tests because these are usually using maths and/or bit manipulations as well as file accesses (input stream and output stream). There are usually formal, often certified, test tools for the cryptographic programs to ensure they not only compile correctly but perform correctly. Since for cryptographic routines speed is also important, performance benchmarks are also run.

A compiler is an extremely complex system and changing one thing in one place and testing just the one thing is not enough. This is because a slight change in one place may have a ripple like effect elsewhere; perhaps causing very subtle changes in behaviour that will change the behaviour of the compiler in some circumstances. The full suites of tests must be run again after every change.

To make test times acceptable, testers usually run test suites as on-host tests using target simulator software in conjunction with a debugging protocol.

Normally compiler developers have a permanent compiler test and validation suite(s) set up on dedicated test computer(s) with staff whose full time job is running the tests.

NOTE: for formal compiler validation you do need a dedicated, configuration-controlled test computer.

13. Choosing a Test Suite

An important principle for designing critical software is: whatever is not there cannot go wrong. Nothing that is not required for critical software to fulfil its intended function should be included. A second principle is: whatever is there should be made as simple as possible (but no simpler). These principles may be applied to test suites themselves.

The authors believe that the Plum Hall suite has been overtaken in coverage by both the Perennial and Solid Sands suites. In practice, and for critical systems development, the choice is between Perennial and Solid Sands.

Perennial's use of an API for external test drivers breaches the first principle for critical software and has been seen by some as clumsy. On the other hand the parts of the test programs that are actually required for testing of language usage were very carefully designed and the suite has proved reliable in use, even if it does do something that is technically unnecessary. A further less-than-desirable feature of the Perennial suite is that it requires a hosted C compiler on the test host to self-check that it has been set up correctly. On Windows platforms it is, on balance, best to use Microsoft's Visual Studio compiler for this purpose. The free version will do but it unfortunately relies very heavily on Windows environment variables and is easily misconfigured, even by experts. In practice this also makes a Windows environment editor a necessary auxiliary tool. Although the free EVeditor is perfectly adequate, we see yet another example of unnecessary complexity, this time indirectly due to Microsoft.

Solid Sands SuperTest also does some things that are not strictly necessary. It requires a POSIX-compliant system API because, among other reasons, it uses UNIX tools within its test driver software. For this purpose the suite uses CygWin when run on Windows platforms. This has to be downloaded separately from the test suite and makes configuration fiddlier on a Windows test host as well as introducing reliance on a CygWin download site for CygWin version information. A compensating advantage, however, is that several modern C cross-compilers are based on GCC and themselves need either CygWin or MinGW to be used under Windows. For these compilers, use of CygWin is a distinct advantage. It would, however, be a little cleaner if SuperTest did not rely on CygWin for test driver implementation.

Both Solid Sands and Perennial test suites come with separate scripts or programs that, in conjunction with the suites themselves, provide a workable compiler validation kit. In both cases too, these separate items could be simplified and, better still, be accompanied by an automated configuration tool. Lack of automated configuration remains a technical weakness of all C test suites at the time of writing (2017). These niggles cry out for Tcl-based test drivers, which would have the merit of being readily portable among test host environments, native UNIX, Windows, or CygWin/MinGW. Unfortunately, neither Perennial nor Solid Sands have chosen to use fully cross-platform drivers.

As regards test coverage, SolidSands SuperTest contains tests that go over and above the basic conformance and deviance tests offered by Perennial. Testing of arithmetic operations is based on boundary-value coverage. There are also tests that attempt to stress the compiler's code generator, notably in exercising register allocation. Although these have limitations, they are useful as a means of bridging, at least partially, the gap between conformance/deviance tests and pseudo-randomly generated stress tests. Typically a fixed test suite is weak at testing compiler back end functions whereas pseudo-random stress test tools such as Csmith are quite good at it.

A simple guideline in selecting between Perennial and Solid Sands is to consider the criticality of the project for which validation is needed and the use the project makes of C's standard libraries. For critical projects using only the minimal libraries required in a free standing implementation, Perennial CVS augmented by the free Csmith stress tester will probably give a good fit to technical requirements. Otherwise, if more than the minimum library is used, Solid Sands SuperTest is probably the better option and can be used in conjunction with Csmith if there is a particular need for stress testing of the code generator.

14. Using a Compiler Validation Service

There is a lot more to compiler validation than buying the test suite, loading it and pressing the "go" button. While for on-host testing, this will usually give a reasonable measure of confidence, for on-target testing or for submission to an external body for a SIL validation it is almost certainly technically inadequate. See our papers:

C Compiler Validation for Embedded Targets

Repeatability and Reproducibility in C Compiler Testing: Why testers sweat the details

Stress Testing Compilers

Code Generator Validation

What is a compiler?

Compiler Benchmarks

In particular read *Repeatability and Reproducibility in C Compiler Testing* (subtitle *Why testers sweat the details*) which explains why you will need a computer solely dedicated to running the test suite with no network/Wi-Fi connections or updates to the OS. Compiler testing, as we stress, has to be both repeatable and reproducible and great care must be taken over seemingly small technical details to achieve this.

In addition to the testing environment the test suite will need very careful configuration as will Csmith and any other additional test sets required. The outputs will need careful recording and documenting. Of course you need to document the test environment and the process to the degree required by the process standard and SIL level for the organisation validating the project. Reliably reproducible compiler validation is not something most software developers can do without significant training.

Many things can go wrong with compiler validation and only those with specialist training and experience in the field really know how to avoid the pitfalls. It takes only a single tiny error to invalidate a full run of a validation suite on a chosen compiler.

Given the cost of a CVS, the time taken to set up the environment and run the tests it is far more cost effective to have an expert do it for you.

C Compiler Validation: Choosing a Validation Suite

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

The Art in Embedded Systems comes through Engineering discipline.