Cradle
(aka “tiny bed”)

A Testbed for TinyOS
Outline

Requirements of a testing system
  General
  TinyOS specific issues
Types of tests
A little theory
The framework
Particular tests.
Goals of a Testing System

Stability

Catch minor bugs – before they are major.
Catch major bugs – before they are released.

Flexibility

Allow changes without fear of regression.
Encourage re-factoring and re-engineering – or at least not discourage it.

Design Documentation

Tests, more than source, provide a reference for intent of code.
Capture knowledge of subtle issues.
General requirements

Writing (and organizing) tests
- Many types of tests (described shortly)
- Hierarchy: allow tests to be grouped
- Dependencies: prevent cascades of failures
- Pre- and post-conditions on tests

Running tests
- Shouldn't be hard to use
- Selection: run subsets of tests
- Encapsulation: tests shouldn't crash the framework
- Collation: results stored for later use
- Presentation: allow selection of form
General: Writing a test

A test should encapsulate one problem, clearly explained.

Tests need to be automatic (obviously) so they need to have clearly defined results. Some “standard” ones:

Success – Test succeeded.

Failure – Test failed.

Unexpected – Test was not actually run due to failure of a precondition. (Perhaps it's unimplemented 😊).

System failure – Testing system failed internally.

Need well defined conditions for each result state.
TinyOS Requirements

“Well-defined”? Not so easy.
Nondeterminism makes testing hard: # of states potentially unbounded.

Big problem: How do you ensure **repeatability**?

Let's think about this....
Types of tests

Test operate at a wide variety of scales:

Unit tests (component scale): Test all features (interfaces) of one component.

Module scale: Test features of combinations of components.

(Not the same as a TinyOS module)

Application scale: Test features of an actual application as a whole.

At the bottom scale these can be made fairly deterministic.
Component (unit) tests

Component tests should test every command and event on a component.

Configurations are the point of interface with the world:

configuration ADCC
{
    provides {
        interface ADC[uint8_t port];
        interface ADCCControl;
        interface StdControl;
    }
}
implementation
{
    components ADM, MSP430ADC12C;

    ADC = ADM.ADC;
    ADCCControl = ADM;
    StdControl = ADM;
    StdControl = MSP430ADC12C;

    ADM.MSP430ADC12Single -> MSP430ADC12C.MSP430ADC12Single[unique ("MSP430ADC12")];
}
Component test - example

msp430.ADCC wiring:

ADC C

StdControl

ADC [uint8_t]

ADCControl

StdControl

ADC M

MSP430 ADC12 C
You can rewire to test components. Same as any other renaming: change -Iincludes
Component tests - Mocks

*Mock* — test components.

Resemble original components in interfaces – but have a restricted implementation.

At component level mocks generally feed the component a range of inputs.

Can be crafted to fail under particular circumstances.

Mocks are designed to try and facilitate both black-box and *white-box* testing (replace internal components.)

Can do similar things with C functions, if they are included.
Module tests

This technique can scale up to larger collections of components.
Not just MSP430 ADC or Radio, but entire sensor or radio stack.
Scale of operation depends on what you want to test...
This obliges thinking about system invariants.

Con:
As number of tests grows, so do number of mocks.
Can end up with one entire set of test components (one new directory) for each test.
Over time build libraries of reusable mocks.
Application scale is the scale of a program you would actually deploy (*Surge*, GDI, ...) As you reach this level invariants become less about details, and harder to specify.

In beginning best to write tests at this scale. However application tests often dependent on good behavior at lower scales.

At this point still focused on trying to be deterministic: deploying under carefully crafted conditions.

Application scale becomes focused on measurement.
Invariants

Invariants are one of the keys to good testing. Describe system behavior and that it should do. That means explicitly describing what it shouldn't do.

In testing unexpected success is actually failure. And expected failure is actually success.

Make mocks which specifically cause failure conditions.

When making changes, add tests which fail if change is unsuccessful. Old code should fail the test.

Important: Focus on repeatability.
Important: Over time this collection of thou shalt nots grows to define the system better than code itself.
Types of tests (more)

How do you test an application *in vivo*?

aka How do you do non-deterministic invariants?

Functional tests: Black box: Feed application values, and measure.

Stress tests: Long running functional test, ideally with wide range of values.

Tricky to define correct inputs and outputs.

End up needing ranges of values.

Not obvious what correct values are.

Statistical tests: Designed to gather data on correct behavior to compare with.

Dependent on correct behavior at lower levels.
Good ways to measure expected outputs? Open question. Basis of work.

Gathering data into common format and generating significant samples.

Tests feed samples back: clear definitions of invariants still of key importance.

Good tools for measurement hugely useful:

SNMS

Statistical methods?

(XML based results – lots of tools.)

*In vitro*, mocks can do fine-tuned gathering.
“Quality management” tests.
http://www.codesourcery.com/qmtest/
Framework in Python (open source)
Tests can be written in any language (almost)
Hierarchy: tests can be placed in suites.
Interaction: Command-line or web browser.
Encapsulation: Tests can run in own threads, processes, and hosts.
Collation: Tests stored in database – of your choice (XML in our case.)
Presentation: Web pages. Can use XML to change.
Individual tests have a set of properties you apply before you run (analogous to Java Beans...)

- Environment variables
- Temporary files
- Actual test programs (in whatever language).
  - Provides binary, shell, and Python scripts.
  - Can run other programs (e.g. Java) inside shells.
  - User can create new test types with new properties. (Java tests.)

Tests are boolean – They return SUCCESS or FAILURE (plus UNEXPECTED or ERROR)
QMTest – Test suites

Suites are collections of tests, useful for grouping. Suites can be used to establish invariants that apply to all their tests. Like tests, have a return value.

Directories treated as automatic suites:

tos.apps.{mica2, telosa, pc}

User can create new test suites with new properties.

Tests can be dependent on other tests. Suites can be built from sets of dependencies.
QMTest – Test results

Results from tests are return codes, and error messages.

Format customizable:
- Raw text
- Python pickles.
- XML

Users can create new formats.

Collated and stored in a database.

Multiple database types:
- SQL (e.g. MySQL, Postgres, <your choice here>)
- XML

Users can create new database types.
QMTest – Test targets

“Target” is QMTest term for program execution contexts.

Test programs should not corrupt test framework.
May want to run tests on separate machines.

Types of targets:
- Standard - synchronous
- Thread – One test per thread
- Process – One test per process.
- RemoteShell – One test per chosen host.

Yes, you can create your own targets.
Resources are external system resources that you don't want to reallocate for each test.

Examples:
- Database connections
- Set of input files
- Motes with loaded programs
- Running serial forwarders.

Resources are set at test time, but each run can have a *context* provided which sets test specific properties. (Analogous to Java properties files).

Resources can be test dependencies. These are not as precisely definable as one would like.
Either command line, through web interface, or as remote server.
Can provide context at runtime.
Can select any subset of tests via hierarchical names (directories and suites).

Targets not chosen at runtime.
New resources not chosen at runtime.
Web interface lets you see results in a fairly useful manner.

Since data is stored in databases, can create own presentation formats.

XSL over XML results obvious here.

Not high priority: Tinderbox like interface:

http://tinderbox.mozilla.org/showbuilds.cgi?tree=Firefox
QMTest example – Test results
QMTest example – Test Suite
QMTest example – Test creation
QMTTest - Overall

Still learning it.

Pros:

Very customizable and readable (good code)
Flexible setup
Not hard to use.

Cons:

Documentation not ideal.
Not specifically designed for systems like ours (but nothing freely available is.)
Test setup

Still evolving.
tinyos/
   components/...
   platforms/... (including simulator)
libraries/...
applications/...

Under each test
Mock components
Makefile
Data files for input
Presentation formats.
How to write tests

Later...
What do we test?

So what do we test?
- Out of resources?
- Resource conflicts?
- Wiring problems?
- Expected patterns of events?

Packet loss patterns?
.... ?