# Cross-Platform Testing Overview

This topic provides an overview of C++test's cross-platform testing process and requirements. For details on using C++test with the supported environments, see Supported Environments Details.

Sections include:



## About C++test's Cross-Platform Testing

Cross-platform testing allows C++test tests to be generated and extended on the host (the development environment where C++test is installed), then executed on one or more targets (environments where the software will execute, which might be a target embedded device, a simulator or emulator, or another platform to which you have ported your software). This is especially useful for testing code that you cross compile for use on an embedded device or on another platform.

The general procedure for performing target-based testing is as follows:

1. Open (or import) the project you want to test with C++test.
2. Introduce any additional compiler/linker flags to the C++test build options (as required for your particular environment).
3. Build the test executable.
    a. Generate the test cases with a built-in or custom test generation Test Configuration (for example, Generate Unit Tests).
    b. Create a custom Test Configuration to build the test executable. For instance, you can disable the use of stubs and customize the test execution flow to specify where you want the results log files written.
    c. Build the test executable by running the above Test Configuration.
4. Move the test executable to the target.
5. Run the test executable on the target.
6. Ensure that the host can access results, then use the C++test GUI for results analysis by running one of the "Utilities> Collect Results" Test Configurations.

These steps are described in detail in the topics included in this section.

## Supported Environments

- Host development environments:
    - OS: Windows or Linux
    - Compilers: GNU GCC 2.95.x – 3.4.x (Windows), GNU GCC 2.95.x – 4.1.x (Linux), Green Hills MULTI v4.0.x Native (Windows), Green Hills MULTI v5.1.x Embedded V800 (Windows), Microsoft Visual Studio (Windows), Sun CC compilers
- Embedded environments (non-exclusive):
    - ARM  DS-5 5.18 and higher
    - Embedded Linux
    - Green Hills MULTI
    - IAR Embedded Workbench for ARM (starting from 5.3x)
    - Renesas High-performance Embedded Workshop 4
    - Texas Instruments Code Composer 5.x, 6.0, 7.4, 8.0
    - Wind River Workbench 3.3

See Supported Environments for compiler support details.

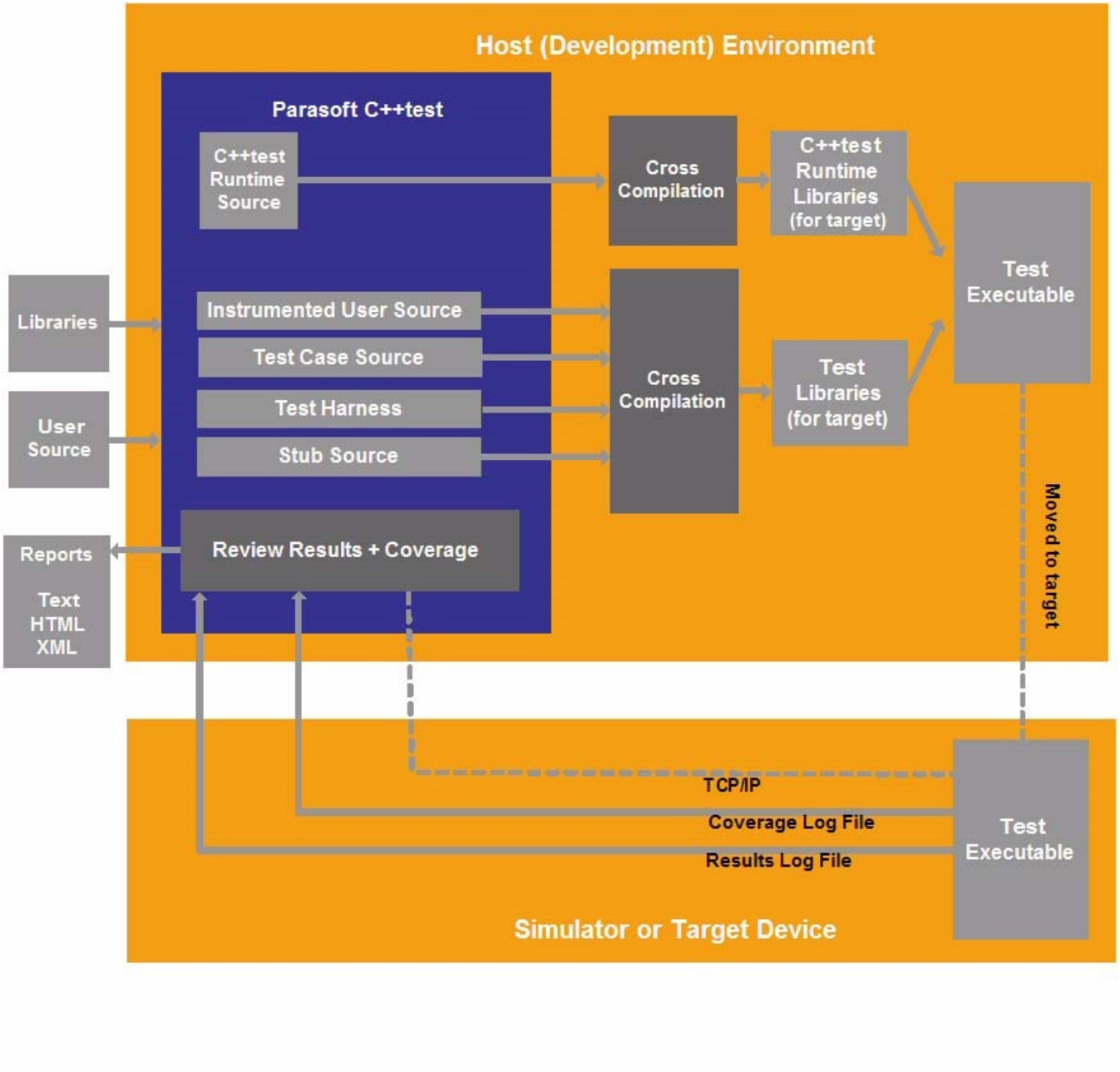
## Understanding the General Strategy

C++test removes the barrier to effective, comprehensive embedded testing by automatically generating extensible test cases that can be executed on the host, simulator, and in the actual target environments to verify code robustness, verify functional results, and obtain code coverage metrics. It can also monitor running application to detect memory problems and to track code coverage.

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. This test suite can then be extended as needed for functional testing, automatically configured for regression testing, and executed on the host if desired (with the target dependencies automatically replaced by configurable stubs).

The same test suite can then be cross-compiled for execution in a target environment. Target test results are saved to a file that can be loaded into the C++test GUI for evaluation/analysis. Test results can be automatically sent to the C++test GUI via TCP/IP sockets. Coverage metrics, including branch, simple condition, and MC/DC coverage, are collected for all tests. The 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.

Your original application can be also instrumented for detection of memory related problems, cross-compiled and started on the target to pinpoint existing memory bugs and to collect code coverage. This coverage data can be combined with the coverage data from unit testing.




## Building a Test Executable

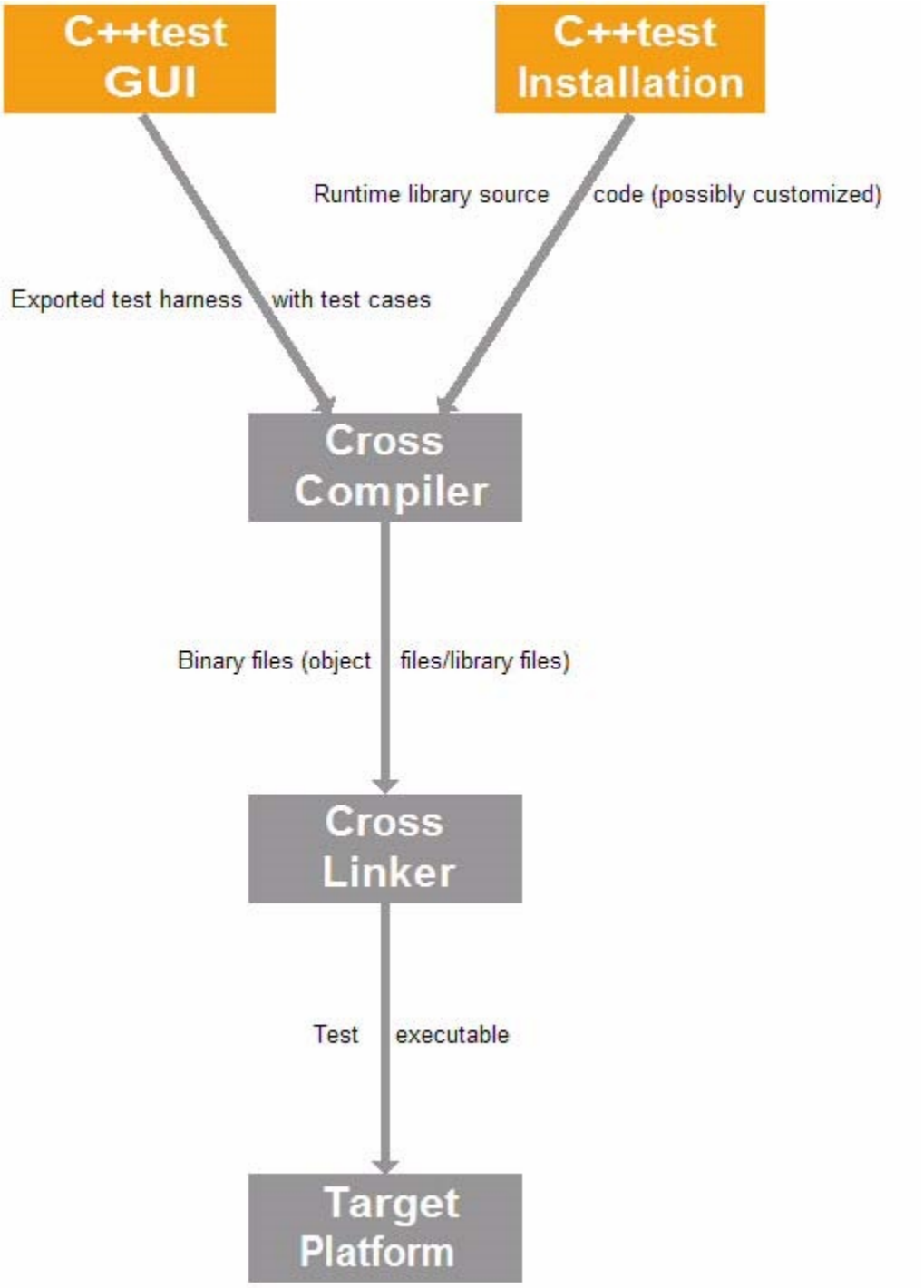
The test executable consists of a test harness built around instrumented source code and the C++test runtime library.

For host-based testing, C++test uses a prebuilt version of the C++test runtime library, which is shipped with the C++test distribution.

For target-based testing, a cross-compiled C++test runtime library is be needed. C++test will automatically prepare a build of the runtime library. In some cases when non-standard customization is required, you will need to manually prepare a build of the runtime library (as described Working with the C++test Runtime Library) and use a cross-compiler to build the test harness source code. The cross-compiled test harness and runtime library should then be linked together with libraries containing standard OS functions (a list of external functions used in the C++test runtime is provided in Symbols Used By the C++test Runtime Library).

The process of cross-compiling the test harness and linking it with the C++test runtime library is usually automated by C++test (See Working with the C++test Runtime Library) and requires the cross-compiler to be correctly defined in C++test (See Configuring Testing with the Cross Compiler). This process does not require user interaction.

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




## Running the Test Executable and Collecting Results

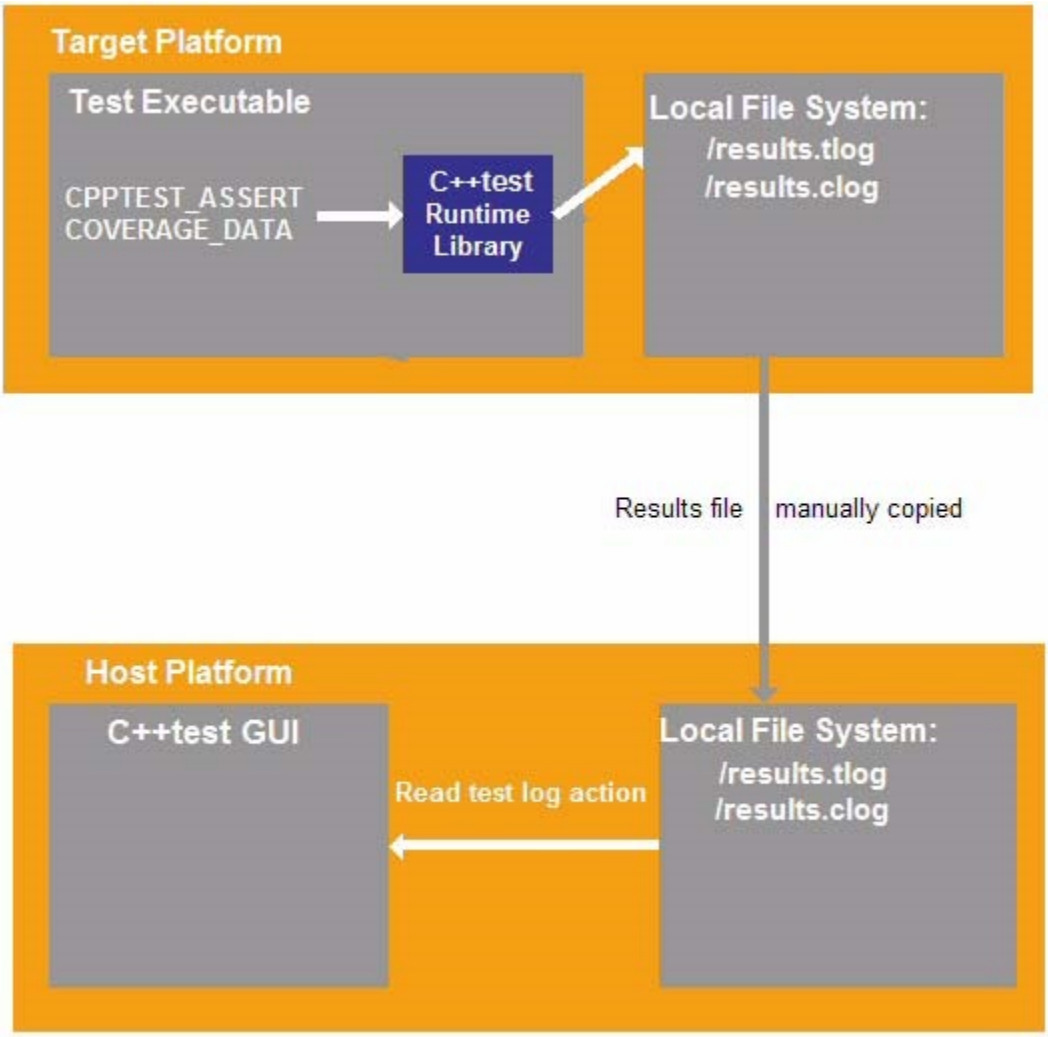
After deploying the test executable to the target environment, you start it and collect test results. To start the test executable, you call the "main" function or let your system call it. Ideally, your environment will provide the facilities to automate the process of deploying and running the test executable. In this case, these processes can become part of a test flow definition managed by C++test.

If the C++test runtime library is built with a socket communication channel or RS232 communication channel, the results will be sent to a listening agent provided with the C++test distribution.

If the C++test runtime library is built with support for file communication, the test results will be sent to a results file.

## Collecting Results Through the File Communication Channel

After the test executable is started, initialization routines will create two files for test results and coverage results. When coverage is disabled, only one test results file will be created. These files are created on the target environment's local file system. On some environments, files can be stored on the hard drive; others will have file I/O implemented based on flash memory, etc. The C++test runtime library uses ANSI standard I/O functions for managing the file communication channel. During the test execution, asserts are logged to the file as presented in the following graphic:




After testing is completed, the test log needs to be transferred back to the host so you can review results in the C++test GUI. This transfer can be automated by C++test (as described in Customizing the Test Execution Flow) or performed manually.

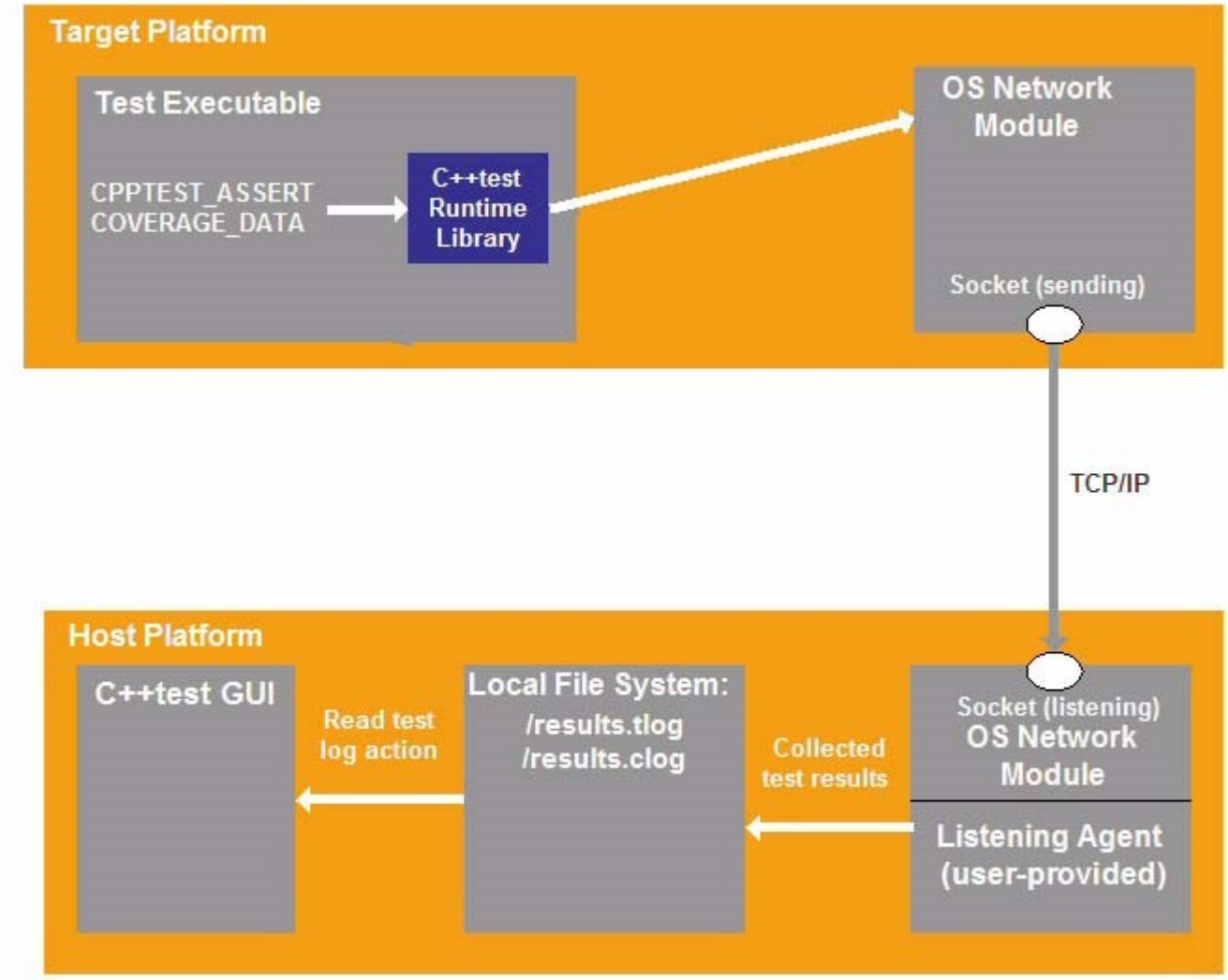
## Collecting Results through Socket Communication

When the test executable is built with support for a socket communication channel, two TCP/IP sockets are opened 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 will be opened.

To collect results thought socket communication, you need a listening agent that can listen on the given port and write data to a file on the host machine. A basic implementation of a listening agent is provided with C++test. However, you can use any utility program capable of listening on a port and dumping data to a file.

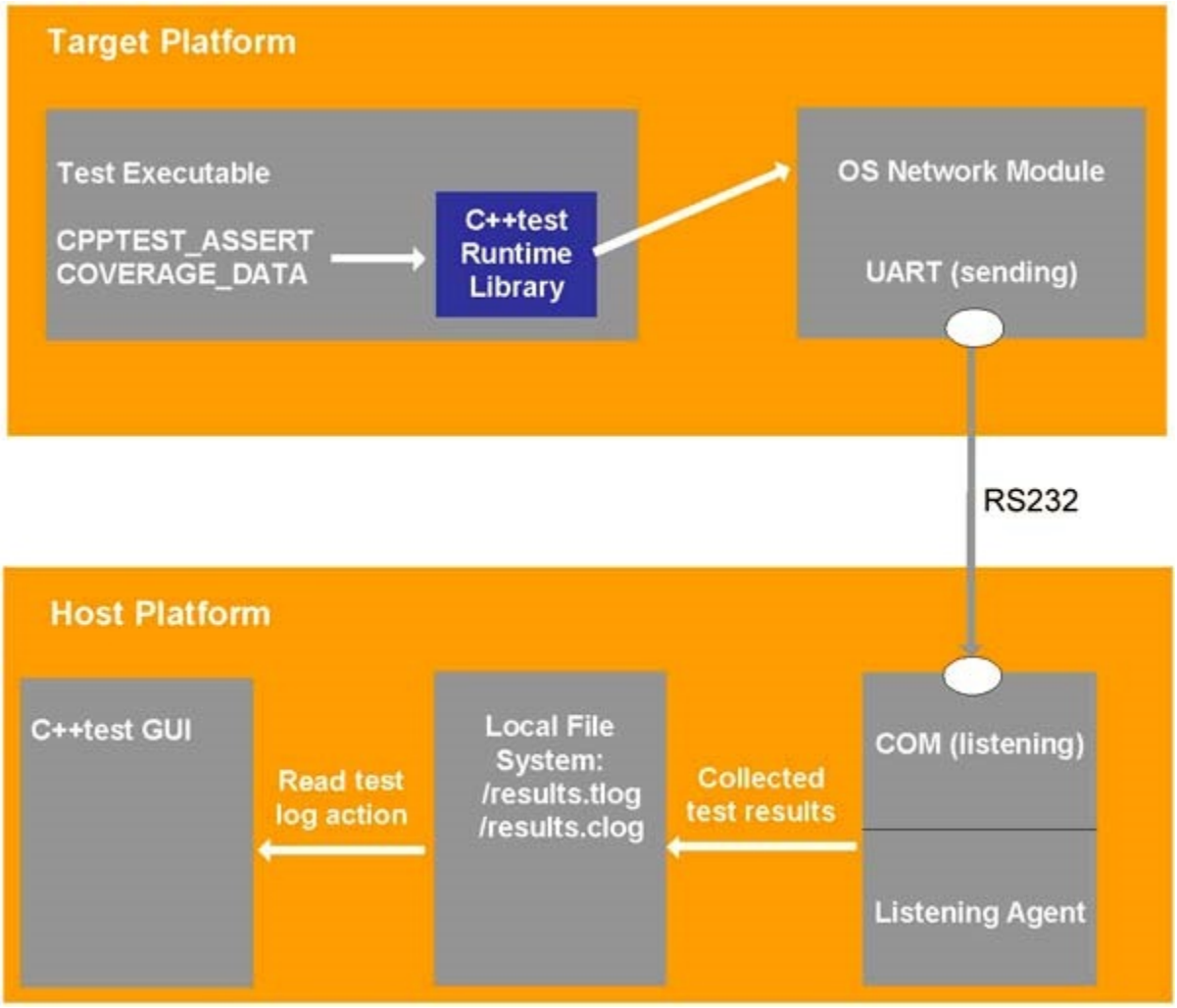
After the test execution completes, the listening agents will flush the results files. You can then read the results in the C++test GUI, just as you would for file-based communication.

The socket-based communication process is illustrated in the following graphic:




## Collecting Results Through Serial Communication

When the test executable is built with support for a serial communication channel, 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. You can then read the results in the C++test GUI, just as you would for file-based communication.  The serial communication process is illustrated in the following graphic:




# Automating the Test Execution Flow for Testing on the Target

By defining a custom execution flow, you can enable test automation for development environments where the default C++test test flow cannot be used—such as embedded/cross-platform development, where testing involves preparing a test harness, building that harness with a cross-compiler, deploying and starting it on the target device, downloading results back to the host machine, and prompting C++test to read the results. Custom execution flows can be used to execute any external utility which can be started in the operating system (for instance, make, ftp utility, target communication manager and so on).

Customizing the Test Execution Flow describes how you can use a custom execution flow to automate testing on the target. The expected workflow that we describe how to automate is:

1. Prepare the test harness (instrument user source code, generate/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 tests to execute.
5. Download test and coverage results to the host machine and have C++test read them.