The use of Pre-Tests For On-target C Compiler Validation

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

Compiler Validation 3

The use of Pre-Tests For On-target C Compiler Validation

Many software engineers have heard of compiler validation tests but there is much less awareness of a crucial kind of test known as a pre-test.

1 Introduction

Many software engineers have heard of compiler validation tests but there is much less awareness of a crucial kind of test known as a pre-test. Pre-tests for compiler validation were introduced in the early 1980s by Wichmann and Cziechanowicz (WiCz, 1983) in connection with their testing of Pascal compilers.

The need for such tests has been largely ignored by developers of C compiler validation suites. This is particularly unfortunate since such tests can save a lot of time when setting up validation, even more so for C compilers because C has a large number of variations. For cross-compilers for embedded targets, in particular, if you are doing on–target validation they are essential.

2 What are pre-tests?

As originally proposed by Wichmann and Cziechanowicz, pre-tests are tests that precede the test suite tests to show that a compiler correctly handles a program that contains the same code as an error-handling test apart from the single construct that elicits the error. Thus, for example, a test program might somewhere contain lines such as:

```
a = 0;
x /= a;
```

to test the behaviour on division by zero (typically to help determine the format of run-time error messages). A

corresponding pre-test would then contain the lines:

```
a = 1;
x /= a;
```

and differ from the division by zero case only in the 1 on the right-hand side of the assignment to be. If the pre-test compiles and runs, then there is an increased degree of confidence that the program containing the error is failing only because of the division by zero and not for some other reason.

The key thing here is that before you assume that a test program contains only one error, you first ask whether supposedly "correct" corresponding code actually behaves the way you expect it to. What appears "correct" code may not be handled by the compiler in the way expected, particularly for embedded targets. This may not be at all obvious in the tests or the results. This is particularly important for C whose standard from ISO-C90 to C18 has an annex, G in C 90 and J in C99 to C18, of implementation defined, undefined or unspecified behaviour that runs to over four hundred items.

The ISO C standards leave the implementer wide latitude in how certain things are implemented, hence the long lists of implementation-defined, unspecified and undefined features in the annexes to the standard. It is by no means easy to devise programs that determine the nature of such features by testing in the execution environment. Moreover, it turns out that for C compilers pre-tests have a much larger role in testing such features than was present in the context in which Wichmann and Cziechanowicz originally used pre-tests.

In fact, there are some surprisingly subtle problems in testing diagnosing even well-known implementation-defined and unspecified features such as:

- Whether plain char is signed or unsigned: signed char is always signed and unsigned char is always unsigned but plain char on its own can be either – and this can often be set by a compiler option.
- Whether right-shifts propagate sign bits: again, the compiler may offer an option to set this.

More subtle is that compilers for embedded MCU's will typically add, omit or modify things to work with the physical MCU architecture in addition to the implementation defined items in the C standard. For example there are some 8-bit compilers that implement enum as an 8-bit integer type (unsigned char) rather than a 16-bit int.

Then there are unspecified and undefined items. Most developers don't realise there are these two classifications, much less what they mean or indeed why they are different. An example of unspecified behaviour is the order in which the arguments to a function are evaluated, whereas an example of undefined behaviour is: the behaviour on integer overflow.

Further to confuse things, there is also Locale-Specific implementation. An example of locale-specific behaviour is whether the islower() function returns true for characters other than the 26 lowercase Latin letters.

Developers who are not familiar with these matters and the distinctions among them need to read the version of the ISO-C standard that is relevant to their compilers along with the compiler documentation. Particular attention should also be paid to the 17 (as of C99) "common extensions" also listed in annex G for C90 or annex J for C99-18. Both the compiler writer and the test suite writer will have worked from the official ISO 9899 C language standard, as will any validating organisation. ***Using anything other than the relevant official ISO standard is pointless.*** Draft standards and textbooks are just that, they will be different to the published standard.

This is why compiler validation is best not left to the normal project developers. Compiler validation requires a painstaking discipline with a precise and detailed approach to the appropriate C standard and the compiler documentation. The pre-tests are the bedrock without which any validation will not be valid.

Compiler Validation 3

3 The role of pre-tests for on-target C compiler validation

Neither of the two main C compiler validation suites (Perennial CVSA and SuperTest) contain deviance tests and therefore neither uses pre-tests in the way originally proposed by Wichmann and Cziechanowicz for Pascal compiler testing. Also neither Perennial nor SuperTest contain tests that seek to determine implementation-defined or unspecified characteristics. But this does not mean that pre-tests are of no use for C compilers.

In critical cases, and especially when using high-end static analysis tools, it becomes essential to know what the implementation-defined and unspecified aspects of the implementation are. One example of this might be to determine whether **abs** yields the same result both as a function and as a macro, for which the following test and pre-test might be used:

Test

```
/* TestAbsAfterPreTestExample.c */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int absfunc = (abs)(-1);
    /* will call the abs function */
    int absmacr = abs(-1);
    /* possibly here a macro call */


printf(

\n(abs)
     result same as abs result?

);
     (void)(absfunc == absmacr ?

printf(

yes\n

) :
printf(

no\n

));

     return 0;
}
```

A problem here is that you don't know whether abs is implemented as a macro, so the test is inconclusive. You could easily put in an **#ifdef** to test to for the presence of a definition of abs as a macro but this takes effect in the translation environment, which is not, if you are testing on-target, the execution environment. Moreover, you cannot be sure whether or not the **#ifdef** segment would affect the interpretation of the rest of the code.

A way to address this problem is to use the pre-test:. Precautionary pre-test

```
/* PreTestAbsExample.c */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int absfunc = (abs)(-1);
 /* will call the abs function */
    int absmacr = abs(-1);
 /* possibly here a macro call */


printf(

\n(abs)
    result same as abs result?
```

```
);

      (void)(absfunc == absmacr ?

printf(

yes\n

) :
printf(

no\n

));

      /* pre-test-specific code begins
*/

printf(

\nabs defined as a macro?

);

);

  #ifdef abs

printf(

yes\n

);
  #else

printf(

no\n

);
  #endif
     /* pre-test specific code ends */

     return 0;
  }
```

Note that the Pre-test differs from the on-target test ONLY in adding the additional call to **printf()** and the **#ifdef** segment and that this use of the pre-processor occurs after the code that actually prints the results of the comparing the two results from the calls.

Now, there is a basis for differential comparison. The main on-target test has only the presence of **#include <stdlib.h>** to tell it whether **abs** is a macro or not. If the results of comparing the call results are the same in both programs, then one can reasonably infer that the presence of the **#ifdef** segment in the Pre-test has not actually affected the result of the on-target test but it has also given us the useful diagnostic information of whether or not abs is a macro.

Though this might seem rather paranoid, the author knows of a case in which, using NFS over a large private network, a compiler included a header file from a different compiler on a different machine, leading to a hard-to-trace error. This is exactly the kind of thing one needs to guard against in compiler testing. Wayward configuration control can easily make the results of a large compiler testing effort wholly useless.

When performing on-target testing of C -compilers, there are other kinds of needs for these tests. All compiler testing has to make reasonable assumptions about the compiler in order to have test procedures that can be applied with little or no adaptation to any compiler for which testing is needed.

One assumption in on-target testing of C compilers is that the compiler provides some means for delivering output from the target to the host environment. Another assumption is that means exist to determine when a program running on-target has terminated. A tester uses pre-tests to obtain information on the compiler under test (CUT) to extract the information that enables him/her to set up correct parameters to guide a test control program. It is highly desirable that such information be extracted in a fixed and robust way since ad-hoc methods can be both time-consuming and unreliable.

Core pre-tests for on-target C compiler validation must include programs to determine among other things:

(a) How results can be delivered back to the on-host test control program,

(b) How program termination is detected,

(c) Which libraries are provided in the target environment,

(d) Differences between on-host and on-target behaviour of certain standard li-brary functions.

(e) The behaviour of implementation-defined and unspecified features.

Items (a), (b), and (c) above are assistance mainly to the tester. Items (d) and (e) will typically be of interest to software engineers who need to configure static analysis tools for trustworthy verification of c programs. We now address these matters n turn.

3.1 Delivery of results to the host environment

Most modern C cross-compilers support at least a function-limited **<stdio.h>** in target environments though some may not. This is because for embedded systems the target architecture and system hardware may not support some forms of I/O. For example there may not hard disk, or even a file system to write to and in any case limited memory. Also USB, serial or other comms may or may not be present. Often the debug channel is used. This may be JTAG, BDM or similar. Some JTAG debuggers do have a serial RS232 like communication channel that can be utilised.

Tests will have to be done to discover the size of test that can be fitted in the target memory, speed of download of the test set, speed of recovery of results. This is an often over looked aspect as a test suite will be anything from 50,000 tests upwards to 100,000 tests. Imagine 100,000 cycles of compile, link, download, run, recover results and even if automated this will take time. Some tests will be loaded in groups rather than one at a time. However, depending on target and available memory the selection of groups of tests will need care. Care that the tests can fit, run and the results can be stored in memory and then be recovered.

If the largest single test block can not be fitted in memory, can it be split? Often, where on-target testing is not required an instruction-set simulator is used and the memory spaces can be expanded to be larger on the physical MCU. This is permitted for some validations but not for on-target testing. Unless results can be delivered from the target to the host, on-target validation will not be possible.

The key aim of pre-tests in this context is to determine what I/O facilities actually exist in the execution environment and whether they can be used to deliver results back to test control software running on a separate host.

3.2 Specifying the invocation options

The key thing to determine about program termination is whether the host-target debugging protocol can recognise a **return** from **main** or whether recognition depends on calling the **exit** function with an appropriate argument. Where **exit** is needed but the given test suite uses only **return**, then a stream editing process will be needed to change each final **return** from **main** to a call to **exit**.

The relevant information can be determined by two test programs, one being Hello World with a final return and the other being Hello World with a final exit. In addition one needs to know whether the program termination was normal, and any values returned or abnormal.

3.3 Libraries supported on-target

The ISO-C standard differentiates between the standard library for hosted and self-hosted programs. This causes a lot of confusion. When running an OS like Windows, UNIX, Linux the program is clearly hosted. When the program is single threaded with no form of operating system it is freestanding. However there is a huge grey area as one moves from simple schedulers to the common RTOS of varying complexity. The other big problem is that most compilers for embedded systems

tend to not only include some, many, most, or all of the hosted library. They can provide modified non-standard versions of both the nominally hosted and nominally freestanding standard libraries. Here again a detailed, precise study of the compiler documentation is required. In addition the MCU documentation needs to be examined if maths is involved to see if there is maths, or cryptographic hardware and how the compiler determines if it uses software libraries or the hardware. There may be two versions of the library, one for software only and one for using the hardware maths support.

Pre-tests for whether any particular library is available on-target follow a common form. For each standard library, pre-test programs are constructed to test for the presence of the functions and macros that the library provides. The presence of library functions can be tested by attempting to take their addresses while the presence of macros can be tested by attempting to compile specimen invocations. There should be one pre-test per function and one pre-test per macro.

3.4 Modified behaviour of standard library functions

Pre-testing for the behaviour of standard library functions that differ between host and target environments is less straightforward than the pre-tests already described. Two broad approaches are possible.

In principle, the tester may exclude such matters from pre-testing and leave it to the main test suite to show an error if on-target behaviour differs from on-host behaviour. Since the C standard mandates only a few headers in freestanding environments, this is the strict-conformance testing approach. Since, however, a cross compiler may make extensive modifications to the standard library for embedded use, this approach, though in principle correct, can be impractically laborious.

Alternatively, the tester may seek to determine what the relevant on-target behaviour is. This approach requires a set of programs specifically designed to diagnose implementation-defined, modified on-target standard function behaviour. Such programs can be used as pre-tests or, more conveniently, may be run in addition to the programs of whatever test suite is being used. This, however, requires much more work than the other pre-tests described above. A compromise is simply to determine the on-target behaviour of those functions needed for testing to be possible at all, and of those functions that an application actually uses.

3.5 Implementation-defined and unspecified features

When using a high-end static analysis tool, the tool will need to be given parameters that specify the behaviour of implementation-defined and unspecified language features. For this purpose, a substantial set of tests may be needed, each comprising a pre-test and main test, as in the example of checking whether abs produces the same results as a function or a macro. In practice, the number of such tests is likely to be considerably greater than the number of tests needed to handle each of the preceding four categories of pre-tests. There are some 400 unspecified, undefined and implementation defined items specifically highlighted in the C Standard.

4 A core set of pre-tests

A typical core set of pre-tests that normally suffices for on-target validation using a micro-controller development board will cover:

(a) Checking whether on-target **`<stdio.h>`** can return results from a Hello World program via a USB debugging link between a host and a development board.

(b) Checking how termination status is returned to the host from the development board.

(c) Checking for the presence but not the functionality of all other standard libraries.

(d) Checking the on-target behaviour of all library functions that an application actually uses.

Compiler Validation 3

(e) Determining the nature of any implementation-defined, unspecified, or locale-specific features that a static checking tool needs to know about.

Such a core set determines all of the information that the tester needs to set up reasonably robust test control and to facilitate trustworthy static analysis of software. It does not provide information that may be required to resolve test failures resulting from non-conforming on-target behaviour of standard functions but these matters are almost always most conveniently addressed on a case-by-case basis in anyway.

From the foregoing, it is clear that most of the relevant kinds of pre-tests for C compilers are run to determine how a fully automated test control program should be set up. Hence the procedure for pre-testing is to run the core set of pre-tests/main test pairs using a simple test driver that does not seek to automate everything. This requires manual supervision, which is another reason for using core tests that do not try to determine all on-target library function behaviour. Hence also the set of pre-tests / main-test pairs needs to be small enough so that manually supervised testing is feasible.

The basic test sequence is to determine how program termination is detected, then how results are delivered and only then go on to examine the characteristics of other libraries.

Ad-hoc modification of the test control program may sometimes be required to complete core pre-tests.

5 Testers' responsibilities

The bulk of the above-described kinds pre-tests help the tester by showing whether on-target testing is possible, how much is possible, how much it can be automated and the how a test suite's test control program should be set up. Knowledge of these things is indispensable for on-target testers.

Currently no major C compiler test suite or tool has a coherent set of pre-tests / main test pairs. This is understandable as there is much variation among compilers, target MCUs and on-target implementation. Pre-test requirements must therefore be addressed by the tester, on a case-by-case basis. This further emphasises the need for compiler validation to be done by properly qualified testers.

6 Conclusion and recommendations

The pre-tests are not tasks that can be undertaken lightly or by developers not experienced in Compiler Validation. You need a copy of the applicable ISO-C standard for the compiler, all the compiler documentation and you need to be familiar with them. This is necessary to trace test results to relevant ISO-C clause numbers. There may or may not be a 100% match on this but it is critical that it is documented and understood.

It should be understood by clients for compiler validation that this pre-test work will need to be carried out. You don't just "buy, install and run" a test suite: the correct setting up of the test suite with pre-tests is as crucial as the running of the test suite. Otherwise the compiler validation results will be worthless.

References

Wichmann, B.A., Ciechanowicz, Z.J., 1983. Pascal compiler validation. Wiley, Chichester, ISBN: 0-471-90133-4

The use of Pre-Tests For On-target C Compiler Validation

First edition January 2022

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 IDEs, 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.