Importing and Testing ADS 1.2 CodeWarrior Projects

This topic explains how to import ADS 1.2 CodeWarrior projects and test them with C++test. Sections include:

Prerequisites

- For C++test standalone, you need to have the "mcp2make" utility (provided by ARM, available for free on ARM's support downloads page) downloaded and installed.
- If you have several different ARM development environments (or debuggers/emulators) you need to use the ARM Suite Switcher utility to set the proper environment (ADS 1.2/AXD 1.2.1).

Testing ADS 1.2 Projects Using C++test (Standalone)

To test CodeWarrior based projects with C++test standalone:

1. Using CodeWarrior's "Export Project" option (**File> Export Project**), save a project's definition file as an XML file. By default, it will be saved in the same directory as the original (.mcp) file.
2. Use mcp2make to convert the .xml file to a proper Makefile. The procedure is straightforward, and the resulting Makefile is also created in the same directory.
3. Use `cpptestscan` to scan the Makefile, extracting the options for C++test (see Creating a C++test Project Using an Existing Build System).
4. Import the generated .bdf (Build Data File) into C++test via the GUI or the command line. See Creating a C++test Project Using an Existing Build System details.

Testing ADS 1.2 Projects Using C++test for ARM (RVDS Plugin)

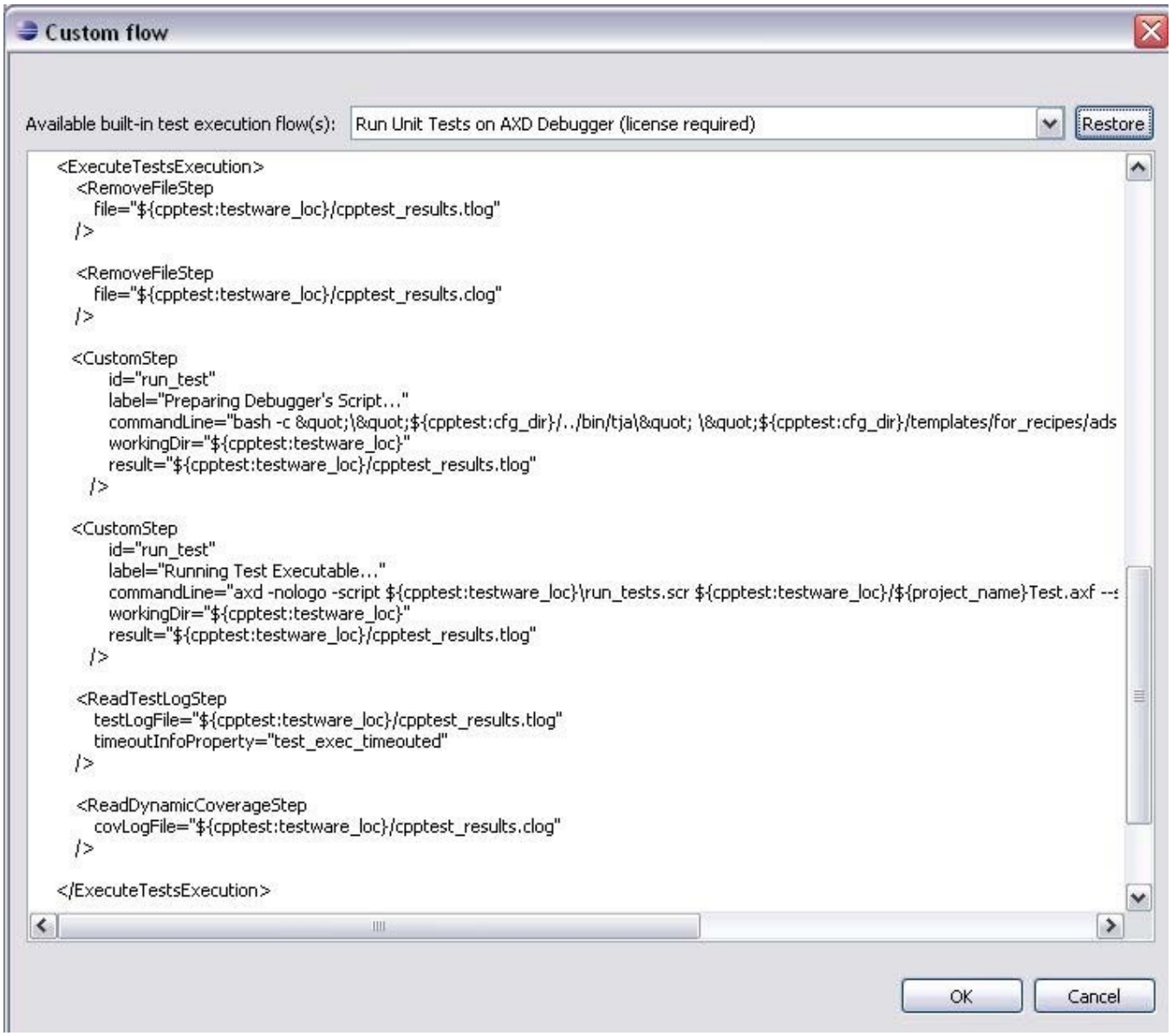
To test CodeWarrior based projects with the C++test plugin for ARM (RVDS):

1. Using CodeWarrior's "Export Project" option (**File> Export Project**), save a project's definition file as an XML file. By default, it will be saved in the same directory as the original (.mcp) file.
2. Using the RVDS Eclipse IDE, import the XML file to create a new project in the workspace. To do this, open the ARM Eclipse IDE, choose **File> Import**, then choose **C/C++> CodeWarrior Project exported as XML.**

You can then continue testing in the normal manner.
Note that this process must be repeated each time the project's settings or structure change. This will be improved in future versions of C++test.

Understanding the ADS 1.2 Test Configuration

C++test provides a Test Configuration designed specifically to work with ADS projects: Run ADS 1.2 Tests. It is designed to run the previously built test executable on AXD Debugger and then collect its results. The core of the test flow looks like the following:



The Test Configuration is available in the **Builtin> Embedded Systems> ARM** category.

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. Much of the flow always remains the same across different environments, and in most cases, the order and parameters of flow steps shouldn't be altered. However, in certain situations, some critical steps must be adjusted. To allow this adjustment, C++test introduces the concept of a "test flow recipe"; this recipe defines the order and parameters of flow steps. The recipe is an XML-like file with entities/tags taken from a predefined set and each representing one step. The required or optional step parameters are provided using attributes.

To customize a recipe:

1. Open the Test Configurations panel by choosing **Parasoft> Test Configurations.**
2. Select the user-defined Test Configuration that you plan to use for test execution.
3. Open the **Execution> General** tab.
4. Select **Custom flow (license required)** from the **Test Execution flow** combo box, then click the **Edit** button
5. Enter your modified test flow.
    - If you want to adapt an existing test flow, choose that test flow from the **Availablebuilt-in test execution flow** box, then click **Restore**. The XML for that test flow will then be displayed, and can be edited as needed (e.g., for ADS 1.2 testing, **Run Unit Tests on AXD Debugger** can be chosen as the base for modifications).
6. Click **OK** to save the modified file. The XML document will then be validated. If any problems are found during the validation, they will be reported.

Some steps may need to be customized for embedded testing: typically, steps that control how the test executable is transferred into the target device and how the test results are being sent back to C++test. Currently, there are two options: file or socket communication. In the end, the results data must always exist in the form of runtime log files on the host machine so that C++test can read the logs. With file communication, the test object itself generates these logs. With socket communication, a listener must be running on the host to receive and store the data streamed from the test object.

With file communication, the following steps should be reviewed/adjusted:

- **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`

With socket communication, the following steps should be reviewed/adjusted:

- **TestRunnerWithSocketsGenerationStep** - Connection parameters (to host), attributes: `resultsHost, testLogPort, covLogPort`
- **listener's settings**
- **ReadTestLogStep** - A path to Test Log File (host), attribute: `testLogFile`
- **ReadDynamicCoverageStep** - A path to Coverage Log File (host), attribute: `covLogFile`

Here are explanations of the main ADS configuration steps. If you are going to adjust the flow recipe, it is helpful to understand these main steps:

Part 1

The following tag informs C++test that we want to treat the next statements as part of the test execution process.

```
<ExecuteTestsExecution>
```

Part 2

The following two tags remove any results files that were created from prior tests.

```
<RemoveFileStep
   file="${cpptest:testware_loc}/cpptest_results.tlog"
/>
<RemoveFileStep
   file="${cpptest:testware_loc}/cpptest_results.clog"
/>
```

Part 3

The following step runs C++test's internal scripts utility named Tja. It creates the run script for the RVDS Debugger to run the produced test executable (`run_tests.scr`). Note that BASH is used to redirect results from standard output to a script file – BASH must be on the system PATH variable in order for this process to work smoothly.

```
<CustomStep
      id="run_test"
      label="Preparing Debugger's Script..."
         commandLine="bash -c &amp;quot;\&amp;quot;${cpptest:cfg_dir}/../
bin/tja\&amp;quot; \&amp;quot;${cpptest:cfg_dir}/templates/for_recipes/
ads_test.tja\&amp;quot; &amp;gt;
\&amp;quot;${cpptest:testware_loc}\run_tests.scr\&amp;quot;&amp;quot;"
               workingDir="${cpptest:testware_loc}"
               result="${cpptest:testware_loc}/cpptest_results.tlog"
/>
```

Part 4

The following step invokes the axd tool and passes the argument of the `run_tests.scr` batch file. This gives the ADS Debugger the details it needs to load the appropriate target, open the executable, run it, and produce the `.tlog` and `.clog` results files.

```
<CustomStep
        id="run_test"
        label="Running Test Executable..."
        commandLine="axd -nologo -script
${cpptest:testware_loc}\run_tests.scr ${cpptest:testware_loc}/
${project_name}Test.axf --start-after=${cpptestproperty:
test_case_start_number}"
        workingDir="${cpptest:testware_loc}"
        result="${cpptest:testware_loc}/cpptest_results.tlog"
/>
```

Part 5

The following two steps read the results and display them in the GUI and in the final report.

```
<ReadTestLogStep
    testLogFile="${cpptest:testware_loc}/cpptest_results.tlog"
    timeoutInfoProperty="test_exec_timeouted"
/>
<ReadDynamicCoverageStep
    covLogFile="${cpptest:testware_loc}/cpptest_results.clog"
/>
```

Part 6

The following tag ends the execution section.

```
</ExecuteTestsExecution>
```

For more information on Test Flow Recipes, see Customizing the Test Execution Flow

Troubleshooting

An error message such as the following indicates that a file was too large for the ADS 1.2 compiler to handle:

```
--------------------------------------------------------------------------
---------------
line 9: Fatal error: Internal fault: 0x5e0c in
'TestSuite_KZ2dL_c_fa6c0755_test_main_4'
Please contact your supplier.
Process exited with code: 1 (error).
-----------------------------------------------
```

If this occurs, try the following workarounds:

- Reduce number of test cases for the file.
- Change the instrumentation mode (for example, disabling coverage instrumentation), which should result in a smaller instrumented file. You can do this by opening the Test Configuration Manager, going to the **Execution> General** tab, then select the **Full runtime w/o coverage** option for **I nstrumentation Mode**.