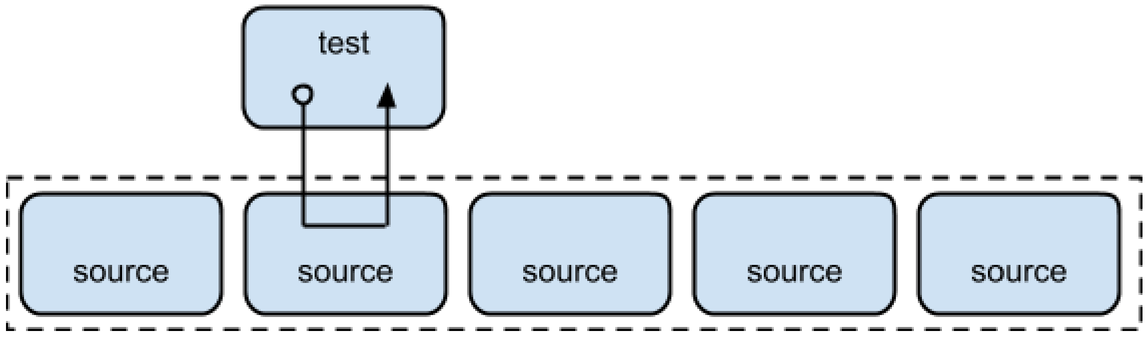
What’s the difference? Unit, Integration, and System Testing

This course almost exclusively addresses unit testing. Of course, there are other types of testing (see the document *Testing Term Potpourri*). And we call out two additional types of testing in the main course materials: integration testing and system testing. Here we will give you a detailed understanding of the distinctions among unit, integration, and system testing.

Before diving in let us briefly repeat a significant line of thinking in the main course. Despite your inclination otherwise, unit tests (and integration tests) are usually best executed *off* target through cross-compiling and/or simulators. System tests, however, are run *on* target. This document does not cover this reasoning. See *Lecture 9: Breaking It Down* for much more on the big picture of on-target and off-target test execution.

Unit Testing



Let’s review the key concepts of unit testing:

- Unit tests exercise a small piece of source code *in isolation* from other source code. For example, a signal processing filter might be composed of several mathematical and data manipulation steps. Each step can be unit tested apart from the other steps.
- Unit tests exist in parallel to your source code. That is, in almost all circumstances your source code and test code do not mix. In certain circumstances you may introduce features or access means to allow testing but no actual test code exists intermixed with your source code (e.g. no `ASSERT` macros inside your source code).
- Unit test code is almost certainly written in the same language as your source code and built and executed with the same development tools.
- Unit tests are composed of truth assertions. Expected states and values are compared to actual states and values. A discrepancy is a test failure and causes the test code to

terminate and log the error. To take a look under the hood to better understand this, see MinUnit — A minimal unit testing framework for C in 3 lines of code.

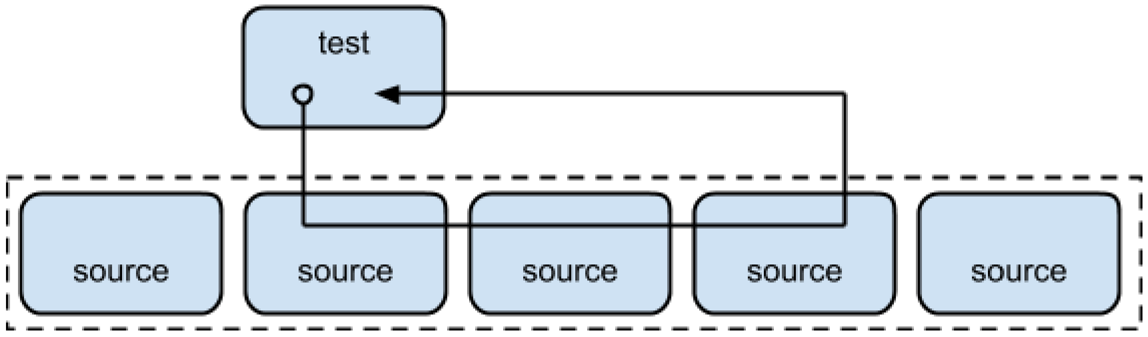
- In addition to verifying your logic, operations, and memory accesses, unit testing—particularly with mocks and fakes—allows you to test difficult to exercise cases in your code. For instance, unit testing can verify memory allocation error handling impossible or highly impractical to produce in a live system.
- Unit testing can be accomplished by hand and ad hoc. However, because the pattern of assembling truth assertions and reporting the results of executing them is so simple and repeated so often, testing frameworks are built to simplify writing unit tests. Frameworks exist for essentially every programming language.

Integration Testing

Integration testing can be thought of as an expanded but specialized form of unit testing. Integration testing occurs relatively infrequently. If a project has thousands of unit tests, that same project may have only tens of integration tests.

In certain cases, despite thorough unit testing of *individual* source units, the *composition of multiple* source units may yet contain serious flaws. To catch such bugs we use integration tests. Integration tests are often needed when large chunks of memory are manipulated by multiple, successive operations. Protocol handling, string builders, and complex data structures are examples. In these instances, it's possible to "lie to yourself" in indexing the memory block. Your source code and test code may both exercise and verify the same index value, but what if the index itself is wrong? Complex pointer operations can cause similar issues. Consequently, an area of memory may suffer a gap in data or certain locations may be erroneously overwritten or any number of bad things with pointers may befall your system.

So instead of testing a single source module in isolation we test multiple interrelated source modules all together with our unit testing rig. That is, our integration tests trigger multiple operations in multiple modules, and we verify the result of the sequence of source operations.



In some cases the complexity of your code may warrant unit testing all the source modules individually and also integration testing the source modules interoperating together. In other cases, a single set of integration tests without individual unit tests may be sufficient to thoroughly test your source code. In the case of the former, be careful to not over-test. Integration testing is meant to verify the big picture operations all the way through and not all the nooks and crannies (that's for unit testing).

Note: To create an integration test, you configure your test build environment to link together multiple compiled source object files with your test code. You'll accomplish this with conventions (e.g. `#include`'ing in your test file the headers of the source files to be compiled and linked) and your build tools (e.g. scripting, make, Ceedling). See *Test Builds: Using A Toolchain and Build Manager for Fun and Profit* for more on this and Ceedling documentation.

Key concepts of integration testing:

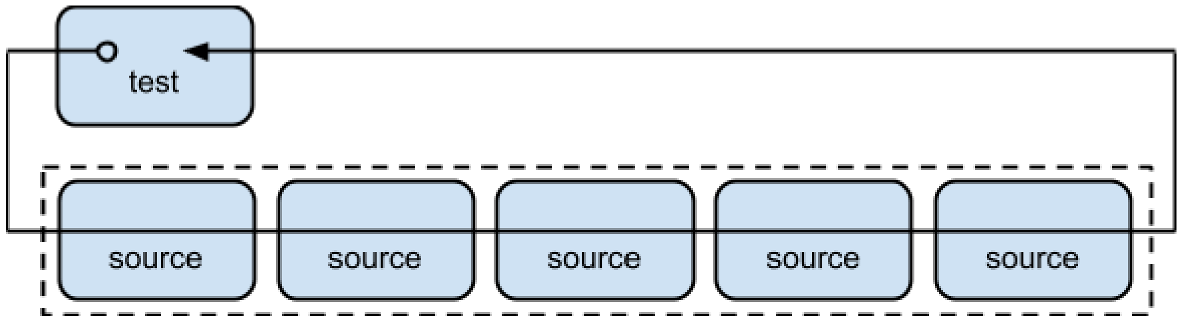
- Integration tests exercise *multiple interrelated modules* of source code in *isolation* from other source code.
- Like unit tests, integration tests exist in parallel to your source code. The test verifies the source code by way of the results of the source code operations (e.g. the contents of a block of memory).
- Integration test code is almost certainly written in the same language as your source code and built with the same development tools.
- Just like unit tests, integration tests are composed of truth assertions. Expected states and values are compared to actual states and values. A discrepancy is a test failure and causes the test code to terminate and log the error.
- Integration testing can be accomplished by hand and ad hoc. However, if you are already employing a unit testing framework, integration tests are best and most easily implemented using that same unit testing framework (see note addressing integration testing builds on preceding page).
- A simple naming convention of your test code files is really all that's necessary to distinguish integration tests from unit tests. As far as your build environment is concerned, there is no meaningful difference between integration and unit tests. The distinction is largely an abstraction apparent and useful only to developers.

System Testing

System testing verifies end-to-end features of your entire system. A system test presents input in the form of files, network streams, button presses, logic line level changes, etc. to your system to stimulate a response that is then captured and compared to expectations.

Unit tests are the lowest level of testing while system tests are the highest. Unit tests can, for example, exercise the logic of error condition handling that may be difficult or impossible to simulate in a live system. Conversely, system tests demonstrate that all your code together accomplishes the desired goals (features) of your product that simply cannot be done at the scope of a unit test. For instance, a system test can verify that a button press triggers a motor output and changes LED status lights.

Unit tests and system tests operate at different scopes and accomplish different goals. That said, of course, there is some overlap between system and unit tests given that the source code exercised in a system test is invariably also tested by unit tests.



Key concepts of system testing:

- System tests exist separately from your source code and, in fact, exist entirely outside your functioning software product. Each system test verifies a feature of your system as a whole. With a system test you're running a live system and interacting with it rather than selectively compiling source code together with test code.
- It's entirely common for system tests to overlap to some degree as resources in your product are invariably shared among features. Unit tests generally overlap very little.
- Implementing system tests may motivate adding features to your product in order to better exercise your product under test. This may mean adding extra protocol commands, memory dumps, UI captures, or specialized reporting modes. Note that these can usually be conditionally included or provisionally disabled for memory space and security reasons.

- System testing embedded systems nearly always entails external hardware components driven by your system tests. For instance, at minimum you will likely need digital and analog I/O boards wired to your product to simulate button presses and sensor readings.
- Though you may have a unit testing framework at your disposal, it is inappropriate and a poor match for system testing:
    - Unit tests fail and log at the first failed assertion encountered. However, in system testing you almost certainly want to exercise as much of the system at one time as possible. If one test case of a feature fails, you'll likely want to see the results of the rest of the test cases for that feature as well to look for patterns and interactions of failures. The core mechanic of unit testing is mismatched to system testing.
    - Further, in the course of creating stimulus for your system tests you will almost certainly want a more powerful and flexible language to work in than that of your unit testing. For instance, producing and capturing USB traffic with a scripting language and a pre-existing library is far easier and more productive than doing so in C (if, for example, you were attempting to employ Unity for system testing).
- System testing is most often accomplished with scripting languages and ad hoc assemblies of software libraries (and hardware components). Your system testing environment is something of its own software project and will generally grow and mature alongside your embedded software product.
- The desire to package up and automate system testing marries nicely with revision control, versioning, Continuous Integration, well defined release build procedures, and other such good practices.

Except in environments with highly specified behaviors and protocols (e.g. web, enterprise systems) system testing is often accomplished with an ad hoc framework built for your project's specific needs. This is especially true in embedded systems work. Setting up, executing, and reporting a test case is simple to abstract—especially in a high powered language available to you outside your embedded source code environment. The particulars of stimulating input and collecting results will demand custom code specific to your product. As such, in embedded software development the bulk of a system testing framework and corresponding hardware I/O components will be created and assembled by the developers building the product to be system tested. If you are working with the same sort of product with the same sort of hardware over and over again, you may end up crafting a single system testing framework for internal use.

The desire to automate system tests naturally motivates the adoption of versioning procedures and automated build systems on top of source code management (e.g. git). Building code and loading it onto a device at a developer's workstation is risky and cumbersome at any level of testing beyond debugging. Continuous Integration and automated build systems are nearly a necessity with system testing of any significance (see our supplemental document *Test Builds: Using A Toolchain and Build Manager for Fun and Profit*). In general, good practice entails setting up a “gold master” system that is not a developer machine dedicated to running all tests and generating final production artifacts (i.e. executable binaries). This promotes good environment replication and limits “magic” happening through a developer’s local environment setup.