PARASOFT®

TECHNICAL WHITEPAPER

How to Perform Unit Testing With Code Coverage on Target

OVERVIEW

Testing on host and embedded target hardware, which Parasoft refers to as cross-platform testing, means running tests on the target environment with little to no change in the tool interface or usage. “Target,” in this case, usually refers to remotely connected hardware or a simulation that resembles the runtime environment of the product under development.

These target environments often run embedded operating systems on hardware that's more constrained by memory and performance than host environments where developers work and develop code. In addition, the compiler/linker/debugger toolchain is different as is the target processor.

THE IMPORTANCE OF TARGET-BASED TESTING

Target based testing is essential to validate correct functionality, security, and performance of embedded systems because the host environment lacks the embedded OS, hardware interfaces, and peripherals of the end product. In addition, the performance and behavior of target systems are different than the host environment and can affect the outcome of tests.

ESSENTIAL ELEMENTS OF TARGET-BASED TESTING OF EMBEDDED SYSTEMS

Software verification and validation is an inherent part of embedded software development. Testing is a key way to demonstrate correct software behavior. Unit testing is the verification of the module design. It ensures that each software unit does what it's required to do.

Additionally, safety and security requirements may require that software units don't behave in unexpected ways and aren't susceptible to manipulation with unexpected data inputs. Vulnerabilities may manifest in the target system due to the differences in behavior from the host environment. Attack surfaces for embedded devices are hard to simulate in host environments and proper testing requires target hardware.

Collecting and analyzing code coverage metrics is important for safety-critical software. Code coverage measures the completion of test cases and executed tests. It provides evidence that validation is complete, at least as specified by the software design.

Traceability between test cases, test results, source code, and requirements must be recorded and maintained. Therefore, data collection during target-based testing is critical.

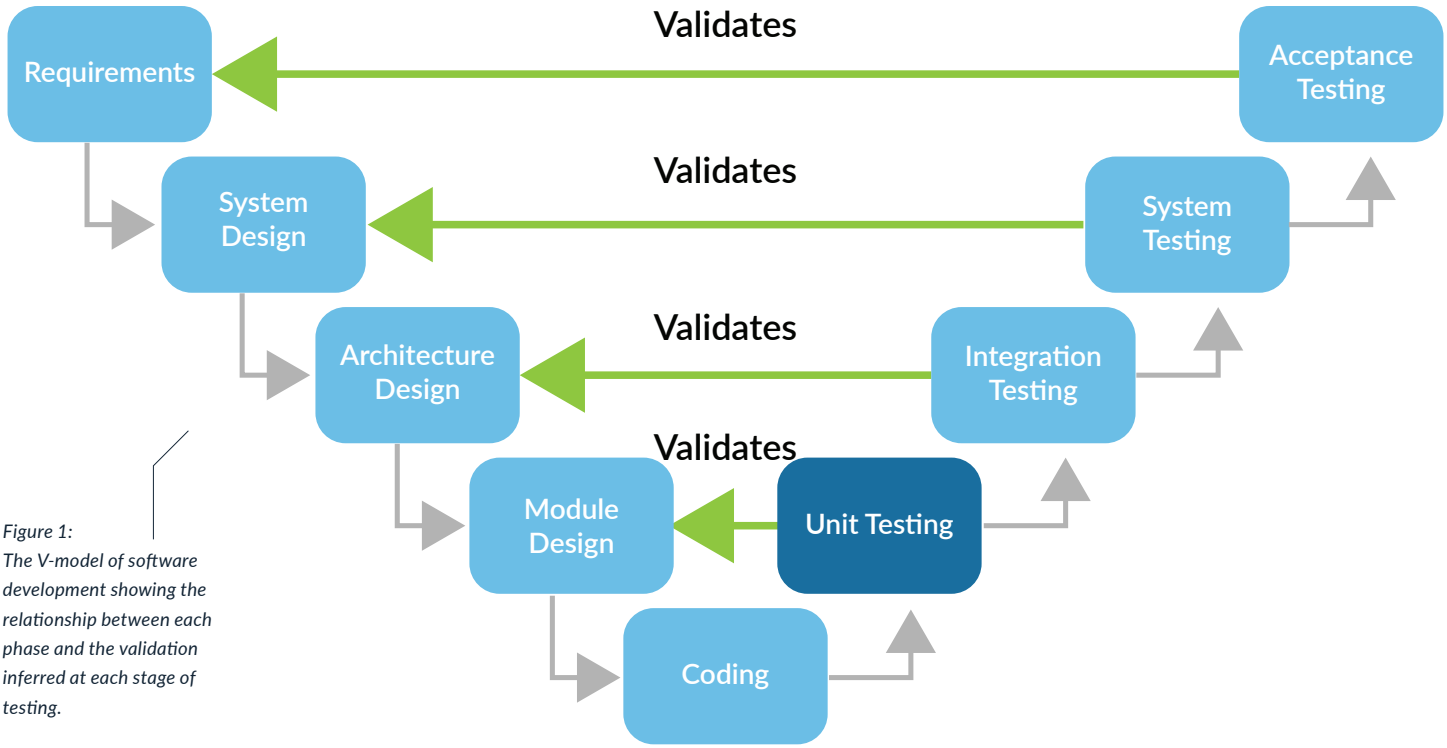

Figure 1:
The V-model of software development showing the relationship between each phase and the validation inferred at each stage of testing.

AUTOMATING TARGET-BASED TESTING OF EMBEDDED SYSTEMS

Automating the testing of embedded systems is challenging due to the complexity of initiating and observing tests on embedded targets. Limited access to target hardware is another challenge that software teams face. Nevertheless, automation is essential to make embedded testing workable on a continuous basis from the host development system to the target system.

Testing embedded software is particularly time consuming. Automating the regression test suite provides considerable time and cost savings. Collecting test results and code coverage data from the target system is essential for validation and standards compliance. Target hardware may have limited physical connectivity making it more difficult to retrieve test results. Fortunately, it's possible to extract the data from various ports, such as serial, ethernet (TCP/IP Sockets), JTAG connector, and other methods.

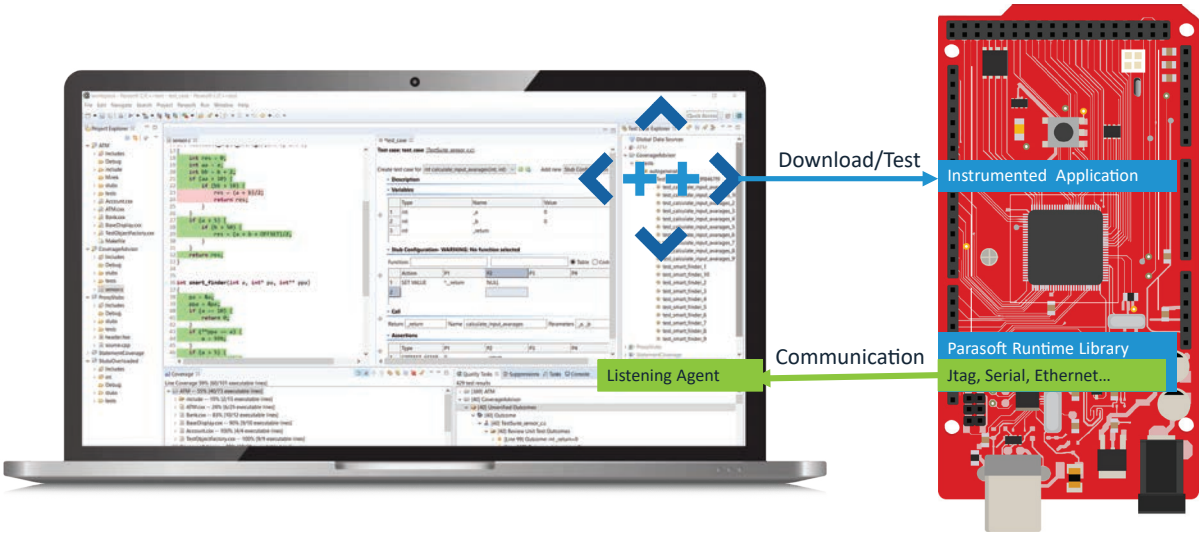

Figure 2:
Visual overview of host and target connectivity

CROSS PLATFORM TESTING FOR VERIFICATION & VALIDATION ON TARGET & HOST SYSTEMS

Cross-platform testing allows tests to be generated and extended on the host—the development environment where the tools are installed—and then executed onto one or more targets. This is especially useful for testing code that you cross compile for use on an embedded device or on another platform.

As an example, Parasoft C/C++test removes the barrier to effective embedded testing by automatically generating test cases that can be executed in any cross-platform situation—host, simulator, and in the actual target environments. It's also possible to collect test results and code coverage metrics. Instrumentation detects runtime defects like memory leaks in the running applications.

On the host environment, developers can automatically generate a core set of unit and API test cases designed to identify unexpected function responses to corner case conditions. With a different configuration, the generated tests will capture current software behavior at the method/function level. Next steps for this test suite include:

1. Extending as needed for functional testing.
2. Automatic configuration for regression testing.
3. Executing on the host, if desired, automatically replacing the target dependencies with configurable stubs.

The same test suite then gets cross-compiled, generating executables for a different processor and hardware environment for execution in a target environment. C/C++test saves and uses the target test results later in the GUI for evaluation and analysis. TCP/IP sockets can automatically send test results to the C/C++test GUI, which collects coverage metrics, including branch, simple condition, and MC/DC coverage for all tests. The C/C++test GUI provides extensive facilities for debugging test cases, including support for many host debuggers, stack trace reporting, reporting of call sequences, and detailed display of test case results.

C/C++test can instrument the original application to detect memory related problems, and then cross-compile and start on the target to pinpoint existing memory bugs to collect code coverage. Teams can combine this coverage data with that from other unit tests on host or target.

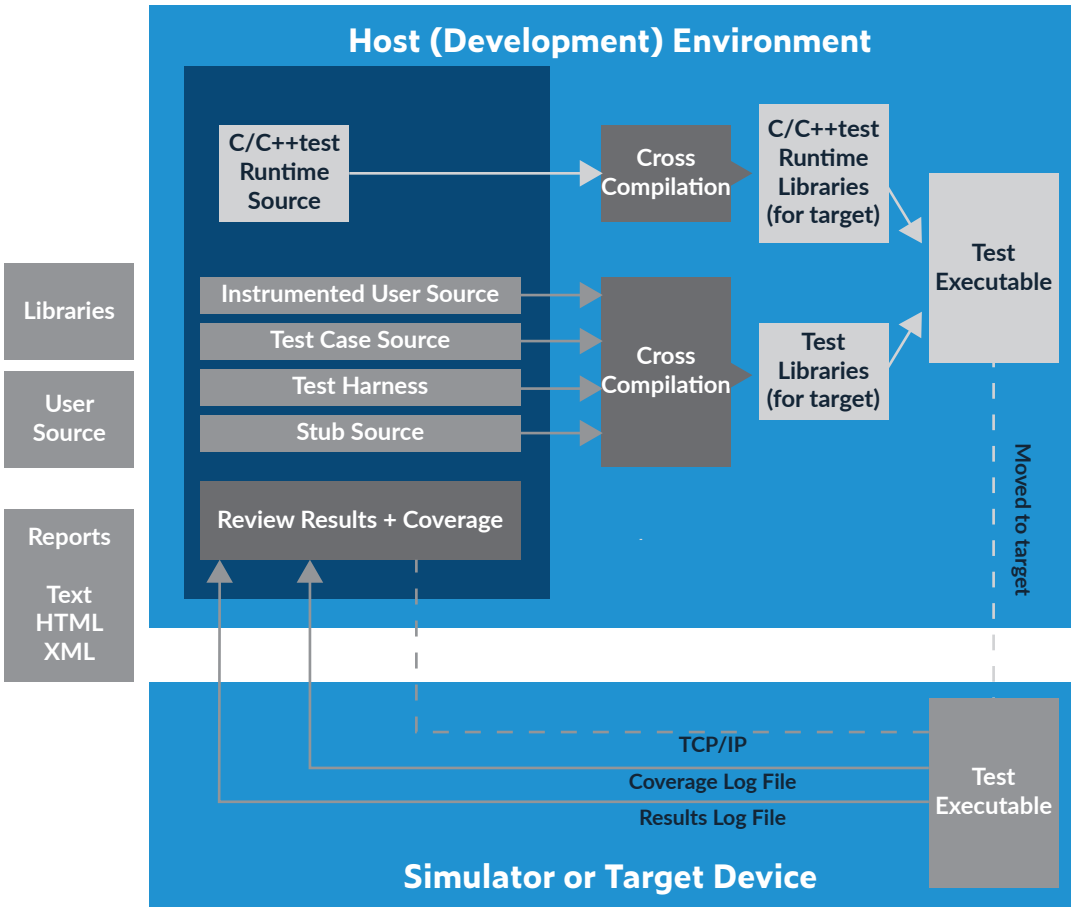

Figure 3:
The development (host) environment has all the tools to create a target executable. The target executable communicates results back to the host.

BUILDING A TEST EXECUTABLE

The test executable consists of a test harness built around instrumented source code and the C/C++ runtime library. In the case of Parasoft C/C++test, it uses a prebuilt version of the C/C++test runtime library, which ships with the C/C++test distribution.

For target-based testing, teams need a cross-compiled C/C++test runtime library. C/C++test automatically prepares a build of the runtime library. In rare cases that require non-standard customization, it's possible to manually prepare a build of the runtime library and use a cross-compiler to build the test harness source code.

C/C++test automates the process of cross compiling the test harness and linking it with the C/C++test runtime library, which requires that C/C++test correctly define the cross-compiler. This does not require user interaction.

The following graphic illustrates the process of building a test executable.

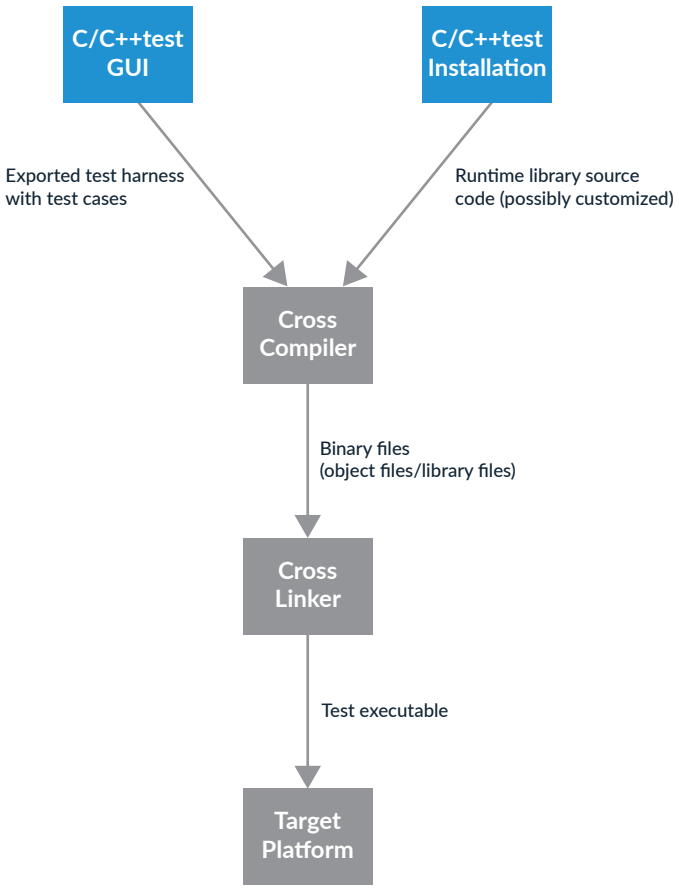

Figure 4:
The process flow for building target-based test executables.

RUNNING THE TEST EXECUTABLE & COLLECTING RESULTS

After deploying the test executable to the target environment, it's time to collect test results. To start the test executable, the target development environment provides the facility to automate the process of deploying and running the test executable. In this case, these processes can become part of a test flow definition managed by C/C++test.

If C/C++test builds the runtime library with a socket communication channel or a serial RS232 communication channel, then it sends results to a listening agent provided with the C/C++test distribution. Alternatively, C/C++test can build the runtime library with support for file communication and send test results to a results file.

COLLECTING RESULTS THROUGH THE FILE COMMUNICATION CHANNEL

If using Parasoft C/C++test following test execution, then it creates two files for test results and coverage results on the target environment's local file system. On some environments, teams can store files on the hard drive. Others will have file I/O implemented based on flash memory and the like. The C/C++test runtime library uses ANSI standard I/O functions for managing the file communication channel.

During test execution, the code automatically logs asserts into the results file as presented in the following diagram.

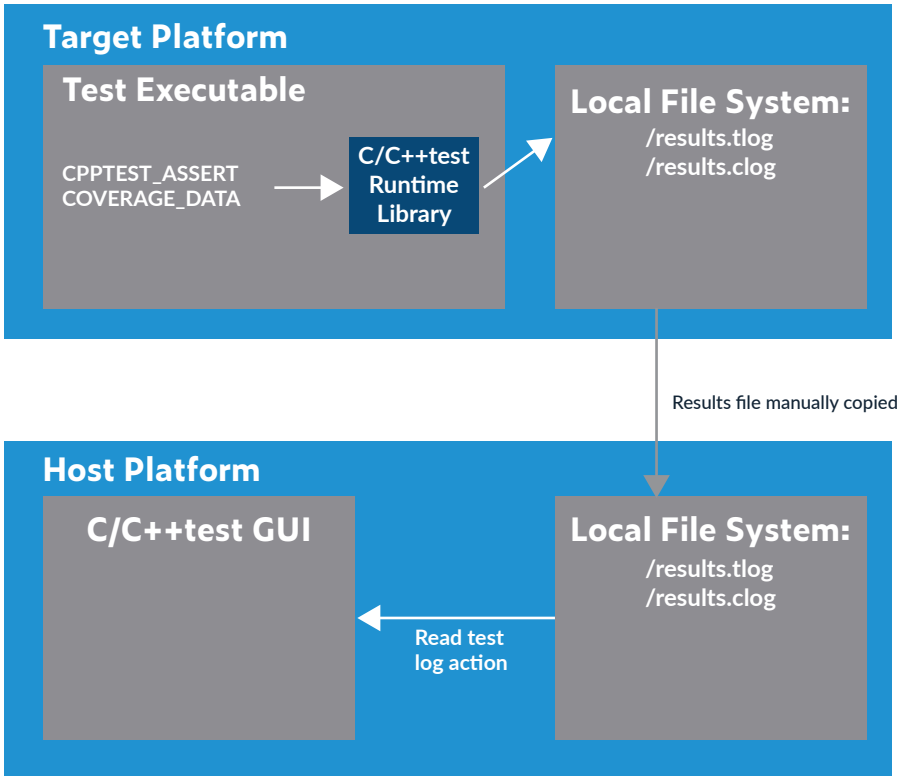

Figure 5:
A local file on the target system communicates test results, which teams can access manually from the host.

COLLECTING RESULTS THROUGH SOCKET COMMUNICATION

If using the socket communication channel between the target and Parasoft C/C++test, two TCP/IP sockets open at the start of testing. One socket is for sending test results. The other is for sending coverage results. If coverage is not enabled, only one socket opens.

Collecting results thought socket communication requires a listening agent on the host side that can listen on the given port and write data to a file on the host machine. C/C++test provides a basic implementation of a listening agent. However, you can use any utility program capable of listening on a port and dumping data to a file.

After the test execution is complete, the listening agents will flush the results files and the results are ready in the C/C++test GUI, just as for file-based communication. The following diagram illustrates the socket-based communication process.

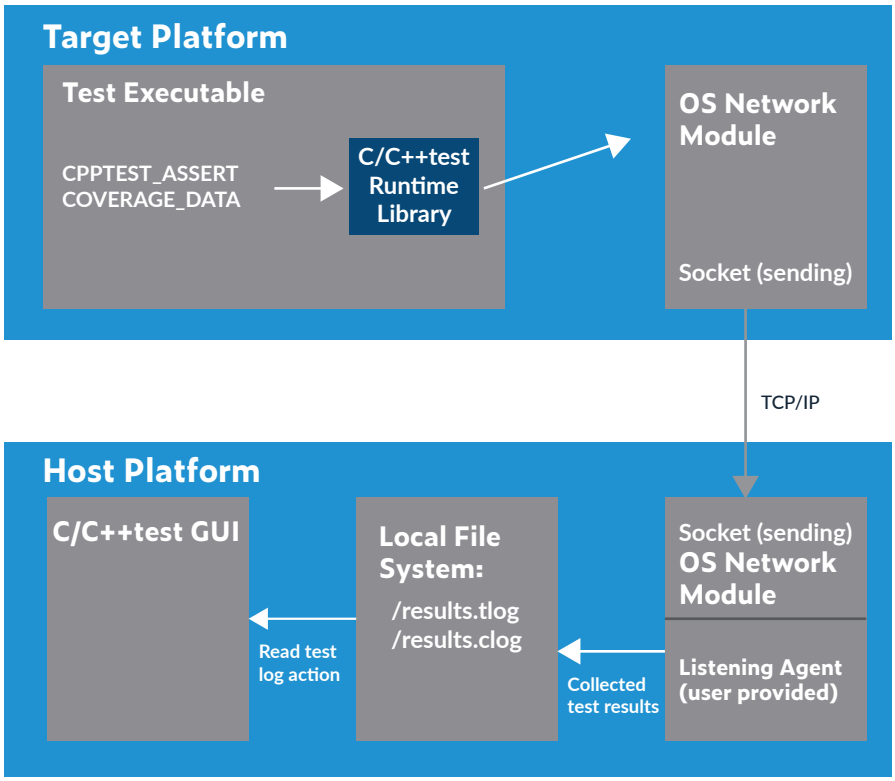

Figure 6:
A TCP/IP socket communicates test results to the host system.

COLLECTING RESULTS THROUGH SERIAL COMMUNICATION

When the test executable is built with support for a serial communication channel, a serial connection will be initialized upon the start of the test executable, and testing data will be sent from the target to the listening agent, which is started on the host machine. The listening agent will decouple test data from coverage data and save them in separate files. Teams can then read the results in the C/C++test GUI, just like for file-based communication.

The following diagram illustrates the serial communication process.

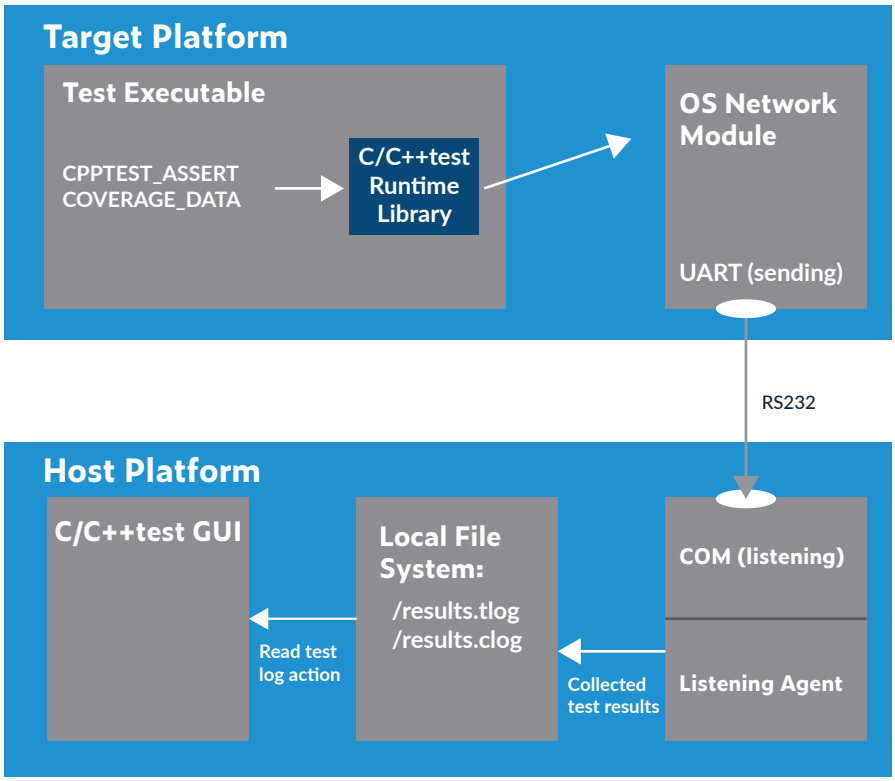

Figure 7: The RS232 serial link communicates test results to the host.

AUTOMATING THE TEST EXECUTION FLOW FOR TESTING ON THE TARGET

A custom execution flow, such as for embedded/cross-platform development, enables test automation for development environments, which cannot use the default C/C++test flow. Testing here involves the following tasks.

1. Prepare a test harness.
2. Build that harness with a cross-compiler.
3. Deploy and start it on the target device.
4. Downloading results back to the host machine.
5. Prompt C/C++test to read the results.

Teams can use custom execution flows to execute any external utility, such as make, FTP, a target communication manager, and more.

With Parasoft C/C++test, customizing the test execution flow automates testing on the target which is traditionally a manual and tedious process. Here's the expected workflow.

1. Prepare the test harness—instrument user source code, generate and collect test cases.
2. Build the test harness.
3. Deploy the test executable to the target device.
4. Execute the tests automatically or wait for the users to carry out the tests.
5. Download test and coverage results to the host machine and have C/C++test read them.

THE PARASOFT C/C++TEST RUNTIME LIBRARY

Runtime libraries are a critical component of target-based testing since tests require an appropriate runtime environment for execution. Teams must build the runtime library with a cross compiler and appropriate libraries for execution, like the application and test cases.

Parasoft C/C++test ships with a prebuilt, full-featured runtime shared library. However, the design of this runtime is for rich host platforms. Considering the multitude of embedded platforms on which testing may occur, as well as their capabilities and limitations, teams need to adjust the runtime before building it for different embedded environments. That’s why the runtime library's pure-C sources are also available.

Teams can configure the library to support features that are available on the target platform or to block those that are not. Embedded developers are familiar with the pros and cons of the platform(s) they use and should be able to build an appropriately configured library and append its path to the linker command line.

WORKING WITH THE C/C++TEST RUNTIME LIBRARY

Parasoft C/C++test builds the runtime library automatically as a part of the test-executable preparation phase the teams system requires a custom build of the C/C++test runtime library.

The C/C++test runtime library distribution is in the form of C source and header files. In most cases, teams can let C/C++test automatically build the runtime library. If desired, teams can build the runtime library using the make utility and one of the pre-configured make configurations for the supported platforms. C/C++test provides special project files that facilitate runtime library building for popular IDEs.

The following steps describe that general approach.

1. Build the runtime library using the provided make files or one of the preconfigured IDE projects.
2. Locate the library files with a fixed location so that multiple users can point to it without changes. C/C++test also supports using a regular environment variable for the runtime library.
3. Add the resulting ".a" file with an appropriate path to project linker options.

The runtime library sources include configuration macros for all officially supported embedded platforms, as well as lower-level function-based configurations. These macros need to be correctly set for a build configuration in the IDE. Thus, if the IDE requires more than one target architecture, then a build of the C/C++test runtime library is needed for each.

Building the C/C++test Runtime Library

Parasoft C/C++test can build the testing runtime automatically as a part of the test-executable preparation phase. This is the recommended approach. In most cases, manual preparation of the C/C++test runtime library is not required.

Basically, building the runtime library involves selecting the communication channel, then running the IDE builder or "make" in the directory containing the Makefile. Teams may also need to change the compiler and platform-specific configuration if the default is unusable or if there is no default configuration for the compiler and target platform.

CONFIGURING THE C/C++TEST RUNTIME LIBRARY

Designed to suit different environments, the Parasoft C/C++test runtime library is easy to configure. Included macro definitions can assist with the C/C++test runtime library configuration. In rare cases, if the target environment has some non-standard limitations, it may be necessary to make additional changes to the runtime source code.

In most cases, if the C/C++test runtime library requires changes to the value of a configuration macros, there's no need to modify or create a new configuration file. Rather, teams can modify the CFLAGS variable in the target configuration file. Alternatively, in instances of using an IDE, teams can modify compiler options with the proper "-D" options to set the macros to the required values.

CONFIGURING TESTING WITH THE CROSS COMPILER

Teams can add a custom compiler definition to Parasoft C/C++test to use for all projects that normally compile and build for the target system. There is documentation available to guide you through custom compiler definition, as well as importing custom compiler definitions and setting build options when defining a custom compiler. Teams can also modify compiler/linker names and the specific patterns for how C/C++test uses the compiler and linker.

CUSTOMIZING THE TEST EXECUTION FLOW

Running a "test configuration," as Parasoft C/C++test calls it, prompts a series of actions that lead to the unit testing results being loaded into the C/C++test UI. A test flow definition specifies these actions stored in XML format and saved as part of the test configuration. All built-in test configurations that perform test execution have corresponding preconfigured test flow definitions.

In addition to providing preconfigured execution flows designed specifically for host-based testing, C/C++test also allows the customization of test flows to provide a straightforward way of working with non-standard environments.

In most cases, there's no need to create a custom execution flow because built-in test configurations allow easy editing of the most critical and commonly modified flow properties. An example is shown below. Teams can adjust the specifics for the environment using the existing examples.

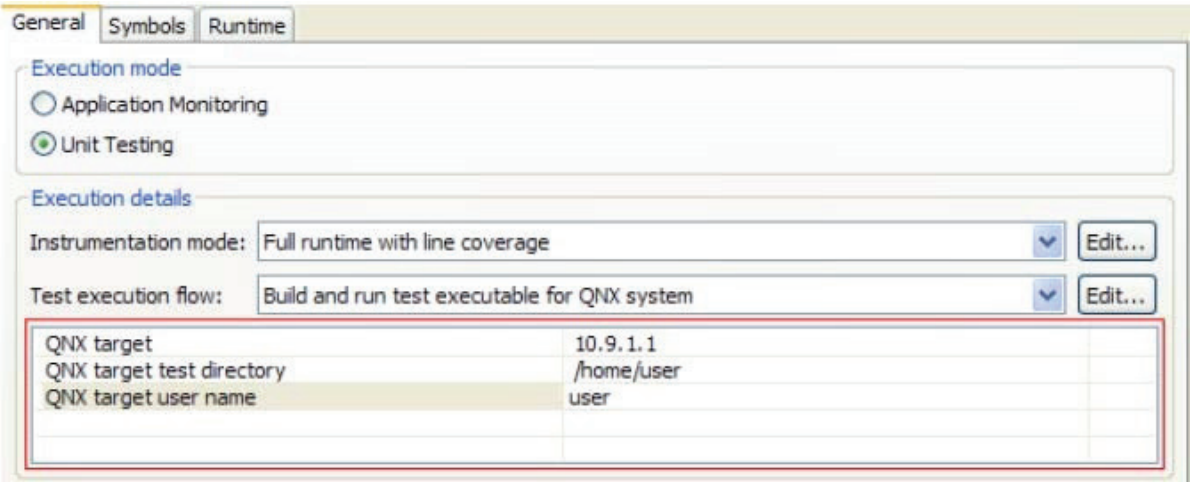

Figure 8: An example of test configuration in Parasoft C/C++ test.

WHY DO WE NEED CROSS PLATFORM TESTING?

First, target-based testing might be an absolute requirement for the product your team is developing. Most safety-critical development standards and guidelines will require performing testing in a target system due to the differences in performance, constraints, toolchain, and even process execution with host-based systems.

If target-based testing isn't an absolute requirement, most embedded software projects will end up testing on their real hardware eventually. Here are some of the advantages.

» **Shift left target testing.** Testing on target is inevitable for most embedded software projects. Leaving it until later stages of development is risky. Making target-based testing part of the CI/CD pipeline with automation helps reduce late-stage integration bugs and rework.

» **Integrate complex target-based testing into CI/CD pipeline.** Modern tools like Parasoft C/C++test make it possible to automate and integrate complex target testing environments. Building regression target test suites that run automatically helps achieve earlier and more regular testing while measuring and increasing code coverage.

- **Increase code coverage of target-based testing.** Using the tools within Parasoft DTP and C/C++test, it's possible to increase code coverage through automatic test generation. Teams can use these tests to increase code coverage on host and target.
- **Flexible options for data retrieval—sockets, file, or serial.** Regardless of the device type and how you build it, there must be a way to retrieve test data.
- **Same frameworks for host, simulator or target.** Tests run in any environment have the same features. Although the hardware environment differs, the testing environment does not. This means host and simulation testing is still beneficial even in parallel with target testing.
- **Efficient target usage allows for team sharing.** Test automation means that target testing is fast and efficient. Multiple developers can queue up their test suites to make use of available target hardware.

SUMMARY

Software verification and validation is an essential part of embedded software development and testing. Unit testing is necessary to ensure that each software unit does what it's required to do. Automating testing embedded systems is more challenging due to the complexity of initiating and observing tests on embedded targets.

Embedded software uses cross-platform testing to test on both host and target systems. Target environments are more constrained by memory and performance than host environments, and the compiler/linker/debugger toolchain is likely to be different from the target processor.

Parasoft C/C++test supports cross-platform testing and enables test automation for target-based testing. All this helps shift left target testing and increase code coverage while integrating complex target testing into CI/CD pipelines.

TAKE THE NEXT STEP

Learn the benefits and strategies of continuous testing and CI/CD workflows for embedded software development. Download the whitepaper.

ABOUT PARASOFT

Parasoft helps organizations continuously deliver quality software with its market-proven, integrated suite of automated software testing tools. Supporting the embedded, enterprise, and IoT markets, Parasoft's technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software by integrating everything from deep code analysis and unit testing to web UI and API testing, plus service virtualization and complete code coverage, into the delivery pipeline. Bringing all this together, Parasoft's award-winning reporting and analytics dashboard delivers a centralized view of quality enabling organizations to deliver with confidence and succeed in today's most strategic ecosystems and development initiatives—security, safety-critical, Agile, DevOps, and continuous testing.