Directly Testing Wind River Workbench Projects

This section covers:

- Testing VxWorks RTP Projects
- Testing VxWorks DKM Projects
- Debugging Test Cases

Testing VxWorks RTP Projects

C++test supports direct testing of VxWorks Real Time Process Projects and Downloadable Kernel Module Projects. This topic describes how to configure testing for Real Time Process Projects

Prerequisites

Before you can start project setup, you need to complete the following actions:

1. Build the C++test Runtime library.
2. Add the library to the linker options. To do this, open the project properties panel and go to **Parasoft> C++test> Build Settings**. Then, modify the **Linker options** setting by appending the existing options with the path to the C++test runtime library.
3. If the runtime library was built with support for testing multi-threaded applications you need to specify the define `"-DCPPTEST_THREADS=1"` to enable support for multi-threaded testing.

Unit Testing Example

To explain how to use C++test to perform unit testing from Workbench, we will provide an example that walks you through the complete process of setting up the unit testing framework—starting from creating a new project and ending with reviewing unit testing results. For this example, we will use the "Timer" source code from the C++test examples directory (<C++test installation directory>/examples/Timer).

Creating a New Project

To create this project:

1. From the **File** menu, select **New> VxWorks Real Time Process Project**.
2. Enter the name `Timer`.
3. In the next panel, select **Flexible** build support.
4. In the Build Support panel, you might want to modify the make command line template to facilitate options scanning (for more details, see Important Note - Forcing Make to Unconditionally Build All Targets).
5. In the next panel, select the build specs. For this example, we will use `SIMPENTIUMgnu_RTP`.
6. Finish project setup.

Adding the Source Files to the Project

The next step is to add the source files to the project which will be tested. These source files can be added as a linked directory as follows:

1. In the Project Navigator, right-click the Timer project context, then choose **New> Folder** from the shortcut menu.
2. In the Folder panel, enter the name `src`.
3. Click **Advanced** to expand the additional settings area.
4. Select **Link to folder in the file system**.
5. Click the **Browse** button, then use the file chooser to select the **<C++test Install Dir>/examples/Timer** directory.
6. Click **Finish**.

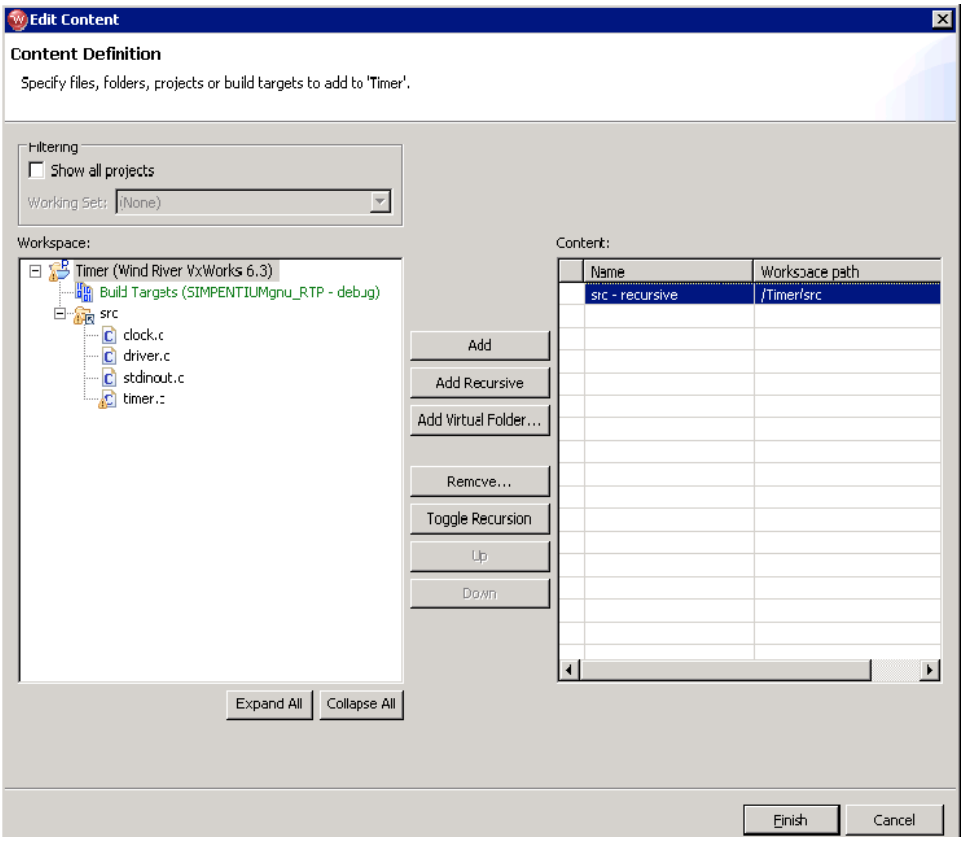
> **Note**
>
> Source files do not need to be added as linked directories. You can add them in any way that Workbench allows.

Defining the Build Target

Before you can generate test cases for a project, that project must be properly configured and able to successfully build. This requires defining a Wind River build target as follows:

1. From the Application Development perspective, go to the Project Navigator, right-click the **Build Targets** node, then choose **New Build Target** from the shortcut menu.
2. In the Build Target panel, leave the default settings as is (the name should be the same as the project name [Timer] and the build tool should be Linker), and click **Next**.

3. In the Content Definition panel, select src, then click the **Add Recursive** button to add the entire src directory to the build target definition. This will be the source base for the testing process. The current contents of the Project Navigator are shown in the right side.
4. Click **Finish**.



5. Choose **Project> Build Project**. The Standard Workbench build procedure will start.

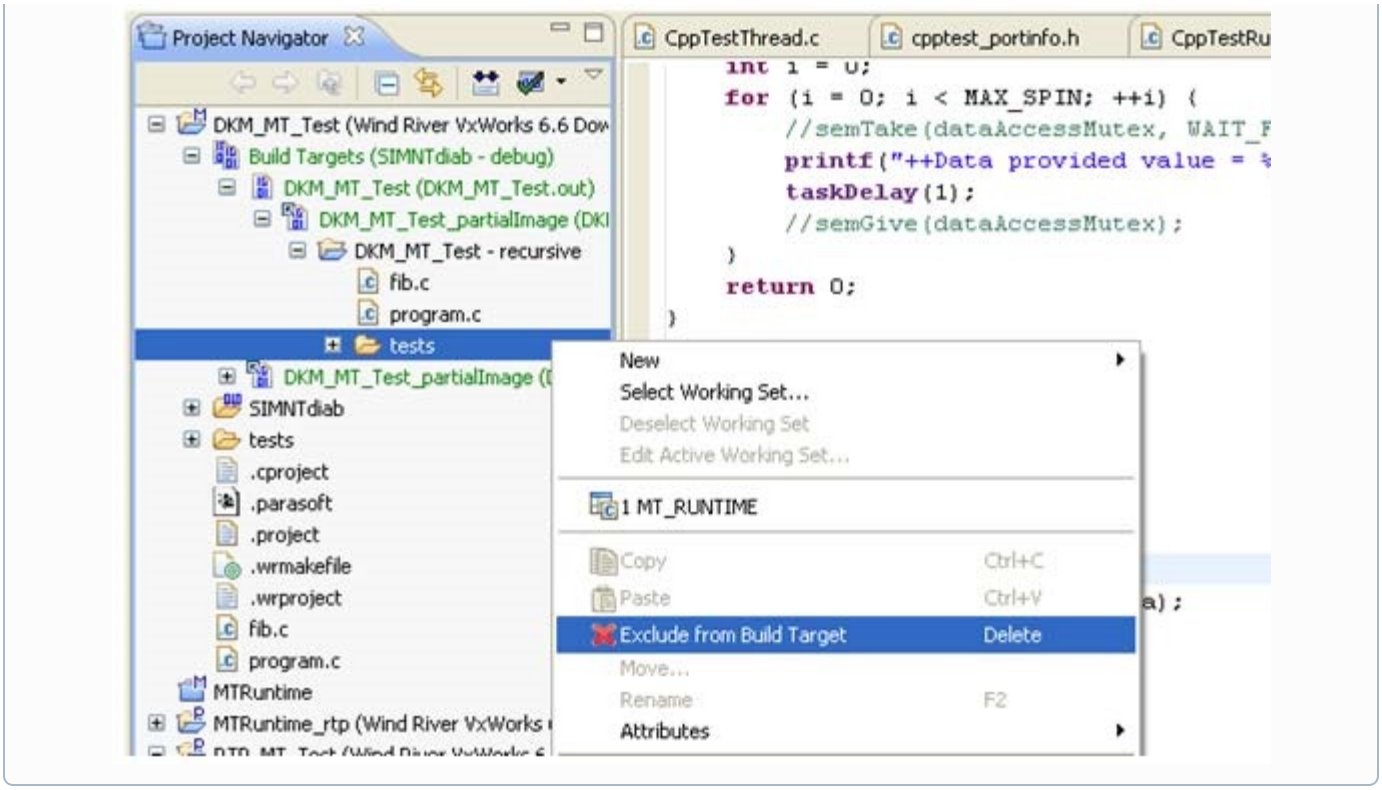
Important Note - Ensuring that C++test-Generated Directories are not Included in the Original Build Process

When you are configuring the build for a project that will be unit tested with C++test, it is important to realize that C++test generates test components into the main project directory, but you don't want these test components to become a part of your original project build. That's why your build configuration should be organized in such a way that the C++test-generated directories (for example `tests` or `stubs`) in the main project directory are not included in the original project build process. Otherwise, when you start to build your original application after C++test has generated test cases, the build will probably fail when it attempts to compile the generated test cases.

In short, C++test-generated test components are not designed to be a part of your original application. Test components are managed by C++test during the testing process only.

For projects with Standard build support, C++test automatically excludes the generated test components from the original build. No additional action is required. For Flexible build support, this automatic exclusion is not supported and should be done manually.

To exclude C++test generated directories from the build, you need to locate these directories in the build target definition, select them one-by-one, right-click the selection, and choose **Exclude from build target** from the shortcut menu.



Generating the Test Cases

Once the project can built to the VxWorks Real Time Process (Application), you can start automated test case generation as follows:

1. Open the C++test perspective.
2. Select the Navigator node that represents the resource for which you want to generate tests cases.
    - A valid selection should contain testable C/C++ source files, directories, or the entire project. Selection of other project items (including selection of nodes representing the contents of Flexible build targets) is not currently supported.
3. Start the test case generation in one of the following ways:
    - From the **Parasoft** menu, choose **Test Using> Builtin> Unit Testing> Generate Unit Tests**.
    - Open the pull-down menu for the **Test Using** toolbar button (this is a blue triangle), then choose **Test Using> Builtin> Unit Testing> Generate Unit Tests**.

C++test will scan the compilation options and then generate test cases. The summary from the generation is available on the C++test progress view. After you close this panel, you will see that a new `tests` directory was added to the project tree. This directory contains the automatically-generated test cases.

Building the Tests

Generated test cases need to be built together in the executable application. This process is fully managed by C++test and its internal builder. By default, C++test provides a group of Test Configurations which are preconfigured to work with the Wind River workbench environment. These Test Configurations are located in the **Builtin> Embedded Systems> Wind River Workbench** Test Configurations category.

The following Test Configurations are provided:

- **Build VxWorks Test Executable - RTP (PassFS):** This configuration will try to collect existing test cases, analyze stub configurations, prepare test instrumentation, and build the test executable with results sent via the Wind River's Pass-through File System (PassFS) communication channel (which means that the test executable will produce the results using standard ANSI C file i/o functions to the file system). This file communication channel works with VxWorks simulators (VxSim) only.
- **Build VxWorks Test Executable - RTP (Socket):** This configuration will try to collect existing test cases, analyze stub configurations, prepare test instrumentation, and build the test executable with results sent via the TCP/IP Sockets communication channel (which means that the test executable will open two ports—one for test data and one for coverage data—and try to send test executions results via this channel).
- **Build VxWorks Test Executable - RTP (TSFS):** This is similar to the 'PassFS' equivalent, except that it uses Wind River's Target Server File System (TSFS) communication channel. It may be used on 'real' targets as well.

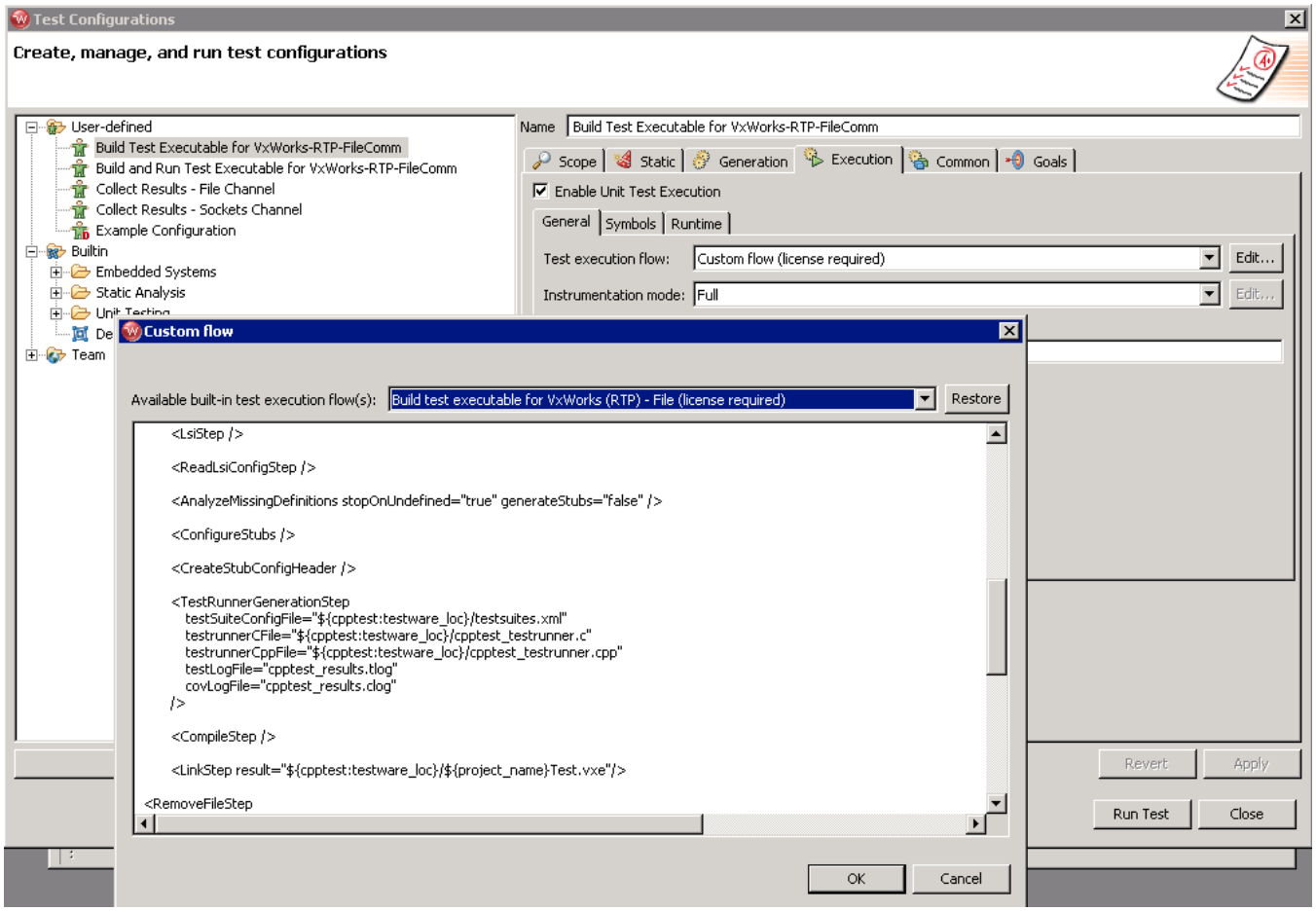
Using the File Communication Channel

If you choose the file communication channel (Test Configurations marked with 'PassFS' or 'TSFS'), you might need to configure the path where the results files will be generated. By default C++test is using following paths to store the results.

- TSFS configuration:
  `testLogFile="/tgtsvr/cpptest_results.tlog"`
  `covLogFile="/tgtsvr/cpptest_results.clog"`
- PassFS configuration:
  `testLogFile="host:${cpptest:testware_loc}/cpptest_results.tlog"`
  `covLogFile="host:${cpptest:testware_loc}/cpptest_results.clog"`

To change any of this paths:

1. Duplicate the appropriate Test Configuration to the User-Defined area.
2. Open the **Execution> General** tab.
3. In the **Test execution flow** box, select **Custom flow**.
   - Note that this requires the Embedded Support license option.
4. Click the **Edit** button to the right of the **Test execution flow** box.
5. In the Custom Flow dialog that opens, select **Build test executable for VxWorks (RTP) – File Channel on PassFS (license required)** or **Build test executable for VxWorks (RTP) – File Channel on TSFS (license required)** from the **Available built-in test execution flow** box, then click **Restore**.



The test execution flow definition will display in the main area of the Custom Flow dialog. At this point, you probably want to modify the flow step which controls the communication channel definition.

This step is typically defined as follows:

```
<TestRunnerGenerationStep
                    testSuiteConfigFile="${cpptest:testware_loc}/testsuites.xml"
                    testrunnerCFile="${cpptest:testware_loc}/cpptest_testrunner.c"
                    testrunnerCppFile="${cpptest:testware_loc}/cpptest_testrunner.cpp"
                    testLogFile="/tgtsvr/cpptest_results.tlog"
                    covLogFile="/tgtsvr/cpptest_results.clog"
                    appendLogs="false"
          / >
```

If your system requires different path formats or final file locations, you can configure this here by modifying the testLogFile and covLogFile attribute values. In most cases, the default values for TSFS and PassFS should be fine.

Once you have made all the required modifications, you can run the customized Test Configuration that defines your preferred flow. After the Test Configuration is done running, the test executable binary file will be available. After this executable is started on the target device or on the simulator, the results files will be produced according to your Test Configuration (as defined in the <TestRunnerGenerationStep> tag).

The next step is collecting these results and having them loaded into the C++test results view. This can be achieved with the help of the built-in **Utilities> Load Test Results (Files)** Test Configuration. You may need to modify this Test Configuration to adjust the paths to the results files. In general, you need to ensure that the result files path specified in the following flow steps from the configuration used for building tests

```
<TestRunnerGenerationStep
                  testSuiteConfigFile="${cpptest:testware_loc}/testsuites.xml"
                  testrunnerCFile="${cpptest:testware_loc}/cpptest_testrunner.c"
                  testrunnerCppFile="${cpptest:testware_loc}/cpptest_testrunner.cpp"
                  testLogFile="cpptest_results.tlog"
                  covLogFile="cpptest_results.clog
       />
```

remain in sync with the paths in the "read logs file" tags. Below is a snippet from the **Load Test Results (Files)** Test Configuration's test flow definition:

```
<ReadTestLogStep
                  testLogFile="cpptest_results.tlog"
                  timeoutInfoProperty="test_exec_timeouted"
       />
       <ReadDynamicCoverageStep
                  covLogFile="cpptest_results.clog"
       />
```

If your system allows automated execution of the test executable, you can extend the test flow used to build the test binary so that it also runs the executable and collect results. This allows automation of the complete testing loop.

After your results have been successfully loaded into Workbench, you can analyze them, generate reports, and modify the test cases accordingly. For details, see:

- Viewing Results
- Exploring Test Results
- Extending and Modifying the Test Suite
- Understanding Reports

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

Using the Socket Communication Channel

If you select the socket communication channel, you might need to configure the host IP address and port values which will be used to send results. To do this:

1. Duplicate the Build VxWorks Test Executable - RTP (Socket) Test Configuration to the User-Defined area.
2. Open the **Execution> General** tab.
3. In the **Test execution flow** box, select **Customflow**.
    - Note that this requires the Embedded Support license option.
4. Click the **Edit** button to the right of the **Test execution flow** box.
5. In the Custom Flow dialog that opens, select the **Build test executable for VxWorks (RTP) –Socket (license required)** built-in test execution flow definition from the **Available built-in test execution flow box**, then click **Restore**.

The test execution flow definition will display in the main area of the Custom Flow dialog.

At this point, you probably want to modify the flow step which controls the communication channel definition. This step typically is defined as follows:

```
<TestRunnerWithSocketsGenerationStep
                  testSuiteConfigFile="${cpptest:testware_loc}/testsuites.xml"
                  testrunnerCFile="${cpptest:testware_loc}/cpptest_testrunner.c"
                  testrunnerCppFile="${cpptest:testware_loc}/cpptest_testrunner.cpp"
                  resultsHost="127.0.0.1"
                  testLogPort="2567"
                  covLogPort="2568"
/ >
```

Depending on your network configuration, you might want to configure the results host IP address (the machine on which your development environment with C++test is running) and the values used for the port.

Once you have made all required modifications, you can run the customized Test Configuration that defines your preferred flow. After the Test Configuration is done running, the test executable binary file will be available. Before the test executable is started, run the **"Utilities> Load Test Results (**

Sockets)" Test Configuration. Then, after this executable is started on the target device or on the simulator, the two ports will be opened for sending the test and coverage results.

Note - More about the "Utilities> Load Test Results (Sockets)" Test Configuration

For collecting results via the socket communication channel, you need to start this Test Configuration *before you start the test executable*. Otherwise, the Test Configuration will hang waiting for results to arrive on the specified ports. The results collection is performed via a simple listening agent (written in Java) which is capable of listening on the given port and writing the data from this port to a file. The source code of this listener, together with its compiled versions, is located in `<C++test Install Dir>/engine/runtime/listeners/socket_listener`

After your results have been successfully loaded into Workbench, you can analyze them, generate reports, and modify the test cases accordingly. For details, see:

- Viewing Results
- Exploring Test Results
- Extending and Modifying the Test Suite
- Understanding Reports

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

Running the Test Executable and Reading Results

You can run a built Test Executable using the **"Builtin> Embedded Systems> Wind River> Workbench> Load and Run VxWorks Test Executable (RTP)"** Test Configuration and then use the **"Utilities> Load Test Results (Files)"** Test Configuration.

To build and run your application in the application mode use one of the all-in-one **"Builtin> Embedded Systems> Wind River> Workbench> Run VxWorks Application with Mem Monitoring - RTP"** Test Configurations. There is one for PassFS and one for TSFS communication.

Testing VxWorks DKM Projects

There are two Test Configurations designed specially for testing the VxWorks Downloadable Kernel Modules (DKMs):

- **Builtin> Embedded Systems> Wind River> Workbench> Build VxWorks Test Module -DKM (PassFS):** Builds the Test Module using Wind River's Pass-through File System
('PassFS') communications. This configuration builds the test binary in the form of a download-able kernel module, including ctdt.c file generation.
- **Builtin> Embedded Systems> Wind River> Workbench> Build VxWorks Test Module -DKM (TSFS):** Builds the Test Module using Wind River's Target Server File System ('TSFS') communications. This configuration builds the test binary in the form of a downloadable kernel module, including ctdt.c file generation.

'PassFS' may be used only with VxWorks simulators (VxSim) and provides direct access to the hosts' file system. 'TSFS' uses the Target Server's ability to access the host's file system and Target Agent <-> Target Server WTX channels for communication. It can be used on 'real' targets too.

All Workbench-related recipes contain some initial properties to be set or adjusted, like the preferred toolchain or Wind River Shell executable. You may choose between the provided options according to their comments or redefine the values for your needs. The initial echo steps are there to present you console messages reminding you about this during testing.

Note

When using 'PassFS' communications on Windows hosts, you must prepend the paths of result logs with the 'host:' prefix. When using 'TSFS' communications, use the '/tgtsvr/' prefix.

For more information on VxWorks features refer to the Workbench documentation.

There are two options for testing DKM projects: simple and full.

The Simple Approach

The simple option doesn't allow for automated stub generation and advanced unresolved symbols reporting, but it doesn't require generation of an external symbol lists. Consequently, less effort is required to begin testing.

To prepare a configuration for the simple approach:

1. Open the Test Configurations manager.
2. Duplicate the **"Builtin> Embedded Systems> Wind River> Workbench> Build VxWorks Test Module - DKM"** Test Configuration.

3. Open the **Execution> Symbols** tab.
4. Set **Library Symbols Identification mode** to '**Off** (assume all symbols are available).'
5. Click **Apply**, then **Close**.

The Full Approach

The full option allows full unit testing, but requires the external symbol list to be provided during Test Module creation. The external symbol list must contain all the exported/defined kernel space symbols from the VxWorks image used inside your DKMs.For information on how to create the external symbol lists, see Providing an External List of Library Symbols.To build a Test Module using this approach, use one of the default **"Builtin> Embedded Systems> Wind River> Workbench> Build VxWorks Test Module - DKM"** Test Configurations.

Running the Test Module and Reading Results

To run the built Test Module, use the **"Builtin> Embedded Systems> Wind River> Load and Run VxWorks Test Object (DKM)"** Test Configuration. This launches your test object using the Wind River Shell.

To read the results use the **"Utilities> Load Test Results (Files)"** Test Configuration.

> **Tip**
>
> For more information on Test Flow Recipes, which you may need to customize in order to achieve the expected testing results, see Customizing the Test Execution Flow.

To build and run your application in the application mode, use one of the all-in-one (excluding library symbols extraction) **"Builtin> Embedded Systems> Wind River> Workbench> Run VxWorks Application with Mem Monitoring - DKM"** Test Configurations. There is one for PassFS and one for TSFS communication.

If your entry point's name is not main, adjust your entry point—both through the Test Configuration properties and by adding `edg.pMainFunctionName <entry_name>` to the **Other Settings> Advanced options** area in the C++test project properties panel.

Debugging Test Cases

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

Use appropriate Debug/Launch Configuration for your original/tested project to load Test Executable and set breakpoints on wanted Test Cases manually.