TUnit Quick Start Reference

David Moss
Rincon Research Corporation

Building a TUnit Test Suite

1. Create a new directory to place your test suite in.
2. Create a new configuration file as the main entry point to your test / application.
3. Include all the components you need to run your test, and include at least one instance of TestCaseC. Be sure to rename TestCaseC to the name of the actual test. Wire everything up to do the test.

```
configuration MyTestC {
}

implementation {
  components MyTestP,
         new TestCaseC() as TestMultiplyC;

  MyTestP.Test1 -> TestMultiplyC;
}
```

4. Create a module to run the test. Be sure to include `#include "TestCase.h"` at the top. The test begins when you get a run() event. At that point, you can assert anything you want. Let the test know when it's done by calling back done().

```
#include "TestCase.h"
module MyTestP {
  uses {
    interface TestCase as Test1;
  }
}

implementation {

  event void Test1.run() {
    assertEquals("Wrong answer.", 16, 2*2*2*2);
    call Test1.done();
  }
}
```

5. Create a suite.properties to tell TUnit which test runs this test would apply to (for instance, the example above only applies to test runs using a single node). Create a Makefile to compile the application.

6. Execute the test by running the TUnit Java program directly inside the test suite's directory, or by running the Ant build script to run all the tests. Only the ant build script will create HTML reports.

   - `java com.rincon.tunit.TUnit` (from any location)
   - `ant` (from the directory containing build.xml)

Building a Multiple-node TUnit Test Suite

1. Wire up your main configuration, including at least one TestCaseC instance. Be sure your module uses SetUpOneTime and TearDownOneTime interfaces, which can be pulled from any single TestCaseC instance:

```
configuration MyRfTestC {
}

implementation {
  components MyRfTestP,
         new ActiveMessageC,
         new AMSenderC(0),
         new TestCaseC() as TestRfC;

  MyTestP.SetUpOneTime -> TestRfC;
  MyTestP.TestRf -> TestRfC;
  MyTestP.TearDownOneTime -> TestRfC;

  MyTestP.SplitControl -> ActiveMessageC;
  MyTestP.PacketAcknowledgements -> ActiveMessageC;
  MyTestP.AMSend -> AMSenderC;
}
```

2. Setup the wiring for the module:

```
module MyRfTestP {
  uses {
    interface TestControl as SetUpOneTime;
    interface TestCase as TestRf;
    interface TestControl as TearDownOneTime;

    interface SplitControl;
    interface PacketAcknowledgements;
    interface AMSend;
  }
}
```

3. Turn on the radio during SetUpOneTime. This will run on every node in the test. Be sure to let TUnit know when SetUpOneTime is done().

```
event void SetUpOneTime.run() {
  call SplitControl.start();
}

event void SplitControl.startDone() {
  call SetUpOneTime.done();
}
```

4. Run the test. Here, we'll send a message from the driving node to a helper node, making sure we get an acknowledgement back. Since we

know the driving node getting the run() command is address 0, any other node in the test will have address 1.

Alternatively, we could setup a Receive event in the helper node, and let it assertSuccess() whenever it receives a message.

Be sure to let TUnit know when your test is complete.

```
event void TestRf.run() {
  call PacketAcknowledgements.requestAck(&myMsg);
  call AMSend.send(1, &myMsg, 28);
}

event void AMSend.sendDone(message_t *msg, error_t error) {
  assertTrue("Didn't get an ack.", call
PacketAcknowledgements.wasAcked(msg));

  call TestRf.done();
}
```

5. Be nice to future tests by turning off all radios when the test is complete. Java will issue this command, so all nodes will run TearDownOneTime. Be sure to let TUnit know when TearDownOneTime is complete.

```
event void TearDownOneTime.run() {
  call SplitControl.stop();
}

event void SplitControl.stopDone() {
  call TearDownOneTime.done();
}
```

Assertions

`assertSuccess();`

The test being executed was successful.

`assertFail(failMsg);`

The test being executed failed, including the reason why.

`assertEquals(failMsg, expected, actual);`

Fails the test if expected != actual.

`assertNotEquals(failMsg, expected, actual);`

Fails the test if expected == actual

`assertResultIsAbove(failMsg, threshold, result);`

Fails the test if result <= threshold

`assertResultIsBelow(failMsg, threshold, result);`

Fails the test if result >= threshold

`assertCompares(failMsg, expectedArray, actualArray, length);`

Fails the test if the contents of the expectedArray != the contents of the actual array.

`assertTrue(failMsg, condition);`

Fails the test if the condition is false.

`assertFalse(failMsg, condition);`

Fails the test if the condition is true.

`assertNull(pointer);`

Fails the test if the pointer is not null, logging the name of the pointer.

`assertNotNull(pointer);`

Fails the test if the pointer is null, logging the name of the pointer.

TestCase Interface

```
interface TestCase {

  event void run();

  async command void done();

}
```

TestControl Interface

TestControl is used as SetUpOneTime, SetUp, TearDown, and TearDownOneTime.

```
interface TestControl {

  event void run();

  command void done();

}
```

Statistics Interface

Statistics must be sent one after another! If you have multiple statistics to collect, log one and wait for the logDone() event before logging the next.

```
interface Statistics {

  command error_t log(char *units, uint32_t value);

  event void logDone();

}
```

Ant Options

ant <build task>:

- **tunit**
  - Default task, run all tests, generate reports, and toggle lava lamps if connected.
- **tunitrerun**
  - Reruns all tests that just failed
- **tunitreport**
  - Generates HTML reports from the last test runs
- **tunitfeedback**
  - Toggles lava lamps to display the last results, if connected

Questions.

Q: Where can I place assertions in my test?

A: Assertions can be placed anywhere a TestCase will execute code. TestCase.h must be #included at the top of your file. If you place assertions in areas where the TestControl-type interfaces will execute, you may get strange results.

Q: How do I handle multiple TestCase's executing the same code?

A: Use a state machine:

```
event void Test1.run() {
  call State.forceState(S_TEST1);
  runTest();
}

event void Test2.run() {
  call State.forceState(S_TEST2);
  runTest();
}

void runTest() {
  if(call State.isState(S_TEST1)) {
    assertSuccess();
    call Test1.done();

  } else if(call State.isState(S_TEST2)) {
    assertTrue("This is impossible!", call State.isState(S_TEST2));
    call Test2.done();
  }
}
```

Q: What's the default queue size that stores assertions to be sent over serial?

A: 5 messages. The text used in assertions may be spread across several messages. If you need more queue space to correctly assert all the results, include the following in your Makefile:

```
CFLAGS += -DMAX_TUNIT_QUEUE=10
```

Q: I keep getting timeouts in my tests

A: Some part of your test is not calling done() like it should. All interfaces, including SetUpOneTime, SetUp, TearDown, TearDownOneTime, and TestCase ***MUST*** call back to their respective interface's done() command when they are finished. TinyOS and nesC are split-phase and event driven, so there aren't many options around this requirement.