Proposed Test Suite Design

Michael Hope (michaelh @ juju.net.nz)

2001-07-13

Abstract

This article describes the goals, requirements, and suggested specification for a test suite for the output of the Small Device C Compiler (sdcc). Also included is a short list of existing works.

1 Goals

The main goals of a test suite for sdcc are

1. To allow developers to run regression tests to check that core changes do not break any of the many ports.
2. To verify the core.
3. To allow developers to verify individual ports.
4. To allow developers to test port changes.

This design only covers the generated code. It does not cover a test/unit test framework for the sdcc application itself, which may be useful.

One side effect of (1) is that it requires that the individual ports pass the tests originally. This may be too hard. See the section on Exceptions below.

2 Requirements

2.1 Coverage

The suite is intended to cover language features only. Hardware specific libraries are explicitly not covered.

2.2 Permutations

The ports often generate different code for handling different types (Byte, Word, DWord, and the signed forms). Meta information could be used to permute the different test cases across the different types.

2.3 Exceptions

The different ports are all at different levels of development. Test cases must be able to be disabled on a per port basis. Permutations also must be able to be disabled on a port level for unsupported cases. Disabling, as opposed to enabling, on a per port basis seems more maintainable.

2.4 Running

The tests must be able to run unaided. The test suite must run on all platforms that sdcc runs on. A good minimum may be a subset of Unix command set and common tools, provided by default on a Unix host and provided through cygwin on a Windows host.

The tests suits should be able to be sub-divided, so that the failing or interesting tests may be run separately.

2.5 Artifacts

The test code within the test cases should not generate artifacts. An artifact occurs when the test code itself interferes with the test and generates an erroneous result.

2.6 Emulators

sdcc is a cross compiling compiler. As such, an emulator is needed for each port to run the tests.

3 Existing works

3.1 DejaGnu

DejaGnu is a toolkit written in Expect designed to test an interactive program. It provides a way of specifying an interface to the program, and given that interface a way of stimulating the program and interpreting the results. It was originally written by Cygnus Solutions for running against development boards. I believe the gcc test suite is written against DejaGnu, perhaps partly to test the Cygnus ports of gcc on target systems.

3.2 gcc test suite

I don't know much about the gcc test suite. It was recently removed from the gcc distribution due to issues with copyright ownership. The code I saw from older distributions seemed more concerned with esoteric features of the language.

3.3 xUnit

The xUnit family, in particular JUnit, is a library of in test assertions, test wrappers, and test suite wrappers designed mainly for unit testing. PENDING: More.

3.4 CoreLinux++ Assertion framework

While not a test suite system, the assertion framework is an interesting model for the types of assertions that could be used. They include pre-condition, post-condition, invariants, conditional assertions, unconditional assertions, and methods for checking conditions.

4 Specification

This specification borrows from the JUnit style of unit testing and the CoreLinux++ style of assertions. The emphasis is on maintainability and ease of writing the test cases.

4.1 Terms

PENDING: Align these terms with the rest of the world.

- An *assertion* is a statement of how things should be. PENDING: Better description, an example.
- A *test point* is the smallest unit of a test suite, and consists of a single assertion that passes if the test passes.
- A *test case* is a set of test points that test a certain feature.
- A *test suite* is a set of test cases that test a certain set of features.

4.2 Test cases

Test cases shall be contained in their own C file, along with the meta data on the test. Test cases shall be contained within functions whose names start with 'test' and which are descriptive of the test case. Any function that starts with 'test' will be automatically run in the test suite.

To make the automatic code generation easier, the C code shall have this format

- Test functions shall start with 'test' to allow automatic detection.
- Test functions shall follow the K&R intention style for ease of detection. i.e. the function name shall start in the left column on a new line below the return specification.

4.3 Assertions

All assertions shall log the line number, function name, and test case file when they fail. Most assertions can have a more descriptive message attached to them. Assertions will be implemented through macros to get at the line information. This may cause trouble with artifacts.

The following definitions use C++ style default arguments where optional messages may be inserted. All assertions use double opening and closing brackets in the macros to allow them to be compiled out without any side effects. While this is not required for a test suite, they are there in case any of this code is incorporated into the main product.

Borrowing from JUnit, the assertions shall include

- FAIL((String msg = "Failed")). Used when execution should not get here.
- ASSERT((Boolean cond, String msg = "Assertion failed"). Fails if cond is false. Parent to REQUIRE and ENSURE.

JUnit also includes may sub-cases of ASSERT, such as assertNotNull, assertEquals, and assertSame.

CoreLinux++ includes the extra assertions

- REQUIRE((Boolean cond, String msg = "Precondition failed"). Checks preconditions.
- ENSURE((Boolean cond, String msg = "Postcondition failed"). Checks post conditions.
- CHECK((Boolean cond, String msg = "Check failed")). Used to call a function and to check that the return value is as expected. i.e. CHECK((fread(in, buf, 10) != -1)). Very similar to ASSERT, but the function still gets called in a release build.
- FORALL and EXISTS. Used to check conditions within part of the code. For example, can be used to check that a list is still sorted inside each loop of a sort routine.

All of FAIL, ASSERT, REQUIRE, ENSURE, and CHECK shall be available.

4.4 Meta data

PENDING: It's not really meta data.

Meta data includes permutation information, exception information, and permutation exceptions.

Meta data shall be global to the file. Meta data names consist of the lower case alphanumerics. Test case specific meta data (fields) shall be stored in a comment block at the start of the file. This is only due to style.

A field definition shall consist of

- The field name
- A colon.
- A comma separated list of values.

The values shall be stripped of leading and trailing white space.

Permutation exceptions are by port only. Exceptions to a field are specified by a modified field definition. An exception definition consists of

- The field name.
- An opening square bracket.
- A comma separated list of ports the exception applies for.
- A closing square bracket.
- A colon.
- The values to use for this field for these ports.

An instance of the test case shall be generated for each permutation of the test case specific meta data fields.

The runtime meta fields are

- port - The port this test is running on.
- testcase - The name of this test case.
- function - The name of the current function.

Most of the runtime fields are not very usable. They are there for completeness.

Meta fields may be accessed inside the test case by enclosing them in curly brackets. The curly brackets will be interpreted anywhere inside the test case, including inside quoted strings. Field names that are not recognised will be passed through including the brackets. Note that it is therefore impossible to use some strings within the test case.

Test case function names should include the permuted fields in the name to reduce name collisions.

4.5 An example

I don't know how to do pre-formatted text in LaTeX. Sigh.

The following code generates a simple increment test for all combinations of the storage classes and all combinations of the data sizes. This is a bad example as the optimiser will often remove most of this code.

```
/** Test for increment.
  type:  char, int, long
  Z80 port does not fully support longs (4 byte)
  type[z80]:  char, int
  class:  "", register, static */

static void
```

```
testInc{class}{types}(void)
{
  {class} {type} i = 0;
  i = i + 1;
  ASSERT((i == 1));
}
```