



Techniques for Unit Testing Embedded Systems Software

Testing embedded system software presents a unique challenge. Since the software is being developed on a different platform than the one it will eventually run on, you cannot readily run a test program in the actual deployment environment, as is possible with desktop programs. There are several explanations for this disconnect. Engineers may not yet have target hardware, hardware may cost too much to give software developers access to it, the full environment may be difficult to replicate in a development shop, and so forth.

This paper provides an overview of how unit testing can help developers of embedded systems software address this challenge. In a nutshell, it recommends leveraging stubbing to perform a significant amount of testing from the host environment or on a simulator. This allows you to start verifying code as soon as it is completed—even if the target hardware is not yet built or available for testing. As a result, the majority of the problems with the application logic can be exposed early—when error detection and remediation is easiest and fastest—and target testing can focus on verifying the interface between the hardware and the software.

Why Unit Test Embedded Systems Software?

One very effective way of testing software for an embedded system is to apply unit testing— or, more generally, API testing— in the host environment or on a simulator. This allows testing to start much earlier (concurrent with code development) and largely decouples the testing task from the availability of target hardware. One of the premises of unit testing is that code is tested in isolation from the rest of the system, which is emulated by stub functions in such a scenario. In this manner, most of the functionality of the code under test can be verified independent of the rest of the system, without running on the target hardware. This has two extremely important benefits for embedded developers.

1. Unit testing lets you start the test cycle before the hardware is available, and you can perform the initial testing directly on the development platform (rather than on the target). Early testing gives the team more time to find and repair defects. In addition, early testing distributes test efforts across the product-development cycle and helps prevent the 11th-hour testing rush.
2. Unit testing promotes a "divide-and-conquer" strategy that lets you partition complex systems so they can be tested in quasi-independent modules. Test tools manage any module-to-module software dependencies and use stubs to simulate them.

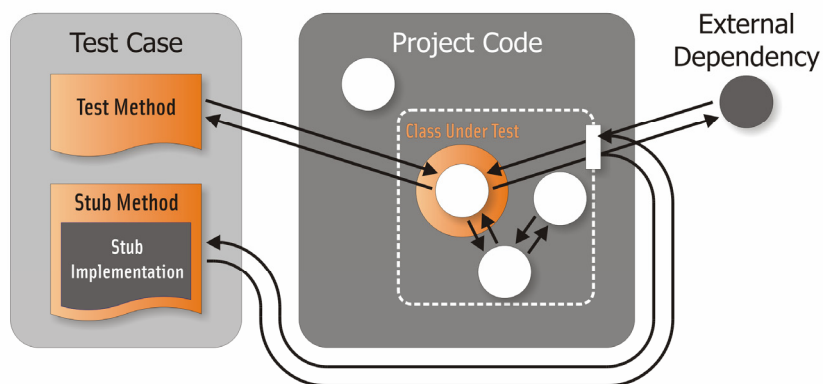


Figure 1: By using stubs, developers can test code under realistic and controlled conditions without needing hardware or software that might not be available until later in the project. The test environment substitutes a stub for external dependencies.

Protecting Code Integrity as Applications Evolve

One of the most significant worries for developers of complex systems is that code modifications might change or break existing functionality. To address this concern, you can create a baseline unit test suite that captures the project code's current functionality. To detect changes from this baseline, you run your evolving code base against this test suite on a regular basis. Because unit tests can test parts of the system's code in isolation, such a regression suite can be continuously executed without having access to the target hardware. This type of testing does not exclude separate application regression tests, which test the overall application.

The resulting test suite serves as a change-detection safety net; you can rest assured that if you accidentally break existing functionality, you will be notified immediately. As new target hardware becomes available, you can leverage your host-based tests to validate that the code will operate properly on the target hardware under realistic conditions. Running the regression suite on a host system does not diminish the need to automate the system tests on the target hardware to the equivalent degree.

Verifying Error Handling

System test scenarios are further complicated by requirements for reliable error handling in consumer products. Since the use scenarios cannot be reliably predicted, systems cannot be designed and tested just for the nominal case. Rather, the system must be verified to handle a broad range of incorrect and unexpected inputs.

Error testing is also much simplified by utilizing unit testing with stubs. In general, testing error conditions at the application level can be very time consuming because putting the application into the "proper state of error" may require preparing relatively complex input data and putting the application into the appropriate state, out of a large state space. In contrast, it is very easy to test error handling for a given function using an approach of "error simulation."

For example, a function that has error handling connected with its inputs is easy to test:

```
float signalToNoiseRatio(float signal, float noise, MODE mode)
{
    if (MODE_MEASUREMENT == mode)
    {
        ...
        if (signal < 0 || noise < 0)
        {
            handle_bad_data();
        }
    }
}
```

In this case, it is simple to test the call to `handle_bad_data()` in context because the expression of the corresponding `if` statement is controllable from the inputs to the function.

More often, however, control conditions are not directly controllable from the function interface, but rather depend on a specific system state, as in the example below. Putting the system in that error state may be quite complicated, or may even involve polling the status of a device interface, so this condition needs to be simulated in a test case.

```
float shutDown()
{
```

```
if (uploadingData())
{
    userMessage("Cannot execute shutdown while uploading data");
    recoverShutDown();
}
else
{
    // shut down indeed
}
}
```

Using advanced test tools that support “smart stubs,” unit testing for complex error conditions is no more difficult than the previous case. “Smart stubs” allow code execution for both the original function being stubbed, as well as implementation of specific behavior necessary for testing. Practically speaking, while the application being tested is clearly not in the error state, the specific function being tested is invoked as if the specific application error actually occurred – hence the term “error simulation.” In the above example, testing the error handler requires the stub for `uploadingData()` to have at least one case when it returns TRUE.

Additional Considerations

Testing on a host system may imply that the compiler used to build the code differs from the compiler used to produce code for the target hardware. If the cross-compiler vendor also supplies a compiler for the development platform (for example, the native compilers from Green Hills Software), take this route. Alternatively, you can freely use the GNU Compiler Collection (GCC) available for many host systems. Although keeping the code portable between the host and target compilers may slightly increase the software-maintenance cost, the benefits of early testing outweigh this expense.

Unit testing is unlikely to uncover error conditions caused by synchronization errors at the application level or errors that occur at the interfaces with real devices. However, in the development of embedded software, unit testing helps you identify many types of defects much earlier, thus improving the overall efficiency of their system development and removing test bottlenecks.

Automating Unit Testing with C++test

One option for automating unit testing for embedded systems software is to use Parasoft C++test.

Parasoft C++test is an integrated solution for automating a broad range of best practices proven to improve software development team productivity and software quality. C++test enables coding policy enforcement, static analysis, comprehensive code review, and unit and component testing to provide teams a practical way to ensure that their C and C++ code works as expected. C++test can be used both on the desktop under leading IDEs (including Wind River Workbench and Eclipse) as well as in batch processes via command line interface for regression testing. C++test integrates with Parasoft’s GRS reporting system, which provides interactive Web-based dashboards with drill-down capability, allowing teams to track project status and trends based on C++test results and other key process metrics.

For embedded and cross-platform development, C++test can be used in both host-based and target-based code analysis and test flows. On the host, developers can verify code by using C++test for coding policy enforcement, static analysis, comprehensive code review, and unit and component testing for “test as you go” verification, as well as regression testing. External dependencies for code under test are automatically replaced by stubs that can be used to mock up realistic runtime conditions without actually accessing the referenced hardware or software.

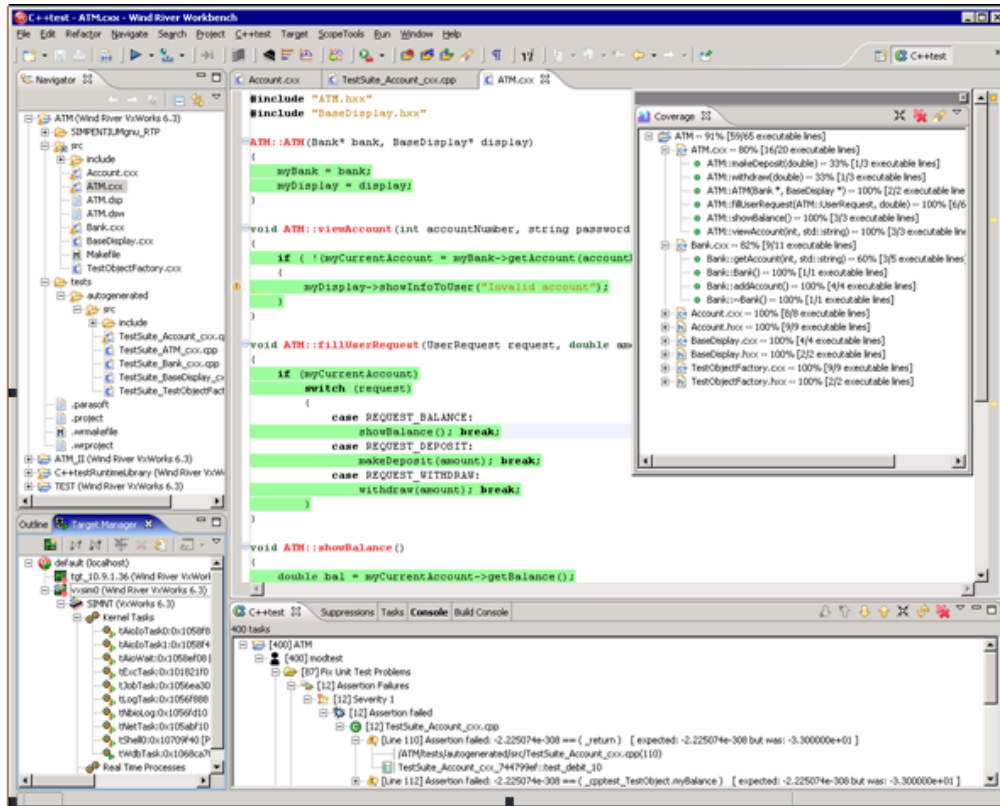


Figure 2: The C++test plugin for Wind River Workbench provides Workbench users with easy access to C++test’s full code analysis and unit testing capabilities.

By enabling extensive host-based testing, C++test allows users to start verifying code as soon as it is completed—even if the target hardware is not yet built or available for testing. As a result, the majority of the problems with the application logic can be exposed early—when error detection and remediation is easiest and fastest—and target testing can focus on verifying the interface between the hardware and the software. Moreover, host-based tests are much easier to automatically run and maintain, enabling developers to check the validity of their platform-independent code without tying up additional embedded development tools.

When developers are ready to test on a simulator or the actual target, the test suite that was generated and refined on the host can be reused to validate software functionality on the target. The stubs that were previously used can now be replaced by the real code or system interfaces for integrated system testing—without changing the test code. C++test also provides a built-in capability to automatically capture the test outcomes from execution and turn them into “golden” data sets for subsequent regression testing.

C++test automates the complete test execution flow, including test case generation, cross-compilation, deployment, execution, and loading results (including coverage metrics) back into the GUI. Testing can be driven interactively from the GUI or from the command line for automated test execution, as well as batch regression testing. In the interactive mode, users can run tests individually or in selected groups for easy debugging or validation. For batch execution, tests can be grouped based either on the user code they are linked with, or their name or location on disk.

At the same time, C++test allows full customization of its test execution sequence. In addition to using the built-in test automation, users can incorporate custom test scripts and shell commands to fit the tool into their specific build and test environment. Its runtime library can also be customized and cross-compiled for a wide variety of target operating systems. This unparalleled flexibility enables users to realize their desired test flow without being constrained by the preset tool capabilities.

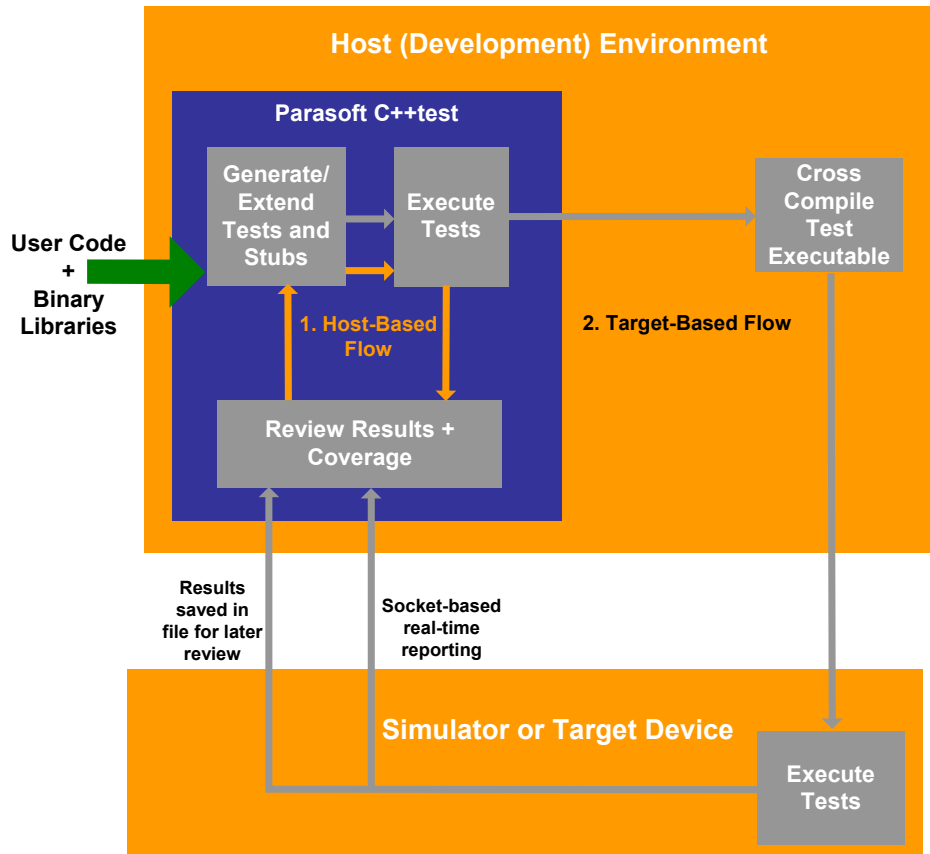


Figure 3: C++test's customizable workflow allows users to test code as it's developed, then use the same tests to validate functionality/reliability in target environments

Unit Testing Support for Embedded Development

- True unit (function/class) and component testing
- Automated generation of complete structured tests in C or C++ source
- Support for data sources
- Interactive execution of single tests or arbitrary groups of tests from the GUI
- Automated generation of regression tests via capture of actual test results after execution
- Uniform environment for test execution on host and target
- Completely customizable test flow and runtime support
- Coverage analysis for statement, block, branch/condition, and path metrics
- HTML and XML reports with test results
- GUI/desktop and command line modes



Supported Target Compilers

- Wind River GCC 3.4.x and DIAB 5.4+
- GCC 2.95.x - 4.1.x cross-compilers
- Green Hills 4.0.x

To learn more about how Parasoft solutions can help you automate unit testing for embedded systems software, contact Parasoft as described below.

Contacting Parasoft

USA

101 E. Huntington Drive, 2nd Floor
Monrovia, CA 91016
Toll Free: (888) 305-0041
Tel: (626) 305-0041
Fax: (626) 305-3036
Email: info@parasoft.com
URL: www.parasoft.com

Europe

France: Tel: +33 (1) 64 89 26 00
UK: Tel: +44 (0)1923 858005
Germany: Tel: +49 89 4613323-0
Email: info-europe@parasoft.com

Asia

Tel: +886 2 6636-8090
Email: info-psa@parasoft.com

Other Locations

See <http://www.parasoft.com/jsp/pr/contacts.jsp?itemId=268>

© 2007 Parasoft Corporation

All rights reserved. Parasoft and all Parasoft products and services listed within are trademarks or registered trademarks of Parasoft Corporation. All other products, services, and companies are trademarks, registered trademarks, or servicemarks of their respective holders in the US and/or other countries.