Unit Testing for Wind River Tornado

This topic explains how to configure and run unit testing on code that is designed to be compiled/built using Wind River Tornado compilers and/or written with the aid of Wind River Tornado IDE.

In this section:

About Unit Testing

The term "unit testing" encompasses everything related to building and running test objects. Before you perform unit testing, it is helpful to familiarize yourself with the following C++test terms/features:

- **Generation**: Generating test cases grouped inside test suites.
- **Instrumentation**: Modifications to the source code under test that allow C++test to collect information during the test object runtime. You may customize instrumentation features.
- **Coverage**: Measures how well the tests cover the source code under test.
- **Stubbing**: Substitution of unwanted/unsafe routines (and generation of missing ones) to facilitate successful building and safe running of test objects.
- **Runtimelibrary**: See Understanding and Building the Runtime Library.
- **Building the test object**: This includes everything required to successfully build the test object—the final artifact that is produced from your source files and the test suites, and which must be executed to collect runtime data (such as test case results and coverage data).
- **Execution**: The process of running a test object. Runtime data is collected during this phase and presented after it terminates.

Because VxWorks test objects/relocatables are linked with the -r option, the existence of symbol (function/variable) definitions is not checked, so any symbol resolution errors that occur will not be reported during linking. As a result, you won't know if you forgot to define these definitions, or made a scope mistake that prevented their inclusion—until the execution phase, when the test object crashes. To receive advance warning about such problems, ensure that the Test Configuration you use for unit testing has the **Perform early check for potential linker problems** option enabled (in the **Execution> Symbols** tab) and that it is configured to use the generated external VxWorks image symbols list(as described in Providing an External List of Library Symbols).

You must have all project settings set properly in order to perform unit testing successfully (for details on settings, see Setting Target/Platform Dependent Options).

The following sections in this topic explain how to build the C++test Runtime library (in case you want do this manually for additional control/customization), describe how to generate an external symbols list from your VxWorks image, remind you what Tornado tools must be up and running before you can actually execute your test object, familiarize you with Test Configurations designed specially for Tornado testing, and explain how customization of the test flow allows you to adjust the test object as needed to suit the circumstances of embedded testing.

For more information on unit testing, see Test Creation and Execution.

Understanding and Building the Runtime Library

To learn about the runtime library, see Working with the C++test Runtime Library.

Example: Building the Runtime Library for Tornado with VxWorks

To build the runtime library with Tornado tools and for the VxWorks OS (assuming that you have the required Tornado environment variables set correctly - see Prerequisites):

1. Go to the Runtime Library sources root directory (C++test_install_dir/engine/runtime).
2. Enter the command
   `make TARGET_CFG:=`
   followed by either `WR_egcs_simnt_VxWorks5_4.mk` or `WR_gcc2_9_simnt_VxWorks5_5.mk`
   (depending on your Tornado/VxWorks version).

You can also specify the following variables on the command line:

- The OUT_DIR variable.
- The CPPTEST_INC_DIR variable in case of building from alternate sources.

Example: Using the Provided Tornado Project/Workspace

To build the runtime library from the Tornado project or workspace provided with C++test:

1. Go to `C++test_install_dir/engine/runtime/projects`.
2. Open the appropriate project or workspace. Choose from the ones in `Tornado2_0` or `Tornado2_2`, depending on your Tornado/VxWorks version.
3. Build the opened C++testRtLib project.

You can copy and edit any .mk target configuration available in `C++test_install_dir/engine/runtime/target` and provide it for the TARGET_CFG argument. You can also add other Build Targets to Tornado projects to adjust building options and/or change targets.

Generating an External VxWorks Image Symbols List

An external symbols list is required to build the VxWorks test object using Test Configuration that have the **Perform early check for potential linker problems** option enabled—for example, all built-in Tornado-related Test Configurations. The list should contain all symbols defined in the VxWorks kernel image that you use for running your applications.

You can generate this list as follows:

1. Ensure that you have the required Tornado environment variables set correctly - see Prerequisites.
2. Duplicate the built-in "Embedded Systems> Wind River> Extract Symbols from VxWorks Image" Test Configuration.
3. Open the **Execution** tab.
4. Adjust the **Symbols listing tool name** Test Flow property to match the name of the symbols dumping tool suited for the compiler used to create the image—e.g., nmsimpc (for ccsimpc)
5. Adjust **Symbols listing options** if needed
6. Set the **Path to a VxWorks image to scan for symbols** Test Flow property to the path to your VxWorks image—e.g., `"%WIND_BASE%\target\config\simpc\vxWorks"`
7. Run the adjusted configuration

For more information on external symbol lists, see Providing an External List of Library Symbols.

Executing Test Objects

When we refer to the term "test object" in this section of the user's guide, we mean the final artifact that is produced from your source files and the test suites, and which must be executed to collect runtime data (such as test case results and coverage data). However in other parts of the C++test User's Guide, the term "test executable" is used instead. We make this distinction when discussing embedded development because embedded systems don't always support "executables"—the type of binary files that can be called application images, and can be loaded (unpacked) and executed by the system itself. For example, in the case of VxWorks 5.4 and VxWorks 5.5, you can build a full system image that links in your application procedure—or you can build an incomplete relocatable object that contains symbols from only your application and is linked against a prebuilt system's image at a later time.

In the case of Tornado testing for VxWorks, C++test will try to build the second type of objects, which can be best described as "test relocatables," and run them using an original Tornado tool called Tornado Shell (windsh). You will need to have the Tornado Registry (wtxregd) and Target Server (tgtsvr) up and connected with a target running VxWorks prior to launching the runtime testing (see Prerequisites).

For example, to prepare the testing environment to test on a VxWorks simulator (assuming you have the required Tornado environment variables set correctly - see Prerequisites) you can use the original Tornado interface or the following example steps/commands:

1. Set the WIND_UID environment variable to your UID. For more information on WIND_UID, see the Tornado User's Guide.
2. Launch the Tornado Registry.
   ```
   wtxregd & '
   ```
3. Launch VxSim.
   ```
   "%WIND_BASE%\target\config\simpc\vxWorks" &
   ```
4. Launch Target Server.
   ```
   tgtsvr vxsim -B wdbpipe -R "C:\Temp" -RW &
   ```

The default Tornado-shipped version of VxSim has preconfigured support for the Pass Through File System (PassFS) and Target Server File System (TSFS). This support, plus the above setup, enables the test object to be linked against the default version of the Runtime library with file communication (see Understanding and Building the Runtime Library) and to use one of the File Systems for transferring results.

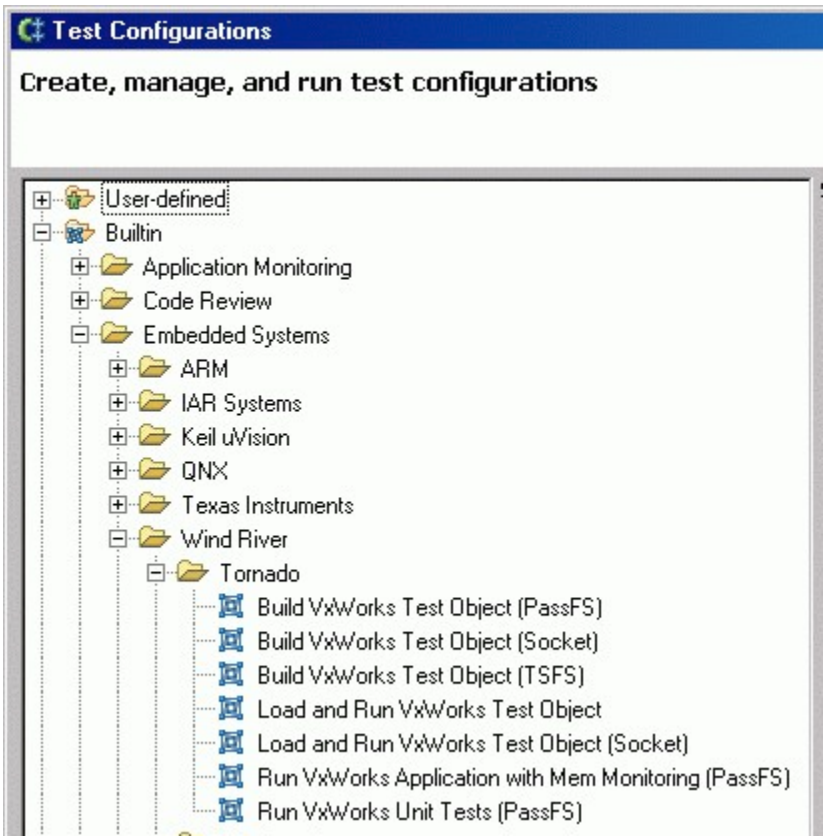
For more information on Tornado tools and VxWorks, refer to the Tornado manual.

Understanding the Tornado Test Configurations

There are several Test Configurations designed specifically to work with Tornado projects:

- Build VxWorksTest Object (PassFS)
- Build VxWorksTest Object (Socket)
- Build VxWorks Test Object (TSFS)
- Load and Run VxWorks Test Object
- Load and Run VxWorks Test Object (Socket)
- Run VxWorks Unit Tests (PassFS)
- Run VxWorks Application with Mem Monitoring (PassFS)

All are available in the **Embedded Systems> Wind River> Tornado** category.



These Test Configurations are all based on the following specially-designed Test Flow Recipes (see Customizing Test Configurations through Test Flow Recipes):

- **Build VxWorks test object - File Channel on PassFS**
  `(WRTornadoBuildVxWorksPassFS.recipe)`
- **Build VxWorks test object - Socket Channel**
  `(WRTornadoBuildVxWorksSocket.recipe)`
- **Build VxWorks test object - File Channel on TSFS**
  `(WRTornadoBuildVxWorksTSFS.recipe)`
- **Load and run VxWorks test object using Wind River Shell**
  `(WRTornadoLoadAndRun.recipe)`
- **Load and run VxWorks Test Object using Wind River Shell - Socket Comm**
  `(WRTornadoLoadAndRunSocket.recipe)`
- **Run unit tests on VxWorks - File Channel on PassFS**
  `(WRTornadoVxWorksPassFSFull.recipe)`
- **Run application on VxWorks - File Channel on PassFS**
  `(WRTornadoVxWorksPassFSApp.recipe)`

Here are short explanations of the terms used in the above Test Configurations and Test Flow recipes.

- **Build** - These Test Configurations build the Test Object, which includes the generation and reading of static coverage data (if enabled), but does not launch the Test Object or read any results.
- **Load and run** - These Test Configurations launch your Test Object using the Wind River Shell (Executing Test Objects for information on Test Object execution). The one for sockets uses a tool called "socket listener" to receive a result data stream from the machine running the test.
- **Run** -These Test Configurations are all-in-one configurations that build the Test Object, launch it, and read the results. Currently available for VxSim.
- **PassFS** - These use the Pass-through File System for results handling. This FS is only available on VxSim and—since VxSim is just a normal host application—provides a normal host's file-I/O. The results produced by a PassFS-using Test Object may be read using the default **"Utilities> Load Test Results (File)"** Test Configuration.
- **TSFS** - These use the Target Server File System feature and thus leverage the Target Server's ability to handle the target's file-IO requests and direct them to the host. The advantage of using the TSFS is that it can be used both with VxSim and the "real" targets. The TSFS components must be properly enabled and initialized in the VxWorks image that you use to run your application and the Target Server must be properly configured to allow it to write to a specific directory on the host's FS. Results produced by a TSFS-using Test Object are stored under this directory; to read them you have to adjust the **"Load Test Results - File"** Test Configuration to point to appropriate result files (see Customizing Test Configurations through Test Flow Recipes).
- **Socket** - These use network sockets to stream the test results between the target and host. A special listener is used to receive them. The network and socket components must be properly enabled and initialized in the VxWorks image that you use to run your application. Results may be read using the default **"Load Test Results (File)"** Test Configuration.
- **Application** - These test the application in Application Mode using its default entry point instead of executing Unit Tests.

- **Mem Monitoring** - These perform memory analysis on the tested application. For general information on performing runtime error detection with C++test, see Runtime Error Detection.

Other builtin Test Configurations you may need are:

- **"Unit Testing> Generate Unit Tests"**
- **"Utilities> Generate Stubs Using External Library Symbols"**
  Be sure to use this instead of **"Unit Testing> Generate Stubs"**
- **"Embedded Systems> Wind River> Extract Symbols from VxWorks Image"**

If you need to build a customized VxWorks/VxSim version to suit your needs (e.g., for C++ code, TSFS, network), refer to the Tornado manual for instructions.

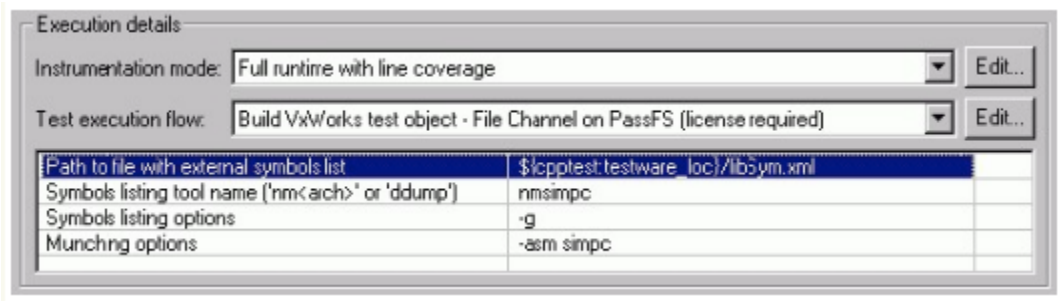
For more information on Test Configurations, see Configuring Test Configurations and Rules for Policies.

Customizing Test Configurations through Test Flow Recipes

It's important for embedded developers to become familiar with the flow of unit testing processing and the chaining of tools that C++test applies to the source code. This understanding is crucial for building a robust process that behaves exactly as you expect.

Much of the flow always remains the same across different environments. In most cases, the order and parameters of flow steps shouldn't be altered. However, in certain situations, some critical steps must be adjusted. You can configure test execution flow parameters to suit your specific needs by using an editor in the Test Configuration manager. Usually, there is no need to create a custom version of the execution flow because built-in Test Configurations allow easy editing of the most critical and commonly-modified flow properties.

For details on how to customize the test flow, see Customizing the Test Execution Flow.



In rare cases, more advanced customizations may be needed. In such cases, you can define a custom test flow as described in Defining a Custom Test Execution Flow: Advanced. If you need to do this, here are some tips and suggestions:

- For all communication modes (file, socket, and rs232), at the end of testing, the results data must always exist in the form of runtime log files stored on the host machine. These will be read by the C++test installation on the host.
- With file communication, the Test Object itself generates the runtime log files. Socket/rs communication must involve some listener running on the host that receives and stores the data streamed from the Test Object.
- The TSFS communication is file communication type for C++test. The I/O mechanism is beyond the tool's scope.
- The following steps and attributes may be a subject to review:
  - **TestRunnerGenerationStep** - paths of runtime log files (target's view), attributes: testLogFile, covLogFile
  - **ReadTestLogStep** - a path to **Test Log File** (host), attribute: **testLogFile**
  - **ReadDynamicCoverageStep** - a path to **Coverage Log File** (host), attribute**: covLogFile**

Debugging Test Cases

C++test does not support direct Test Cases debugging for this environment.

Load Test Executable to target kernel using 'hostShell' and set breakpoints on wanted Test Cases manually.