VECTOR



VectorCAST/QA

User's Guide

VectorCAST 2026

New editions of this guide incorporate all material added or changed since the previous edition. Update packages may be used between editions. The manual printing date changes when a new edition is printed. The contents and format of this manual are subject to change without notice.

Generated: 10/19/2026, 8:24 PM

Rev: 6be4125

Part Number: VectorCAST/QA User's Guide for VectorCAST 2026 SP5

U.S. Government Restricted Rights

This computer software and related documentation are provided with Restricted Rights. Use, duplication or disclosure by the Government is subject to restrictions as set forth in the governing Rights in Technical Data and Computer Software clause of

DFARS 252.227-7015 (June 1995) and DFARS 227.7202-3(b).

Manufacturer is Vector North America, Inc. East Greenwich RI 02818, USA.

Imprint

Vector reserves the right to modify any information and/or data in this user documentation without notice. Vector disclaims all liabilities for the completeness or correctness of the contents and for damages which may result from the use of this documentation. This documentation nor any of its parts may be reproduced in any form or by any means without the prior written consent of Vector. To the maximum extent permitted under law, all technical data, texts, graphics, images and their design are protected by copyright law, various international treaties and other applicable laws. Any unauthorized use may violate copyright and other applicable laws or regulations.

© Copyright 2026, Vector Informatik, GmbH. All rights reserved.

Protected Trademarks

All brand names in this documentation are either registered or non-registered trademarks of their respective owners.

Third-Party copyright notices are contained in the folder: 3rdPartyLicenses, located in the VectorCAST installation directory.

TABLE OF CONTENTS

Introduction 10

VectorCAST/QA Overview 11
How It Works 11

Work Flow Scenarios 13

Initial Setup 14
Preliminary Setup 14
Launch VectorCAST/QA 16
Create a New System Testing Environment 16

Instrument For Coverage 22
Set Coverage Type 22
Instrument for Coverage 22
Append Cover IO 23
Close the Environment 24

Edit Testing Script and Build 25
Edit the systems_tests.py script 25
Build the Instrumented Code 28

Run a Test Interactively 30
Run a Single Test 30
View the Execute Log 31
View Coverage Results 32
Execute All Tests 33

System Test Automation 34

The Python Configuration Script 35

Add a Test Case Using Python Configuration Script 36

Add a Manual Test 37

Apply Coverage to Source Tree Option 40
Benefits of the Apply Coverage to Source Tree Option 40
How the Apply Coverage to Source Tree Option Works 40
Apply Instrumentation to Source Tree Using the "During Build" Option 41
Apply Instrumentation to Source Tree Using the "Always" Option 41
Apply Instrumentation to Source Tree Using the "Never" Option 42
Changing the Apply Coverage to Source Tree Option 42

Incremental Build and Execute 42
Run Incremental Build/Execute Automatically 43

Interactive Execution of Tests 43

Component Coverage 46

Enabling Component Coverage 46
Building and Executing on Components 47
Viewing Component Coverage Data 49

Change Impact Report 50
Estimated Change Impact Report 51
Full Change Impact Report 51

VectorCAST Cover Environment 53

Creating a System Testing Environment 54
To Set the Working Directory 54
Troubleshooting the Working Directory 55
To Create a System Testing Environment using the Wizard 56
Step 1: Choose Compiler 57
Step 2: Name the Environment 58
Step 3: Locate Source Files 59
Step 4: Coverage Options 63
Step 5: Summary 64

Creating a Coverage Environment 65
Step 1: Choose Compiler 66
Step 2: Name the Environment 67
Step 3: Locate Source Files 68
Step 4: Coverage Options 72
Step 5: Summary 73
Create a Regression Script for the Environment 75
To Use a JSON Compilation Database to Build an Environment 79

Working With Environments 80
To Close an Environment 80
To Rename an Environment 80
To Update an Environment 81
To Create a Cover Environment Script 85
To Delete an Environment 86
To Open an Environment 87

Adding Source Files to the Environment 87
To Add a Source Code File 87
To Add Header Files as Source Files (C/C++ only) 89
To View Source Files in a Hierarchy 90
To View Source Code 91
To Remove a Source File 92
To Remove All Source Files 92

Working With Base Directories 92
Moving Base Directories 92

Setting Coverage 97
To Set Coverage Type 97
Instrumenting Files 99
View the Instrumented Source File 101

Instrument for Statement Coverage ... 101
Instrument for Branch Coverage ... 102
Instrument for Basis Paths Coverage ... 103
Instrument for MC/DC Coverage ... 104
Suppressing MC/DC Analysis ... 105
Instrument Statement+MC/DC ... 106
Instrument Statement+Branch ... 107
Instrument for Function Coverage ... 107
Instrument for Function + Function Call Coverage ... 108
Instrument for Probe Point ... 108
List the Number of Bytes Needed to Instrument Each Unit in an Environment ... 109
Change the Type of Coverage ... 109
Un-Instrument Source Files ... 110
Re-Instrument All Source Files ... 111
Avoid Instrumenting Sections of Source Code ... 112

Compiling the Instrumented Source Files ... 113
Simple Compilation Examples ... 113
The Cover I/O File ... 114
To Customize the Cover I/O File ... 115
To Change the Output Options for Test Results ... 116
To Create a Test Result File ... 118

Viewing Coverage Results ... 118
To Add a Test Result ... 118
To Update a Test Result ... 120
To Open the Test Results Viewer ... 121
The Notes Tab of the Test Results Viewer ... 122
The Requirements Tab of the Test Results Viewer ... 123
To Open the Coverage Viewer ... 125
To Turn on Coverage Results ... 125
The Coverage Viewer ... 126
To Customize the Coverage Viewer ... 132
Map Line Selection to Original Source View ... 132
To Determine Which Test Result Covers a Line ... 132
To Delete All Result Files ... 133

Viewing Coverage Reports ... 133
To View the Metrics Report ... 134
To View the Function Call Report ... 137
To View a Basis Paths Report ... 138
Understanding Basis Paths ... 139
To Build Test Cases from Basis Path Analysis ... 139
MC/DC Coverage ... 140
To View the Equivalence Matrices Report ... 141
Understanding MC/DC Analysis ... 142
To View an MC/DC Report ... 144
To Print Coverage Reports ... 144
To Save Coverage Reports to Disk ... 145
To View Code Coverage Summary ... 147

Viewing Execution Flow with Animated Coverage 151
To Activate Coverage Animation 151
To Play the Coverage Animation 152
The Animation Toolbar 153
To Set a Breakpoint 154
Importing Coverage Results 154
To Import Coverage Results 154
To Export a Coverage Script 157
To Import a Coverage Script 158
Delete Imported Results 158
Setting Compiler Options 159
Options: C/C++ Tab 161
To Automatically Add Include Path and Defined Variables 167
To Test Your Compiler Settings 170
Setting Coverage Options 188
General Coverage Options 188
Instrumentation Options 194
MC/DC Options 208
Miscellaneous Options 211
Suppressed Coverable Functions Option 216
Result Changes 220
Input / Output Options 220
CLICAST-Only Coverage Options 222
The Options Append Command 223
Setting Coverage Viewer Options 223
To Format the Text in the Coverage Viewer 223
To Change the Fonts 224
To Reset the Fonts to Default 224
To Change Text Color 224
To Change Background Color 225
Other Background Colors 226
Cover Environment Options 226
Auto Generate Test Result Names 226
Append to Coverage Data File 227
Starting Unit Number 227
Types File Extension (Ada) 228
Ada File Extensions 228
Setting Report Options 229
Report Content Options 229
Report Format Options 236
Change-Based Testing (CBT) 243
Change-Based Testing 244
Using VectorCAST Covered By Analysis 245

Covered By Analysis (CBA) ... 246
To Add Coverage Analysis ... 246
Using the Coverage Analysis Editor ... 247
To Edit an Existing Analysis Result ... 250
Viewing CBA Coverage in the Coverage Viewer ... 251
To Change Covered-By-Analysis Display Color ... 252
Working With Analysis Files ... 253
To Import Analysis Files ... 253
To Export Analysis Files ... 253
To Remove Analysis Files ... 253
Re-Instrumenting With CBA Data ... 253
Viewing Analysis Data in Reports ... 253
Covered By Analysis Report ... 253
Aggregate Coverage Report ... 256
Metrics Report ... 257
Using Coverage Analysis With SFP ... 257
To Add Coverage Analysis ... 258
To Add or Remove Coverage Analysis for Multiple Lines ... 258
To Add or Remove Coverage Analysis for MC/DC ... 258
To Edit an Existing Result ... 259
Changes for the Covered By Analysis Report ... 259
The Aggregate Report ... 259
Using VectorCAST Probe Points ... 260
Probe Points ... 261
Function Probe Points ... 261
Function Entry Probe Points ... 262
Function Exit Probe Points ... 263
Probe Point Order of Execution ... 264
Using the Probe Point Editor ... 265
Open the Probe Point Editor ... 265
Insert a Probe Point ... 266
Insert a File Scope Probe Point ... 267
Test Compile a Probe Point ... 268
Saving and Applying Probe Points ... 270
Apply All Probe Points ... 271
Probe Point Status Buttons ... 272
Edit a Probe Point ... 272
Built-in Functions and Macros ... 273
Deactivate/Activate Probe Points ... 274
Remove Probe Points from Editor ... 275
Delete All Probe Points ... 275
Probe Point-Only Instrumentation ... 275
Capture Local Variable Values ... 277

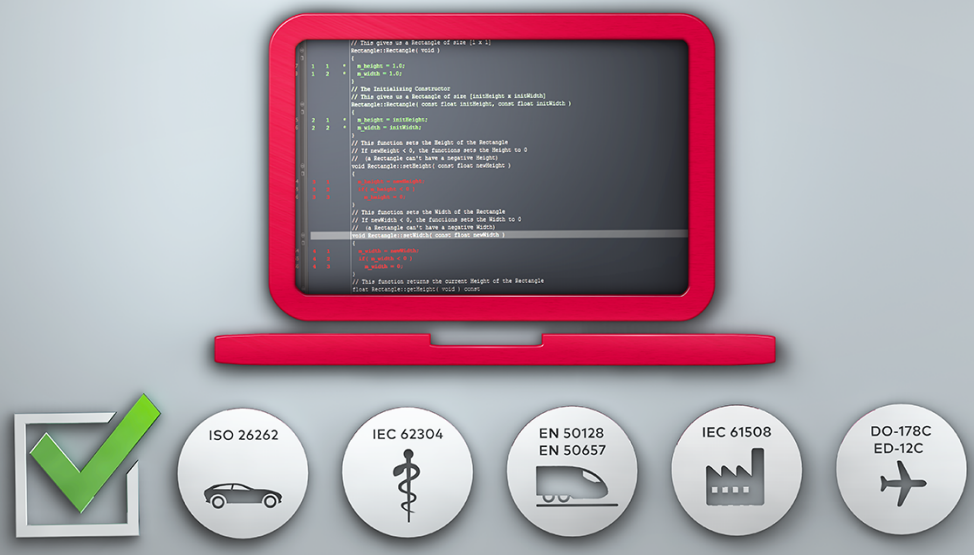
Inject Spurious Values 278
Patch Faulty Code 278
Probe Point Listing 279
Probe Point Output Report 280
Using the Probe Point API 281
The Probe Point File 281
Probe ID Number 282
Export a Probe Point 283
Import a Probe Point 284
Enable, Disable and Remove Probe Points 284
Create a Probe Point Report 285

Using Coupling Verification 287

Understanding Control and Data Coupling 288
Using VectorCAST/Coupling 289
The Cover Environment 290
Create the Components File 290
Perform Coupling Analysis 291
Apply Coupling to Instrumented Source Files 292
Build the Instrumented Application 292
Run Test Cases and Import Trace Data 293
Generate Coupling Coverage Report 293
Re-analyze Coupling 295
Setting Coupling Options 295
Coupling Reports 295
Component Report 296
Coupling Coverage Report 297
Coupling Access Order Full Report 300
Coupling Access Order Error Report 301
View Coupling Logs 303
Example - Analysis Log 304
Example - Instrumentation Log 305
Example - Coupling Report Log 306
Example - Access Order Full Report Log 306
Example - Access Order Error Report Log 306
Data and Control Coupling Limitations and Error Checking 307
Combining Coupling and Coverage Instrumentation 307
Data Couples That Are Compound Types 307
Data Couple Access Errors 307
Function Pointers 309
Control Couples Called via Function Pointers 310

VectorCAST/QA and CUDA® Integration 313

Using VectorCAST/QA with CUDA® Applications 314
CUDA Language Extension Support 314
Multi-GPU Build Process Support 314
CUDA Processing Scripts 314
Building 315
Partial Instrumentation 315
VectorCAST CUDA® Example 316
VectorCAST/QA and Jenkins Integration 317
VectorCAST/QA and Jenkins Integration 318
Index 319



Introduction

VectorCAST/QA Overview

VectorCAST/QA is a system testing automation tool with code coverage. VectorCAST/QA provides the following benefits:

- Reduces test time with Change-Based Testing (CBT)
- Simplifies running tests with automation control
- Measures testing completeness with code coverage metrics
- Takes the guesswork out of qualifying release readiness criteria

System Testing means testing the entire integrated application to verify user-facing functionality. With the automation provided by VectorCAST/QA, the complexity of running system test is taken care of by the tool, allowing anyone to run the tests at any time and fostering collaboration.

VectorCAST/QA is designed for use with build automation systems such as Jenkins. This extends the traditional DevOps infrastructure to include embedded device software and allows for a common set of processes for all the software in your project. Using a DevOps infrastructure allows you to efficiently perform system tests and to test common code across multiple configurations.

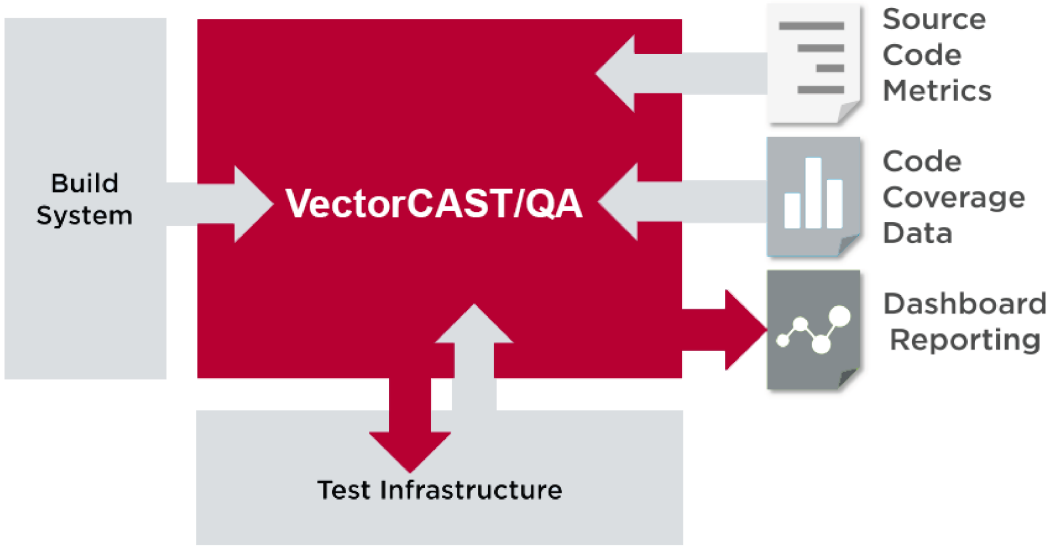
VectorCAST/QA leverages the VectorCAST Change-Based Testing technology to reduce the testing cycle time. Change-Based Testing uses advanced heuristics path and call tree analysis to identify the subset of tests affected by each source change and then automatically reruns only those tests. This incremental execution of tests is much faster than executing the entire test suite, which can take hours. It also means that when the QA team does run the full test suite, they are likely to have a clean test run.

In short, VectorCAST/QA allows team members to collaborate on test activities, shortens test times and provides up-to-date metrics on release readiness.

How It Works

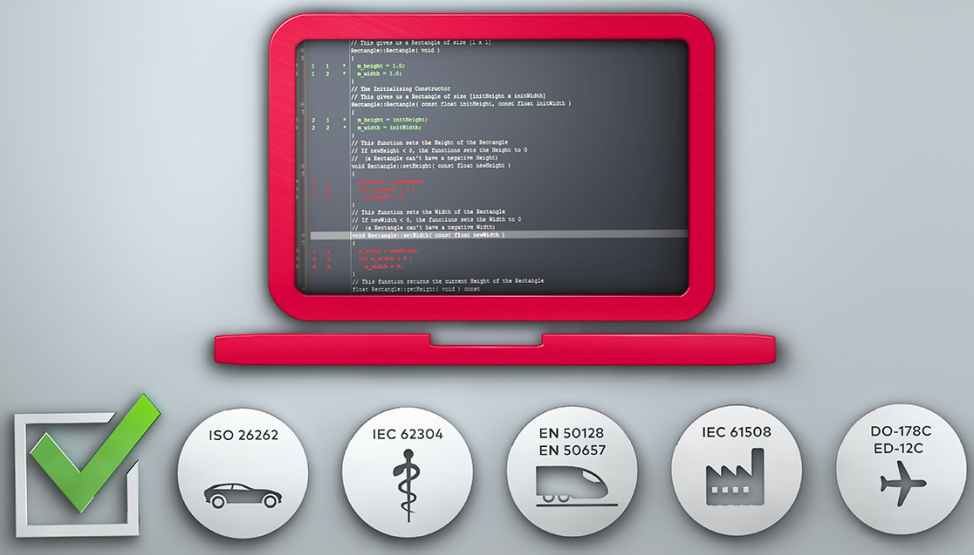
VectorCAST/QA integrates with your build system and existing test infrastructure to collect key metrics such as code complexity, test case status, and code coverage data.

No changes to your existing work flow or tools are required. As your normal system testing activities take place, a data repository is constructed. This repository is queried to provide answers to questions such as "What tests do I need to run for this set of code changes?"



VectorCAST/QA provides the following capabilities:

- **Test Collaboration** - VectorCAST/QA allows users to easily run all flavors of test without needing to learn new tools or processes. Connectors are configured once and then leveraged by the entire team.
- **Integrated Code Coverage** - VectorCAST/QA automates the capture and maintenance of code coverage data during testing, allowing users to quickly identify untested portions of the application and determine resources needed to improve testing thoroughness.
- **Change-Based Testing** - Using the data gathered from the build system and from monitoring system test activities, VectorCAST/QA identifies correlations between tests and code. As the code changes, it automatically computes the minimum set of tests required to provide complete testing of the change.
- **Change Impact Analysis** - Change Impact Analysis can be used to identify the impact of a set of source code changes on the quantity of testing required. This provides developers with the ability to make better decisions when implementing their changes.
- **Test Case Maintenance** - Legacy test cases are often poorly documented and seldom evaluated for improvement as the application matures. VectorCAST/QA provides visibility into the parts of the application that each test stimulates, allowing you to gauge the value of each test and identify redundant tests.
- **Continuous Testing** - VectorCAST/QA integrates easily with Continuous Integration (CI) Servers such as Jenkins to allow tests to be distributed across a farm of physical or virtual test machines.



Work Flow Scenarios

Initial Setup

Before you begin:

These example Work Flows are run on a Windows platform and the following assumptions are made:
> You are running on Windows (note that Linux is also supported, but our examples are run on Windows).
> VectorCAST 2026 is installed.
> You are using the FlexLM license server.
> You are licensed for VectorCAST/QA Enterprise.
> You have source code located under one parent directory (note that multiple directories are supported, but the Work Flow examples assume only one directory).
> You are able to build the source code on Windows.
> You have test cases that run on Windows.

Using the Lua Example

The Work Flow examples use Lua source code. The Lua application is an open source project for a language interpreter for the Lua language. It is shipped with VectorCAST and everything that you need to run these examples is located in the VectorCAST installation directory under **`Examples\cover\lua_demo`**.

The Lua example already has all the source code under one parent directory, provides us with a way to build the source code on Windows, and also provides us with test cases that run on Windows.

`lua_demo` contains three main directories: **`doc`**, **`src`**, and **`tests`**, as well as a **`Makefile`** and a **`README`** file.

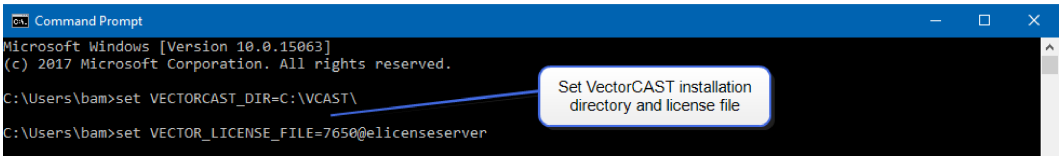
All source code is located in the **`src`** directory. The source is built with the command '**`make generic`**'. Create an empty directory. The directory path should not contain any blanks. The directory can be located anywhere, but in our example it is located in C:\ and named **`qa_example`**. The make target **`'generic'`** refers to the Lua platform target and works with our examples.

The test cases contained in the **`tests`** directory are Lua programs that test different parts of the Lua interpreter application and are part of the open source project. The tests are run by adding them to the Lua command line.

Preliminary Setup

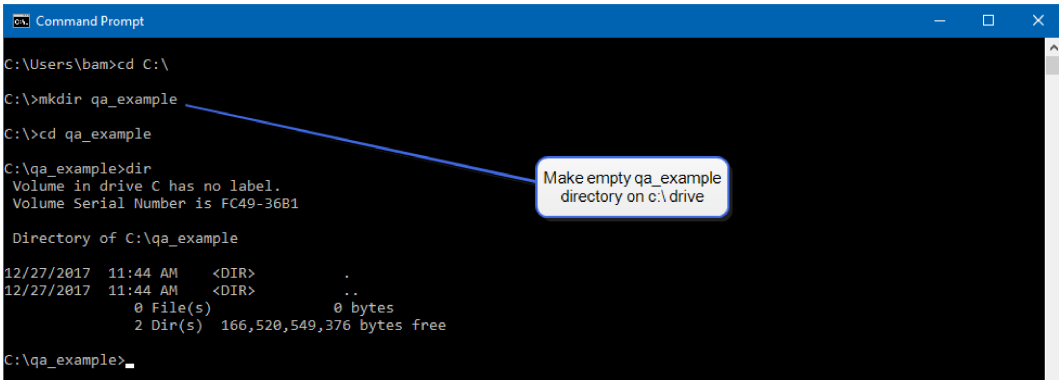
1. Open a Windows Command Prompt.
   Set VECTORCAST_DIR to your installation directory.
   Set VECTOR_LICENSE_FILE to your license file.

```
set VECTORCAST_DIR=<installation-directory>

set VECTOR_LICENSE_FILE=<license-file>
```



2. Create an empty directory. The directory path should not contain any blanks. The directory can be located anywhere, but in our example it is located in C:\ and named **qa_example**. Open your newly created directory. In our example we change to **qa_example**.

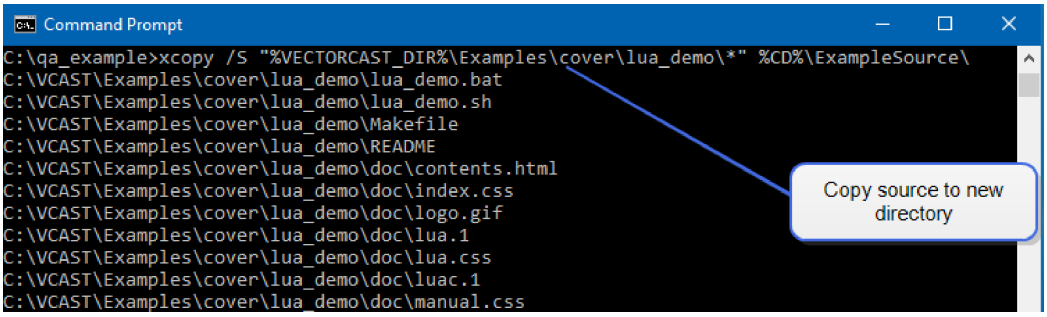
```
cd c:\

mkdir qa_example

cd qa_example
```



3. For our example, we will use the Lua application source located in the VectorCAST installation directory under **Examples\cover\lua_demo**.

   Copy the **lua_demo** directory to the newly created **qa_example** directory. Note that you must name the directory **ExampleSource**, since our script will look for that name.

```
xcopy /S "%VECTORCAST_DIR%\Examples\cover\lua_demo\*"
%CD%\ExampleSource\
```



   We now have an **ExampleSource** directory containing three subdirectories: **doc, src**, and **tests**. The **src** directory contains all the source code for the example. The **tests** directory

contains the test scripts to test Lua. The application is built using the **`Makefile`** found in the **`ExampleSource`** directory.



Now the preliminary setup is complete and we can start VectorCAST.

Launch VectorCAST/QA

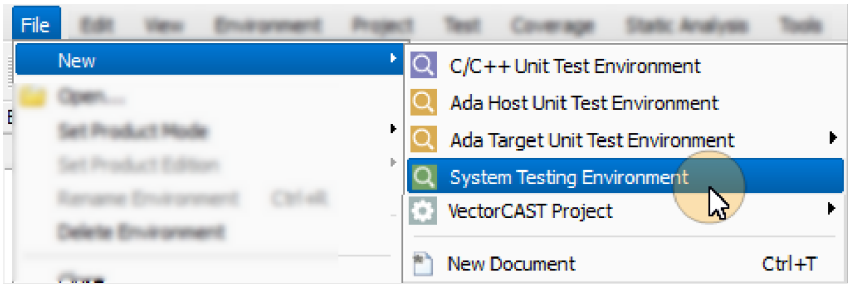
1. Use the following command to start VectorCAST/QA.

```
c:\qa_example>%VECTORCAST_DIR%\vcastqt
```

Confirm that your working directory is your newly created source directory. In our example, our working directory is **`C:/qa_example`**. If it is not correct, set it to the correct directory by selecting **File => Set Working Directory...** from the Menu Bar.

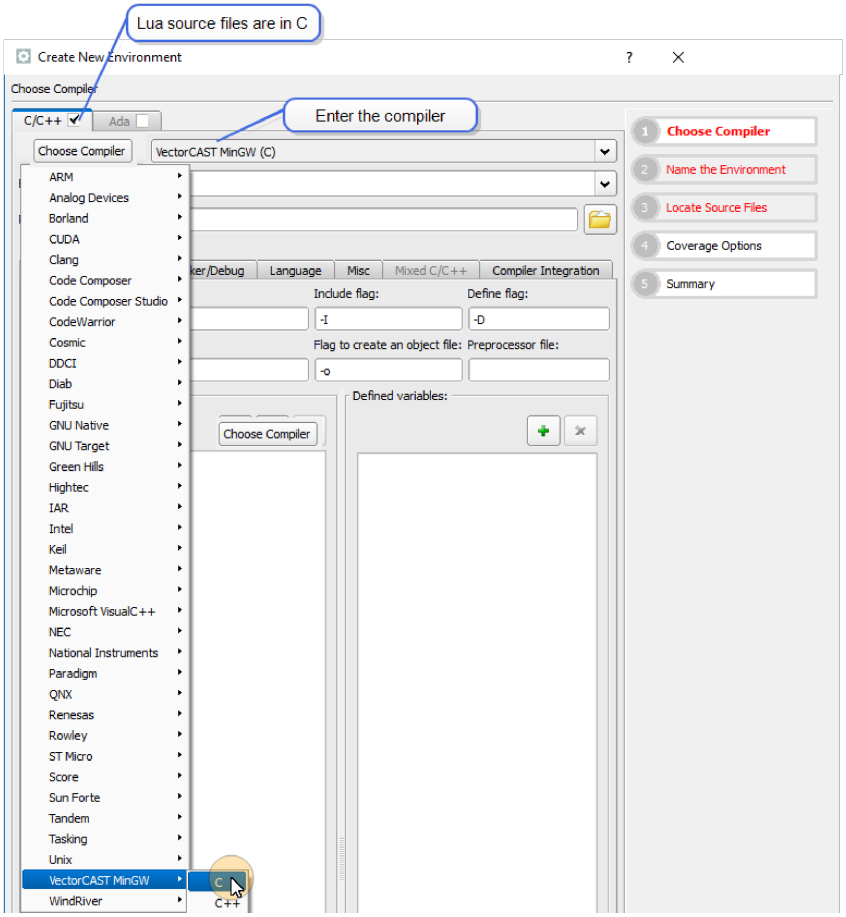
Create a New System Testing Environment

1. From the Menu Bar, select **File => New => System Testing Environment**.

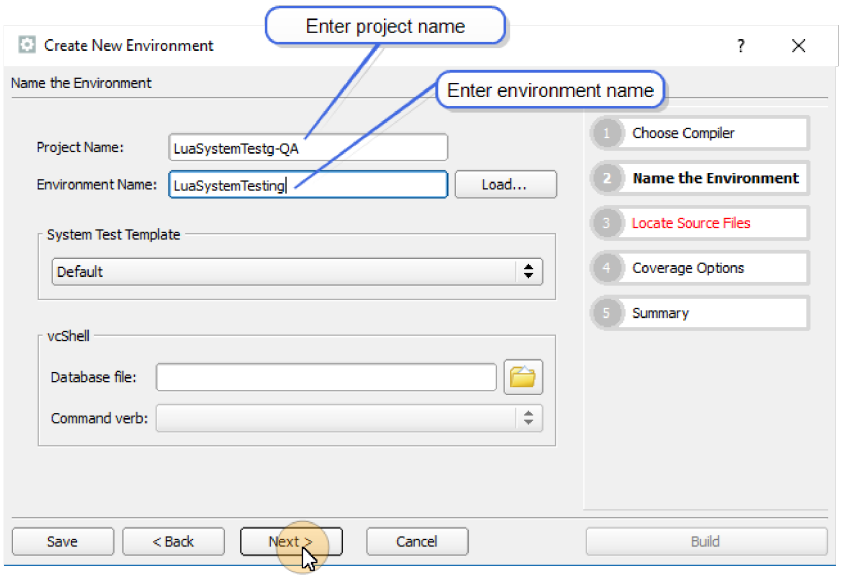


2. The Create New Environment dialog opens on the Choose Compiler pane. Select the compiler **VectorCAST MinGW => C** using the **Choose Compiler** drop-down menu. Verify that the C/C++ box is checked for the language of the source files, and that the Ada box is unchecked.

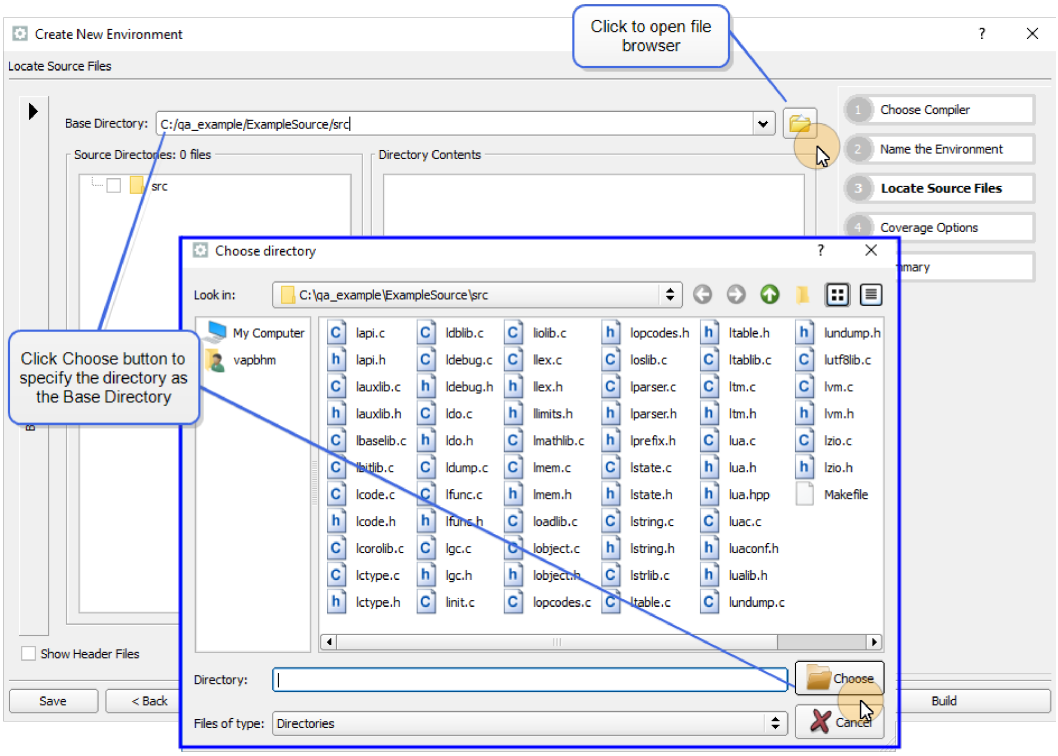
Click the **Next** button.



3. The Create New Environment dialog opens on the Name the Environment pane. Enter **`LuaSystemTesting-QA`** for the Project Name. Enter **`LuaSystemTesting`** for the name of the environment. Select the **Next** button.



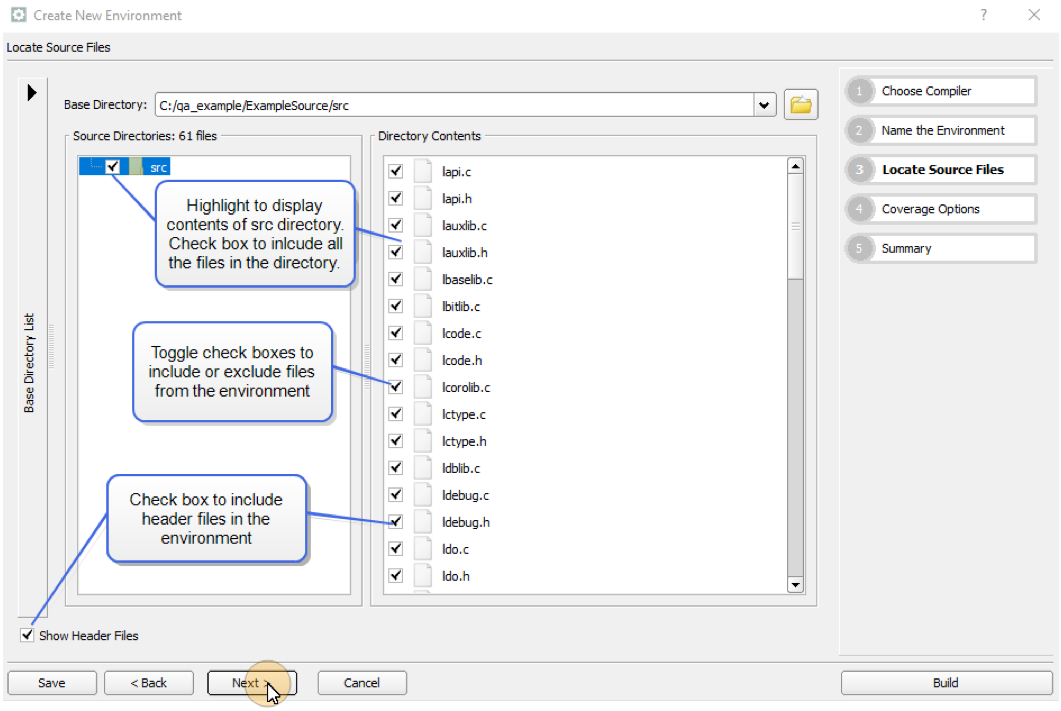
4. Locate the source files and set the Base Directory by clicking the file browser button to the right of the Base Directory field. Use the file browser to navigate to the location of our source directory at `C:\qa_example\ExampleSource\src`. Click the **Choose** button.



5. Highlight the source directory to display the contents (files and folders) of the source directory in the Directory Contents pane on the right. We will be including our header files in our environment, so we also must check the **Show Header Files** box at the lower left corner of the window. All of the source files are listed in the Directory Contents pane on the right, and all are unchecked.

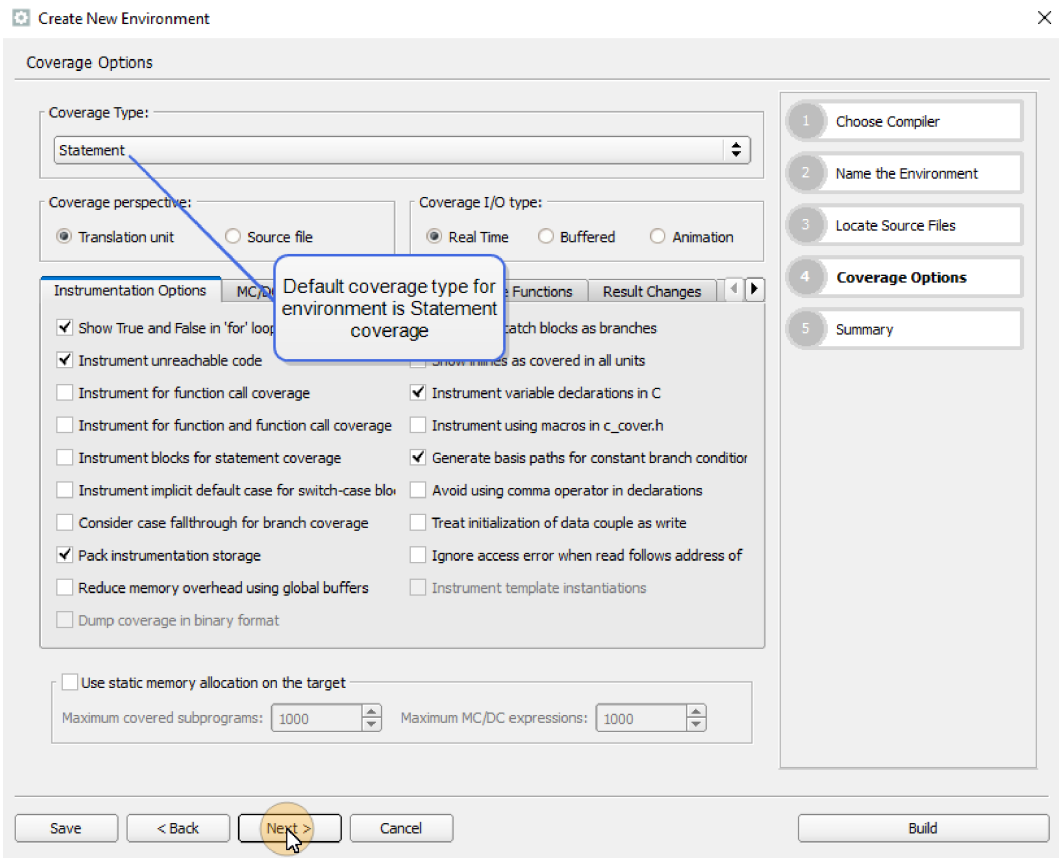
   Check the box by the `src` directory to include all the files in the directory. By default, all of the files are checked as included.

   Uncheck source files in the Directory Contents pane to exclude them from the Cover environment. For our example, we will select all 61 files. Click the **Next** button.

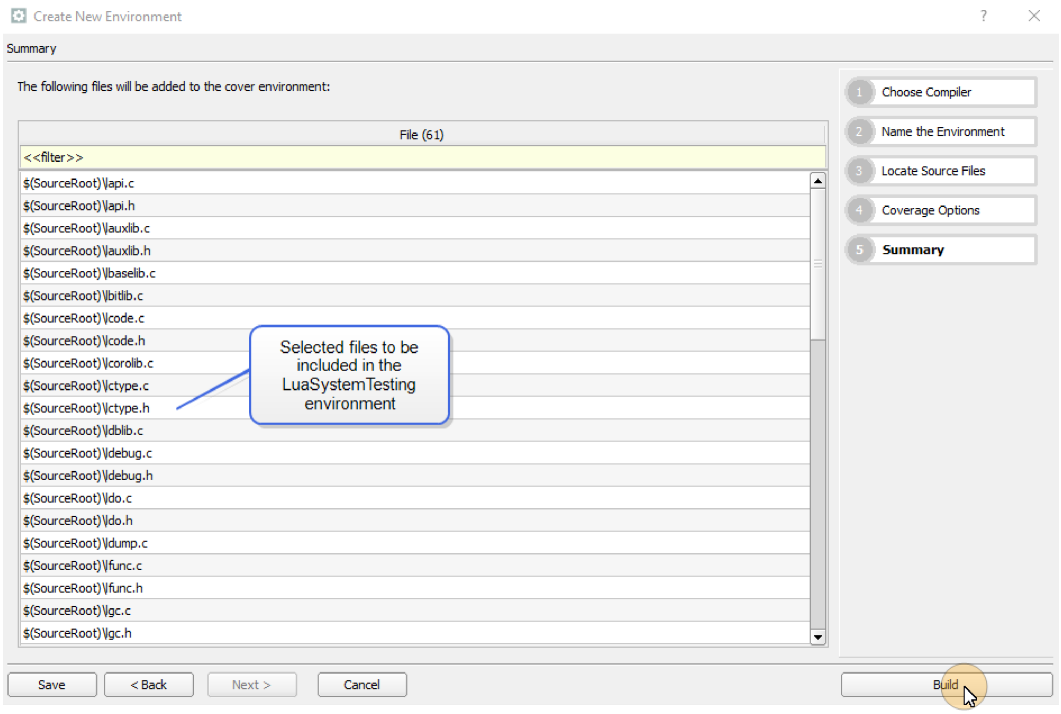


Tip: We recommend that you include source files by checking directories. Doing so makes it easier to sync your environment with your source repository.

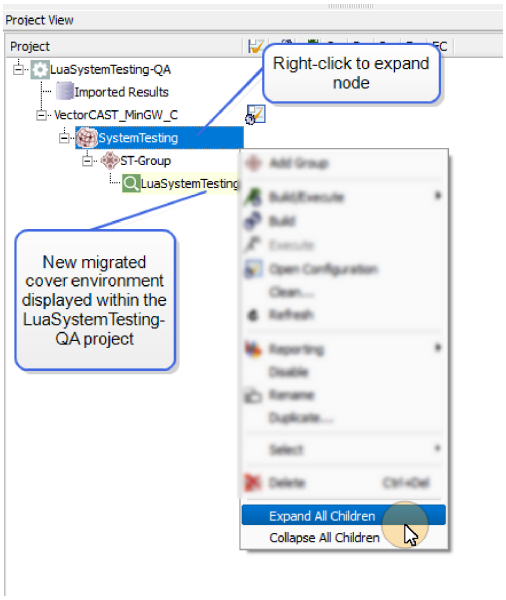
6. The Coverage Options pane opens. Select the Coverage Type to be set for the environment after building it. We will use Statement coverage, which is the default coverage type. Click the **Next** button to view the environment Summary.



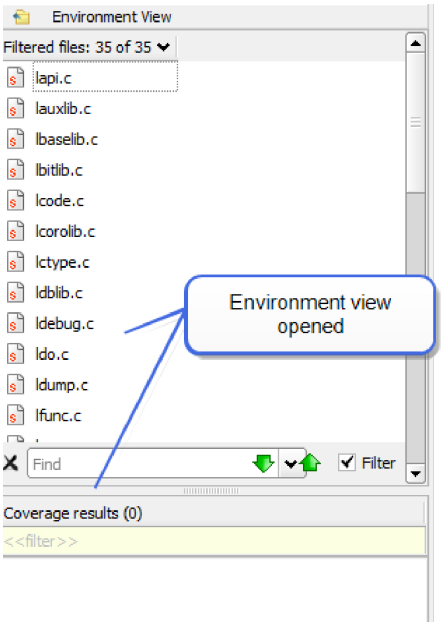
7. The Summary page shows the 61 files that we have selected to be added to our environment. Click the **Build** button to build the environment.



8. Our newly created Cover environment, **`LuaSystemTesting`**, is displayed in the Environment View pane. If it is not, in the Project Tree located under the Project View, right-click on the SystemTesting node and select **Expand All Children**.



9. If the Environment View panes are not already open, double-click the Environment node to open the environment in the Environment View panes.



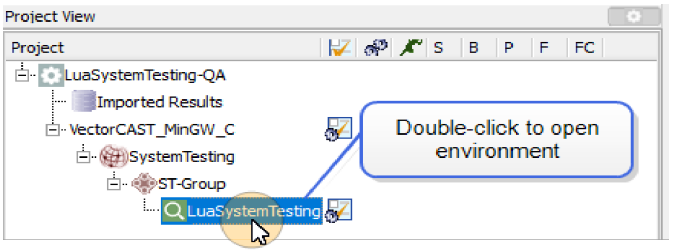
10. Close the environment by selecting **File => Close Environment** from the Menu Bar.

Instrument For Coverage

In this Work Flow, we will instrument our LuaSystemTesting environment for Statement coverage.

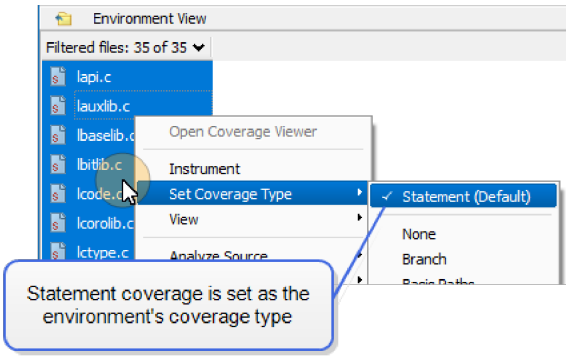
Open the Environment

1. In the Project Tree, select the **`LuaSystemTesting`** environment and double-click. (If you do not see the environment, right-click on the **SystemTesting** node and select **Expand All Children**).



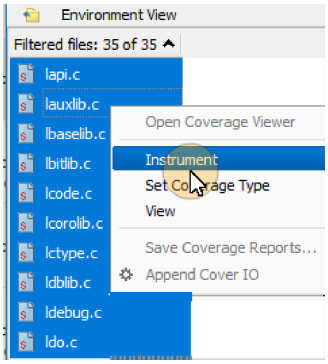
Set Coverage Type

1. In the Environment View, select ALL of the source files. Right click anywhere in the source file list and select **Set Coverage Type** from the context menu. Note that Statement coverage, which we selected when the environment was originally created, is already selected as the default coverage type for our environment.



Instrument for Coverage

1. In the Environment View, select ALL of the source files. Right click anywhere in the source file list and select **Instrument** from the context menu.



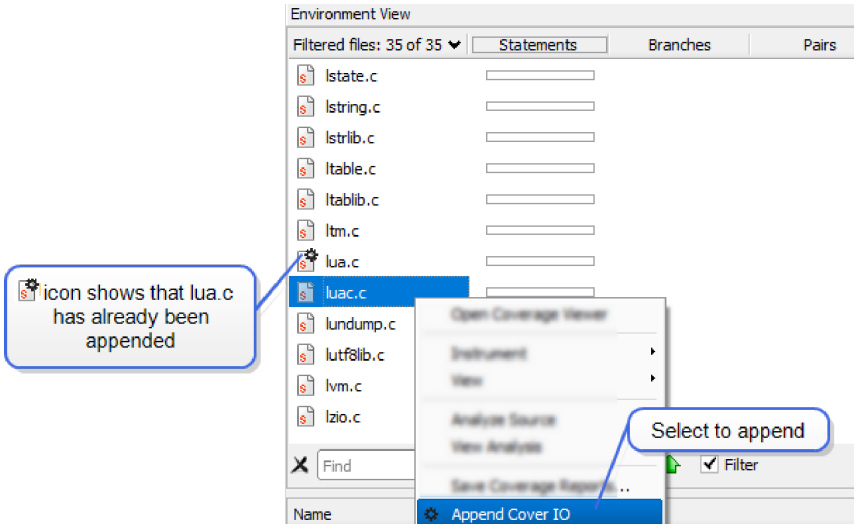
2. You will see a message in the Message window as the files are instrumented for Statement coverage.



Append Cover IO

Some files should have the **`c_cover_io.c`** file appended during instrumentation. By doing so we eliminate the need to add the **`<cover env>/c_cover_io.c`** file on the command line when compiling the instrumented sources. Typically, only one unit per Cover environment is selected in order to avoid redefinition errors during compilation, and it is usually the one with the **`main()`** function. In our example, we are selecting two source files, **`lua.c`** and **`luac.c`**, to append.

1. In the Environment view, right click on the source file **`lua.c`** and select **Append Cover IO** from the context menu.

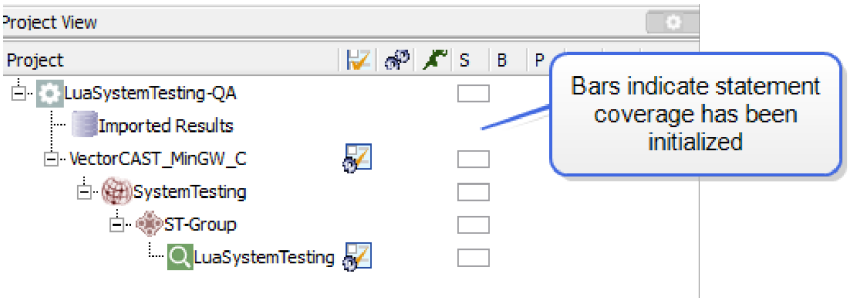


2. The source file will be immediately reinstrumented. Repeat the step for the source file **`luac.c`**

Close the Environment

1. Close the **`LuaSystemTesting`** coverage environment and return to the System Testing project by selecting **File => Close Environment** from the Menu Bar. You will be prompted to save your changes to the VectorCAST Project. Select the **Save** button.

   Notice that the status panel now indicates that Statement coverage has been initialized.

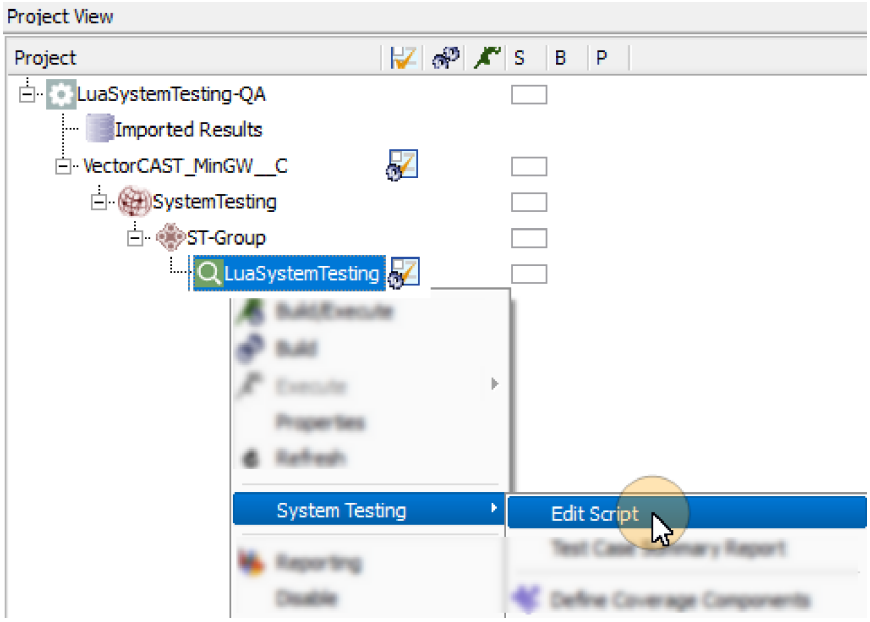


Edit Testing Script and Build

Edit the systems_tests.py script

To run the full test suite for our example, we must first update the testing script (**`system_tests.py`**). Each time a new system test environment is created, its own **`<environment name>_system_tests.py`** script is created, which defines its own build and execution commands.

1. Right-click on the **`LuaSystemTesting`** Environment node in the Project Tree and select **System Testing => Edit Script** from the context menu.



2. For our example, the **`LuaSystemTesting_system_tests.py`** script is created and opens in the Text Editor. The configuration script is fully customizable and allows you to define the application build commands, list of test cases, and test execution commands for an application.

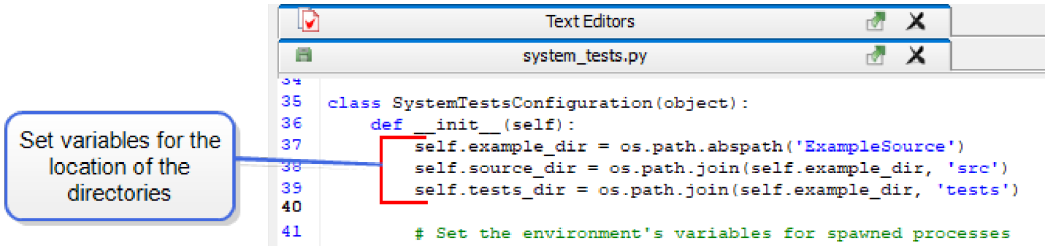
   At a minimum, VectorCAST/QA requires the following to be updated:

   > The directory location where we run the make command (**`self.locationWhereWeRunMake`**)
   > The top level make command (**`self.topLevelMakeCommand`**)
   > The directory location where we run the tests (**`self.locationWhereWeRunTests`**)
   > The name of the test executable (**`self.nameOfTestExecutable`**)
   > The test case names (**`self.masterListOfTestCases`**)
   > The command used to run the tests (implementation for function **`commandToRunATest`**)
   > The interpretation of the test results (implementation for the function **`interpretTestResults`**)

   The following steps will discuss each modification to the script required for our example.

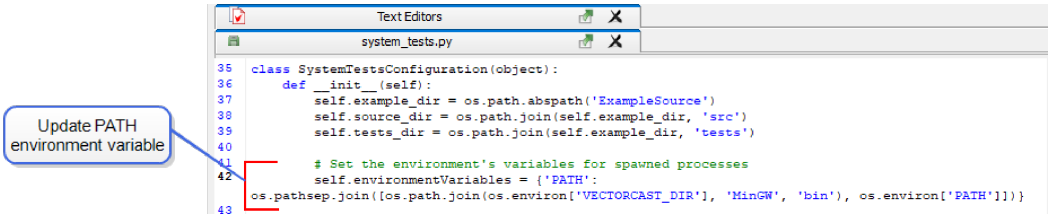
3. First, for our example, three variables are created to help find the location of the main **`ExampleSource`** directory, the source code directory (**`src`**), and the test scripts directory

(**tests**).



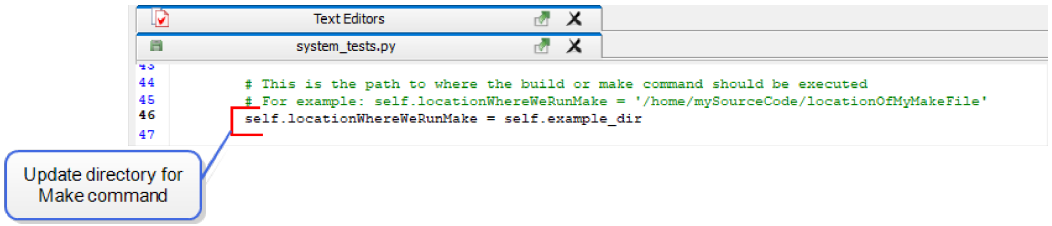

```
self.example_dir = os.path.abspath('ExampleSource')
self.source_dir = os.path.join(self.example_dir, 'src')
self.tests_dir = os.path.join(self.example_dir, 'tests')
```

4. Next, the PATH environment variable is updated to point to the compiler used in our example.



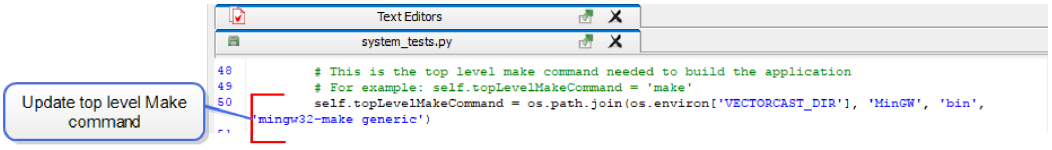

```
self.environmentVariables = {'PATH': os.pathsep.join([os.path.join
(os.environ['VECTORCAST_DIR'], 'MinGW', 'bin'), os.environ
['PATH']])}
```

5. Update the directory location where we run the Make command.



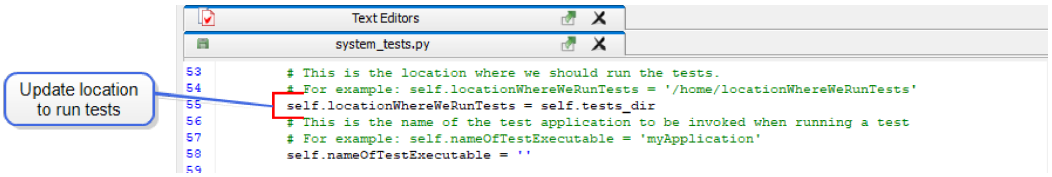

```
self.locationWhereWeRunMake = self.example_dir
```

6. Update the top level Make command.



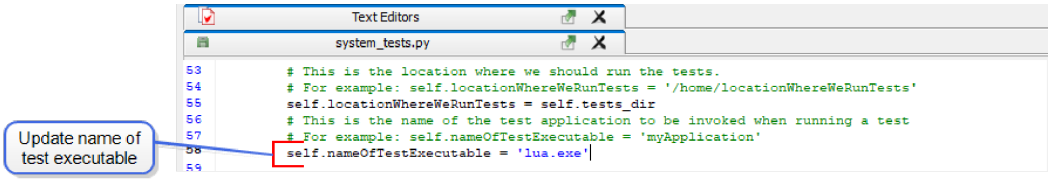
```
self.topLevelMakeCommand = os.path.join(os.environ['VECTORCAST_
DIR'], 'MinGW', 'bin', 'mingw32-make generic')
```

7. Update the directory location where we run the tests.



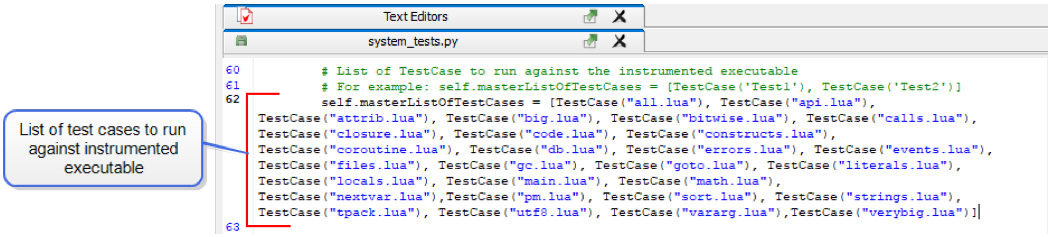

```
self.locationWhereWeRunTests = self.tests_dir
```

8. Update the name of the test executable.



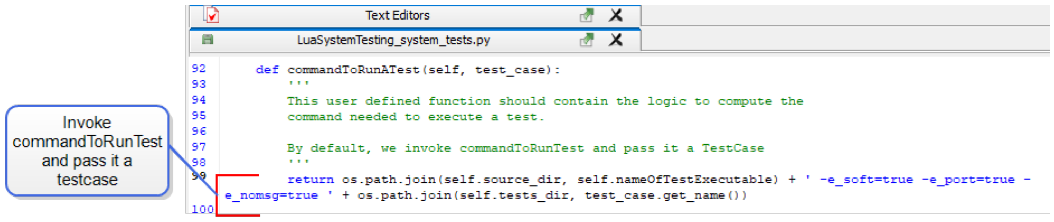

```
self.nameOfTestExecutable = 'lua.exe'
```

9. List the names of the test cases.




```
self.masterListOfTestCases = [TestCase("all.lua"), TestCase
("api.lua"), TestCase("attrib.lua"), TestCase("big.lua"), TestCase
("bitwise.lua"), TestCase("calls.lua"), TestCase("closure.lua"),
TestCase("code.lua"), TestCase("constructs.lua"), TestCase
("coroutine.lua"), TestCase("db.lua"), TestCase("errors.lua"),
TestCase("events.lua"), TestCase("files.lua"), TestCase("gc.lua"),
TestCase("goto.lua"), TestCase("literals.lua"), TestCase
("locals.lua"), TestCase("main.lua"), TestCase("math.lua"),
TestCase("nextvar.lua"),TestCase("pm.lua"), TestCase("sort.lua"),
TestCase("strings.lua"), TestCase("tpack.lua"), TestCase
("utf8.lua"), TestCase("vararg.lua"),TestCase("verybig.lua")]
```

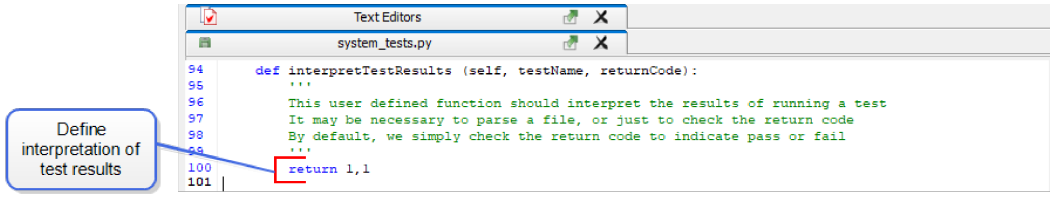
10. Enter the command used to run the tests.




```
return os.path.join(self.source_dir, self.nameOfTestExecutable) +
' -e_soft=true -e_port=true -e_nomsg=true ' + os.path.join
(self.tests_dir, test_case.get_name())
```

11. Next we define the interpretation of the test results. The function uses **`2-tuple`**, where the first element represents the number of passed expecteds and the second element represents the total expecteds. These values are displayed in the "Expecteds" column in the Test Case Summary Report. (To access the Test Case Summary Report, right-click on an Environment node in the Project Tree and select **Reporting => Test Case Summary Report** from the context menu.)

    For our example, the tests do not provide any easily interpreted return values, so the tuple is simply set to 1,1 (meaning one expected value is passed out of one for all of the tests).




```
return 1,1
```

12. Save your changes to the **`LuaSystemTesting_system_tests.py`** file by selecting **File => Save** from the Menu Bar. Alternatively, select the **Save** button on the Toolbar.

Build the Instrumented Code

1. To build our environment, select the **`LuaSystemTesting`** environment node and right-click. Select **Build/Execute => Incremental** from the context menu.
2. The Manage Status viewer opens automatically showing the build progress. At the completion of the build, the Manage Incremental Rebuild Report is displayed.

Manage Incremental Rebuild

Environments Affected

Environment	Rebuild Status	Unaffected Tests	Affected Tests	Total Tests
VectorCAST_MinGW_C / SystemTesting / LuaSystemTesting	Full Rebuild Succeeded	0	0	0
Totals	**1 / 1 (100%)**	**0**	**0**	**0**

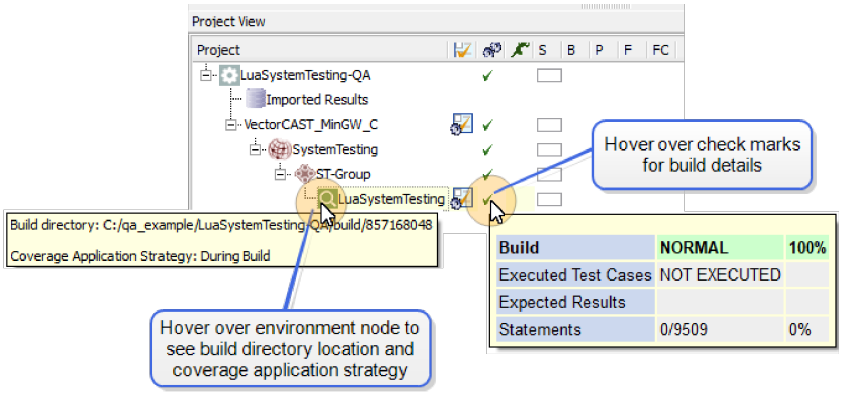
Files and Functions Affected

Environment	Changed Files	Changed Functions	Global Scope Change
VectorCAST_MinGW_C / SystemTesting / LuaSystemTesting	None	None	No

Test Cases Affected

Environment	Affected Tests	Test Case Type	Affected Test List	Reason
VectorCAST_MinGW_C / SystemTesting / LuaSystemTesting	0 / 0 (0%)		None	

Also note that when the build is complete the Build Status column in the Project Tree now shows green check marks indicating successful build status. Hover over the check marks to see popups with the build details. Hover over the environment node to see the location of the build directory and the coverage application strategy.

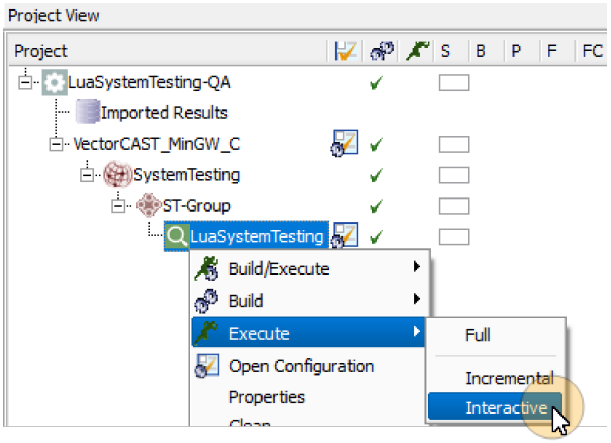


Run a Test Interactively

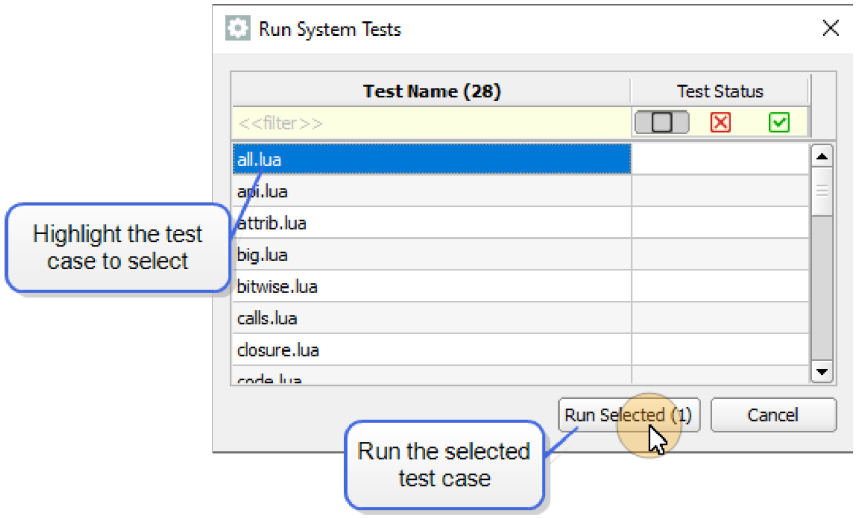
VectorCAST provides the ability to manually select a single test or a set of tests for execution in a System Test environment. Tests can be selected by Test Name and Test Status, allowing the user to run all or any subset of tests for the environment.

Run a Single Test

1. For our example, we will select a single test to run interactively. Right-click on the **`LuaSystemTesting`** environment in the Project Tree and select **Execute => Interactive** from the context menu.

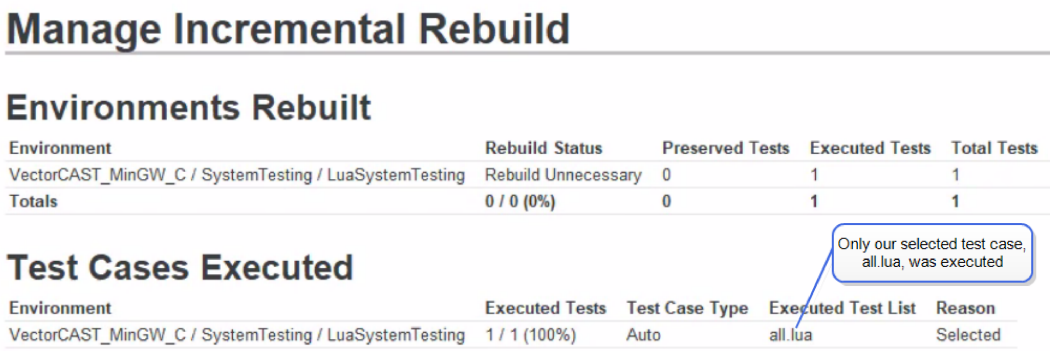


2. The Run System Tests dialog opens. We will run a single test case, **`all.lua`**. Highlight the **`all.lua`** test case located in the Test Name column and select the **Run Selected** button.



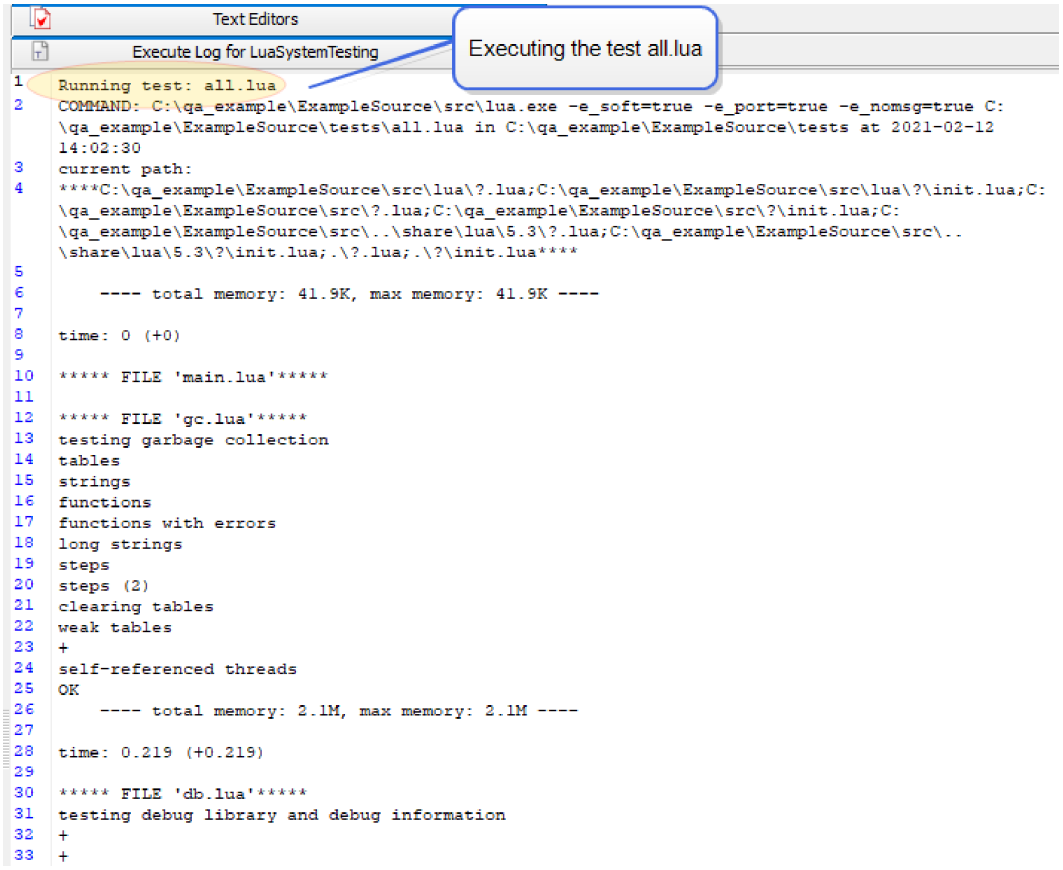
3. The Manage Incremental Rebuild Report is displayed. Note that only our manually selected test

case **`all.lua`** was executed.



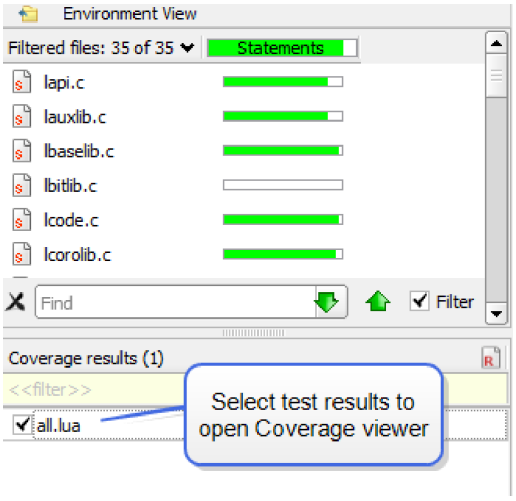
View the Execute Log

1. To view the Execute Log for **`LuaSystemTesting`**, right-click on the **`LuaSystemTesting`** environment in the Project Tree and select **View Execute Log** from the context menu.
2. The Execute Log opens in the Text Editor. The log indicates that only our selected test case, all.lua, is executed.

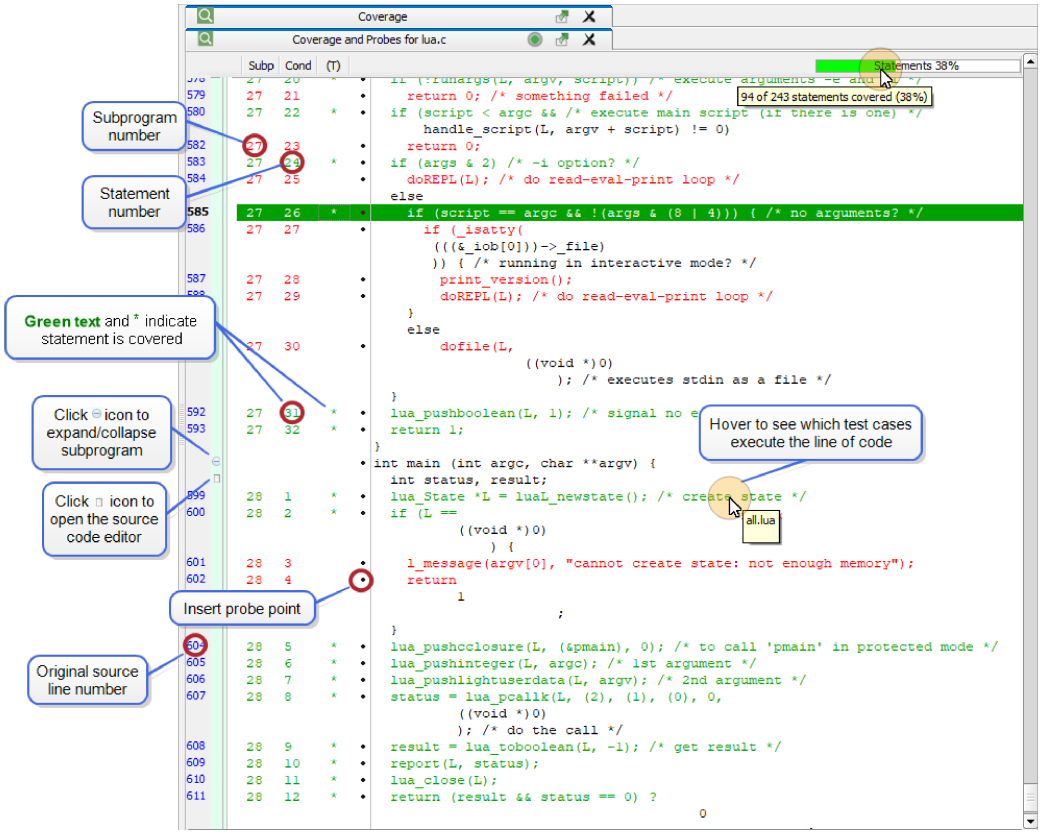


View Coverage Results

1. Next, we will view the Test Results for **`all.lua`**. Double-click on the **`LuaSystemTesting`** environment node in the Project Tree to open the environment. In the Environment View, check the box next to the **`all.lua`** test results.

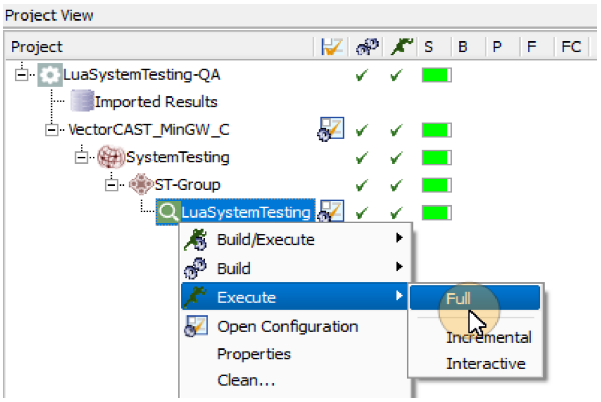


2. The Coverage Viewer opens in the MDI window. For more information on viewing coverage results, see "The Coverage Viewer" on page 126.

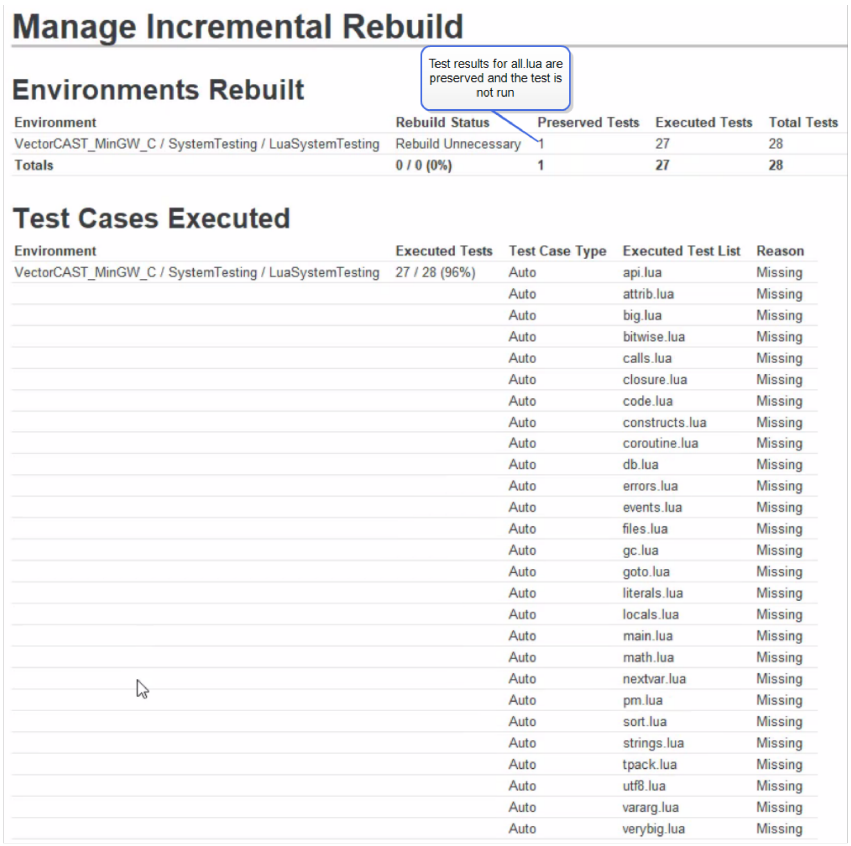


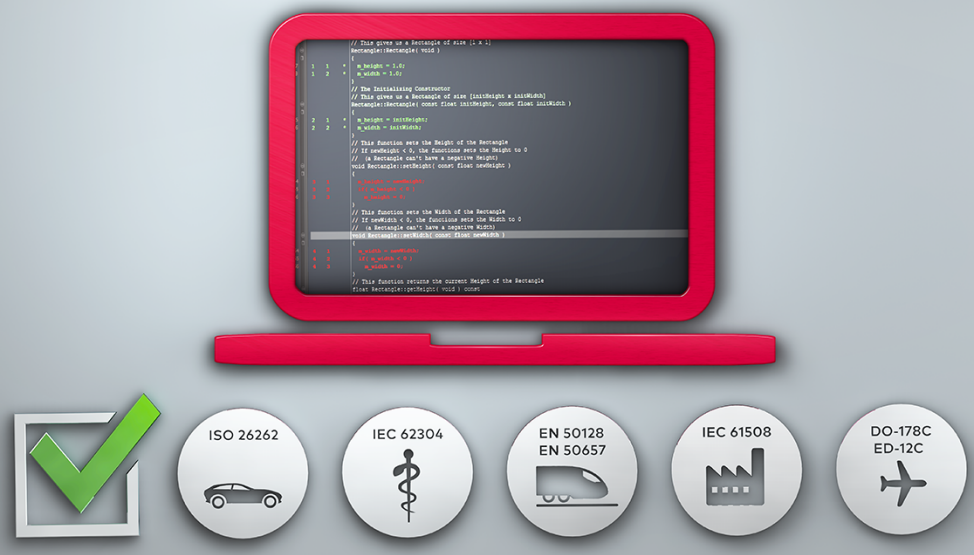
Execute All Tests

1. Finally, we will execute all our tests by right-clicking on the **`LuaSystemTesting`** environment node and selecting **Execute => Incremental.**



2. The Manage Incremental Build Report is displayed. Note that all our tests are run, but VectorCAST/QA Enterprise with Change-based testing preserves the previous result for **`all.lua`**, since no source code change occurred to affect that test.



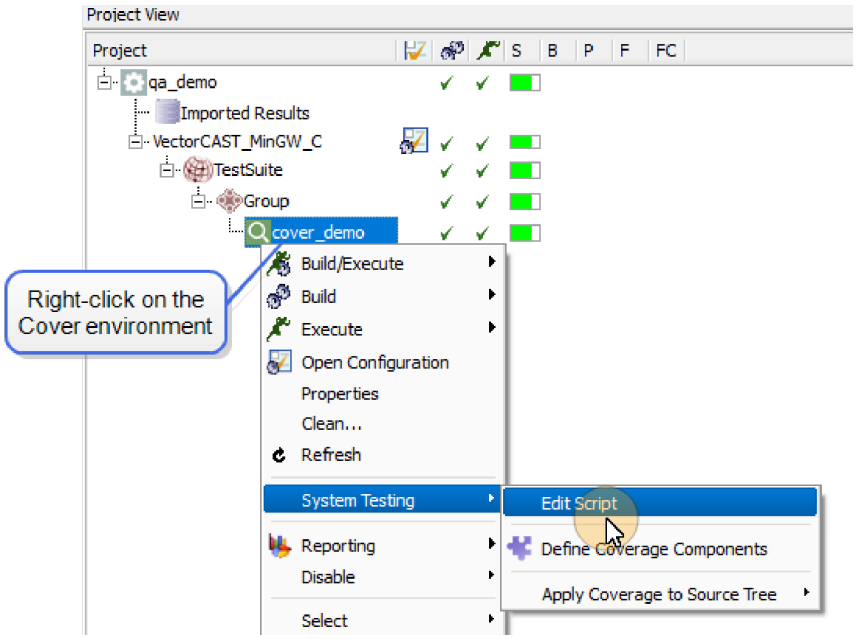


System Test Automation

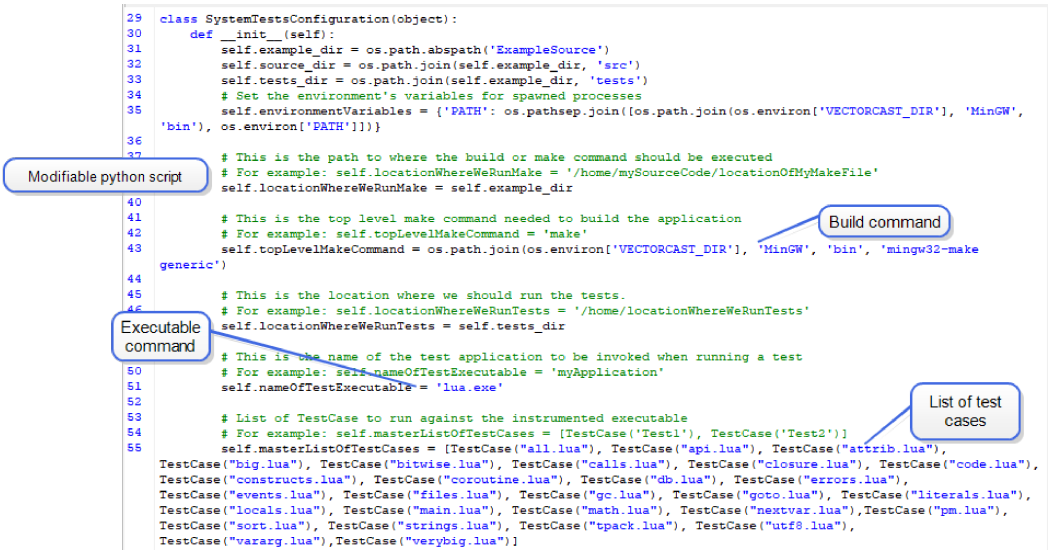
The Python Configuration Script

The Python Configuration Script contains functions that are invoked by VectorCAST to provide custom build and execute commands. Each time a new system test environment is created within a project, its own **`<environment name>_system_tests.py`** script is created, which defines its own build and execution commands. The configuration script is fully customizable and allows the user to define the application build commands, list of test cases, and test execution commands for an application.

To open and edit the Configuration Script, right-click on a Cover Environment node and select **System Testing => Edit Script** from the context menu.



The Python Configuration Script opens in the Script Editor. Modify the script as required and save changes.

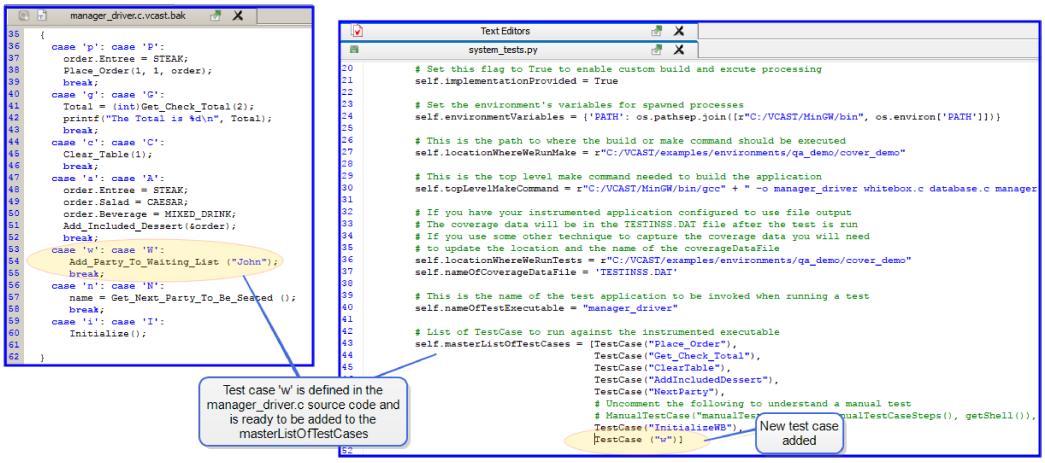



```
class SystemTestsConfiguration(object):
    def __init__(self):
        self.example_dir = os.path.abspath('ExampleSource')
        self.source_dir = os.path.join(self.example_dir, 'src')
        self.tests_dir = os.path.join(self.example_dir, 'tests')
        # Set the environment's variables for spawned processes
        self.environmentVariables = {'PATH': os.pathsep.join([os.path.join(os.environ['VECTORCAST_DIR'], 'MinGW',
'bin'), os.environ['PATH']])}

        # This is the path to where the build or make command should be executed
        # For example: self.locationWhereWeRunMake = '/home/mySourceCode/locationOfMyMakeFile'
        self.locationWhereWeRunMake = self.example_dir

        # This is the top level make command needed to build the application
        # For example: self.topLevelMakeCommand = 'make'
        self.topLevelMakeCommand = os.path.join(os.environ['VECTORCAST_DIR'], 'MinGW', 'bin', 'mingw32-make
generic')

        # This is the location where we should run the tests.
        # For example: self.locationWhereWeRunTests = '/home/locationWhereWeRunTests'
        self.locationWhereWeRunTests = self.tests_dir

        # This is the name of the test application to be invoked when running a test
        # For example: self.nameOfTestExecutable = 'myApplication'
        self.nameOfTestExecutable = 'lua.exe'

        # List of TestCase to run against the instrumented executable
        # For example: self.masterListOfTestCases = [TestCase('Test1'), TestCase('Test2')]
        self.masterListOfTestCases = [TestCase("all.lua"), TestCase("api.lua"), TestCase("attrib.lua"),
TestCase("big.lua"), TestCase("bitwise.lua"), TestCase("calls.lua"), TestCase("closure.lua"), TestCase("code.lua"),
TestCase("constructs.lua"), TestCase("coroutine.lua"), TestCase("db.lua"), TestCase("errors.lua"),
TestCase("events.lua"), TestCase("files.lua"), TestCase("gc.lua"), TestCase("goto.lua"), TestCase("literals.lua"),
TestCase("locals.lua"), TestCase("main.lua"), TestCase("math.lua"), TestCase("nextvar.lua"),TestCase("pm.lua"),
TestCase("sort.lua"), TestCase("strings.lua"), TestCase("tpack.lua"), TestCase("utf8.lua"),
TestCase("vararg.lua"),TestCase("verybig.lua")]
```

Add a Test Case Using Python Configuration Script

To add a new test case that is already defined in the source code, first open the Python Configuration Script by right-clicking on the environment node and selecting **System Testing => Edit Script** from the context menu.

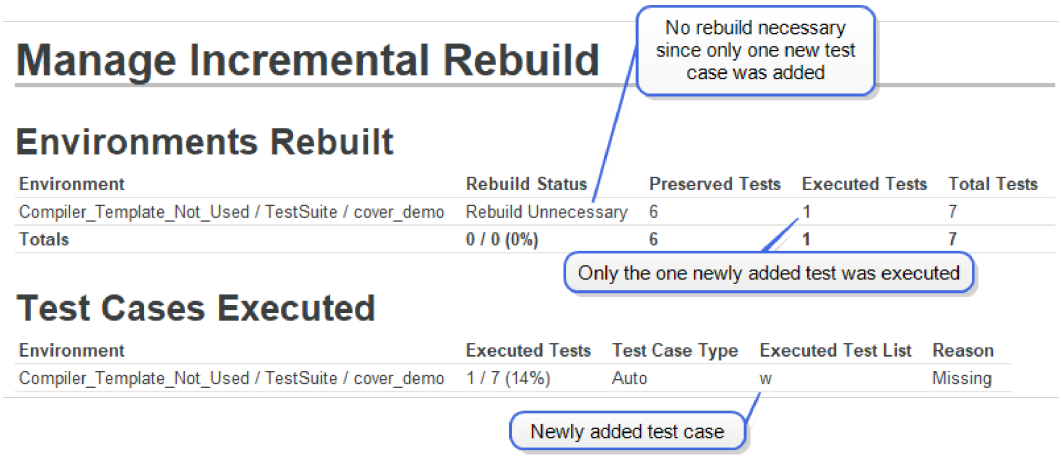
The configuration script opens in the Script Editor. Scroll down to the list of system test case names and add the new test name to the end of the Python list. In our example, we add the test case "w".




```
{
  case 'p': case 'P':
    order.Entree = STEAK;
    Place_Order(1, 1, order);
    break;
  case 'g': case 'G':
    Total = (int)Get_Check_Total(2);
    printf("The Total is %d\n", Total);
    break;
  case 'c': case 'C':
    Clear_Table(1);
    break;
  case 'a': case 'A':
    order.Entree = STEAK;
    order.Salad = CAESAR;
    order.Beverage = MIXED_DRINK;
    Add_Included_Dessert(&order);
    break;
  case 'w': case 'W':
    Add_Party_To_Waiting_List ("John");
    break;
  case 'n': case 'N':
    name = Get_Next_Party_To_Be_Seated ();
    break;
  case 'i': case 'I':
    Initialize();

}
```

```
        # Set this flag to True to enable custom build and excute processing
        self.implementationProvided = True

        # Set the environment's variables for spawned processes
        self.environmentVariables = {'PATH': os.pathsep.join([r"C:/VCAST/MinGW/bin", os.environ['PATH']])}

        # This is the path to where the build or make command should be executed
        self.locationWhereWeRunMake = r"C:/VCAST/examples/environments/qa_demo/cover_demo"

        # This is the top level make command needed to build the application
        self.topLevelMakeCommand = r"C:/VCAST/MinGW/bin/gcc" + " -o manager_driver whitebox.c database.c manager

        # If you have your instrumented application configured to use file output
        # The coverage data will be in the TESTINSS.DAT file after the test is run
        # If you use some other technique to capture the coverage data you will need
        # to update the location and the name of the coverageDataFile
        self.locationWhereWeRunTests = r"C:/VCAST/examples/environments/qa_demo/cover_demo"
        self.nameOfCoverageDataFile = 'TESTINSS.DAT'

        # This is the name of the test application to be invoked when running a test
        self.nameOfTestExecutable = "manager_driver"

        # List of TestCase to run against the instrumented executable
        self.masterListOfTestCases = [TestCase("Place_Order"),
                                      TestCase("Get_Check_Total"),
                                      TestCase("ClearTable"),
                                      TestCase("AddIncludedDessert"),
                                      TestCase("NextParty"),
                                      # Uncomment the following to understand a manual test
                                      # ManualTestCase("manualTes                ualTestCaseSteps(), getShell()),
                                      TestCase("InitializeWB"),
                                      TestCase ("w")]
```

Save the script changes. Right-click on the environment node and select **Execute => Full** from the context menu. VectorCAST will execute only the newly added test.



Add a Manual Test

VectorCAST/QA supports manual tests in addition to automated test cases. Adding the test case class **`ManualTestCase`** to the Python Configuration Script invokes a dialog allowing the user to implement a manual test. Manual test results are added to the Cover environment and included in the total number of tests.

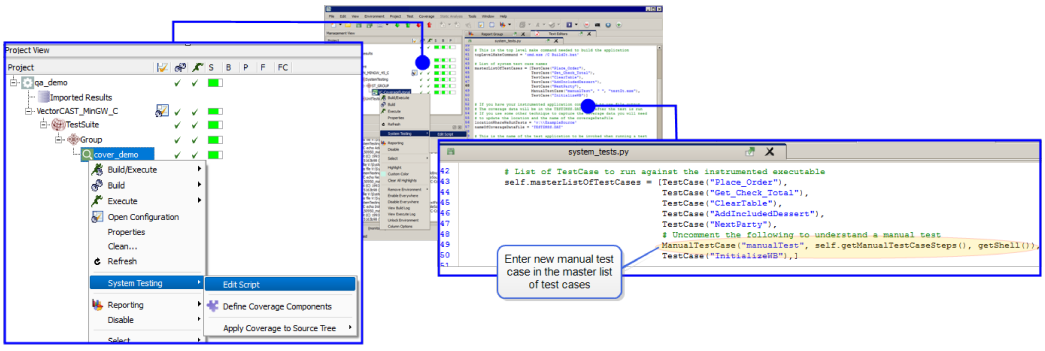
A manual test case is implemented by right-clicking on the environment node and selecting **System Testing => Edit Script** from the context menu.

The Python Configuration Script opens in the Script Editor. Scroll down to the list of system test case names and add the manual test entry to the Python list using the following syntax:

```
ManualTestCase('<TestName>','[<TestSteps>]','<NameOfTestExecutable>')

where the parameters required for the ManualTestCase class are <TestName>
and <NameOfTestExecutable>. The parameter <TestSteps> is optional.
```

For example:

```
ManualTestCase ("manualTest", self.getManualTestCaseSteps(), getShell())
```



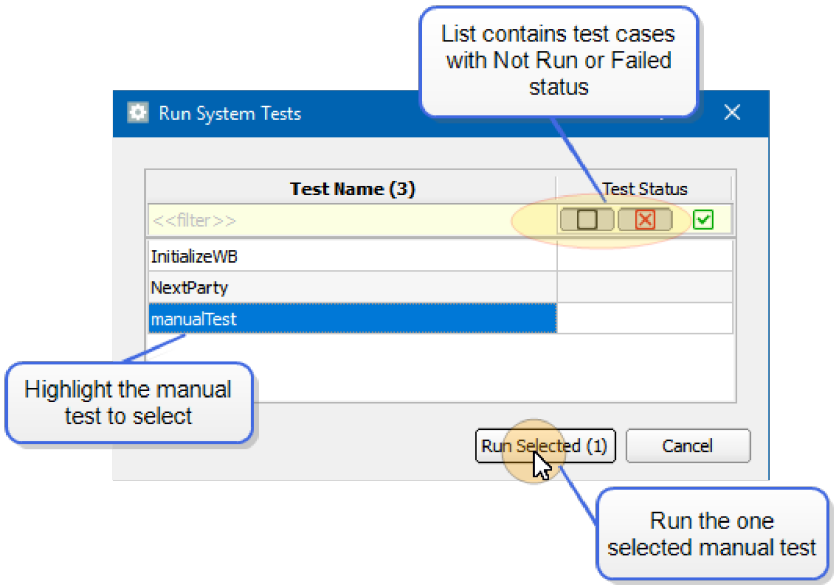
On Windows platforms, test case names are case insensitive. To avoid unexpected results, be sure to give each test case a unique name that does not rely on case to differentiate. For example, use

`manualTest1, manualTest2`, instead of **`ManualTest, manualTest`**.

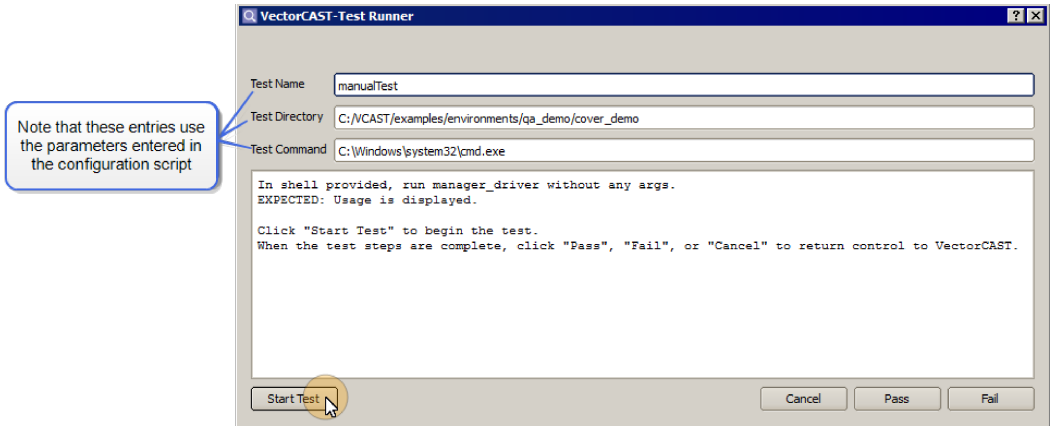
Save the script changes. To run a manual test case, right-click on the environment node and select **Execute => Interactive** from the context menu. The Run System Tests dialog opens.

You can filter the list by entering the Test Name. You can also refine the list by enabling the Test Status (**Not Run**, **Failed**, or **Passed**). The selection of multiple statuses is supported. By default, the Failed and Not Run test statuses are enabled.

For our example, we will run only the **`manualTest`** test case. Our test case has not been run, so it appears in the listing because the Not Run Test Status is enabled. To run the test, highlight the Test Name, **`manualTest`**, and select the **Run Selected** button.

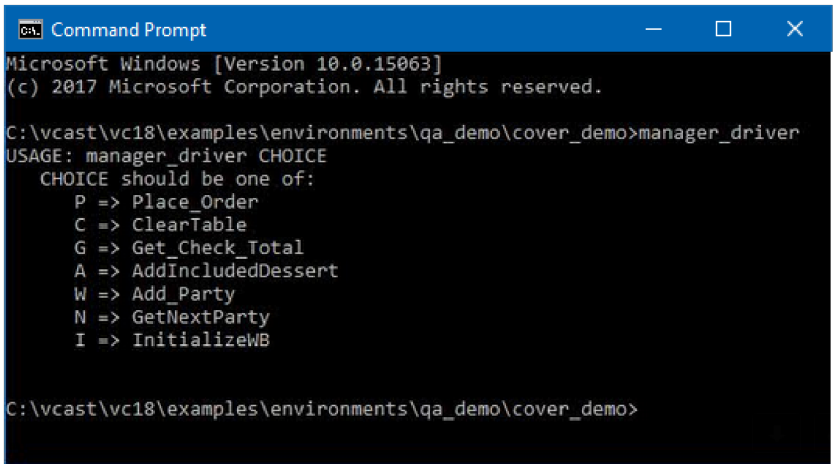


VectorCAST opens the Test Runner dialog for the manual test.



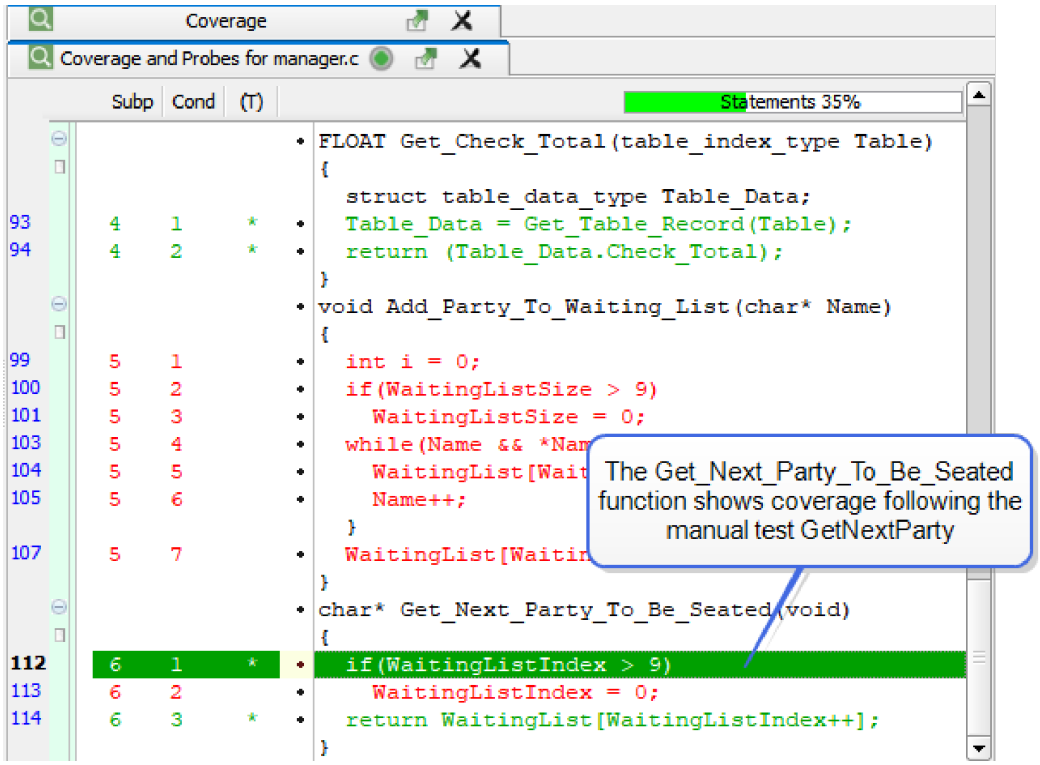
The fields displayed in the Test Runner dialog match the parameters passed to the ManualTestCase class. Click the **Start Test** button to run the test executable application.

A DOS command window opens displaying an input prompt. Select the test to be run. In our example, we enter "**N**" to run the **GetNextParty** test.



When the manual test is completed, record the results by clicking the appropriate **Pass** or **Fail** button on the Test Runner dialog. The dialog closes and the test results are added to the Cover environment. Clicking the **Cancel** button closes the dialog and does not add the test results to the total.

In our example we selected the **Pass** button at the conclusion of our manual test. Open the Coverage Viewer for **manager.c** and note that coverage is now shown for the **Get_Next_Party_To_Be_Seated** function.



Apply Coverage to Source Tree Option

This option provides full control of where instrumented files are stored and how those instrumented files are used by VectorCAST. This option defines three strategies for managing instrumented files:

- **Always** - In this mode, instrumented files are always stored in the Source Code Directory.
- **During Build** - In this mode, the instrumented files are copied during the build process.
- **Never** - In this mode, the instrumented files are never copied during the build process.

Benefits of the Apply Coverage to Source Tree Option

When determining which mode to select for managing instrumented files, consider the following:

Reasons for Using "During Build"

- This is the recommended and default option
- No need to worry where the instrumented file is located
- Minimum impact to SCM systems
- If the project is deleted, the instrumented files aren't left behind
- The same file can be instrumented in more than one project
- Source code can be changed without first un-instrumenting

Reasons for Using "Always"

- Debugging the instrumented executable
- Building from an IDE outside of VectorCAST

Reasons for Using "Never"

- Wanting to use VectorCAST/QA as a test runner
- Comparing test results with instrumented and un-instrumented code

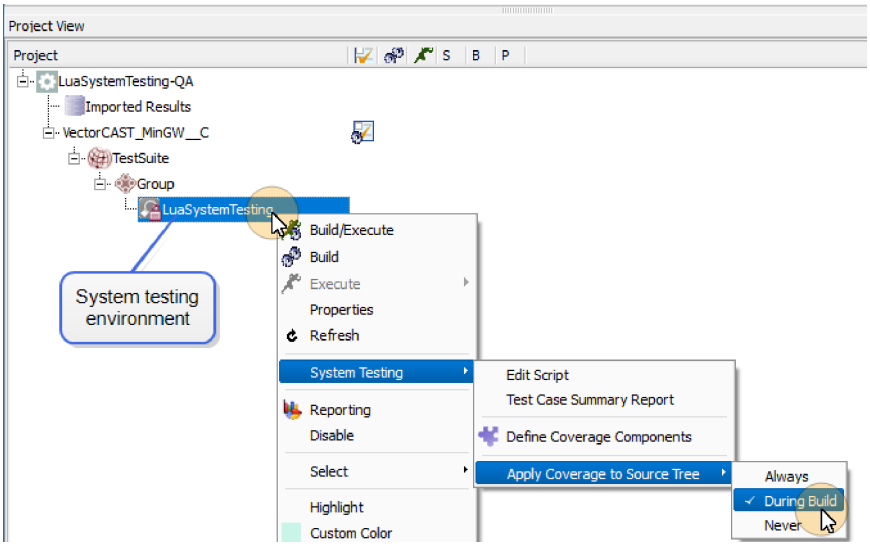
How the Apply Coverage to Source Tree Option Works

When a System Test Environment is created, the Source Directories and a separate Instrumentation Directory are defined. By default, the Instrumentation Directory is set to **`vc-inst`** and placed under the environment directory.

When VectorCAST is commanded to initialize coverage for a coverage type for a file or set of files, it reads the source code, instruments it for the coverage type requested and then stores the instrumented files.

The location where the instrumented files are stored is configurable using the Apply Coverage to Source Tree Option. There are three choices for this option: **Always**, **During Build** and **Never**. The default is **During Build**.

To set the option, right-click on the System Testing environment in the Project Tree, and from the context menu select **System Testing => Apply Coverage to Source Tree**, and then select the desired option.



The Manage command is:

```
manage -p <PROJECT> --level <Compiler/Test Suite> -e <ENVIRONMENT> --apply-
instrumentation=[BUILD|ALWAYS|NEVER]
```

Apply Instrumentation to Source Tree Using the "During Build" Option

With the **During Build** option, the instrumented files are stored in the Instrumentation Directory. When VectorCAST is commanded to perform a Build operation, the following process occurs:

1. The source code files are copied to backup files.
2. The instrumented files are then copied into the Source Code Directory.
3. The build is performed with the files in the Source Code Directory
4. When the build completes, the instrumented files are moved back to the Instrumentation Directory.
5. The source files are restored from the backup files.

Apply Instrumentation to Source Tree Using the "Always" Option

The **Always** option directs VectorCAST to always keep the instrumented files in the Source Code Directory. With the **Always** option, when VectorCAST is commanded to initialize coverage for a coverage type for a file or set of files, the following process occurs:

1. The source code file is copied to a backup file (**`.vcast.bak`**).
2. The source code file is read and instrumented for the coverage type requested.
3. The instrumented file is stored in the Source Code Directory.

When the instrumentation is complete, you will see that each instrumented source code file has an associated file with the file extension **`.vcast.bak`**. The file with the **`.vcast.bak`** extension is the un-instrumented file that was stored as backup. If VectorCAST is commanded to un-instrument the file, then the original file is restored from the backup.

Note: When migrating a monitored Cover environment, note that the option cannot be set to **Always**. This prevents having two Cover environments pointing to the same instrumented in-place source files.

Apply Instrumentation to Source Tree Using the "Never" Option

With the **Never** option, the instrumented files are stored in the Instrumentation Directory. When VectorCAST is commanded to perform a Build operation, the instrumented files are *not* copied into the Source Code Directory, and the build is performed with the un-instrumented files in the Source Code Directory.

Changing the Apply Coverage to Source Tree Option

The following table illustrates the actions which occur when changing options for applying coverage to the source tree:

Current State	To "Always"	To "During Build"	To "Never"
From "Always"		1. The instrumented files are moved to the Instrumentation Directory. 2. The source code files are restored from the backup files (.vcast.bak).	1. The instrumented files are moved to the Instrumentation Directory. 2. The source code files are restored from the backup files (.vcast.bak).
From "During Build"	1. The source code files are moved to the backup files (.vcast.bak). 2. The instrumented files are stored in the Source Code Directory.		**No action needed.**
From "Never"	1. The source code files are moved to the backup files (.vcast.bak). 2. The instrumented files are stored in the Source Code Directory.	**No action needed.**	

Incremental Build and Execute

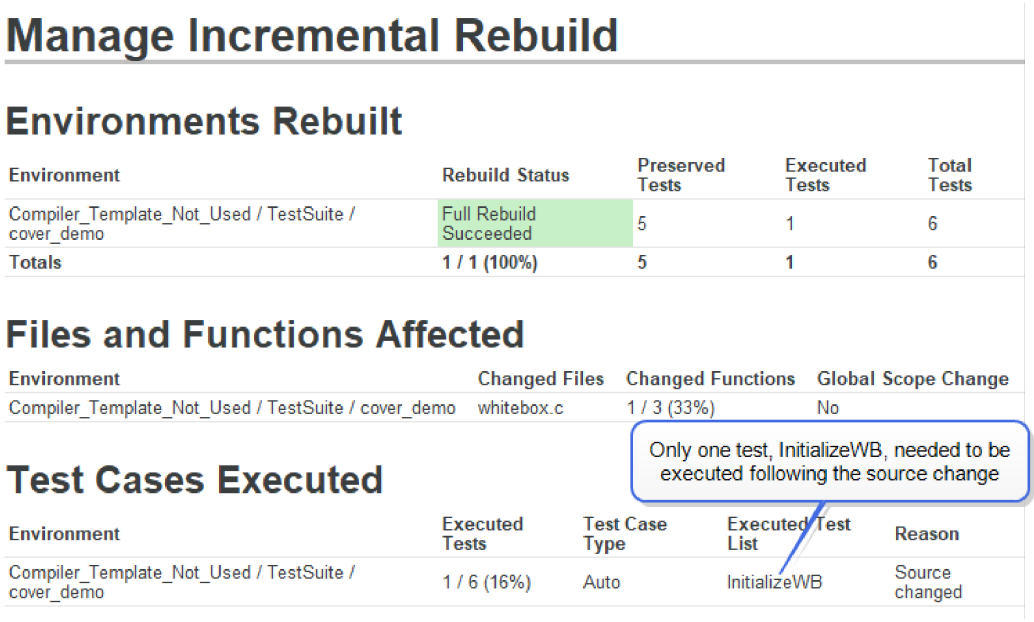
Once the Python configuration has been set up, the incremental build and execute functionality performs only the work that needs to be done, based on the current state of the project.

For example, if a change is made to the project's source files, performing an incremental build and execute will perform only the work that needs to be done based on the source change.

Run Incremental Build/Execute Automatically

To automatically run an incremental Build/Execute, right-click on the Environment node and select **Build/Execute => Incremental** from the context menu. For unit environments and migrated Cover environments, VectorCAST will fully build or rebuild environments and execute all affected or missing tests.

For example, the example report below shows that one test (InitializeWB) was affected due to a source file change in **`whitebox.c`**.



Manage Incremental Rebuild

Environments Rebuilt

Environment	Rebuild Status	Preserved Tests	Executed Tests	Total Tests
Compiler_Template_Not_Used / TestSuite / cover_demo	Full Rebuild Succeeded	5	1	6
Totals	**1 / 1 (100%)**	**5**	**1**	**6**

Files and Functions Affected

Environment	Changed Files	Changed Functions	Global Scope Change
Compiler_Template_Not_Used / TestSuite / cover_demo	whitebox.c	1 / 3 (33%)	No

Test Cases Executed

Only one test, InitializeWB, needed to be executed following the source change

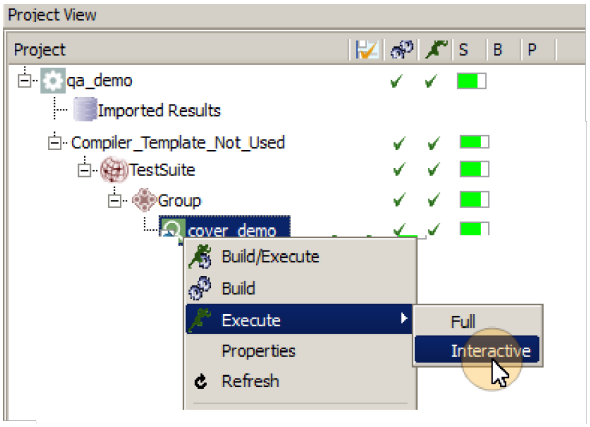
Environment	Executed Tests	Test Case Type	Executed Test List	Reason
Compiler_Template_Not_Used / TestSuite / cover_demo	1 / 6 (16%)	Auto	InitializeWB	Source changed

The Manage Incremental Rebuild Report shows that only one file was instrumented and only one test, InitializeWB, was run as a result of the source code change.

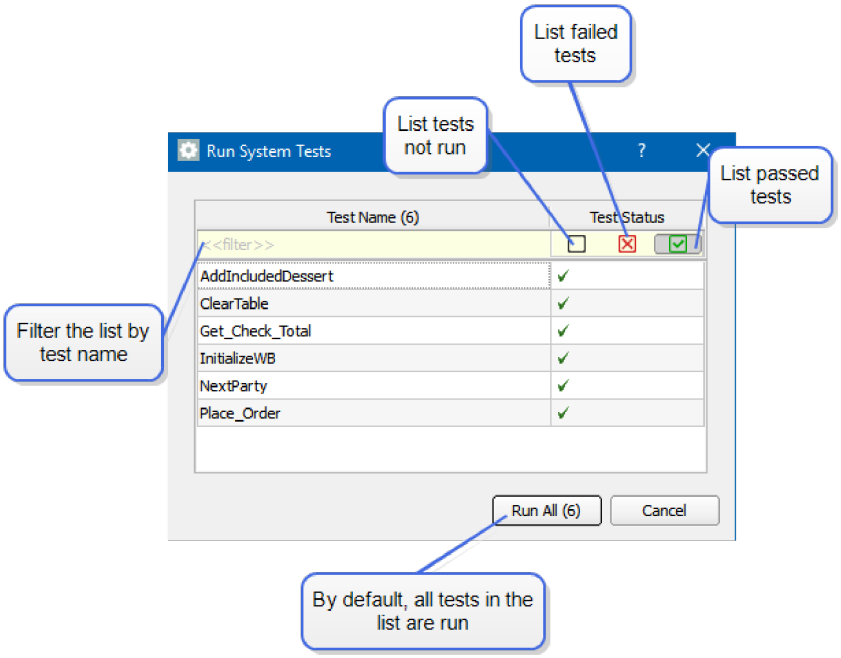
Interactive Execution of Tests

VectorCAST provides the ability to manually select a single test or a set of tests for execution in a System Test environment. Tests can be selected by Test Type and Test Name, allowing the user to run all or any subset of tests for the environment.

To select tests, right-click on a System Test environment in the Project Tree and select **Execute => Interactive** from the context menu.

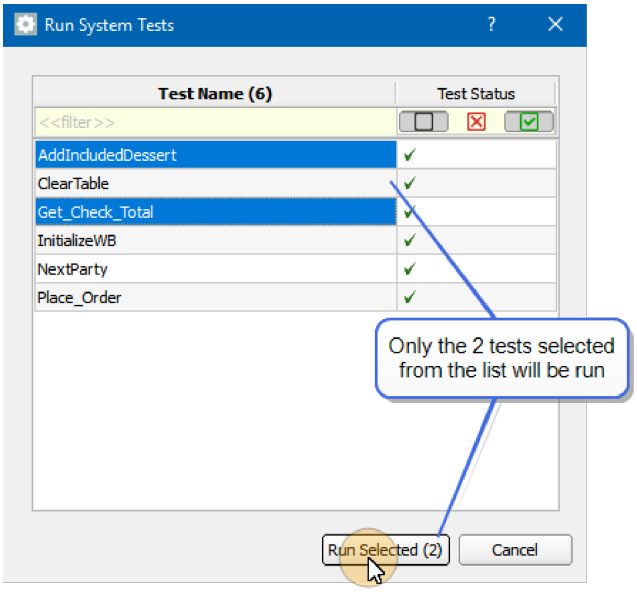


The Run System Tests dialog opens. You can filter the list by Test Name and Test Status (Not Run, Failed, or Passed), allowing you to quickly select a specific set of tests to run. The selection of multiple statuses is supported. By default, the Failed and Not Run test statuses are enabled.

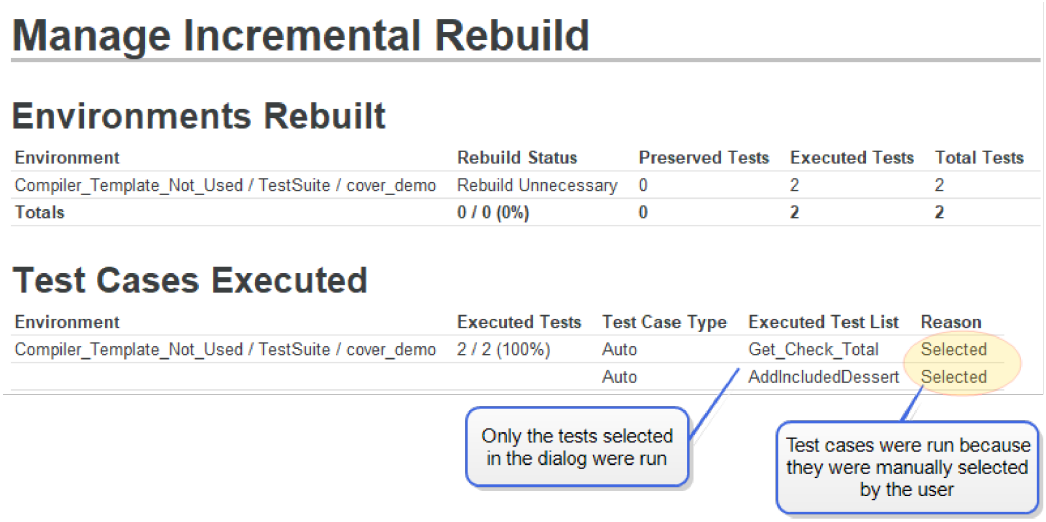


You can further refine the set of tests to be run by highlighting the Test Name(s) and selecting the **Run Selected** button. The button will display the total number of tests being selected to run. By default, VectorCAST runs all of the tests listed when no selections are made.

In our example, we will only run two tests, **`AddIncludedDessert`** and **`Get_Check_Total`**.



Select the **Run Selected** button to execute the list of selected test cases. The Manage Incremental Rebuild Report is displayed. Note that only our two manually selected test cases are executed.




```
manage -p <project> --level <level> -e <system test env> --system-
tests-status[=<filename>]

Lists the system tests and their execution status for the project or
the specified --level and environment. (default='')
```

```
manage -p <project> --level <level> -e <system test env> --execute-
system-tests

Executes all system tests for the given environment. To specify a sub-
```

```
set of all tests, use one or more of the optional flags:
--automated
--manual
--not-run
--failed
--passed
--testcase-name=<TEST_CASE_NAME> (May provide multiple test case
names.)

Note: this command differs from the global --build-execute or --
execute commands which cause only the not-run, automated tests to run.
```

Component Coverage

VectorCAST/QA provides support for component coverage. Component coverage is often used when an entire instrumented image is too large to fit on a target. When component coverage is active, the instrumentation of the application is performed for one component at a time, and the full set of tests is run for each component. Using component coverage reduces the amount of memory required to run tests since only a portion of the application is instrumented at any time.

Enabling Component Coverage

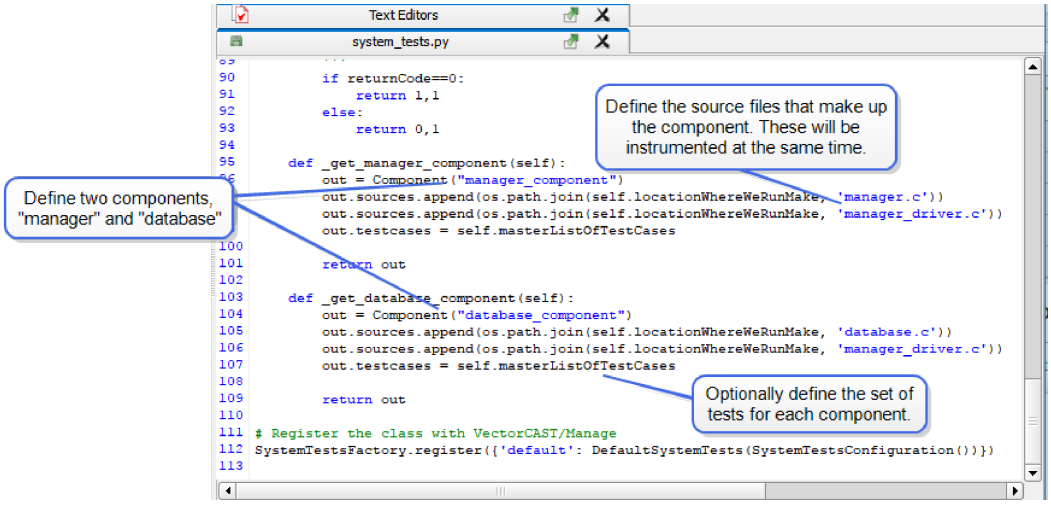
Component coverage is disabled by default. To enable component coverage, you must modify the python configuration script, **`system_tests.py`**. To open and edit the configuration script, right-click on a Cover Environment node and select **System Testing => Edit Script** from the context menu.

The **`system_tests.py`** script opens in the Script Editor. Modify the script as required and save changes. For our example, we first uncomment the following line of code to enable component coverage for two components, **`manager_component`** and **`database_component`**, and save the change:

```
self.components = [self._get_manager_component(), self._get_database_
component()]
```

`self.components` is a list of VectorCAST components. Each component is a sub-set of the files in the application. In our example, the components are built in the **`_get_manager_component`** and **`_get_database_component`** functions defined toward the bottom of our **`system_tests.py`** file.

Component names must be unique. Note that the **`system_tests.py`** script lists the source files that make up each component.



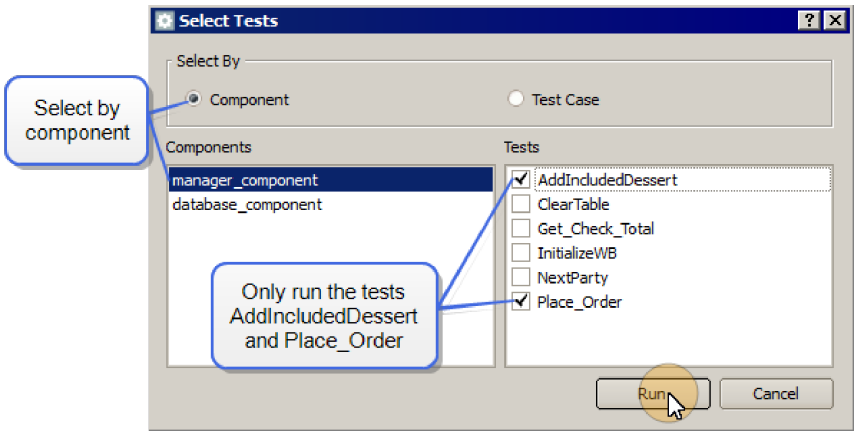
In the example above **`manager_component`** is defined on line 95. The source files that make up **`manager_component`** are listed. This set of files will be instrumented at the same time. Source files can be added or removed from a component by editing the configuration script and saving the changes.

Building and Executing on Components

Performing a full Build/Execute action on a Cover environment will run all the tests for each component. Optionally, an interactive Build/Execute can be performed allowing you to select specific test cases to be run for a component. For example, you may only want to run a single test for all components, or you may want to run just one test on one component.

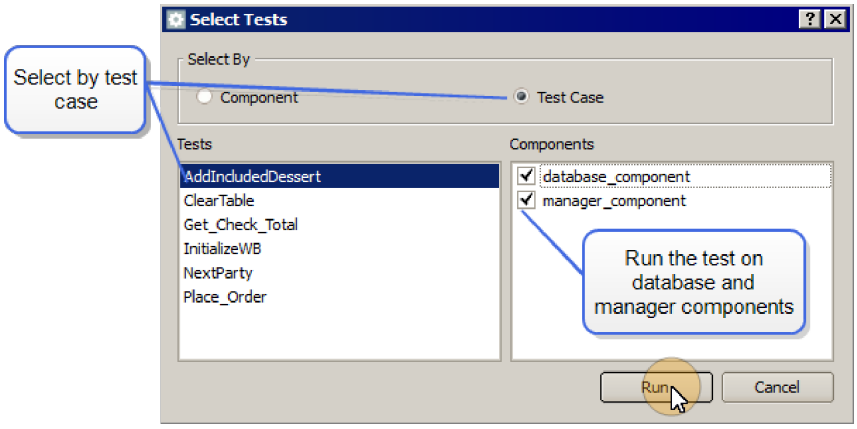
When component coverage is active, clicking on a Cover environment in the Project Tree and selecting **Build/Execute => Interactive** from the context menu opens the Select Tests dialog. Select the tests to execute based on Component or based on Test Case.

To select by Component, select the **Component** radio button. A list of available components for the environment is shown in the left pane. Highlight a component and the associated tests are listed in the right pane. Check the desired test(s) and select the **Run** button.

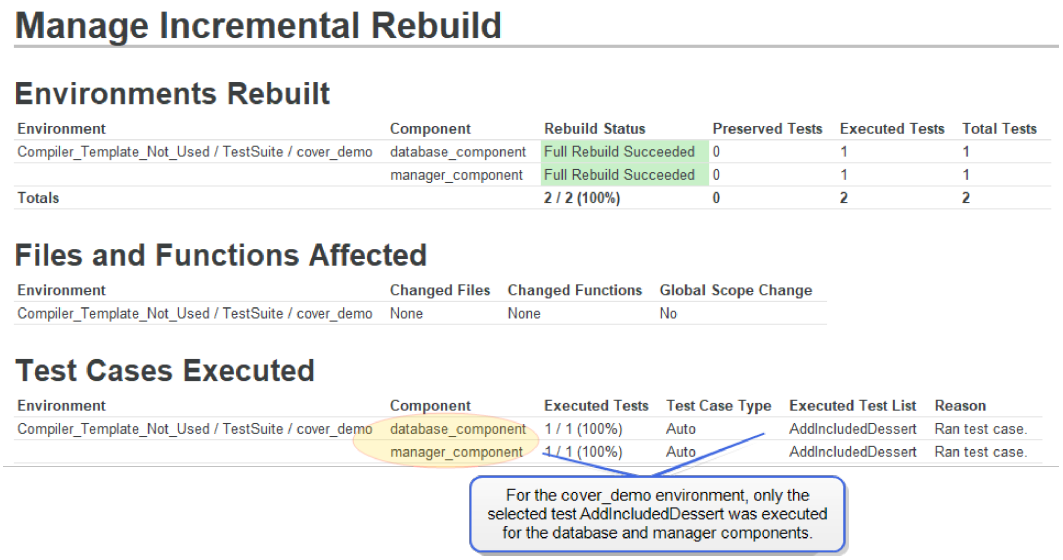


To select by Test Case, select the **Test Case** radio button. A list of available tests is shown in the left

pane. Highlight a test case and the associated components are listed in the right pane. Check the desired component(s) and select the **Run** button.



In our example, we ran the test **`AddIncludedDessert`** on both **`database_component`** and **`manager_component`**, and the results are provided in the Incremental Build Report. Running a test only on the components it touches saves time and resources.

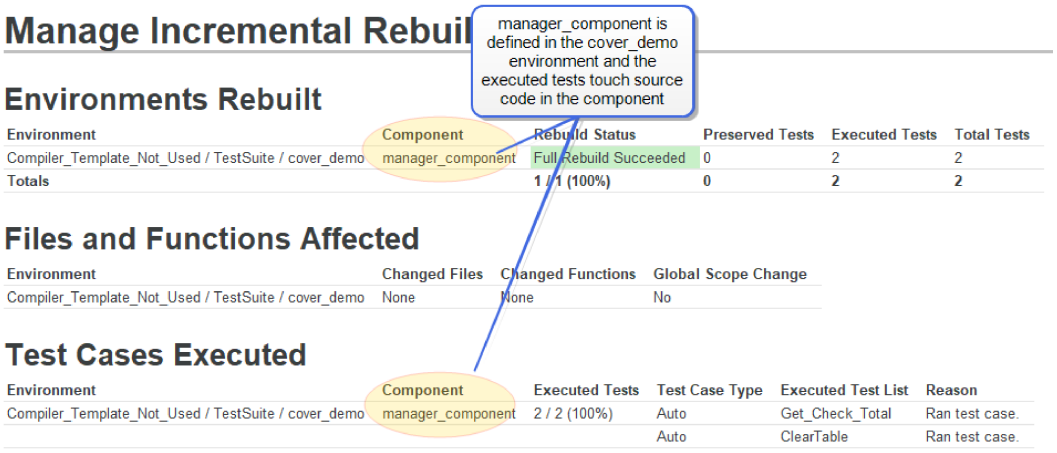



```
manage -p <project> --level <level> -e <system test env> --component-
name=<component_name> --testcase-name=<test_case_name> --execute-
system-tests

Executes system tests for components in the given environment. To
specify a component, use the optional flag --component name.

To specify a sub-set of tests, use the optional flag --testcase-name.

Note: the --component-name and --testcase-name options may be provided
multiple times.
```

Viewing Component Coverage Data

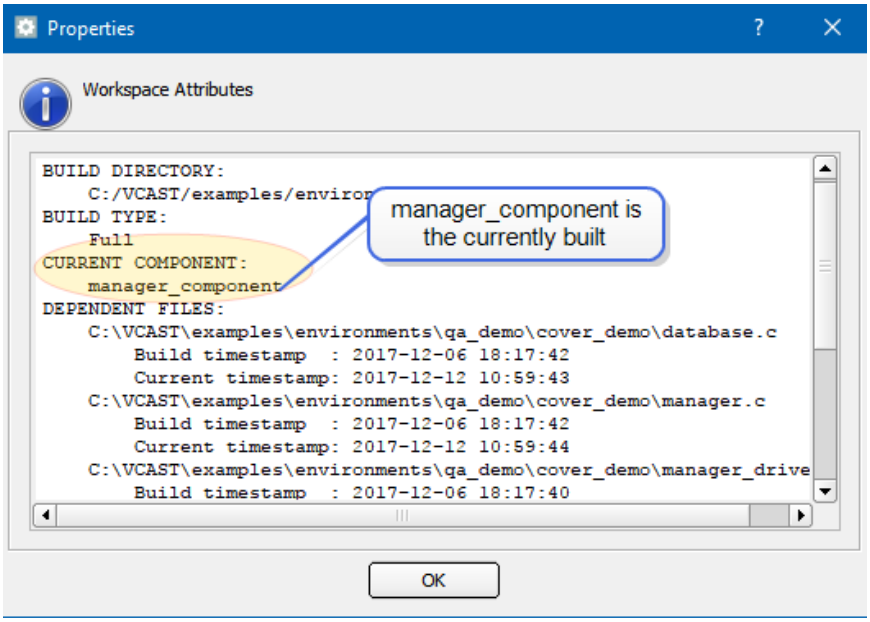
Manage Incremental Rebuild Report

When performing a Build/Execute action with Component Coverage active, the Incremental Rebuild Report includes a Component column in the Environments Rebuilt section. This column identifies the component(s) which are affected when an environment is rebuilt. The report also includes a Components column in the Test Cases Executed section which indicates the component that is affected when a test case is executed that touches source files that make up the component.

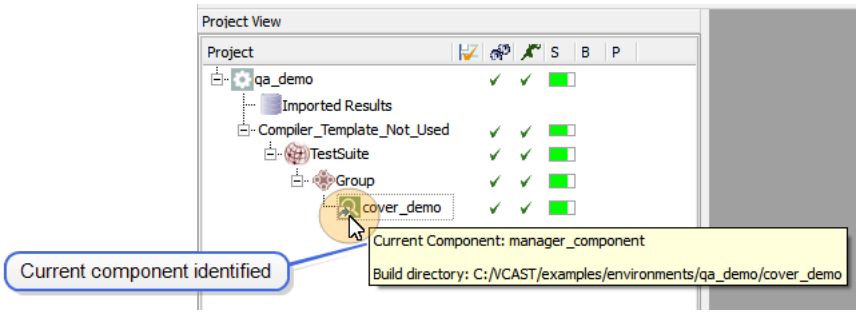


Properties Dialog

Right-clicking on an environment node in the Project Tree and selecting **Properties** from the context menu opens the Properties Dialog for the Workspace Attributes. The currently built component is listed.



The current component can also be quickly identified by hovering over the environment in the Project Tree and displaying the associated tool-tip.

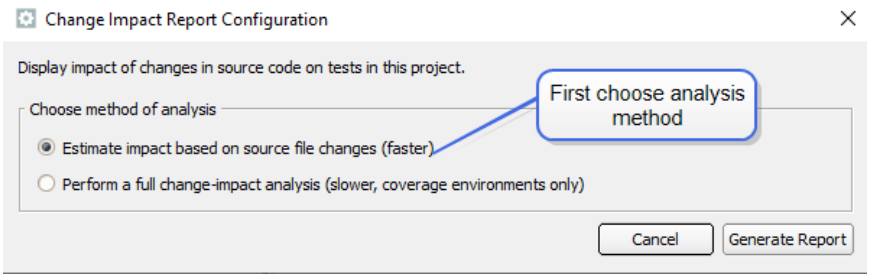


Change Impact Report

VectorCAST/QA provides the ability to evaluate the effect of source code changes on test cases using the Change Impact Report. This feature evaluates "what-if" situations.

For example, you may be considering re-factoring a widely-used function. The Change Impact Report shows that the planned changes will trigger the need to re-run 85% of the existing tests. Knowing this in advance, you might choose to postpone the planned changes until schedule pressure eases.

To generate a Change Impact Report, select **Project => Change Impact Report...** from the Menu Bar. The Change Impact Report Configuration dialog opens.



The dialog provides access to both the estimated Change Impact Report for unit test and Coverage environments and the Build/Execute Dry Run report for Coverage environments.

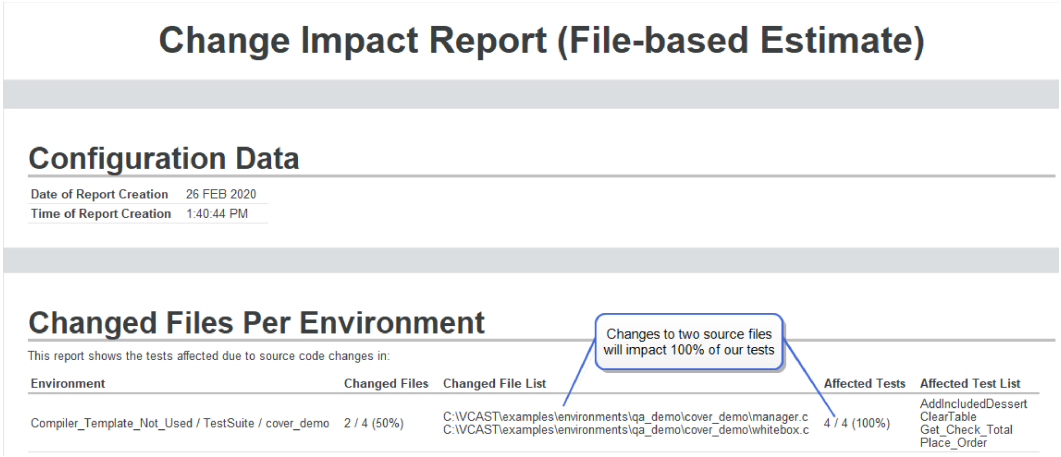
Estimated Change Impact Report

The Estimated Change Impact Report method can be used for unit test and Coverage environments. This method also supports displaying the impact from changes in an alternative root source directory on tests in the Project. Select the **Estimate impact based on source file changes** radio button.

Note: All tests that touch a file are considered affected by any change to that source file. This may differ from the actual action taken by incremental build/execute.

Select the **Generate Report** button. The title of report is "Change Impact Report (File-based Estimate)." When generated in the GUI, the report is saved automatically in the **`<project>/build/vcast_data`** directory.

```
manage -p <project> [--level=<:level>] --change-impact-report

The default filename is <project>_change_impact_report.html in the working
directory.
```



Full Change Impact Report

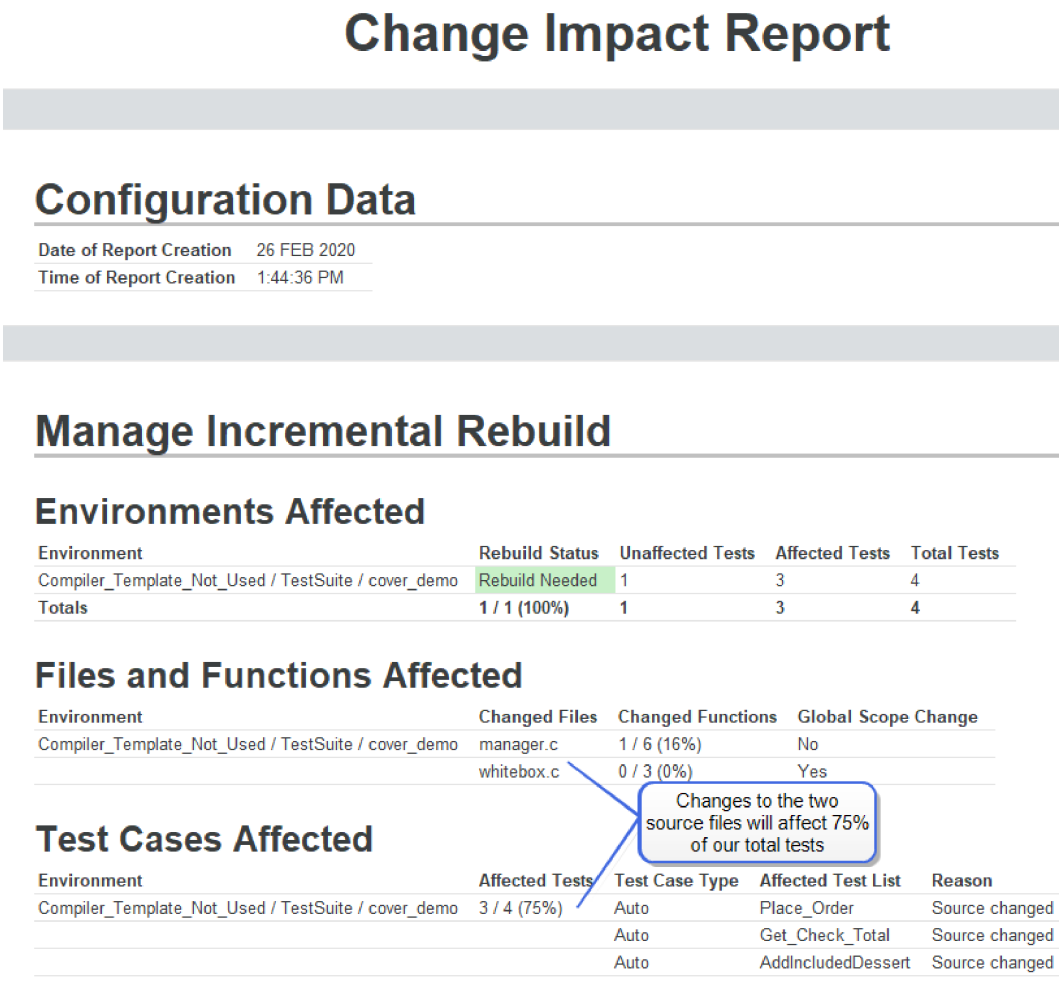
The Full Change Impact Report method is only used for Coverage environments that have a system test implementation defined in the **`system_tests.py`** file. Select the **Perform a full change-**

impact analysis radio button.

Note that only those tests that touch a changed function are considered affected. This method is slower due to preprocessing, but produces output matching the actual actions taken by incremental build/execute.

Select the **Generate Report** button.The title of report is "Change Impact Report." When generated in the GUI, the report is saved automatically in the **<project>/build/vcast_data** directory.

```
manage -p <project> [--level=<:level>] --change-impact-report --full-
analysis

The default filename is <project>_manage_incremental_rebuild_report.html in
the working directory.
```



Change Impact Report

Configuration Data

Date of Report Creation	26 FEB 2020
Time of Report Creation	1:44:36 PM

Manage Incremental Rebuild

Environments Affected

Environment	Rebuild Status	Unaffected Tests	Affected Tests	Total Tests
Compiler_Template_Not_Used / TestSuite / cover_demo	Rebuild Needed	1	3	4
Totals	**1 / 1 (100%)**	**1**	**3**	**4**

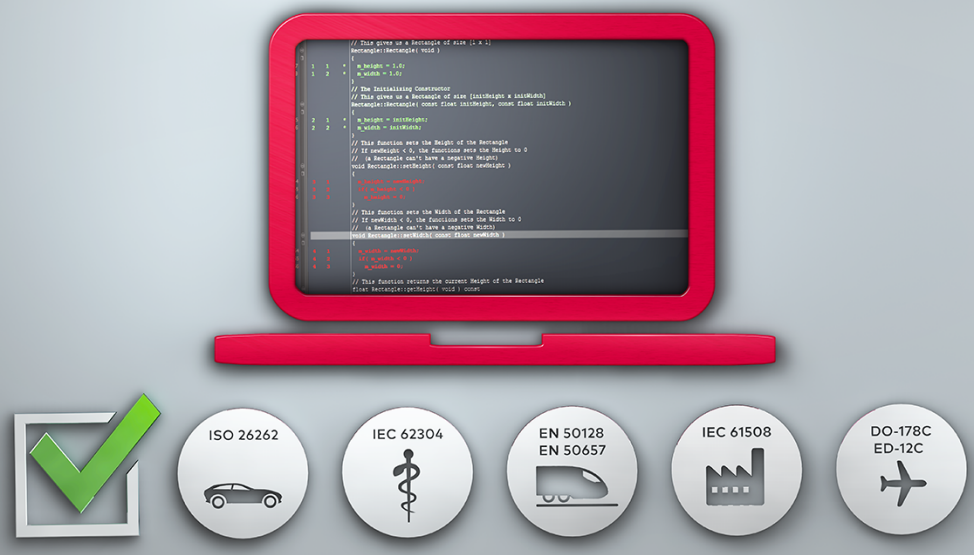
Files and Functions Affected

Environment	Changed Files	Changed Functions	Global Scope Change
Compiler_Template_Not_Used / TestSuite / cover_demo	manager.c	1 / 6 (16%)	No
	whitebox.c	0 / 3 (0%)	Yes

Changes to the two source files will affect 75% of our total tests

Test Cases Affected

Environment	Affected Tests	Test Case Type	Affected Test List	Reason
Compiler_Template_Not_Used / TestSuite / cover_demo	3 / 4 (75%)	Auto	Place_Order	Source changed
		Auto	Get_Check_Total	Source changed
		Auto	AddIncludedDessert	Source changed



VectorCAST Cover Environment

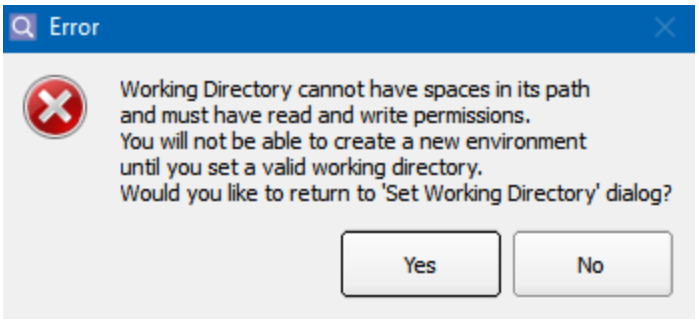
Creating a System Testing Environment

To Set the Working Directory

The working directory is the default location VectorCAST uses for building environments and when loading an environment script that has been saved with relative paths. The local configuration file is also saved there. This file contains all tool options that are modified in the **Tools => Options** dialog. Also, it is the default directory in the File Save dialog when saving a report or exporting a script to a file, and the default directory in the File Open dialog when opening an environment or importing a script.

The **File => Set Working Directory** command is used to change the current working directory before creating a new environment. This command is not available when an environment is open. When you select this command, a dialog appears enabling you to navigate to the directory of choice or create a new directory.

If you select a directory which contains spaces or does not have read / write permissions, you will receive the following error and be prompted to return to the Set Working Directory dialog to set a valid directory.



The “Remember last working directory” option is located on the **Tools => Options dialog, GUI tab**. Whenever this option is enabled, VectorCAST opens in the last-used working directory, regardless of how the VectorCAST application is invoked.

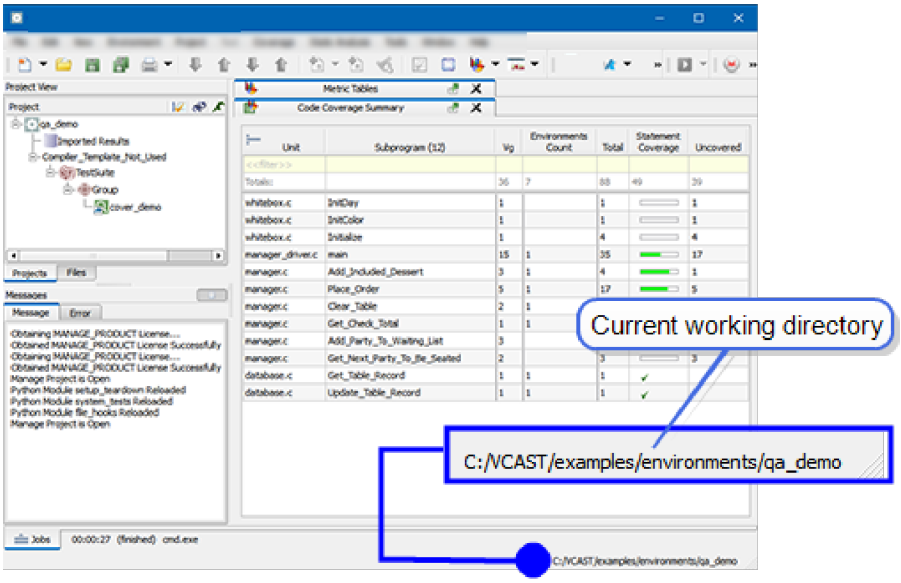
When you start the VectorCAST application from the command line, the working directory is the directory from which you invoked the application, unless the “Remember last working directory” option is set.

When you start VectorCAST from the Windows Start menu, the working directory is the Environments directory in the VectorCAST installation directory, unless the “Remember last working directory” option is set. On Windows, you can change the default working directory by modifying the Windows properties of the VectorCAST shortcut.

Note: The working directory must not contain spaces and must have Read and Write permissions. If you start VectorCAST from such a directory, the status bar has a red outline. The Set Working Directory error dialog appears, prompting you to enter a valid directory.

If you install VectorCAST in the C:\Program Files\ directory, then the Environments directory will have spaces in its path and therefore will be invalid. If this happens, set the working directory to a different location on your drive.

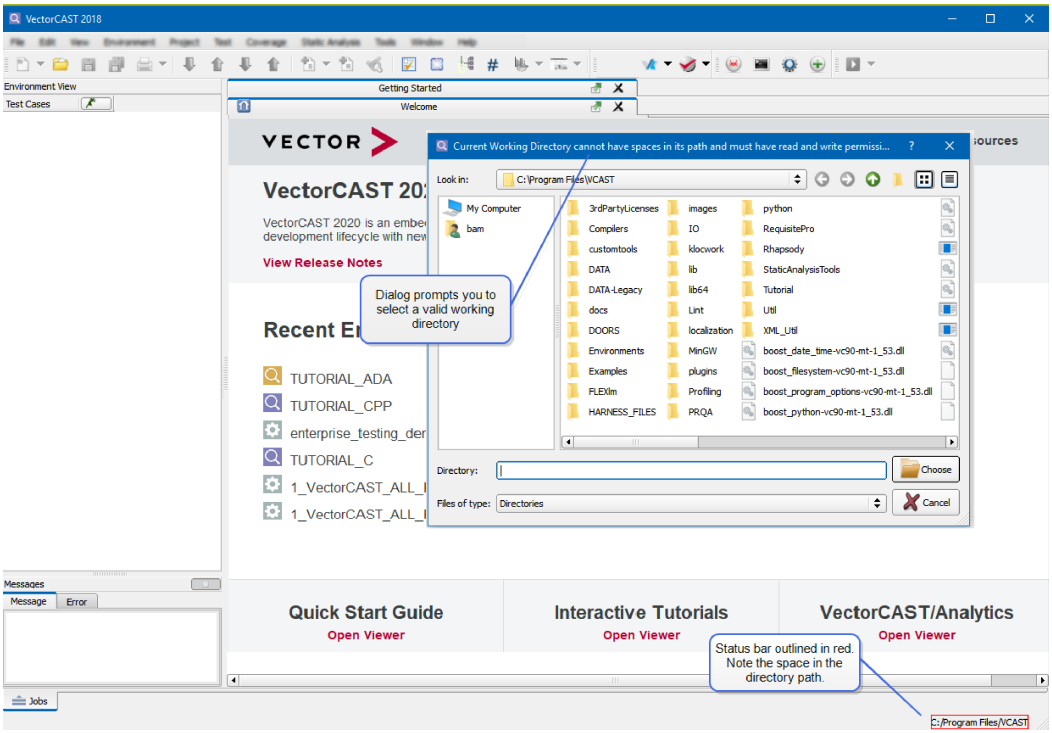
The current working directory is indicated in the status bar in the main application window.



Troubleshooting the Working Directory

Space in Working Directory Path, or Current Working Directory is Not Writeable

For Windows users, VectorCAST does not support spaces in the default directory path, and you must have write permission. (For some Windows users, the C:\ drive is not writeable.)

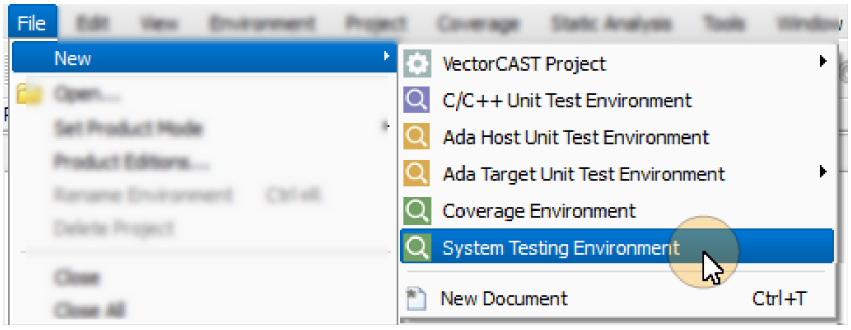


If you start VectorCAST from such a directory, the status bar has a red outline. The Set Working Directory dialog appears, prompting you to enter a valid directory.

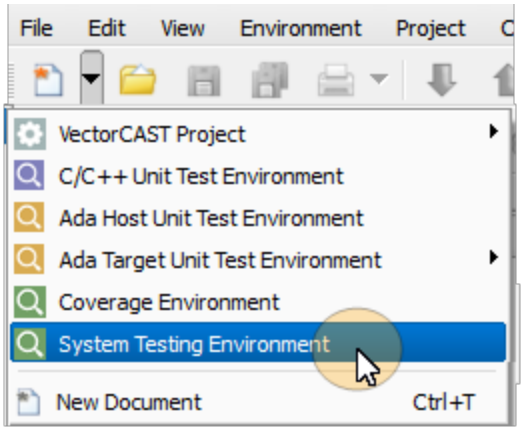
To Create a System Testing Environment using the Wizard

The System Testing Environment Wizard creates a migrated Cover environment within a VectorCAST project.

To create a System Testing environment using the Wizard, select **File => New => System Testing Environment** from the Menu Bar.



Alternatively, click the New button  on the Toolbar, or click the arrow ▼ to the right of the button and select System Testing Environment.

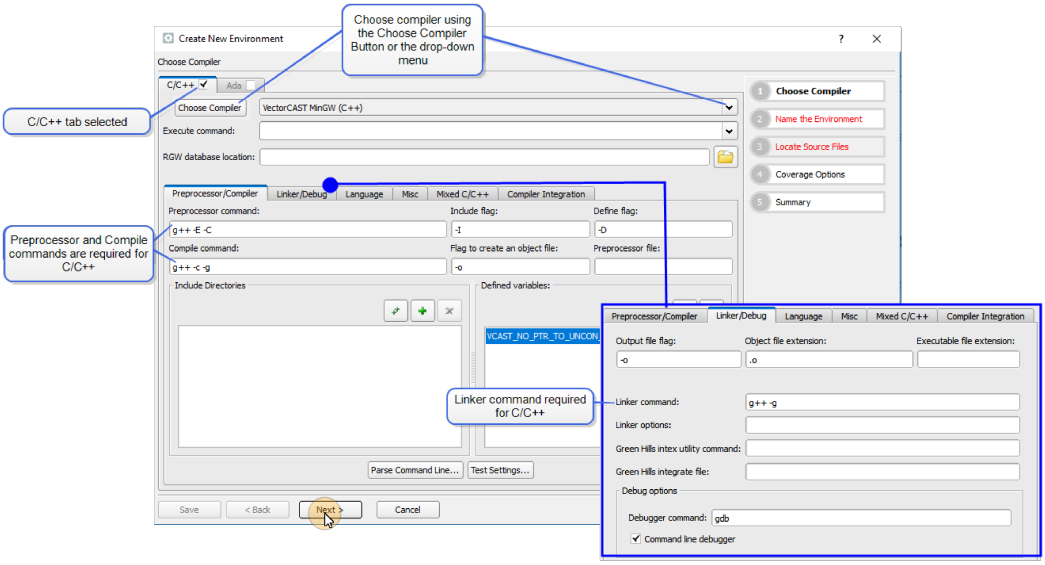


Step 1: Choose Compiler

By default, the **C/C++** tab is enabled with a check mark. Enable the **Ada** tab by putting a check mark in the tab heading. Enabling the **Ada** tab allows you to specify Ada file extensions. You can choose one or both languages.

On this page you perform the following:

- Choose a C/C++ compiler template from the **Choose Compiler** button drop down menu, or enter the commands for preprocess and compile. See "To Set the Working Directory" on page 54 for more information (required if C/C++ tab enabled).
- Add paths to the list of Include directories (optional). These directories are searched for header files needed for preprocessing.
- Add or remove source file extensions on the Language sub-tab if your C/C++ source code uses anything other than **.c** for C source files, or **.cpp, .c**++, .cxx, **.cc** for C++ files (optional)
- Add or remove source file extensions on the Ada tab if your Ada source code uses anything other than **.ada**, **.ADA**, **.adb**, **.ADB**, **.a,** or **.A** (optional).
- Add #defines needed during preprocessing of C/C++ source files (optional)



If the **C/C++** tab is enabled, you must specify a valid preprocessor command and compile command on the Preprocessor/Compiler tab, and on the Linker/Debug tab you must specify a valid linker command for the **Next** button to be enabled.

However, only the preprocessor command, compile command, and the Parser Flags are used during instrumentation if the option “Preprocess files before instrumenting” is on.

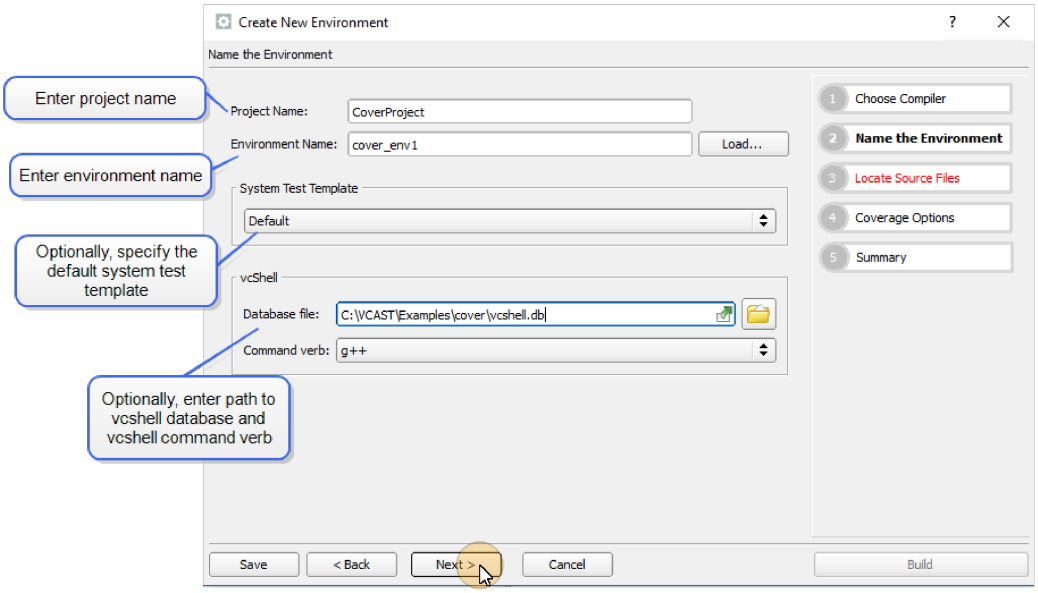
If only the **Ada** tab is enabled, nothing is required.

Click **Next** to proceed to Step 2: Name the Environment.

Step 2: Name the Environment

On this page you perform the following:

- Enter the name for the project (required)
  Any character except * ( ) \ | “ ‘ can be used; spaces are converted to underbars (_). If a project with that name exists already, the name is outlined in red and the Next button is dimmed.
- Enter the name for the environment (required)
  Any character except * ( ) \ | “ ‘ can be used; spaces are converted to underbars (_). If a directory with that name exists already, the name is outlined in red and the Next button is dimmed.
- Specify the default system test template (optional)
  A template is used to create the system test script for the given environment. For example, the Google Test template should be chosen to connect with an existing Google Test framework.
- Specify the path to the vcshell database (vcshell.db) (optional)
- Select the command verb used for the source files to be tested in the environment (optional)



Once you have provided the project and environment names, click the **Next** button to proceed to Step 3: Locate Source Files.

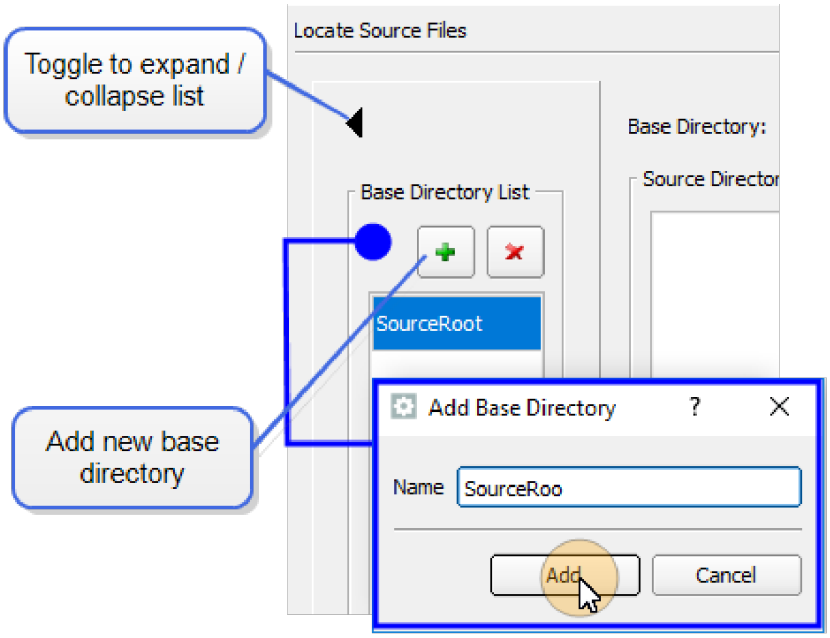
Step 3: Locate Source Files

On this page you perform the following:

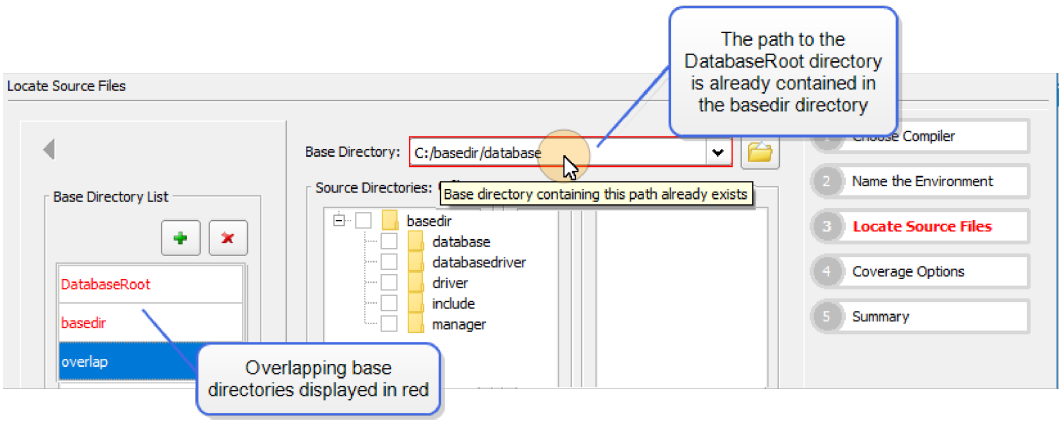
- Specify the location of your source code repository (required)
- Select files and folders to include or exclude from the environment (required)
- Include header files in the environment (optional)

First, a Base Directory must be specified. The Base Directory List shows the Base Directories, or source root directories, for the source code repositories that comprise the environment. Click the arrow on the upper left to expand the Base Directory List.

To add a repository to the list, select the **Add Base Directory** button , or right-click in the pane and select **Add...** from the context menu. Enter the name of the new directory and select the **Add** button.

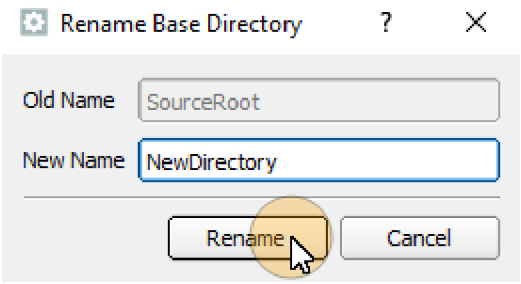


Note when adding directories that base directories cannot overlap. Overlapping base directories are displayed in red in the Base Directory List, and the text box is displayed with a red outline.




```
clicast -lc -e <env> Cover Base_dir ADd <name> <path>

This specifies the location where source files reside. This can be set to
an environment variable or relative path.
```

Base Directories can be renamed by right-clicking on the directory and selecting **Rename...** from the context menu. The Rename Base Directory dialog opens. Enter the new name for the directory and select the **Rename** button.




```
clicast -lc -e <env> Cover Base_dir RENname <old> <new>

Change the name identifier for a base directory.
```

To remove a Base Directory from the list, highlight the directory and select the **Remove Base Directory** button , or right-click on the directory and select **Remove...** from the context menu. A dialog is displayed confirming that you want to remove the selected directory.

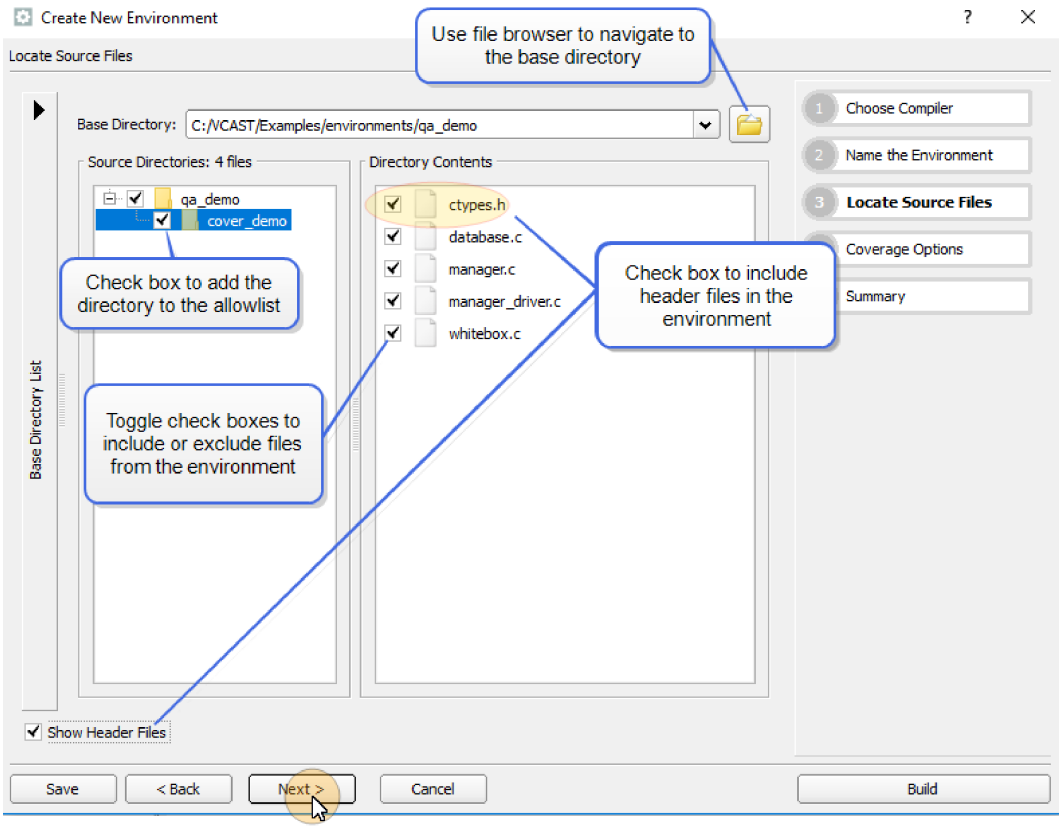
```
clicast -lc -e <env> Cover Base_dir REMove <name>

After this base directory is removed from the environment, the allowlisted
items within this directory will no longer be updated.
```

Next, you must enter the path to the Base Directory. The directory path can be an environment variable (for example, $(ENV_VARIABLE), a relative path, or an absolute path.

Note: It is recommended that the path to the Base Directory be portable. Although supported, the use of an absolute path is discouraged.

Use the file browser button located to the right of the Base Directory field to navigate to the location of the source directory and select the **Choose** button.

The path to the Base Directory is saved in the environment script (`.enc`). This means only a single change is needed if loading the environment script in a different directory.



Highlight the source directory to view the contents in the Directory Contents pane on the right. Selecting the checkbox next to the directory includes all the files in the directory. When a directory is checked, it is added to the allowlist. When a user's source repository changes (for example, a file is added or removed from the allowlisted directory) and the user updates the environment or runs the environment script (`.enc`), the Cover environment automatically identifies any changes that have occured in that directory.

Note: To take advantage of the automatic syncing of the files in the Cover environment, it is recommended that you check the topmost directories in your base directory and then denylist those files and directories where you do not want to collect code coverage. This allows the Cover environment to evolve with the source code.

In the Directory Contents pane, use the check boxes next to the source file names to include and exclude files from the environment. Checked source files are allowlisted and included in the environment. Unchecked files are denylisted and excluded from the environment.

The topmost checked paths are added to the allowlist. It is recommended to allowlist those parent directories that contain source code. Subdirectories and files can be unchecked. Any unchecked items are denylisted from the Cover environment.

```
clicast -lc -e <env> Cover Base_dir ALlowlist <name> <path>

The given path will be included in the base directory's allowlist. By
adding paths to the allowlist, the cover environment can compare its state
```

```
with state of the source tree and make any needed updates.
```

```
clicast -lc -e <env> Cover Base_dir Denylist <name> <path>

Denylisted files and directories will be excluded from the cover
environment on update.
```

A file or directory is included in the allowlist if and only if it is:

- Allowlisted explicitly, or
- Not explicitly denylisted and its parent directory is in the allowlist.

To include header files in the environment, check the **Show Header Files** box at the lower left corner of the window. When checked, header files are listed alongside the source files in the Directory Contents pane. Checked headers are used for code coverage. Note that checking this also sets the VCAST_COVERAGE_FOR_HEADERS option.

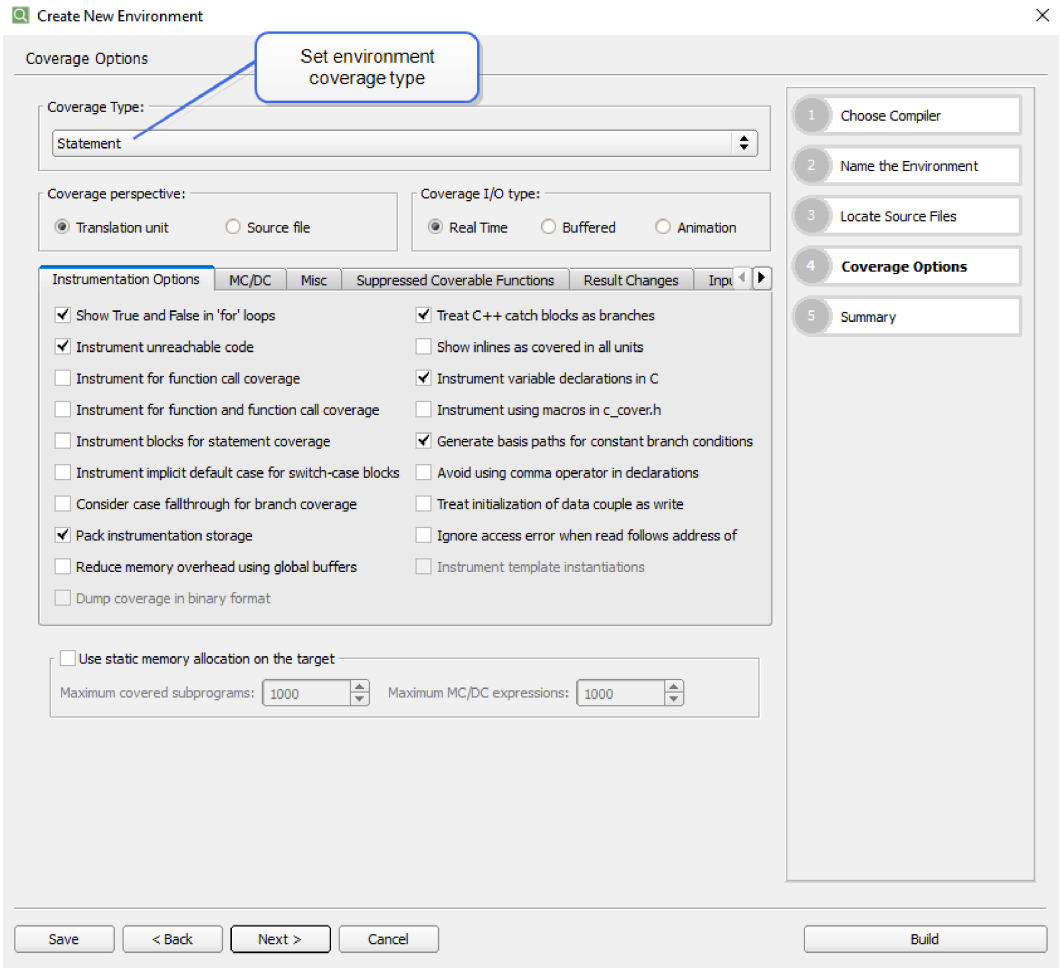
Click **Next** to proceed to Step 4: Coverage Options.

Step 4: Coverage Options

This page of the Cover wizard has a subset of the coverage options from the Options dialog, Coverage tab.

On this step you perform the following:

- Specify the default value of coverage type to set for the environment immediately after building it (optional). When setting this option, the environment coverage type is set, but no files are instrumented when the environment is built. Note that coverage type None is not allowed. The default coverage type is Statement.
- Specify the coverage perspective (optional). The default perspective is Translation Unit.
- Specify the coverage I/O type (optional). The default I/O type is Real Time.
- Specify the instrumentation directory (optional). The default is **`vc-inst`**.
- Change instrumentation options or static memory options (optional).

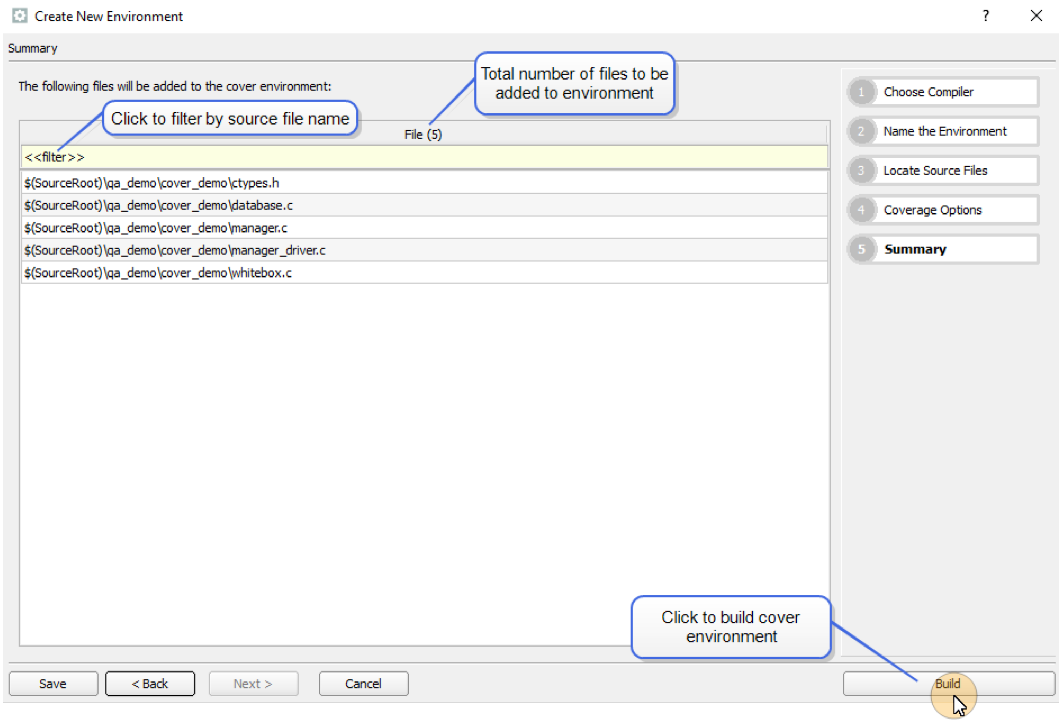


There are no required actions on this page.

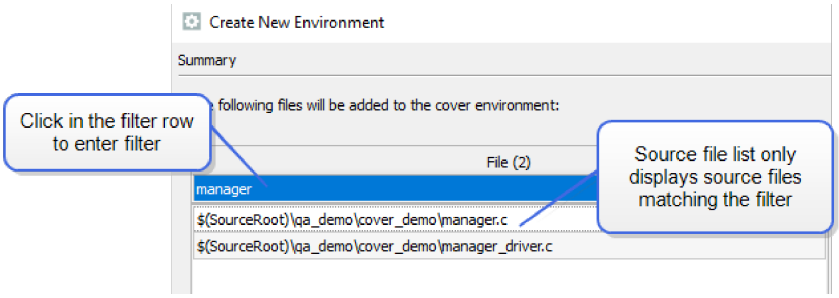
Click **Next** to view a Summary of the environment source files.

Step 5: Summary

This page of the Cover wizard shows a summary of the files to be added to the cover environment, based on the selections made on the Locate Source Files window.



Filtering is available to locate source files of interest. Filter by clicking in the filter row and typing into the row. In the example below, the list has been filtered to only show source files containing "manager" in the file name.



To clear the filter, highlight the filter row, right-click and select **Clear Filter** from the context menu.

There are no required actions on this page.

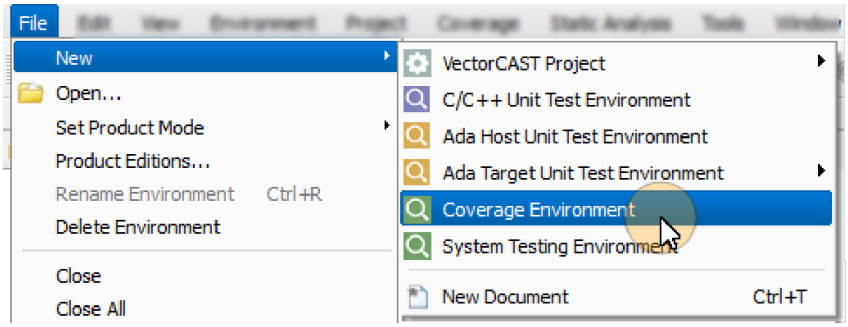
Click **Build** to create the System Testing environment. No coverage instrumentation is performed.

Once the System Testing environment is built, you are able to use VectorCAST/QA as you normally would to set options, instrument, add or remove source files, etc.

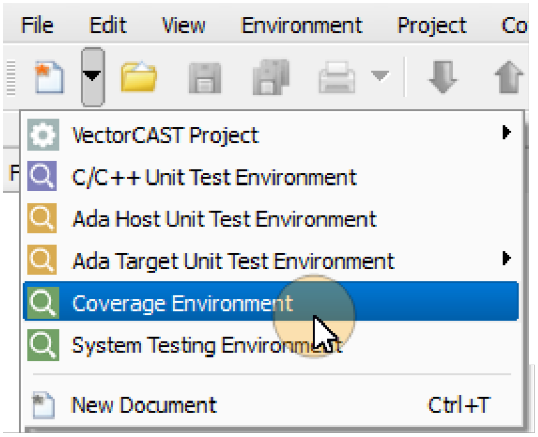
Creating a Coverage Environment

VectorCAST provides the ability to create a standalone Cover environment outside of a VectorCAST project.

To create a standalone Cover environment using the Wizard, select **File => New => Coverage Environment** from the Menu Bar.



Alternatively, click the New button  on the Toolbar, or click the arrow ▼ to the right of the button and select Coverage Environment.



The Create New Environment Wizard opens.

Step 1: Choose Compiler

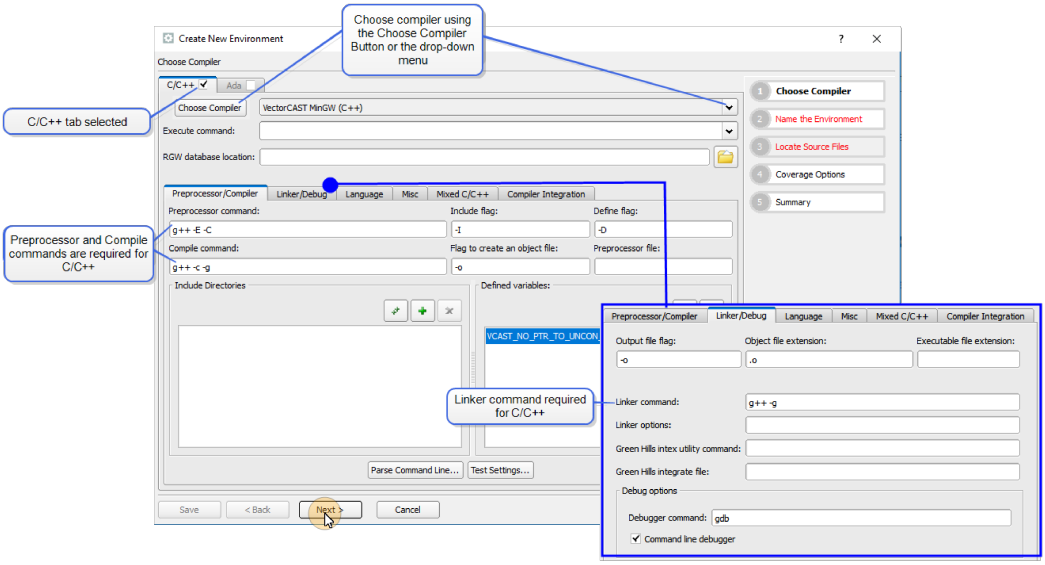
By default, the **C/C++** tab is enabled with a check mark. Enable the **Ada** tab by putting a check mark in the tab heading. Enabling the **Ada** tab allows you to specify Ada file extensions. You can choose one or both languages.

On this page you perform the following:

- Choose a C/C++ compiler template from the Choose Compiler button drop down menu, or enter the commands for preprocess and compile. See "To Set the Working Directory" on page 54 for more information (required if C/C++ tab enabled).
- Add paths to the list of Include directories (optional). These directories are searched for header files needed for preprocessing.
- Add or remove source file extensions on the Language sub-tab if your C/C++ source code uses anything other than **.c** for C source files, or **.cpp, .c**++, .cxx, **.cc** for C++ files (optional)

> Add or remove source file extensions on the Ada tab if your Ada source code uses anything other than **.ada**, **.ADA**, **.adb**, **.ADB**, **.a,** or **.A** (optional).

> Add #defines needed during preprocessing of C/C++ source files (optional)



If the **C/C++** tab is enabled, you must specify a valid preprocessor command and compile command on the Preprocessor/Compiler tab, and on the Linker/Debug tab you must specify a valid linker command for the **Next** button to be enabled.

However, only the preprocessor command, compile command, and the Parser Flags are used during instrumentation if the option “Preprocess files before instrumenting” is on.

If only the **Ada** tab is enabled, nothing is required.

Click **Next** to proceed to Step 2: Name the Environment.

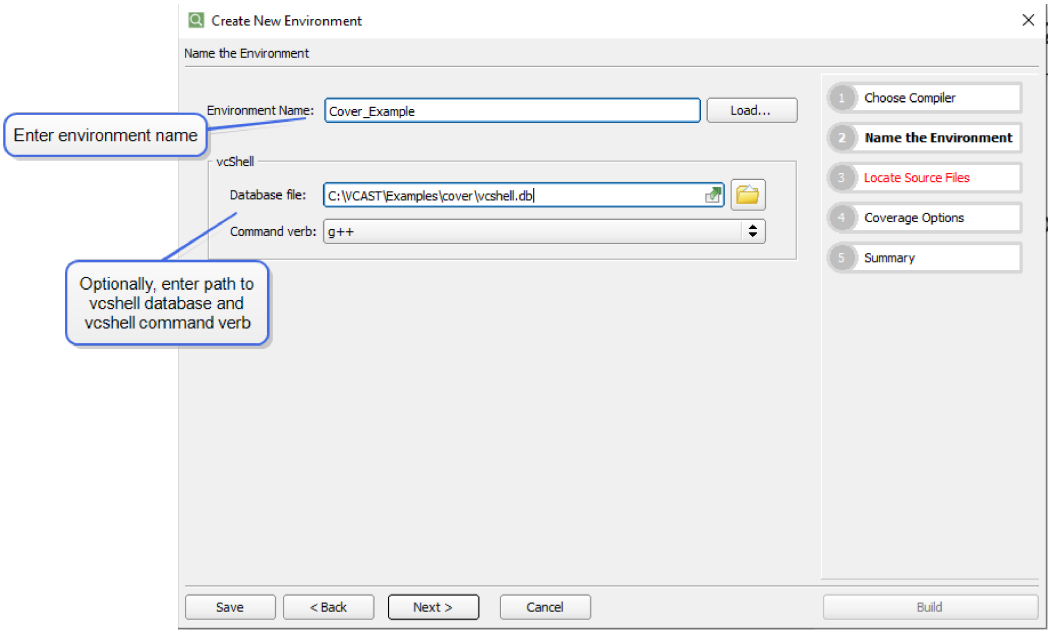
Step 2: Name the Environment

On this page you perform the following:

> Enter the name for the environment (required)
Any character except * ( ) \ | “ ‘ can be used; spaces are converted to underbars (_). If a directory with that name exists already, the name is outlined in red and the Next button is dimmed.

> Specify the path to the vcshell database (vcshell.db) (optional)

> Select the command verb used for the source files to be tested in the environment (optional)



Once you have provided the environment name, click the **Next** button to proceed to Step 3: Locate Source Files.

```
clicast -lc Cover ENvironment Create <environment-name>

Create a new environment.
```

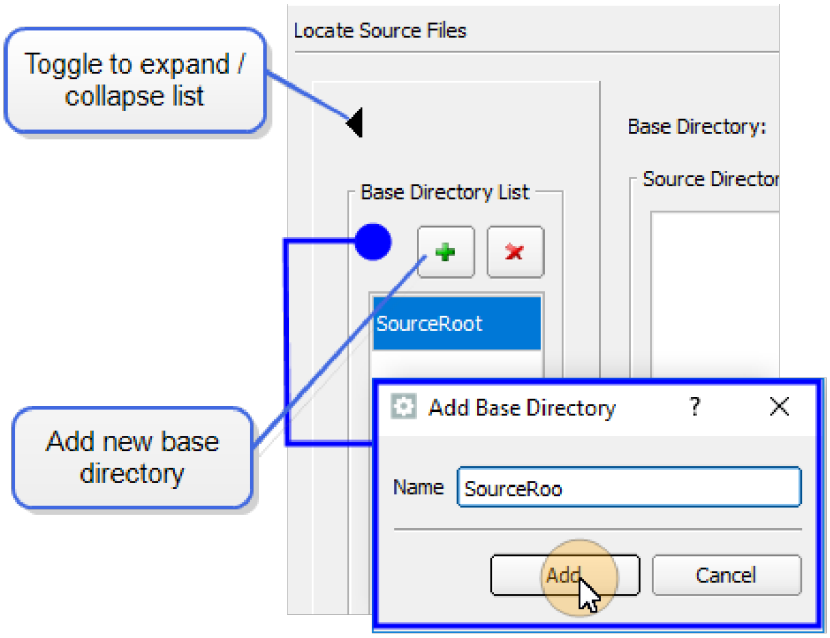
Step 3: Locate Source Files

On this page you perform the following:

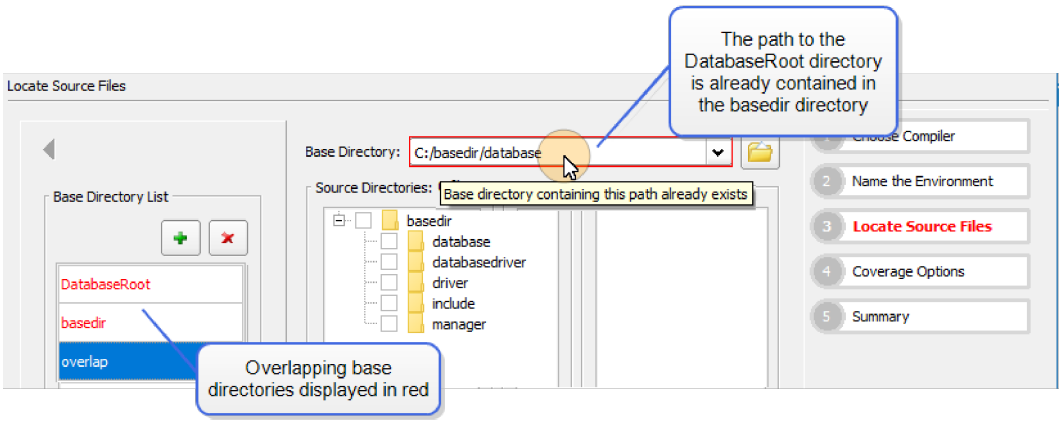
- Specify the location of your source code repository (required)
- Select files and folders to include or exclude from the environment (required)
- Include header files in the environment (optional)

First, a Base Directory must be specified. The Base Directory List shows the Base Directories, or source root directories, for the source code repositories that comprise the environment. Click the arrow on the upper left to expand the Base Directory List.

To add a repository to the list, select the **Add Base Directory** button , or right-click in the pane and select **Add...** from the context menu. Enter the name of the new directory and select the **Add** button.

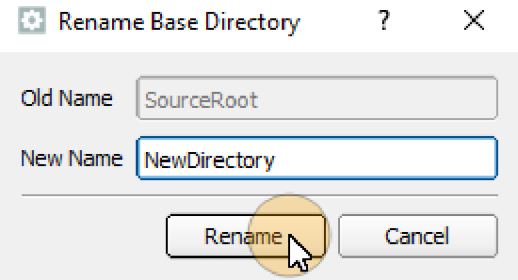


Note when adding directories that base directories cannot overlap. Overlapping base directories are displayed in red in the Base Directory List, and the text box is displayed with a red outline.




```
clicast -lc -e <env> Cover Base_dir ADd <name> <path>

This specifies the location where source files reside. This can be set to
an environment variable or relative path.
```

Base Directories can be renamed by right-clicking on the directory and selecting **Rename...** from the context menu. The Rename Base Directory dialog opens. Enter the new name for the directory and select the **Rename** button.




```
clicast -lc -e <env> Cover Base_dir REName <old> <new>

Change the name identifier for a base directory.
```

To remove a Base Directory from the list, highlight the directory and select the **Remove Base Directory** button , or right-click on the directory and select **Remove...** from the context menu. A dialog is displayed confirming that you want to remove the selected directory.

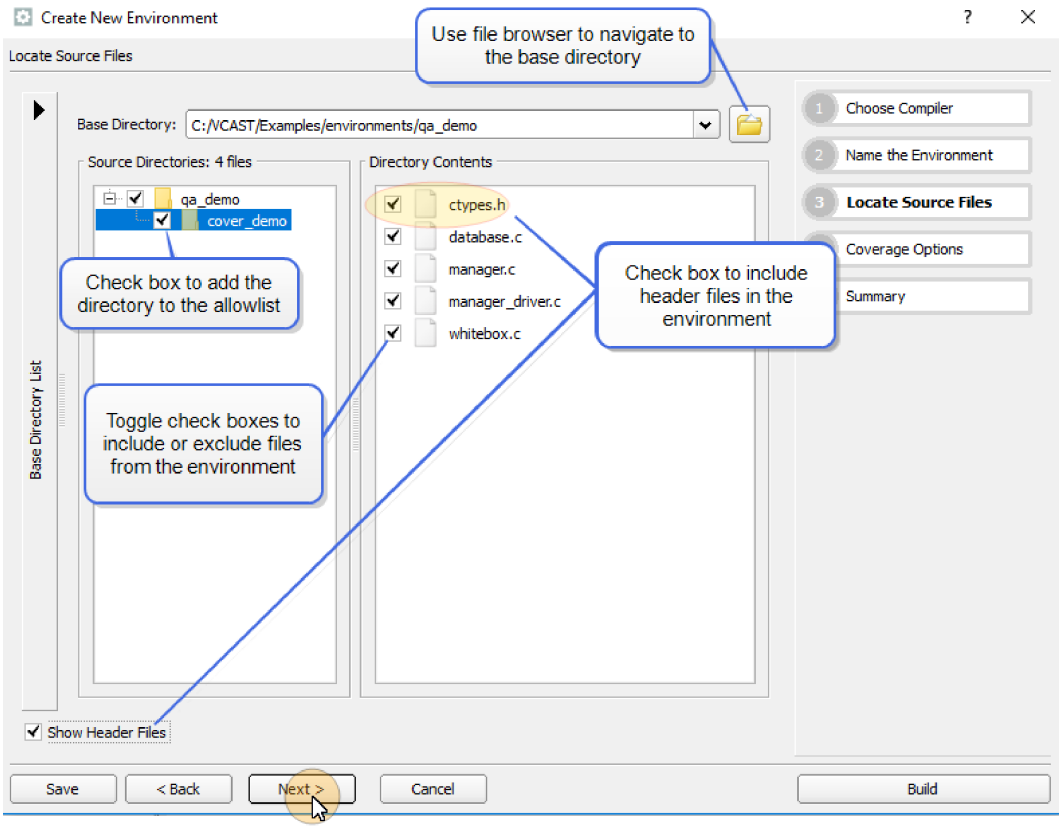
```
clicast -lc -e <env> Cover Base_dir REMove <name>

After this base directory is removed from the environment, the allowlisted
items within this directory will no longer be updated.
```

Next, you must enter the path to the Base Directory. The directory path can be an environment variable (for example, $(ENV_VARIABLE), a relative path, or an absolute path.

Note: It is recommended that the path to the Base Directory be portable. Although supported, the use of an absolute path is discouraged.

Use the file browser button located to the right of the Base Directory field to navigate to the location of the source directory and select the **Choose** button.

The path to the Base Directory is saved in the environment script (`.enc`). This means only a single change is needed if loading the environment script in a different directory.



Highlight the source directory to view the contents in the Directory Contents pane on the right. Selecting the checkbox next to the directory includes all the files in the directory. When a directory is checked, it is added to the allowlist. When a user's source repository changes (for example, a file is added or removed from the allowlisted directory) and the user updates the environment or runs the environment script (`.enc`), the Cover environment automatically identifies any changes that have occured in that directory.

Note: To take advantage of the automatic syncing of the files in the Cover environment, it is recommended that you check the topmost directories in your base directory and then denylist those files and directories where you do not want to collect code coverage. This allows the Cover environment to evolve with the source code.

In the Directory Contents pane, use the check boxes next to the source file names to include and exclude files from the environment. Checked source files are allowlisted and included in the environment. Unchecked files are denylisted and excluded from the environment.

The topmost checked paths are added to the allowlist. It is recommended to allowlist those parent directories that contain source code. Subdirectories and files can be unchecked. Any unchecked items are denylisted from the Cover environment.

```
clicast -lc -e <env> Cover Base_dir ALlowlist <name> <path>

The given path will be included in the base directory's allowlist. By
adding paths to the allowlist, the cover environment can compare its state
```

```
with state of the source tree and make any needed updates.
```

```
clicast -lc -e <env> Cover Base_dir Denylist <name> <path>

Denylisted files and directories will be excluded from the cover
environment on update.
```

A file or directory is included in the allowlist if and only if it is:

- Allowlisted explicitly, or
- Not explicitly denylisted and its parent directory is in the allowlist.

To include header files in the environment, check the **Show Header Files** box at the lower left corner of the window. When checked, header files are listed alongside the source files in the Directory Contents pane. Checked headers are used for code coverage. Note that checking this also sets the VCAST_COVERAGE_FOR_HEADERS option.

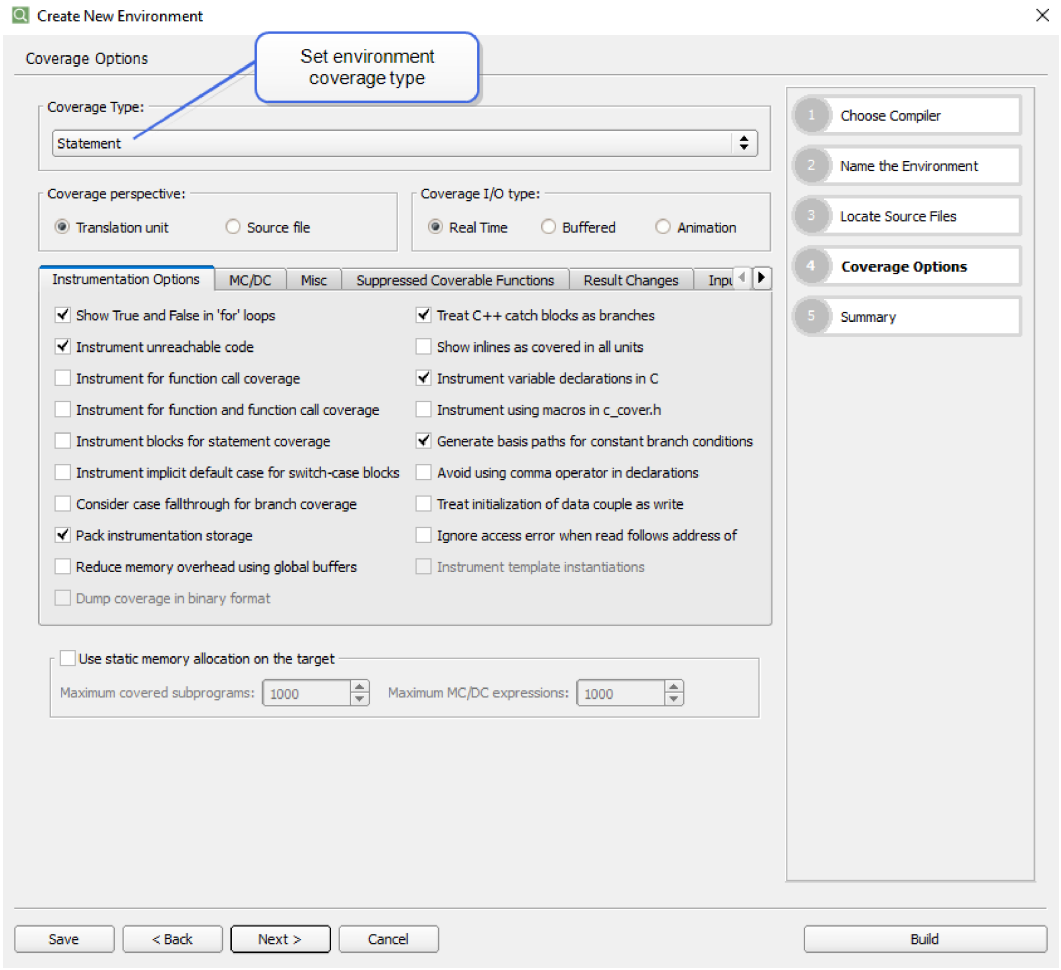
Click **Next** to proceed to Step 4: Coverage Options.

Step 4: Coverage Options

This page of the Cover wizard has a subset of the coverage options from the Options dialog, Coverage tab.

On this step you perform the following:

- Specify the default value of coverage type to set for the environment immediately after building it (optional). When setting this option, the environment coverage type is set, but no files are instrumented when the environment is built. Note that coverage type None is not allowed. The default coverage type is Statement.
- Specify the coverage perspective (optional). The default perspective is Translation Unit.
- Specify the coverage I/O type (optional). The default I/O type is Real Time.
- Specify the instrumentation directory (optional). The default is **`vc-inst`**.
- Change instrumentation options or static memory options (optional).

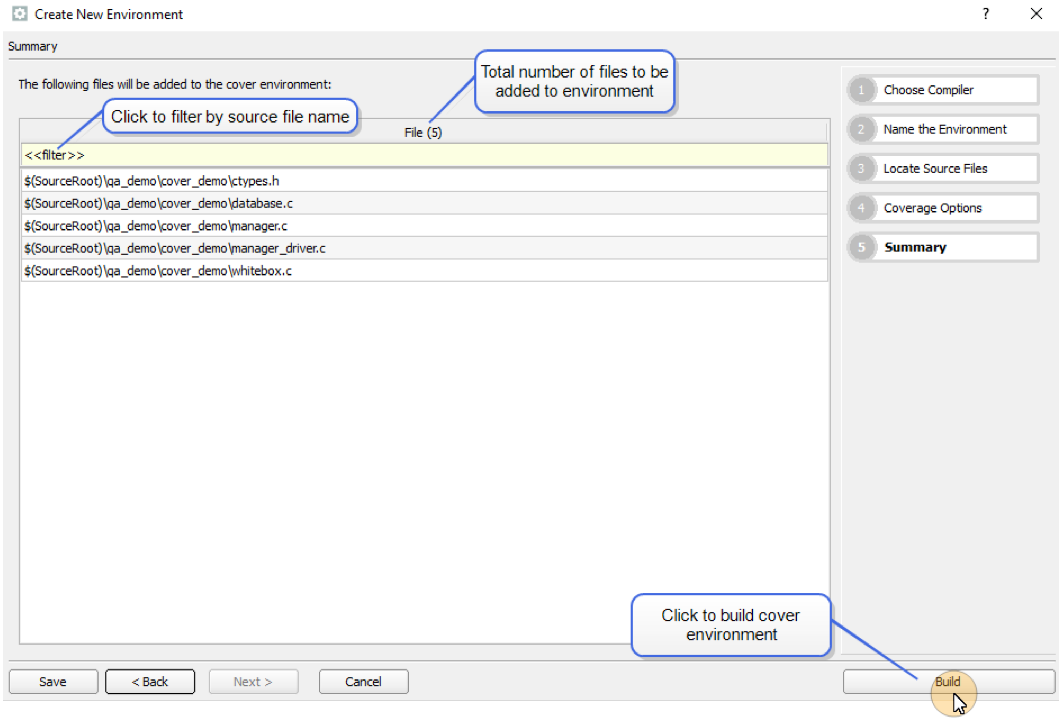


There are no required actions on this page.

Click **Next** to view a Summary of the environment source files.

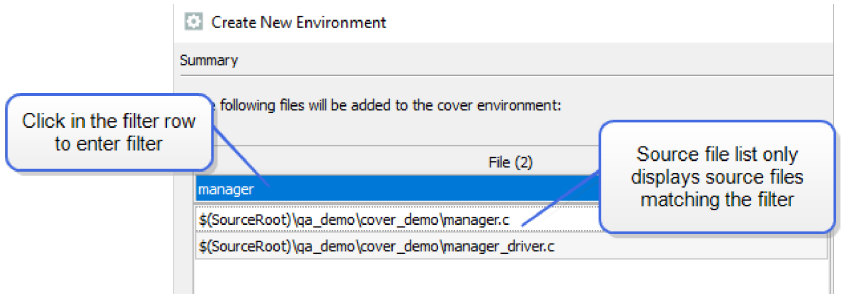
Step 5: Summary

This page of the Cover wizard shows a summary of the files to be added to the Cover environment, based on the selections made on the Locate Source Files window.




```
clicast -lc -e <env> Cover Base_dir STatus

Show files that will be added or removed on update.
```

Filtering is available to locate source files of interest. Filter by clicking in the filter row and typing into the row. In the example below, the list has been filtered to only show source files containing "manager" in the file name.



To clear the filter, highlight the filter row, right-click and select **Clear Filter** from the context menu.

There are no required actions on this page.

Click **Build** to create the Cover environment. No coverage instrumentation is performed.

```
clicast -lc -e <env> Cover Base_dir Update

Any newly added source files within the base directory will be added to the
cover environment. Any source files that no longer exist will be removed
from the cover environment.
```

Once the Cover environment is built, you are able to use the environment as you normally would to set options, instrument, add or remove source files, etc.

Create a Regression Script for the Environment

The **Environment => Create Regression Script...** command creates a regression script in the directory you choose. Note that regression scripts are not needed for migrated Cover environments. The file created is a batch file or a shell script (depending in your platform):

- environment_name.bat
- environment_name.sh

The command also creates a Cover environment script (.enc) in XML format. The .enc file contains source file information as well as any Probe Points n the environment. The .enc file is called by the .bat/.csh file.

If you have imported coverage results present in the environment, an additional file is created:

- *environment_name.cvr*, which includes the imported coverage results themselves

This command enables you to generate the minimum set of files that allow for the regeneration of the tool options, test environment, and all source files. The test environment directory does not need to be maintained when you are not actively testing.

These regression files contain everything you need to recreate the test environment. You simply run the batch file or shell script and VectorCAST performs the following:

- creates the Cover environment
- sets options
- sets the instrumentation directory if it is different than the default (the environment directory)
- adds the source files
- set any unit-specific compilation options (C/C++)
- instruments the source files

The regression script does not add test results.

cover_demo.bat (Windows)

```
del commands.tmp

echo options ASM_FUNCS_BEHAVE_AS_INLINES FALSE >> commands.tmp

echo options ASSEMBLER_CMD >> commands.tmp

echo options COMREADER_ENABLED FALSE >> commands.tmp

echo options C_ALT_COMPILE_CMD >> commands.tmp
```

```
    echo options C_ALT_EDG_FLAGS >> commands.tmp
    echo options C_ALT_PREPROCESS_CMD >> commands.tmp
    echo options C_COMPILER_CFG_SOURCE BUILT_IN_TAG >> commands.tmp
    echo options C_COMPILER_FAMILY_NAME >> commands.tmp
    echo options C_COMPILER_HIERARCHY_STRING VectorCAST MinGW_C >>
    commands.tmp
    echo options C_COMPILER_PY_ARGS >> commands.tmp
    echo options C_COMPILER_TAG BUILTIN_MINGW_63_C >> commands.tmp
    echo options C_COMPILER_VERSION_CMD gcc --version >> commands.tmp
    echo options C_COMPILE_CMD gcc -c -g >> commands.tmp
    echo options C_COMPILE_CMD_FLAG -c >> commands.tmp
    echo options C_COMPILE_EXCLUDE_FLAGS -o** >> commands.tmp
    echo options C_DEBUG_CMD gdb >> commands.tmp
    echo options C_DEBUG_HELP_FILE >> commands.tmp
    echo options C_DEFINE_FLAG -D >> commands.tmp
    echo options C_DEFINE_LIST >> commands.tmp
    echo options C_EDG_FLAGS -w --gcc --gnu_version 60300 >> commands.tmp
    echo options C_EXECUTE_CMD >> commands.tmp
    echo options C_EXEC_HELP_FILE >> commands.tmp
    echo options C_INCLUDE_FLAG -I >> commands.tmp
    echo options C_LINKER_VERSION_CMD ld --version >> commands.tmp
    echo options C_LINK_CMD gcc -g >> commands.tmp
    echo options C_LINK_OPTIONS >> commands.tmp
    echo options C_OBJECT_EXT .o >> commands.tmp
    echo options C_OUTPUT_FLAG -o >> commands.tmp
    echo options C_PREPROCESS_CMD gcc -E -C >> commands.tmp
    echo options C_PREPROCESS_FILE >> commands.tmp
    echo options EXECUTABLE_EXTENSION >> commands.tmp
    echo options MAX_VARY_RANGE 20 >> commands.tmp
    echo options PRECOMPILE_CMD >> commands.tmp
    echo options PRECOMPILE_EXT >> commands.tmp
    echo options SBF_LOC_MEMBER_IN_NSP DECL_NAMESPACE >> commands.tmp
    echo options SBF_LOC_MEMBER_OUTSIDE_NSP DECL_NAMESPACE >> commands.tmp
```

```
    echo options SBF_LOC_NONMEMBER_IN_NSP DECL_NAMESPACE >> commands.tmp
    echo options SBF_LOC_NONMEMBER_OUTSIDE_NSP DECL_NAMESPACE >>
    commands.tmp
    echo options SOURCE_EXTENSION .c >> commands.tmp
    echo options SUBSTITUTE_CODE_FOR_C_FILE FALSE >> commands.tmp
    echo options VCAST_ALT_WB_METHOD FALSE >> commands.tmp
    echo options VCAST_ASSEMBLY_FILE_EXTENSIONS s >> commands.tmp
    echo options VCAST_ASSIGN_WITHOUT_COPY_CTOR TRUE >> commands.tmp
    echo options VCAST_AUTO_CLEAR_TEST_USER_CODE FALSE >> commands.tmp
    echo options VCAST_AVOID_COMMA_OPERATOR FALSE >> commands.tmp
    echo options VCAST_BUFFER_OUTPUT FALSE >> commands.tmp
    echo options VCAST_COLLAPSE_STD_HEADERS COLLAPSE_SYSTEM_HEADERS >>
    commands.tmp
    echo options VCAST_COMMAND_LINE_DEBUGGER TRUE >> commands.tmp
    echo options VCAST_COMPILER_SUPPORTS_CPP_CASTS FALSE >> commands.tmp
    echo options VCAST_COVERAGE_FOR_AGGREGATE_INIT TRUE >> commands.tmp
    echo options VCAST_DISABLE_CPP_EXCEPTIONS FALSE >> commands.tmp
    echo options VCAST_DISABLE_STD_STRING_DETECTION FALSE >> commands.tmp
    echo options VCAST_DISABLE_STD_WSTRING_DETECTION TRUE >> commands.tmp
    echo options VCAST_DISABLE_TI_BITFIELD FALSE >> commands.tmp
    echo options VCAST_DISABLE_TI_STRING FALSE >> commands.tmp
    echo options VCAST_DISPLAY_UNINST_EXPR FALSE >> commands.tmp
    echo options VCAST_DUMP_BUFFER FALSE >> commands.tmp
    echo options VCAST_ENABLE_FUNCTION_CALL_COVERAGE FALSE >> commands.tmp
    echo options VCAST_ENVIRONMENT_FILES >> commands.tmp
    echo options VCAST_ESCAPE_LINE_DIRECTIVES FALSE >> commands.tmp
    echo options VCAST_EXECUTE_WITH_STDIO FALSE >> commands.tmp
    echo options VCAST_EXECUTE_WITH_STDOUT FALSE >> commands.tmp
    echo options VCAST_FAR_STDIN_DATA FALSE >> commands.tmp
    echo options VCAST_FILE_INDEX FALSE >> commands.tmp
    echo options VCAST_FILE_PREFIX >> commands.tmp
    echo options VCAST_FORCE_ELAB_TYPE_SPEC TRUE >> commands.tmp
    echo options VCAST_FORCE_NO_USERGLOBALS FALSE >> commands.tmp
    echo options VCAST_GH_INTEX_CMD >> commands.tmp
```

```
    echo options VCAST_GH_INT_FILE >> commands.tmp
    echo options VCAST_HAS_LONGLONG TRUE >> commands.tmp
    echo options VCAST_INST_FILE_MAX_LINES 0 >> commands.tmp
    echo options VCAST_MAIN FALSE >> commands.tmp
    echo options VCAST_MAX_COVERED_SUBPROGRAMS 1000 >> commands.tmp
    echo options VCAST_MAX_HEAP_SIZE 1000 >> commands.tmp
    echo options VCAST_MAX_MCDC_STATEMENTS 1000 >> commands.tmp
    echo options VCAST_MAX_STRING_LENGTH 1000 >> commands.tmp
    echo options VCAST_MICROSOFT_LONG_LONG FALSE >> commands.tmp
    echo options VCAST_MINIMAL_TERMINATION FALSE >> commands.tmp
    echo options VCAST_NO_EXIT FALSE >> commands.tmp
    echo options VCAST_NO_FFLUSH FALSE >> commands.tmp
    echo options VCAST_NO_FLOAT FALSE >> commands.tmp
    echo options VCAST_NO_LIMITS FALSE >> commands.tmp
    echo options VCAST_NO_LONG_DOUBLE TRUE >> commands.tmp
    echo options VCAST_NO_MALLOC FALSE >> commands.tmp
    echo options VCAST_NO_SETJMP FALSE >> commands.tmp
    echo options VCAST_NO_SIGNAL FALSE >> commands.tmp
    echo options VCAST_NO_STDIN FALSE >> commands.tmp
    echo options VCAST_NO_STDLIB FALSE >> commands.tmp
    echo options VCAST_NO_STD_FILES FALSE >> commands.tmp
    echo options VCAST_OUTPUT_BUFFER_SIZE 20000 >> commands.tmp
    echo options VCAST_PREPEND_TO_PATH_DIRS $(VECTORCAST_DIR)/MinGW/bin >>
    commands.tmp
    echo options VCAST_PREPROCESS_PREINCLUDE >> commands.tmp
    echo options VCAST_REMOVE_PREPROCESSOR_COMMENTS TRUE >> commands.tmp
    echo options VCAST_STDIO FALSE >> commands.tmp
    echo options VCAST_TYPEOF_OPERATOR TRUE >> commands.tmp
    echo options VCAST_USE_COMPOUND_FOR_BATCH FALSE >> commands.tmp
    echo options VCAST_USE_EDG_PREPROCESSOR FALSE >> commands.tmp
    echo options VCAST_USE_STATIC_MEMORY FALSE >> commands.tmp
    echo options VCAST_USE_STD_STRING TRUE >> commands.tmp
    echo options VCAST_USE_VCPP FALSE >> commands.tmp
```

```
echo options VCAST_VXWORKS FALSE >> commands.tmp

echo options WHITEBOX YES >> commands.tmp

echo cover environment script_run C:/VCAST/Examples/environments/qa_
demo/cover_demo.enc >> commands.tmp

"%VECTORCAST_DIR%\CLICAST" /L:C tools execute commands.tmp false
```

To Use a JSON Compilation Database to Build an Environment

An alternative to the vcshell database is to use a JSON compilation database. This integration improves interoperability with other software development tools and reduces friction when incorporating VectorCAST into toolchains that rely on JSON compilation databases. Additionally, because tools such as CMake can generate these databases without requiring a full compilation, this support helps reduce workflow overhead. Not all VectorCAST workflows are compatible with this database. For instance, using environments with the VectorCAST Project workflow is not fully supported. See more in the section "Cautions" on page 80.

The JSON compilation database follows the Clang specification.

The JSON Compilation Database Integration

Previously, users that wished to use a compilation database were required to record a full compile with the vcshell executable. The resulting vcshell database (vcshell.db) contained compilation information that could be used to build environments.

The addition of support for an alternate database provides users with the choice to specify a JSON Compilation Database in place of a vcshell database.

Two configuration options are used to select the JSON compilation database:

- Options Rules: Instructs VectorCAST how to use compilation arguments captured in the compilation database.
- Transform Pass: (optional) A python script that allows users to perform transformations on the data read from the compilation database.

How To Use the JSON Compilation Database Integration

Here are the steps for the VectorCAST Coverage GUI.

- Go to “Name” page of the Wizard to find the “Compilation Database” group box.
- Provide a path for a JSON compilation database for the “Database” field by either typing it in the textbox or using the file selector.
- The tab labelled “JSON” will be active and enabled so that users can configure the “Options Rules” and “Transform Pass”.
- Options Rules: Unless overwritten by a value in the CFG or from a loaded ENV, the default values will be displayed in the text box. Users can provide any value according to the Options Rules Syntax described below.
- Transform Pass: (optional) A path to a Python script that implements and registers a TransformPass object. Examples are provided at: “$VECTORCAST_

DIR/python/vector/lib/compdb/examples”

Here are the steps for the VectorCAST command line.

Use the following options in the VectorCAST CFG file:

- VCDB_FILENAME
- VCAST_COMPDB_OPTIONS_RULES
- VCAST_COMPDB_TRANSFORM_PASS

Cautions

JSON compilation databases are not currently supported for all workflows. The following executables and scripts do not have support:

- $VECTORCAST_DIR/vcutil
- $VECTORCAST_DIR/vcdb
- $VECTORCAST_DIR/python/vector/apps/vcshell/EnvCreate.py
- $VECTORCAST_DIR/python/vector/apps/AutomationController/AutomationController.py

The following executable has limited support:

$VECTORCAST_DIR/manage

- Supports monitoring and migrating environments built with a JSON compilation database.
- Does not support creating new environments with a JSON compilation database.

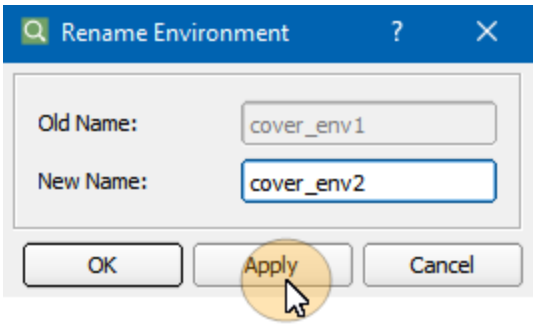
Working With Environments

To Close an Environment

The **File => Close Environment** command closes the environment and any open files in the MDI window. If any file is unsaved, you are prompted to save before it closes.

To Rename an Environment

Choose **File => Rename Environment** to give the current open environment a new name. A dialog appears with the current name at the top and the new name is at the bottom. If the new name is not unique, then it is outlined in red. Type a new name and click **OK** or **Apply**.



Once you click **OK** or **Apply**, VectorCAST renames the environment directory and the .vce file.



Note: The environment script is not changed when an environment is renamed.

To Update an Environment

From the Update Environment wizard, you can:

- Update your compiler and compiler settings
- Update the Base Directories
- Modify allowlisted files and directories
- View a Summary of source files included in the environment and their status (existing, removed, added)

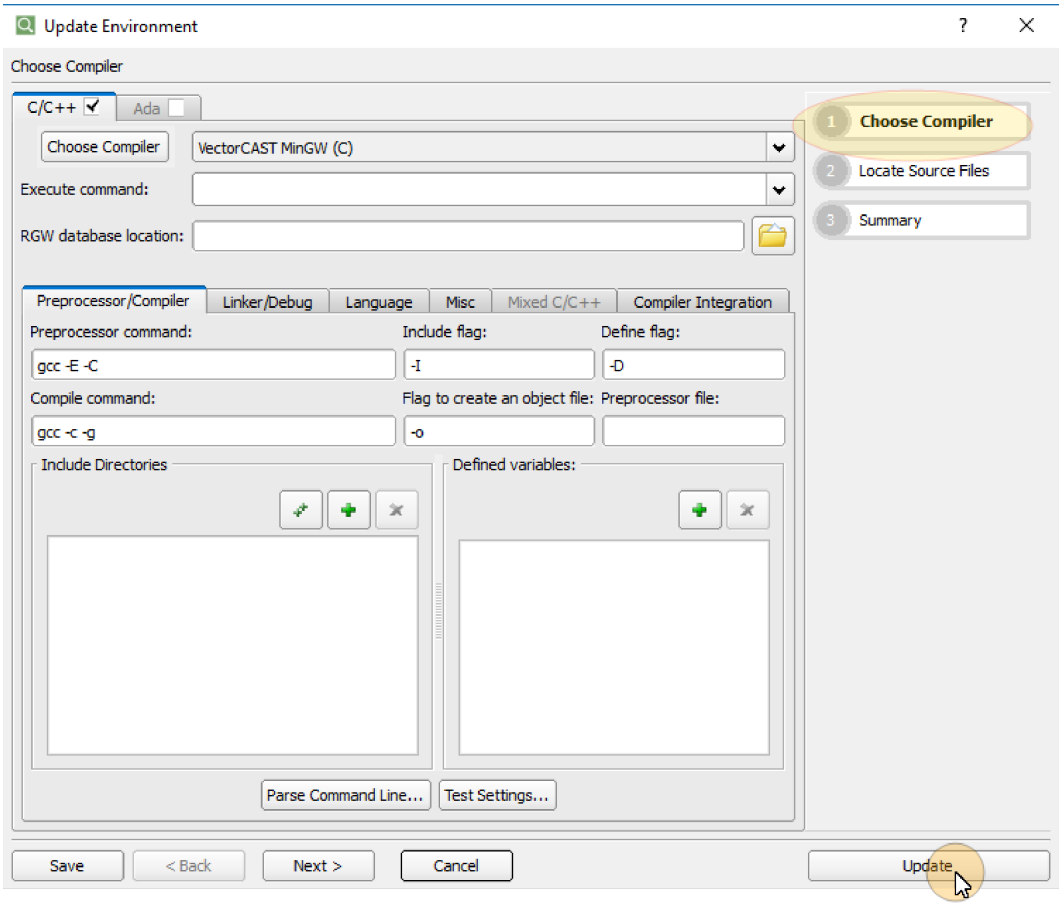
Choose **Environment => Update Environment...** from the Menu Bar to open the Update Environment Wizard.



Note: Base Directories cannot be added, removed, or modified when the environment is opened within a VectorCAST project and the Update Environment wizard is requested. However, the Allow/Deny list can be updated.

Step 1: Choose Compiler

By default, the Update Environment wizard opens on Step 2: Locate Source Files. Selecting Step 1: Choose Compiler allows you to update the compiler and compiler settings.

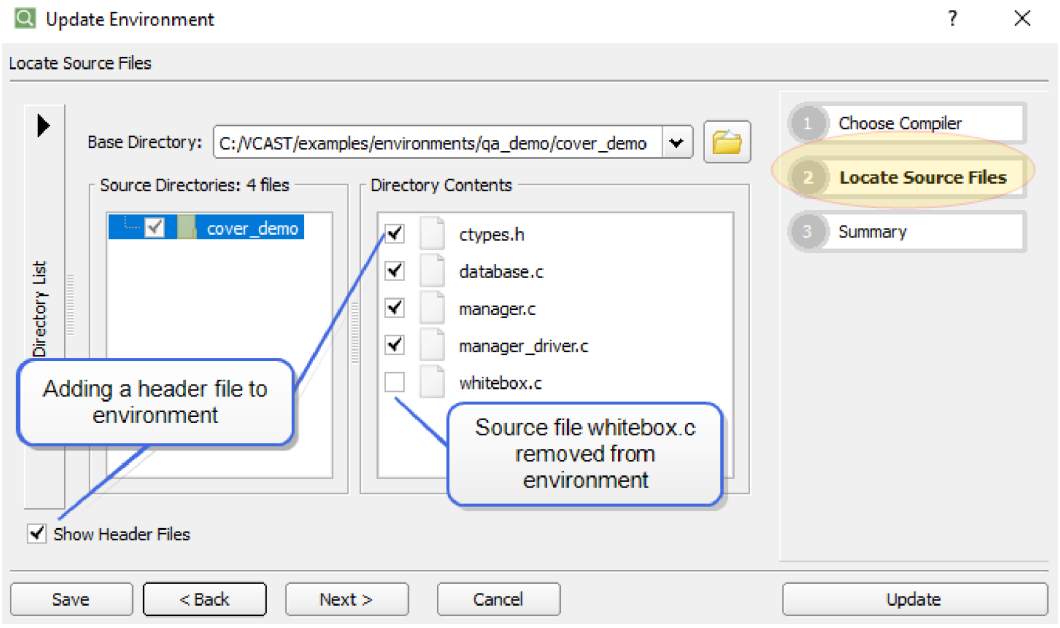


For more detailed information on compiler settings, see "Step 1: Choose Compiler" on page 57 in the "Creating a System Testing Environment" section.

Step 2: Locate Source Files

By default, the Update Environment wizard opens on Step 2: Locate Source Files. Any sources detected in the selected directory are displayed. Select or deselect directories and source files to include or exclude them from the environment.

```
clicast -lc -e <env> Cover Base_dir Status

Show files that will be added or removed on update.
```



A file or directory is included in the allowlist if and only if it is:

- Allowlisted explicitly, or
- Not explicitly denylisted and its parent directory is in the allowlist.

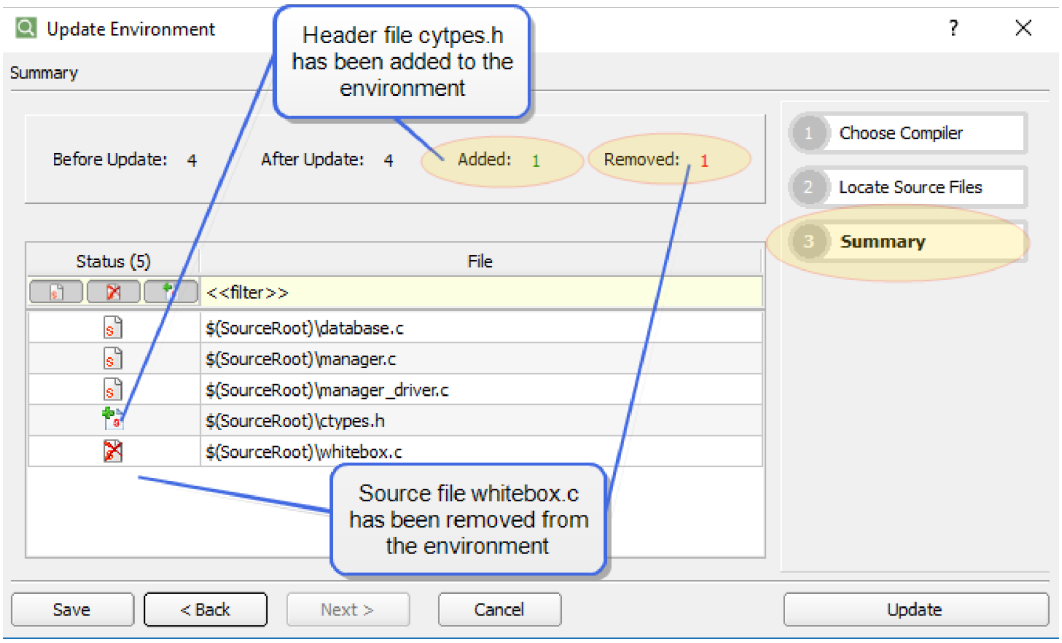
Note: If the Cover environment was created without a base directory but contains source files, a base directory will automatically be created for the user.

For more detailed information on locating source files, see "Step 3: Locate Source Files" on page 68 in the "Creating a System Testing Environment" section.

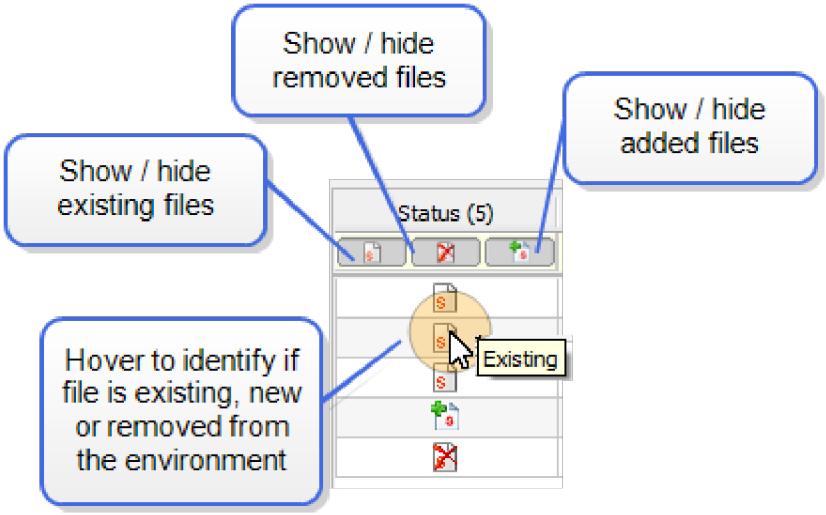
Step 3: Summary

To view how your source code repository differs from the sources within a Cover environment, select Step 3: Summary. The Summary provides the following information:

- Before Update - the number of files currently in the environment
- After Update - the new number of files that will be in the environment following update
- Added - the number of files added to the environment
- Removed - the number of files removed from the environment



A Status column is provided which indicates the status of each file in the environment (Existing, Removed or Added). Hovering over a status icon will indicate whether a file is newly added, retained or removed from the environment.



Use the toggle buttons located at the top of the Status column to show / hide the file types. By default, all toggle buttons are selected. The Status column heading includes the total number of files being displayed.

To compute status, the set of files on disk are collected by walking over the directories and files given in the allowlist. Denylisted items are excluded. The status is determined as follows:

> Existing files are those that have been found on disk and that are currently in the Cover environment.

> Removed files are those that are in the Cover environment, but not on disk.

> Added files are those that are on disk but not in the Cover environment.

```
clicast -lc -e <env> Cover Base_dir STatus

Show files that will be added or removed on update.
```

When all updates are complete, select the **Update** button to update the environment.

```
clicast -lc -e <env> Cover Base_dir Update

Any newly added source files within the base directory will be added to the
Cover environment. Any source files that no longer exist will be removed
from the Cover environment.
```

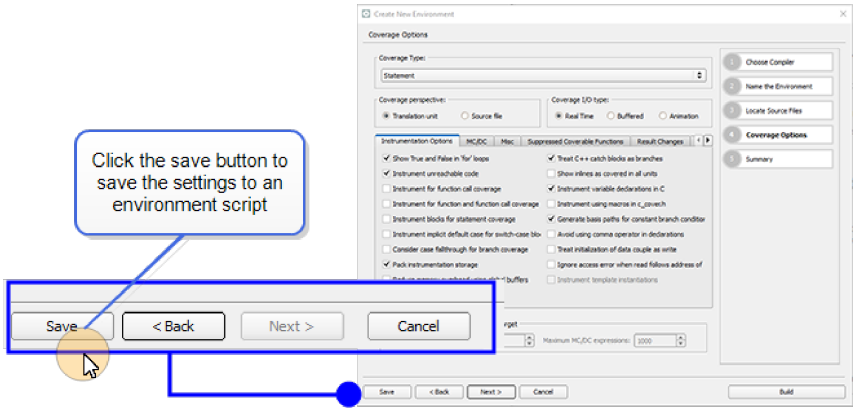
Note: Newly added sources will not be instrumented when the **Update** button is selected. To instrument newly added sources for the default coverage type, perform a build within the VectorCAST project.

To Create a Cover Environment Script

A Cover Environment script (.enc) in XML format is created for Cover and System Testing environments. The file contains source file information and any Probe Points in the environment.

The Cover Environment script is not created automatically after an environment build. Use one of the following methods to create the script:

> Click the **Save** button in the Create New Cover Environment wizard.



> Create a regression script by selecting The **Environment => Create Regression Script...** from the Menu Bar. The Cover Environment script is automatically created when creating regression scripts.

```
clicast -e <COVER_ENV> Cover ENvironment SCRIPT_Create <FILENAME>.enc
[force]

Use the optional 'force' argument to overwrite an existing .enc file.
```

To create a new Cover environment based on an existing .enc file:

> Load an .enc file from within the Create New Cover Environment wizard. Click the **Load** button and select the appropriate .enc file.

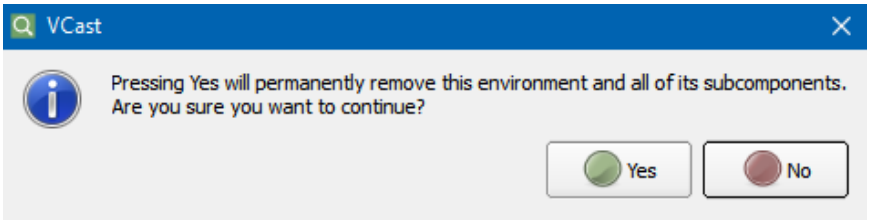
```
clicast -e <COVER_ENV> Cover ENvironment SCRIPT_Run <scriptfile> [force]
[verbose]

Create cover environment from an enc script. Use the 'force' argument to
recreate the environment. Use the 'verbose' argument to display success
messages.
```

To Delete an Environment

The **File => Delete Environment** command deletes the current open environment or enables you to select an environment to delete. Deleting an environment deletes the sub-directory that was created for that environment and all files contained in the sub-directory. It leaves the .env file that can be used to re-create the environment later. Note that the test cases are not saved for you. To avoid losing any work, you should create Regression Scripts before deleting.

Note: The directory created by VectorCAST when building a new environment should only contain VectorCAST-created files. Do not store any files that are not tool-specific in these directories. When the **File => Delete Environment** command is invoked, all files contained in the specified directory are deleted.

If you wish to delete the current open environment, choose **Environment =>Delete Environment**. You are asked to confirm deleting the open environment.



If no environment is open when you choose **File => Delete Environment**, the standard Choose File dialog appears, providing the opportunity to navigate to find any environment to delete. Locate the .vce or .vcp file for the environment you want to delete. Click the **Open** button.

```
clicast -e <env> ENvironment Delete

Delete the specified environment.
```

To Open an Environment

There are several ways to open an environment.

- Click the **Open** button  in the toolbar or use **File => Open** if you need to navigate to find the environment you want.
- Choose **File => Recent Environments** to open a recently-opened environment from a list. The most recently-opened environment is at the top of the list. Coverage environments are shown with a green icon  ; Ada environments with an orange icon  ; and C environments with a blue icon  .
- Double-click the environment's icon (Windows)
  To use this method, the **VECTORCAST_DIR** environment variable must be set at the system level. The Windows installation program does this for you.
- Use the command line:

  ```
  $VECTORCAST_DIR/vcastqt -e <env> (Linux)
  %VECTORCAST_DIR%\vcastqt -e <env> (Windows)
  ```

  To use this method, the **VECTORCAST_DIR** environment variable must be set in the shell window.

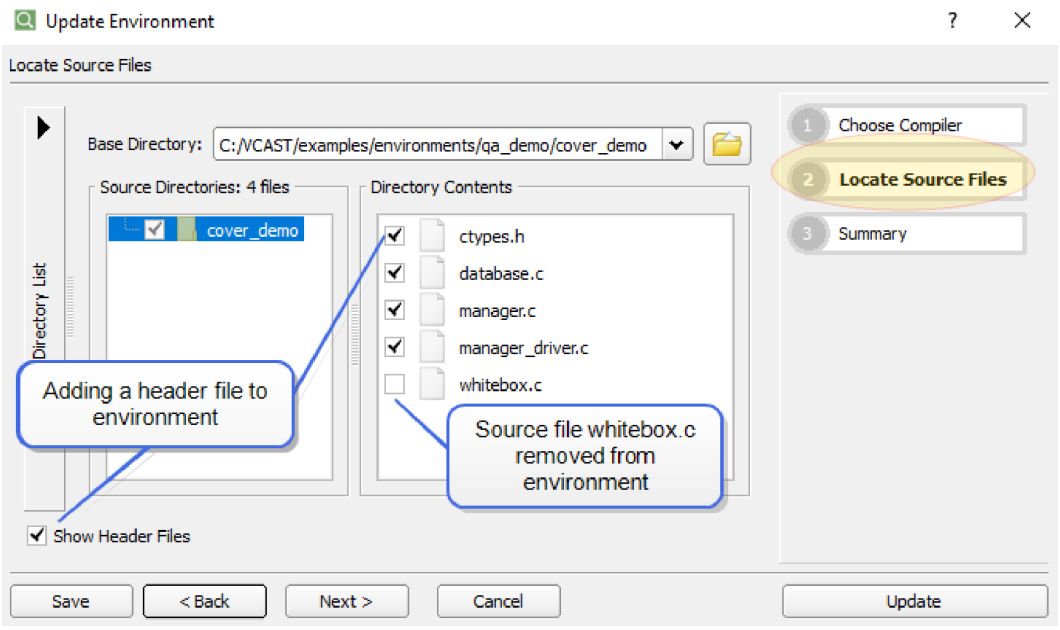
Once you open an environment, the working directory is set to the location of the newly-opened environment.

Because VectorCAST/QA writes data to the .vcp file on closing the environment, you are restricted from opening a System Testing environment that is write-protected.

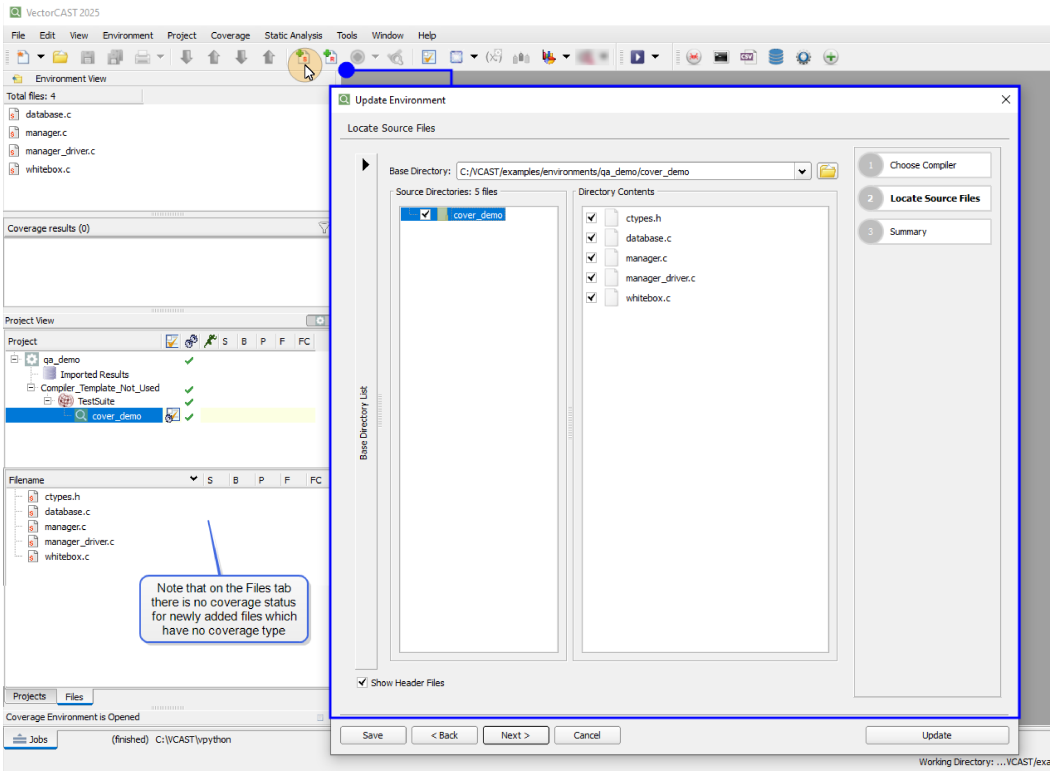
Adding Source Files to the Environment

To Add a Source Code File

Select **Environment => Add Source Files** from the Help Menu to open the Update Environment wizard. Alternatively, select the Add Source File icon  on the Toolbar to open the Update Environment wizard. This enables you to choose which file(s) in your application are to be instrumented for code coverage.



Once source files have been added to a Coverage environment, they are listed in the Source Files pane. The columns to the right of the filename column provide coverage status information for Statement (**S**), Branch (**B**), MC/DC Pairs (**P**), Function (**F**), and Function and Call (**FC**) coverage.These columns are currently empty because no coverage type has been selected yet.



The name and path to each source file is saved in the *envname*.vcp file, along with the location of its

instrumented version. Each unit is assigned a unit number, starting with 1 by default.

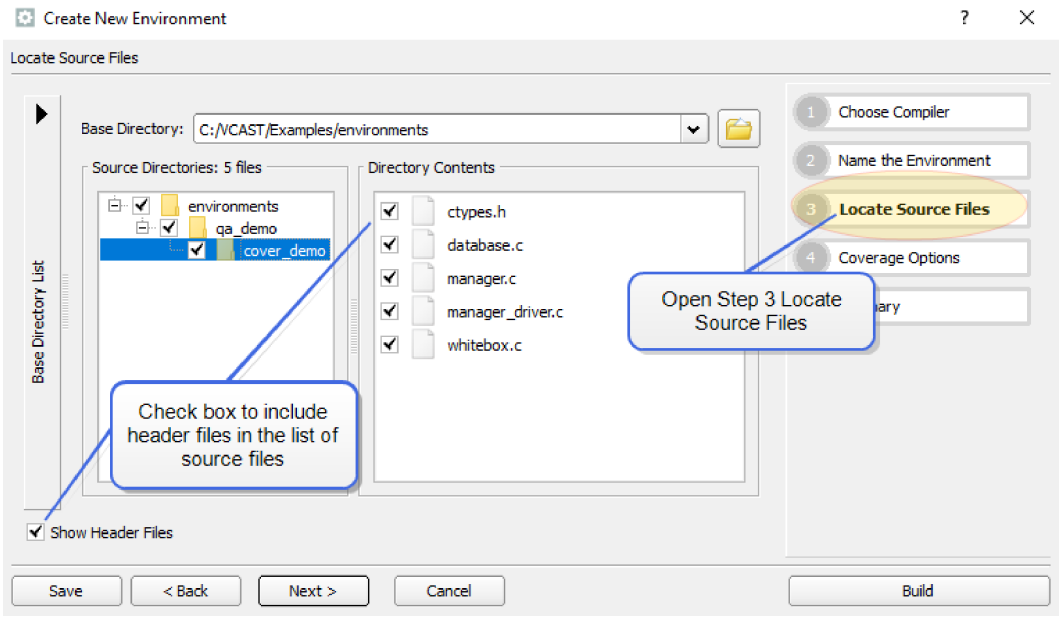
Source files with the same name from different locations may be added to a Cover environment.

For more detailed information on locating source files, see "Step 3: Locate Source Files" on page 68 in the "Creating a System Testing Environment" section.

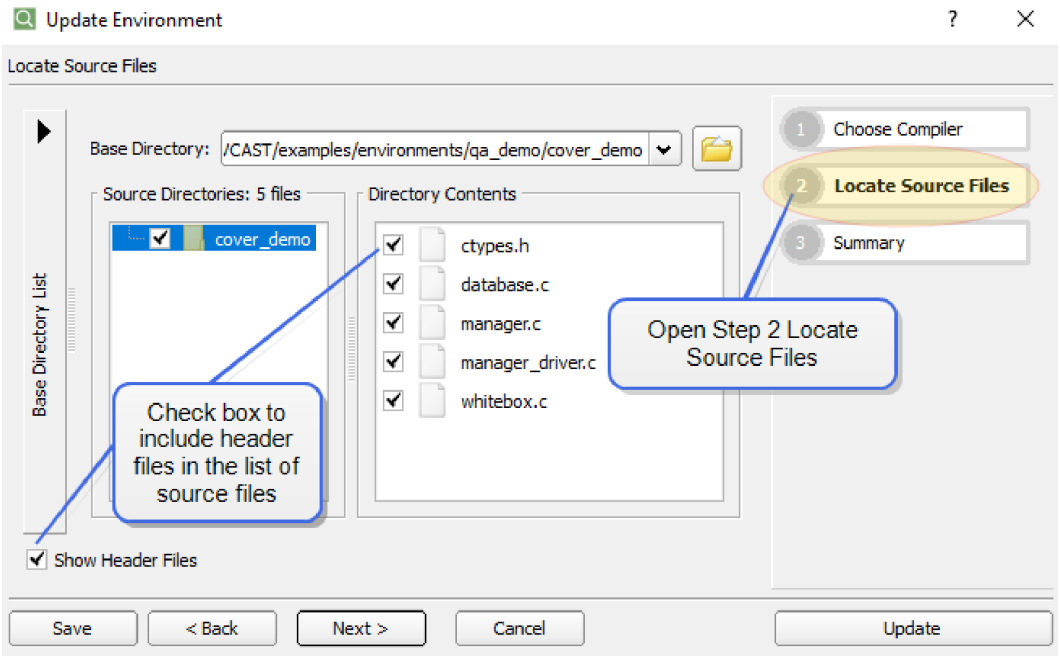
To Add Header Files as Source Files (C/C++ only)

Achieving code coverage on header files which have executable lines, such as inlined functions, is possible with VectorCAST Cover.

To include header files when creating a new environment, go to Step 3 Locate Source Files in the Create New Environment wizard. Check the **Show Header Files** box at the lower left corner of the window. When checked, header files are listed alongside the source files in the Directory Contents pane. Checked headers are used for code coverage. Note that checking this also sets the VCAST_COVERAGE_FOR_HEADERS option.



To include header files when updating an existing environment, go to Step 2 Locate Source Files in the Update Environment wizard. Check the **Show Header Files** box at the lower left corner of the window. When checked, header files are listed alongside the source files in the Directory Contents pane. Checked headers are used for code coverage. Note that checking this also sets the VCAST_COVERAGE_FOR_HEADERS option.

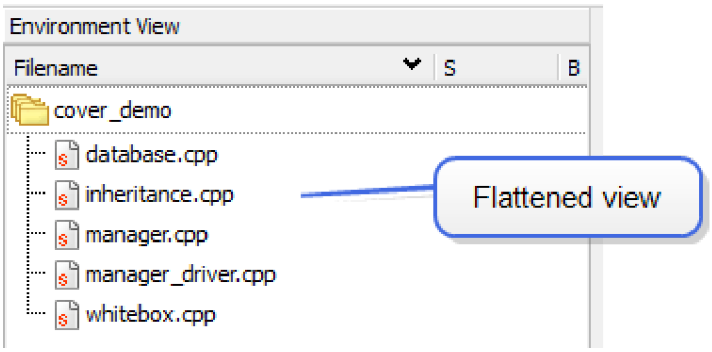


After instrumentation, the Coverage Viewer of a source unit expands lines where the header file was #included to show the full text of the header file. The header files are not listed in the Source Files Pane as source code units.

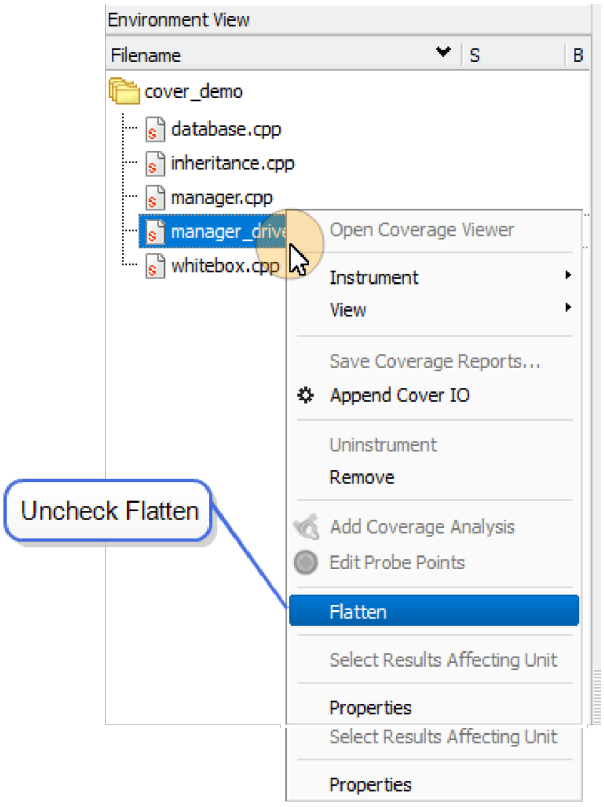
Only header files located in the same directory as a source unit and header files located in Search directories are expanded so as to achieve code coverage. (Header files in Library Include directories or Type-handle-only directories are not expanded so as to achieve code coverage.)

To View Source Files in a Hierarchy

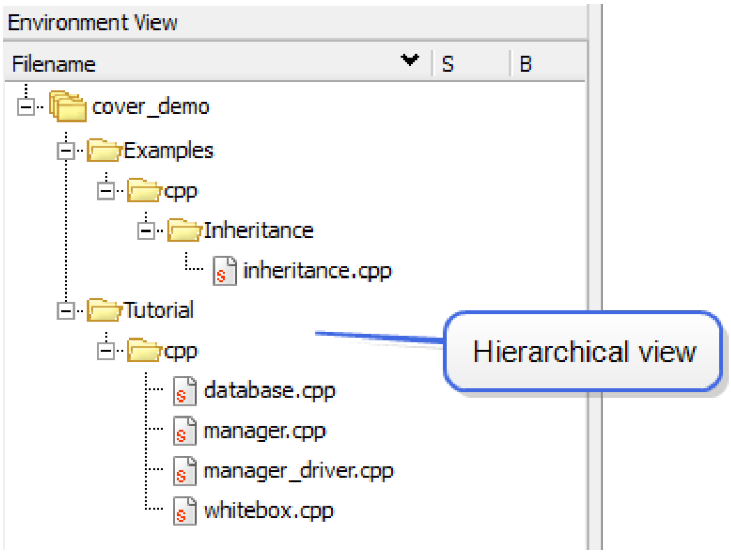
When source files are first added to the environment, they are listed alphabetically, at the same level, in the Source Files pane. This single-level display is called “flattened.”



To “unflatten” the display, right-click on any source file in the Source Files pane, and uncheck **Flatten**.



When you uncheck “Flatten,” the Source Files pane changes to a hierarchical representation of the location of the source files in the file system.



To View Source Code

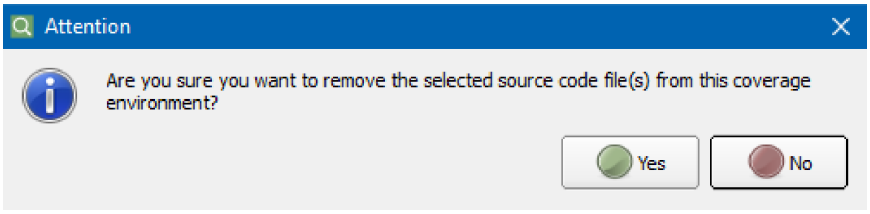
You can view the source code for any file in the Source Files pane. Right-click a source file, and

choose **View => Original Source Code**. A text file opens in a Text Editor, in read-only mode.

Note: The source code can be modified by first clicking the “Writable” checkbox at the bottom of the Text Editor. Once modified and saved, the source file must be re-instrumented and re-compiled for the changes to take effect.

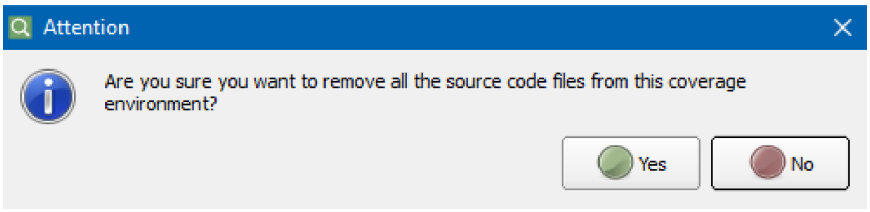
To Remove a Source File

To remove a source file from the environment, right-click the source file and choose **Remove**. A confirm dialog appears. Click **Yes** to remove the file; click **No** to cancel the operation.



To Remove All Source Files

The **Environment => Remove All Source Files** enables you to remove all source code files from the environment. Click Yes to confirm the deletion.



Working With Base Directories

VectorCAST allows you to enables you to specify a "base directory," or source root directory, of source files to be added to the Cover environment. The path to the base directory is saved in the environment script (`.enc`) which means only a single change is needed if loading the environment script in a different directory.

The following sections discuss common scenarios associated with working with base directories.

For information on suppressing test code coverage, see "Suppressing Test Code Coverage" on page 218.

Moving Base Directories

When using a Continuous Integration (CI) or Continuous Testing (CT) workflow, the source code repository is often stored separately or moved away from the Cover environment that tests it.

In cases where a source code repository is identified as missing, VectorCAST supports the user in opening such an environment and generating reports, but instrumenting and other features are not

available.



Note: When in Source File perspective, Aggregate Coverage reports and the Coverage Viewer are not able to display the source code when the repository is missing.

However, when the source code repository is not missing but just moved to a new location (on a machine of the same platform on which the environment was created), VectorCAST supports updating the Base Directory path.



Note: Moving or copying base directories for Cover environments that are part of a VectorCAST System Test Project is not supported.

The following sections discuss three common scenarios associated with moving base directories.

Base Directory as a Full Path

When the Base Directory was originally specified as a full path to the source code repository which has now been moved, the user needs only to update the Base Directory in the Cover environment to point to the new source code location.

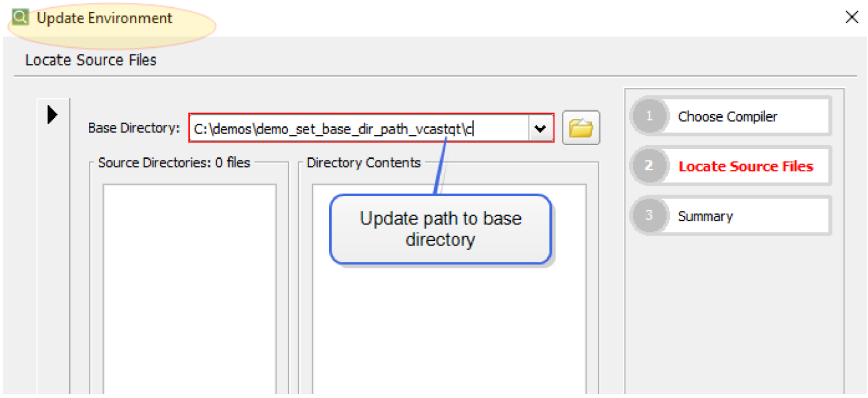
To add a Base Directory:

```
clicast -e <cover env> Cover Base_dir Add <name of base directory> <full
path to repository>

This specifies the location where source files reside.
```

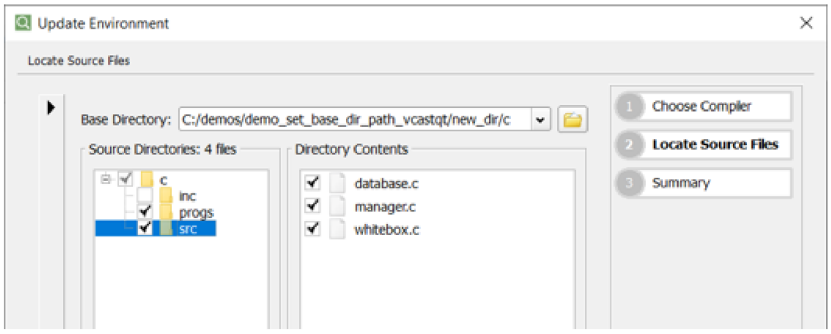
To update the Base Directory path:

```
clicast -e <cover env> Cover Base_dir SEt_path <name of the base directory>
<path to new location>

Change the location where the base directory resides.
```

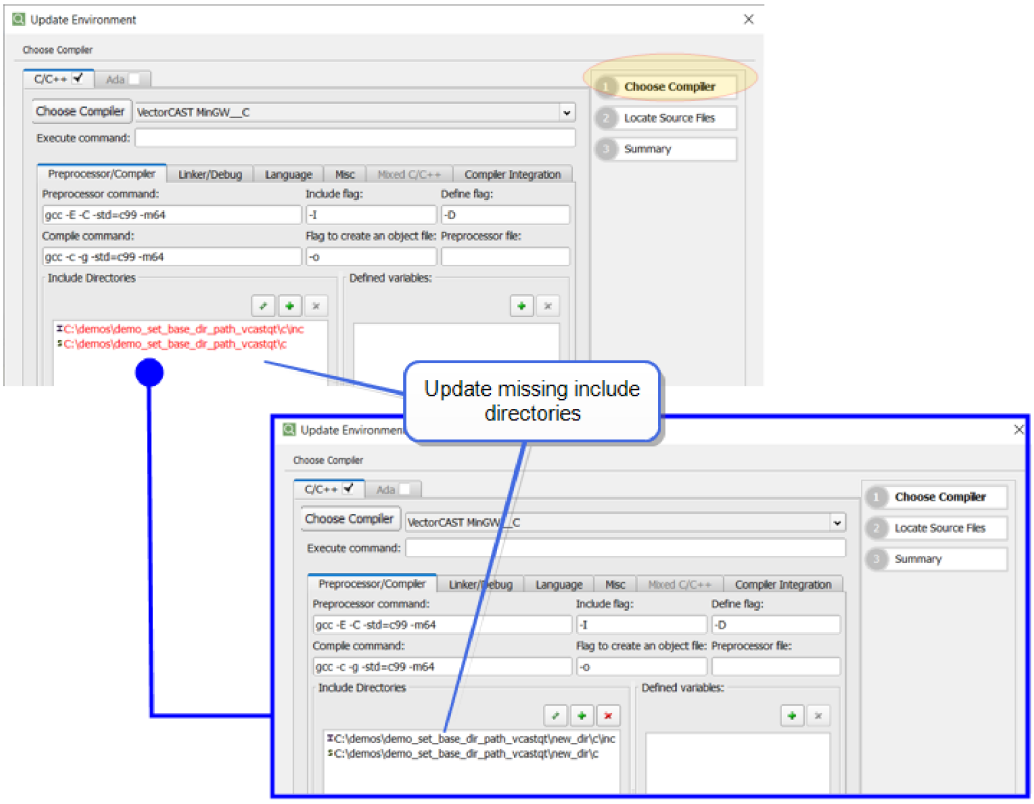
Alternatively, when opening a Cover environment using the VectorCAST GUI after the source code repository was moved, the Update Environment Wizard automatically displays, prompting the user to adjust the Base Directory path in the Wizard.



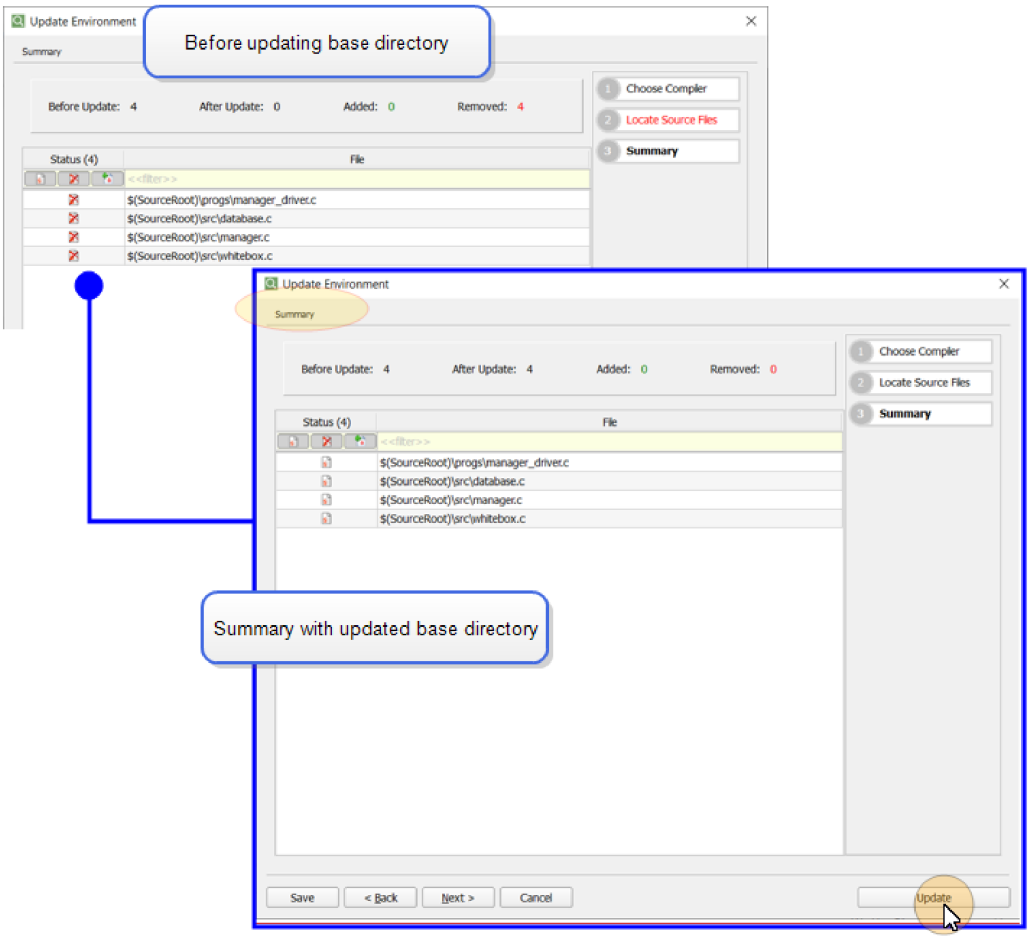
The path should be set to the root of the original Base Directory so that the directory structure is retained. This permits the allowlist, denylist, and source files to be loaded into the Update Environment wizard.



Note: Any missing Include directories located on the Choose Compiler page will also need to be updated to point to their new locations.



Navigate to the Summary page to verify that the source files in the original Base Directory are preserved.



After clicking the Update button, the user can proceed with normal functionality such as generating reports, opening the Coverage Viewer, and instrumenting sources.

Base Directory as an Environment Variable

Some users have set up an environment variable that points to the location of the source code repository under test. After the environment variable is changed to point to a different location, opening the Cover environment is seamless and no further updates need to be made.

```
clicast -e <cover env> Cover Base_dir Add <name of base directory> '$(ENV_
VAR)'

where ENV_VAR is the name of the environment variable that has been set to
the root location of the source repository.
```

For example, suppose the environment variable SOURCE has been set to **'/home/work/source1'** and a Cover environment created with a Base Directory added like this:

```
clicast -e <cover env> Cover Base_dir Add SourceRoot '$(SOURCE)'
```

Later, if the repository has been moved to the directory **'/home/work/source2'**, or simply the

code in the directory `'source2'` is now to be used in the Cover environment, then the environment variable is updated to reflect the new location. In this case, opening the Cover environment is seamless and no further updates need to be made.



Note: The VectorCAST GUI must be closed when the environment variable is set to a new path.

Base Directory as a Relative Path

If the Base Directory was originally added using a path relative to the Cover environment, then moving the source code repository and the Cover environment together is a seamless operation and no updates need to be made.

For example, if the Cover environment is in a directory named `'work'`, and the source code repository in `'work/source'`, then a Base directory named `SourceRoot` can be added like this:

```
clicast -e <cover env> Cover Base_dir Add SourceRoot source
```

When both the environment and source code repository are moved together, the environment can be used without needing to update the Base Directory path.

Setting Coverage

To Set Coverage Type

Any source file added to an environment is set by default to the environment coverage type. Once a source file has been added to an environment, you may then indicate which type of coverage you want, and you may assign a coverage type different from the environment coverage type. Different source files can be set with different types of coverage, if appropriate. For example, code with few branches may be best suited for Statement coverage, while code with many branches may be better suited to Branch coverage.

You can choose from several types of coverage. The coverage type choices are dependent upon the current Industry Mode. The Default industry mode offers the following coverage types:

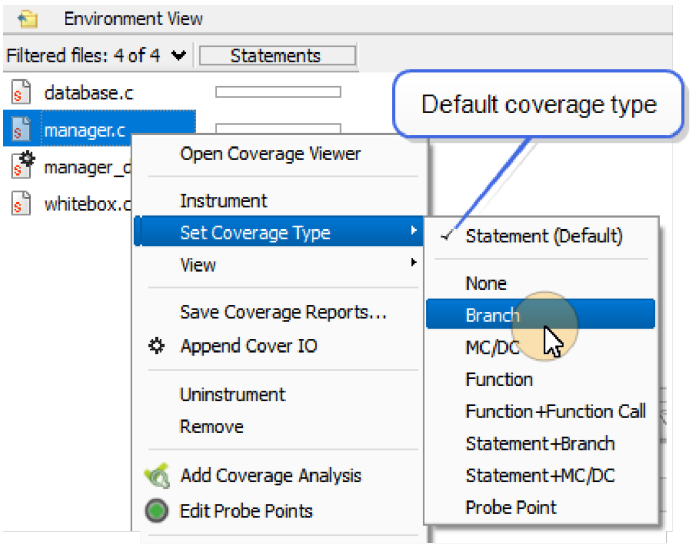
- **Statement** – reports on each executable statement
- **Branch** – reports on the outcome of each decision point
- **Basis Paths** – reports on basis paths through the source code
- **MC/DC** – reports on branch outcomes as well as equivalence pairs
- **Function** – reports on whether or not a subprogram in the unit under test has been entered during test execution
- **Function + Function Call** – identifies all of the calls that each function makes to other functions and reports on whether those calls have been tested
- **Statement + MC/DC** – reports on MC/DC and Statement coverage
- **Statement + Branch** – reports on Branch and Statement coverage
- **Probe Point** – reports on Probe Point-only coverage
- **Coupling** – reports on data and control coupling

The Default Industry coverage types map to industry types as shown in the following table:

Default	Avionics	Automotive	Industrial	Railway	Medical
Statement	Level A Level B Level C	Unit Level ASIL A Unit Level ASIL B/C Unit Level ASIL D	SIL 1/2 SIL 3 SIL 4	SIL 1/2 SIL 3 SIL 4	Class A Class B Class C
Branch	Level A Level B	Unit Level ASIL B/C Unit Level ASIL D	SIL 3 SIL 4	SIL 3 SIL4	Class B Class C
MC/DC	Level A	Unit Level ASIL D	SIL 4	SIL4	Class C
Statement + Branch	Level A Level B	Unit Level ASIL B/C Unit Level ASIL D	SIL 3 SIL 4	SIL 3 SIL 4	Class B Class C
Statement + MC/DC	Level A	Unit Level ASIL D	SIL 4	SIL 4	Class C
Function		Architectural Level ASIL A/B			
Function + Function Call		Architectural Level ASIL C/D			

The default coverage type is set at the environment level. To set the default coverage type for the environment, select **Coverage => Set Coverage Type => *<coverage_type>*** from the Menu Bar. The current default coverage type is indicated with a checkmark on the Coverage Type menu.

To set the coverage type for an individual source file, right-click a source file in the Source Code Files window and select the coverage type from the context menu.



Note: Changing the Industry Mode may affect the coverage type displayed in the coverage type column of the Environment View. In some instances when Industry Mode is changed, the new mode may not have a coverage type corresponding to the current environment coverage type. In such a case, no coverage type in the list will appear with a checkmark against it.

```
clicast -lc -e <env> [-u <unit>] Cover ENvironment SET_Coverage_type --
clear | None | STATEMENT+MC/DC | STATEMENT+BRANCH | FUNCTION+FUNCTION_CALL
| FUNCTION | MC/DC | BASIS_PATHS | BRANCH | STATEMENT | PROBE_POINT |
COUPLING

Set the coverage type for a cover environment or source file. Use --clear
to clear the coverage type for a source file or for all source files within
the environment.
```

For example, to set the coverage type for the entire environment, enter the following:

```
clicast -e <env> Cover ENvironment SET_Coverage_type STATEMENT
```

This allows the user to change the environment coverage type and consequently the coverage type of all the files within the environment that are set to the default coverage type.

To set the coverage type for a specific file, enter the following:

```
clicast -e <env> -u <filename.c> Cover ENvironment SET_Coverage_type
STATEMENT
```

To clear the coverage type for a specific file, enter the following:

```
clicast -e <env> -u <filename.c> Cover ENvironment SET_Coverage_type --
clear
```

This resets the source coverage type to be the default coverage type.

To clear the coverage type for all files, enter the following:

```
clicast -e <env> Cover ENvironment SET_Coverage_type --clear
```

This resets the source coverage type of ALL files to be the default coverage type.

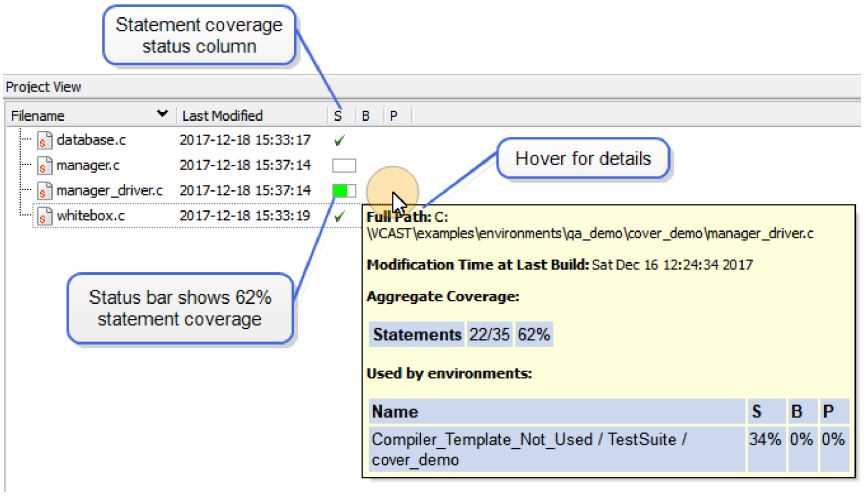
Instrumenting Files



Note: When changing coverage types, instrumentation does not occur.You must select **Instrument** from the right-click menu to instrument an individual source file, or re-instrument all files by selecting **Environment => Re-Instrument All Source** from the Menu Bar.

After instrumenting a file, its coverage status column contains a status bar for the selected coverage. The status bars show the percentage of code coverage achieved for each file. For example, with a file that has 50% coverage, half of the status bar will be green and half will be white. This allows you to quickly see a dashboard of code coverage percentages.

Additionally, you may hover over a file name to see the exact coverage percentages.



If the Source Files pane is displaying the files in a hierarchy, instrumentation applies to the source files in the selected directory.

```
clicast -e <env> [-u <unit>] Cover INstrument <type>

Instrument for code coverage. Specify a coverage type to set the
Environment Coverage Type before instrumentation. Specify a unit and
coverage type to set the Source Coverage Type before instrumentation. Set
coverage type as None to uninstrument. Note that uninstrument will not
change the Environment or Source Coverage Type.

For Default Industry Mode:

None | STATEMENT+MC/DC | STATEMENT+BRANCH | FUNCTION+FUNCTION_CALL |
FUNCTION | MC/DC | BASIS_PATHS | BRANCH | STATEMENT | PROBE_POINT |
COUPLING

For DO-178B/C (Avionics):

None | LEVEL_A | LEVEL_B | LEVEL_C | PROBE_POINT | COUPLING

For ISO-262262 (Automotive):

None | ASIL_D | ASIL_B/C | ASIL_A | ASIL_C/D | ASIL_A/B | PROBE_POINT |
COUPLING

For IEC-61598 (Industrial):

None| SIL_4 | SIL_3 | SIL_1/2 | PROBE_POINT | COUPLING

For EN-50128 (Railway):

None | SIL_4 | SIL_3 | SIL_1/2 | PROBE_POINT | COUPLING

For IEC-62304 (Medical):

None | CLASS_C | CLASS_B | CLASS_A | PROBE_POINT | COUPLING

If another unit with the same base name is already present in the
environment, then a full path or a relative path (to working directory)
```

```
must be specified.
```

Coverage instrumentation creates four files:

- Instrumented file, (same *basename.extension* as the source file)
  If the instrumented file has the same base name as another unit in the Cover environment, then the instrumented file is named *basename.unitnumber.extension*.
- Statement coverage listing file (file extension ".LIS")
- Basis Path file (file extension ".PTH.xml")
- Branch coverage listing file (file extension ".PTL")
- Coverage data file (file extension "*.xml")

View the Instrumented Source File

You can view the instrumented source code for any file in the Source Files pane. Right-click a source file, and choose **View => Instrumented Source Code**. The instrumented version of the source file opens in a Text Editor.

Instrument for Statement Coverage

Statement coverage reports on which executable lines of source code are being executed as the program runs. Each executable statement in the file under test appears with a corresponding subprogram number and executable line number.

```
1 5     *    switch (Order.Entree)
             {
             case NO_ENTREE :
1 6            break;
             case STEAK :
1 7     *     Table_Data.Check_Total = Table_Data.Check_Total + 14.0;
1 8     *     break;
             case CHICKEN :
1 9     *     Table_Data.Check_Total = Table_Data.Check_Total + 10.0;
1 10    *     break;
              ...
```

To instrument a source file for Statement coverage, first check to see that the environment coverage type is set to **Statement**. (If not, set the coverage type at either the environment level or at the individual file level.) Then select the source file, right-click and choose **Instrument**.

Statement coverage maps to the Industry Modes as shown in the following table:

Default	Avionics	Automotive	Industrial	Railway	Medical
Statement	Level A Level B Level C	Unit Level ASIL A Unit Level ASIL B/C Unit Level ASIL D	SIL 1/2 SIL 3 SIL 4	SIL 1/2 SIL 3 SIL4	Class A Class B Class C

To instrument Statement coverage when using Industry Modes, choose **Coverage => Set Coverage Type => Statement** based on the current Industry Mode. Then select **Environment =>**

Reinstrument All Source from the Menu Bar.

As VectorCAST instruments the test harness, a process dialog appears and a message is displayed in the Message window:

```
Instrumenting file <source-file-name>
```

Before you can inspect coverage results, you need to execute a test case.

```
clicast -e <env> [-u <unit>] Cover INstrument STATEMENT

Instrument for Statement coverage. Specify a coverage type to set the
Environment Coverage Type before instrumentation. Specify a unit and
coverage type to set the Source Coverage Type before instrumentation. Set
coverage type as None to uninstrument. Note that uninstrument will not
change the Environment or Source Coverage Type.
```

Instrument for Branch Coverage

Branch coverage displays which outcomes have occurred for each branch point in the program. Each source line that contains a branch point is displayed with a subprogram number, the branch point number, and space for the condition value.



Note: The branch point numbers will not necessarily be consecutive.

Each branch point will have space for either one or two condition values. Boolean decision points (i.e., if statements) are displayed with two place holders, because the condition can be either true or false, as in the following example:

```
4 3 (T) (F)  if A > B
```

Switch statements are handled by treating each “case” as a single decision branch point, as in the following example:

```
1 1      switch (Order.Entree)
             {
1 2    ( )  case NO_ENTREE :
              break;
1 4    (T)  case STEAK :
              Table_Data.Check_Total = Table_Data.Check_Total + 14.0;
              break;
1 6    (T)  case CHICKEN :
              Table_Data.Check_Total = Table_Data.Check_Total + 10.0;
              break;
              ...
```

To instrument a source file for Branch coverage, first check to see that the environment coverage type is set to **Branch**. (If not, set the coverage type at either the environment level or at the individual file level.) Then select the source file, right-click and choose **Instrument**.

Branch coverage maps to the Industry Modes as shown in the following table:

Default	Avionics	Automotive	Industrial	Railway	Medical
Branch	Level A Level B	Unit Level ASIL B/C Unit Level ASIL D	SIL 3 SIL 4	SIL 3 SIL4	Class B Class C

To instrument Branch coverage when using Industry Modes, choose **Coverage => Set Coverage Type => Branch** based on the current Industry Mode. Then select **Environment => Reinstrument All Source** from the Menu Bar.

```
clicast -e <env> [-u <unit> | <path>] Cover INstrument BRANCH

Instrument for Branch coverage. Specify a coverage type to set the
Environment Coverage Type before instrumentation. Specify a unit and
coverage type to set the Source Coverage Type before instrumentation. Set
coverage type as None to uninstrument. Note that uninstrument will not
change the Environment or Source Coverage Type.
```

Instrument for Basis Paths Coverage

Basis paths coverage displays basis paths through the program that have been covered. Each source line that contains a branch point is displayed with a subprogram number, the branch point number, and space for the condition value.



Note: The branch point numbers will not necessarily be consecutive.

Each branch point will have space for either one or two condition values. Boolean decision points (i.e., if statements) are displayed with two place holders, because the condition can be either true or false, as in the following example:

```
1 0       (T)    Add_Included_Dessert
1 1       (T) (F)    if((Order->Entree == STEAK &&
                         Order->Salad == CAESAR &&
                         Order->Beverage == MIXED_DRINK))
                   {
                      Order->Dessert = PIE;
                   }
                   else
1 3       (T) (F)       if((Order->Entree == LOBSTER &&
                            Order->Salad == GREEN &&
                            Order->Beverage == WINE))
                         {
                            Order->Dessert = CAKE;
                         }
                  }
```

Switch statements are handled by treating each “case” as a single decision branch point, as in the following example:

```
1 1      switch (Order.Entree)
             {
1 2    ( )   case NO_ENTREE :
               break;
1 4    (T)   case STEAK :
               Table_Data.Check_Total = Table_Data.Check_Total + 14.0;
               break;
1 6    (T)   case CHICKEN :
               Table_Data.Check_Total = Table_Data.Check_Total + 10.0;
               break;
               ...
```

To instrument a source file for Basis Paths coverage, first check to see that the environment coverage type is set to **Basis Paths**. (If not, set the coverage type at either the environment level or at the individual file level.) Then select the source file, right-click and choose **Instrument**.

```
clicast -e <env> [-u <unit> | <path>] Cover INstrument BASIS_PATHS

Instrument for Basis Paths coverage. Specify a coverage type to set the
Environment Coverage Type before instrumentation. Specify a unit and
coverage type to set the Source Coverage Type before instrumentation. Set
coverage type as None to uninstrument. Note that uninstrument will not
change the Environment or Source Coverage Type.
```

Instrument for MC/DC Coverage

MC/DC coverage is similar to branch coverage, but in addition to reporting on the outcomes of all Boolean expressions, it also reports on the values of all Boolean components of a complex conditional. A complex conditional is one in which there is more than one "term" in the conditional. For example: if (a && b) is a complex conditional with two terms.

To instrument a source file for MC/DC coverage, first check to see that the environment coverage type is set to **MC/DC**. (If not, set the coverage type at either the environment level or at the individual file level.) Then select the source file, right-click and choose **Instrument**.



Note: VectorCAST displays a progress bar when initializing MC/DC or Level A coverage on a source file having an expression with more than 10 sub-expressions.

MC/DC coverage maps to the Industry Modes as shown in the following table:

Default	Avionics	Automotive	Industrial	Railway	Medical
MC/DC	Level A	Unit Level ASIL D	SIL 4	SIL4	Class C

To instrument MC/DC coverage when using Industry Modes, choose **Coverage => Set Coverage Type => MC/DC** based on the current Industry Mode. Then select **Environment => Reinstrument All Source** from the Menu Bar.

Pairs Status

When MC/DC coverage is instrumented, an additional “Pairs” metric is available in the coverage

Metrics Report, **Environment => View => Metrics Report** and the Test Results Management Report, **Environment => View => Management Report**

Pairs Status consists of the total number of MC/DC pairs satisfied compared to the total number of MC/DC pairs in the unit.

As VectorCAST instruments the test harness, a process dialog appears and a message is displayed in the Message window:

```
Instrumenting file <source-file-name>
```

Before you can inspect coverage results, you need to execute a test case.

```
clicast -e <env> [-u <unit> | <path>] Cover INstrument MC/DC

Instrument for MC/DC coverage. Specify a coverage type to set the
Environment Coverage Type before instrumentation. Specify a unit and
coverage type to set the Source Coverage Type before instrumentation. Set
coverage type as None to uninstrument. Note that uninstrument will not
change the Environment or Source Coverage Type.
```

Suppressing MC/DC Analysis

In some rare cases, you may wish to suppress MC/DC coverage for a particular statement or set of statements in your application. It is possible to do this by adding comment tags to the source code.

The Comment Tag Method

To disable MC/DC coverage for a certain line of source code, place the comment VCAST_DONT_DO_MCDC before the line. MC/DC instrumentation is suppressed, but the expression still has branch coverage.

The comments must exist on lines with no other text, immediately preceding the source code line for which you want to modify the coverage. The text can be upper- or lowercase.



Note: For C and C++ source code, the comments should be inside the function's curly brackets; for Ada, the comments should be inside the **`begin`** block. There can be no spaces after the comment characters.

For example:

```
//VCAST_DONT_DO_MCDC (C example)
or
/*VCAST_DONT_DO_MCDC*/
or
--VCAST_DONT_DO_MCDC (Ada example)
Specific statement to which you do not want MC/DC coverage applied

//VCAST_DO_MCDC (C example)
or
```

```
/*VCAST_DO_MCDC*/ (C example)
or
--VCAST_DO_MCDC (Ada example)
Specific statement to which you DO want MC/DC coverage applied
```

MC/DC instrumentation can also be suppressed on multiple lines of code by placing a pair of comments around the code in the source file where suppression is desired.

The comments are :

```
/*VCAST_DONT_DO_MCDC_START*/

...one or more lines for which MC/DC instrumentation should be suppressed

/*VCAST_DONT_DO_MCDC_END*/
```

In the following example, the comments direct VectorCAST to avoid applying MC/DC instrumentation on **`a && b`** when MC/DC or Statement+MC/DC coverage is used. MC/DC tests will also omit test generation for **`a && b`**.

```
/*VCAST_DONT_DO_MCDC_START*/
if (a && b)
{
}
/*VCAST_DONT_DO_MCDC_END*/
```

Instrument Statement+MC/DC

Statement+MC/DC coverage maps to the Industry Modes as is shown in the following table:

Default	Avionics	Automotive	Industrial	Railway	Medical
Statement + MC/DC	Level A	Unit Level ASIL D	SIL 4	SIL 4	Class C

To instrument a source file for Statement+MC/DC coverage, first check to see that the environment coverage type is set to **Statement+MC/DC**. (If not, set the coverage type at either the environment level or at the individual file level.) Then select the source file, right-click and choose **Instrument**.

As VectorCAST instruments the test harness, a process dialog appears and a message is displayed in the Message window:

```
Instrumenting file <source-file-name>
```

Before you can inspect coverage results, you need to execute a test case.

```
clicast -e <env> [-u units] Cover INstrument STATEMENT+MC/DC

Instrument for Statement and MC/DC coverage. Specify a coverage type to set
the Environment Coverage Type before instrumentation. Specify a unit and
coverage type to set the Source Coverage Type before instrumentation. Set
coverage type as None to uninstrument. Note that uninstrument will not
```

```
change the Environment or Source Coverage Type.
```

Instrument Statement+Branch

To instrument a source file for Statement+Branch coverage, first check to see that the environment coverage type is set to **Statement+Branch**. (If not, set the coverage type at either the environment level or at the individual file level.) Then select the source file, right-click and choose **Instrument**.

Statement+Branch coverage maps to the Industry Modes as is shown in the following table:

Default	Avionics	Automotive	Industrial	Railway	Medical
Statement + Branch	Level B	Unit Level ASIL B/C	SIL 3	SIL 3	Class B

To instrument Statement+Branch coverage when using Industry Modes, choose **Coverage => Set Coverage Type => Statement+Branch** based on the current Industry Mode. Then select **Environment => Reinstrument All Source** from the Menu Bar.

As VectorCAST instruments the test harness, a process dialog appears and a message is displayed in the Message window:

```
Instrumenting file <source-file-name>
```

Before you can inspect coverage results, you need to execute a test case.

```
clicast -e <env> [-u units] Cover INstrument STATEMENT+BRANCH

Instrument for Statement and Branch coverage. Specify a coverage type to
set the Environment Coverage Type before instrumentation. Specify a unit
and coverage type to set the Source Coverage Type before instrumentation.
Set coverage type as None to uninstrument. Note that uninstrument will not
change the Environment or Source Coverage Type.
```

Instrument for Function Coverage

To instrument a source file for Function coverage, first check to see that the environment coverage type is set to **Function**. (If not, set the coverage type at either the environment level or at the individual file level.) Then select the source file, right-click and choose **Instrument**.

Function coverage maps to the Industry Modes as is shown in the following table:

Default	Avionics	Automotive	Industrial	Railway	Medical
Function		Architectural Level ASIL A/B			

To instrument Function coverage when using Industry Modes, choose **Coverage => Set Coverage Type => Function** based on the current Industry Mode. Then select **Environment => Reinstrument All Source** from the Menu Bar.

As VectorCAST instruments the test harness, a process dialog appears and a message is displayed in the Message window:

```
Instrumenting file <source-file-name>
```

Before you can inspect coverage results, you need to execute a test case.

```
clicast -e <env> [-u units] Cover INstrument FUNCTION

Instrument for Function coverage. Specify a coverage type to set the
Environment Coverage Type before instrumentation. Specify a unit and
coverage type to set the Source Coverage Type before instrumentation. Set
coverage type as None to uninstrument. Note that uninstrument will not
change the Environment or Source Coverage Type.
```

Instrument for Function + Function Call Coverage

To instrument a source file for Function +Function Call coverage, first check to see that the environment coverage type is set to **Function+Function Call**. (If not, set the coverage type at either the environment level or at the individual file level.) Then select the source file, right-click and choose **Instrument**.

Function + Function Call coverage maps to the Industry Modes as is shown in the following table:

Default	Avionics	Automotive	Industrial	Railway	Medical
Function + Function Call		Architectural Level ASIL C/D			

To instrument Function + Function Call coverage when using Industry Modes, choose **Coverage => Set Coverage Type => Function+Function Call** based on the current Industry Mode. Then select **Environment => Reinstrument All Source** from the Menu Bar.

As VectorCAST instruments the test harness, a process dialog appears and a message is displayed in the Message window:

```
Instrumenting file <source-file-name>
```

Before you can inspect coverage results, you need to execute a test case.

```
clicast -e <env> [-u units] Cover INstrument FUNCTION+FUNCTION_CALL

Instrument for Function and Function Call coverage. Specify a coverage type
to set the Environment Coverage Type before instrumentation. Specify a unit
and coverage type to set the Source Coverage Type before instrumentation.
Set coverage type as None to uninstrument. Note that uninstrument will not
change the Environment or Source Coverage Type.
```

Instrument for Probe Point

In Cover environments, VectorCAST Probe allows users to apply Probe Point-only instrumentation. This feature allows users to insert probe points without first instrumenting the environment for coverage.

To instrument a source file for Probe Point coverage, first check to see that the environment coverage

type is set to **Probe Point**. (If not, set the coverage type at either the environment level or at the individual file level.) Then select the source file, right-click and choose **Instrument**.

```
clicast -e <env> [-u units] Cover INstrument PROBE_POINT

Instrument for Probe Point coverage. Specify a coverage type to set the
Environment Coverage Type before instrumentation. Specify a unit and
coverage type to set the Source Coverage Type before instrumentation. Set
coverage type as None to uninstrument. Note that uninstrument will not
change the Environment or Source Coverage Type.
```

List the Number of Bytes Needed to Instrument Each Unit in an Environment

To report the number of bytes needed to instrument each unit in a given VectorCAST Cover environment use the following clicast command:

```
clicast -e <env> Cover REPort Ram_usage <outputfile>

Create a text report in csv format detailing the number of bytes used to
instrument all units for 16-bit and 32-bit architectures. The report is
saved to <ENV_NAME>_ram_usage.csv unless <filename> is provided.
```

An example of the report obtained on a Cover environment using the C tutorial files instrumented for MCDC coverage is:

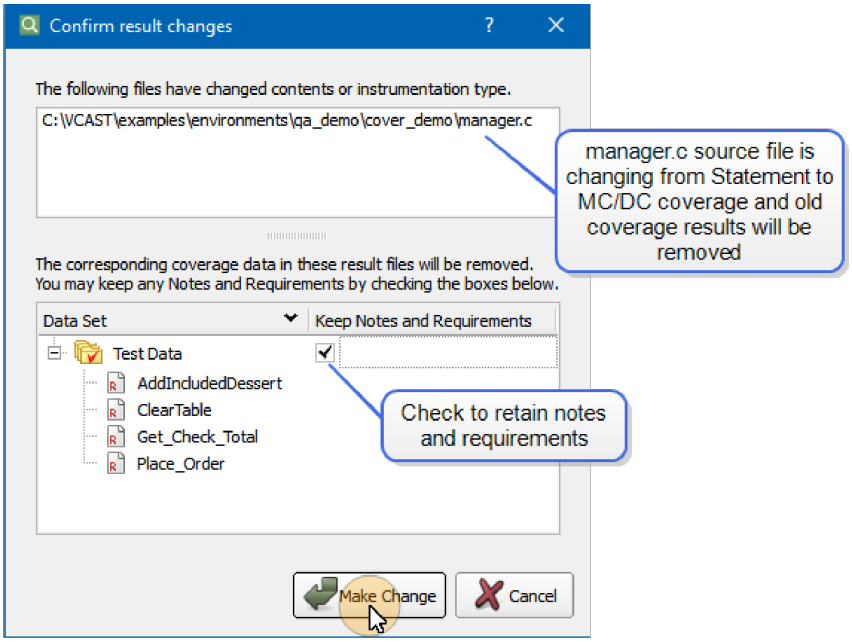
```
Unit Name, 16-bit Total, 32-bit Total
manager.c,154 ,308
database.c,34 ,68
manager_driver.c,46 ,92
whitebox.c,48 ,96
Mcdc Overhead,12000 ,24000
Totals,12282 ,24564
```

The RAM usage for "Mcdc Overhead" is affected by the value of the option "Maximum MC/DC expressions" (**VCAST_MAX_MCDC_STATEMENTS**).

Change the Type of Coverage

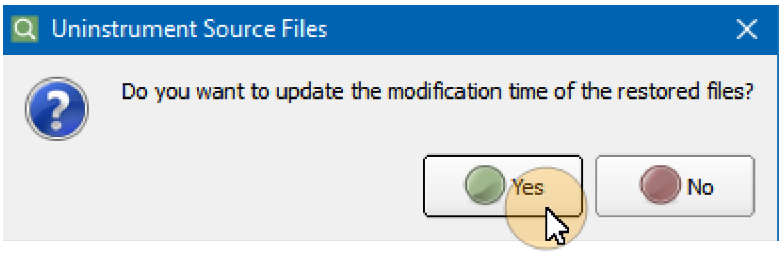
If you instrument a source file with one type of coverage, add results, and then change the type of coverage for that file, the results are now invalid.

A Confirm Result Changes dialog may open, alerting the user to a change in instrumentation type for the source file. Note that the coverage data in the result files will be removed. To retain any Notes and Requirements, select the checkbox in the bottom pane. Select the **Make Change** button to confirm. A process dialog will appear as the file is re-instrumented.



Un-Instrument Source Files

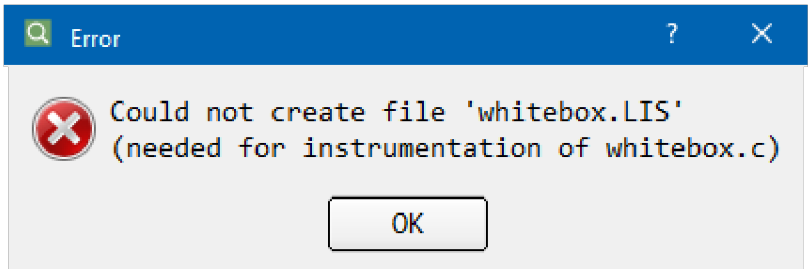
To remove the coverage initialization from a source file, right-click a source file, and choose **Uninstrument**. The coverage format reverts to "(none)". A dialog appears in which you can indicate whether or not you want to update the time stamp on the file. If you do, click **Yes**, otherwise, click **No**.




```
clicast -e <env> [-u <unit>] Cover INstrument None

Un-Instrument all source files or the specified source file.
```

Troubleshooting: Error during uninstrument

While trying to Uninstrument a unit in a Cover environment you see the following message, it is because the original source file is no longer present.



To solve this problem, either delete the unit from the Cover environment or replace the missing unit in the original location.

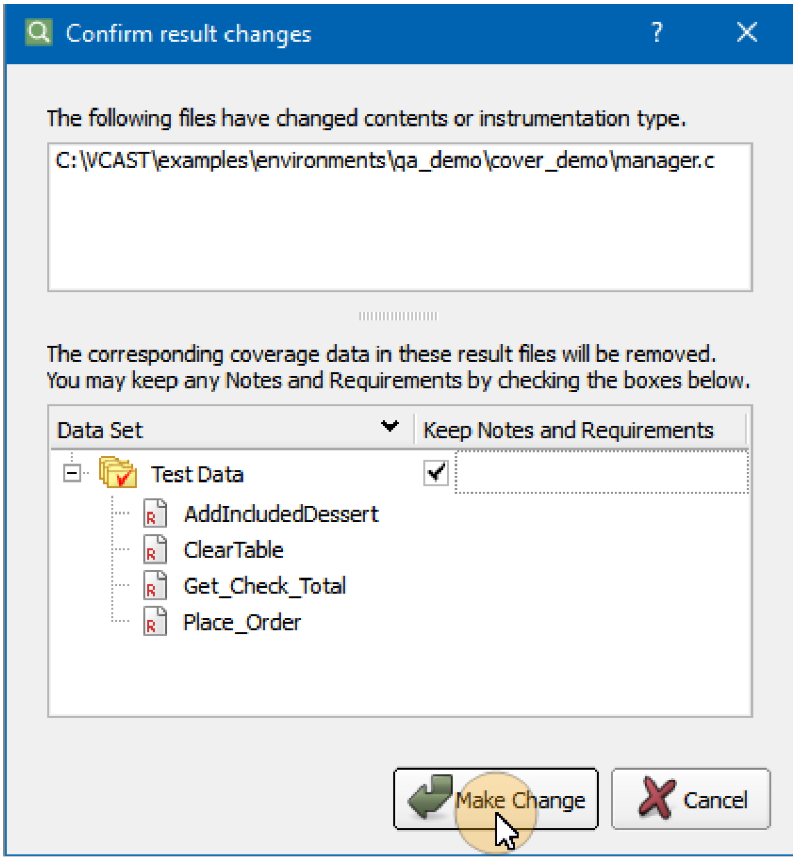
Re-Instrument All Source Files

The **Environment => Reinstrument All Source** command enables you to reinstrument all previously instrumented source files. This feature is useful when you have different coverage types for different source files, and you change a coverage option, such as the Coverage I/O type.

Reinstrument All Source files if:

- you have modified your source code files
- you have changed the coverage I/O type
- you have changed a coverage option that is saved to the env.vcp file or the CCAST_.CFG file
- you have changed a compiler option that is saved to the CCAST_.CFG file

After reinstrumenting a source file, the test results may be invalid. A Confirm Result Changes dialog opens, identifying the files that have changed. The coverage data in the result files will be removed. To retain any Notes and Requirements, select the checkbox in the bottom pane. Select the **Make Change** button to confirm. A process dialog appears as the file is re-instrumented.




```
clicast -e <env> Cover Source REInstrument

Re-Instrument all source files, retaining the same coverage type but using
the current Coverage I/O type and options.
```

Avoid Instrumenting Sections of Source Code

There may be sections of your source code that you do not want to instrument for code coverage. VectorCAST supports two methods of controlling instrumentation: using comments or using pragmas.

Controlling Instrumentation Using Comments

You can instruct VectorCAST to ignore sections of your source code by delimiting the section with the following source code comment markers:

```
/*VCAST_DONT_INSTRUMENT_START*/
C or C++ code that you do not want to be instrumented
/*VCAST_DONT_INSTRUMENT_END*/

//VCAST_DONT_INSTRUMENT_START
```

```
C++ code that you do not want to be instrumented
//VCAST_DONT_INSTRUMENT_END
```

The text can be upper- or lowercase, but there can be no spaces between the comment characters and the text of the delimiter. Care must be used to ensure that the start and end delimiters are within the same scope. That is, if you have a **case** statement, you do not insert the delimiters such that half of the case statement is not instrumented and half is instrumented for code coverage.

For C and C++, you must ensure that your compiler's pre-processor leaves comments in the preprocessor output. For most compilers, this option is 'dash capital C' (-C). To turn on this option, go to the **Tools => Options** dialog, **C/C++** tab, and add **-C** to the Preprocessor command.

Controlling Instrumentation Using Pragmas

The method of controlling instrumentation using pragmas is useful in cases where you want to remove the argument that retains comments (for example, -C for GNU) from the compiler's preprocessor command. VectorCAST treats the following pragmas the same as the equivalent instrumentation comments:

vcast_dont_instrument_start

vcast_dont_instrument_end

vcast_dont_do_mcdc_start

vcast_dont_do_mcdc_end

vcast_dont_do_mcdc

vcast_do_mcdc

For example:

```
#pragma vcast_dont_do_mcdc_start
int a = b && c;
#pragma vcast_dont_do_mcdc_en
```

tells VectorCAST not to do MC/DC instrumentation on the statement.

Compiling the Instrumented Source Files

After instrumenting, compile and link the instrumented source code files to create an executable.

Simple Compilation Examples

The instrumented source files are in the original source file location, making it easy to use your existing build process (Makefile, etc.). You will need to copy or include the cover I/O file (see below) and its header file (vcast_c_options.h/vcast_ada_options.ad*) from the environment directory.

Compiling Tutorial files into an Application

Example in C
Cover env with source files instrumented, in the original source file location. The cover I/O files are

located in the environment directory. We include (-I) that directory to get them.

```
gcc -o driver manager_driver.c manager.c database.c \
COVER_ENV/c_cover_io.c -ICOVER_ENV
```

Example in Ada

Cover env with source files instrumented, in the original source file location. The cover I/O files are located in the environment directory. We include (-I) that directory to get them.

```
gnatchop -w manager_driver.adb manager.adb database.adb
gnatmake manager_driver -ICOVER_ENV
```

The Cover I/O File

When compiling the instrumented source files, you must include one I/O file that VectorCAST Cover creates. The I/O files are in the environment directory and are called:

c_cover_io.c	for C source files
c_cover_io.cpp	for C++ source files
vcast_cover_io.adb	
vcast_cover_io.ads	for Ada source files
vcast_cover_io_ada83.ads	for Ada83
vcast_cover_io_gmart.adb	for GMART
vcast_cover_io_mixed_lang.adb	for mixed language environments
vcast_cover_io-restricted.adb	for Ada source with pragma restriction No Recursion (located in the VectorCAST installation directory, IO sub-directory)

Depending on the language of your source files, you would compile and link one of them with your instrumented source files.

IMPORTANT: Note that you may need to customize this I/O file to support the particular I/O mode and I/O API that may exist on your target. See the section below "To Customize the Cover I/O File" on page 115 that describes the changes you may need to make to c_cover_io.c before you include this in your project and build with the instrumented source files.



Note: If you are using GNAT, you must run **`gnatchop -w`** on the instrumented source files before compiling them into the executable program.

Note that the file **`<cover env>/coupling/couplingData.c`** (which implements the coupling functions) must be compiled and linked when compiling C or C++ source files. This is done in the same manner as compiling and linking the **`c_cover_io.c`** file. See "Build the Instrumented Application" on page 292 for more information on coupling instrumentation.

Compiling with Visual Studio in Secure Mode

In cases where instrumented source files are compiled in a Cover environment with Visual Studio in secure mode (using /W3), a compilation warning may occur due to the use of **`fopen()`** in

VectorCAST's **c_cover_io.c/.cpp** file.

In this situation, users can direct VectorCAST to generate the **c_cover_io.c/.cpp** file with **fopen_s()** instead. This is accomplished by adding the defined variable VCAST_USE_FOPEN_S to the C_DEFINE_LIST in the CCAST.CFG file or by adding it as a defined variable to the compile command for the instrumented source files.

Mixed Language Environments

In cases where you want to instrument Ada code and C or C++ code that is part of the same application, you will need to make changes to the Ada version of the VectorCAST Cover I/O library file. The delivered file “vcast_cover_io.adb” is set up for Ada-only testing.

The file vcast_cover_io_mixed_lang.adb supports mixed language VectorCAST Cover environments. To use it, rename the vcast_cover_io_mixed_lang.adb file in the Cover environment directory to vcast_cover_io.adb and use it in your build process.

Pragma Restrictions: No Recursive Calls

In cases where your source files use the "pragma restrictions" clause to enforce No Recursive Calls, the default I/O package supplied with the Cover tool does not compile because it uses recursive calls to implement MC/DC coverage. In the VectorCAST installation directory, IO sub-directory, there is a new I/O package body found in the "vcast_cover_io-restricted.adb" file that removes the MC/DC coverage capability, and therefore removes the recursive calls.

To use this new file, first copy it from the VectorCAST installation directory, IO subdirectory. (If using GNAT, you will need to gnatchop it.) Then compile your instrumented source files into an executable.



Note: Coverage types MC/DC and DO-178B Level A are not supported when using this coverage I/O file. If you instrument one of these types, you will get coverage data that is missing the MC/DC branch data. That is, MC/DC data will be completely empty, and Level A data will only include Statement data.

To Customize the Cover I/O File

...for Embedded Targets that don’t support a file system

If your embedded target does not support a file system then you will want to configure your system to capture coverage data from a serial port. Serial port configuration is target dependent, but as an example, if the serial port communications registers on your target are configured for 9600 baud 8-N-1 then you can configure the HyperTerminal application on your test PC with the same settings to capture the coverage data that will get sent out the serial port. HyperTerminal also allows you to stream the serial port data it receives to a user specified file name, e.g. MY_COVERAGE_DATA.DAT. Then you can import this file with your coverage data into your coverage environment. To configure the c_cover_io.c for this mode, you should #define the **VCAST_CUSTOM_STD_OUTPUT** variable, and then search through the c_cover_io.c file for that variable name and insert your code there as per the commented instructions in the source code. Next, modify your command line as follows when you compile the c_cover_io.c file:

```
cl -DVCAST_CUSTOM_STD_OUTPUT -c c_cover_io.c
```

After compiling your instrumented source code with your custom c_cover_io.c file you can run the resultant executable to produce the coverage data out the serial port. By sure that your serial port cable is properly hooked up to your test PC and that you have connected via the HyperTerminal application before you run the executable.

...for Embedded Targets that do support a file system

When you compile your instrumented source files and then run your executable to create test results, the results are stored in the TESTINSS.DAT file by default. When you are using VectorCAST Cover with an embedded target that has a proprietary file IO API, then the user must provide code to invoke the necessary custom file IO functions to write coverage data to the TESTINSS.DAT file. First, the user should search for the **VCAST_CUSTOM_FILE_OUTPUT** flag inside the c_cover_io.c file, and follow the commented guidelines for inserting your code in place. There will be 3 functions that you may typically need to modify. These include the functions that open and close the TESTINSS.DAT file as well as the function that writes the coverage data to the TESTINSS.DAT file. You should NOT change the name of these functions, but modify the content accordingly to conform to your particular API. Next, modify your command line as follows when you compile the c_cover_io.c file:

```
cl -DVCAST_CUSTOM_FILE_OUTPUT -c c_cover_io.c
```

This compile command defines **VCAST_CUSTOM_FILE_OUTPUT**, which causes the user supplied code to be included instead of the standard default VectorCAST file IO code. If you forget to insert some user code in these functions, you may get any of the following error messages when you compile your project. (where Ennnn is a compiler specific error code).

```
Filename(line_number): Error!
Ennnn: You need to insert code in vCAST_CREATE_INST_FILE()

Filename(line_number): Error!
Ennnn: You need to insert code in VCAST_CLOSE_FILE()

Filename(line_number): Error!
Ennnn: You need to insert code in VCAST_WRITE_TO_INST_FILE_COVERAGE()
```

To Change the Output Options for Test Results

When you compile your source files after instrumenting and then run your executable to create test results, by default the results are stored in the file TESTINSS.DAT. When you are using VectorCAST Cover with an embedded target, and you want to use standard file IO for the data, modify your command line as follows when you compile the c_cover_io.c file:

```
cl -DVCAST_STD_IO -c c_cover_io.c
```

This compile command defines VCAST_STD_IO, which causes the data to be streamed to the standard out.

Use the following defined variables when you compile the c_cover_io.c or c_cover_io.cpp file to set options for the test output:

Defined Variable	Effect on Test Results	Notes
(none)	Output is written to TESTINSS.DAT.	Default behavior.
VCAST_APPEND_TO_TESTINSS	Open output file TESTINSS.DAT for append, rather than for writing each invocation of application.	Also available as an option in GUI and in CLICAST.
VCAST_USE_STDOUT	Output is written to stdout using 'puts'.	
VCAST_CUSTOM_STD_OUTPUT	User-provided code is inserted to support a custom I/O API that writes output to TESTINSS.DAT on targets that don't support a file system.	Must have VCAST_USE_ STDOUT defined.
VCAST_CUSTOM_FILE_OUTPUT	User-provided code is inserted to support a custom I/O API that writes output to TESTINSS.DAT on targets that do support a file system.	
VCAST_USE_PUTCHAR	Output is written to stdout using 'putchar'.	Must have VCAST_USE_ STDOUT defined.
VCAST_USE_GH_SYSCALL	Output is written to stdout using the GH syscall (for Green Hills INTEGRITY 178B).	Must have VCAST_USE_ STDOUT defined.
VCAST_USE_STD_STRING	Turns on the use of 'memset' from the system header 'string.h'.	
VCAST_USE_STATIC_MEMORY	No 'malloc' is available, use alternate data.	
VCAST_MAX_MCDC_STATEMENTS	The number of MC/DC statement conditions that can be reached when 'malloc' is not available.	Must have VCAST_USE_ STATIC_MEMORY defined.
VCAST_MAX_COVERED_ SUBPROGRAMS	The number of subprograms that may be covered.	Must have VCAST_USE_ STATIC_MEMORY defined.
VCAST_ATTRIBUTE_CODE	Specify an attribute that will be placed before the ASCII, binary, and subprogram coverage pool of global variables. This is useful for putting this data in specific places in memory.	
VCAST_FLUSH_DATA	Use the flush system call after each string is written with fprintf in VCAST_ WRITE_TO_INST_FILE_ COVERAGE. The default is disabled. Define to any value to enable.	
VCAST_APPEND_WIN32_PID	Append the process ID to the TESTINSS-<PID>.DAT file name.	Uses Windows-specific system calls to get the pid.
VCAST_APPEND_POSIX_PID	Append the process ID to the TESTINSS-<PID>.DAT file name.	Uses Posix-specific system calls to get the pid.

Defined Variable	Effect on Test Results	Notes
VCAST_APPEND_SECONDS_SINCE_EPOCH	Append the process ID to the TESTINSS-<PID>.DAT file name.	Use this define if your compiler does not support the either of the two API calls above.
VCAST_ENABLE_DATA_CLEAR_API	Enable this macro to add the API VCAST_CLEAR_COVERAGE_DATA, which allows you to clear the currently collected coverage data during the execution of the instrumented executable.	Used with Buffered Coverage I/O only.
VCAST_DUMP_CALLBACK	If this is defined to a function name, then when the user's source code called VCAST_DUMP_COVERAGE_DATA(), the function is defined to will be called. The purpose is to allow the user's main loop to be given a chance to run within a certain time frame.	Used with Buffered Coverage I/O only.

To Create a Test Result File

Run the executable that you compiled and linked using the instrumented source code and the IO file. Use your program as usual, which will create a results file TESTINSS.DAT with coverage results.

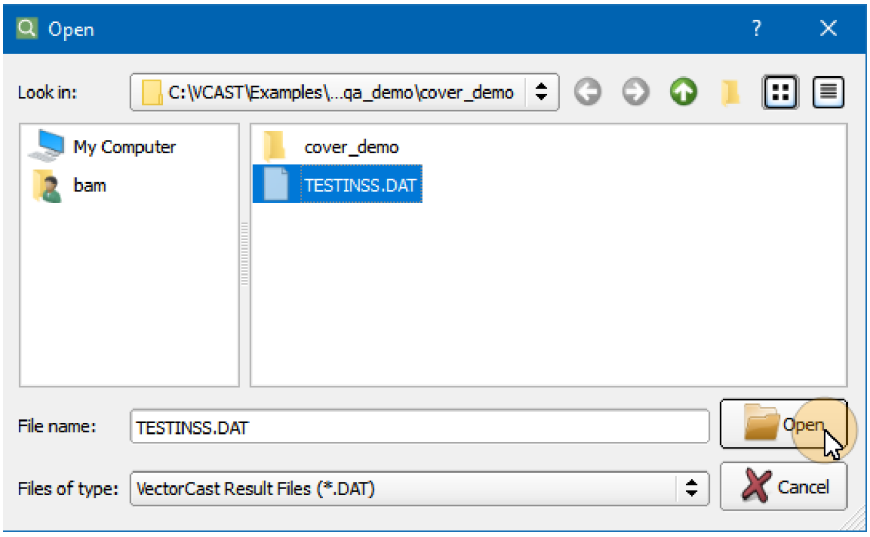
Add in the results file TESTINSS.DAT to the VectorCAST Cover environment whenever you need to see the coverage results of your actions.

See also “"Compiling the Instrumented Source Files" on page 113”.

Viewing Coverage Results

To Add a Test Result

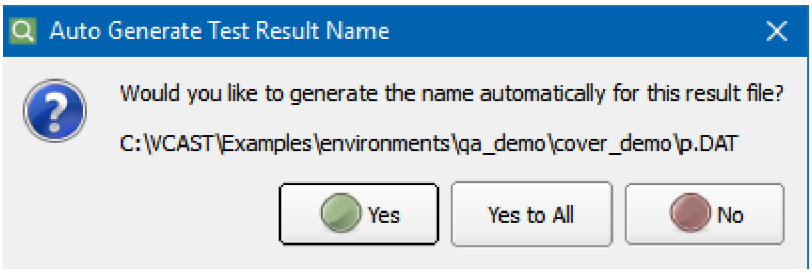
After you have executed your program that has been compiled using the instrumented source files, a TESTINSS.DAT file is created. This file contains information about the unit number and the lines or branches covered during the test execution. The **Environment => Add Test Results...** command brings up a dialog box enabling you to navigate to the directory where the TESTINSS.DAT file has been stored.



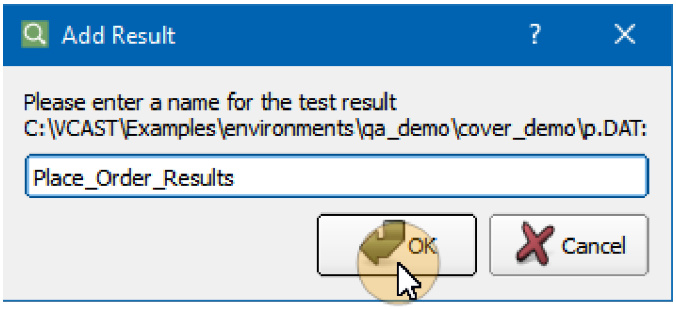
You can also access this command by clicking on the icon  in the toolbar.

```
clicast -e <env> Cover RESult Add <result-file> [<optional-result-name>] [-
n<notes text> | <notes file>] [-r<requirements text> | <requirements file>]

Add the result file <result-file> to the Cover environment and name it
<results-name>If the optional <results-name> is not specified, VectorCAST
uses the file name as the results name.

The optional <notes text> or <notes file> provides the contents of the
result's Notes tab, and the optional <requirements text> or <requirements
file> provides the contents of the result's Requirements tab.
```

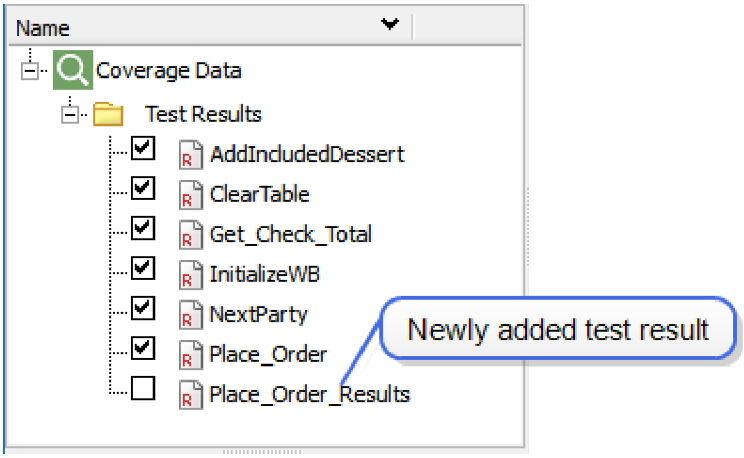
If you choose more than one .DAT file, most likely you have renamed them something other than TESTINSS.DAT. VectorCAST Cover provides the opportunity to retain those names. When more than one result file is selected, a dialog appears, asking if you want to name the test result in VectorCAST Cover the same name as the result file itself. In the example below, the result file is named p.DAT.



Clicking **Yes** names the result the same name as the file, and repeats for the other files you selected. Clicking **Yes to All** adds all of the result files and auto-names them. If you choose **No**, another dialog box will prompt you to enter a name for the test result.



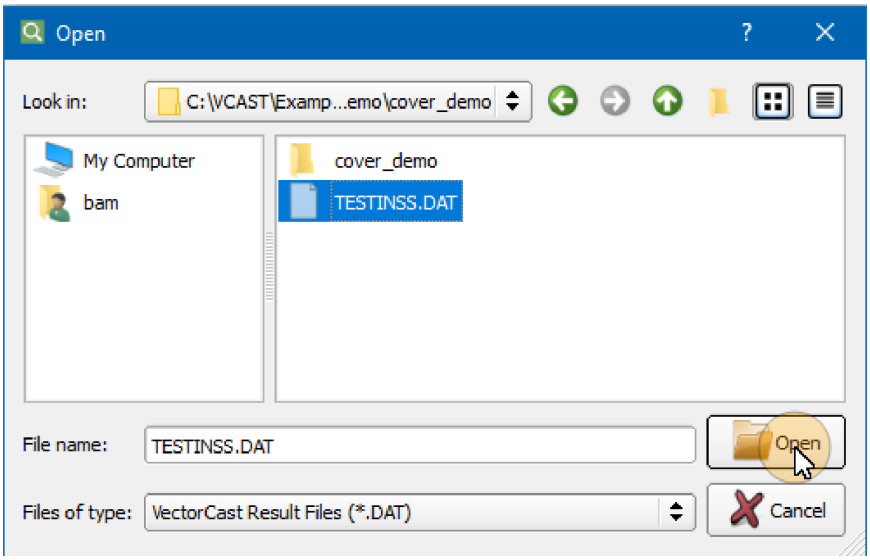
The test result is then displayed in the Test Result window.



To Update a Test Result

If you re-run your instrumented program again and want to make a correction to a test result that has already been added to a Cover environment, you can update it with new test data. Additionally, you can use the command line to update a test result or add Notes or Requirements.

To update a test result, right-click it in the Test Results pane and choose **Update**. Doing so brings up a dialog box enabling you to navigate to the directory where the TESTINSS.DAT file has been stored.



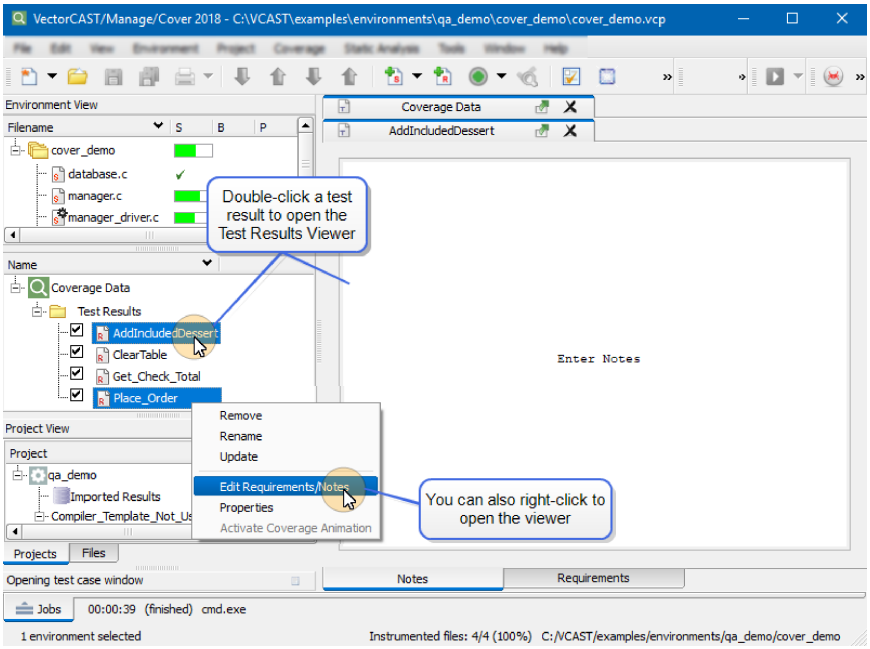
In this Open File dialog, you are restricted to choosing one .DAT file. Once you click **Open**, the data in that file replaces the data in the test result. Any Notes or Requirements already present in the Test Result are not removed.

```
clicast -e <env> Cover RESult Update <result-name> [<new-result-file>] [-
n<notes text> | <notes file>] [-r<requirements text> | <requirements file>]

Update the test result named <result-name> with <new-results file> and/or
Notes and/or Requirements, if provided.
```

To Open the Test Results Viewer

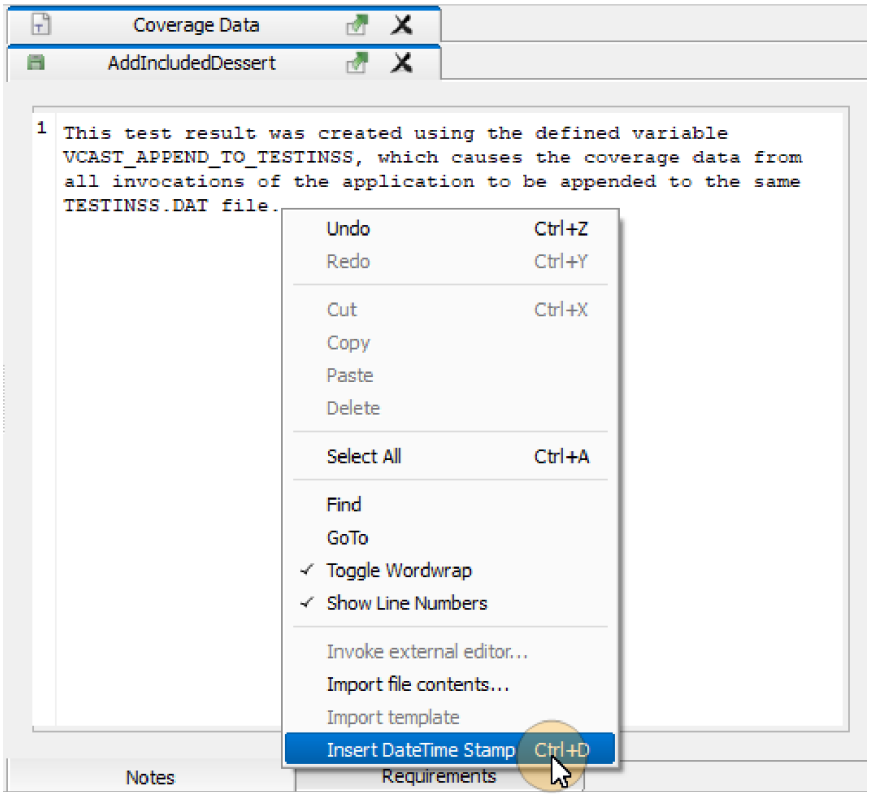
The Test Result Viewer displays the test result's Notes and Requirements, if any. To open the Test Result Viewer:

- double-click a test result in the Test Results pane
- right-click a test result and choose **Edit Notes/Requirements**



The Notes Tab of the Test Results Viewer

Every test result has a Notes tab. Enter text here which you want to associate with the test result. Use the familiar right-click menu to Import file contents, insert a timestamp, toggle wordwrap, etc.



The notes follow the test result when exported to a coverage script and when imported to another environment. When exported to a coverage script, they are notated as:

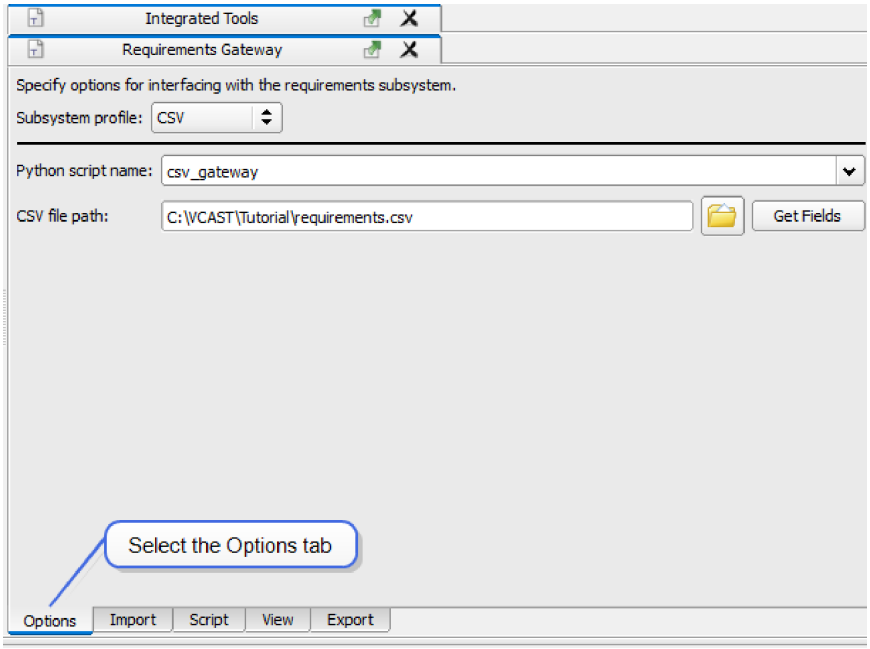
```
RESULT.NOTES
This test result was created using the defined variable VCAST_APPEND_TO_TESTINSS,
which causes the coverage data from all invocations of the application to be
appended to the same TESTINSS.DAT file.
12/20/2017 4:04:50 PM
RESULT.END_NOTES
```

The Requirements Tab of the Test Results Viewer

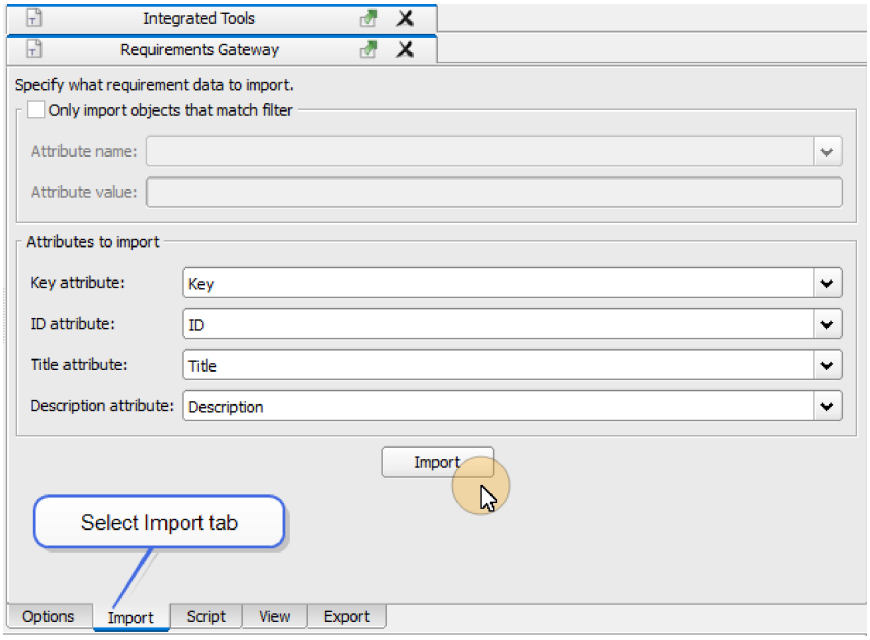
When a repository is specified (**Tools => Options** dialog, **C/C++** tab, **Settings repository**) that contains a requirements.xml file, an additional tab appears when you open a test result.

To illustrate, the Requirements Gateway will be used to import the requirements.csv file, located in the **`%VECTORCAST_DIR%\Tutorial`** directory.

Select **Tools=> Requirements Gateway => Requirements Gateway**. When the Requirements Gateway opens, click the Options tab, and then select CSV as the Subsystem profile and set the CSV file path to c:\VCAST\Tutorial\requirements.csv.

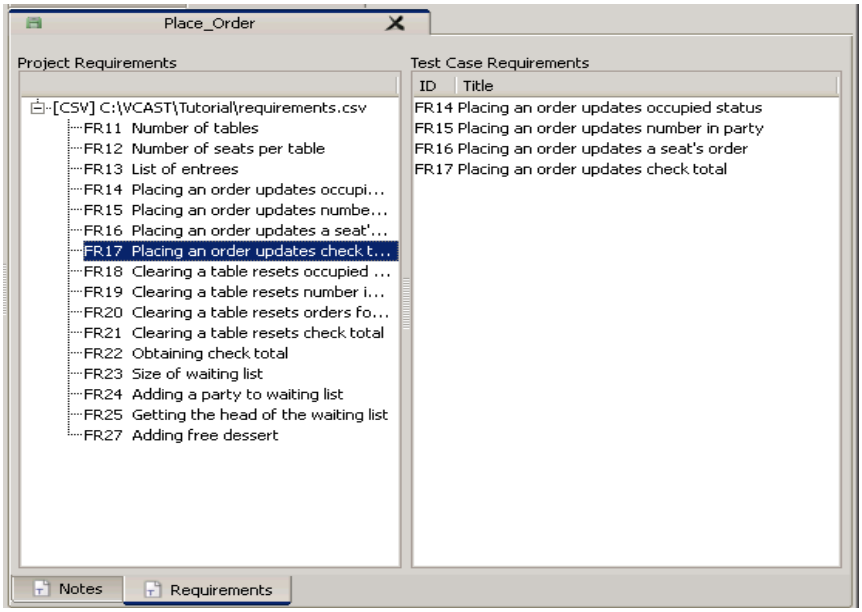


Click **Get Fields** and then select the Import tab. In the Attributes to import panel set Key attribute to Key, ID attribute to ID, Title attribute to Title and the Description attribute to Description. Finally, click **Import**.



After the import completes, recompile the source files, execute the tests and import the test results.

When the test result is opened, the Requirements tab contains all requirements on the left. You can double-click particular requirements to move them to the right, which indicates that they are associated with this test result.



The requirements follow the test result when exported to a coverage script and when imported to another environment (which has the same repository setting). When exported to a coverage script, they are notated as:

```
RESULT.REQUIREMENTS
FR14
FR15
FR16
FR17
RESULT.END_REQUIREMENTS
```

To Open the Coverage Viewer

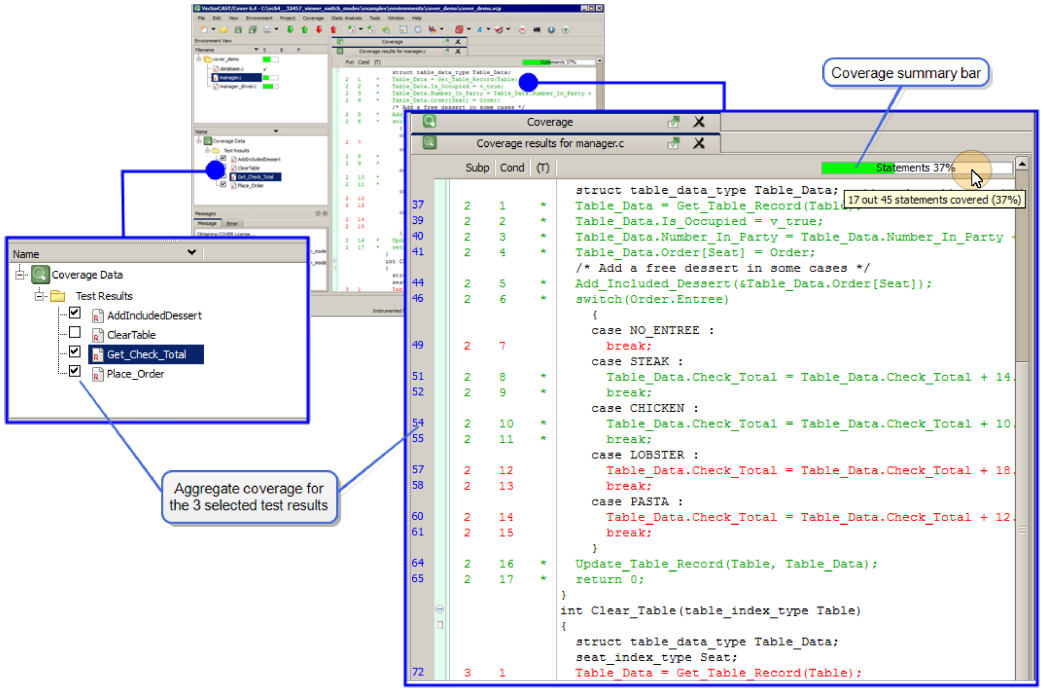
The Coverage Viewer displays the code coverage achieved for a source code unit, colorized to indicate the coverage level achieved. To open a Coverage Viewer, do one of the following:

- Double-click a source file in the Source Code Files pane
- Place a check in the checkbox next to a Test Result (opens the Coverage Viewer for the first unit referenced in the Test Result's data)
- Right-click a source file and choose **View => Coverage Viewer**
- Right-click a source file and choose **Open Coverage Viewer**

To Turn on Coverage Results

Click a checkbox next to the name of a test result to see the coverage achieved in the Coverage Viewer. Alternatively, right-click on the Test Results node to see a pop-up menu. The menu item **Select All Children** causes all results to be selected. Uncheck them by choosing **Deselect All Children**.

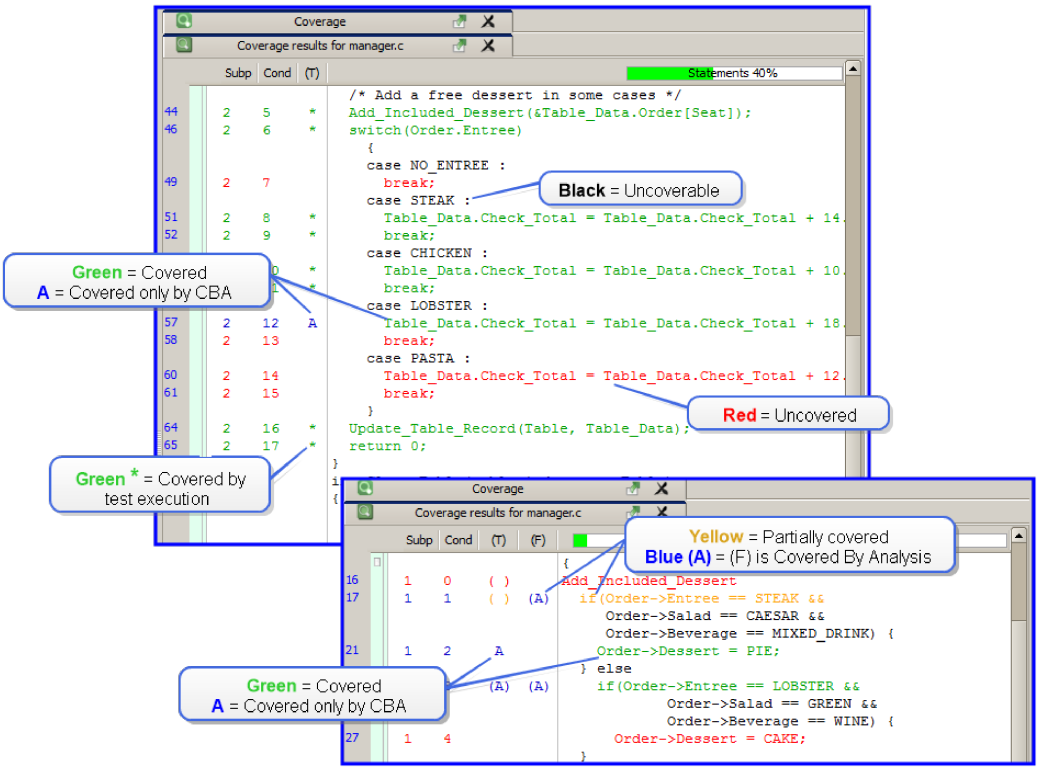
By checking more than one results checkbox, you can see the aggregate coverage achieved by those test results. In the upper right corner of the Coverage Viewer, a summary bar of the coverage is displayed. For Statement coverage, the display reads "Statements." Hover over the display to see the tool tip showing how many statements are covered out of the total, and a percentage. For branch and MC/DC coverage, the display reads "Branches," and the tool tip shows "n of m branches covered (%)." For MC/DC coverage, an additional display reads "Pairs" and the tool tip shows "n of m pairs covered (%)".



The Coverage Viewer

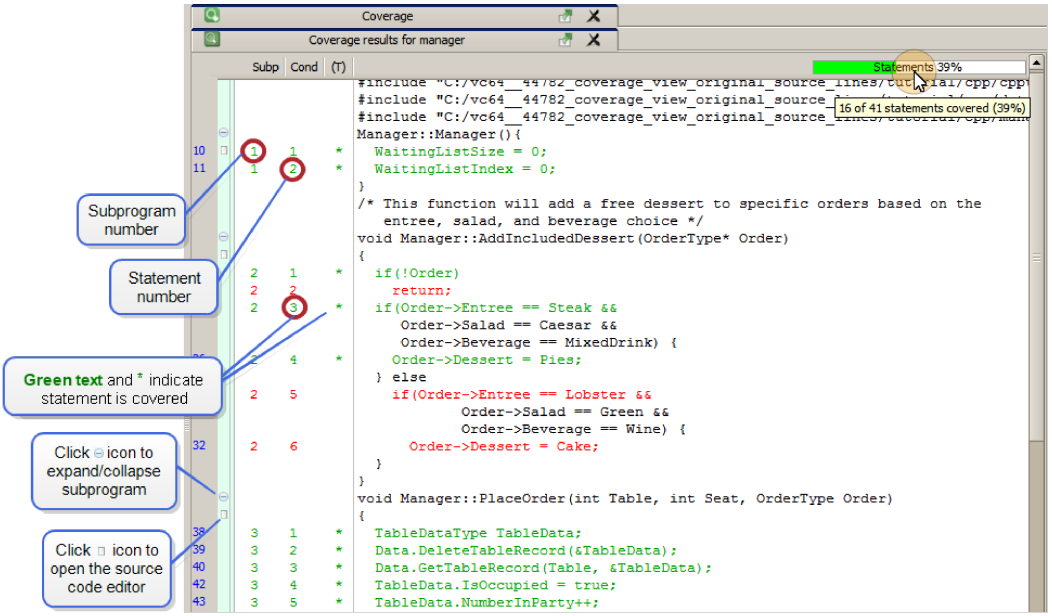
Once some test results are enabled, the Coverage Viewer uses the following colorization to indicate coverage level:

- Covered lines are displayed in green
- Lines covered only by test execution results are displayed in green (indicating the line is covered) with a green asterisk "*" (indicating the line is covered *only* by test execution results)
- Lines covered only by CBA results are displayed in green (indicating the line is covered) with a blue "A" (indicating the line is covered *only* by CBA)
- Lines covered by both regular execution results *and* CBA results are displayed in green (indicating the line is covered) and with a blue asterisk "*" (for statement) or a blue "T" or "F" (for branch), which indicates that it is *also* covered by CBA
- Uncovered lines are displayed in red
- Partially covered lines are displayed in yellow
- Uncoverable lines are displayed in black



Statement Coverage

When Statement coverage is active, the Coverage Viewer looks like the following example:

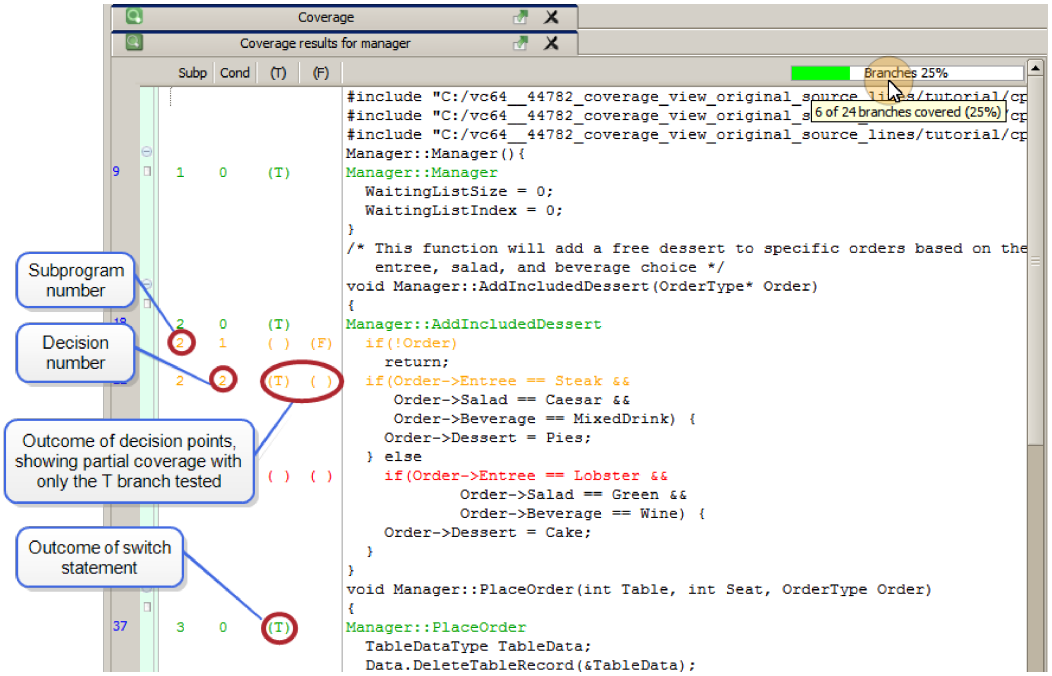


The numbers along the left-hand margin indicate the subprogram number and the statement number.

An asterisk (*) indicates that the statement was covered (along with the color-coding).

Branch Coverage

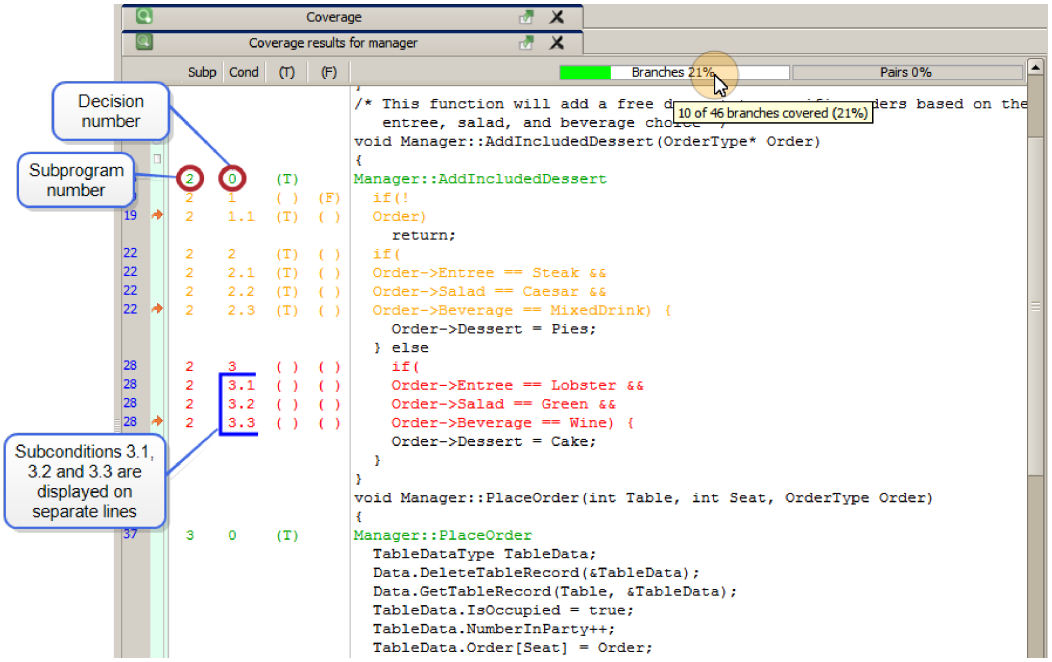
When Branch coverage is active, the Coverage Viewer looks like the following example:



The left-most number indicates the subprogram number, and the second number indicates the decision number. For Boolean decision points, two sets of parentheses are visible and are filled in with a T or F when the True or False outcomes have been tested. For switch statements, one set of parentheses is visible and a **(T)** is filled in when that branch outcome has been tested.

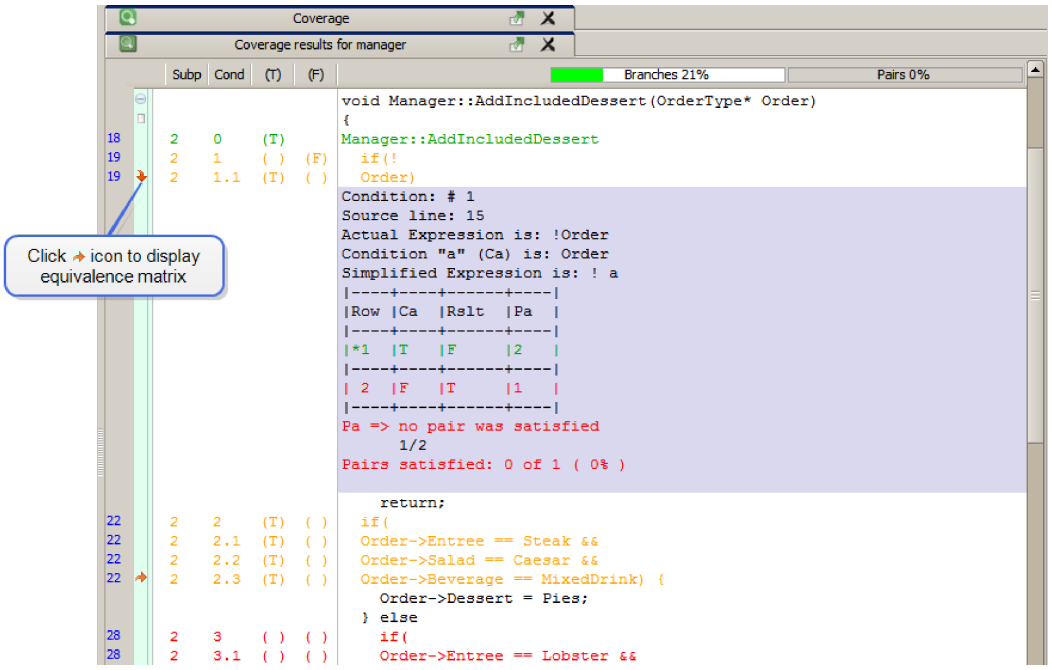
MC/DC Coverage

When MC/DC coverage is active, the Coverage Viewer looks like the following example:



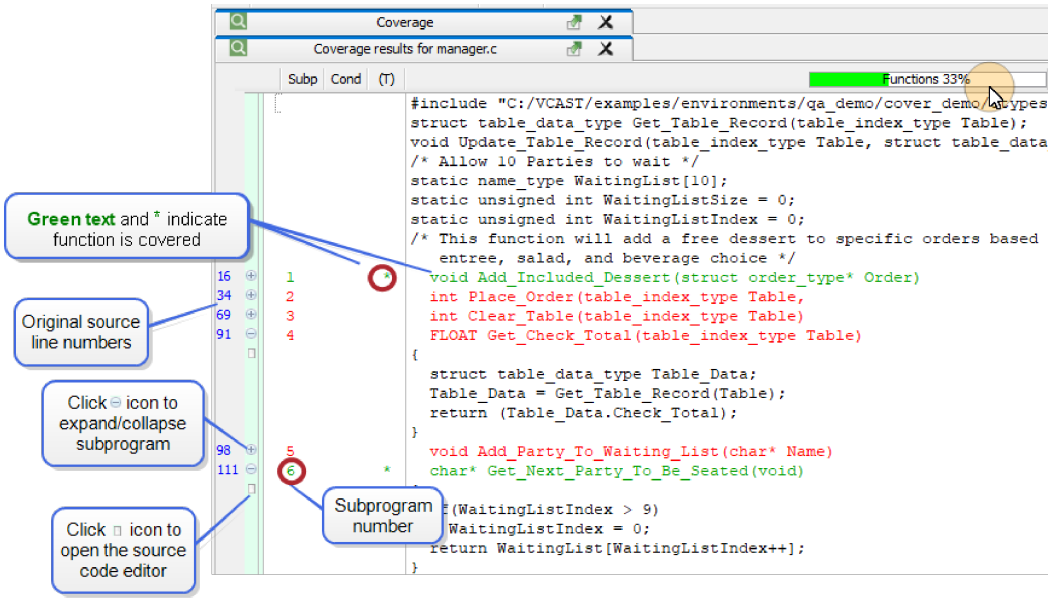
The left-most number indicates the subprogram number, and the second number indicates the decision number. For Boolean decision points, two sets of parentheses are visible and are filled in with a T or F when the True or False outcomes have been tested. For decision points that have multiple components (e.g., **`a || b`**), each sub-condition is displayed on a separate line. The sub-conditions are numbered **`.1`**, **`.2`** to **`.n`**, where **`n`** is the total number of sub-conditions.

For MC/DC or STATEMENT+MC/DC coverage, the left margin includes a red arrow that, when clicked, opens the Equivalence Matrices below the associated subprogram in the Coverage Viewer.



Function Coverage

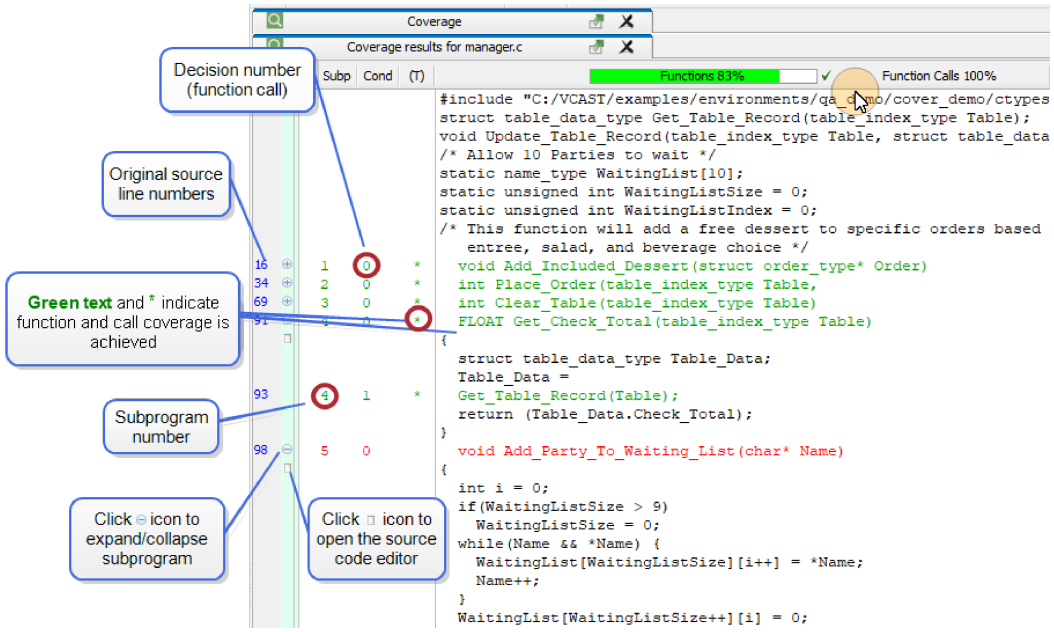
When Function coverage is active, the Coverage Viewer looks like the following example:



The numbers along the left-hand margin indicate the subprogram number. An asterisk (*) indicates that the function was covered (along with the color-coding). A subprogram is considered covered 100% in Function coverage if it is entered during test execution, and 0% if it is not.

Function and Function Call Coverage

When Function and Function Call coverage is active, the Coverage Viewer looks like the following example:



The numbers along the left-hand margin indicate the subprogram number. An asterisk (*) indicates that the function was covered (along with the color-coding). A subprogram is considered covered 100% in Function coverage if it is entered during test execution, and 0% if it is not. Function Call coverage identifies all of the calls that each function makes to other functions and identifies whether those calls have been tested.

Navigating Coverage Results

Four buttons are available in the toolbar to allow navigation of the coverage listing.

 Previous Covered Result (green)

 Next Covered Result (green)

 Previous Uncovered Result (red)

 Next Uncovered Result (red)

When the arrow buttons are clicked, the coverage source window scrolls to show the next covered or uncovered line. Partially-covered lines are considered both covered and uncovered. Contiguous blocks of lines are skipped over, until a line of a different block type is found or a line off-screen is found.

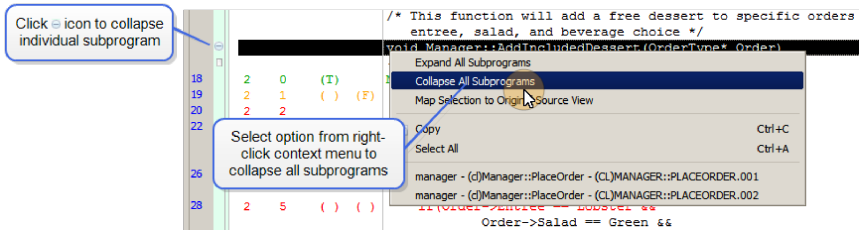
If there are no lines found, the message “Search reached bottom” or “Search reached top” appears, depending on which direction you were searching.

In addition, pressing the **Home** or **End** key brings the cursor to the beginning or end of the line, respectively; pressing **Ctrl+Home** or **Ctrl+End** brings the cursor to the beginning or end of the file,

respectively.

To Customize the Coverage Viewer

To customize the display in the Coverage Viewer, right-click anywhere in the window. The context menu appears allowing you to expand or collapse all subprograms in the viewer.



Use the Expand and Collapse menu items as “filters” or “layers” to get the exact display you want in the Coverage Viewer.

You can also expand and collapse individual subprograms by clicking the (collapse) and (expand) icons located on the left of the viewer.

Collapsing and expanding the subprograms in the Coverage Viewer does not affect any of the Coverage reports.

Map Line Selection to Original Source View

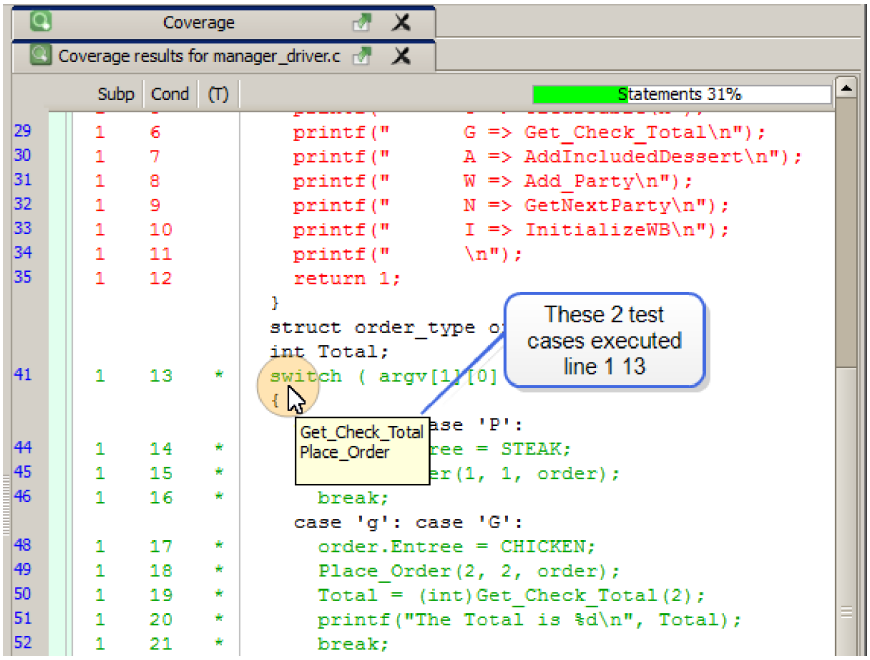
The Coverage Viewer displays the line numbers of the original source file on the left side of the viewer. If both the Coverage Viewer and the Original Source are open concurrently, any line selection in one viewer will be reflected in the other viewer.

By default, selecting a line in the Coverage Viewer, Original Source text editor or CBA Editor will automatically scroll to the corresponding line in any other open editor. To change the behavior, right-click within the Coverage Viewer or CBA Editor and uncheck the **Map Selection to Original Source View** option on the context menu.

Alternatively, in the Original Source text editor, right-click and uncheck the **Map Current Line to Coverage View** option on the context menu.

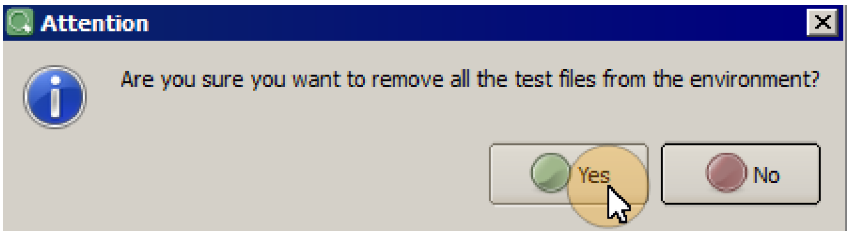
To Determine Which Test Result Covers a Line

When viewing code coverage, if you hover the cursor over a covered (green) or partially-covered (yellow) line, a pop-up list is displayed that shows the names of all test cases that cause that line to be executed. Hovering over a branch will display (T) if the test case executes the True branch, (F) if the test case executes the False branch, and (TF) if the test case executes both the True and False branches.



To Delete All Result Files

The **Environment => Remove All Result Files** command enables you to remove all test result files from the environment. Click **Yes** to confirm the deletion.




```
clicast -e <env> Cover RESult Remove <result-name> | ALL

Delete the result file named <result-name> from the Cover environment, or
all results, if "ALL" is specified.
```

Viewing Coverage Reports

The **Environment => View** commands reflect the item selected in the Environment View pane. For example, if you expand the Environment View pane to display the Source Directories and select the top-level node, then the report generated using the **Environment => View** menu includes coverage for all source files. You can narrow the content down by selecting particular source files and directories in the Environment View pane for which you want coverage.

VectorCAST provides the ability to control what coverage data appears in reports containing the Aggregate Coverage Report or Metrics Report.To set the coverage data used, select **Environment**

=> View => Coverage data used in reports from the Menu Bar. Then, select one of the sub-menu options:

- **All results** (default) - Use coverage data from all test results in the environment, even if unchecked (unselected) when generating reports.
- **Only checked results** - Use coverage data only from the checked (selected) test results when generating reports.

The setting is saved in the Cover environment's project file (**`.vcp`**), so it applies to that specific environment, and carries over to any other environment created in that instance of VectorCAST.



Note: This option does not apply to reports generated via **`clicast`**, which uses all results.

To View the Metrics Report

For each unit selected in the Source Files Pane, the Metrics Report displays a metrics table giving the complexity and the level of coverage achieved for each subprogram in the selected unit(s).

The report is sensitive to what level is selected in the Source Files pane. If you select the top-level node, the Metrics table includes coverage achieved by all test results. You can narrow the content down by selecting particular source files in the Source Files pane for which you want coverage achieved. Once you have made a selection, choose **Environment => View => Metrics Report**.

The report contains coverage data from all test results within the selection, regardless of whether they are toggled on to show coverage or not.

If coverage data is not available, then only the Unit, Subprogram, and Complexity columns are present in the Metrics Report.

```
clicast -e <env> [-u <unit>] Cover Report Unit_metrics [<outputfile>]

Extract the Metrics report for all units or the specified unit to standard
output or the specified output file. If VCAST_CUSTOM_REPORT_FORMAT is HTML
and <outputfile> is not specified, the file is saved to metrics_
report.html.
```

If the report format is HTML, the Metrics Report looks similar to the following:

Metrics Report

Configuration Data

Environment Name	cover_demo
Date of Report Creation	14 FEB 2019
Time of Report Creation	1:55:31 PM

Metrics

Unit	Subprogram	Complexity	Statements
manager.c	Add_Included_Dessert	3	3 / 4 (75%)
	Place_Order	5	12 / 17 (70%)
	Clear_Table	2	12 / 12 (100%)
	Get_Check_Total	1	2 / 2 (100%)
	Add_Party_To_Waiting_List	3	0 / 7 (0%)
	Get_Next_Party_To_Be_Seated	2	0 / 3 (0%)
TOTALS	**6**	**16**	**29 / 45 (64%)**
database.c	Get_Table_Record	1	1 / 1 (100%)
	Update_Table_Record	1	1 / 1 (100%)
TOTALS	**2**	**2**	**2 / 2 (100%)**
GRAND TOTALS	**8**	**18**	**31 / 47 (65%)**

Pairs Status

When MC/DC or STATEMENT+MC/DC coverage is initialized, an additional “Pairs Status” column is added to the Metrics Report section. MC/DC Pairs consists of total number of MC/DC pairs satisfied compared to the total number of MC/DC pairs in the unit.

If the report format is HTML, the report looks similar to the following:

Metrics Report

Configuration Data

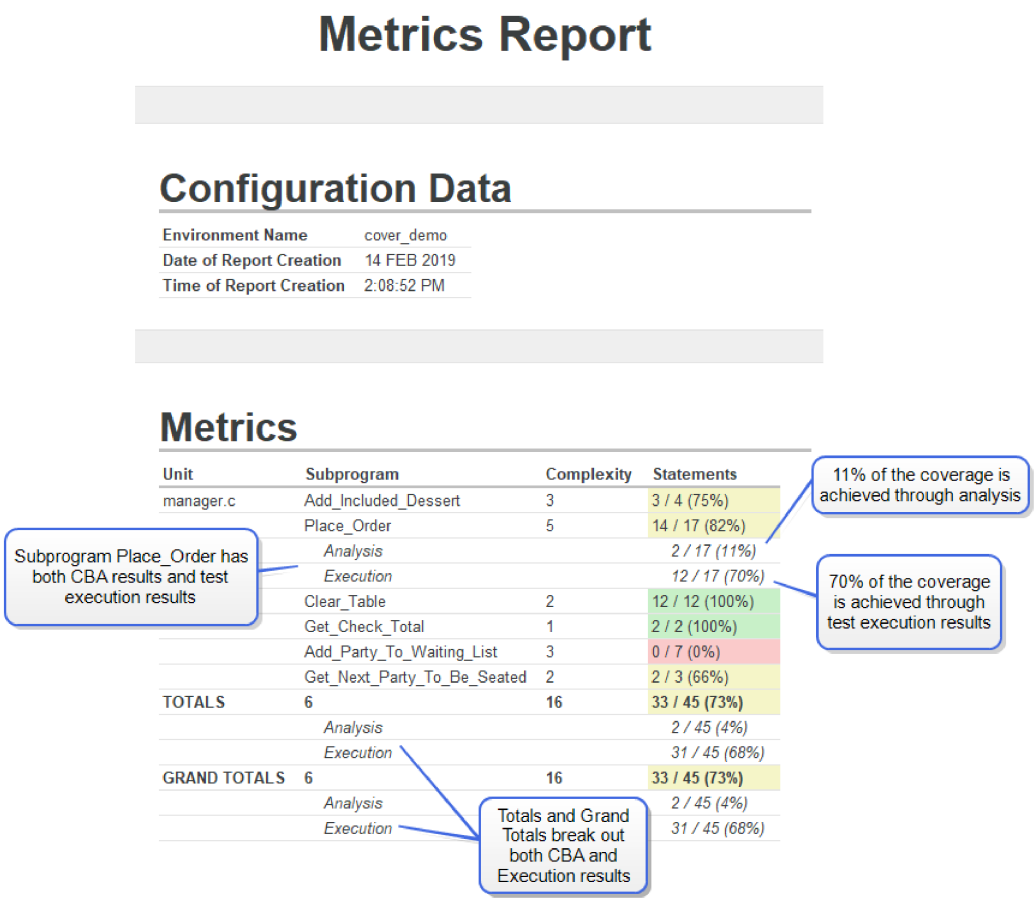
Environment Name	cover_demo
Date of Report Creation	14 FEB 2019
Time of Report Creation	2:01:32 PM

Metrics

Unit	Subprogram	Complexity	Statements	Branches	Pairs
manager_driver.c	main	15	28 / 49 (57%)	9 / 19 (47%)	0 / 1 (0%)
TOTALS	**1**	**15**	**28 / 49 (57%)**	**9 / 19 (47%)**	**0 / 1 (0%)**
manager.c	Add_Included_Dessert	3	3 / 4 (75%)	10 / 17 (58%)	2 / 6 (33%)
	Place_Order	5	14 / 22 (63%)	3 / 6 (50%)	
	Clear_Table	2	12 / 12 (100%)	5 / 5 (100%)	1 / 1 (100%)
	Get_Check_Total	1	2 / 2 (100%)	1 / 1 (100%)	
	Add_Party_To_Waiting_List	3	0 / 7 (0%)	0 / 11 (0%)	0 / 3 (0%)
	Get_Next_Party_To_Be_Seated	2	2 / 3 (66%)	3 / 5 (60%)	0 / 1 (0%)
TOTALS	**6**	**16**	**33 / 50 (66%)**	**22 / 45 (48%)**	**3 / 11 (27%)**
database.c	Get_Table_Record	1	1 / 1 (100%)	1 / 1 (100%)	
	Update_Table_Record	1	1 / 1 (100%)	1 / 1 (100%)	
TOTALS	**2**	**2**	**2 / 2 (100%)**	**2 / 2 (100%)**	
GRAND TOTALS	**9**	**33**	**63 / 101 (62%)**	**33 / 66 (50%)**	**3 / 12 (25%)**

Covered By Analysis Results

In the Metrics Report, when CBA results are present in the environment, they are displayed in italics in the row below the subprogram and the coverage achieved by test execution is displayed below the CBA results.



To View the Function Call Report

The Function Call Report is available for coverage types Statement, Statement + Branch or Statement + MC/DC.

The first section of the report displays by function, showing where the corresponding calls to the function are made from, if each call is covered, and the percentage of covered calls.

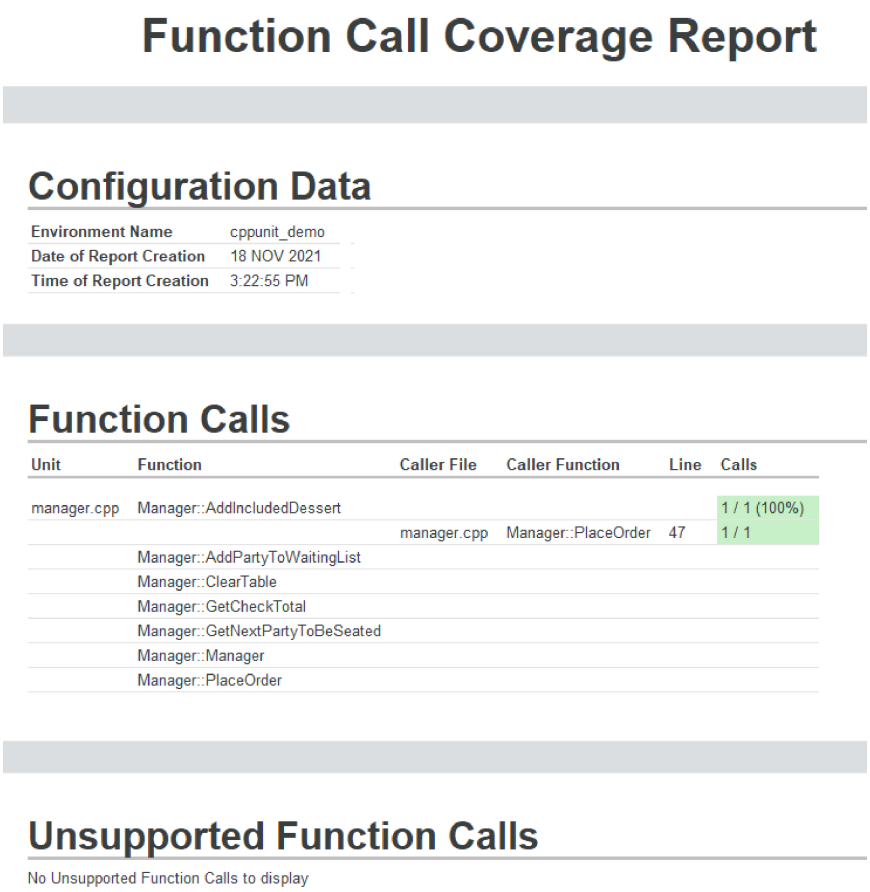
The second section of the report displays function calls in which the caller function cannot be determined. For example, C++ Virtual functions are shown in the Unsupported Function Calls section of the report for this reason.

To access the Function Call Report, right-click on the source file in the Source Files pane and select **Environment => View => Metrics Report**. Note that this option is only available when function call coverage is present.

```
clicast -e <env> [-u <unit>] Cover REPort FUNCTION_Call [<outputfile>]

Create a Function Call Report and output to HTML file <env>_function_call_
report.html, standard output as text, or to the specified output file. If
the Unit parameter is specified, the Function Call table includes only the
specified unit; otherwise, the Function Call table includes all units in
```

```
the environment.
```

The following is an example Function Call Report.



Function Call Coverage Report

Configuration Data

Environment Name	cppunit_demo
Date of Report Creation	18 NOV 2021
Time of Report Creation	3:22:55 PM

Function Calls

Unit	Function	Caller File	Caller Function	Line	Calls
manager.cpp	Manager::AddIncludedDessert				1 / 1 (100%)
		manager.cpp	Manager::PlaceOrder	47	1 / 1
	Manager::AddPartyToWaitingList				
	Manager::ClearTable				
	Manager::GetCheckTotal				
	Manager::GetNextPartyToBeSeated				
	Manager::Manager				
	Manager::PlaceOrder				

Unsupported Function Calls

No Unsupported Function Calls to display

To View a Basis Paths Report

The **Environment => View => Basis Path Report** command enables you to view the results of the computation of test paths for test case building. The contents of this report reflect the items selected in the Test Case Tree.

If this menu item is dimmed, choose **Tools => Basis Path => Generate Tests.**

The number of paths identified equals the code complexity, also referred to as V(g). The code complexity corresponds to the number of test cases that must be developed to exercise the unique paths in a subprogram.

This information can also be viewed by accessing the Basis Paths tab of the Coverage window.

```
clicast -e <env> [-u <unit>] TOols Basis_path [<outputfile>]

Extract the basis path analysis for all units or the specified units to
```

```
standard output or the specified file. If VCAST_CUSTOM_REPORT_FORMAT is
HTML and <outputfile> is not specified, the file is saved to <env>_basis_
path_report.html.
```

Understanding Basis Paths

The output from the Basis Path Analysis consists of two parts: the list of basis paths and an associated annotated source listing. The following example shows a single basis path along with the corresponding annotated source listing for the example.

```
  Test Path 1
         (1) if ( value < min ) ==> FALSE
         (3) if ( value > max ) ==> FALSE
  Test Path 2
         (1) if ( value < min ) ==> FALSE
         (3) if ( value > max ) ==> TRUE
  Test Path 3
         (1) if ( value < min ) ==> TRUE

    Complexity: 3

  Source Code Listing

                int Compute (int min, int max, int value) {
                    int return_value;
1 0     ( )   Compute
1 1     ( )( )    if ((value < min)){
                        return_value = min;
                    }
                    else
1 3     ( )( )       if ((value > max)){
                           return_value = max;
                       }
                       else{
                           return_value = value;
                       }
                    return return_value;
                }
```

To Build Test Cases from Basis Path Analysis

When using the basis path analysis to build a test case, you should build at least one test case for each basis path. The test case should contain the collection of data that will force the conditions listed in the basis path listing to be set to the listed value.

In most instances, the value of the branch point condition will make use of a subprogram call, or some intermediate data computation. In these cases, more work is required to find the value or values required to force a condition to the desired value.

Basis paths are determined by parsing the code and applying the basis path algorithm to the list of

branch point nodes that result. As a result, it may be impossible to satisfy a particular branch point when building test cases. The following examples illustrate this point:

Example 1:

```
const int road_speed_limit = 55;
const int truck_speed_limit = 50;
int index;

if (truck_speed_limit <= road_speed_limit)
full_throttle ();
else
half_throttle ();
```

Because truck_speed_limit is a constant, and always less than road_speed_limit, the “else” condition can never be reached.

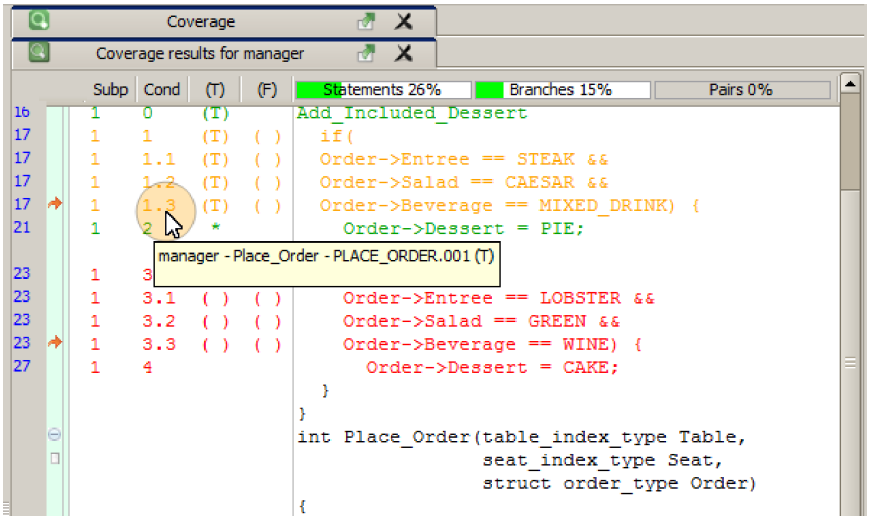
Example 2:

```
for (index=1; index<10; index++)
keep_going();
```

Because the loop boundaries are constant, the loop will always get executed and thus the basis path calling for the loop to not be entered cannot be satisfied.

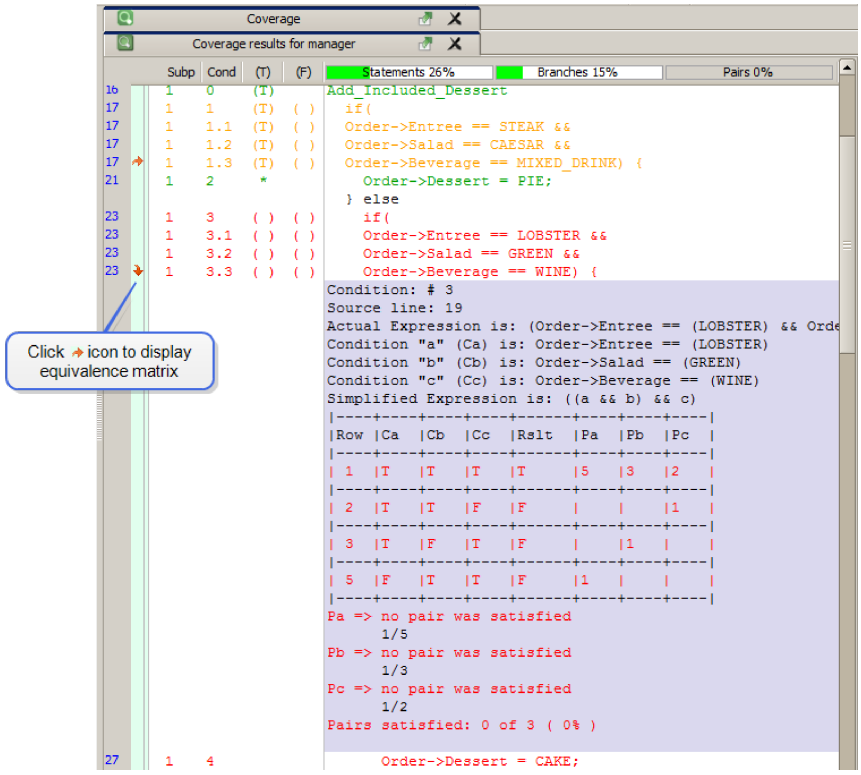
MC/DC Coverage

When you have the Coverage Viewer open, and MC/DC or STATEMENT+MC/DC coverage is initialized, if you hover your cursor over any fully or partially covered line of code, a tool tip displays the test case(s) which cover the line.



For MC/DC or STATEMENT+MC/DC coverage, the left margin includes a red arrow that, when

clicked, opens the Equivalence Matrix below the associated subprogram in the Coverage Viewer.



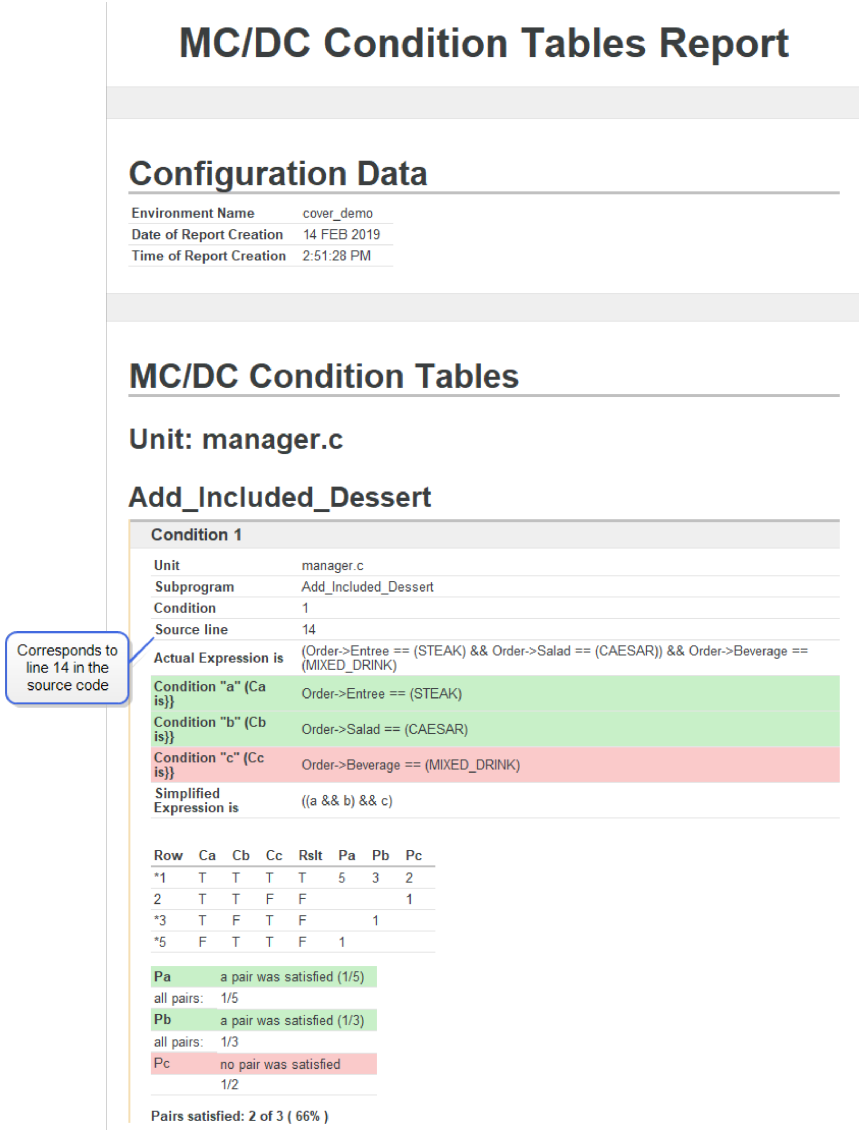
Clicking the arrow again closes the panel. The matrices can also be viewed using the **Environment => View => MC/DC Equivalence Report** and viewing the MC/DC Condition Tables Report.

To View the Equivalence Matrices Report

Choose **Environment => View => MC/DC Equivalence Report** to display the MC/DC Condition Tables report. This report includes the equivalence matrices for all units instrumented for MC/DC or DO-178B Level A coverage in the environment as well as the MC/DC equivalence pair status.

```
clicast -e <env> [-u <unit>] Cover Report Mcdc [<outputfile>]

Extract MC/DC Equivalence matrices for all units or the specified unit to
standard output or the specified output file. If VCAST_CUSTOM_REPORT_FORMAT
is HTML and <outputfile> is not specified, the file is saved to mcdc_
tables_report.html.
```

If the report format is HTML, the report looks similar to the following:



The **Source line #** refers to the source code line in the Coverage Viewer that corresponds to the expression being evaluated in the matrix. After noting the line number, right-click the unit in the Source Files pane, and choose **Open Coverage Viewer**. Then, in the Coverage Viewer, type **Ctrl+F** (or choose **Edit => Find** from the Menu Bar), and type in the line number.

Understanding MC/DC Analysis

The output from the MC/DC analysis consists of two parts: an equivalence pair matrix, and an annotated source listing.

For each boolean conditional there is a corresponding matrix. Each row in the matrix shows a unique combination of values for the components of the complex conditional, as well as the result of the condition for each combination. The equivalence pairs are pairs of rows which demonstrate that the result of the conditional changes from True to False as one component changes, while all other components are held constant.

The Equivalence Pair Matrix shows both the static pair analysis and the pair coverage achieved. The Pair A through Pair *n* columns (where *n* is the number of sub-conditions) show candidate row pairs for proving condition components. An asterisk next to a row number indicates that that row has been covered. When an equivalence pair has been covered, the Pa through P*n* summary lines reflect that information.

The following example shows a single complex conditional with the corresponding equivalence pair matrix.

```
              void MCDC_Example (int a, int b)
              {
                  int local;

4 0     (T)   MCDC_Example

4 1     (T)(F)    if ((
4 1.1   (T)(F)        a &&
4 1.2   (T)( )        b )  ) {
                              local = 1;
                  }                   }
===== Unit: manager
===== Subprogram: MCDC_Example
===== Condition number 1
Source line: 63
Actual Expression is: ( a && b )
Condition "a" (Ca) is: a
Condition "b" (Cb) is: b
Simplified Expression is: ( a && b )
|-----+-----+-----+-----+-----+-----|
|Row |Ca  |Cb  |Rslt |Pa  |Pb  |
|-----+-----+-----+-----+-----+-----|
|-----+-----+-----+-----+-----+-----|
|*1  | T  | T  | T  |3   |2   |
|-----+-----+-----+-----+-----+-----|
| 2  | T  | F  | F  |    |1   |
|-----+-----+-----+-----+-----+-----|
|*3  | F  | T  | F  |1   |    |
|-----+-----+-----+-----+-----+-----|
|*4  | F  | F  | F  |    |    |
|-----+-----+-----+-----+-----+-----|
Pa = 1/3 => a pair was satisfied
Pb = 1/2 => no pair was satisfied
```

This source listing shows that the “if” condition has been executed with A=True, B=True (Row 1); A=False, B=True (Row 3); and A=False, B=False (Row 4). The MC/DC branch coverage is 5/6. Five out of a total of six outcomes have been satisfied, and the pair coverage is 1 of 2 pairs satisfied.

The **Source line #** refers to the source code line in the Coverage Viewer line that corresponds to the expression being evaluated in the matrix. After noting the line number, right-click the referenced unit, and choose **Open in a New Window**. Then type **Ctrl+G** (or choose **Edit => Goto Line**), and type the line number.

To View an MC/DC Report

The **Tools => Basis Path => View Basis Path** command enables you to view the MC/DC test case paths.

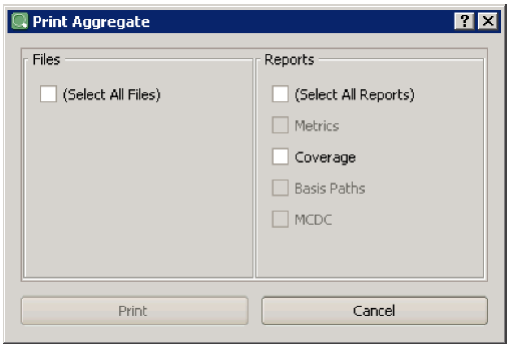
This information can also be viewed by accessing the MC/DC tab of the Coverage window.

```
clicast -e <env> [-u <unit>] TOols Basis_path [<outputfile>]

Extract the MC/DC path analysis for all units or the specified units to
standard output or the specified file. If VCAST_CUSTOM_REPORT_FORMAT is
HTML and <outputfile> is not specified, the file is saved to <env>_basis_
path_report.html.
```

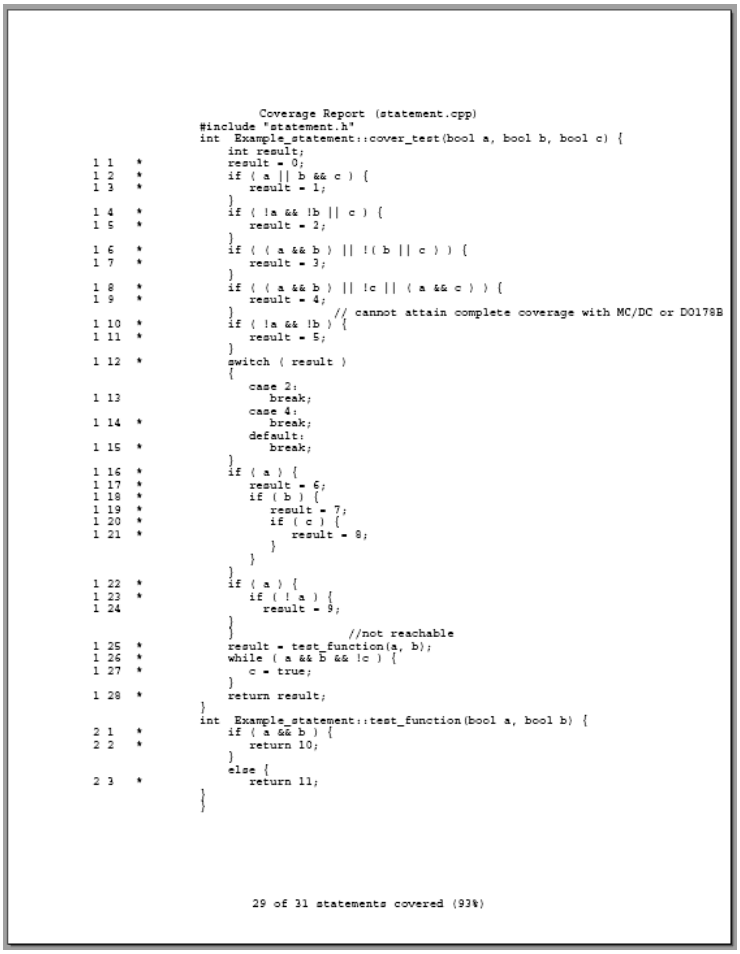
To Print Coverage Reports

The **File => Print => Aggregate Report** command enables you to print the total coverage information from all of the test results that have been added to the environment. A source file must be instrumented for a coverage type and open in the Coverage Viewer for it to be printed.

Choose a source file or files first, then specify which reports you want to print in the Aggregate Report.



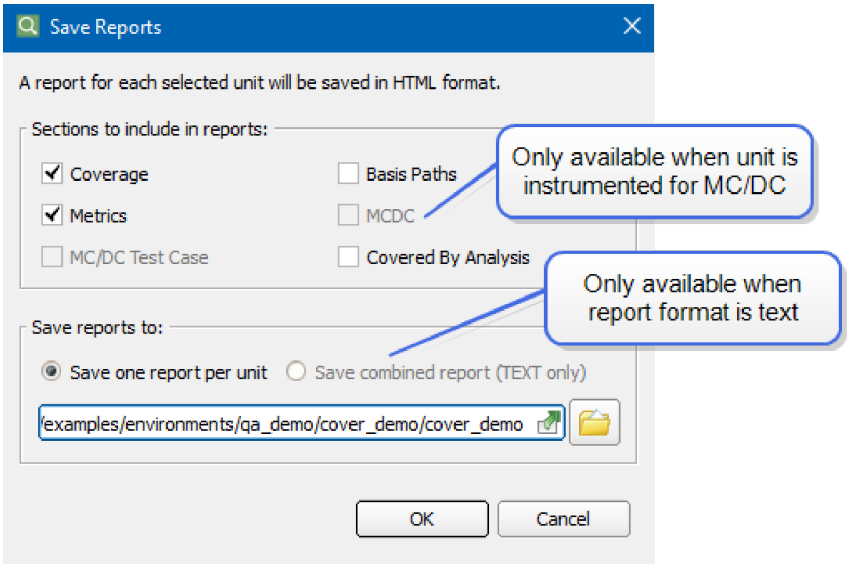
An example of the first page of an Aggregate Report that includes Metrics, Coverage, and Basis Paths is shown below.



```
                    Coverage Report (statement.cpp)
                #include "statement.h"
                int Example_statement::cover_test(bool a, bool b, bool c) {
                    int result;
1 1   *             result = 0;
1 2   *             if ( a || b && c ) {
1 3   *                result = 1;
                    }
1 4   *             if ( !a && !b || c ) {
1 5   *                result = 2;
                    }
1 6   *             if ( ( a && b ) || !( b || c ) ) {
1 7   *                result = 3;
                    }
1 8   *             if ( ( a && b ) || !c || ( a && c ) ) {
1 9   *                result = 4;
                    }                    // cannot attain complete coverage with MC/DC or DO178B
1 10  *             if ( !a && !b ) {
1 11  *                result = 5;
                    }
1 12  *             switch ( result )
                    {
                       case 2:
1 13                      break;
                       case 4:
1 14  *                   break;
                       default:
1 15  *                   break;
                    }
1 16  *             if ( a ) {
1 17  *                result = 6;
1 18  *                if ( b ) {
1 19  *                   result = 7;
1 20  *                   if ( c ) {
1 21  *                      result = 8;
                          }
                       }
                    }
1 22  *             if ( a ) {
1 23  *                if ( ! a ) {
1 24                     result = 9;
                    }
                    }                          //not reachable
1 25  *             result = test_function(a, b);
1 26  *             while ( a && b && !c ) {
1 27  *                c = true;
                    }
1 28  *             return result;
                }
                int Example_statement::test_function(bool a, bool b) {
2 1   *             if ( a && b ) {
2 2   *                return 10;
                    }
                    else {
2 3   *                return 11;
                }
                }




                    29 of 31 statements covered (93%)
```

To Save Coverage Reports to Disk

When you select multiple units and create an Aggregate Coverage report, it contains data for all units. Especially in HTML format, the size of the report may become unwieldy. To create a separate report for each unit selected and save them directly to disk, choose **Environment => View => Save Coverage Reports**, or **right-click** and choose **Save Coverage Reports**.



The reports are saved in HTML or TEXT format, depending on the setting of the Report Format option. Note that saving a combined report is only available when the report format is TEXT.

You can choose which sections you want included in each unit's report by checking the check boxes. The MCDC option is dimmed unless one of the selected units is instrumented for MC/DC or DO-178B Level A coverage type. Coverage and Metrics sections are included by default.

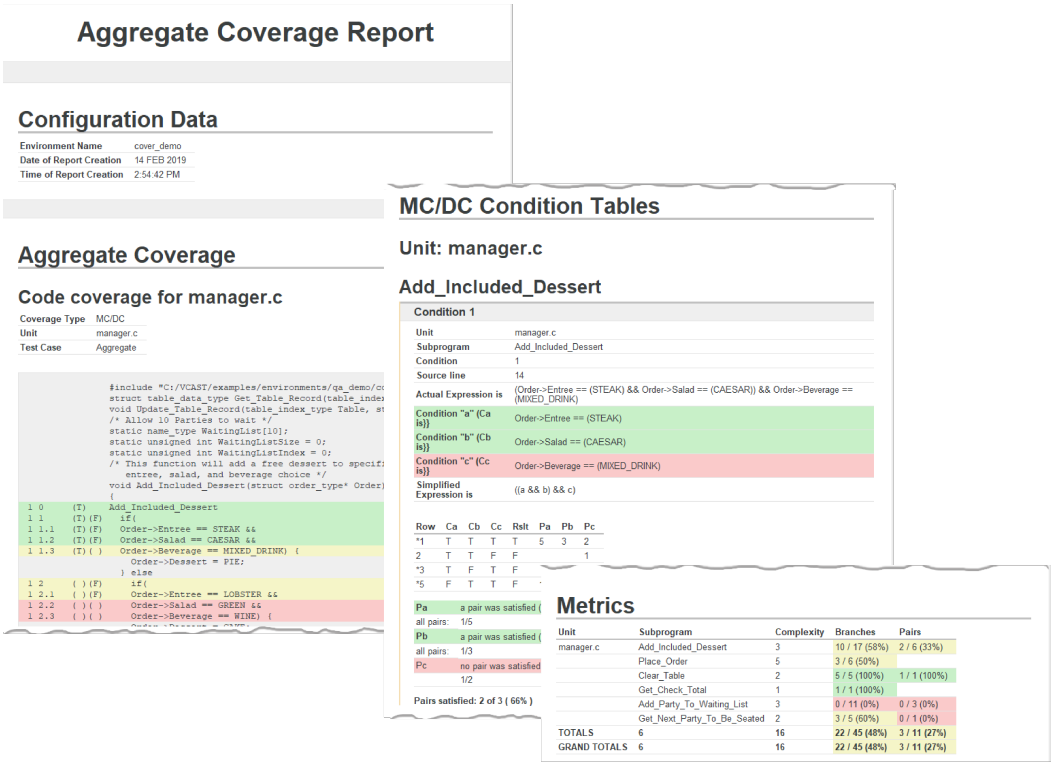
When you have selected the content, enter or browse to a directory in which to save the reports. If the directory is invalid, it is outlined in red.

When you click OK, one report for each unit is saved in the currently selected format, HTML or TEXT. The report names combine the environment name and the unit name: *ENV_unit*_report.html (or.txt).

When a report filename is used more than once, "Save Coverage Reports" inserts the unit number into the report filename.

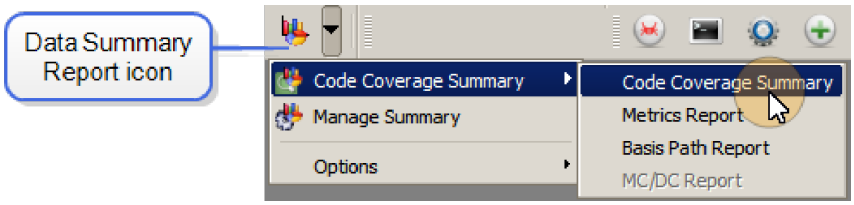
Note: If the selected source files have not yet been instrumented for coverage, Save Coverage Reports will show as dimmed on the right-click menu.

An example of the report file **`cover_demo_aggregate_coverage_report.html`** is shown below.



To View Code Coverage Summary

The Code Coverage Summary table allows the user to easily view coverage information for all the units in an environment or all the environments of a Manage project. To open the summary from the Toolbar, select **Code Coverage Summary** from the Data Summary Report icon drop-down menu.



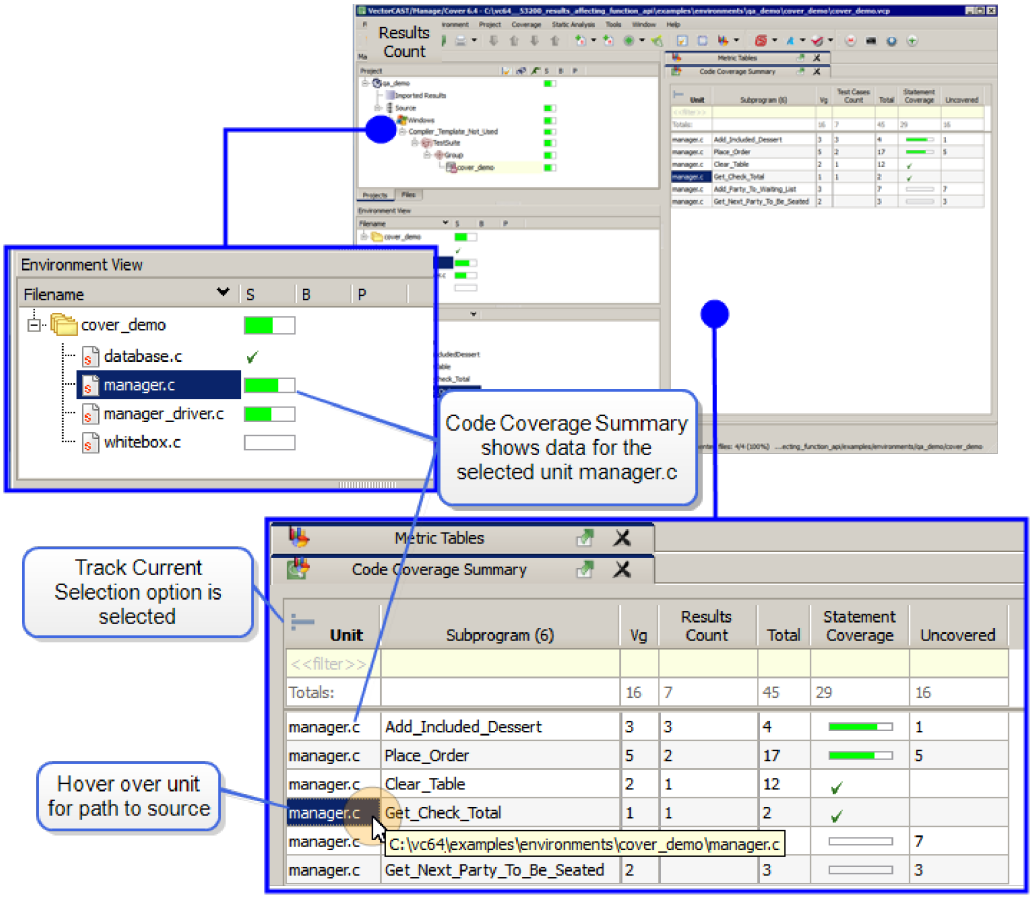
Alternatively, from the Menu Bar, select **Coverage => Code Coverage Summary => Code Coverage Summary**.

Viewing Code Coverage

By default, the contents of the Code Coverage Summary reflect the units selected in the Source Files pane. A tracking icon is displayed at the top of the Unit column indicating that the Code Coverage Summary is currently tracking selected units from the Source Files pane.

To override tracking of selected units in the Source Files pane, open the drop-down menu for the Data Summary Report and select **Options => Track Current Selection**. Remove the check next to the option. The tracking icon on the Summary table will change to gray to indicate that the

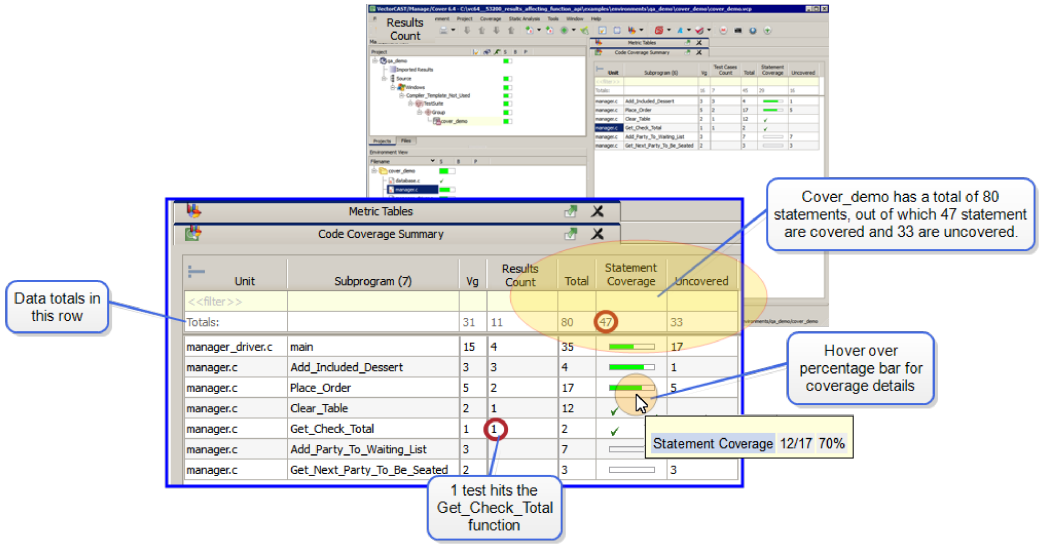
summary is now tracking all units in the Source Files pane.



The Code Coverage Summary table is dynamic. When the **Track Current Selection** option is enabled, as units are selected and deselected in the Source Files pane, the Code Coverage Summary table updates in real time reflecting the selections.

The data displayed in the Code Coverage Summary includes the Unit name, the Subprogram name, the Cyclomatic Complexity (Vg), the Results Count (showing the number of coverage results that touch each subprogram), and the achieved coverage for each coverage type.

Note: When I/O type is set to "Animation" (e.g. when using Basis Path coverage), the Test Cases Count column indicates the number of times a function is entered. This total can also include multiple slot iterations of a compound test.



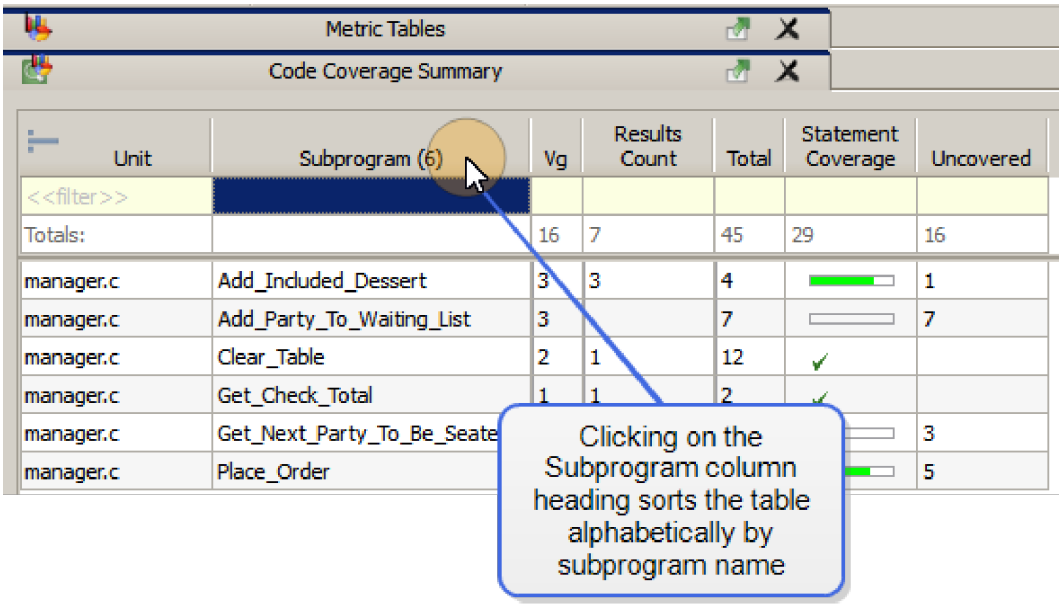
The Totals row at the top of the table displays the totals for each data column. In the example above, note that in the cover_demo environment we have a total of 80 statements, of which 47 statements are covered and 33 are uncovered.

The Summary table updates whenever coverage data is updated. For example, the table refreshes when coverage is initialized, or following reinstrumentation.

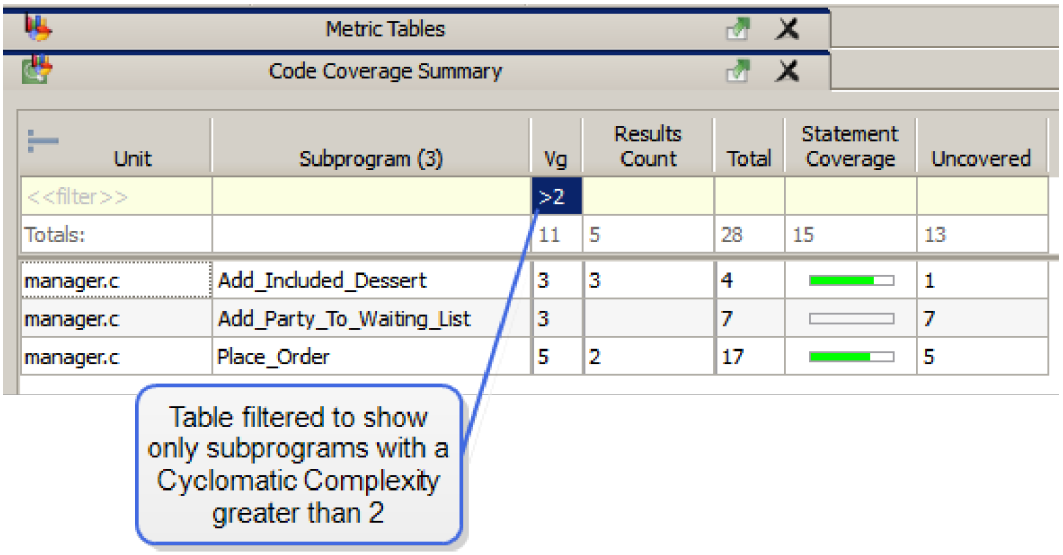
Double-clicking on a line in the Code Coverage Summary opens the corresponding UUT in the Coverage Viewer.

Sorting and Filtering Coverage Data

Sorting and filtering is available to locate data of interest. Sort by clicking on any column heading. The data will sort in alphabetic or numeric order, as appropriate. Clicking the heading again reverses the order.



Coverage data can be accessed all the way down to the individual subprogram level by filtering. Access the filter by typing into the top row of any column. Clear the filter by right-clicking in the top row and selecting **Clear Filter** from the context menu. In the example below, the table has been filtered to only show data for test cases with Cyclomatic Complexity greater than 2.



Filtering supports the following symbols: <, >, =. For example, the summary can be filtered to show only subprograms with a Cyclomatic Complexity greater than 2. Other examples of filtering inputs are:

10 - lists subprograms matching the specific value of "10" in the selected column

>50 - lists subprograms greater than the value of "50" in the selected column

<90 - lists subprograms less than the value of "90" in the selected column

=100 - lists subprograms matching the specific value of "100" in the selected column

< - lists subprograms with empty values in the selected column

> - lists subprograms with non-empty values in the selected column

= - lists subprograms with non-empty values in the selected column

Saving and Printing Code Coverage Summary

Saving the Code Coverage Summary in .html or .csv format is supported.

Printing the contents of the Code Coverage Summary is supported.

Viewing Execution Flow with Animated Coverage

Coverage Animation allows you to view the flow of execution for a test case in the Coverage Viewer. For example, you can view individual statements being executed as the test executes. If a unit calls a subprogram, you can view the execution flow to the called subprogram.

Coverage Animation displays the flow of control from one unit to another, using the test's coverage data. Calls to stubbed units are not included in Animation. Not-stubbed units are included if they have coverage initialized using **Coverage => Initialize Custom**.

Coverage can be animated for only one test case in the at a time. The item currently activated for Animation has a check in the box to the left of its name in the Test Case Tree. If you check another item, the Animation toolbar dims until you activate another test result for animation.



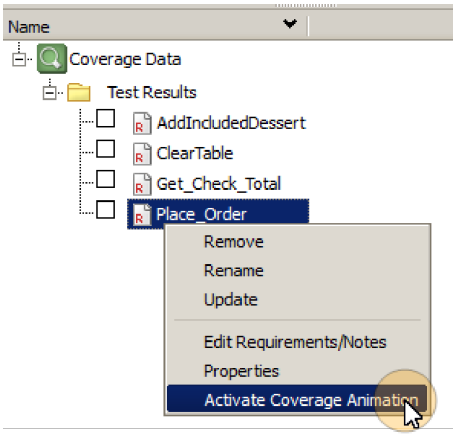
Note: If the Animation toolbar is dim, activate a test case for animation.

To Activate Coverage Animation

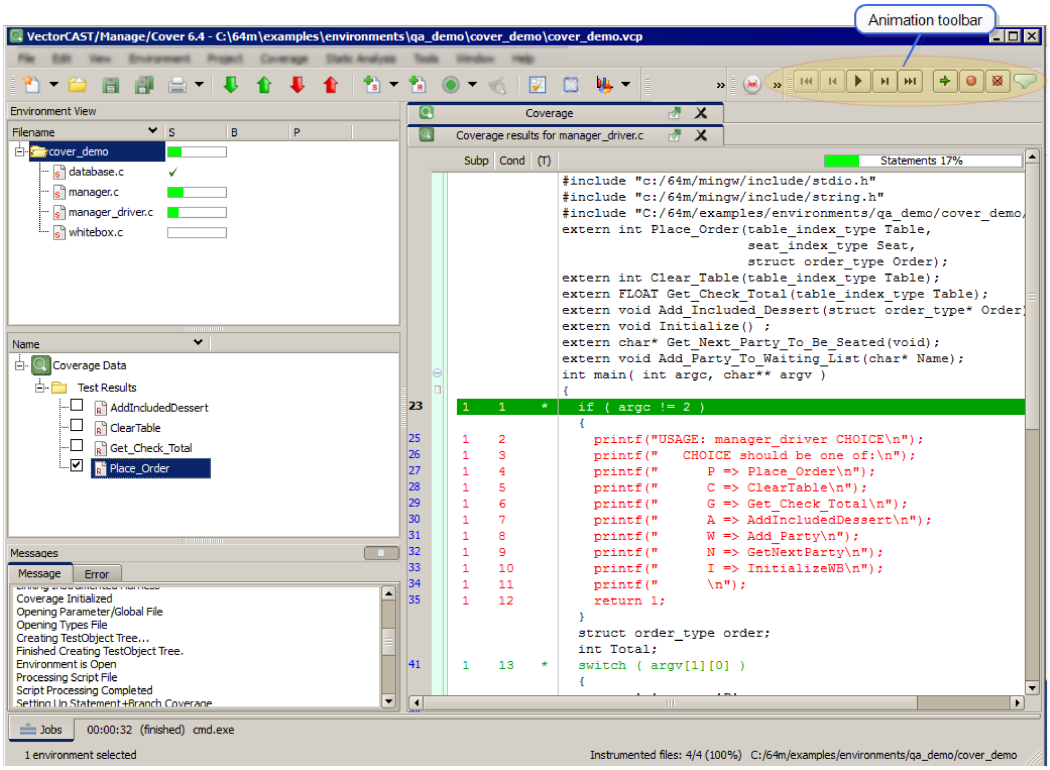
To use the Animation feature, select **Tools => Options** dialog, Coverage tab, and then select Animation for the **Coverage I/O Type**.

After setting the Coverage I/O type to Animation, instrument for a type of coverage. If you want to see each statement being executed, you would instrument for Statement coverage; if you want to see the choice made at each branch point, you would instrument for Branch coverage. To see each entry point, decision point, and statement covered, choose MC/DC, STATEMENT+MC/DC, or STATEMENT+BRANCH.

Next, execute a test case. Right-click the test case in the , and then choose **Activate Coverage Animation** from the popup menu:



The Coverage Viewer for the unit containing the first covered line in the test result opens, with the current line on the first covered line. The coverage checkbox next to the test case is checked.



If not already open, the Coverage Animation toolbar (pictured below) appears in the main toolbar.



To Play the Coverage Animation

Click the **Play** button

 to begin the animation. In the Coverage Viewer, the current line progresses steadily from the first covered line to the last. The speed of each step is 1 second. The source code in the Coverage Viewer scrolls up, keeping the current line in the middle. If a subprogram in another unit is called and that unit has coverage initialized, then its tab comes to the front of the Coverage Viewer and animation continues in that tab.

In the Coverage Viewer:

- the blue arrow  indicates the current line in the animation of control flow
- the green arrow  indicates the first line
- the red arrow  indicates the last line

The Animation Toolbar

When a test case is activated for animation, the Animation toolbar is displayed in the main toolbar. To display or hide the Animation toolbar manually, right-click the main toolbar and toggle **Coverage Animation**. If there is a checkmark, the toolbar is visible; otherwise it is hidden. Alternatively, choose **View => Dock Control** and toggle the setting for Coverage Animation.

The following controls are available on the Coverage Animation toolbar:

	**Rewind**	Returns the current-line indicator to the first executed line.
	**Back**	Returns the current-line indicator to the previous executed line.
	**Play**	Starts dynamic display of the execution; continues dynamic display after an interruption. See Pause and Breakpoint.
	**Pause**	Pauses dynamic display of the execution. See Continue and Play.
	**Next**	Moves the current-line indicator to the next executed line.
	**Fast Forward**	Moves the current-line indicator to the last line executed.
	**Continue**	Jumps the current-line indicator to the next breakpoint or to the last line of animation.

	**Set Breakpoint**	Sets a breakpoint at the current line.
	**Remove Breakpoint**	Removes a breakpoint from the current line.
	**Progress Indicator / Scroll Knob**	Indicates the relative position of the current line during dynamic display of the execution. Drag this knob to set the position of the current line.

To Set a Breakpoint

Setting a breakpoint causes the animated display of the execution flow to pause on the line marked with the breakpoint  . To set a breakpoint, select the line to make it current, then click the **Set Breakpoint** button  .

From here you can:

- play from this point 
- step to the next line in the flow of execution 
- jump to the next breakpoint  (or the end, if there is none)
- remove the breakpoint 

There is no limit to the number of breakpoints you can set.

Importing Coverage Results

To Import Coverage Results

You can import coverage data from another VectorCAST Cover environment, from a VectorCAST unit test environment which has coverage, or from a coverage script created by exporting coverage from another VectorCAST environment. To import coverage results from a VectorCAST unit test or Cover environment, choose **Coverage => Import Results from Environment**.

Preparing to Import Results

In order to import coverage, several conditions must be met.

- The same files must be used in both environments. VectorCAST compares the filenames and file checksums to ensure that the files are the same or copies of each other.
- Only the units that are present in both environments will have their coverage imported.
- The same type of coverage must be used in both environments.

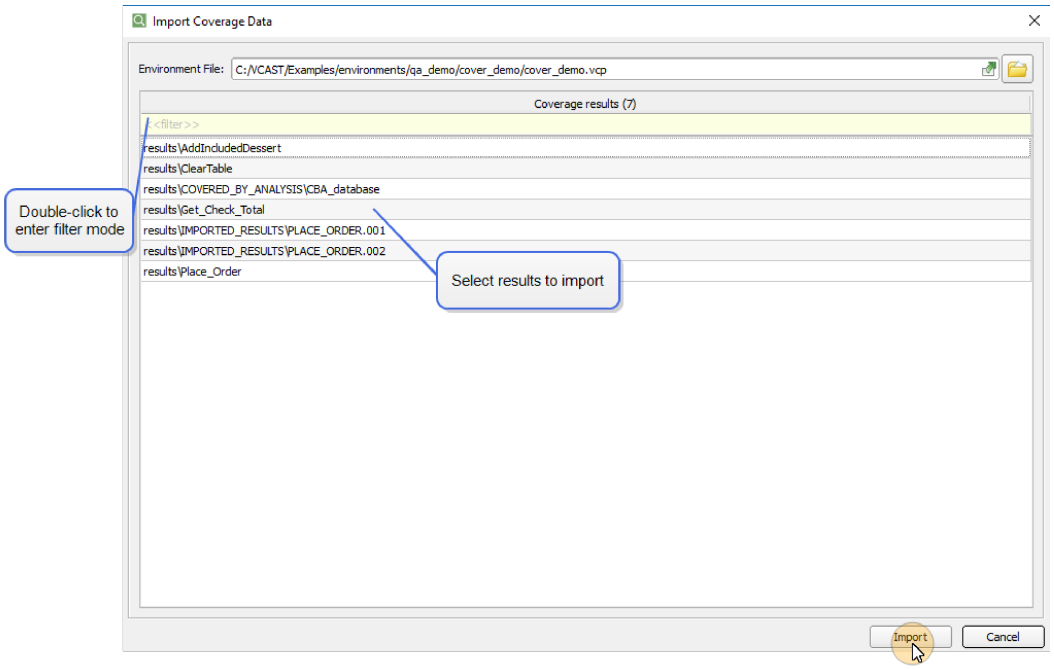
Importing the Results

Choose **Coverage => Import Results from Environment**, and a dialog appears which enables you to browse for a VectorCAST Cover environment (*.vcp) or a VectorCAST environment (*.vce). Note that you select an environment from which to import, not a test result.

Select the results files from that environment to import. You can choose to import the other environment's test execution results, imported results, and Covered By Analysis (CBA) results.

Test execution results are listed as **`results/<result name>`**. Imported results in the other environment are listed as **`results/IMPORTED_RESULTS/<result name>`**. CBA results are listed as **`results/COVERED_BY_ANALYSIS/<result name>`**. These names correspond to the locations where they are stored in the environment directory.

A filter is provided at the top of the pane making it easier to filter out some results when working with a large set of test results. By entering text or a regular expression in the <<filter>> field, the user can filter by name.



When you are ready to import the coverage results, click the **Import** button. To exit the dialog box without importing, click the **Cancel** button.

After the import is complete, a Coverage Import/Export log is displayed showing a series of status (S) or error (E) messages. Some of the messages you may see in the log file:

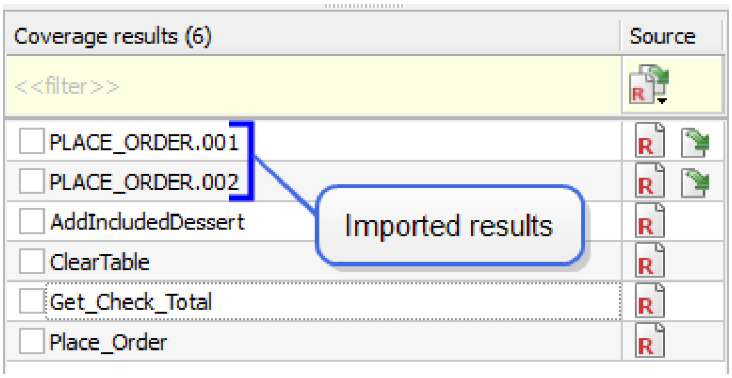
- **`(S) Source file matched`**
  The source file exists and was successfully matched.
- **`(S) No match for file <file> referenced in script file`**
  `>>> References to unit <unit number> will be ignored.`
  Not an error. The coverage results that were imported refer to a unit that is not present in the environment to which the results are being imported. This situation is typical if <file> is stubbed

in the environment to which the results are being imported, but not-stubbed in the other environment. Therefore, the coverage data for <file> are just being ignored.

- **(E) Coverage not on for source file**
  The source file exists in the environment to which results are being imported, but coverage is not initialized for that unit. An example is when there are non-stubbed dependent units in the environment but only the UUT was initialized for coverage. Use Initialize Custom to enable coverage for non-stubbed dependent units.
- **(E) No match found for source file**
  Although the file names may be the same, the checksums of the source files differ.
- **(E) Coverage types differ**
  The type of coverage in the importing environment is different than the type of coverage in the environment to which results are being imported. They must match for a successful import.
- **(E) No translatable data for result**
  A result was found in the importing environment, but the data did not match any source file in the current environment.
- **(S) Coverage data was loaded**
  A result file was successfully imported.

The Import Log file is named IMPORT.LOG and resides in the environment directory.

The imported results are displayed in the Coverage Results pane.



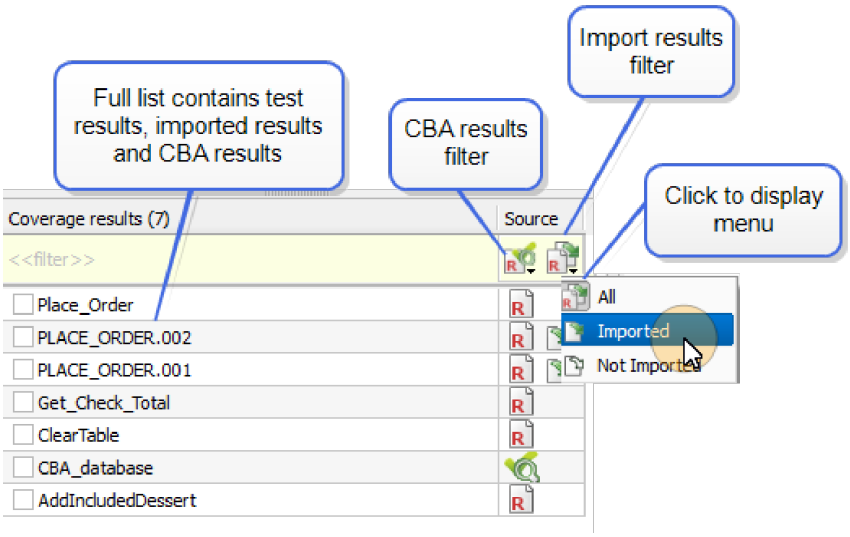
The Coverage Results pane is a filterable table, making it easier to filter out some results when working with a large set of test results. By entering text in the <<filter>> field, the list can be filtered by name. Partial matches are supported.

The list can also be filtered by source:

- Test Results
- Coverage By Analysis (CBA) Results
- Imported Results

By default, all results are displayed in the Coverage Results pane. Filter buttons are provided at the top of the Source column. For environments with test results and imported results, a filter button is displayed and the options are: All, Imported, and Not Imported. For environments with test results and CBA results another filter button is displayed, and the filter options are: All, CBA results, and Test results.To filter by source, click the appropriate drop-down menu in the Source column and select the

desired filter.



To Export a Coverage Script

To aid in regression testing, you can export a script to achieve the coverage results that you imported. Use **Coverage => Export Script** to create the script file.

There are several options. Choose **Coverage => Export Script => Imported Results** to create a script containing only results that were imported to the environment. VectorCAST looks in the **`<env>/results/IMPORTED_RESULTS`** directory for the data to export. **Choose Coverage => Export Script => Testcase Results** to create a script containing only results that are "native" to the environment. For Cover environments, that refers to test results added after executing the instrumented application. VectorCAST looks in the **`<env>/results`** directory for the data to export.

This script can be imported to VectorCAST Cover or into another VectorCAST environment. VectorCAST automatically generates a coverage script when regression test scripts are created if coverage results have been imported.

After the script file is exported, a Coverage Import/Export log is displayed showing a series of status (S) or error (E) messages. Some of the messages you may see in the log file:

- **`(S) Script creation started`**
  The process of creating the script file (.cvr) is beginning.
- **`(S) Looking for results directory IMPORTED_RESULTS`**
  VectorCAST uses the results in the directory IMPORTED_RESULTS, located in the environment directory, to create the script file containing imported results. This line is also (inaccurately) printed when exporting testcase results, even though they are not stored there.
- **`(E) Error reading imported results directory`**
  There was an error reading the directory IMPORTED_RESULTS, located in the environment directory. Make sure the results have not been deleted or corrupted.
- **`(E) No imported results found`**
  VectorCAST could not find any results file in the directory IMPORTED_RESULTS. The script file could not be successfully created.

- **(E) Script creation aborted**
  The script file could not be created due to other errors.
- **(S) Script creation completed**
  The script file was created successfully.

```
clicast -e <env> Cover EXPORT_Results_coverage <scriptfile>

Create a coverage script containing test execution results that are present
in <env>, including imported results. Coverage scripts have the extension
.cvr.
```

```
clicast -e <env> Cover EXPORT_Coverage <scriptfile>

Create a coverage script containing only imported coverage results that are
present in <env>.
```

```
clicast -e <env> Cover EXPORT_Test_coverage <scriptfile>

Create a coverage script containing only test execution results that are
present in <env>.
```

To Import a Coverage Script

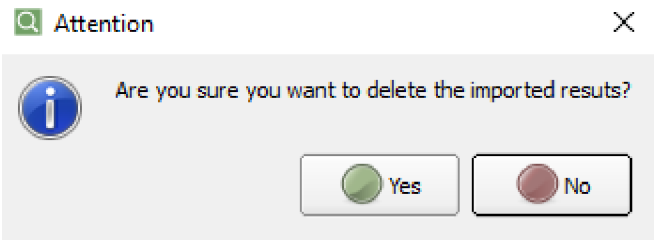
The **Coverage => Import Script** command imports a coverage script that was previously generated using the Export Script command. The default file extension is .cvr.

```
clicast -e <env> Cover IMPORT_Results <from_project|scriptfile(.cvr)>

Import results from Project.
```

Any errors that occur during importing of a coverage script are logged in a file named Import.log. To view this file, choose **Coverage => View Import Log**.

Delete Imported Results

To delete all imported coverage results, choose **Coverage => Delete All Imported Results**. A dialog appears asking you to verify that you want to continue with this operation, as shown below.



Clicking **No** cancels the operation without removing the imported coverage results. Clicking **Yes** removes the imported coverage results from the environment and the **IMPORTED_RESULTS** directory from the environment directory.

```
clicast -lc -e <env> Cover RESult REMove <result-name|IMPORTED|ALL>

Remove specified result file, Imported results (not CBA results), or ALL
result files (including imported results and CBA results).
```

Setting Compiler Options

When using C or C++ source files in a VectorCAST Cover environment, VectorCAST needs to know which compiler settings to use in order to preprocess the source files before instrumenting. The template settings are stored in the CCAST_.CFG file. For this reason, the C/C++ tab is displayed in the **Tools => Options** dialog. The Ada tab is present to enable you to change the option “Ada file extensions,” which is none-the-less stored in the CCAST_.CFG file.

Useful Coverage Options

In addition to storing compiler settings in the CCAST_.CFG file, VectorCAST Cover stores the settings for various coverage options there as well. The option settings from the Coverage tab in the Options dialog are stored in the CCAST_.CFG file, while the settings on the Cover Environment sub-tab are stored in the environment’s .vcp file.

The following table lists the options useful to VectorCAST Cover environments, and which file they are stored in. For more information, click on the Option Name.

Tab in Options Dialog	Option Name	Stored In File
Coverage tab, Options sub-tab	Coverage I/O type	CCAST_.CFG
Coverage tab, Options sub-tab	Save data in ASCII format in memory	CCAST_.CFG
Coverage tab, Options sub-tab	Maximum size for ASCII buffer	CCAST_.CFG
Coverage tab, Options sub-tab	Dump buffered coverage data on exit	CCAST_.CFG
Coverage tab, Options sub-tab	Use static memory allocation on the target	CCAST_.CFG
Coverage tab, Options sub-tab	Maximum covered subprograms	CCAST_.CFG
Coverage tab, Options sub-tab	Maximum MC/DC expressions	CCAST_.CFG
Coverage tab, Options sub-tab	Uncovered line indicator	CCAST_.CFG
Coverage tab, Options sub-tab	Animation play speed	CCAST_.CFG
Coverage tab, Options sub-tab, Instrumentation Options sub-tab	Show True and False in 'for' loops	CCAST_.CFG
Coverage tab, Options sub-tab, Instrumentation Options sub-tab	Instrument for function call coverage	CCAST_.CFG
Coverage tab, Options sub-tab, Instrumentation Options sub-tab	Instrument for function and function call coverage	CCAST_.CFG
Coverage tab, Options sub-tab, Instrumentation Options sub-tab	Instrument blocks for statement coverage	CCAST_.CFG
Coverage tab, Options sub-tab, Instrumentation Options sub-tab	Instrument implicit default case for switch-case blocks	CCAST_.CFG
Coverage tab, Options sub-tab, Instrumentation Options sub-tab	Consider case fallthrough for branch coverage	CCAST_.CFG

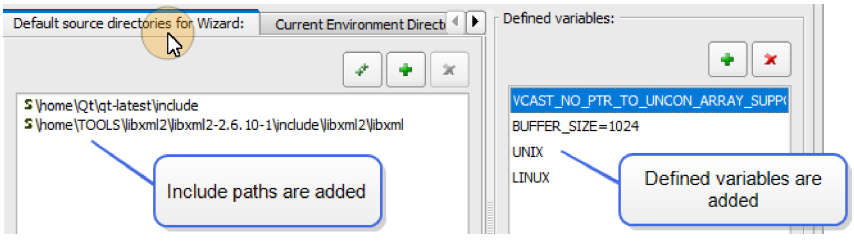
Tab in Options Dialog	Option Name	Stored In File
Coverage tab, Options sub-tab, Instrumentation Options sub-tab	Pack instrumentation storage	CCAST_.CFG
Coverage tab, Options sub-tab, Instrumentation Options sub-tab	Treat initialization of data couple as write	CCAST_.CFG
Coverage tab, Options sub-tab, Instrumentation Options sub-tab	Treat C++ catch blocks as branches	CCAST_.CFG
Coverage tab, Options sub-tab, Instrumentation Options sub-tab	Show inlines as covered in all units	CCAST_.CFG
Coverage tab, Options sub-tab, Instrumentation Options sub-tab	Instrument variable declarations in C	CCAST_.CFG
Coverage tab, Options sub-tab, Instrumentation Options sub-tab	Instrument using macros in c_cover.h	CCAST_.CFG
Coverage tab, Options sub-tab, Instrumentation Options sub-tab	Generate basis paths for constant branch conditions	CCAST_.CFG
Coverage tab, Options sub-tab, Instrumentation Options sub-tab	Avoid using comma operator in declarations	CCAST_.CFG
Coverage tab, Options sub-tab, MC/DC sub-tab	Abort instrumentation when absolute MC/DC subconditions limit exceeded	CCAST_.CFG
Coverage tab, Options sub-tab, MC/DC sub-tab	Simplified Condition Coverage	CCAST_.CFG
Coverage tab, Options sub-tab, MC/DC sub-tab	Use Masking MC/DC	CCAST_.CFG
Coverage tab, Options sub-tab, MC/DC sub-tab	Set absolute limit for MC/DC subconditions to 31	CCAST_.CFG
Coverage tab, Options sub-tab, MC/DC sub-tab	Maximum subconditions for MC/DC table pre-calculation	CCAST_.CFG
Coverage tab, Options sub-tab, MC/DC sub-tab	Maximum subconditions for MC/DC table row display	CCAST_.CFG
Coverage tab, Options sub-tab, MC/DC sub-tab	Use optimized MC/DC instrumentation	CCAST_.CFG
Coverage tab, Options sub-tab, MC/DC sub-tab	Optimized MC/DC storage threshold	CCAST_.CFG
Coverage tab, Options sub-tab, Misc sub-tab	Disable boolean cast in instrumentation	CCAST_.CFG
Coverage tab, Options sub-tab, Misc sub-tab	Coverage field width	CCAST_.CFG
Coverage tab, Options sub-tab, Misc sub-tab	Maximum coverage database cache (MB)	CCAST_.CFG
Coverage tab, Options sub-tab, Misc sub-tab	Asm functions behave as inlines (C/C++ only)	CCAST_.CFG
Coverage tab, Options sub-tab, Misc sub-tab	Run script after instrumentation	CCAST_.CFG
Coverage tab, Options sub-tab, Suppressed Coverable Functions sub-tab	Suppressed coverable functions	CCAST_.CFG

Tab in Options Dialog	Option Name	Stored In File
Coverage tab, Options sub-tab, Result Changes sub-tab	Keep Notes and Requirements - Covered By Analysis Data	.vcast_qt.ini
Coverage tab, Options sub-tab, Result Changes sub-tab	Keep Notes and Requirements - Imported Data	.vcast_qt.ini
Coverage tab, Options sub-tab, Result Changes sub-tab	Keep Notes and Requirements - Test Data	.vcast_qt.ini
Coverage tab, Options sub-tab, Input/Output sub-tab	Store Coverage Results in Directory	CCAST_.CFG
Coverage tab, Options sub-tab, Input/Output sub-tab	Append to Coverage Data File	CCAST_.CFG
Coverage tab, Options sub-tab, Input/Output sub-tab	Execution results go to stdout (C/C++ only)	CCAST_.CFG
Coverage tab, Cover Environment sub-tab	Auto generate test result names	*env*.vcp
Coverage tab, Cover Environment sub-tab	Append to coverage data file	*env*.vcp and CCAST_.CFG
Coverage tab, Cover Environment sub-tab	Starting Unit Number	*env*.vcp
Coverage tab, Cover Environment sub-tab	Types File Extension (Ada)	CCAST_.CFG
C/C++ tab, Language sub-tab	C Extensions	CCAST_.CFG
C/C++ tab, Language sub-tab	C++ Extensions	CCAST_.CFG
Ada tab	Ada File Extensions	CCAST_.CFG

Options: C/C++ Tab

The C/C++ tab consists of configurable items applicable to your compiler. The options defined here should be analogous to the options you use for source code compilation, when using the command line interface to the compiler. Pass your cursor over any of the edit boxes to see an explanation of that option in a tool-tip.

With an environment open, choose **Tools => Options**. Alternatively, click the  button on the toolbar and click the C/C++ tab.



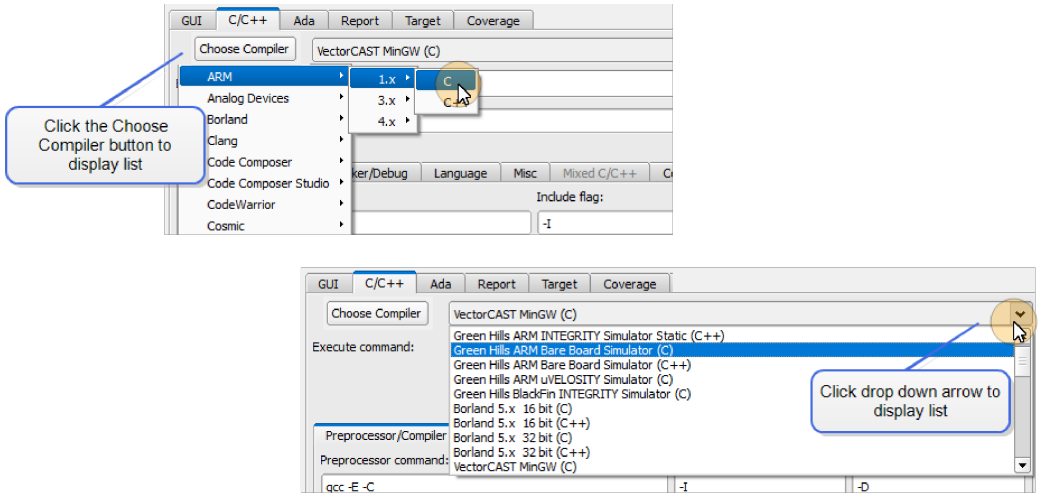
Tip: Selecting a Compiler Template loads in settings for other edit boxes automatically. You can modify these settings if necessary.

Compiler Template

Choosing a compiler template results in all options being set to default values for that compiler. There are two ways to specify a compiler template:

- Click the **Choose Compiler** button to display a cascading menu of compilers supported by VectorCAST.
- Choose from the drop-down list of compiler templates to the right of the **Choose Compiler** button.

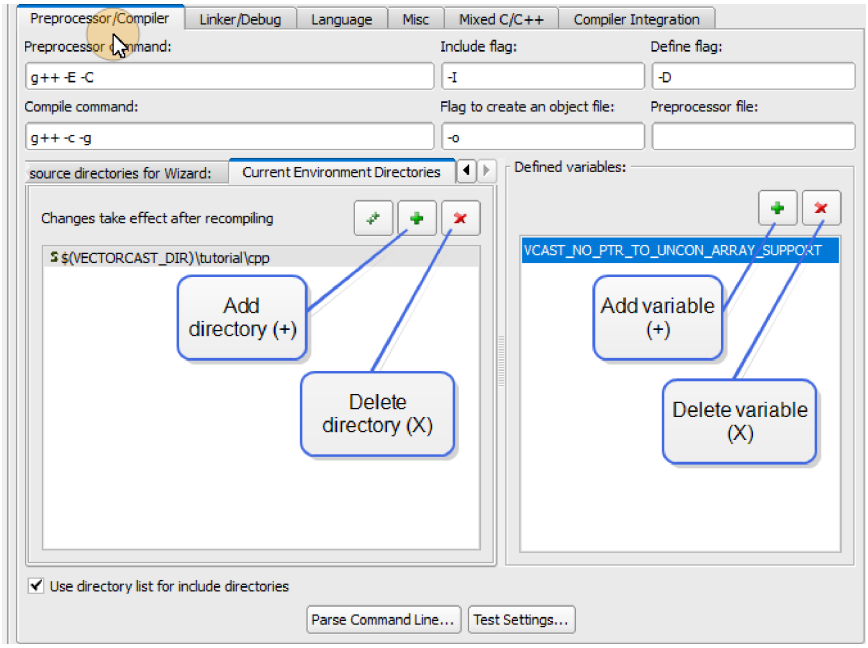
When using C++ language source files, you must choose a template that has “C++” following the name. For example, when using C source files and the GNU 3.3 compiler, choose **Choose Compiler => GNU Native => 3.3 => C**. When using C++ files, choose the template **Choose Compiler => GNU Native => 3.3 => C++,** as shown in the figure below.




```
clicast -lc TEMPLATE <template_name>

Set the compiler to <template_name>, and set all related options as well.
To get a list of legal templates, type clicast -lc TEMPLATE all.
```

Preprocessor/Compiler Tab

The **Preprocessor/Compiler** sub-tab on the **C/C++** tab contains settings pertinent to the compiler and preprocessor used to compile and link C and C++ source files. Most settings are specified by the compiler template, but you can add your own defined variables to the compiler command. In addition, the default list of Search, Library Include, and Type-handled directories can be specified here, to be used each time a new Create New Environment wizard is started.



Preprocessor Command

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Preprocessor/Compiler** tab.

The command used to preprocess a C or C++ source code unit.

```
clicast -lc option C_PREPROCESS_CMD <command>

<command> is the command used to preprocess C/C++ source files. Its default
value is set by the compiler template.
```

Include Flag

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Preprocessor/Compiler** tab.

The command line option used to specify include paths to the compiler or preprocessor.

```
clicast -lc option C_INCLUDE_FLAG <flag>

<flag> is the command line option used to specify include directories when
compiling C/C++ files, such as "-I", "/I". Its default value is set by the
compiler template.
```

Define Flag

With an environment open, choose **Tools => Options**, and click the **C/C++ tab**. Then click the **Preprocessor/Compiler tab**.

The command line option used to specify defined variables to the compiler or preprocessor.

In CLICAST and in the CCAST_.CFG file, this option is C_DEFINE_FLAG. Its default value is set by the compiler template.

Compile Command

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Preprocessor/Compiler** tab.

The compiler command used to compile C or C++ harness files.

```
clicast -lc option C_COMPILE_CMD <command>

<command> is used to compile C/C++ harness files. Its default value is set
by the compiler template.
```

Flag to Create an Object File

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Preprocessor/Compiler** tab.

Command line option for the compiler to create an object file. This option is used with the Visual Studio compiler when using the Stub None option.

```
clicast -lc option C_COMPILER_OUTPUT_FLAG <compiler output flag>

Command line option for the compiler to create an object file.
```

Preprocessor File

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Preprocessor/Compiler** tab.

Template for the name of the file created by the preprocessor (applicable to certain compilers). Consists of preprocessor output file name with "?" in place of source file name. For example, if the compiler convention is to preprocess the file 'manager.c' into 'manager.I', the value for this option would be '?.I' This option is only needed if your compiler does not send preprocessor output to stdout by default.

```
clicast -lc option C_PREPROCESS_FILE <preprocessor_file_template>

<preprocessor_file_template> is the template for the name of the file
created by the preprocessor. Its default value is set by the compiler
template.
```

Include Directories

With an environment open, choose Tools => Options, and click the C/C++ tab. Then click the Preprocessor/Compiler tab.

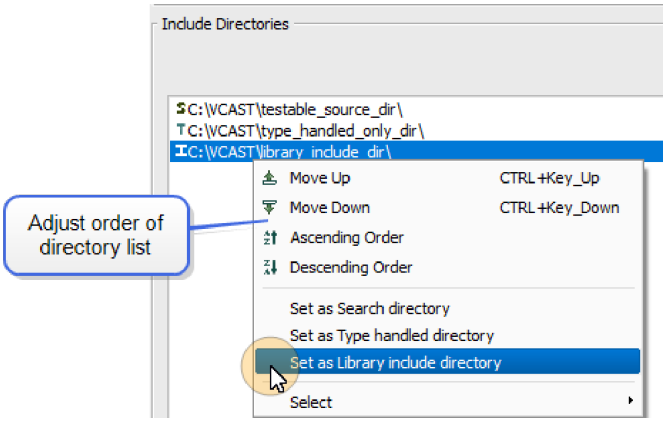
The search, library include, and type-handled directories are useful when you want to pre-process C/C++ source code units before instrumenting. A compiler template must be specified, as well as source directories to enable VectorCAST to find the header files it needs to complete the preprocessing.

To add a directory to the list, click the Add button or Add Recursively button . The Add Source Directory dialog appears; browse to the directory you want. To delete a path, first select the item you want to delete, then click the Remove button .

Once a directory is added, you can change the type. Right-click and choose:

- **Set as Search directory**, to make the path a default search directory for testable source units. It has an 'S' icon.
- **Set as Library include directory**, to make the path a default library include directory. It has an 'I' icon.
- **Set as Type-handled directory**, to make the path a default directory in which VectorCAST searches for types only, if needed. It has a 'T' icon.

The VectorCAST parser searches the directories in order, from top to bottom. You can adjust the order that the directories are listed by selecting a directory and pressing Ctrl+Up-Arrow or Ctrl+Down-Arrow.



Note: On Windows, directories with spaces in the path name are not supported. If you enter a path with a space, VectorCAST converts the paths to a DOS-safe path.

```
clicast -lc option TESTABLE_SOURCE_DIR <testable source directory>

Directory containing source code files that you would like to test or stub.
You can have more than one instance of this command in the CCAST_.CFG file,
each specifying a different directory. <testable source directory> becomes
a default search directory in the Create New Environment wizard.
```

```
clicast -lc option LIBRARY_INCLUDE_DIR <library include directory>

Directory to pass to the compiler during harness compilation. Any entity
residing here will not be defined by VectorCAST and therefore must be
linked in through a library. <library include directory> becomes a default
library include directory in the Create New Environment wizard. You can
```

have more than one instance of this command in the CCAST_.CFG file, each specifying a different directory.

clicast -lc option TYPE_HANDLED_SOURCE_DIR <type-handled source directory>

Directory containing source code files that you would like to parse for type information. Any entities residing here will not be defined by VectorCAST, and therefore must be linked in through a library. <type-handled source directory> becomes a default type-handled directory in the Create New Environment wizard. You can have more than one instance of this command in the CCAST_.CFG file, each specifying a different directory.

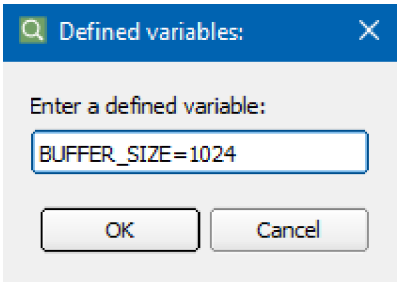
clicast -e <env> cover option
SET_Source_dirs <type> <path> [<path> ...]
[<type> <path> ...]

Add search directories, library include directories, and/or type-handled directories as default source directories in one command. <type> must be one of the following: **TESTABLE_SOURCE_DIR**, **LIBRARY_INCLUDE_DIR**, or **TYPE_HANDLED_SOURCE_DIR**. <path> can be a full path, a quoted full path that contains spaces, or a path relative to the working directory. Each <type> can have more than one <path> and can be mentioned more than once on the line. If the same <path> is used more than once, the last time it appears on the command line becomes the one that is used.

Defined Variables

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Preprocessor/Compiler** tab.

This option provides a list of preprocessor variables and definitions that are used when compiling the test harness. To add a variable to the list, click the **Add** button . A dialog appears, with an edit box for you to type the variable name and value. To enter a defined variable name that contains spaces, enclose the name in quotes, as in “one two”. When you are done, click **OK**.



To delete a defined variable, first select the item you want to delete, then click the **Delete** button . To edit an entry after it has been added, double-click it. Change the text and press **Enter**.

```
clicast -lc option C_DEFINE_LIST <list of definitions>

List of preprocessor variables and definitions to use when compiling C/C++
files. <list of definitions> is a space-separated list, as in “var1 var2”.
Its default value is set by the compiler template.
```

VectorCAST provides a customization feature which enables a user-defined function to be called at the very end of test harness processing, right before the **`exit()`** function is called. To use this feature, add the macro VCAST_CUSTOM_END to the defined variables list.

```
clicast -lc option C_DEFINE_LIST VCAST_CUSTOM_END=myEnd

where myEnd is a function with C linkage and with the signature: 'void
myEnd(void);'.
```

Alternatively, a **`#define`** directive in User Code can be used to define the macro, or VCAST_CUSTOM_END can be added to the Compile command as a command line macro.

Troubleshooting: Undefined symbol VCAST_exit

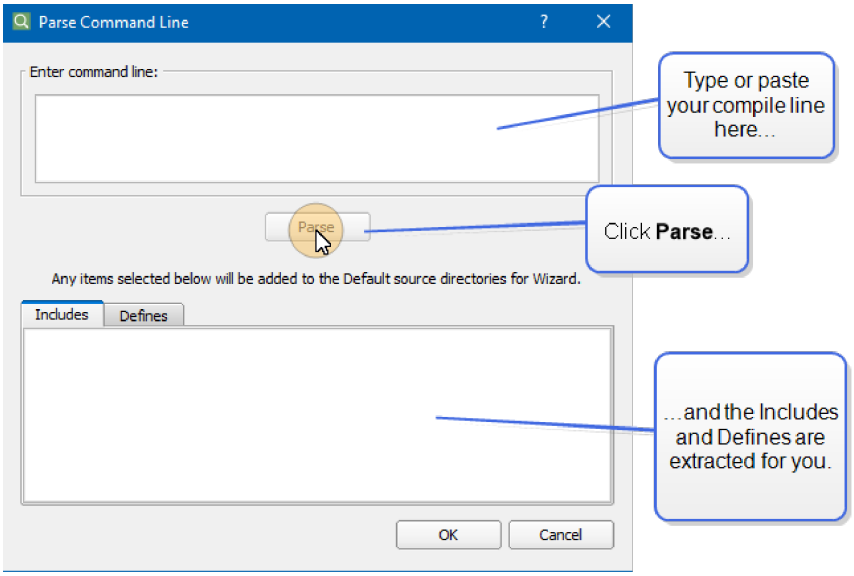
By default, the VectorCAST test harness #defines exit to VCAST_exit. If your source code #includes the C standard library (which #undefines exit), you may see a compile or link error while building the environment such as “undefined symbol VCAST_exit” or “undeclared ‘exit’.” Add the defined variable **`VCAST_DONT_RENAME_EXIT`** to the list of defined variables, and then either recompile or relink the test harness (**Environment => Recompile => Automatic** or **Environment => Relink**).

To Automatically Add Include Path and Defined Variables

Choose **Tools => Options**, and click the **C/C++ tab**. Then click the **Preprocessor/Compiler tab**.

The **Parse Command Line** button is convenient when you have a complex set of compiler options for the source code under test. This feature enables you to paste or type your compile command line into VectorCAST, and then have VectorCAST parse out the include paths and defined variables for you. Once that is accomplished, you can then add more or delete some, as needed to complete the configuration.

For example, you may be able to open the log file from the execution of your make and copy the compile line. Then, in VectorCAST, click the Parse Command Line button, and paste the text into the upper text edit area of the dialog that is displayed.

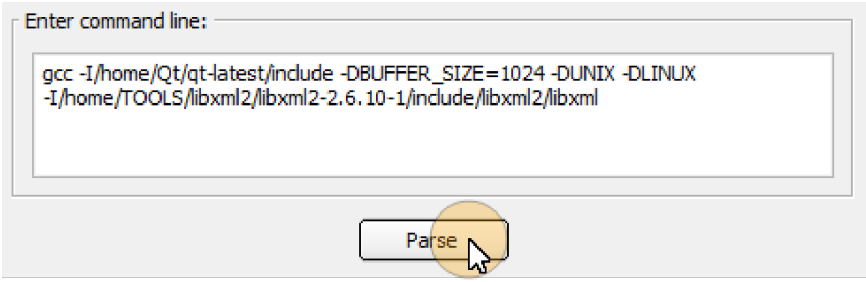


After pasting the command line, click the **Parse** button to have VectorCAST process the command. The **Includes** tab and **Defines** tab are populated with the information extracted from the command line.

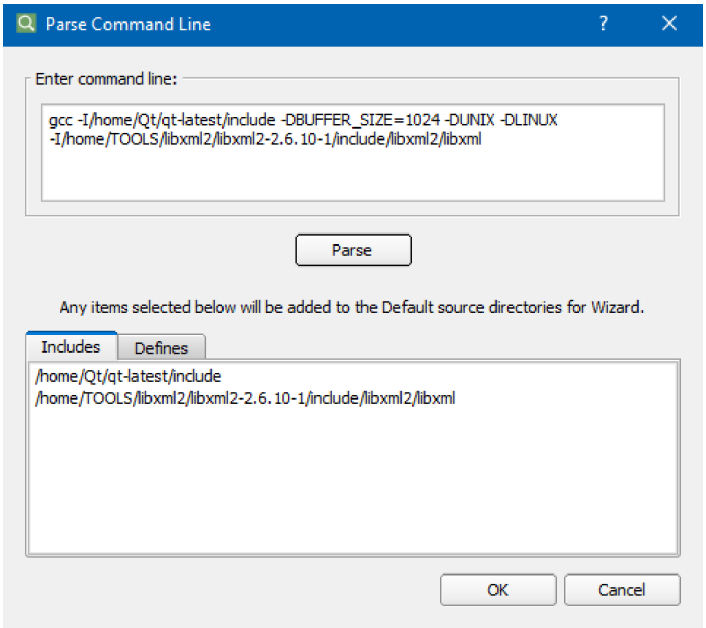
For example, if you are using the gcc compiler, and your normal compile command is:

```
gcc
-I/home/Qt/qt-latest/include
-DBUFFER_SIZE=1024
-DUNIX
-DLINUX
-I/home/TOOLS/libxml2/libxml2-2.6.10-1/include/libxml2/libxml
```

Paste it into the command line text edit area, as shown below.



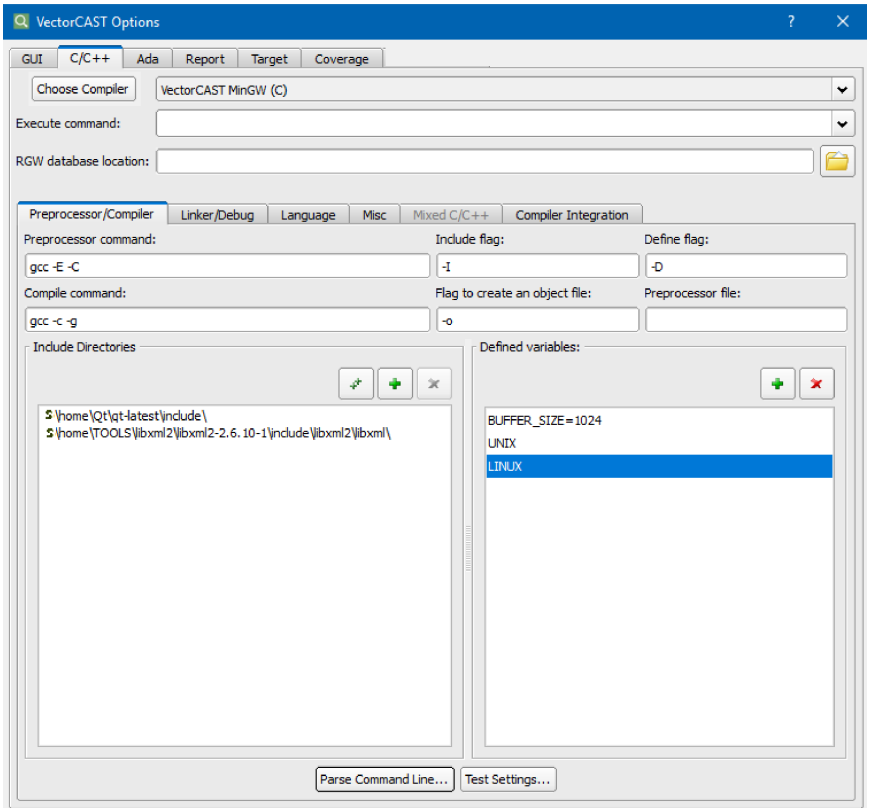
Click the **Parse** button. Using the Include Flag and Define Flag specified on the C/C++ tab of the Options dialog, VectorCAST parses the command line. The parsing process finds any include paths and places them, one per line, in the **Include** tab in the lower part of the dialog, and finds any defined variables and places them in the **Defines** tab. The result of our example command line is shown below, for both the **Includes** tab and the **Defines** tab.



By default, each item is selected. When you click OK, items that are selected on the **Includes** tab are saved as Search directories to the “Default source directories for Wizard” option, on the C/C++ tab. Items that are selected on the **Defines** tab are saved to the “Defined variables” option on the C/C++ tab.

To eliminate an item, unselect it. When you have selected the items you want from the **Includes** tab and the **Defines** tab, click OK.

The Parse Command Line dialog closes, and the Include paths and Defined variables are now present in their respective edit boxes.



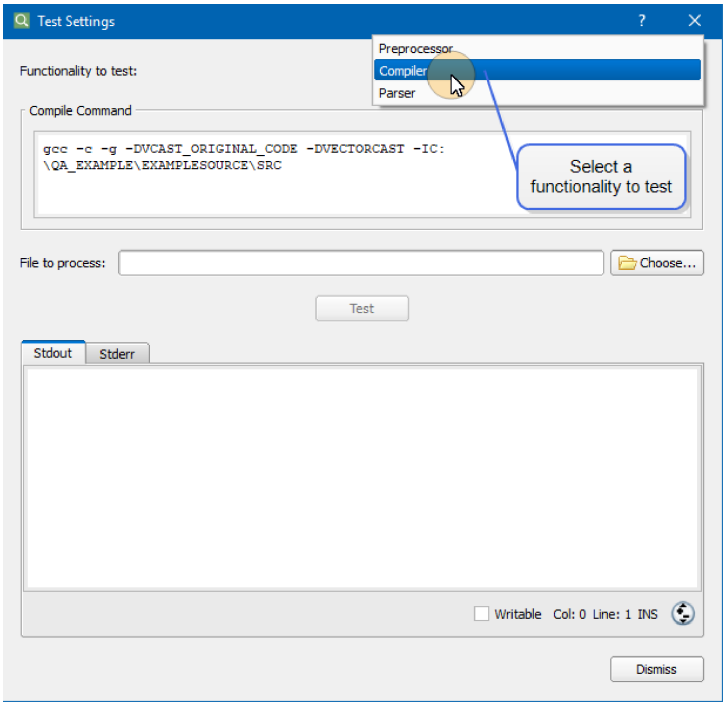
To Test Your Compiler Settings

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Preprocessor/Compiler** tab.

The Test Settings dialog enables you to “try out” the compiler settings on a source file. Once you have chosen a compiler, filled in the Library Include directories and defined variables etc., use the **Test Settings** button to determine if these compiler settings will work when you later attempt to build an environment. Some reasons that the compiler settings may not work are:

- Incorrect compiler chosen
- Compiler not on your PATH
- Missing library include directories, so a file cannot be found
- Missing or incorrect defined variables
- Source directories not in correct order or of wrong type

The Test Settings dialog is used to test the settings for the preprocessor, compiler, and the QuickParser. Choose an action from the Functionality to Test drop-down list.

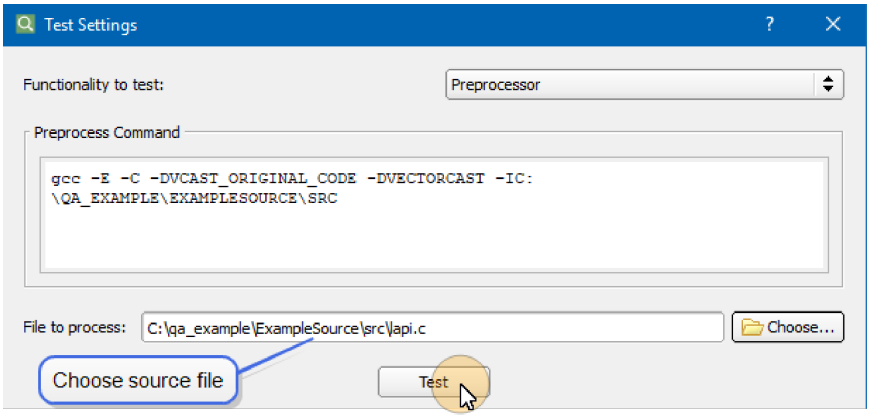


When **Preprocessor** is selected, the preprocess command, with any defines or includes, is displayed in the “Preprocess Command” box. When **Compiler** is selected, the compile command is displayed in the “Compile Command” box. When **Parser** is selected, the Parser Flags are displayed in the “Parser Flags” box.

Tip: The command box is *editable*; if an action fails, you can try out different settings until you get a successful preprocess, compile, or QuickParse. When you done, you must transfer any changes to the C/C++ tab manually.

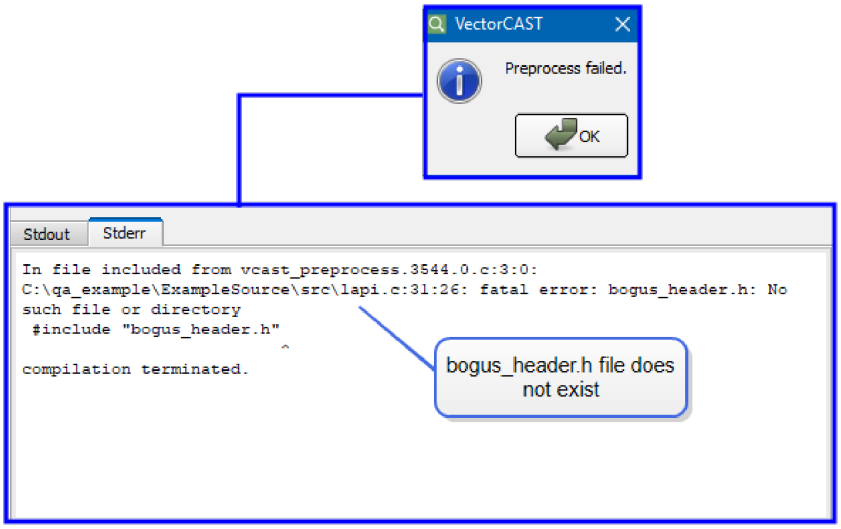
Some defined variables may be added by VectorCAST based on options set in the Builder tab. If you have an environment open, then all paths in the Source directories are listed here as -I<search dir>. When Parser is selected, the Parser flags are displayed in the “Parser Flags” box.

Clicking the browse file icon ( ) opens a dialog enabling you to choose to a source file. The path to the file is displayed in the “File to Process:” edit box. Alternatively, you may type a full path to a source code file to preprocess or compile. Spaces are not allowed in the path. You may use a relative path to a file, relative to the current working directory, whose path is visible in the status bar of VectorCAST’s main window.

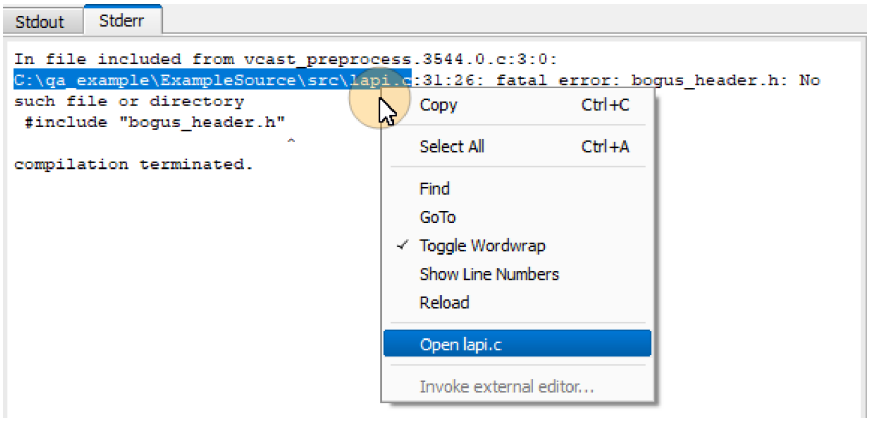


Click the **Test** button. VectorCAST uses the displayed command to preprocess or compile the file. A message dialog appears indicating whether the command was successful or failed.

After acknowledging the status, use the **Stdout** and **Stderr** tabs to diagnose any problems. In the example, below, a non-existing file ("bogus_header.h") was #included in the source file unit. The preprocess failed, since the preprocessor could not find that header file. The error is listed in the **Stderr** tab, as shown below.



You can select the text and save the output to a file or open the source file from the **Stdout** and **Stderr** tabs to a file. Right-click and choose **Select All** or **Open <filename>**.

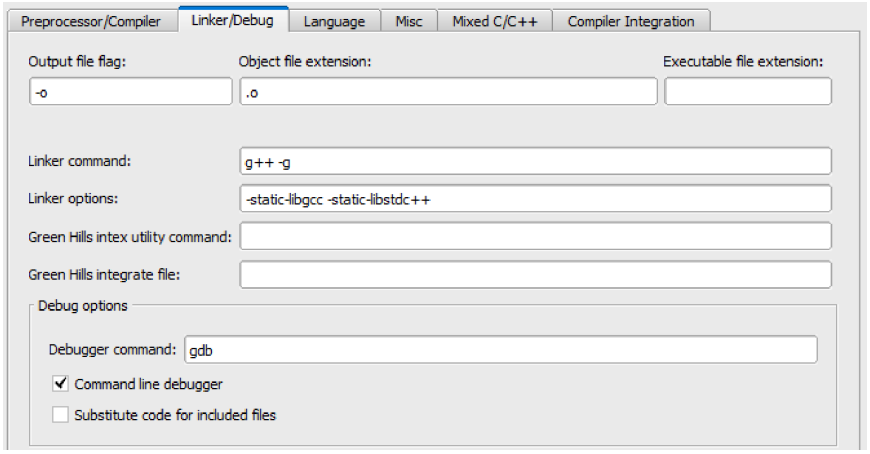


When you are done testing the compiler settings, click the **Dismiss** button. You return to the **Tools => Options dialog, C/C++** tab. If you made any changes to the command being tested, then you must transfer these changes to the C/C++ tab manually.

This functionality is also available in **Help => Diagnostics => Test Compiler Settings**.

Linker/Debug Tab

The **Linker/Debug** tab on the C/C++ tab provides additional options for the compiler's linker and for using the compiler's debugger during test case execution. You can specify the link files for the two test harness executables here (UUT_INTE, and UUT_INST), or a Green Hills integrate file and Green Hills intex utility command.



Output File Flag

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Linker/Debug** tab.

The command line flag to set the name of the linked executable.

```
clicast -lc option C_OUTPUT_FLAG <flag>

Command line option used to specify name of executable created by linker,
such as "-o", "/OUT". Its default value is set by the compiler template.
```

Object File Extension

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Linker/Debug** tab.

The default extension of object files created by the compiler.

```
clicast -lc option C_OBJECT_EXT <extension>

File extension used by C/C++ compiler to specify object files, such as
".o", ".obj". Its default value is set by the compiler template.
```

Executable File Extension

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Linker/Debug** tab.

The default file extension of executables created by VectorCAST.

```
clicast -lc option EXECUTABLE_EXTENSION <extension>

The default file extension of executables created by VectorCAST. Its
default value is set by the compiler template.
```

Linker Command

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Linker/Debug** tab.

The linker command.

```
clicast -lc option C_LINK_CMD <command>

<command> is used to link object files into an executable C/C++ harness.
Its default value is set by the compiler template.
```

Linker Options

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Linker/Debug** tab.

Any miscellaneous options to pass to the linker, such as `-lc` to link in the standard C library.

```
clicast -lc option C_LINK_OPTIONS <options>

Additional link options used when linking C/C++ harness. Its default value
is set by the compiler template.
```

Green Hills intex Utility Command and Green Hills Integrate File

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Linker/Debug** tab.

VectorCAST invokes the Green Hills intex utility with this command immediately after linking the test harness. For example:

```
intex -sp sim800 -os_dir c:\GHS\int4010 -intfile vcast.int
```

where **vcast.int** is a custom integrate file. Your custom integrate file must contain the line **Filename VCAST_FILE** to specify the address space where the VectorCAST test harness runs. The default integrate file **vcast.int**, located in the $VECTORCAST_DIR/DATA directory contains the following text:

```
Kernel
 Filename  DynamicDownload
EndKernel

AddressSpace vcast
Filename      VCAST_FILE
MemoryPoolSize 0x30000
LanguageC++
Task Initial
StackLength0x8000
StartIt    false
EndTask
EndAddressSpace
```

See the option “VCAST_GH_INT_FILE” for information on specifying a custom integrate file.

clicast -lc option VCAST_GH_INTEX_CMD <intex command>

VectorCAST invokes the Green Hills intex utility with <intex command> immediately after linking the test harness. <intex command> does not have a default value.

clicast -lc option VCAST_GH_INT_FILE <intex filename>

This is the custom integrate file passed to the Green Hills 'intex' command. This file should follow the general format of the default file found in the VectorCAST installation directory, which means it should contain a 'Filename' line with the text VCAST_FILE (to be replaced with the VectorCAST executable name) and a 'StartIt' line with the value 'true'.

Debugger Command

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Linker/Debug** tab.

The name of your debugger. This command will be executed when you choose “Execute with Debug.”

```
clicast -lc option C_DEBUG_CMD <command>

Command used to invoke C/C++ debugger. Its default value is set by the
compiler template..
```

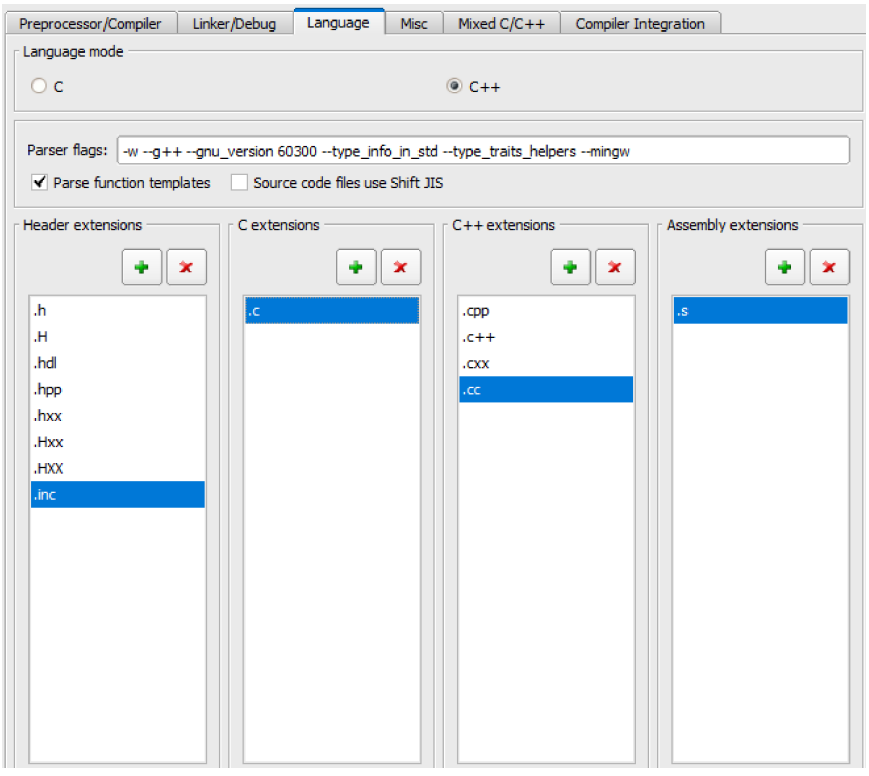
Command Line Debugger

When this option is checked when you execute a test case with debug, VectorCAST brings up a shell window in which the debugger is running.

```
clicast -lc Option VCAST_COMMAND_LINE_DEBUGGER True | False

This option causes VectorCAST to bring up a shell window to run the
debugger. Its default value is false except for the GNU Native and SCORE
compilers.
```

Language Tab

The **Language** tab on the C/C++ tab provides options to specify the language mode, the file extensions for header files, C source files, C++ source files, and assembly files, as well as the compiler's parser flags. The options here are set by the compiler template, but can be modified.



Language Mode

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Language** tab.

The language mode option puts the VectorCAST quick-parser into C or C++ mode, and determines the file extension of the generated test harness source files. If the compiler template has “(C++)” after the name, then the language mode is set to C++. Otherwise, it is set to “C”.

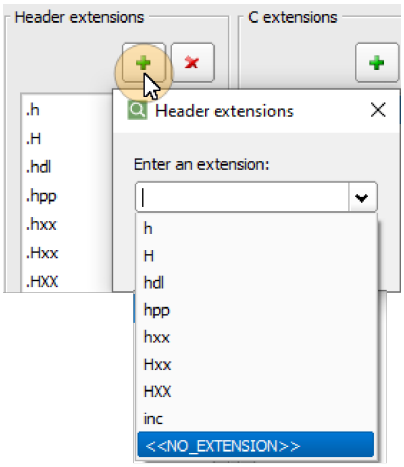
```
clicast -lc option SOURCE_EXTENSION .c | .cpp

Indication of whether the selected compiler template is for C source code,
or C++ source code. Its default value is set by the compiler template.
```

Header Extensions

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Language** tab.

Any file with any of the listed extensions will be treated as a source code header file. Typical header file extensions are supported by default. The default values are: **`.h, .H, .hdl, .hpp, .hxx, .Hxx, .HXX, .inc.`** Additional extensions can be added to the list by clicking the  button, allowing you to specify an atypical header file extension.

VectorCAST provides the ability to specify and instrument header files without a file extension. This is accomplished by selecting the  button and selecting **<<NO_EXTENSION>>** from the extension drop-down menu.



When entering a new extension, VectorCAST adds a “.” character if one is not provided. There must be at least one Header extension in the list.

```
clicast -lc option VCAST_HEADER_FILE_EXTENSIONS <extension>

List of file extensions indicating C/C++ header files. Typical extensions
are supported by default; this option is only needed when header files do
not follow normal coding conventions. Its default values are: .h, .H, .hdl,
.hpp, .hxx, .Hxx, .HXX, .inc. To include headers with no extensions, the
value <<NO_EXTENSION>> may be used.
```

C Extensions

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Language** tab.

Any file with any of the listed extensions will be treated as a source code file. Additional extensions can be added to the list by clicking the  button, allowing you to indicate that files with a given extension are to be treated as source code files.

When entering a new extension, VectorCAST adds a "." character if one is not provided.

If the Language Mode is C, then there must be at least one C extension in the list. Additionally, the C++ extensions list is dimmed.

```
clicast -lc Option VCAST_C_FILE_EXTENSIONS <c file extensions>

Use this option to specify the C file extensions. Its default value is: c.
```

C++ Extensions

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Language** tab.

Any file with any of the listed extensions will be treated as a source code file. Additional extensions can be added to the list by clicking the  button, allowing you to indicate that files with a given extension are to be treated as source code files.

When entering a new extension, VectorCAST adds a "." character if one is not provided.

If the Language Mode is C++, then there must be at least one C++ extension in the list. The C file extensions list is enabled in order to identify C source files in a mixed C and C++ environment.

```
clicast -lc Option VCAST_CPP_FILE_EXTENSIONS <file extensions>

Use this option to specify the C++ file extensions. Its default values are:
cpp CPP c++ C++ C cxx CXX cc CC. (On Windows, the default values are: cpp
c++ cxx cc.)
```

Assembly Extensions

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Language** tab.

List of file extensions indicating assembly code. These extensions are used to determine whether any startup files should be assembled. If a startup file has a file extension specified on this list, it is assembled before being used to startup the target device. Additional extensions can be added to the list by clicking the  button, allowing you to indicate that files with a given extension are to be treated as assembly code files.

When entering a new extension, VectorCAST adds a "." character if one is not provided.

```
clicast -lc Option VCAST_ASSEMBLY_FILE_EXTENSIONS <file extensions>

Use this option to specify the file extensions for assembly files,
particularly paths to assembly files specified by VCAST_STARTUP_FILE. Any
startup file that is detected to be an assembly file is assembled before
being used to start up the target device. Its default values are set by the
compiler, typically s or asm.
```

Add / Remove / Default Extensions

A source file is processed based solely on the source file extension – if a source file extension does not match any extension in the lists, the file will not be processed. The **Add** and **Remove** buttons enable the user to manipulate the lists. The **Default** buttons reset the lists to the standard language file extension as shown above.

If your compiler uses a different extension than the default extensions provided, additional extensions can be added using the “Add” button. When adding source files recursively, any file matching one of these extensions is recognized as a source file.

```
clicast -e <env> cover options ADD_C_Extension <extension>

Add a non-default filename extension to the list of recognized source file
extensions. Its default values are *.c
```

```
clicast -e <env> cover options ADD_CPP_Extension <extension>

Add a non-default filename extension to the list of recognized source file
extensions. Its default values are *.cpp *.cxx
```

Parser Flags

Choose **Tools => Options**, and click the **C/C++ tab**. Then click the **Language tab**.

Flags to pass to the EDG parser. This option should only be modified under direction of VectorCAST Technical Support.

```
clicast -lc Option C_EDG_FLAGS <parser flags>

This option is set by the compiler template.
```

Parse Function Templates

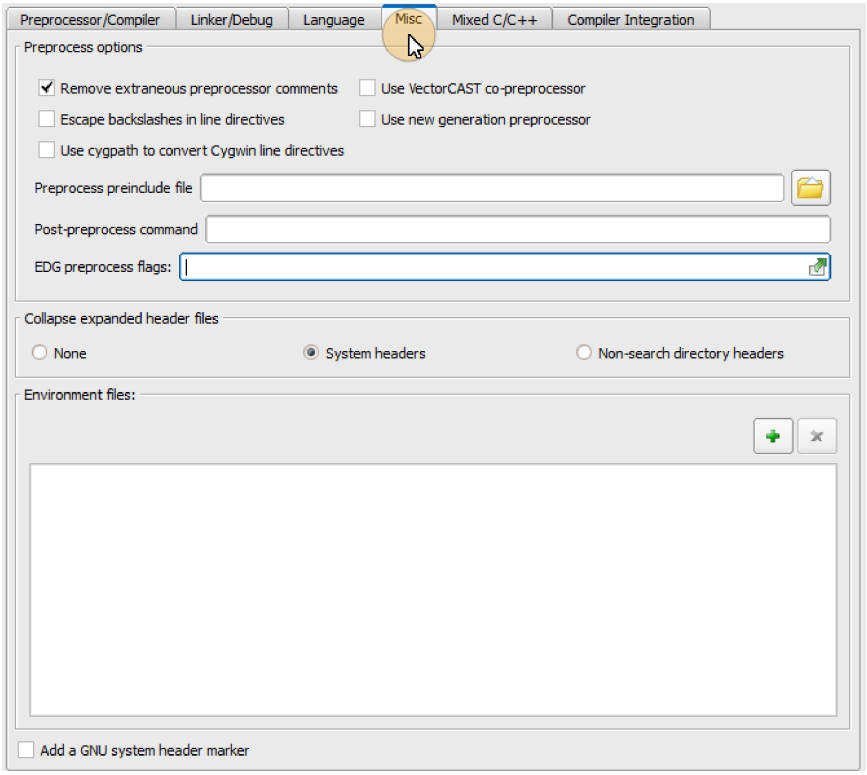
Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Language** tab.

This option processes all function templates during parsing. Set this option to allow coverage instrumentation for template-based functions. Enabling this option will cause uninstantiated templates to be processed and possibly treated as instantiated. Such processing may reveal source code errors not reported by a compiler, in which case this option should be disabled or the source code should be modified.

```
clicast -lc Option VCAST_PARSE_FUNCTION_TEMPLATES True | False

Process all function templates during processing.
```

Misc Tab

The Misc tab on the **C/C++** tab provides additional options for the compiler and preprocessor. Each option here is set by the compiler template.



Remove Extraneous Preprocessor Comments

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Misc** tab.

This option removes the extraneous preprocessor comments that some compilers (preprocessors) put at the beginning and/or end of a preprocessed file. For example, GCC 3.2 puts the comment **`# 1 "<built-in>"`** at the beginning of each preprocessed file. Turning this option on strips these extraneous comments. By default, this option is enabled for all compilers.

```
clicast -lc option VCAST_REMOVE_PREPROCESSOR_COMMENTS True | False

This option is set by the compiler template. Its default value is true.
```

Use VectorCAST Preprocessor

Choose **Tools => Options**, and click the **C/C++ tab**. Then click the **Misc tab**.

This option is needed for compilers whose preprocessor does not adequately mark the start and end of header files in the translation unit. VectorCAST needs reliable line directives to correctly determine dependency information. Note that this option does not replace the use of an external preprocessor. Additional processing is performed both before and after the normal preprocess to produce a translation unit with line directives that accurately mark the start and end of each file section.

```
clicast -lc option VCAST_USE_VCPP true | false

This option is set by the compiler template. Its default value is false
except for some target compilers.
```

Escape Backslashes in Line Directives

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Misc** tab.

Enable this option if your compiler's preprocessor does not escape backslashes in line directives in preprocessed files.

```
clicast -lc Option VCAST_ESCAPE_LINE_DIRECTIVES True | False

This option is set by the compiler template. Its default value is false,
except for some targets such as NEC, Keil, and TriMedia.
```

Use New Generation Preprocessor

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Misc** tab.

The "new generation" preprocessor replaces the VectorCAST preprocessor. This option is needed for compilers whose preprocessor does not retain comments or removes whitespace in the translation unit.

```
clicast -lc Option VCAST_USE_EDG_PREPROCESSOR True | False

This option is set by the compiler template. Its default value is false
except for some target compilers.
```

Preprocess Preinclude File

Choose **Tools => Options**, and click the **C/C++ tab**. Then click the **Misc tab**.

This option allows you to specify a file that will be #included to source files during preprocessing.

```
clicast -lc Option VCAST_PREPROCESS_PREINCLUDE <file to #include to all
source files prior to preprocessing>

The default value is none, with the exception of Code Composer Studio C6xxx
compilers.
```

Post-Preprocess Command

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Misc** tab.

This option allows you to specify a command to be executed after preprocessing a file. VectorCAST will pass 3 arguments to this command: the original file path, the preprocessor output file path, and the path to an output file that does not yet exist. If the external command creates the new output file, it will be used in place of the original preprocessor output, such as for parsing. It will also become the basis for any instrumentation, such as for code coverage. This option can be used to specify a custom external script which transforms non-standard code constructs into code suitable for the VectorCAST parser. When possible, modify original source files directly rather than use this option.

```
clicast -lc Option VCAST_POST_PREPROCESS_COMMAND <command>

Command to be executed after preprocessing a file.
```

Collapse Expanded Header Files

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Misc** tab.

During environment build, whenever VectorCAST modifies a unit for stub-by-function, Visual Studio whitebox, or code coverage, it may first preprocess the unit to expand macros and header files. This option tells VectorCAST which expanded header files should be collapsed (replaced with the original **`#includes`**) before compiling.

The default setting for this option is set by the compiler template.

Choose **None** if your code contains macros defined in seach directory files, but which affect compilation of code in non-search directory headers. Setting the option to **None** avoids collapsing any files. **None** is selected for Microsoft compilers.

Choose either **System Headers** or **Non-search directory headers** if the headers contain code that cannot be compiled unless physically located in its original file location. These settings are selected by most compiler templates and collapse files not in Library Include or Type-Handled directories.

If VectorCAST finds a setting for VCAST_COLLAPSE_STD_HEADERS that is 0 or FALSE when reading regression scripts from previous versions, this value is replaced with the default for the compiler.

Note that changing the value for VCAST_COLLAPSE_STD_HEADERS could also impact test harness compilation.



Note: Contact VectorCAST Technical Support before changing the value of this option.

```
clicast -lc option VCAST_COLLAPSE_STD_HEADERS
COLLAPSE_NONE |
COLLAPSE_SYSTEM_HEADERS |
COLLAPSE_NON_SEARCH_HEADERS

This option is set by the compiler template.
```

Environment Files

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Misc** tab. This option specifies the files that need to be copied into the environment directory during environment build. The default value is set by the compiler template.

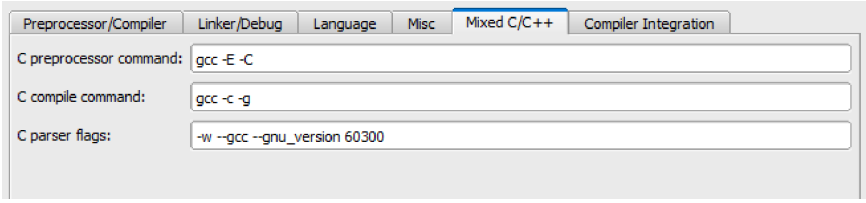
To add a path, click the Add Path button . Browse to the location of the environment file, and click **OK**. To modify a path, double-click it to make it editable. You can include environment variables in the format $(*ENV_VAR*) by editing the CCAST_.CFG file. To delete a path, select it and click the Remove Path button .

```
clicast -lc option VCAST_ENVIRONMENT_FILES <path> [, <path> ... ]

<path> is the full path to a file that needs to be copied into the
environment directory during environment build. Multiple paths are
separated by a comma. Its default value is set by the compiler template.
```

Mixed C/C++ Tab

The **Mixed C/C++** tab on the **C/C++** tab provides settings for using C and C++ source files in the same environment. Start by choosing a C++ compiler template. The settings on the **Preprocessor/Compiler** tab are used when compiling and linking C++ source files in the mixed environment.

The settings on the **Mixed C/C++** tab are used when compiling and linking C source files in a C++ environment. Each option here is set by the C++ compiler template.



C Preprocessor Command

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Mixed C/C++** tab. If it is not enabled, change the compiler to a C++ version, one with “(C++)” at the end of the name.

The “C preprocessor command” option specifies the command to be used to preprocess C source files in a C++ environment.

```
clicast -lc Option C_ALT_PREPROCESS_CMD <preprocessor command>

Command used to preprocess C files in C++ environments. Its default value
is set by the C++ compiler template.
```

C Compile Command

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Mixed C/C++** tab. If it is not enabled, change the compiler to a C++ version, one with "(C++)" at the end of the name.

The "C compile command" option specifies the command to be used to compile C source files in a C++ environment.

```
clicast -lc options C_ALT_COMPILE_CMD <compile command>

Command used to compile C files in C++ environments. Its default value is
set by the C++ compiler template.
```

C Parser Flags

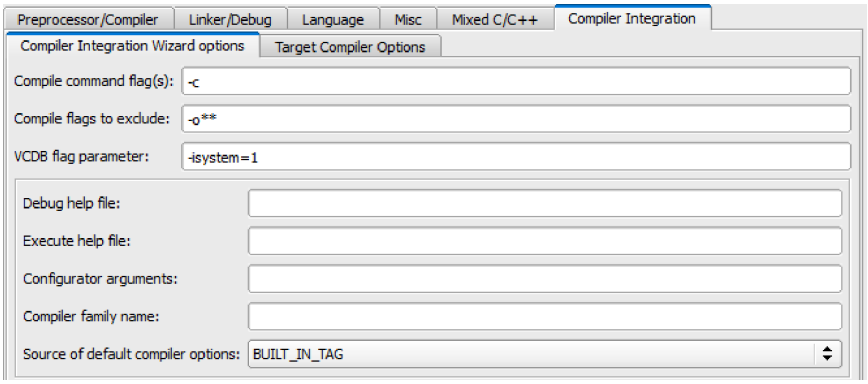
Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Mixed C/C++** tab. If it is not enabled, change the compiler to a C++ version, one with "(C++)" at the end of the name.

The "C parser flags" option specifies the flags to pass to the QuickParser when parsing C source files in a C++ environment.

```
clicast -lc options C_ALT_EDG_FLAGS <parser flags>

Flags to pass to the EDG parser when parsing a C file in a C++ environment.
Its default value is set by the C++ compiler template.
```

Compiler Integration Tab

The options on the **Compiler Integration** tab on the **C/C++** tab provide default commands for the Compiler Integration Wizard and also provide some startup options for target compilers. Each option here is set by the compiler template, but can be fine-tuned as desired.



Compile Command Flag(s)

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Compiler Integration** tab and the **Compiler Integration Wizard options** sub-tab.

The compile flag (such as **-c**) is used to detect a compilation command when parsing build output in the Compiler Integration wizard.

```
clicast -lc option C_COMPILE_CMD_FLAG <flag>

<flags> is the text used to recognize a compilation command when parsing
build output. Its default value is set by the compiler template.
```

Compile Flags to Exclude

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Compiler Integration** tab and the **Compiler Integration Wizard options** sub-tab.

Arguments matching these flags will not be captured by the Compiler Integration Wizard. For example, **/OUT:** . If a specified flag ends with **, then the ** is removed and the flag is considered to take an argument which will also not be captured.

```
clicast -lc option C_COMPILE_EXCLUDE_FLAGS <flag [<flag>...]>

<flag> is the text used to identify a compilation command flag that you do
not want to have captured when parsing build output. Its default value is
set by the compiler template.
```

Debug Help File

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Compiler Integration** tab and the **Compiler Integration Wizard options** sub-tab.

Instructions for debugging the test harness. Click in the field to enter edit mode and enter the name or the path to the help file. The information is stored in the CCAST_.CFG file.

```
clicast -lc option C_DEBUG_HELP_FILE <file name>

Instructions for debugging the test harness.
```

Execute Help File

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Compiler Integration** tab and the **Compiler Integration Wizard options** sub-tab.

Instructions for executing the test harness. Click in the field to enter edit mode and enter the name or the path to the help file. The information is stored in the CCAST_.CFG file.

```
clicast -lc option C_EXEC_HELP_FILE <file name>

Instructions for debugging the test harness.
```

Configurator Arguments

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Compiler Integration** tab and the **Compiler Integration Wizard options** sub-tab.

Enter compiler arguments for python configurator. Click in the field to enter edit mode and enter the arguments. The information is stored in the CCAST_.CFG file.

```
clicast -lc option C_COMPILER_PY_ARGS Compiler Python Args

Compiler arguments for python configurator.
```

Compiler Family Name

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Compiler Integration** tab and the **Compiler Integration Wizard options** sub-tab.

Indicates the compiler family being used. When using the VectorCAST compiler configuration utility, this field is automatically generated. The information is stored in the CCAST_.CFG file.

```
clicast -lc option C_COMPILER_FAMILY_NAME <compiler family name>

Indicates the compiler family being used.
```

Source of Default Compiler Options

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Compiler Integration** tab and the **Compiler Integration Wizard options** sub-tab.

Source of default compiler options. Use the pull-down menu to select the option. Options are BUILT_IN_TAG, PY_CONFIGURATOR, and CONFIG_FILE_62 and CONFIG_FILE_63. The information is stored in the CCAST_.CFG file.

```
clicast -lc option C_COMPILER_CFG_SOURCE Source of default compiler options

Source of default compiler options.
```

Assembler Command

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Compiler Integration** tab and the **Target Compiler Options** sub tab.

The command and options to call the assembler with the startup file.

```
clicast -lc option ASSEMBLER_CMD <command>

<command> is the command to call the assembler. Its default value is set by
the compiler template.
```

Precompile Command

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Compiler Integration** tab and the **Target Compiler Options** sub tab.

The command called before compiling the C/C++ test harness files. This command is only used if your compiler has a two-stage compilation process. After the precompile command is run, a file with the pre-compile extension is produced, and then the compile command is run on that file.

```
clicast -lc option PRECOMPILE_CMD <command>

<command> is called before compiling the C/C++ test harness files. Its
default value is set by the compiler template.
```

Precompile Extension

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Compiler Integration** tab and the **Target Compiler Options** sub tab.

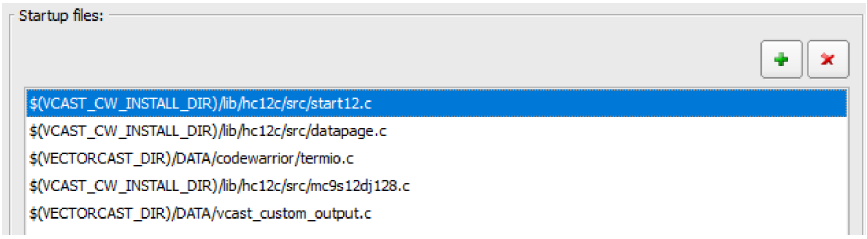
Extension of files resulting from the precompile command.

```
clicast -lc option PRECOMPILE_EXT <command>

The files resulting from calling the precompile command have a file
extension <ext>. Its default value is set by the compiler template.
```

Startup File

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Compiler Integration** tab and the **Target Compiler Options** sub tab.

This option specifies the file(s) containing startup code for your target. The default value is set by the compiler template. For some compilers, this option's value includes several files.



To add a path, click the **Add Path** button . Browse to the location of the startup file, and click **OK**. To modify a path, double-click it to make it editable. You can include environment variables in the format $(*ENV_VAR*) by editing the CCAST_.CFG file. To delete a path, select it and click the **Remove Path** button .

clicast -lc option VCAST_STARTUP_FILE <file>

Path to a file to be copied into the unit test environment directory at build time, then compiled and linked into the test harness. Multiple such paths can specified as separate entries in the CCAST_.CFG.

clicast -lc SET_Startup_files [<file1>[,<file2>…]]

This command takes the comma-separated list of files, <file1>,<file2> …, and writes one line with VCAST_STARTUP_FILE option per file to the CCAST_.CFG file. The path specified in <file> may have spaces. If no <file> is specified, this command clears the option.

Deprecated C/C++ Options

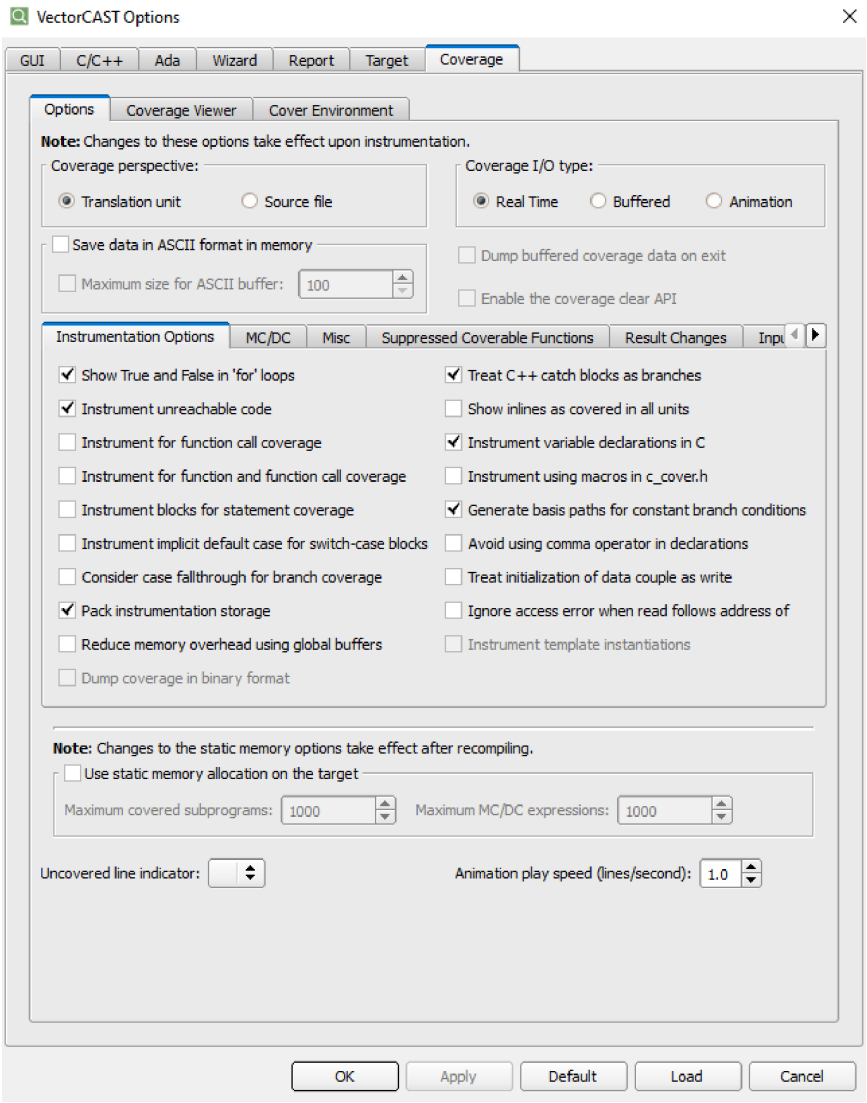
C_INCLUDE_LIST was deprecated in VectorCAST 4.2. Use LIBRARY_INCLUDE_DIR.

Setting Coverage Options

General Coverage Options

Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.

The **Coverage** tab, **Options** sub-tab is used to set coverage options for unit test environments and VectorCAST Cover environments. When used for Cover environments, these options are frequently stored in the CCAST_.CFG file. Some options require reinstrumentation to take effect; others require recompiling the test harness.



Coverage Perspective

Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.

This option allows the user to view code coverage from either the Translation unit perspective or the Source file perspective. The default is Translation unit perspective.

- **Translation unit**: Default. Selecting this option computes and displays coverage based on the translation unit, as a result of preprocessing. Changing this option does not require reinstrumentation.
- **Source file:** Selecting this option computes and displays coverage based on the Source file. Base directories are required to enter Source file perspective mode. If this option is disabled, choose **Environment** => **Update Environment** and then click **Update** to configure the Cover environment for Base directories. Changing this option does not require reinstrumentation.

```
clicast -lc VCAST_COVERAGE_SOURCE_FILE_PERSPECTIVE True | False

When True, this option directs VectorCAST to compute and display coverage
based on the source file. When False, coverage is computed and displayed
based on the translation unit, which is a result of preprocessing. The
default is False.
```

Coverage I/O Type

Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.

Three coverage I/O types are supported: Real-time, Buffered, and Animation. Changes to the Coverage I/O type take effect upon coverage instrumentation.

- **Real-time:** Default. Coverage data is gathered as the test runs, and output to the results file immediately. This I/O type is optimized.
- **Buffered:** Coverage data is stored and then output when the test finishes executing. This method is useful is your target platform is slow. This I/O type is optimized. Buffered I/O gives better performance, but requires more memory than Real-time coverage I/O. The buffered I/O type requires that you add a call to flush the stored coverage data to the results file. At the end of your source code, call VCAST_DUMP_COVERAGE_DATA() or you can set the option 'Dump buffered coverage data on exit.' If using Buffered I/O with Coupling, see "Run Test Cases and Import Trace Data" on page 293.
- **Animation:** This I/O type is not optimized. Instead, coverage data is gathered in the order encountered, in preparation to animate the test result's coverage in the Coverage Viewer. The animated coverage results reveal the flow of control.

After changing this option, you must reinitialize coverage.

```
clicast -lc option COVERAGE_IO_TYPE Real_Time | Buffered | Animation

Type of coverage I/O that the instrumented harness will perform. The
default value is Real_Time.
```

Save Data in ASCII Format in Memory

Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.

The option "Save data in ASCII format in memory" enables VectorCAST to store coverage data in an in-memory data array (inside the instrumented executable), instead of writing the data to disk. When this option is set, VectorCAST calculates the size of the array needed, which depends on the type of coverage and the complexity of any MC/DC expressions that are being tested. The size of the in-memory array is configurable.

This option can be used with any Coverage I/O type, but the most benefit occurs when using buffered Coverage I/O. By default, VectorCAST stores the coverage data in a set of data structures (not in ASCII format) and then writes the data to disk when the application exits. With this option on, the data are stored in ASCII (readable) format and then written to the in-memory data array by a call to VCAST_DUMP_COVERAGE_DATA() in the source code of a Cover environment. In addition, you will need a method to read the in-memory buffer after the instrumented application has exited. (You can do this by printing the buffer's data to the TESTINSS.DAT file.) This option is written to the

CCAST_.CFG file.

It is not recommended to use this option when “Dump buffered coverage data on exit” is on.

After changing this option, you must reinitialize coverage.

When opening a VectorCAST environment prior to version 4.1j, a message appears instructing you to re-instrument all source before initializing coverage.

```
clicast -lc option VCAST_USE_BUFFERED_ASCII_DATA true | false

This option enables VectorCAST to store coverage data in an in-memory data
array in ASCII format, instead of writing the data to disk. Its default
value is FALSE.
```

Maximum Size of the ASCII Memory Buffer

Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.

If "Save data in ASCII format in memory" is True, VectorCAST must allocate a specific amount of space to store coverage data in. The default value is calculated during coverage processing, and usually is large enough to contain the data. However, the default value of the buffer may be too small if your code contains large MC/DC expressions and you have tests that cover a majority of the sub-expressions. In this case, the buffer may overflow and you should increase this option. See the file INVALID_COVERAGE_LINES.LOG, in the environment directory, for more information.

To find the default size of the memory buffer, look in the file **`vcast_c_options.h`**, located in the environment directory after instrumenting. The information is similar to:

```
#define VCAST_USE_BUFFERED_ASCII_DATA 1
#define VCAST_MAX_CAPTURED_ASCII_DATA 662
```

To increase the size of the buffer, set the new value for this option to some number greater than 662 (in this example).

After changing this option, you must reinitialize coverage.

```
clicast -lc option VCAST_MAX_CAPTURED_ASCII_DATA <bytes>

The number <bytes> tells VectorCAST how much memory should be set aside for
buffered ASCII data.  This option should be modified only when using MC/DC
or DO-178B Level A coverage types.
```

Dump Buffered Coverage Data on Exit

Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.

This option is enabled only when the Coverage I/O Type is Buffered. Setting this option forces the coverage data to be dumped to the TESTINSS.DAT file when the test harness or instrumented application terminates (it essentially calls vcast_dump_coverage_data() for you). Because the mechanism used is the system atexit() callback, destructor coverage data for global objects is included in the dump. This option does not cause Coupling data to be dumped. See "Run Test Cases

and Import Trace Data" on page 293.

After changing this option, you must reinitialize coverage.

```
clicast -lc option VCAST_DUMP_COVERAGE_AT_EXIT TRUE | FALSE.

Write buffered coverage data when the test harness or instrumented
application terminates. Uses the system atexit() callback. The default
value is FALSE.
```

Enable the Coverage Clear API

Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.

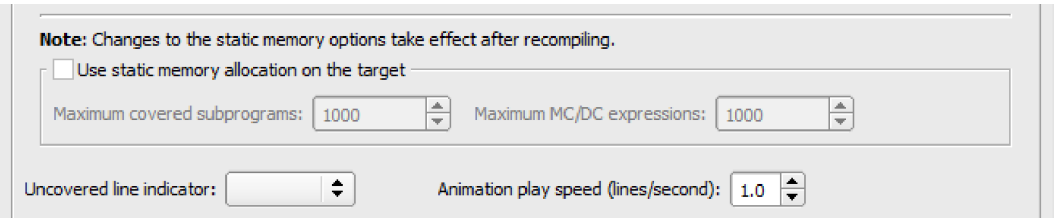
This option is enabled only when the Coverage I/O Type is Buffered. Setting this enables the vcast_clear_coverage() function in the harness or instrumented application. You can call this function from your source code while the instrumented application is running to clear the currently collected coverage data from memory. This function can be combined with vcast_dump_coverage_data() in such a way that you can dump the coverage data from memory to the TESTINSS.DAT file at any time during application execution, then clear the coverage data, and repeat the process until you've collected all the necessary coverage data.

After changing this option, you must reinitialize coverage.

```
clicast -lc option VCAST_ENABLE_DATA_CLEAR_API TRUE | FALSE

When set, enables the vcast_clear_coverage_data() function. The default
value is FALSE.
```

Use Static Memory Allocation on the Target

Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.



When the “Use Static Memory Allocation on the Target” option is checked, VectorCAST uses a static memory model when allocating memory for coverage data. By default, the instrumentation routines allocate memory as needed (dynamic). If you choose the static memory model, you can specify limits for the maximum number of subprograms and MC/DC expressions for which memory should be pre-allocated.

Note: After changing this option, you must recompile.

```
clicast -lc option VCAST_USE_STATIC_MEMORY True | False

This option tells VectorCAST that only static data allocation can be used
on the target platform. This is used when collecting coverage data. If this
is set to True, then use the options VCAST_MAX_MCDC_STATEMENTS and VCAST_
MAX_COVERED_SUBPROGRAMS to configure the static allocation.

Its default value is FALSE.
```

Maximum Covered Subprograms

Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.

To set the “Maximum covered subprograms” option, you must check the box next to “Use static memory allocation” and use the Buffered Coverage I/O type.

The “Maximum covered subprograms” option tells VectorCAST how many unique subprogram calls it should allocate memory for. If a test case or compound test case execution encounters 500 different subprograms calls, and the option is set to 499 or lower, then a text error message is added to the coverage data indicating that the limit was reached. This text error message is used to generate popup messages and CLICAST output messages. This option is written to the CCAST_.CFG file.



Note: After changing this option, you must recompile.

```
clicast option VCAST_MAX_COVERED_SUBPROGRAMS <integer number>

If "Use static memory allocation" is True and Coverage I/O is Buffered,
VectorCAST must allocate a specific amount of space to store coverage data
each time a different subprogram call is encountered. This number tells
VectorCAST how many different subprogram calls for which it should set
aside memory. Its default value is 1000.
```

Maximum MC/DC Expressions

Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.

To set the “Maximum MC/DC expressions” option, you must check the box next to “Use static memory allocation.” You can use any type of Coverage I/O with this option.

The “Maximum MC/DC expressions” option tells VectorCAST how many unique MC/DC expressions it should allocate memory for. If a test case or compound test case execution encounters 500 MC/DC expressions with unique values for its parameters, and the option is set to 499 or lower, then a text error message is added to the coverage data indicating that the limit was reached. This text error message is used to generate popup messages and CLICAST output messages. This option is written to the CCAST_.CFG file.



Note: After changing this option, you must recompile.

```
clicast -lc option VCAST_MAX_MCDC_STATEMENTS <integer number>

If "Use static memory allocation" is True, VectorCAST must allocate a
specific amount of space to store coverage data each time a unique MC/DC
expression is encountered. This number tells VectorCAST how many MC/DC
expressions for which it should set aside memory. Its default value is
1000.
```

Uncovered Line Indicator

Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.

The “Uncovered line indicator” option allows the selection of a character to be used to mark "uncovered" lines in the Coverage Viewer and coverage reports. It allows easy identification of uncovered lines in the VectorCAST reports outside of the VectorCAST GUI.

The feature can be used with Statement, DO-178B Level A, B, and C coverage types.

Valid character choices are: '#', '$', '%', '&', or '+'. The default is "no character".

```
clicast -lc option VCAST_UNCOVERED_LINE_INDICATOR ['#' | '$' | '%' | '&' |
'space']

The uncovered line indicator. The default value is space, that is none.
```

Reinstrumentation is necessary after changing this option. **Coverage => Initialize**, and select **coverage type**

Animation Play Speed

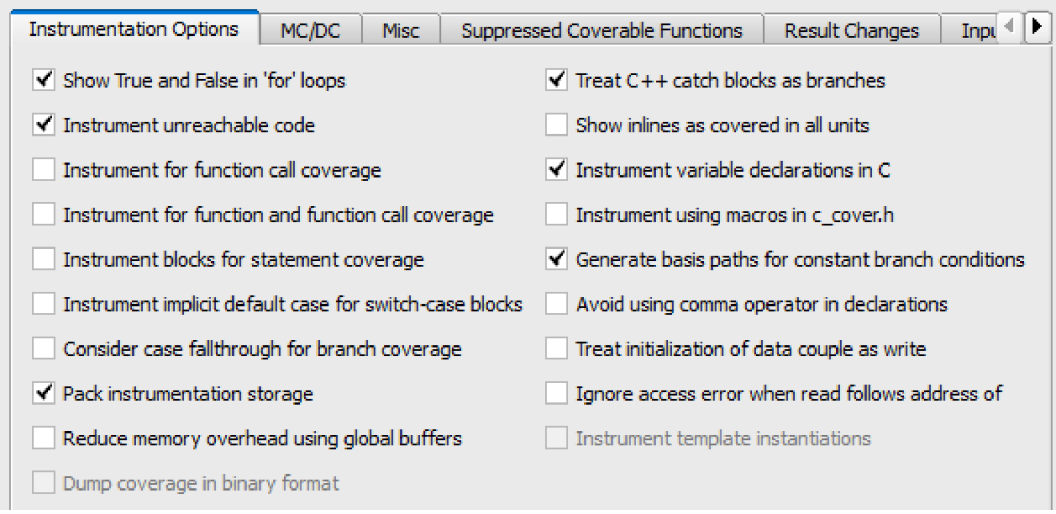
Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.

The “Animation play speed” option enables you to change the speed of coverage animation progress. By default, coverage animation proceeds at the rate of one line per second. To go faster, increase the value. To go slower, decrease the value. The ratio 1.0 represents 1 line / 1 second.

```
clicast option COVERAGE_ANIMATION_PLAY_SPEED_RATIO <floating point number>

The speed ratio for coverage animation. The default value is 1.0, which is
1 line per 1000 msec.
```

Instrumentation Options

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.



Show True and False in 'for' Loops

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

When this option is True, VectorCAST instruments the 'for' loop in Ada to allow both True and False outcomes. Coverage for Ada 'for' statements will show both True if the loop condition has ever evaluated as True, and False if it has ever evaluated as False.

The True condition is met if the loop is entered. The False condition is met if the loop is not entered, or if the loop is exited prematurely due to an exit statement.

When this option is not set, only the True condition is checked.

```
clicast -lc option VCAST_ENHANCED_FOR_LOOP_COVERAGE True | False

The default value is True, meaning Ada "for" loops are instrumented with
both True and False possible outcomes.
```

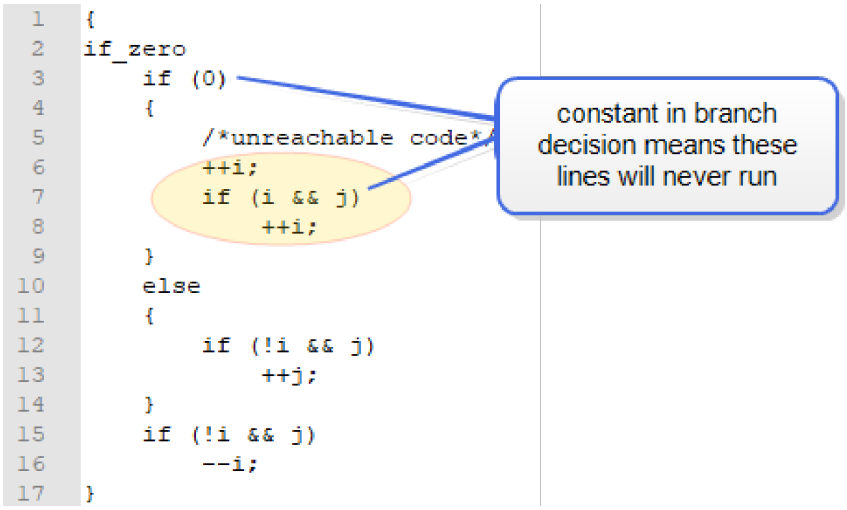
Instrument Unreachable Code

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

When this option is True, the instrumenter will instrument all code that it processes. The default value is True.

When this option is False, the instrumenter will skip instrumenting code that it determines to be unreachable. This is often due to constants in branch decisions, such as an **`if(0)`** or **`while(0)`**, or code following a return statement.

For example, in the source code below on line 3, the expression **`if(0)`** is constant and evaluates to false. That makes the following code unreachable and therefore that code is not instrumented when the code is processed.




```
{
if_zero
    if (0)
    {
        /*unreachable code*/
        ++i;
        if (i && j)
            ++i;
    }
    else
    {
        if (!i && j)
            ++j;
    }
    if (!i && j)
        --i;
}
```

clicast -lc option VCAST_INSTRUMENT_UNREACHABLE_CODE True | False

The default value is True, meaning the instrumenter will instrument all code that it processes. When this option is False, the instrumenter will skip instrumenting code that it determines to be unreachable due to constants in branch decisions, such as if(0) or while(0), or code following a return statement.

Instrument For Function Call Coverage

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

Enabling this option will add additional Function Call coverage in the metrics reports. Note that only the following coverage types with Statement coverage are supported:

- Statement coverage
- Statement+Branch coverage
- Statement+MC/DC coverage

clicast -lc option VCAST_ENABLE_FUNCTION_CALL_COVERAGE True | False

When this option is enabled, additional instrumentation will be added to provide function call coverage support. Only coverage types with statement coverage are currently supported. The default value is False

Note: Coverage must be re-instrumented after enabling this option to view the combined coverage.

For more information on combining Function Call coverage and other coverage types, see the online KnowledgeBase article "Combine Function Call Coverage With Other Coverage Types".

Instrument for Function and Function Call Coverage

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

Enabling this option will add additional Function and Function Call coverage instrumentation. Note that only the following coverage types with Statement coverage are supported:

- Statement coverage
- Statement+Branch coverage
- Statement+MC/DC coverage

```
clicast -lc option VCAST_ENABLE_FUNCTION_AND_CALL_COVERAGE True | False

When this option is enabled, additional instrumentation will be added to
provide both function and function call coverage support. Only coverage
types with statement coverage are currently supported. This option is
incompatible with "Instrument blocks for statement coverage." The default
value is False.
```



Note: Coverage must be re-instrumented after enabling this option to view the combined coverage.

Instrument Blocks for Statement Coverage

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

Enabling this option will change coverage instrumentation to instrument blocks for statement coverage. This reduces the amount of program memory used. This option is applicable for C and C++ source files only.

This feature is intended for users who have limited program memory for their application. The option applies when instrumenting for Statement coverage and any aggregate coverage type that includes Statement (such as DO-178B Level B, which uses both Statement and Branch instrumentation).

When set, this option results in a smaller foot-print for coverage instrumentation because only the last statement of a contiguous block of statements is instrumented when determining Statement coverage. The intermediate statements are considered covered by inference, but data is not collected for them. As a result, the TESTINSS.DAT files and the instrumented files are smaller.

When importing coverage from one VectorCAST to another, both environments must have the same setting for the "Instrument blocks for statement coverage" option when the source files were instrumented. If not, an error message is displayed and the results are not imported.

```
clicast -lc option VCAST_COVER_STATEMENTS_BY_BLOCK True | False

Enabling this option will change coverage instrumentation to instrument
blocks for statement coverage. This reduces the amount of program memory
used.
```

Instrument Implicit Default Case for Switch-Case Blocks

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

This option provides coverage reporting for the implicit "default" case in a switch statement.

By default, this option is off, resulting in no reporting unless the "default" case is explicitly provided in a switch-case block. When set, this option causes an explicit "default" case to be added to the switch statement in the Coverage Viewer and Aggregate coverage reports, will report on coverage of this case.

```
clicast -lc option VCAST_REPORT_ON_IMPLICIT_DEFAULT_CASE TRUE | FALSE

Setting this option will cause the implicit default case not provided at
the end of a switch-case block to be covered when instrumenting branch
coverage. The default value is False.
```

Consider Case Fallthrough for Branch Coverage

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

Setting this option to True causes a case statement reached by fallthrough to be treated as a covered branch.

For example, consider the following source code:

```
switch(foo){
case FIRST:
  i++;
case SECOND:
  j++;
}
```

Branch Coverage:

If the option is true, as in previous versions of VectorCAST, then when foo is FIRST, both case FIRST and case SECOND are covered branches.

If the option is false, then only case FIRST is a covered branch.

Basis Paths:

With the example above, if foo is FIRST, previous versions of VectorCAST would show zero covered basis paths. With the option set to False, basis path coverage now recognizes paths that include the fallthrough, so VectorCAST shows one covered path.

```
clicast -lc option CASE_FALLTHROUGH_BRANCH_COVERAGE True | False

Setting this option will cause a case statement reached by fallthrough to
be treated as a covered branch. The default value is false.
```

Pack Instrumentation Storage

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

When this option is enabled, the instrumentation storage footprint is reduced for statement and branch coverage (C/C++). The statement footprint is reduced by up to a factor of 8 and the branch footprint is reduced by up to a factor of 4. This is accomplished by packing statement and branch points as bits instead of bytes.

When this option is disabled, a byte array is used instead, offering thread-safe instrumentation for Statement and Branch coverage and it may improve test execution performance, due to fewer operations to store the coverage data.

Note that when this option is False, more memory is used during instrumentation or the size of the instrumented executable may be increased, which may impact users executing on a Target.

Enabling or disabling this option requires all source code to be re-instrumented.

```
clicast -lc option VCAST_PACK_INSTRUMENTATION_STORAGE True | False

When this option is enabled, the instrumentation storage footprint is
reduced for statement and branch coverage (C/C++). The statement footprint
is reduced by up to a factor of 8 and the branch footprint is reduced by up
to a factor of 4. This is accomplished by packing statement and branch
points as bits instead of bytes. When this option is disabled, statement
and branch instrumentation (C/C++) is thread safe and may improve execution
performance. The default value is True, meaning to use a bit array.
```

Reduce Memory Overhead Using Global Buffers

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

This option reduces memory overhead for embedded targets with limited memory by using a global buffer instead of a buffer per unit.

- Instrumentation code is reduced by an integer per instrumentation point.
- Instrumentation storage is reduced for each coverage kind via global buffers.
- Buffered Coverage I/O no longer uses linked lists to coverage buffers.
- ASCII buffer storage requirements are reduced.

Enabling or disabling this option requires all source code to be re-instrumented. When this option is enabled, instrumenting a single source file may require you to recompile not just the instrumented source file but all the others that contain higher instrumentation IDs.

```
clicast -lc option VCAST_GLOBAL_BUFFER_OPTIMIZATIONS [True | False]

This option reduces the instrumentation memory overhead in the following
ways:

    > Instrumentation code is reduced by an integer per instrumentation
```

```
point.

> Instrumentation storage is reduced for each coverage kind via global

buffers.

> Buffered Coverage I/O no longer uses linked lists to coverage
buffers.

> ASCII buffer storage requirements are reduced.

Enabling or disabling this option requires all source code to be re-
instrumented. When this option is enabled, instrumenting a single
source file may require you to recompile not just the instrumented
source file but all the others that contain higher instrumentation
IDs. The default value is False.
```

Dump Coverage in Binary Format

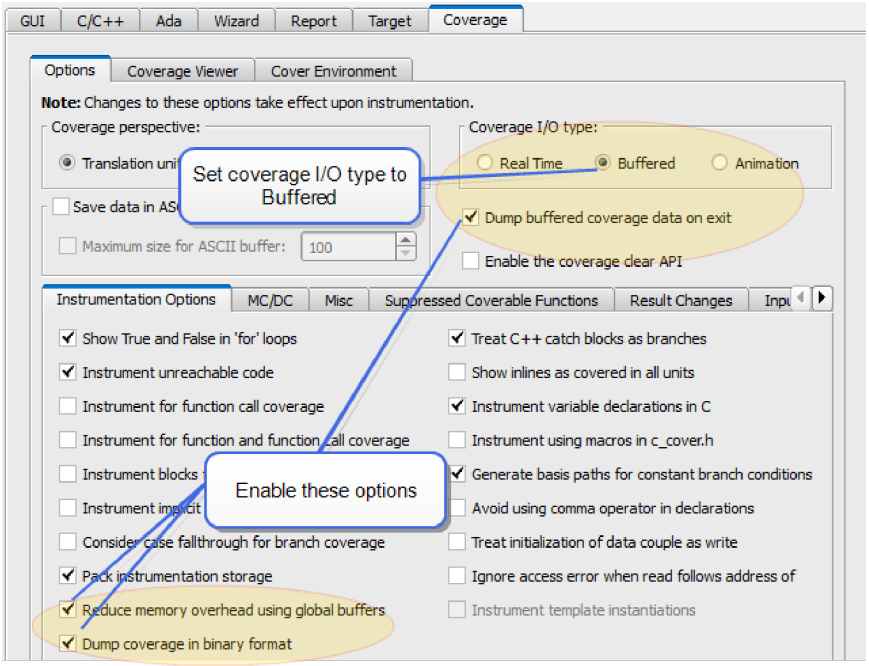
Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

This option decreases the size of the coverage data generated by the instrumented source code by using a binary format.

Coverage TESTINSS.DAT created using binary mode can only be imported into VectorCAST when binary coverage is enabled otherwise, it will return a coverage import error.

To configure binary coverage data in VectorCAST:

- From VectorCAST Options, Coverage Tab, set Coverage I/O type to **Buffered**.
- Enable the option **Dump buffered coverage data on exit**.
- From the Instrumentation Options sub-tab, enable **Reduce memory overhead using global buffers** and **Dump coverage in binary format**.




```
clicast -lc option VCAST_DUMP_BINARY_COVERAGE True | False

When buffered coverage I/O and global buffers are in use, enabling this
option will produce a raw binary dump of the coverage to the "TESTINSS.DAT"
file, eliminating the code and processing overhead of the ASCII data
conversion. In addition, in most cases this file will be much smaller in
size than the ASCII equivalent. The default value is False.
```

Treat C++ Catch Blocks as Branches

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

When set, catch blocks are considered to be branches for coverage purposes.

```
clicast -lc option VCAST_COVER_CATCH_AS_BRANCH [True | False]

Enabling this option will provide code coverage for C/C++ blocks as if they
are branches. The default value is True.
```

Show Inlines as Covered in All Units

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

This option pertains to C/C++ units in Cover and unit test environments. When this options is false, then any function defined in a header file is covered in only one UUT, assuming the option “Provide code coverage in headers” is on, and provided that the function is not inlined by the compiler or called

by multiple UUTs.

Setting this option causes VectorCAST to identify (inline) functions that appear in multiple units and to show the same coverage for each such function in all the units in which it appears. The replication of coverage data across unit occurs dynamically during the processing of the coverage data. The replication will not be explicitly recorded in coverage scripts, but this option can be utilized when importing results in a different environment all long as the destination environment contains the unit for which the original coverage data was recorded and the units in the destination environment were instrumented with the option enabled.

After changing this option, you must reinitialize coverage.

```
clicast -lc option VCAST_SHOW_INLINES_COVERED_IN_ALL_UNITS True | False

Enabling this option will cause inline functions that appear in multiple
units to show the same coverage in each. The default value is FALSE.
```

Instrument Variable Declarations in C

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

When this option is checked, VectorCAST causes coverage instrumentation to be performed for initialized variable declarations in functions in C units. This option can be used with all C compilers, including those that do not permit mixing executable statements with variable declarations. However, setting the option to True causes VectorCAST to instrument only those variable declarations that are initialized; those that are not initialized are left uncoverable. Therefore, if you are using this option, you may see a reduction in the total number of coverable statements.

For C++ units, all declarations are instrumented regardless of the option's setting.

You must re-initialize coverage after turning this option on or off.

This option requires that you specify a preprocessor on **the Tools => Options** dialog, **C/C++** tab.

```
clicast -lc option VCAST_COVERAGE_FOR_DECLARATIONS [True | False]

Enabling this option causes coverage instrumentation to be performed for
initialized variable declarations in functions in C units. (Such
declarations are always instrumented in C++ units.) The default value is
True.
```

Instrument Using Macros in c_cover.h

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

By default, the instrumenter uses the functions defined in the c_cover_io.c file for each instrumentation point in the source file. When this option is set, the instrumenter instead uses the macros defined in the local version of the **`c_cover.h`** file, located in the environment directory. Changing the value of this option takes effect upon instrumentation.

```
clicast -lc option VCAST_COVERAGE_POINTS_AS_MACROS [True | False]

By default, the instrumenter uses the functions defined in the c_cover_io.c
file for each instrumentation point in the source file. When this option is
set, the instrumenter instead uses the macros defined in the local version
of the c_cover.h file, located in the environment directory. Changing the
value of this option takes effect upon instrumentation.) The default value
is True.
```

Generate Basis Paths for Constant Branch Conditions

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

When this option is True, VectorCAST generates basis path tests treating constant **`if, while, do-while`**, and **`for`** conditions, such as "**`if (0)`**", the same as non-constant branches. When the option is False, then constant conditions for those statements do not add to the cyclomatic complexity.

```
clicast -lc option VCAST_BASIS_PATHS_FOR_CONSTANT_BRANCHES True | False

VectorCAST will generate basis path tests treating constant if, while, do-
while, and for conditions, such as "if (0), the same as non-constant
branches. If this option is not selected, then constant conditions for
those statements do not add to the cyclomatic complexity. The default value
is True.
```

Avoid Using Comma Operator in Declarations

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

When VCAST_COVERAGE_FOR_DECLARATIONS is set to true, this option causes VectorCAST to avoid use of the comma operator when instrumenting C variable declarations. This option is necessary for compilers which cannot handle comma operators in initializations.

```
clicast -lc option VCAST_AVOID_COMMA_OPERATOR True | False

When VCAST_COVERAGE_FOR_DECLARATIONS is set to true, this option causes
VectorCAST to avoid use of the comma operator when instrumenting C variable
declarations. This option is necessary for compilers which cannot handle
comma operators in initializations.. The default value is False.
```

Treat Initialization of Data Couple as a Write

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

When this option is enabled, global data initialization expressions will be treated as a write. An initialization expression in a component is enough to create a data couple with other components.

When the option is False:

- Global initialization expressions do not contribute towards establishing a couple.
- Global initialization expressions do not contribute towards coverage.
- A read following an initialization causes the initialization expression to be marked as needing analysis.

When the option is True:

- Global initialization expressions contribute towards establishing a couple.
- Global initialization expressions contribute towards coverage.
- A read following an initialization does not cause the initialization expression to be marked as needing analysis.

```
clicast -lc Option VCAST_TREAT_DC_INITIALIZATIONS_AS_WRITE True | False

When this option is enabled, global data initialization expressions will be
treated as a write. An initialization expression in a component is enough
to create a data couple with other components. Reads following an
initialization will not be flagged as needing analysis. The default value
is False.
```

Ignore Access Error When Read Follows Address of

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Instrumentation Options** sub-tab if it is not selected.

When this option is enabled, VectorCAST ignores data couple access errors for a read following an address of operation.

```
clicast -lc option VCAST_IGNORE_ERROR_WHEN_READ_FOLLOWS_ADDRESS_OF True |
False

When this option is enabled, data couple access errors for reads following
an address of operation will be ignored. The default value is False.
```

Instrument Template Instantiations

When this option is enabled, instrumentation is performed on template instantiations instead of template bodies. To enable this option, the following requirements must be satisfied:

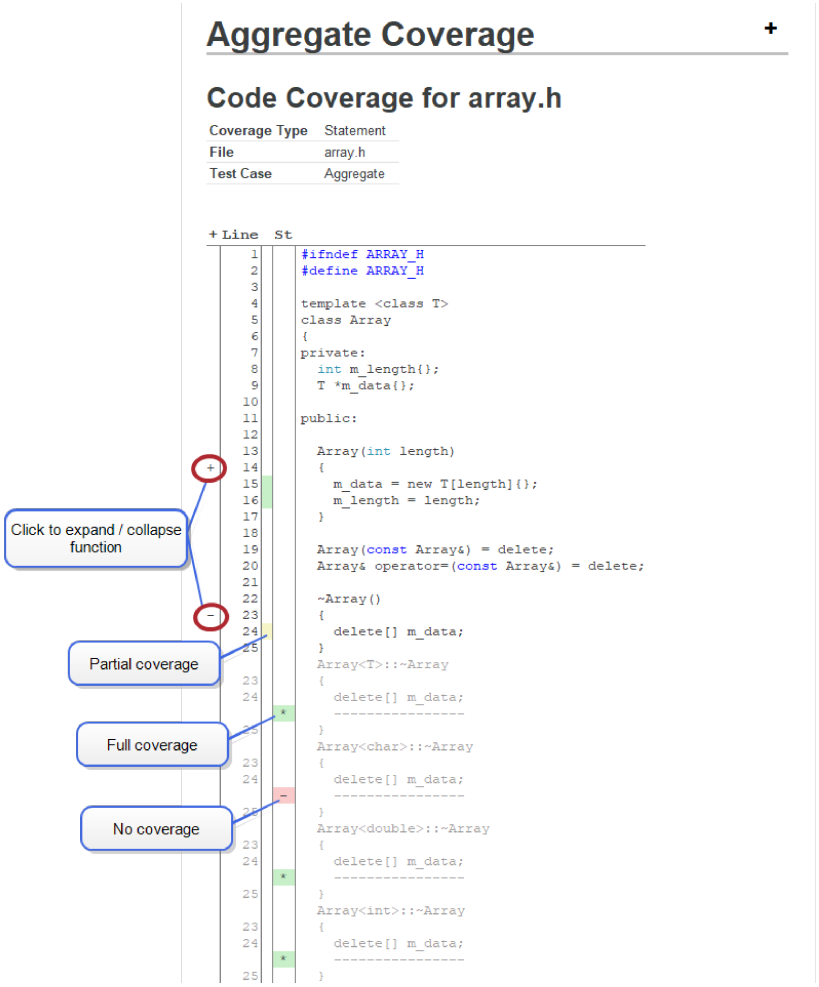
- C/C++ Cover environment
- Source File Perspective is True (VCAST_COVERAGE_SOURCE_FILE_PERSPECTIVE)
- Reduce memory overhead using global buffers is False (VCAST_GLOBAL_BUFFER_OPTIMIZATIONS)

Note that for templates in headers, you must include the header files which define the templates and any source files which instantiate them in the allowlist when configuring your Base Directory.

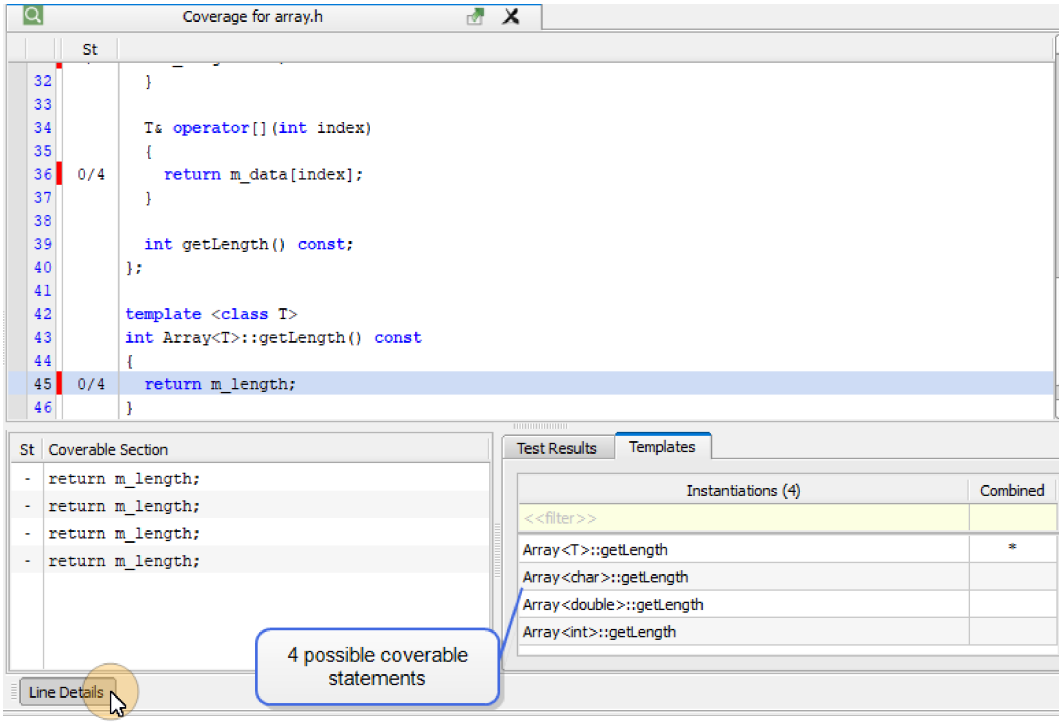
Enabling or disabling this option requires all source code to be reinstrumented.

```
clicast -lc VCAST_COVERAGE_FOR_TEMPLATE_INSTANTIATIONS True | False

When this option is enabled, instrumentation is performed on template
instantiations instead of template bodies. The default value is False.
```

When this option is enabled, the Aggregate Coverage Report uses the aggregate coverage from all instantiations and displays a color strip to indicate the level of coverage (green = covered, yellow = partial coverage, red = no coverage). Coverage can be viewed on the template functions themselves, which are comprised of all instantiations. Coverage can also be viewed on the template instantiations by expanding the template functions and clicking on the **+/-** symbol located to the left of the Line number column.



Enabling this option allows the Coverage viewer to display a Templates tab within the Line Details pane, which lists the template instantiations for the selected line and allows the user to identify which of these instantiations are covered. In the example below, 4 possible coverable statements are shown: one each for types **`char`**, **`double`** and **`int`**, and one for type **`T`**.



Combined Template Instantiations

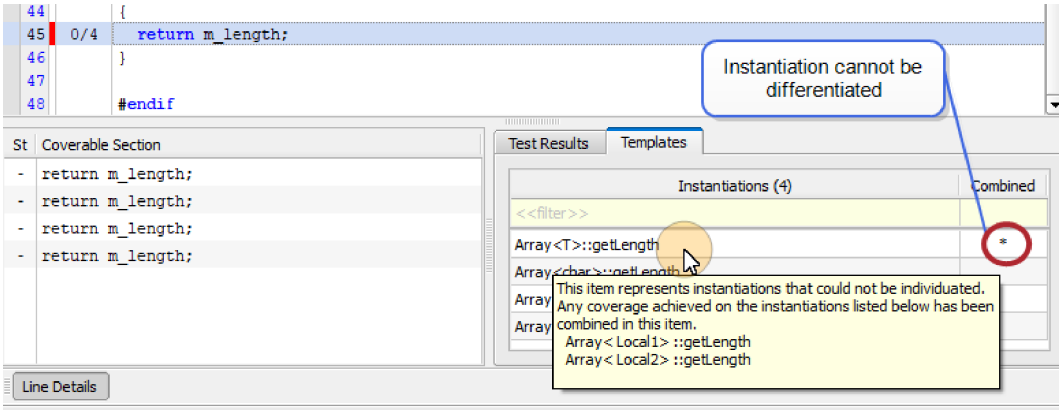
A limitation exists when template functions are instantiated using local types. A local type is a user defined type within function scope. In these cases, the coverage for these instantiations cannot be differentiated from each other. VectorCAST combines these instantiations into a single group which are used to show the merged coverage. The group is labeled using the template function name.

In the example below, coverage shows three functions: **`template_fn(int)`**, **`template_fn(double)`** and **`template_fn(T)`**.

```
1  template <typename T>
2  void template_fn(T value){}
3
4  int main()
5  {
6      template_fn(1);   // <--- instantiation using int
7
8      template_fn(1.0); // <--- instantiation using double
9
10     struct LocalType_1{};
11     LocalType_1 one;
12     template_fn(one); // <--- instantiation using local type
13
14     struct LocalType_2{};
15     LocalType_2 two;
16     template_fn(two); // <--- instantation using local type
17
18     return 0;
19 }
```

Groups representing instantiations from local types for a given template function are identified with an asterisk in the Combined column. Hovering over a combined template instantiation lists each

instantiation comprising the group.



The best way to confirm if you have code containing combined instantiations is to generate either the Metrics Report or the Aggregate Report and verify that the report contains a Combined Template Instantiations table.

The Combined Template Instantiations table lists template instantiations that cannot be tracked individually for code coverage. The table lists instantiations using their parameterized name. The example below shows the results for the instantiations in the **`array.h`** file.

Combined Template Instantiations

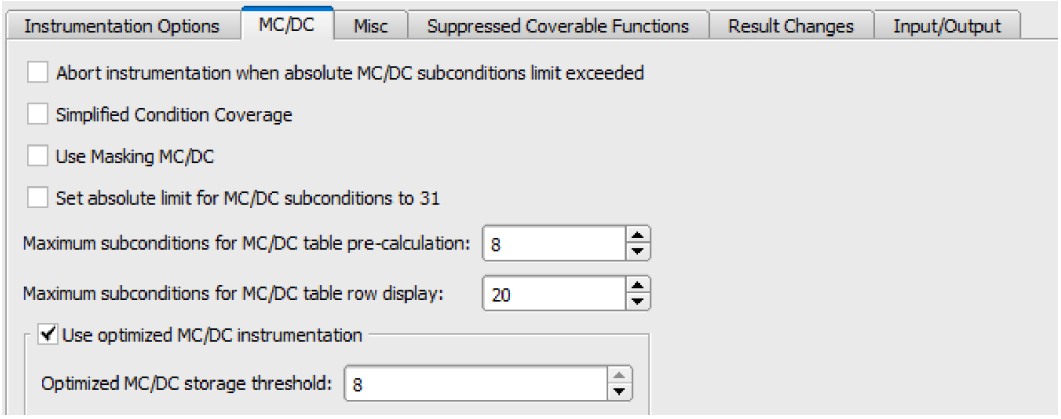
This table lists the template instantiations that could not be tracked individually for code coverage. These instantiations are combined under the prototype name resulting in their coverage being merged.

File	Prototype	Instantiations
array.h	Array<T>::Array(int)	
		Array< Local1> ::Array
		Array< Local2> ::Array
array.h	Array<T>::erase	
		Array< Local1> ::erase
		Array< Local2> ::erase
array.h	Array<T>::getLength	
		Array< Local1> ::getLength
		Array< Local2> ::getLength
array.h	Array<T>::operator[]	
		Array< Local1> ::operator[]
		Array< Local2> ::operator[]
array.h	Array<T>::~Array	
		Array< Local1> ::~Array
		Array< Local2> ::~Array

Note that most cases do not have combined instantiations. The Combined column is only displayed when combined instantiations exist. In those instances without combined instantiations, the table and Combined column are omitted.

MC/DC Options

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **MC/DC** sub-tab if it is not selected.



Abort Instrumentation When Absolute MC/DC Subconditions Limit is Exceeded

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **MC/DC** sub-tab if it is not selected.

When this option is selected, while instrumenting with MC/DC, if a conditional statement contains more conditions than possible to instrument, then VectorCAST aborts instrumenting that file. For C++, the absolute maximum is 52, unless VCAST_HAS_LONGLONG is false or VCAST_UNSIGNED_LONG_MCDC_STORAGE is true, which makes the maximum 31. For Ada, the maximum is 26.

```
clicast -lc option VCAST_MAX_MCDC_ABORT_INST True | False

When this option is selected, while instrumenting with MC/DC, if a
conditional statement contains more conditions than possible to instrument,
then VectorCAST aborts instrumenting that file. For C++, the absolute
maximum is 52, unless VCAST_HAS_LONGLONG is false or VCAST_UNSIGNED_LONG_
MCDC_STORAGE is true, which makes the maximum 31. For Ada, the maximum is
26.
```

Simplified Condition Coverage

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **MC/DC** sub-tab if it is not selected.

This option primarily affects customers in the Medical Device industry. To more closely follow the requirements of IEC-62304, a Coverage option named "Simplified condition coverage" is implemented. When this option is True, entry points, decision outcomes, and all condition values are reported, but equivalence pairs are not reported.

The option automatically takes effect when the environment is instrumented for IEC-62304 (Medical) Class C coverage type.

The option can also be set by selecting **Simplified Condition Coverage** on the MC/DC sub-tab and then instrumenting the environment for MC/DC or Statement+MC/DC coverage type.

When the environment is instrumented with this option True, the user is not permitted to perform the MC/DC Test Case Analysis, as this action requires the MC/DC Equivalence Pairs data to be available.

The default value is FALSE except when in the IEC-62304 (Medical) industry mode, in which case it is TRUE.

```
clicast -lc option VCAST_SIMPLIFIED_CONDITION_COVERAGE True | False

This option suppresses the computation and reporting of Equivalence Pairs
when MC/DC Coverage is active. Since Equivalence Pairs are not computed,
MC/DC tests cannot be generated. This option is primarily used for Medical
Device testing, and is defaulted to TRUE when IEC-62304 (Medical) Class C
Code Coverage is selected. The default is FALSE for all other cases.
```

Use Masking MC/DC

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **MC/DC** sub-tab if it is not selected.

This option performs MC/DC analysis and equivalence pair reporting using MC/DC Masking as defined in the FAA CAST Position Paper-6. This option is of interest to those projects performing DO-178B and DO-178C software certification.

This option requires reinstrumentation after its value has changed.

```
clicast -lc option VCAST_VCAST_MASKING_MCDC True | False

Use masking MC/DC to dertermine pairs for C and C++ files instead of unique
cause MC/DC. The default value is False.
```

Set Absolute Limit for MC/DC Subconditions to 31

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **MC/DC** sub-tab if it is not selected.

For C++, the absolute maximum number of MC/DC sub-conditions supported is 52 by default. Setting this option reduces the maximum number to 31. The config file value for this option is VCAST_UNSIGNED_LONG_MCDC_STORAGE.

```
clicast -lc option VCAST_UNSIGNED_LONG_MCDC_STORAGE True | False

For C++, the absolute maximum number of MC/DC sub-conditions supported is
52 by default. Setting this option reduces the maximum number to 31. The
default value is False.
```

Maximum Subconditions for MC/DC Table Pre-calculation

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **MC/DC** sub-tab if it is not selected.

The “Maximum subconditions for MC/DC table pre-calculation” option enables you to limit the application of MC/DC analysis to conditions that contain a particular number of sub-conditions. The default value for this option is 8. Any statement containing more than 8 conditions will not be analyzed. The maximum value for this option is 26.

This option is written to the Cover environment's .vcp file as MAX_MCDC_CONDITIONS. Its default value is 8.

After changing this option, you must reinitialize coverage.

```
clicast -e <COVER_ENV> Cover Options Max_mcdc_conditions <number>

VectorCAST generates equivalence matrices for up to this many subconditions
in an MC/DC expression. (Example: (A || !B || C) has operands, so it has
three subconditions.) If an expression exceeds this limit, equivalence
pairs are calculated as results are added to the environment, and only
table rows that contribute to a satisfied pair are displayed. MC/DC tests
are not created for expressions exceeding the limit. Its default value is
8.
```

Maximum Subconditions for MC/DC Table Row Display

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **MC/DC** sub-tab if it is not selected.

When VectorCAST generates equivalence matrices for an expression, if the expression has more than this many subconditions, no table information will be displayed. Pair information will still be calculated. The default value for this option is 20. The maximum value for this option is 52.

```
clicast -lc option VCAST_MAX_TABLE_SUBCONDITIONS <integer number>

When VectorCAST generates equivalence matrices for an expression, if the
expression has more than this many subconditions, no table information will
be displayed. Pair information will still be calculated. Its default value
is 20.
```

Use Optimized MC/DC Instrumentation

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **MC/DC** sub-tab if it is not selected.

This option allows smaller conditions to use much less run-time memory. For conditions with a sub-condition count under a configured threshold, a bit-packed array will be used to store run-time data. Conditions exceeding the configured threshold will continue to use the legacy approach of a balanced binary tree to store run-time data. The bit arrays are allocated to allow 100% run-time coverage without the possibility of exhausting the MC/DC statement pool.

This option is enabled by default, but may be disabled to revert to using the previous approach of balanced binary trees for all MC/DC conditions.

Enabling or disabling this option or changing the maximum conditions to use this method requires all source to be re-instrumented.

See "Optimized MC/DC Storage Threshold" on page 211 for more information on defining which storage technique is used for each expression instrurmented.

```
clicast -lc option VCAST_USE_OPTIMIZED_MCDC_INSTRUMENTATION True | False

This option enables optimized MCDC instrumentation for smaller MCDC
conditions. With this enabled, runtime instrumentation data will consume
much less memory. However, the result data generated is not compatible with
the standard instrumentation model. Enabling or disabling this option or
changing the maximum conditions to use this method requires all source to
be re-instrumented. The default value is True.
```

Optimized MC/DC Storage Threshold

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **MC/DC** sub-tab if it is not selected.

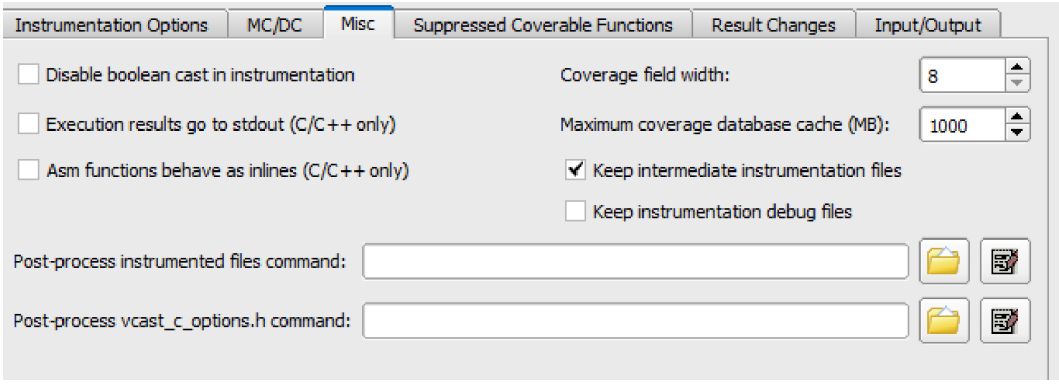
The option "Optimized MC/DC storage threshold" defines which storage technique is used for each expression instrumented. For conditions with a number of sub-conditions fewer than this value, the optimizing bit-array is used to store run-time data. For conditions with a number of sub-conditions greater than this value, the legacy approach of a balanced binary tree is used to store run-time data.

The option may be set in the range of 0 (condition with a single constant) to 8, with 8 being the default.

```
clicast -lc option VCAST_OPTIMIZED_MCDC_STORAGE_THRESHOLD 0..8

The value of this option controls the underlying storage technique for
MC/DC coverage data. For conditions with a number of sub-conditions greater
than this value, VectorCAST uses a self balancing tree for storage. For
conditions with a number of sub-conditions equal to or smaller than this
value, VectorCAST uses a bit array. For most architectures, the "break
even" point for memory usage is 8 sub-conditions, which means for a
condition with 9 sub-conditions, it's possible to use less memory using the
balanced tree approach. The default value is 8.
```

Miscellaneous Options

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Misc** sub-tab if it is not selected.



Disable Boolean Cast in Instrumentation

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Misc** sub-tab if it is not selected.

Set this option to prevent VectorCAST from casting conditional expressions to boolean when instrumenting for Branch or MC/DC coverage.

After changing this option, you must reinitialize coverage.

```
clicast Option VCAST_DISABLE_BOOLEAN_CAST True | False

Select this option to prevent VectorCAST from casting conditional
expressions to boolean when instrumenting for branch or MC/DC coverage. The
default value is False.
```

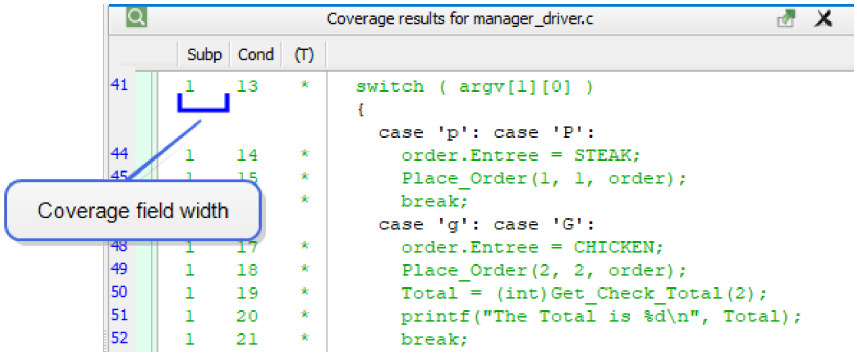
Coverage Field Width

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Misc** sub-tab.

The “Coverage field width” option specifies the width, in characters, of the left column of the Coverage Viewer. Increase this number if you have a large number of subprograms in the unit or a subprogram has a large number of statements or branches. The default value is 8 characters.

Note: This option applies to Cover environments using Ada source.

This option is written to the CCAST_.CFG file.



After changing this option, you must reinitialize coverage.

clicast -lc option VCAST_COVERAGE_FIELD_WIDTH <integer number>

The width (in characters) of the left margin of the coverage viewer for C/C++ files instrumented prior to VectorCAST 2020 and all Ada files. Increase this number if you have a large number of subprograms or a subprogram with a large number of statements or branches. Its default value is 8.

Maximum Coverage Database Cache

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Misc** sub-tab.

This option allows you to adjust the coverage database cache size. Increasing the value may increase performance, but care should be taken not to set the value greater than available memory. Changes to this option take effect when re-opening an environment.

clicast option VCAST_COVERAGE_DATABASE_CACHE_SIZE <integer number>

Increasing the maximum coverage database cache size may increase performance, but take care not to set it to a value greater than the amount of available system memory. For a system with 2GB, a 1000 MB maximum cache is probably sufficient. Changes to this option take effect when reopening an environment.

Asm Functions Behave as Inlines (C/C++ only)

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Misc** sub-tab if it is not selected.

When this option is set to True, VectorCAST tells the compiler to treat asm functions like inline functions so it can skip over them when adding defined functions to the coupling information. The option is True for DIAB and Green Hills compilers, and otherwise False.

```
clicast -lc option ASM_FUNCS_BEHAVE_AS_INLINES True | False

This option should be set to True if the compiler treats asm functions like
inlines. If a definition of the asm function may be in a header #included
into multiple source files in the same executable, then this option should
be set to true. This option is used when determining couples for control
coupling. The default value is True for DIAB and Green Hills compilers, and
otherwise False.
```

Keep Intermediate Instrumentation Files (C/C++ only)

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Misc** sub-tab if it is not selected.

This option is set to True by default, meaning keep the intermediate files.

Users needing to save disk space can set the option to False, which directs VectorCAST to delete the preprocess output, usually in files named **<unit>.1.tu.preproc**, which can get quite large. For the option to take effect, all sources must be reinstrumented.



Note: When the option is False, some functionality is not supported:

- In VectorCAST Project Manager, the Change Impact Report and the Incremental Rebuild Report both indicate that all source changes are global, thus requiring a full rebuild after any source code modification.
- In a Cover environment, Probe Points and Covered By Analysis (CBA) results are not supported, therefore it is not possible to add them in the Coverage Viewer, and importing them displays an error.

This option must be True to enable full support for Change-Based Testing (CBT), Covered By Analysis (CBA), and probe points.

```
clicast -lc Option VCAST_KEEP_INSTRUMENTATION_INTERMEDIATE_FILES True |
False

Set this option to preserve intermediate files (.preproc) produced during
instrumentation. WARNING: This option must be True to enable full support
for Change-Based Testing (CBT), Covered By Analysis (CBA), and probe
points. The default value is True.
```

Keep Instrumentation Debug Files (C/C++ only)

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Misc** sub-tab if it is not selected.

The value for this option is False by default. When this option is set to True, VectorCAST retains the various debug files that are produced during instrumentation, including, for example: **<unit>.c.1.vcinst.in and <unit>.c.1.vcinst.out**.

```
clicast -lc Option VCAST_KEEP_INSTRUMENTATION_DEBUG_FILES True | False

Set this option to preserve debug files produced during instrumentation.
The default value is False.
```

Post-process Instrumented Files Command

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Misc** sub-tab.

The “Post-process instrumented files command” option provides a means to execute a batch file (Windows) or shell script (Linux) once after each unit in the environment is instrumented. This feature is useful for performing post-processing on the unit’s instrumented source code. Specify the full or relative (to the environment directory) path to the batch file or shell script in the option. The full path to the unit is passed as an argument to the script.

This option is written to the CCAST_.CFG file.

A very simple example of a Windows batch file (run_script.bat) is:

```
echo %1 >> report.txt
type %1 >> report.txt
```



Once the path to the example script is specified in the “Post-process instrumented files command” option and the environment is instrumented for coverage, this example script causes the file path to the unit and the instrumented file itself to be appended to the file **`report.txt`**, located in the environment directory. The first few lines of **`report.txt`** are shown below. These lines are repeated once for each unit in the environment.

```
C:\vcast_tutorial\CPP\Tutorial_cover\manager.cpp
/* VectorCAST/Cover */
#ifndef VCAST_CONDITION_TYP
#define VCAST_CONDITION_TYP int
#endif
#ifdef __cplusplus
extern "C" {
#endif

/*
---------------------------------------
-- Copyright 2019 Vector Informatik, GmbH  --
-- East Greenwich, Rhode Island USA --
---------------------------------------
*/
...
```

If you have an external text editor defined on the GUI tab, clicking the Edit File button (

) opens the script in the specified external text edit.

```
clicast -lc option VCAST_CMD_AFTER_INSTRUMENTATION <cmd>

Command to be executed after instrumenting a file for code coverage.
VectorCAST will pass 1 additional argument to this command: the path to the
instrumented file. It has no default value.
```

Post-process vcast_c_options.h Command

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Misc** sub-tab.

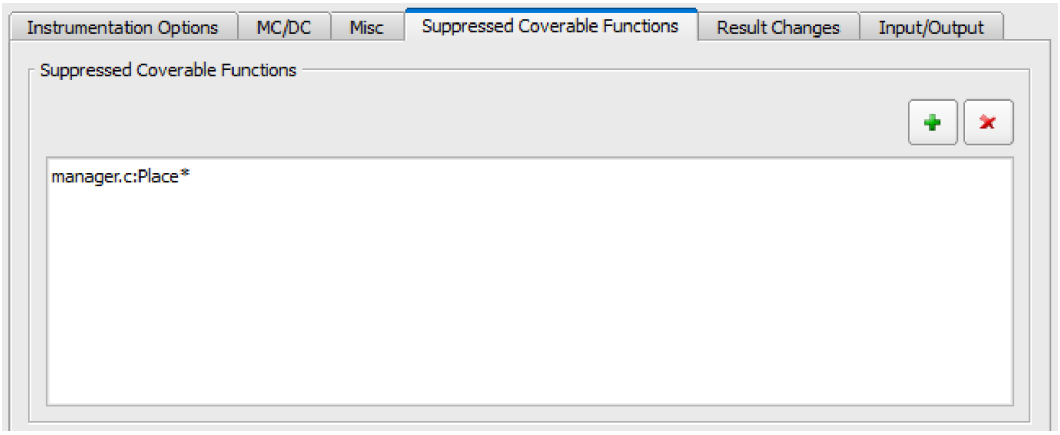
Similar to the "Post-process instrumented files command" option, this option allows customization of the **`vcast_c_options.h`** file after VectorCAST instruments or un-instruments the source files and updates the **`vcast_c_options.h`** file. This option applies to Unit Test and Cover environments with C/C++ source files.

If you have an external text editor defined on the GUI tab, clicking the Edit File button () opens the script in the specified external text edit.

```
clicast -lc option VCAST_CMD_AFTER_C_OPTIONS <cmd>

Command to be executed after vcast_c_options.h is written. The Clicast
option 'VCAST_CMD_AFTER_C_OPTIONS' allows passing additional arguments to
this command separated by spaces. The path to the vcast_c_options.h file is
automatically replaced as the first argument. Supports .py, .sh, and .bat
file extensions.
```

Suppressed Coverable Functions Option

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Suppressed Coverable Functions** sub-tab if it is not selected.



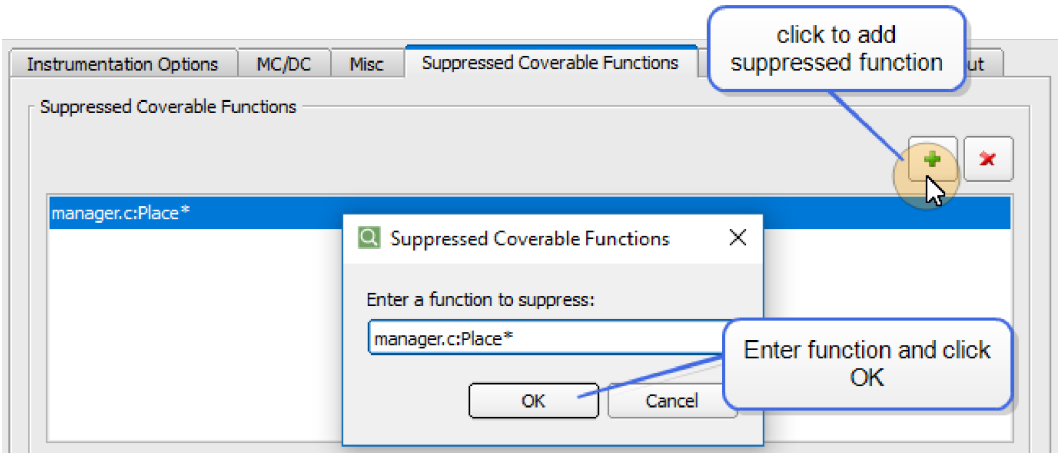
Suppressed Coverable Functions

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Suppressed Coverable Functions** sub-tab.

This option allows you to specify that coverage instrumentation for specific functions or whole files be suppressed in C/C++ source files.

Click the  button to add a file or function to be suppressed during instrumentation, using the following syntax:

```
file.ext:function (for C sources)
file.ext:class::function (for C++ sources)
```



The wildcard character (*****) can be used to match any text.

For example:

- **`manager.c:*`** suppresses instrumentation of all functions in **`manager.c`**
- **`manager.c:Place*`** suppresses instrumentation of any functions in **`manager.c`** starting with "**`Place`**"
- **`manager.cpp:Manager::Place*`** suppresses instrumentation of any functions in **`manager.cpp`**, in the class **`Manager`**, starting with "**`Place`**"
- **`templates.cpp:List<*>::push_back`** suppresses instrumentation for all types of the **`push_back`** function in the **`List`** template class. Suppressing only one type but not another is not supported.

Instrumentation for two source files in the Cover environment having the same name but residing in different locations is suppressed in both files.

The following examples show which functions are denylisted based on the rule, **`Rule -- Denylisted`**.

`PlaceOrder` – **`PlaceOrder`** everywhere

`Place*` – every function starting with **`Place`** everywhere

Manager::Place* – every function starting with **Place** in the C++ class **Manager**

manager.c:PlaceOrder – **PlaceOrder** in **manager.c**

man*:PlaceOrder – **PlaceOrder** in every file starting with **man**

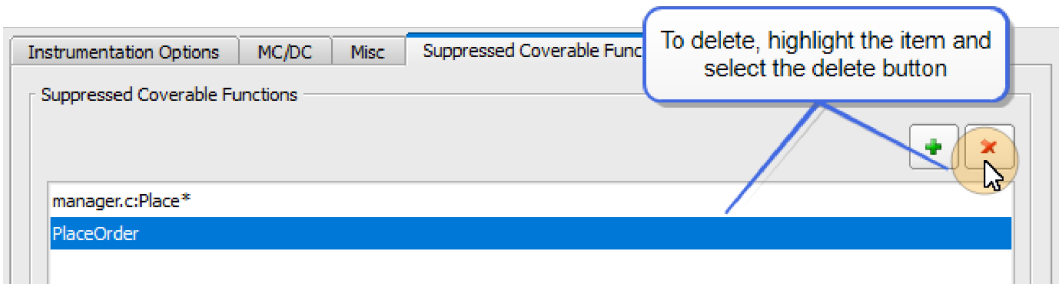
C:/src/manager.c:* – all functions in **manager.c**

/inc/manager.h: – all functions in the **manager.h** header file

/include/*: – all functions in all files in **include** directory

$(SRC_ROOT)/*:* – all functions in all files in the **$(SRC_ROOT)** path directory

To remove a file or function from the list, highlight the item and click the button.



Changes to this option require reinstrumentation to take effect.

```
clicast -lc option VCAST_SUPPRESS_COVERABLE_FUNCTIONS
<file.ext>:<function>,[<file.ext>:<function>]

To suppress directories, files or functions from being instrumented, enter
lines in the form of [path:]function-name.
```

Suppressing Test Code Coverage

Sometimes, the user might be required to add their test code to a Cover environment in order to collect coverage on the tested source code (e.g., to append cover io to a user's unit test main or to collect coverage for a library that is only comprised of header files). In these cases, the user is most likely not interested in measuring coverage on their test code.

The option VCAST_SUPPRESS_COVERABLE_FUNCTIONS is used to suppress coverage for directories, files or functions.

The following sections discuss several ways to suppress test code coverage.

Suppress a Test Code Directory

If you have a directory that only contains test code, the option VCAST_SUPPRESS_COVERABLE_FUNCTIONS should be used to suppress coverage for all of the coverable functions defined in that directory.

VCAST_SUPPRESS_COVERABLE_FUNCTIONS follows the convention **[path:]function-name** where the wildcard character (*) can be used to match any text. For example, if your test code

resides in the directory **/home/repo/progs**, the rule **/home/repo/progs/*:*** should be added to the Suppressed Coverable Functions list.

This path could also be made portable with the following rule, instead: ***/repo/progs/*:***



Note: Note: This rule would match any path that contains **repo/progs** (e.g. **/home/repo/progs and /home/repo/repo/progs**).

If an environment variable was used for a Cover environment base directory, it is recommended to use the rule **$(REPO)/progs/*:***, if the environment variable REPO was set to **/home/repo**.

Suppress a Test Code File

If your test code files do not reside in a specific directory, code coverage can be suppressed for individual files.

For example, the rule **$(REPO)/src/unit_1_tests.cpp** should be used to suppress test code coverage for the file **unit_1_tests.cpp** in the repository's **src** directory. If test code files follow the same naming convention, the rule **$(REPO)/src/*_tests.cpp** can be used to suppress code coverage for all files ending in **_tests.cpp** in the **src** directory.

Suppress Inlined Test Code

There are two ways coverage can be suppressed for inlined test code in user source files.

First, if the test function **my_unit_test_1** is defined in **repo/src/source.cpp**, the rule **$(REPO)/src/source.cpp:my_unit_test_1** can be added to the Suppressed Coverable Functions list to suppress coverage for that test function. If there are several test functions in that file that all start with the prefix **my_unit_test_**, the rule **$(REPO)/src/source.cpp:my_unit_test_*** should be used to suppress coverage for all of them. If all unit test functions in the repository start with the prefix **my_unit_test_**, the rule **$(REPO)/*:my_unit_test_*** should be used.

Second, if inlined test code functions do not adhere to a naming convention or the user does not wish to modify the Cover environment configuration, the **vcast_dont_instrument** comments can be used to suppress coverage for the test code defined within them.

In the example below, the user added the **vcast_dont_instrument** comments to their source file to suppress the functions **one_test** and **test_another**:

```
//vcast_dont_instrument_start

void one_test()
{
  MY_ASSERT( fn_1() );
}

void test_another()
{
  MY_ASSERT( fn_2() );
}

//vcast_dont_instrument_end
```

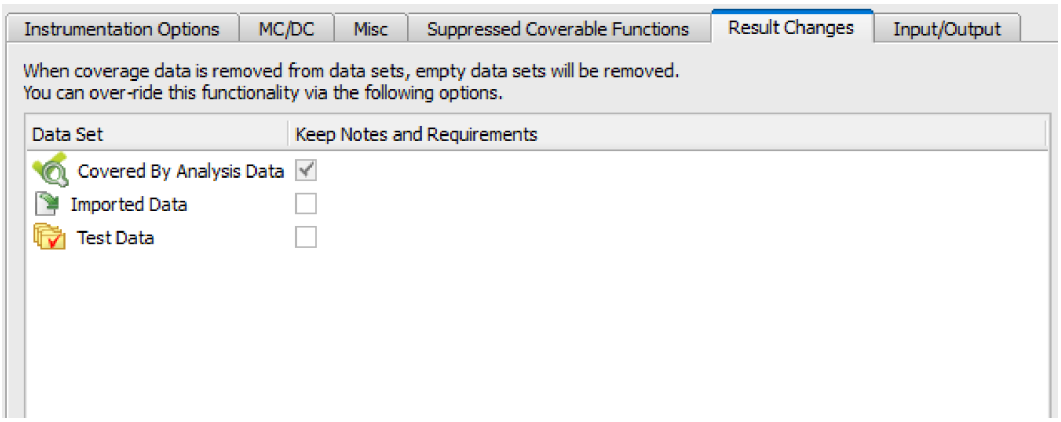
Coverage for Header Only Libraries

In order to collect coverage for header files, all source files that include them must be added to the Cover environment. If you wish to collect coverage for a library that is only comprised of header files, the option VCAST_SUPPRESS_COVERABLE_FUNCTIONS should be used to suppress coverage for all coverable functions in the source file directory(ies).

For example, if your repository contains the directories **`progs`**, **`src`**, and **`inc`**, and the **`inc`** directory contains your header library, the rules **`$(REPO)/progs/*:*`** and **`$(REPO)/src/*:*`** should be added to the Suppressed Coverable Functions list. This will only collect coverage for the header library in **`inc`**.

Result Changes

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Result Changes** sub-tab if it is not selected.

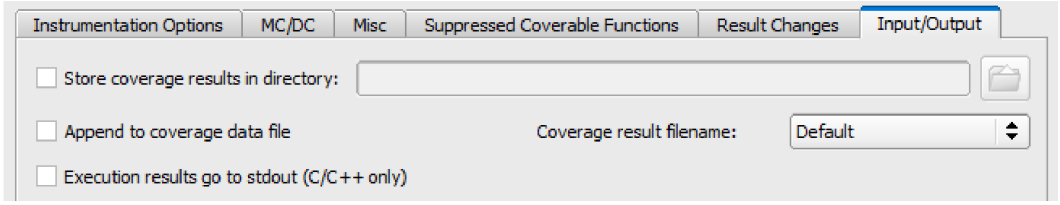


By default, when coverage data is removed from data sets the empty data sets will be removed from VectorCAST. To override this functionality, select the checkbox next to the data sets you wish to retain. There are three data set types:

- Covered by Analysis (CBA) data
- Imported Data
- Test Data

Input / Output Options

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Input/Output** sub-tab if it is not selected.



Store Coverage Results in Directory

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Input/Output** sub-tab.

This option specifies a custom directory for the coverage TESTINSS.DAT files generated by the instrumented executable. This option provides the ability to organize and maintain the coverage result files created during test execution. The specified directory can be absolute or relative and can include environment variable substitution syntax, e.g. **`$(ENV)/directory`**.

All environment variable substitution is done within the instrumented executable. This allows changing the coverage results directory per invocation of the same instrumented executable.

If a relative path is specified, the coverage result files are saved relative to the current working directory of the execution program.

If a directory is not specified, the coverage result files are saved to the current working directory of the executing program.

```
clicast -lc option VCAST_COVERAGE_RESULTS_DIRECTORY <path>

Save coverage result files to the specified directory. This option provides
the ability to organize and maintain the coverage result files created
during test execution. The specified directory can be absolute or relative
and can include environment variable substitution syntax, e.g.
$(ENV)/directory.
```

Append to Coverage Data File

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Input/Output** sub-tab.

This option tells VectorCAST to open the coverage data file for appending, rather than always creating the data file.

By default, coverage data generated by the instrumented executable is stored in a file named TESTINSS.DAT. To avoid overwriting this file during test execution, the filename can be made unique by selecting one of the following options from the **Coverage result filename** drop-down menu to append to the default file name:

- Default
- Append Epoch
- Append PID
- Append PID and Epoch

```
clicast -lc option VCAST_COVERAGE_RESULT_FILE_NAME File_Name_Append_Default
| File_Name_Append_Epoch | File_Name_Append_Pid | File_Name_Append_Pid_
Epoch

By default, the coverage data is stored in a result file named
TESTINSS.DAT. To avoid overwriting this file during execution, make the
```

```
filename unique by appending the Process ID (PID), Epoch time, or both to
that filename.
```

Execution Results go to stdout (C/C++ only)

Choose **Tools => Options** and click the **Coverage** tab. Then click the **Options** tab and the **Input/Output** sub-tab if it is not selected.

When this option is set to True, the content that would otherwise go to the TESTINSS.DAT file is sent to Standard Output after executing the instrumented application. This output can then be redirected to a results filename, such as MY_RESULT.DAT.

The option takes effect after instrumentation and is available for Cover environments with C/C++ source files only.

```
clicast -lc option VCAST_USE_STDOUT TRUE | FALSE

When this option is True, the content that would otherwise go into the
TESTINSS.DAT file is sent to standard output after executing the
instrumented application. This option can be set for Cover environments
with C/C++ source files only, and it takes effect after instrumentation.
The default value is False.
```

CLICAST-Only Coverage Options

```
clicast -lc option VCAST_EMPTY_STATEMENTS_COVERABLE True | False

When this option is true, the empty statement, ';', is considered
coverable.
```

```
clicast -lc option VCAST_HANDLE_TERNARY_OPERATOR True | False

Setting this option will cause the ternary (conditional) operator to be
treated as a branch. Changing this option causes basis paths to be
regenerated the next time the unit is instrumented.
```

```
clicast -lc option VCAST_INSTRUMENT_ASSIGNMENTS True | False

Setting this option to TRUE will cause assignment statements with boolean
or comparison operators to be covered by branch and MC/DC coverage. If this
option is TRUE, MC/DC will be suppressed for statements preceded by the
comment: VCAST_DONT_DO_MCDC. If this option is FALSE, MC/DC will still be
done on assignment statements preceded by the comment: VCAST_DO_MCDC.
```

```
clicast -lc option VCAST_INSTRUMENT_PARAMETERS [True | False]

Setting this option will cause parameters in function calls to be covered
by branch and MC/DC coverage. The default value is True.
```

clicast -lc option VCAST_INSTRUMENT_SEPARATES True | False

This option will cause separate source files which are included into the original source unit to be instrumented when initializing code coverage for C/C++ files. The default value is True.

clicast -lc option VCAST_REPORT_ON_IMPLICIT_DEFAULT_CASE True | False

Setting this option will cause the implicit default case not provided at the end of a switch-case block to be covered when instrumenting branch coverage. The default value is False.

The Options Append Command

For **clicast**, an append command is available to append values to an existing option. The main benefit is adding to an existing option, for example a missing flag, without having to repeat the entire option. A common use case is to update older commands to add the missing 64-bit parser flags.

In this example the original flags might be:

```
C_EDG_FLAGS: -w --g++ --gnu_version 60300 --type_info_in_std --type_traits_helpers --c++98
```

This is fixed with the '**OPTIONS_Append**' command:

```
clicast -e C_EDG_FLAGS OPTIONS_Append '--x86_64 --64_bit_target'
```

clicast [-lc | -lada] OPTIONS_Append <option> <value>

Append value to specified text-based option. Appended values are preceded by an inserted space, except for VCAST_ENVIRONMENT_FILES and VCAST_SUPPRESS_COVERABLE_FUNCTIONS, which use a comma instead.

Setting Coverage Viewer Options

Choose **Tools => Options** and click the **Coverage** tab, then click the **Coverage Viewer** sub-tab.

The **Coverage Viewer** tab enables you to control the fonts and colors used in the Coverage Viewer. By default, covered lines are shown in green, uncovered lines are shown in red, and non-executable statements are shown in black. It is recommended that you use a monospaced font for the Coverage Viewer (e.g. Courier) as that allows the tables and report to be aligned properly.

To Format the Text in the Coverage Viewer

Choose **Tools => Options** and click the **Coverage** tab, then click the **Coverage Viewer** sub-tab. Choose one of the five contexts you want to change.

Covered Lines – lines or branches that have been completely covered.

Partially Covered Lines – lines that have multiple outcomes where some but not all outcomes have been tested.

Covered-by-Analysis Lines - lines or branches that are fully or partially covered-by-analysis.

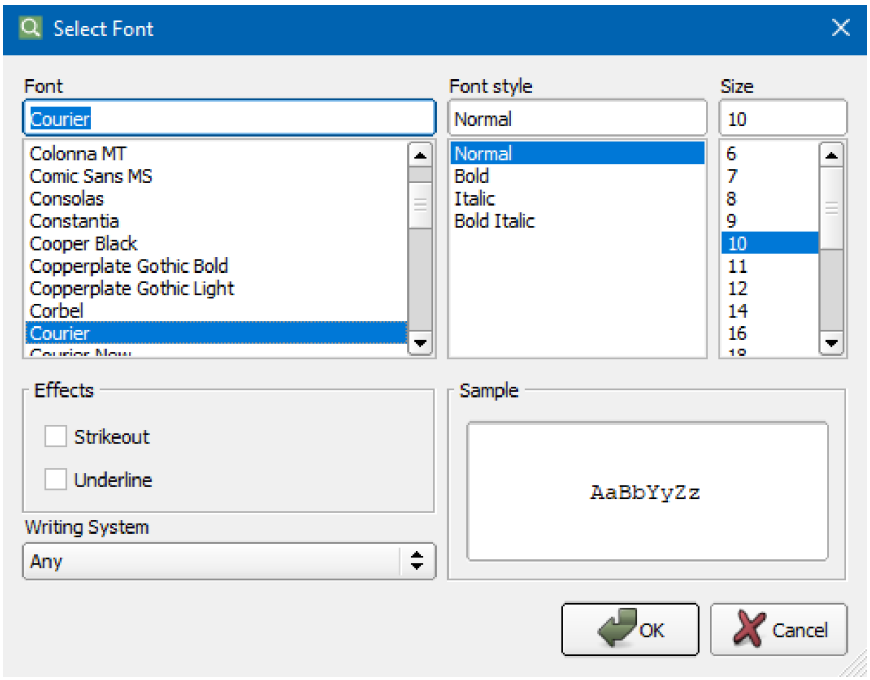
Uncovered Lines – lines or branches that have not yet been executed.

Uncoverable Lines – non-executable statements (e.g., comments).

To Change the Fonts

Choose **Tools => Options** and click the **Coverage** tab, then click the **Coverage Viewer** sub-tab.

The **Font...** button for each line type and the **Change All Fonts...** button enable you to select the font, font style and size for one or all line types in the Coverage Viewer. The **Default Fonts** button enables you to revert to the default font settings. Default font settings are: Courier font, Normal Font style, 10 pt Size.



To Reset the Fonts to Default

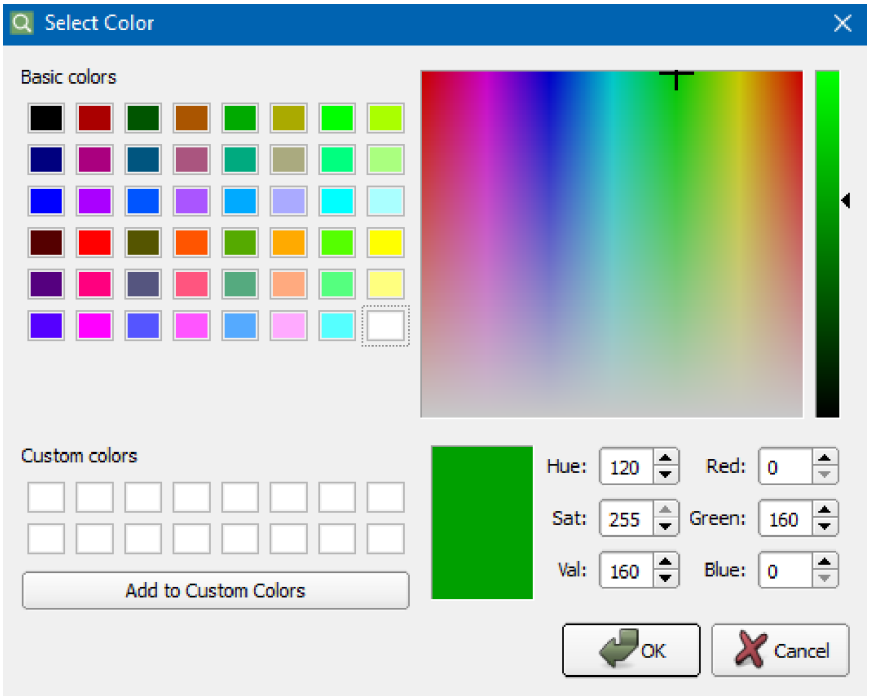
Choose **Tools => Options** and click the **Coverage** tab, then click the **Coverage Viewer** sub-tab.

Click the **Default Fonts** button. This option returns the font and color for each line type to the default settings: Courier font, Normal Font style, 10 pt Size.

To Change Text Color

Choose **Tools => Options** and click the **Coverage** tab, then click the **Coverage Viewer** sub-tab.

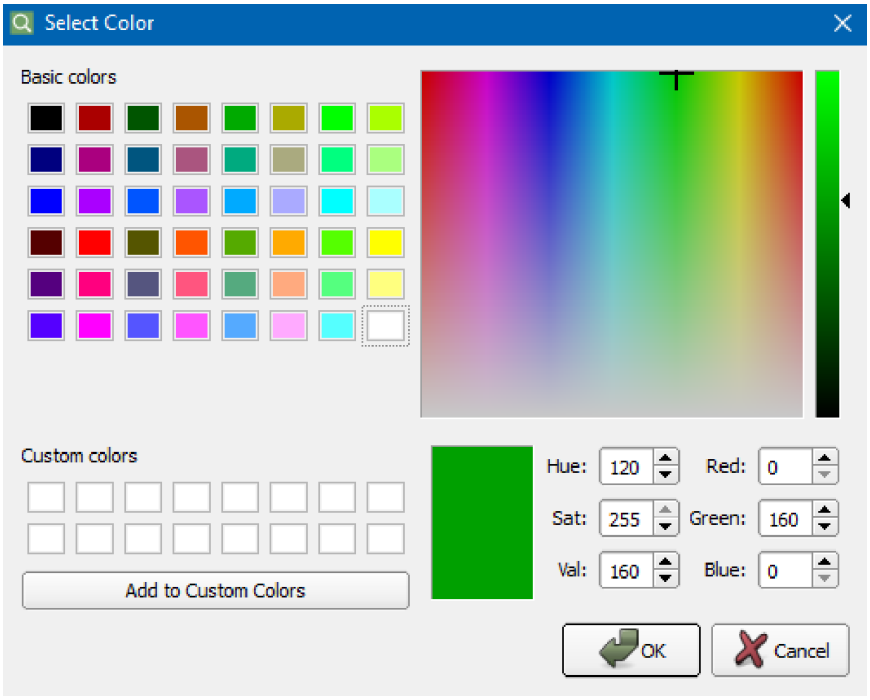
The **Text Color...** buttons enable you to change the text color for each of the five line types.



To Change Background Color

Choose **Tools => Options** and click the **Coverage** tab, then click the **Coverage Viewer** sub-tab.

The **Background Color...** buttons enable you to change the background color for the five line types.



Other Background Colors

Choose **Tools => Options** and click the **Coverage** tab, then click the **Coverage Viewer** sub-tab.

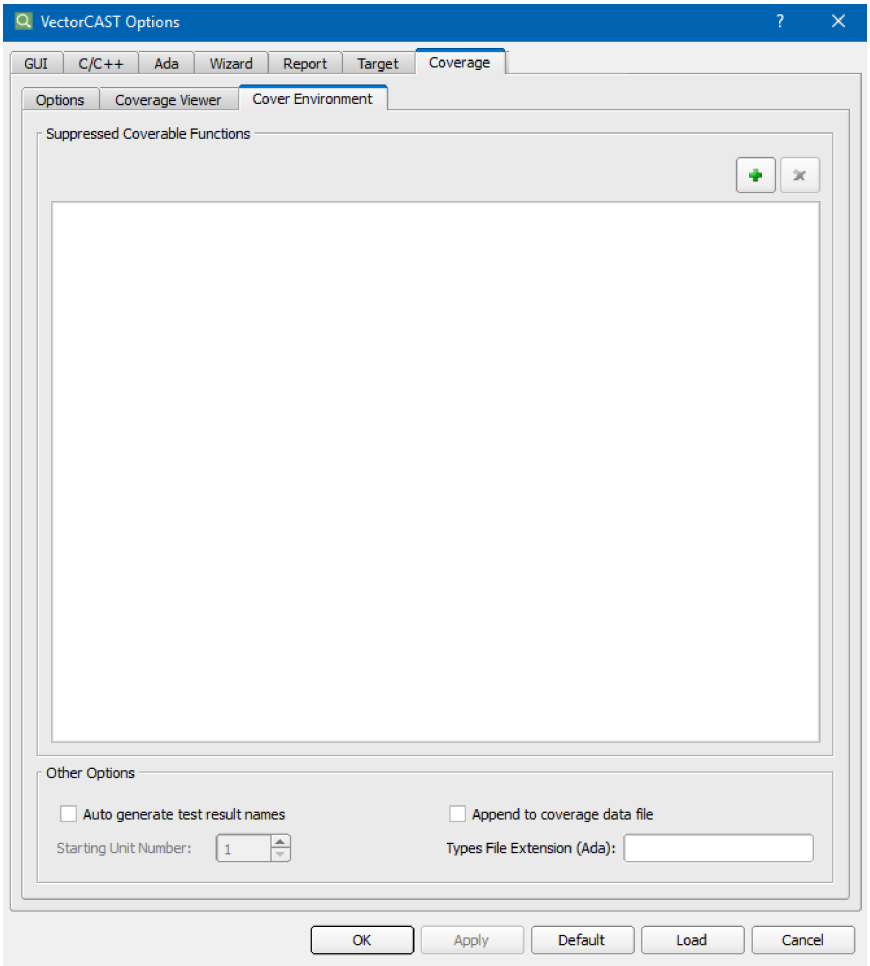
The **Selection...** button enables you to change the background color for selection highlights. This setting can be reset to the default color by selecting **View => Default Layout** from the Menu Bar.

The **Find...** button enables you to change the background color for search highlights. This setting can be reset to the default color by selecting **View => Default Layout** from the Menu Bar.

The **MC/DC Table...** button enables you to change the background color of MC/DC Tables in the Coverage Viewer. This setting can be reset to the default color by selecting **View => Default Layout** from the Menu Bar.

Cover Environment Options

Choose **Tools => Options** and click the **Coverage** tab, then click the **Cover Environment** sub-tab.



Auto Generate Test Result Names

Choose **Tools => Options** and click the **Coverage** tab, then click the **Cover Environment** sub-tab.

By default, adding a single test result file to a Cover environment asks for a name to be entered. When multiple test results are added at once, VectorCAST gives you a choice to automatically name them according to their filenames. This option causes VectorCAST to automatically name a *single* test result using the filename just as is done for multiple test result files. For example, if the test result file TESTINSS.DAT file is added and this option is on, the name of the test result is TESTINSS.

This option is written to the environment's .vcp file as AUTO_NAME_RESULTS: 0 | 1.

```
clicast -e <env> Cover Options AUto_generate_result_name Y | N

When this option is set VectorCAST will automatically generate the result
name for this result file. Its default value is No (0).
```

Append to Coverage Data File

Choose **Tools => Options** and click the **Coverage** tab, then click the **Cover Environment** sub-tab.



Note: For ADA, you must uncomment the appropriate block of code in the OPEN_FILE routine of 'vcast_cover_io.adb' before compiling the instrumented source files..

By default, this option is false, which means that the coverage results gathered during your application execution are written to a new TESTINSS.DAT file each time you invoke the application. If you do not rename the TESTINSS.DAT file in between application invocations, then data is overwritten. Setting the option to true causes VectorCAST to append new coverage results to the same TESTINSS.DAT file across consecutive invocations of your application.

This option is written to the environment's .vcp file as VCAST_APPEND_TO_TESTINSS: 0 | 1.

When set to 1 (True), it adds the line #define VCAST_APPEND_TO_TESTINSS 1 to the file vcast_c_options.h in the environment directory. which is #included by the file c_cover_io.c/cpp. Alternatively, you could use the compile define VCAST_APPEND_TO_TESTINSS when compiling instrumented source files into the instrumented application, or you can add this define as a compilation argument for a source file (**Environment => Edit Source Options**).

After changing this option, you must recompile the instrumented executable.

```
clicast -e <env> Options VCAST_APPEND_TO_TESTINSS True | False

This option tells VectorCAST to open the coverage data file for appending,
rather than always creating the data file. The default value is False..
```

Starting Unit Number

Choose **Tools => Options** and click the **Coverage** tab, then click the **Cover Environment** sub-tab.

This option allows you to set the unit number applied to the first source file added to a Coverage environment. This option is only necessary if you intend to combine coverage data files (TESTINSS.DAT) from several Cover environments into one Cover environment because the unit number is used in the TESTINSS.DAT file and there will be some collisions when several units are all number 1. Before adding any source files, specify that the first unit added to a particular environment should start with <num>, with the maximum being 100,000.

Setting this option is not necessary if you are importing from another environment (Coverage => Import Results from Environment) or importing a coverage script (Coverage => Import from Script) because VectorCAST does the unit-number translation for you.

This option is written to the environment's .vcp file as VCAST_MIN_STARTING_UNIT_NUMBER: <num>. Its default value is 1.

```
clicast -e <env> Cover Options STarting_unit_number <number>

This sets the starting unit number to be used for units added to the
coverage project. Note, this can only be changed prior to adding units to
the project.
```

Types File Extension (Ada)

Choose **Tools => Options** and click the **Coverage** tab, then click the **Cover Environment** sub-tab.

This option causes VectorCAST to put the coverage utilities in a file other than the instrumented unit. Set this option to the file extension you want to use (such as .ads). The filename generated is vcast_types_<unit number>.<file ext>, one for each unit instrumented.

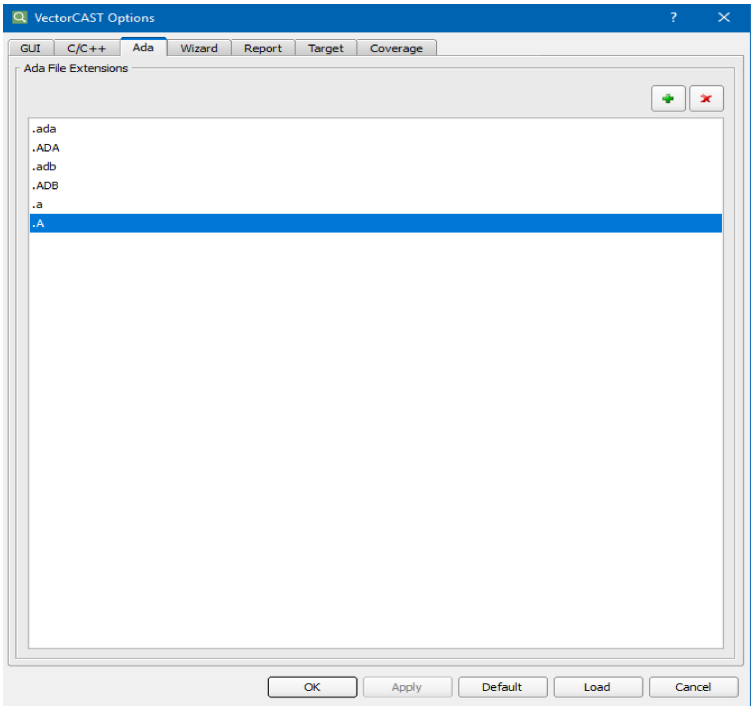
This option is stored in the CCAST_.CFG file.

```
clicast -lc option VCAST_COVER_SEPARATE_TYPES_FILES <ada file ext>

Specify that the coverage utilities be placed in a unit of type <ada file
ext> rather than the instrumented unit.
```

This option was formerly the environment variable VCAST_COVER_SEPARATE_TYPES_FILES.

Ada File Extensions

Choose **Tools => Options** and click the **Ada** tab.



Add/Remove/Default Extensions

A source file is processed based solely on the source file extension – if a source file extension does not match any extension in the lists, the file will not be processed. The **Add** and **Remove** buttons enable the user to manipulate the lists. The **Default** buttons reset the lists to the standard language file extension as shown above.

If your compiler uses a different extension than the default extensions provided, additional extensions can be added using the **Add** button. When adding source files recursively, any file matching one of these extensions is recognized as a source file.

```
clicast -e <env> cover options ADD_ADA_Extension <extension>

    Add a non-default filename extension to the list of recognized source
    file extensions. Its default values are *.ada *.ADA *.adb *.ADB *.a
    *.A
```

Setting Report Options

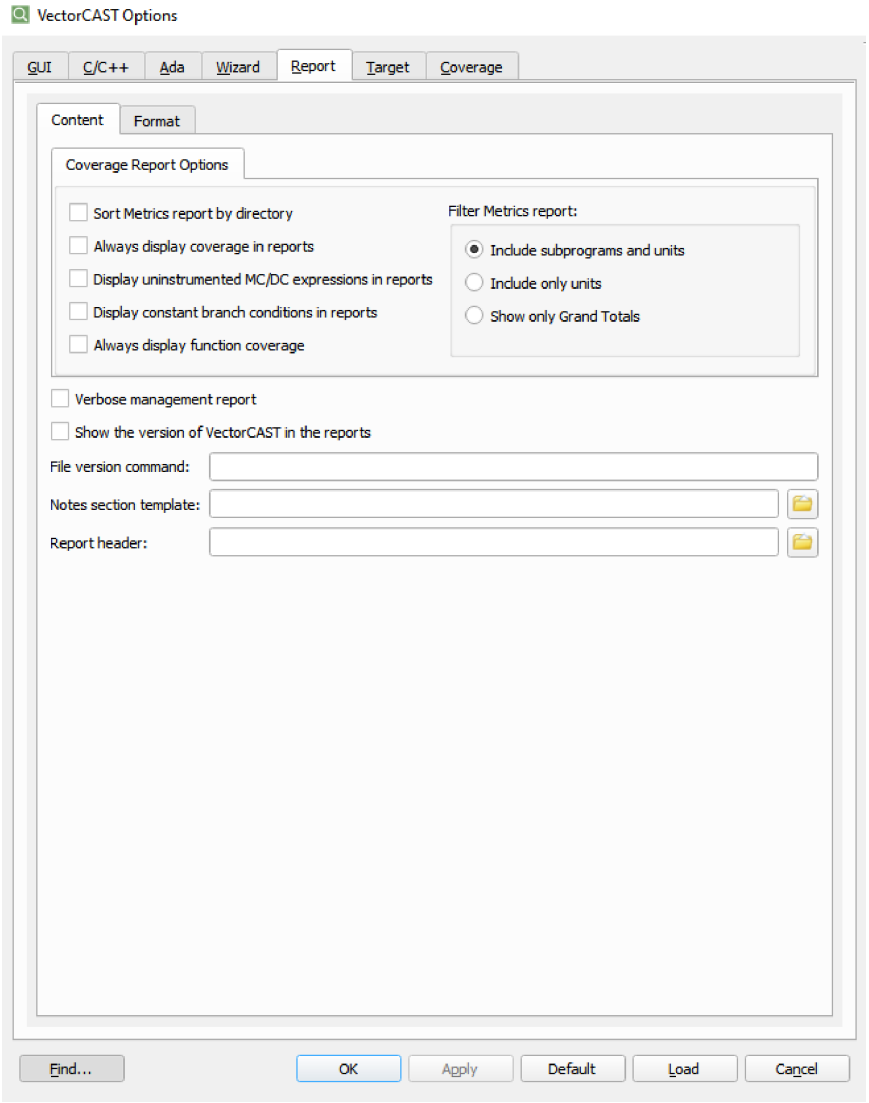
As with the Coverage options, the Report options are saved in the CCAST_.CFG file for VectorCAST Cover environments.

Report Content Options

Choose **Tools => Options** and click the **Report** tab, then click the **Content** sub-tab.

The **Content** sub-tab is used to change the way VectorCAST displays test execution reports and coverage reports on an environment-wide scale. Pass your cursor over any of the options to see an

explanation of that option in a tool-tip.



Sort Metrics Report by Directory

By default, the Metrics report lists all the units in the environment alphabetically. Setting this option causes the report to have one section per directory, sorted alphabetically. For each section, the full search directory path and the units from that directory are listed.

```
clicast -lc Option VCAST_SORT_METRICS_RPT_BY_DIR <True | False>

Set to true if you want the Metrics report to be sorted by search
directory, rather than by unit alphabetically (default). The default value
is False.
```

Always Display Coverage in Reports

By default, the Aggregate Coverage Report includes all units in the environment and if they have no coverage data, “No coverage data exists” is written for that unit. Turning on this option causes the annotated source code for all units to be included in the Aggregate Coverage Report, even if a unit has no coverage data.

```
clicast -lc option VCAST_DISPLAY_EMPTY_COVERAGE True | False

In the Aggregate Coverage Report, include the annotated source code for all
units in the environment, even if a unit does not have coverage data. If
false, “No coverage data exists” is displayed for those units instead. The
default value is False.
```

Display Uninstrumented MCDC Expressions in Reports

To display uninstrumented expressions in the Metrics Report when MC/DC or Statement + MC/DC coverage is initialized, set the "**Display uninstrumented MCDC expressions in reports**" option. This option adds an Uninstrumented Expressions section to the Metrics Report that identifies the unit, subprogram and line number of any MC/DC expressions that require MC/DC analysis but have not been instrumented by VectorCAST.

The report contains the following information which the user may use to refactor the expression so that VectorCAST is able to instrument it completely:

- The unit name that contains the uninstrumented expression
- The subprogram that contains the expression
- The operator in the expression that was not instrumented for MCDC or Statement + MCDC
- The line number in the original source file on which the expression occurs
- The column on the indicated line in the original source file where the operator occurs



Note: Uninstrumented expressions may be due to the user having turned off MC/DC instrumentation by setting options such as "**Instrument logical expressions in assignment statements**" to False. Contact VectorCAST Technical Support for clarification on the effects of instrumentation options on the content of the Uninstrumented Expressions report.

```
clicast -lc option VCAST_DISPLAY_UNINST_EXPR TRUE | FALSE

The default value is FALSE, except for Industry Mode "DO-178 B/C
(Avionics)", for which it is True.
```

Display Constant Branch Conditions in Reports

When enabled, this option adds a new section to the Metrics report that identifies the unit, subprogram, and line number of any branch or MC/DC expressions that have constant values, such as **`"if (1)"`**.

```
clicast -lc Option VCAST_DISPLAY_CONST_BRANCH_CONDITIONS <True | False>

When True, this option adds a section to the Metrics report that identifies
the unit, subprogram, and line number of any branch or MC/DC expressions
that have constant values, such as "if (1)". The default value is False.
```

Filter Metrics Report

This option enables you to reduce the detail that is displayed in the Metrics report. By default the option is set to "Include subprograms and units," (No_Filtering) which generates the familiar Metrics report with all units and subprograms included.

Include Subprograms and Units (NO_FILTERING)

Metrics Report

Configuration Data

Environment Name	FILTER_REPORT
Date of Report Creation	15 FEB 2019
Time of Report Creation	10:28:38 AM

Metrics

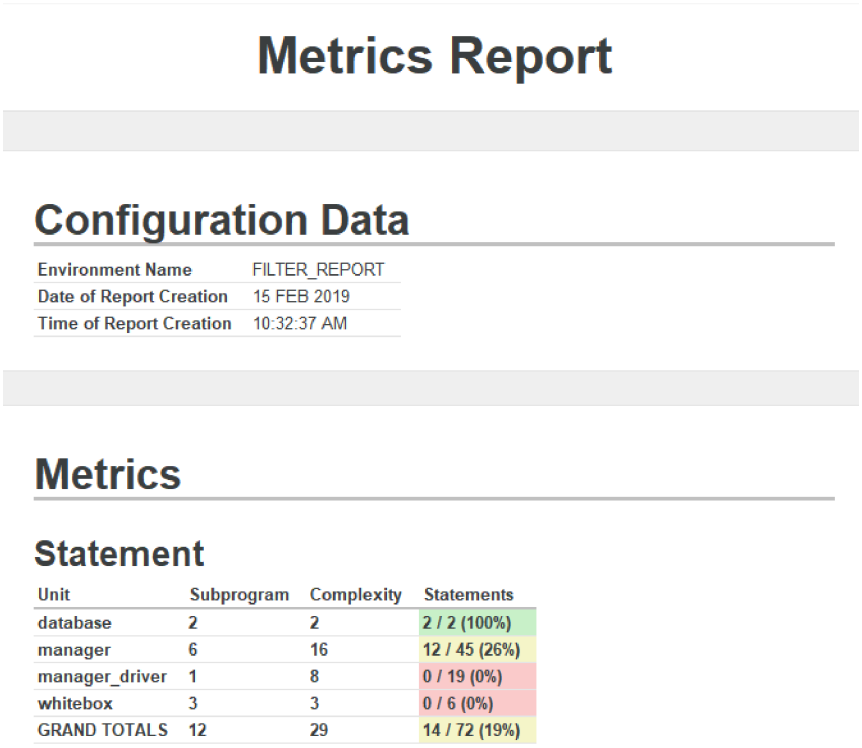
Statement

Unit	Subprogram	Complexity	Statements
database	Get_Table_Record	1	1 / 1 (100%)
	Update_Table_Record	1	1 / 1 (100%)
TOTALS	**2**	**2**	**2 / 2 (100%)**
manager	Add_Included_Dessert	3	2 / 4 (50%)
	Place_Order	5	10 / 17 (58%)
	Clear_Table	2	0 / 12 (0%)
	Get_Check_Total	1	0 / 2 (0%)
	Add_Party_To_Waiting_List	3	0 / 7 (0%)
	Get_Next_Party_To_Be_Seated	2	0 / 3 (0%)
TOTALS	**6**	**16**	**12 / 45 (26%)**
manager_driver	main	8	0 / 19 (0%)
TOTALS	**1**	**8**	**0 / 19 (0%)**
whitebox	InitDay	1	0 / 1 (0%)
	InitColor	1	0 / 1 (0%)
	Initialize	1	0 / 4 (0%)
TOTALS	**3**	**3**	**0 / 6 (0%)**
GRAND TOTALS	**12**	**29**	**14 / 72 (19%)**

The other choices for filtering reduce the amount of detail in the report. Choosing "Include only units" (Subprogram_Detail) causes the Metrics report to show only the unit names and their totals for the

number of subprograms, complexity, and coverage.

Include only Units (SUBPROGRAM_DETAIL)



Metrics Report

Configuration Data

Environment Name	FILTER_REPORT
Date of Report Creation	15 FEB 2019
Time of Report Creation	10:32:37 AM

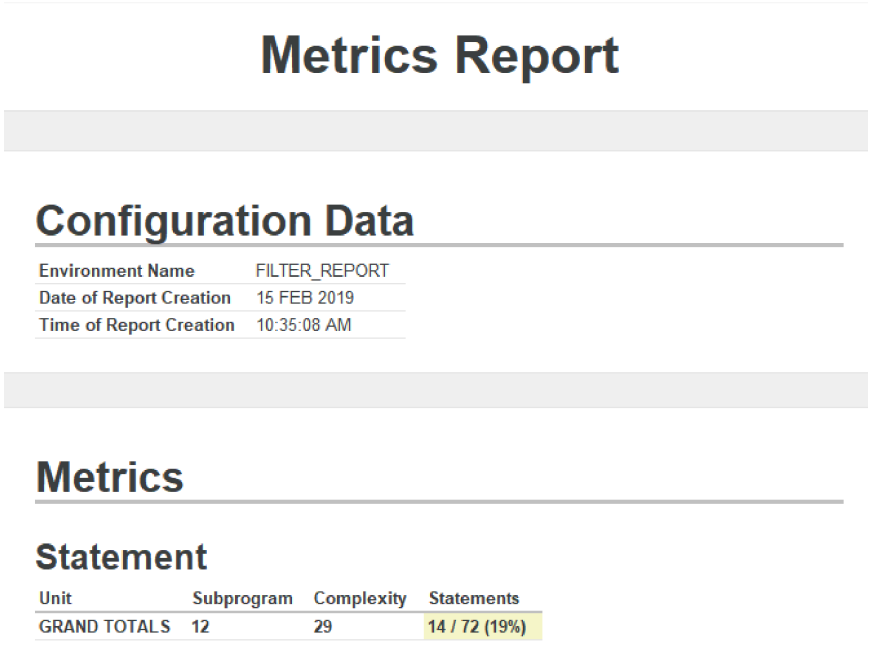
Metrics

Statement

Unit	Subprogram	Complexity	Statements
database	2	2	2 / 2 (100%)
manager	6	16	12 / 45 (26%)
manager_driver	1	8	0 / 19 (0%)
whitebox	3	3	0 / 6 (0%)
GRAND TOTALS	12	29	14 / 72 (19%)

Choosing "Show only Grand Totals" (Unit_Detail) causes the Metrics report to show only the very last line, the GRAND TOTALS for each column.

Show only Grand Totals (UNIT_DETAIL)



Metrics Report

Configuration Data

Environment Name	FILTER_REPORT
Date of Report Creation	15 FEB 2019
Time of Report Creation	10:35:08 AM

Metrics

Statement

Unit	Subprogram	Complexity	Statements
GRAND TOTALS	12	29	14 / 72 (19%)

Note: If the option "Sort Metrics report by directory" is also set, then the "Show only Grand Totals" setting causes the Metrics report to have a single GRAND TOTAL line for each directory.

```
clicast -lc Option METRICS_TOTAL_LEVEL_DISPLAY No_Filtering | Subprogram_
Detail | Unit_Detail

Specify which results to filter out of the Metrics Report. The default
value is No_Filtering.
```

Verbose Management Report

When True, this option copies the contents of the 'Test Notes' box in the Test Case Editor to the Management Report. One possible use for this is for test requirement numbers to be correlated to test cases in the Management Report.

```
clicast -lc option VCAST_VERBOSE_MANAGEMENT_REPORT True | False

Copies the contents of the 'Test Notes' box in the Test Case Editor to the
Test Case Management Report. Its default value is FALSE.
```

Show the Version of VectorCAST in the Reports

When this option is set, the version and build date of VectorCAST is added to the Configuration section of the following reports:

- Test Case Management report
- Aggregate Coverage report

- Metrics report
- Test Case Data report
- Execution Results report

```
clicast -lc option VCAST_RPTS_SHOW_VERSION True | False

Show the VectorCAST version in the configuration section of the reports.
The default value is False
```

File Version Command

If a command is supplied, the command is executed on each UUT source file and any non-stubbed units. The results of the command are included in the Test Case Management Report and the Full Test Report.

For example, setting this option to **`ls -l`** (on Linux) causes the following information to be added to the two reports:

```
FILE VERSION REPORT
  Unit Name: File Name and Version Information (ls -l)
  manager : -rwxr-xr-x   1 mdm     None        2012 May 18 14:31
C:\Cygwin\home\mdm\cvs\sqa_testing\SOURCE\C\TUTORIAL\manager.c
```

The analogous command on Windows is: **`dir /b`**.

On Linux, VectorCAST executes the following command:

```
<supplied command> <full path to each UUT>
```

On Windows, VectorCAST executes the following command:

```
cmd /c "<supplied command> <full path to each UUT>"
```

The full path to each UUT is derived from the Source directories used in the environment.

```
clicast option VCAST_FILE_VERSION_COMMAND <command>

Command executed for each source file in the test environment, and a
section added to the Test Case Management and Full reports listing the name
of each unit in the test environment and the information generated by this
command. If this command is empty, then the File Version section fo the
report will not be created. <command> is a quoted string. It has no default
value.
```

Notes Section Template

In the Test Case Notes tab, you can import a text template. This feature is useful if you want to document the requirements tested by each test case following a standard format. Before importing the template, first specify the template file. To do this, choose **Tools => Options** dialog, **Report** tab,

Content sub-tab. Enter the full path to the template file for the option “Notes section template.”

To use the template in test case Notes, first specify the template file. If the Notes tab of a test case is empty and a template file is specified, then the template is automatically loaded when the test case is opened. You can also right-click in the Notes tab and choose Import template. The text from the template file appears in the Notes tab at the cursor location.

```
clicast option VCAST_NOTES_SECTION_TEMPLATE <Path to Template text file>

This option allows you to enter a path to a text file that contains a
template for the Notes section of the test case.
```

Report Header

This option prepends a text string to the title or adds a new section to the Full Report, just below the title, to include the contents of a file in TEXT or HTML format. Input can be a text string, or a file containing several lines.

If an HTML file is provided, the contents should provide a ** ** section with a table. For example:

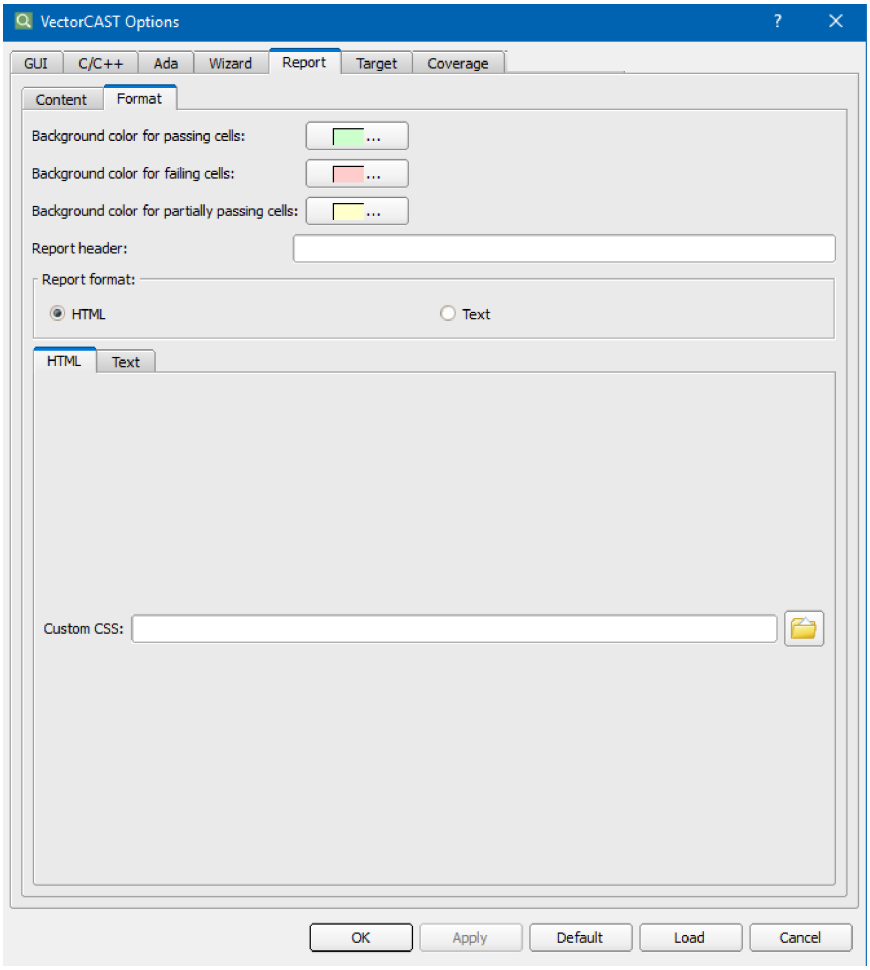
```
<div class='report-block'>
  <h2>Additional Data</h2>
  <table class='table table-small'>
    <tr><th>Tests created by</th><td>Fred Williams</td></tr>
    <tr><th>Source revision</th><td>123abc789</td></tr>
  </table>
 </div>
```

```
clicast option VCAST_RPTS_HEADER <text to prepend to title> | <header text>
| <html file>

Prepend a text string to the title or add a new section to the Full Report,
just below the title, to include the contents of a file in TEXT or HTML
format. Input can be a text string, or a file containing several lines. If
an HTML file is provided, the contents should provide a <div> section with
a table.
```

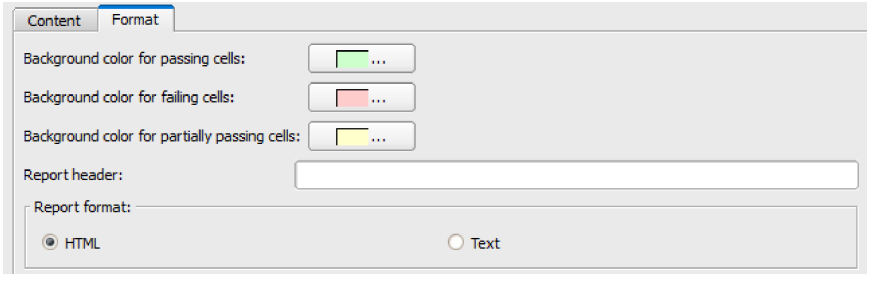
Report Format Options

Choose **Tools => Options** and click the **Report** tab , then click the **Format** sub-tab.

The **Format** sub-tab is used to change the formatting options for HTML and text reports. Pass your cursor over any of the options to see an explanation of that option in a tool-tip.



Setting Format Options



Background Color for Passing Cells

Choose **Tools => Options** and click the **Report** tab, then click the **Format** sub-tab. This background color is used for passing data cells in the Parameter Tree.

```
clicast VCAST_RPTS_TABLE_DATA_PASS_BGCOLOR <color>

Background color used for passing data cells in tables in the UI. The
default color is #ccffcc, a light green.
```

Background Color for Failing Cells

Choose **Tools => Options** and click the **Report** tab, then click the **Format** sub-tab. This background color is used for failing data cells in the Parameter Tree.

```
clicast option VCAST_RPTS_TABLE_DATA_FAIL_BGCOLOR <color>

Background color used for failing data cells in tables in the UI. The
default color is #ffffcc, a light pink.
```

Background Color for Partially Passing Cells

Choose **Tools => Options** and click the **Report** tab, then click the **Format** sub-tab. This background color is used for partially passing data cells in the Parameter Tree.

```
clicast option VCAST_RPTS_TABLE_DATA_PARTIAL_BGCOLOR <color>

Background color used for partially passing data cells in tables in the UI.
The default color is #ffffcc, a light yellow.
```

Report Format

Choose **Tools => Options** and click the **Report** tab, then click the **Format** sub-tab.

The Report Format option determines if reports are displayed in HTML or Text. The default setting is HTML. If you choose HTML, you can view the reports within VectorCAST or in an external browser. If you choose text, you can view them within VectorCAST or in an external text editor.

Note that a test does not need to be re-executed in order to see the Execution report in a different format.

```
clicast -lc option VCAST_CUSTOM_REPORT_FORMAT HTML | TEXT

Output format for VectorCAST reports: HTML or TEXT. The default value is
HTML.
```

Setting Custom CSS

Choose **Tools => Options** and click the **Report** tab, then click the **Format** sub-tab, and the **HTML** sub-tab.

To apply a custom CSS file to use in the HTML reports, enter the path to the custom CSS file.Your custom style sheet is appended when reports are generated and the contents of the custom style sheet are embedded in the **<style>** section following the default styles, when the reports are

generated.

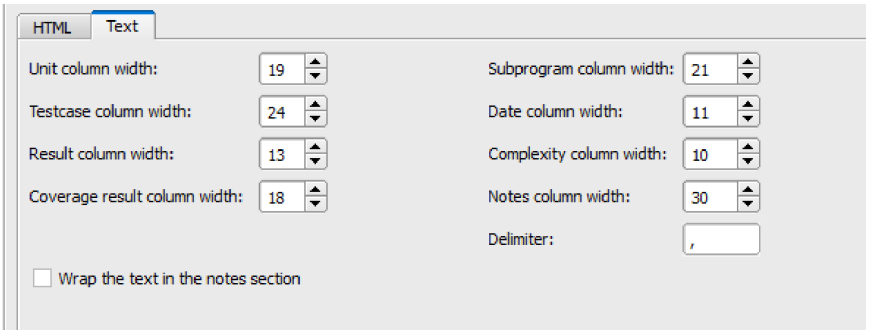
If the option is not set or the file cannot be found, the default CSS is used. The default CSS is located at **$VECTORCAST_DIR/python/vector/apps/ReportBuilder/css**.

```
clicast -lc option VCAST_RPTS_CUSTOM_CSS <path to custom.css file

Reports use the default cascading style sheets (*.css) located in
$VECTORCAST_DIR/python/vector/apps/ReportBuilder/css/, unless a custom
style sheet is specified in this option. Environment variables in the path
to the custom style sheet are supported, using this syntax $(ENV_VAR). When
used, the contents of the custom style sheet are embedded in each report,
between the <style>...</style> tags in the header section after the default
styles allowing for CSS style overriding.
```

Text Report Options

Choose **Tools => Options** and click the **Report** tab, then click the **Format** sub-tab, and the **Text** sub-tab.

The **Text** sub-tab has options that are honored when the Report Format is Text.



Unit Column Width

The Unit Column Width option specifies the width of the “Units” column in Text reports.

The default Unit column width is 19 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_UNIT_COLUMN_WIDTH <width>

Width (in characters) of unit column in text reports. Its default value is
19.
```

Subprogram Column Width

The Subprogram Column Width option specifies the width of the “Subprogram” column in Text reports.

The default Subprogram column width is 21 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_SUBPROGRAM_COLUMN_WIDTH <width>

Width (in characters) of subprogram columns in text reports. Its default
value is 21.
```

Testcase Column Width

The Testcase Column Width option specifies the width of the “Test Case” column in Text reports.

The default Testcase column width is 24 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_TESTCASE_COLUMN_WIDTH <width>

Width (in characters) of testcase column in text reports. Its default value
is 24.
```

Date Column Width

The Date Column Width option specifies the width of the “Date” column in Text reports.

The default Date column width is 11 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_DATE_COLUMN_WIDTH <width>

Width (in characters) of date column in text reports. Its default value is
11.
```

Result Column Width

The Results Column Width option specifies the width of the “Results” (i.e. “Pass/Fail”) column of Text reports.

The default Results column width is 16 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_RESULT_COLUMN_WIDTH <width>

Width (in characters) of result columns in text reports. Its default value
is 16.
```

Complexity Column Width

The Complexity Column Width option specifies the width of the “Complexity” column in Metrics tables of Text reports.

The default Complexity column width is 10 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_COMPLEXITY_COLUMN_WIDTH <width>

Width (in characters) of complexity column in text reports. Its default
value is 10.
```

Coverage Result Column Width

The Coverage Results Column Width option specifies the width of the “Coverage Results” (i.e. “Statement Coverage”) column of Text reports.

The default Coverage Results column width is 18 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_COVERAGE_RESULT_COLUMN_WIDTH <width>

Width (in characters) of coverage result columns in text reports. Its
default value is 18.
```

Notes Column Width

The Notes Column Width option specifies the width of the “Notes” column in Text reports.

The default Notes column width is 30 characters. To change the width, specify another size in the spin box.

```
clicast option VCAST_RPTS_NOTES_COLUMN_WIDTH <width>

Width (in characters) of notes columns in text reports. Its default value
is 30.
```

Delimiter

The character separator for the Cover report created by the CLICAST report commands:**`cover report csv_metrics`** and **`reports alternate`**. Enter the character in quotes, as in “@”. To enter a tab, use “\t”. Valid delimiters are: ? , ' ; | { } [ ] @ ~ # $ _ \t \n

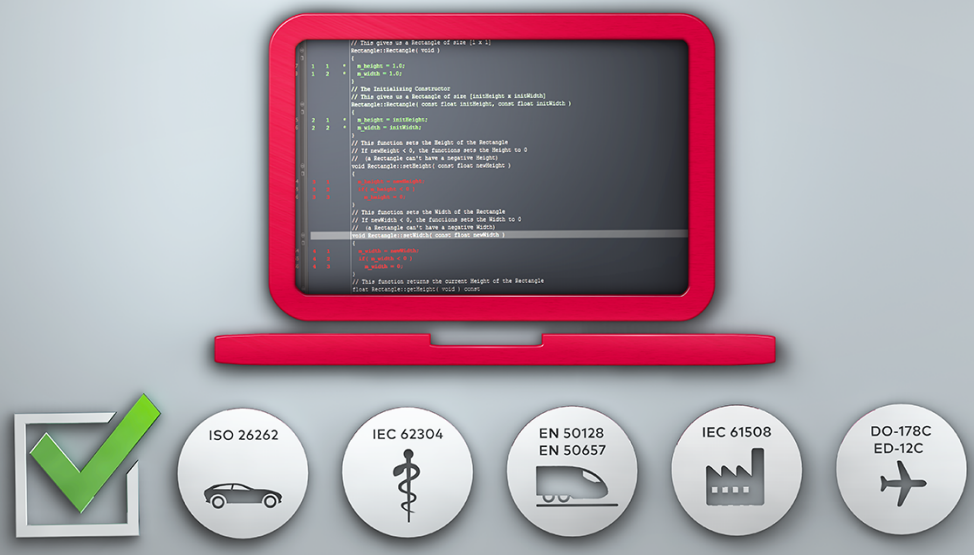
```
clicast Option VCAST_RPTS_DELIMITER <delimiter>

Character separator used in two CLICAST report commands cover report csv_
metrics and reports alternate. The default value is comma ",".
```

Wrap Text in the Notes Section

Setting this option causes the text in the Notes section to wrap at 80 characters in the Test Case Data and Full Reports in either TEXT or HTML format.

clicast Option VCAST_RPTS_WRAP_NOTES True | False

This option will cause the text in the notes section to wrap at the first space before 80 characters. The default value is FALSE.



Change-Based Testing (CBT)

Change-Based Testing

Change-Based Testing (CBT) automatically identifies the minimum tests that must be run for each code change. VectorCAST scans the code for changes since the last test run, identifies the sub-set of tests that are affected by the change and automatically re-runs only those tests.

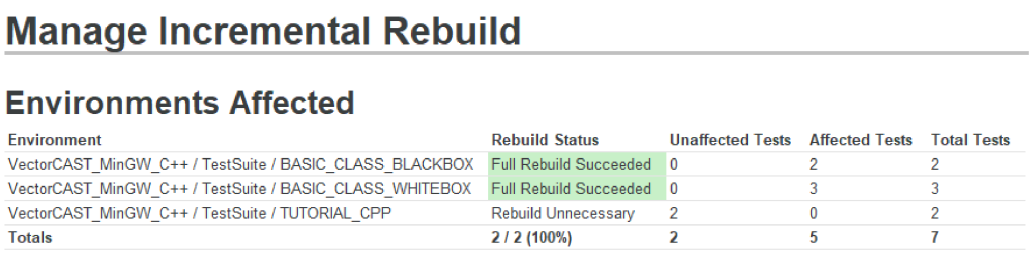
Change-Based Testing results in greatly reduced test time, allowing for more frequent testing, and ensuring that bugs are fixed when they are introduced, instead of during a full test cycle at a later date.

To perform an incremental build, right-click on a node in the Project Tree and select **Build/Execute => Incremental** from the context menu. Upon completion, the Manage Incremental Rebuild Report is produced and displayed in the MDI window.

Performing **Build/Execute => Incremental** on a migrated Cover environment will:

- Create the environment if it does not exist in the workspace,
- Update the sources from the base directories,
- Do an incremental instrument on the source files and remove any affected results/testcases, and
- Execute any removed tests.

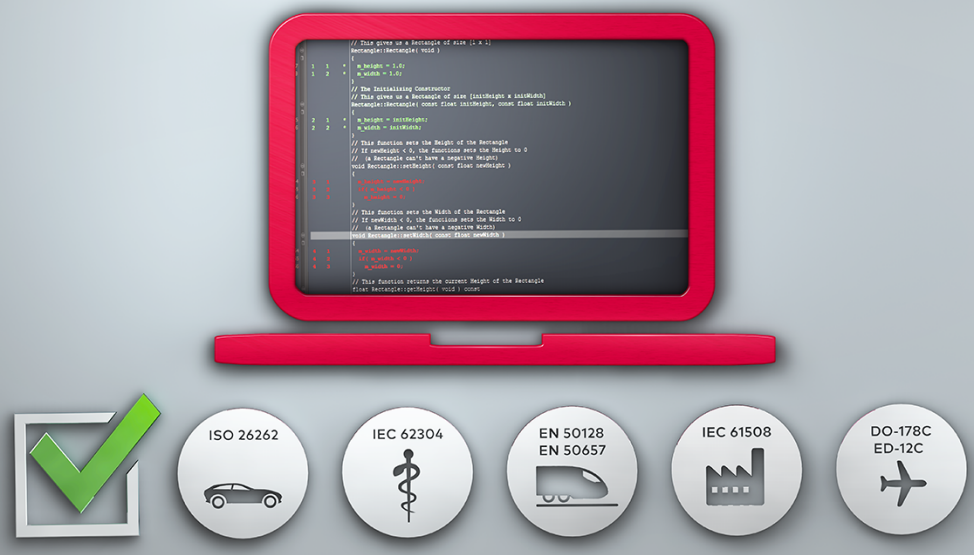
After an incremental rebuild the Manage Incremental Rebuild Report is produced to summarize the results. The report provides the status from the Incremental Rebuild, the number of tests preserved for that environment, and the number of tests that required re-execution.



Manage Incremental Rebuild

Environments Affected

Environment	Rebuild Status	Unaffected Tests	Affected Tests	Total Tests
VectorCAST_MinGW_C++ / TestSuite / BASIC_CLASS_BLACKBOX	Full Rebuild Succeeded	0	2	2
VectorCAST_MinGW_C++ / TestSuite / BASIC_CLASS_WHITEBOX	Full Rebuild Succeeded	0	3	3
VectorCAST_MinGW_C++ / TestSuite / TUTORIAL_CPP	Rebuild Unnecessary	2	0	2
Totals	**2 / 2 (100%)**	**2**	**5**	**7**



Using VectorCAST Covered By Analysis

Covered By Analysis (CBA)

Covered By Analysis (CBA) allows users to augment test coverage metrics with coverage analysis data sets. Small portions of embedded applications are commonly impossible to test, but regulated industries require documented analysis of uncovered code to meet the requirement of 100% structural coverage.

VectorCAST CBA provides the ability to do code analysis directly within VectorCAST using the Coverage Analysis Editor and combines the test and analysis coverage metrics in a single report. VectorCAST CBA can also import analysis files, including those generated by third party tools.

These examples are not intended to be step-by-step tutorials, but rather an overview of the tool features that are applicable to each workflow. All examples were run using Windows.



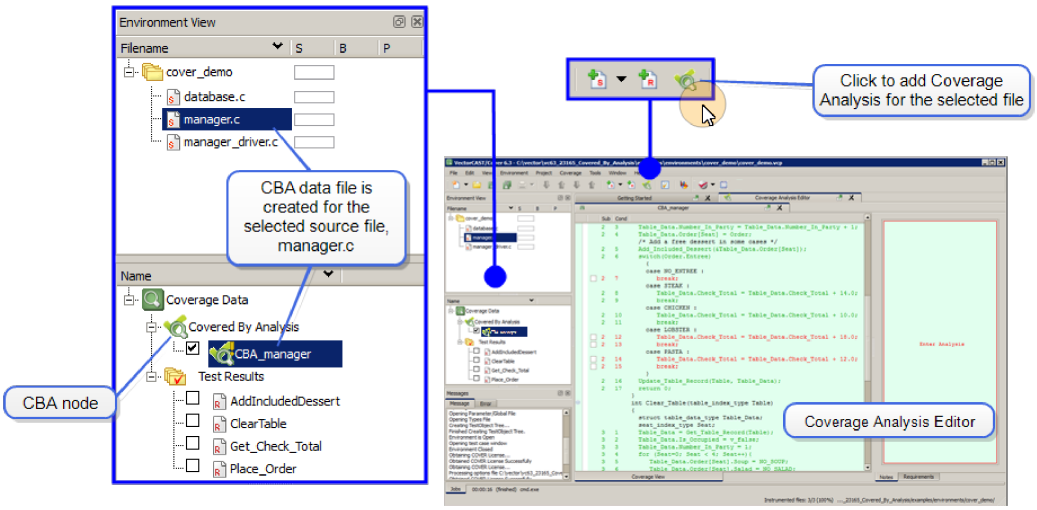
Note: For some time, we have been updating VectorCAST to use an alternate coverage viewer mode called Source File Perspective (SFP). Normally coverage is viewed using the pre-processed file or translation unit. As an alternative to Translation Unit (TU) viewing, you can choose to view the coverage from the original Source File Perspective. There are still more functions in VectorCAST to be updated for this viewing mode. But if you find this view valuable, please let us know as we continue to update other features of VectorCAST to utilize this new viewing mode.

CBA has been updated to use SFP as an option. A section, "Using Coverage Analysis With SFP" on page 257, has been added to end of this chapter to show the workflow using this new perspective.

To Add Coverage Analysis

The environment must first be instrumented for coverage. A CBA Analysis result is associated with one source file. To add Coverage Analysis for a selected source file, select the **CBA** button  on the Toolbar. Clicking the **CBA** button requires a source file to be selected. Alternatively, right-click a source filename in the Source Files pane and select **Add Coverage Analysis** from the context menu.

The Coverage Analysis Editor opens in the MDI Window and a Covered By Analysis node displays in the Results pane. A CBA data file is created for the selected source file and this Analysis result displays beneath the Covered By Analysis node. When an Analysis result is first created, it is empty and contains no data. Empty Analysis results display the  icon, even if they contain notes or requirements. Each CBA data file corresponds to a single source file, but you can create multiple Analysis results per source file.



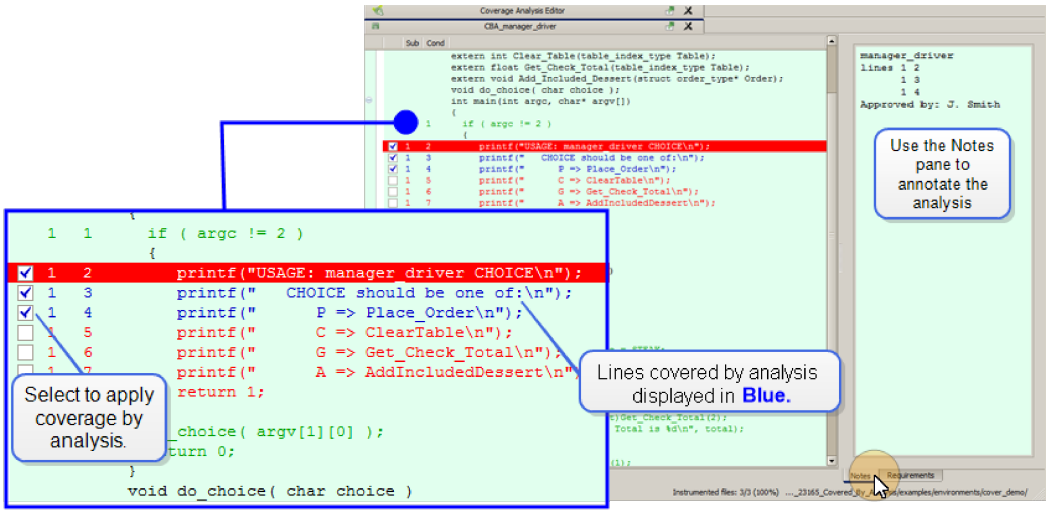
Using the Coverage Analysis Editor

Once an Analysis result has been created, the Coverage Analysis Editor can be opened at any time by double-clicking on the Analysis result.

In the Coverage Analysis Editor, the Notes tab on the right is a free-form text editor allowing you to annotate the associated analysis. Use the Requirements pane to trace the Project Requirements and Test Case Requirements associated with the selected code. The Requirements tab is populated by VectorCAST Requirements Gateway. Use the **Save** button to save inputs.

Statement Coverage

With Statement coverage instrumented, the Coverage Analysis Editor displays boxes on the left for statement outcomes that are uncovered. To mark a statement or condition as "considered covered", select the check box. Lines covered by analysis are displayed in blue.



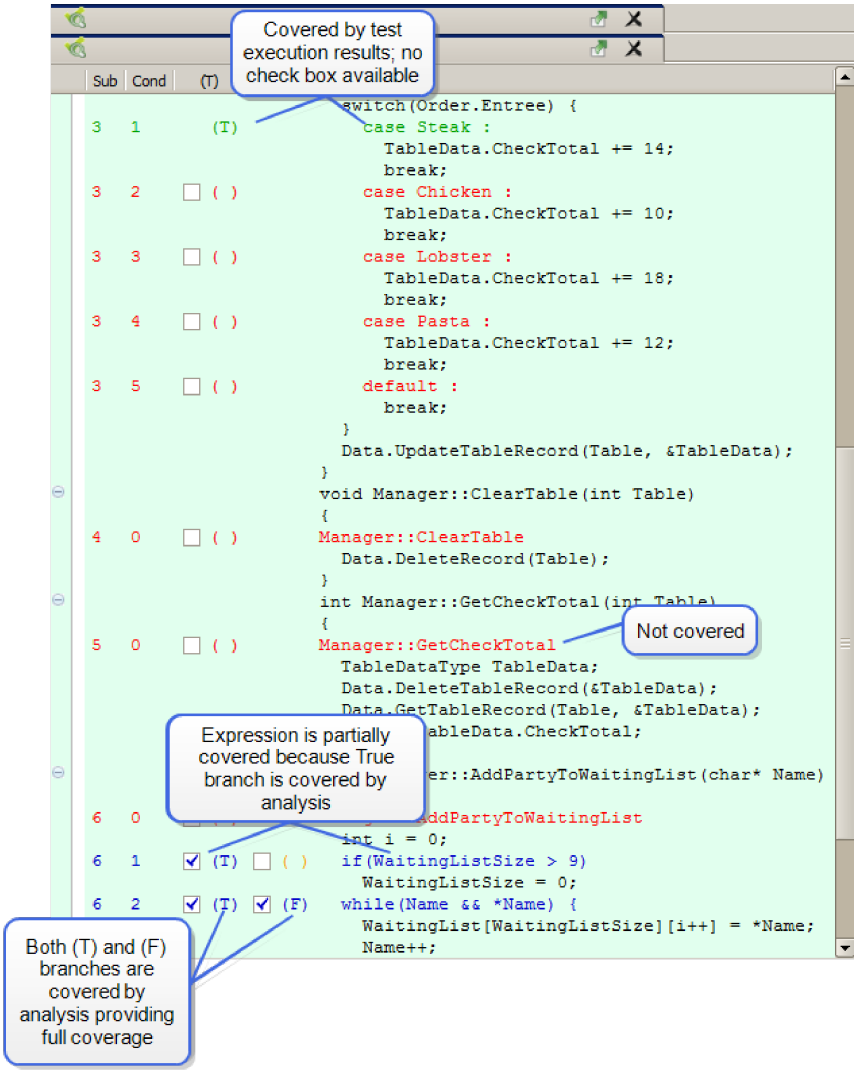
In the Coverage Analysis Editor, a check box does not appear next to a line if it is already covered by a test result. Regular test results, if they are present in the environment, take precedence over CBA results.

Branch Coverage

When Branch coverage is instrumented in the environment, the Coverage Analysis Editor displays each subprogram with a True branch (T) for the entry result, and True and False branches (T) (F) for each expression.

To mark a condition as having the True branch covered, select the check box in the (T) column. The "(T)" is displayed in blue. The branch is now partially covered because the True branch is Covered By Analysis, and the False branch is not covered.

To mark an expression as having the False branch covered, select the check box in the (F) column in the Coverage Analysis Editor. The "(F)" is displayed in blue.

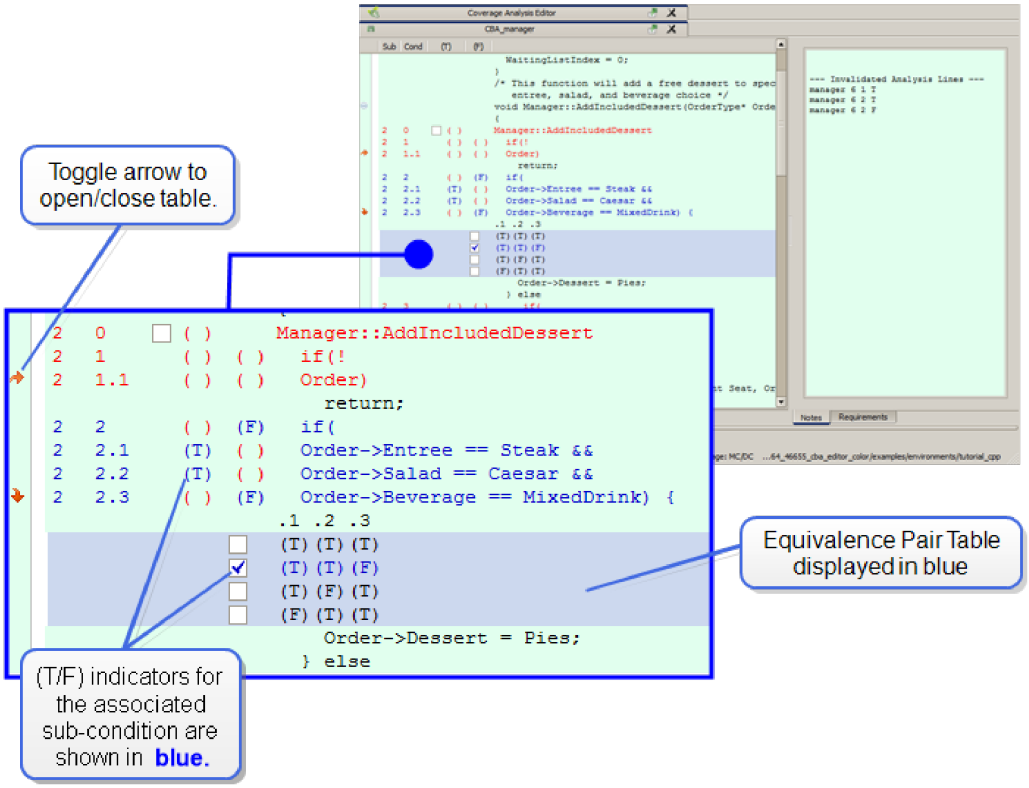


As with Statement coverage, if either or both of the True and False branches is already covered by regular test results, then the check box is not available in the Coverage Analysis Editor, and the expression shows yellow if partially covered, or green if already fully covered.

MC/DC Coverage

To add coverage analysis for a condition with multiple sub-conditions when MC/DC Coverage is instrumented, you annotate that one or more rows of the Equivalence Pairs table is covered.

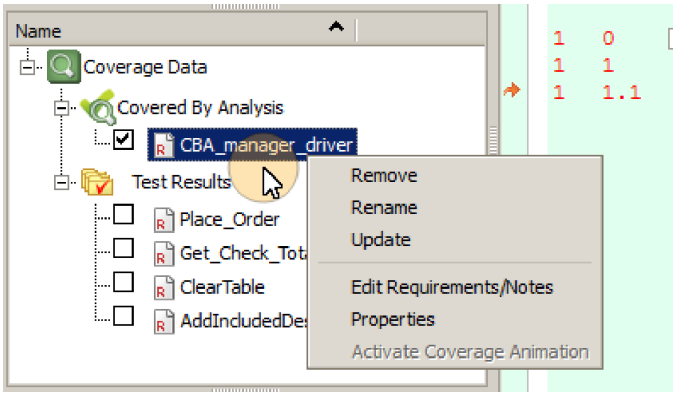
To access the equivalence pair table, click the arrow to the left of the condition. The truth table opens and a check box is displayed for each row. When a checkbox is selected, the associated sub-condition is considered "covered" and displayed in blue.



To Edit an Existing Analysis Result

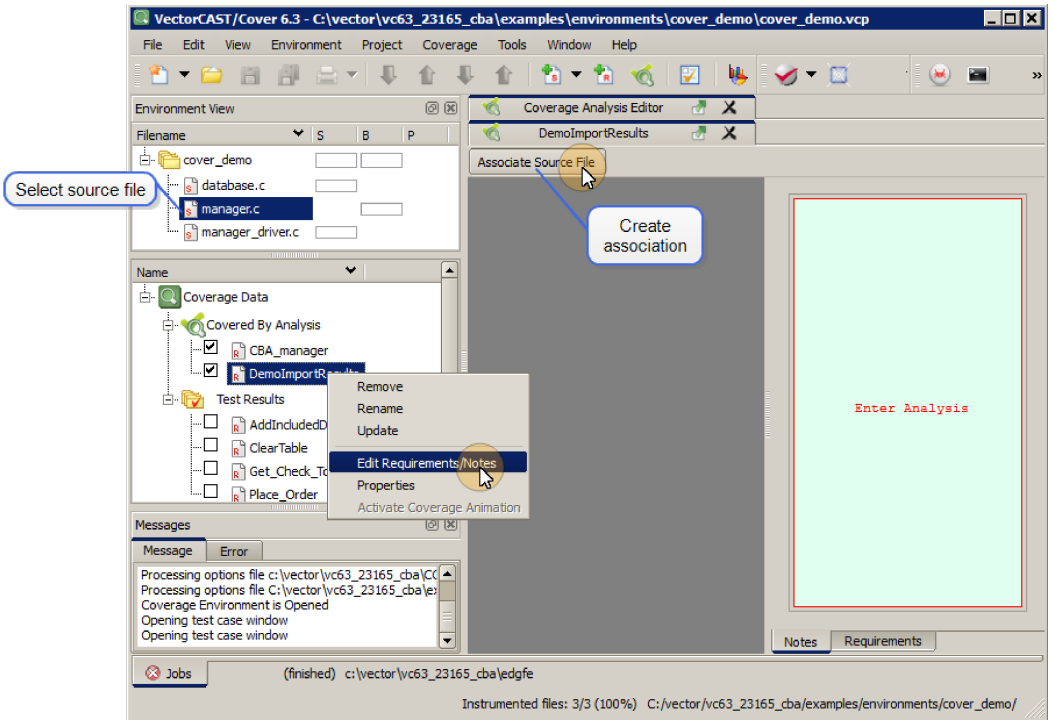
Any existing Analysis results can be edited, regardless of how they were created. This includes Analysis results that are imported via `.cvr` and also after rebuilding an environment.

Double-click an existing Analysis result in the Test Results pane to open the Coverage Analysis Editor. Alternatively, right-click on the Analysis result and select **Edit Requirements/Notes** from the pop-up menu. The contextual menu allows you to Remove, Rename, Update and view the associated properties of the Analysis result. Removing an Analysis result deletes its data, notes, and requirements from the environment.



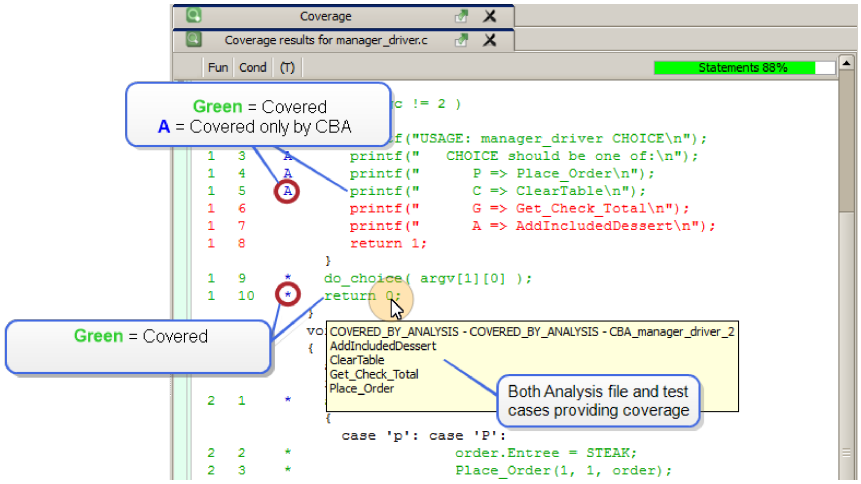
Associating an Analysis Result With Source File

Sometimes an Analysis result loses its association with the unit, as in the case of renaming the Analysis result. If this occurs, you can associate a source file with an imported analysis file. Open the editor by either double-clicking on the imported analysis file name in the CBA node, or right-clicking and selecting **Edit Requirements/Notes** from the menu. Next, select a file from the project and click the **Associate Source File** button to create an association. The associated source file will open in the editor.



Viewing CBA Coverage in the Coverage Viewer

CBA coverage is displayed in blue in the Coverage Viewer. When viewing CBA lines in the Coverage Viewer, hover over a CBA line to see the associated CBA notes.

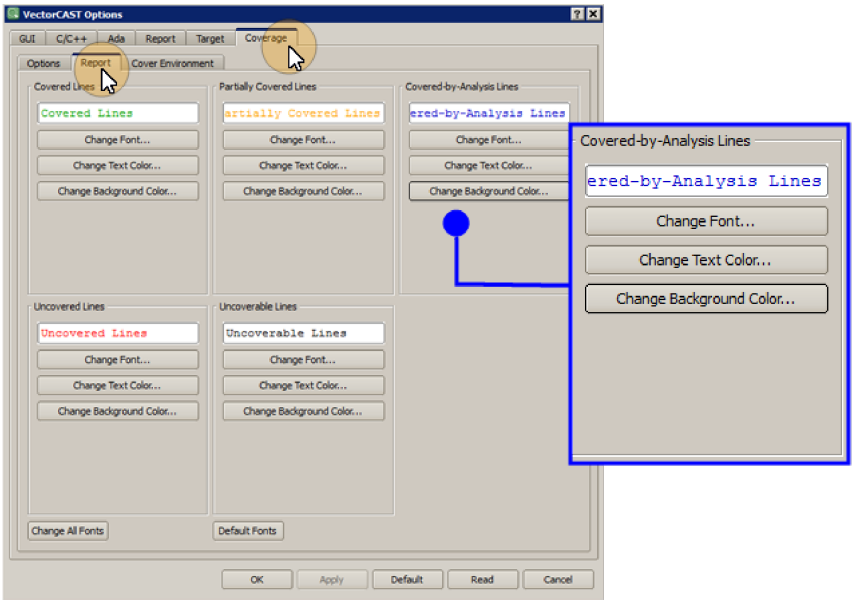


Lines covered only by CBA results are displayed in green (indicating the line is covered) with a blue "A" (indicating the line is covered *only* by CBA).

Lines covered by both regular execution results *and* CBA results are displayed in green (indicating the line is covered) and with a blue asterisk "*" (for statement) or a blue "T" or "F" (for branch), which indicates that it is *also* covered by CBA.

To Change Covered-By-Analysis Display Color

The default Covered-By-Analysis display color is blue. To change the color select **Tools => Options** from the Menu bar. On the **Tools => Options** dialog, select the **Coverage** tab and the **Report** sub-tab.



Use the buttons provided in the Covered-By-Analysis group box to change the font, text color and background color. When changes are complete, select the **Apply** button to save the changes to the .CFG file.

Working With Analysis Files

To Import Analysis Files

Importing Analysis Files allows the creation of analysis files from other tools. To import CBA data from a text file, select **Coverage => Import Coverage Analysis** from the Menu bar. An Open File dialog is displayed allowing you to select the Analysis file to be imported. By default, the file extensions "**.dat**" and "**.cba**" are selected as the two supported file types. The syntax for .cba files is: `unit name:<source code line number>`. Imported CBA Analysis results cannot be edited.

To Export Analysis Files

To export CBA data, select **Coverage => Export Script => Analysis Results...** from the Menu bar. A Save As dialog is displayed allowing you to name the coverage file to be created. By default, the file extension "**.cvr**" is selected as the supported file type.

```
clicast -e <env> Cover EXPORT_CBA <scriptfile>

Generate a script containing Coverage By Analysis results.
```

To Remove Analysis Files

To remove CBA data files, right-click on the Analysis result and select **Remove** from the context menu. The CBA coverage data, Notes, and Requirements are removed.

Alternatively, from the Menu Bar select **Coverage => Delete Coverage Analysis**. This menu item removes CBA Analysis results and their Notes and Requirements from the environment.

Re-Instrumenting With CBA Data

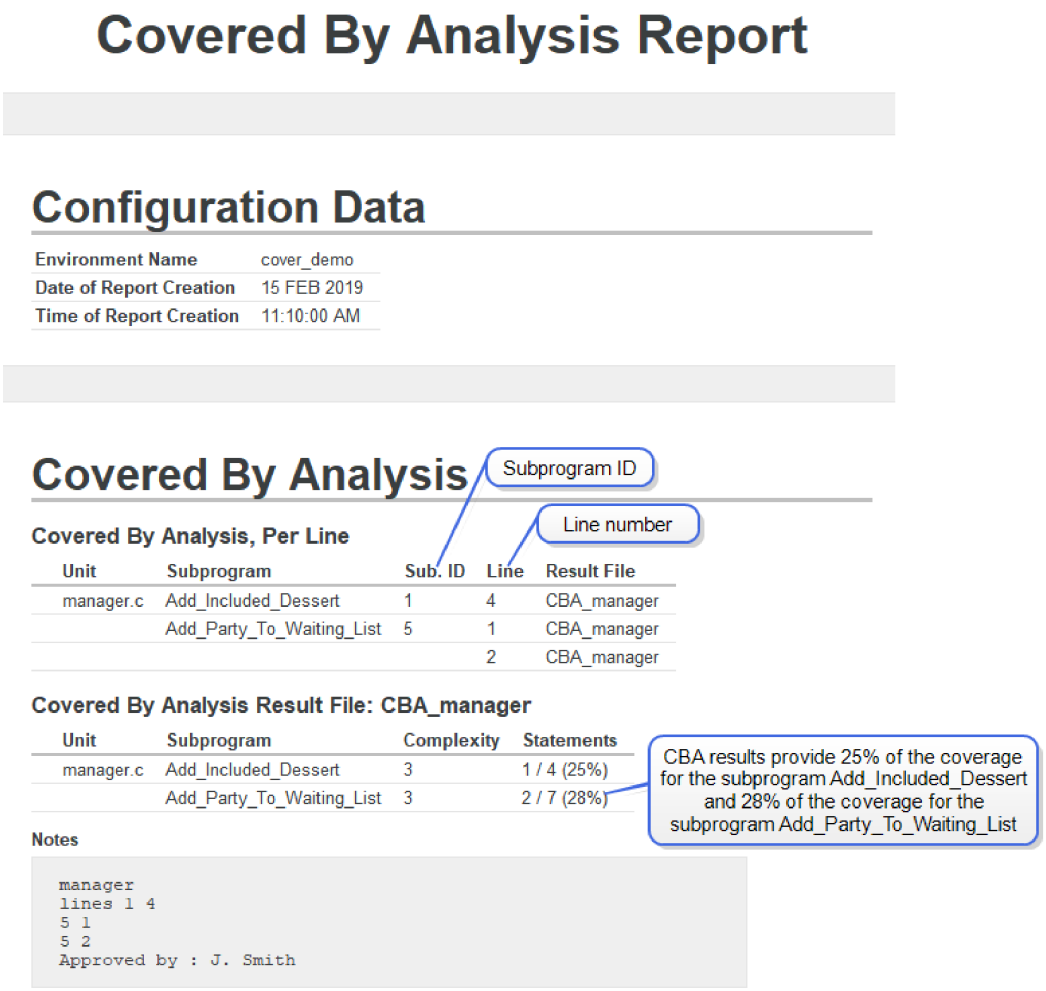
When source files in a Cover environment are un-instrumented or have their coverage type changed, the CBA data is invalidated. However, the Notes and Requirements for CBA are automatically retained and include information on the data that was invalidated. VectorCAST writes information about the deleted data in the Notes tab, appending to any Notes that were already there. When you see a CBA Analysis result with the icon indicating it is empty , double-click the icon to view the information in the Notes.

Viewing Analysis Data in Reports

Covered By Analysis Report

The Covered By Analysis (CBA) Report provides an overview of the Analysis data, including the total number of lines or conditions covered and any associated notes and requirements. The CBA Report includes only CBA results and disregards any regular test case results which normally take precedence over CBA results.

To access the Covered By Analysis Report, select **Environment => View => Covered By Analysis Report** from the Menu Bar.



Covered By Analysis Report

Configuration Data

Environment Name	cover_demo
Date of Report Creation	15 FEB 2019
Time of Report Creation	11:10:00 AM

Covered By Analysis

Subprogram ID

Line number

Covered By Analysis, Per Line

Unit	Subprogram	Sub. ID	Line	Result File
manager.c	Add_Included_Dessert	1	4	CBA_manager
	Add_Party_To_Waiting_List	5	1	CBA_manager
			2	CBA_manager

Covered By Analysis Result File: CBA_manager

Unit	Subprogram	Complexity	Statements
manager.c	Add_Included_Dessert	3	1 / 4 (25%)
	Add_Party_To_Waiting_List	3	2 / 7 (28%)

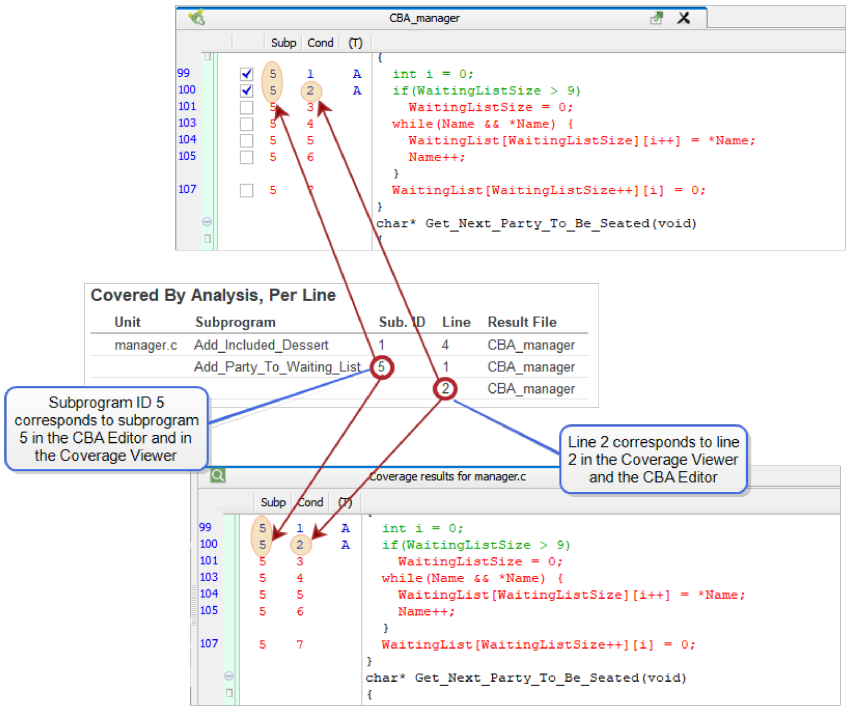
CBA results provide 25% of the coverage for the subprogram Add_Included_Dessert and 28% of the coverage for the subprogram Add_Party_To_Waiting_List

Notes

```
manager
lines 1 4
5 1
5 2
Approved by : J. Smith
```

The CBA Report has two sections: The **Covered By Analysis, Per Line** section and the **Covered By Analysis Result File** section(s).

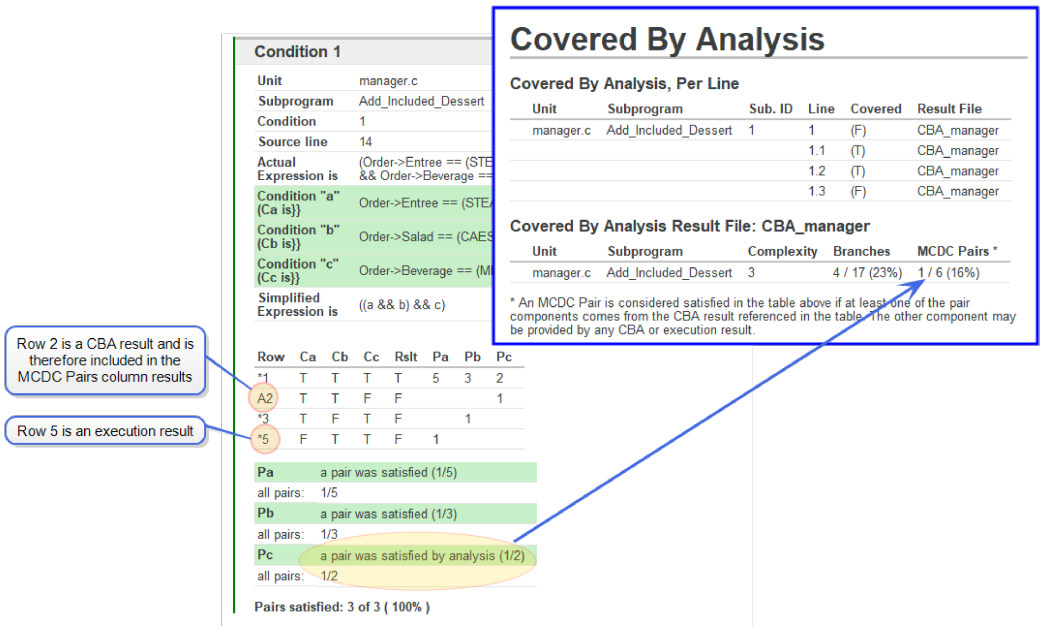
The **Covered By Analysis, Per Line** section lists each covered line in the unit, and identifies which CBA result file covers that line. The Subprogram ID corresponds to the left-hand number in the Coverage Viewer and CBA Editor. The Line number corresponds to the right-hand number in the Coverage Viewer and CBA Editor when the coverage type is Statement. The Line number represents the branch or decision number when the coverage type is other than Statement. There may be more than one CBA result covering a line. If the CBA result covers the True (T) outcome of a branch or decision, "(T)" is displayed in the line column. Similarly, if it covers the False (F) outcome, "(F)" is displayed.



The **Covered By Analysis Result File** section includes one table for each CBA result file. The table is similar to the Metrics Report in that it shows the number of statements and/or branches covered by that CBA result. For Statement and Branch coverage, only the number of lines or conditions that are Covered by Analysis are shown.

For MC/DC or Level A coverage, the number of conditions covered by CBA are shown. However, for the MCDC Pairs column, both CBA results and execution results are considered. An MC/DC Pair is considered satisfied if at least one of the pair components (or row) is a CBA result. The remaining component may be either a CBA result or an execution result.

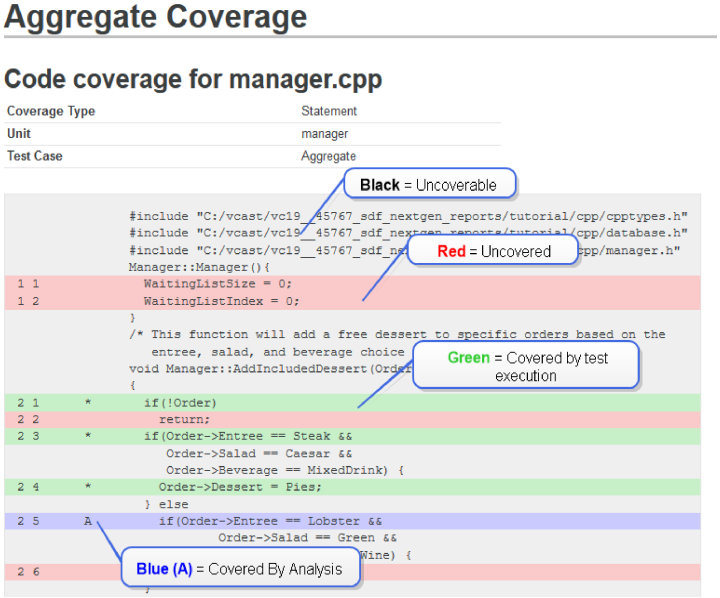
In the example below, one pair is covered. One component is covered by the CBA result file Add_ Included_Dessert (row 1). The other component (row 3) is covered by an execution result.




```
clicast -lc -e <env> Cover REPort CBA

Covered By Analysis Report.
```

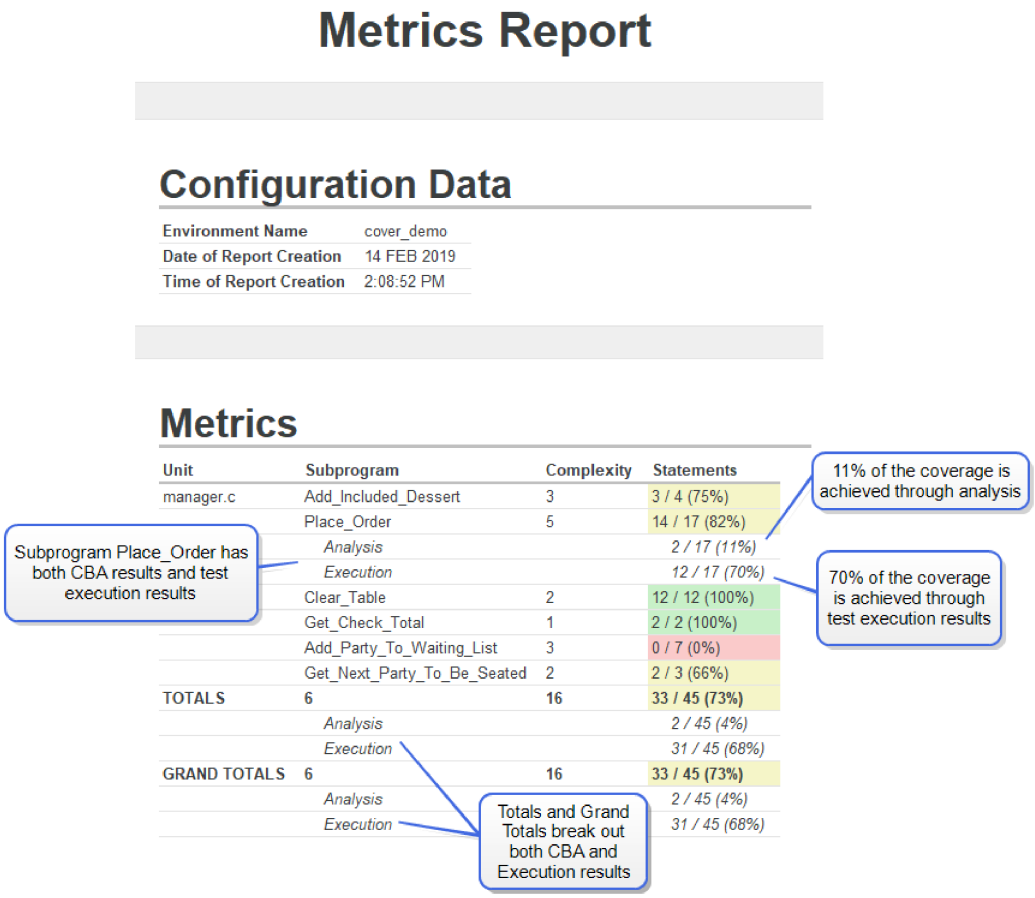
Aggregate Coverage Report

In the Aggregate Coverage Report, items that are only Covered by Analysis are indicated by the character 'A' and a blue background. Otherwise, the execution results are displayed using a green background to indicate covered by execution. Red background indicates uncovered.



Metrics Report

In the Metrics Report, when CBA results are present in the environment, they are displayed in italics in the row below the subprogram and the coverage achieved by test execution is displayed below the CBA results.



Metrics Report

Configuration Data

Environment Name	cover_demo
Date of Report Creation	14 FEB 2019
Time of Report Creation	2:08:52 PM

Metrics

Unit	Subprogram	Complexity	Statements
manager.c	Add_Included_Dessert	3	3 / 4 (75%)
	Place_Order	5	14 / 17 (82%)
	Analysis		*2 / 17 (11%)*
	Execution		*12 / 17 (70%)*
	Clear_Table	2	12 / 12 (100%)
	Get_Check_Total	1	2 / 2 (100%)
	Add_Party_To_Waiting_List	3	0 / 7 (0%)
	Get_Next_Party_To_Be_Seated	2	2 / 3 (66%)
TOTALS	6	16	33 / 45 (73%)
	Analysis		*2 / 45 (4%)*
	Execution		*31 / 45 (68%)*
GRAND TOTALS	6	16	33 / 45 (73%)
	Analysis		*2 / 45 (4%)*
	Execution		*31 / 45 (68%)*

Using Coverage Analysis With SFP

The environment must first be instrumented for coverage. CBA Analysis results in SFP are not associated with source files; therefore, a single result can be used for many source files.

Steps to change the coverage view mode from Translation Unit (TU) to Source File (SFP):

1. Select **Tools > Options …**
2. Select the **Coverage** Tab.
3. In the **Options** sub-tab, select **Source File** in the Coverage perspective area.
4. Select **OK** (any open coverage viewers will automatically closed).
5. Open the coverage viewer for a file by selecting the file, right-clicking, and choosing **Open Coverage Viewer**.

To Add Coverage Analysis

> Right-click source and choose **Add Coverage Analysis** or click Toolbar button  .

> User is prompted to choose a CBA Result or enter a new result.

>> **Note**: "Active CBA Result"

>>> The **CBA** tab must be current to add coverage analysis.

>>> The Active CBA result will store all coverage analysis.

>>> The Active CBA result can be switched.

> Coverage viewer opens with the **CBA** tab set current.

> Uncovered coverable items show empty checkboxes.

>> **Note**: Covered items cannot be marked as covered by analysis.

>> Check the box(es) to mark as covered by analysis.

>> Enter a justification in the **Notes** pane.

>> Click **Save**.

>>> Coverage now reflects the CBA:

- In the Environment view
- In the source view of the Source Coverage Viewer
- In the line details
- In the reports

>> Covered items by analysis are marked with an "A".

>> Source lines with analysis coverage are colored Blue.

To Add or Remove Coverage Analysis for Multiple Lines

> Select one or more source lines in the source view.

> Right-click and choose **Add Coverage Analysis** or **Remove Coverage Analysis**.

> You will notice the checkboxes in the Coverable section change.

> Enter a justification in the **Notes**.

> Click **Save**.

>> Coverage now reflects the CBA.

> **Notes**:

>> These menu items are only enabled when the **CBA** tab is current.

>> Adding/Removing coverage analysis is only done for the Active CBA Result.

To Add or Remove Coverage Analysis for MC/DC

> Adding coverage analysis is done on the Rows in the Equivalence Pairs Table in the bottom **MC/DC** pane.

>> Uncovered pair rows display an empty checkbox under the Covered column.

>> To add coverage analysis for a condition in an MC/DC expression:

>>> Select the condition in the Coverable section. The MC/DC table now shows rows only for that condition.

>>> Check the empty checkbox(es) under the Covered column for the appropriate rows.

>>> Enter a justification in the **Notes**.

>>> **Save**.

>>> Coverage now reflects the CBA

- The pair coverage for this condition will now show an "A" for "Analysis".

To Edit an Existing Result

> If you do not already know which files have CBA results, then identify the source files with analysis for the result by:

>> Generating the CBA report to identify which files to edit.

>> Unchecking all results except for the one you want to edit, and then inspecting Environment view for source files with coverage.

> Open the coverage viewer for the source files.

> To edit existing annotations, go to the source lines colored blue in the source view.

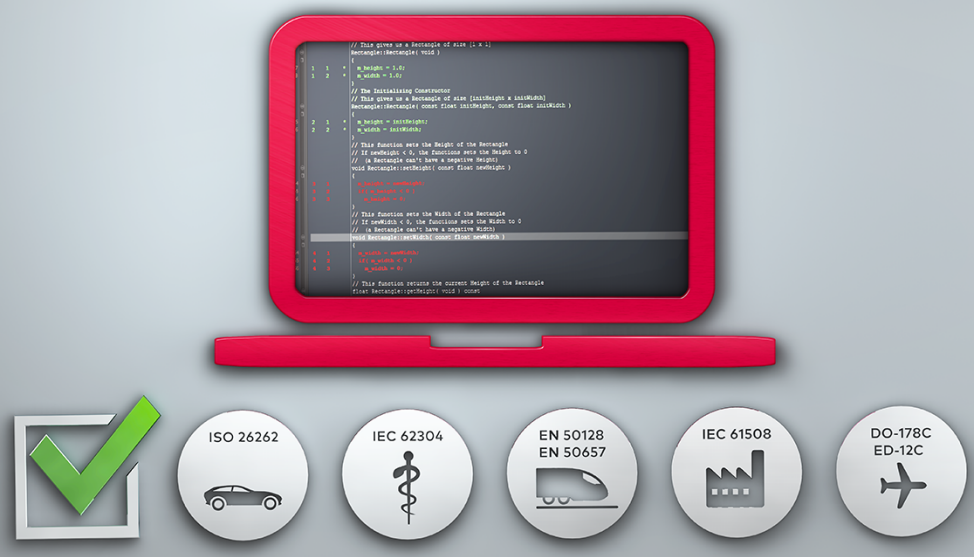
Changes for the Covered By Analysis Report

> The line number is used for SFP (instead of the subprogram ID).

> For MC/DC, the Decision ID and Condition ID are used in SFP.

The Aggregate Report

In the Aggregate Coverage Report, items that are only Covered by Analysis are indicated by the character 'A' and a blue background. To see the coverage information, click the ‘+’ to the left of the line number.



Using VectorCAST Probe Points

Probe Points

VectorCAST Probe Points allow the user to insert user-defined blocks of code, or probe points, before or after any executable statement. Probe points can be inserted during unit, API, or system testing and are created and maintained on a per-unit basis. The probe points are maintained as the source code changes, unless the coverage type changes or the source code changes such that the probe point no longer references the same coverable line. In that case, the probe point is dropped and the user is notified in the Message window.

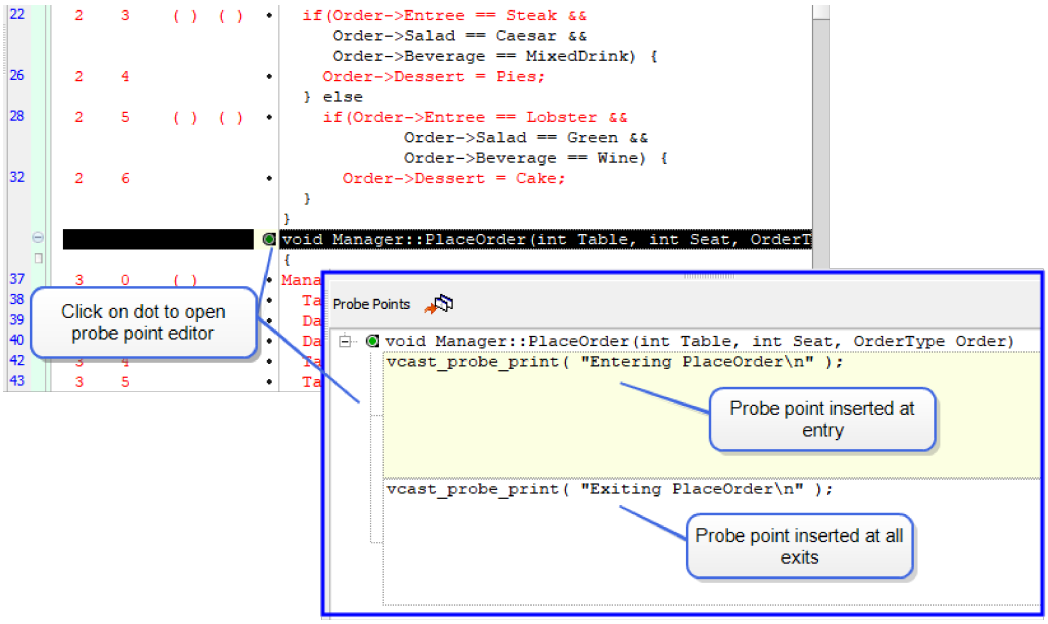
Probe points are frequently used in the following testing scenarios:

- Capturing local variables during program execution
- Injecting spurious values to allow testing of error handling code
- Patching faulty code to test a fix prior to committing the change
- Debugging hard-to-trigger race conditions
- Recording detailed control flow
- Dynamically instrumenting device software to isolate defects

Function Probe Points

Function probe points are associated with a function and are inserted either at the entry point of a function (inserted after the first executable line) or at the end of a function (inserted right before each exit point).

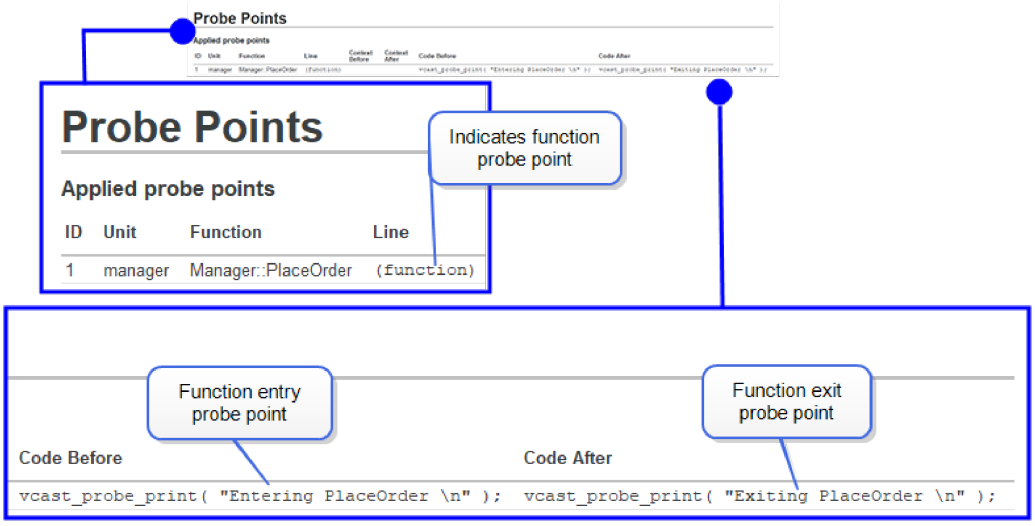
Entry and Exit function probe points can be entered via the Probe Point Editor. Small black dots ● in the left-hand column of the Coverage Viewer indicate a possible probe point location. To insert a probe point, click on the black dot next to the executable line where you want to insert the probe point. The dot will change to a green circle ◉ indicating an active probe point and a new node will be added to the Probe Points editing pane on the right with text edit boxes activated under the node.



The Probe Point Editor allows you to enter the source code for the probe point. To insert and execute a function entry probe point, click on the top text edit box to activate the editor widget and enter the code to be inserted upon entering the function.

To insert and execute a function exit probe point, click on the bottom edit box to activate the editor widget and enter the code to be inserted at all exit points of the function.

In the Probe Point Report, function probe points are indicated by listing **`(function)`** in the **Line** column. Function entry probe points are listed under the **Code Before** column. Function exit probe points are listed under the **Code After** column.



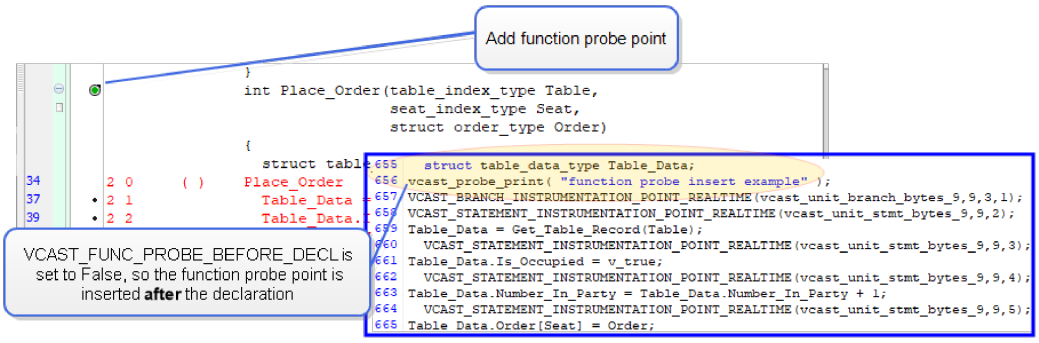
Function Entry Probe Points

Function entry probe points are inserted just inside the opening curly brace in C++, or after the declarative section in C. This clicast-only option is provided to instruct VectorCAST to insert the function probe point before the declarative section for C sources.

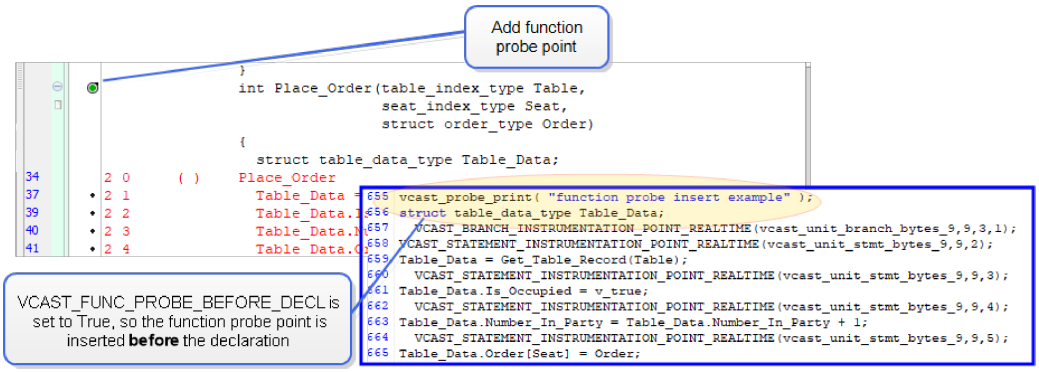
```
clicast -lc option VCAST_FUNC_PROBE_BEFORE_DECL True | False

Set this option to True to force the insertion of function based probe
points before the declarative region of the function in C. The default
value is False.
```

The default value for this option is False, meaning insert the function probe point in the default location, after the declarative section. The environment must be re-instrumented for a change to this option to take effect.

In our example below, we insert a function probe point for the Place_Order function. By default, the probe point is inserted after the declarative section as shown.



When the option is set to True, the function probe point is inserted before the declaration.



In environment scripts (**.env, .enc**) and probe point files (**.pp**), function entry probe points are specified in the usual manner with the addition of the following line to identify the probe point as an function entry probe point:

PROBE_LOCATION: FUNCTION

For example:

```
PROBE_LOCATION: FUNCTION
PROBE_FUNCTION: my_function
PROBE_CODE:
vcast_probe_print("*** BEGINNING OF FUNCTION ***\n");
END_PROBE_CODE:
```

Function Exit Probe Points

Function Exit probe points are inserted right before each exit point. Specifically, if the function is void (), then the probe point is executed at the end of execution. If the function has one or more return() calls, then the probe point executes before the return() call and therefore before any function that might be called in the return() function. A Function Exit probe point is useful for calling VCAST_DUMP_COVERAGE_DATA() in a Cover environment, for example.

In environment scripts (**.env, .enc**) and probe point files (**.pp**), function exit probe points are specified in the same manner as a function entry probe point, with the addition of the following lines :

PROBE_CODE_AFTER

END_PROBE_CODE_AFTER

For example:

```
PROBE_LOCATION: FUNCTION
PROBE_FUNCTION: my_function
PROBE_CODE_AFTER:
vcast_probe_print("*** END OF FUNCTION ***\n");
END_PROBE_CODE_AFTER:
```

Probe Point Order of Execution

A function probe point generally executes prior to any other probe points in the function. However, there are two special cases:

- When using a Branch or Statement+Branch coverage with a regular probe point on the function's entry point (C++ only).
- When using a regular probe point on a declaration that is executable.

For C++ source code, the order of execution is:

1. Entry point probe point (only with Branch or Statement+Branch coverage)
2. Function Entry probe point
3. Probe point "before" an executable line in the declarative section
4. Probe point "after" an executable line in the declarative section
5. Probe point "before" first executable line
6. Probe point "after" first executable line

For C source code with the option VCAST_FUNC_PROBE_BEFORE_DECL set to False (default), the order of execution is:

1. Probe point "before" an executable line in the declarative section
2. Probe point "after" an executable line in the declarative section
3. Function Entry probe point
4. Probe point "before" first executable line
5. Probe point "after first executable line

For C source code with the option VCAST_FUNC_PROBE_BEFORE_DECL set to True, the order of execution is:

1. Function Entry probe point
2. Probe point "before" and executable line in the declarative section
3. Probe point "after" an executable line in the declarative section
4. Probe point "before" first executable line
5. Probe point "after" first executable line

Using the Probe Point Editor

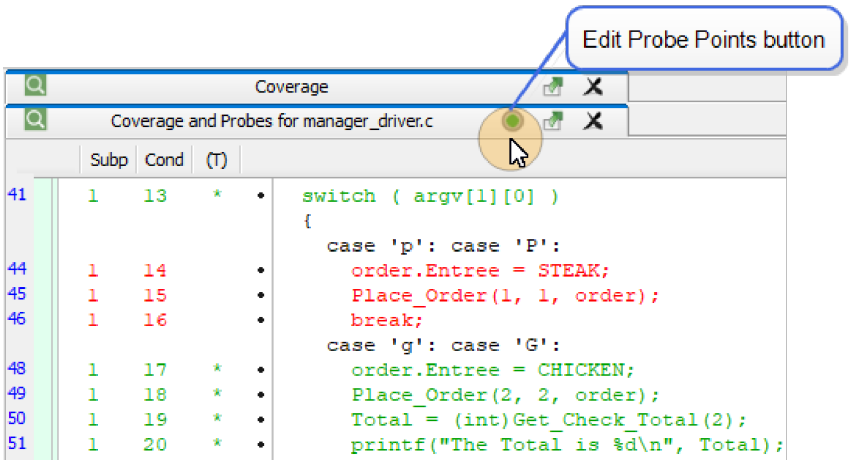
Open the Probe Point Editor

To insert probe points, the environment must first be instrumented for coverage. Available coverage types are Statement, Branch, Basis Paths, MC/DC, Statement+Branch, Statement+MC/DC, and Probe Point. There are multiple ways to open the Probe Point Editor:

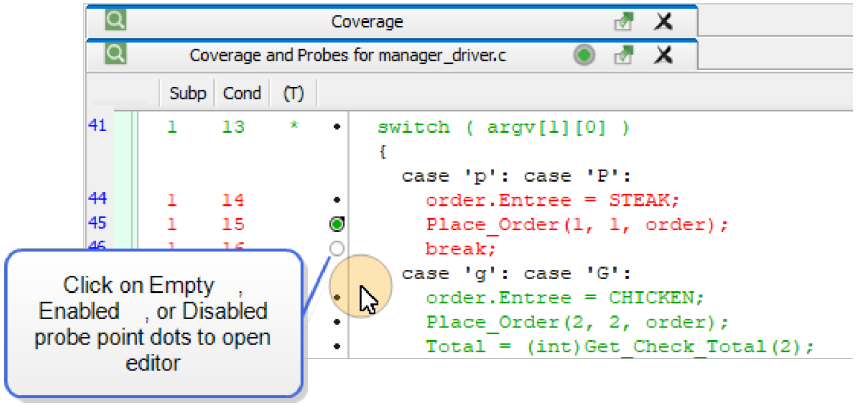
> Select an instrumented source file in the Source Files pane and click the **Edit Probe Points** button on the Toolbar.

> Right-click the instrumented source file in the Source Files pane and select **Edit Probe Points** from the context menu.

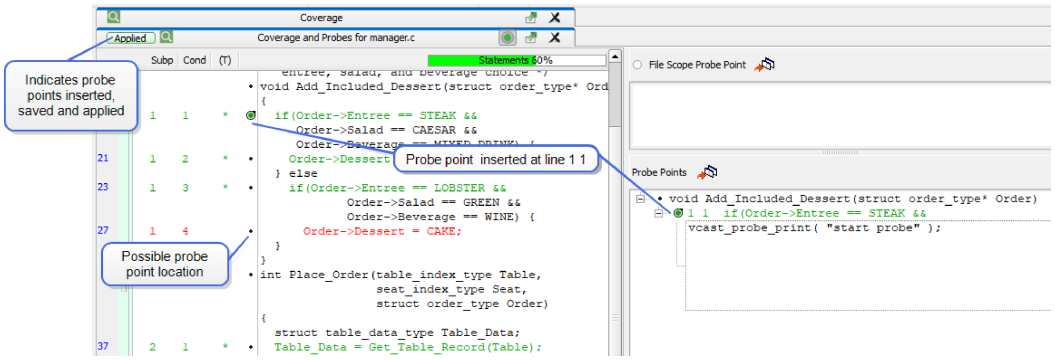
> In the Coverage Viewer, click the **Edit Probe Points** button located on the tab.



> In the Coverage Viewer, click on any Probe Point Dot. Any of the states (Empty, Enabled, or Disabled) will open the Editor.



The Coverage Viewer opens in the MDI area in the left pane and displays the coverage and probe points. The Probe Point Editor opens in the right pane.




```
clicast -lc Option VCAST_ENABLE_PROBE_POINTS True | False

Set this option to False to disable the use of probe points. The default
value is True.
```

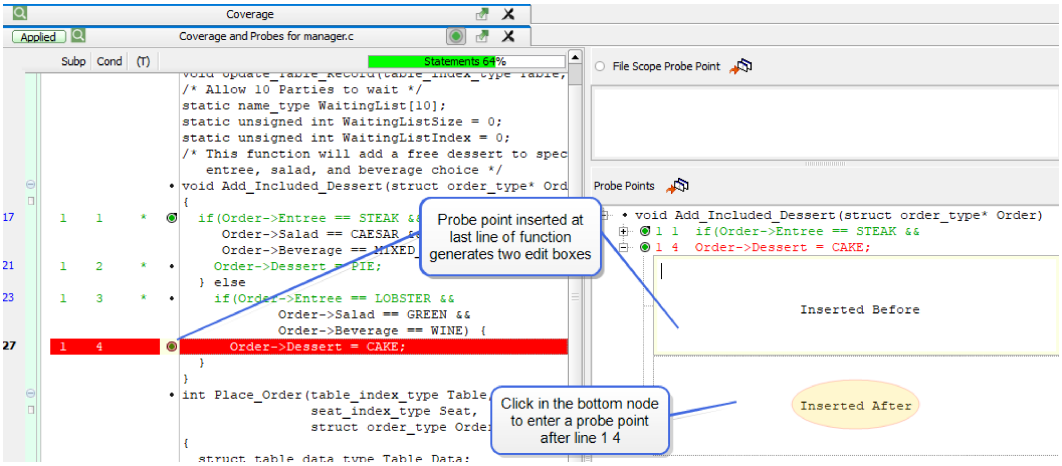
Insert a Probe Point

Probe points are associated with one unit. To open the Probe Point Editor and add a new probe point, select an instrumented source file in the Source Files pane and click the **Edit Probe Points** button on the Toolbar. Alternatively, you can right-click the UUT or subprogram in the Test Case Tree and select **Edit Probe Points** from the context menu.

Small black dots ● in the left-hand column of the Coverage View pane indicate a possible probe point location. To insert a probe point, click on the black dot next to the executable line where you want to insert the probe point. The dot will change to a green circle indicating an active probe point and a new node will be added to the Probe Points editing pane on the right with text edit boxes activated under the node.

Note: Probe points are not available for MC/DC sub conditions. Only coverable statements and branches can be probed.

The Probe Point Editor allows you to enter the source code for the probe point. To insert and execute a probe point *before* the line of code is executed, click on the top text edit box to activate the editor widget. To insert and execute a probe point *after* the line of code is executed, click on the bottom text edit box and activate the editor widget.



When Statement coverage is on, only the top text edit box is available for **`return()`** statements. When Branch coverage is on, no probe point can be set at the entry point to a function.

Insert a File Scope Probe Point

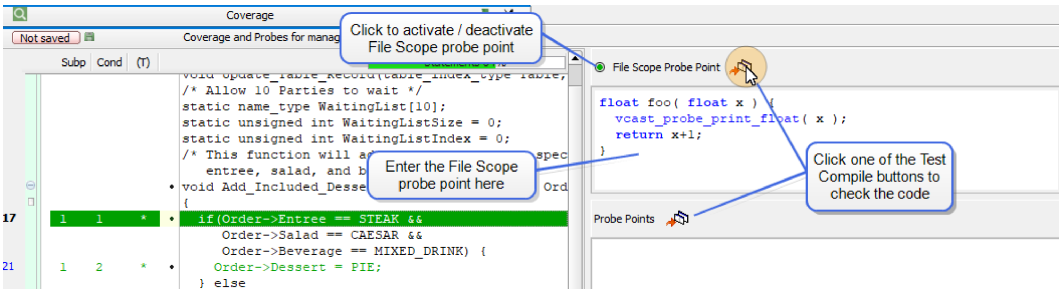
VectorCAST Probe Point supports adding a probe point at file scope. Only one File Scope probe point can be entered per unit. Any valid code is accepted, such as new function definitions or global variables.

To add a File Scope probe point, first instrument the unit or environment for coverage. In the example below, the environment is instrumented for Statement coverage.

Open the Probe Point Editor by selecting an instrumented source file in the Source Files pane and clicking the **Edit Probe Points** button on the Toolbar. Alternatively, you can right-click the UUT or subprogram in the Test Case Tree and select **Edit Probe Points** from the context menu.

The File Scope Probe Point Editor is located in the top right pane of the Probe Point Editor. Click within the editor to activate, and enter the File Scope probe point code.

Clicking on the probe point icon in the File Scope Probe Point Editor toggles the File Scope probe point between the active and inactive states. To deactivate the File Scope probe point, single-click on the active probe point icon . The inactive probe point icon is displayed in the File Scope Probe Point Editor.



Click one of the **Test Compile** buttons to test the code. Clicking the **Test Compile** button will test compile all active probe points (both File Scope and regular function scope). See "Test Compile a

Probe Point" on page 268 for more information.

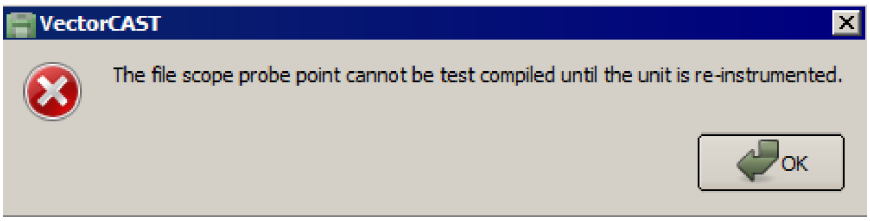
When you save and apply the probe points for the unit, the File Scope probe point is inserted in the instrumented source file just after the last #include line.

File Scope probe points are written to the regression scripts along with the regular function scope probe points.

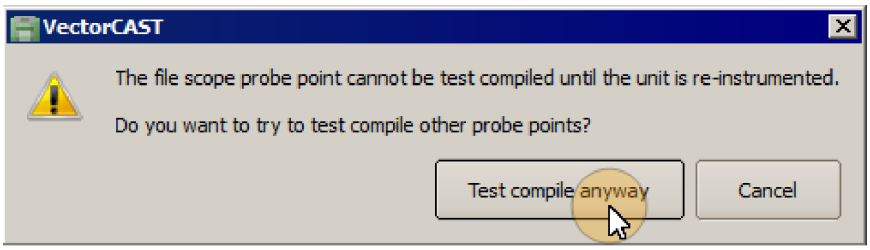
Using File Scope Probe Points with Older Environments

When using environments created with earlier versions of VectorCAST, the source file must be re-instrumented before compiling an environment containing a File Scope probe point.

If you attempt to test compile a unit with a File Scope probe point in an old environment, you will receive an error message notifying you to re-instrument the unit.



If the old environment contains regular probe points and you attempt to add a File Scope probe point, a warning message appears when you attempt to test compile a unit. You have the option to continue the test compile operation on the regular probe point or to abort the operation.



Select the **Test compile anyway** button to continue the operation. A compile error is generated if the regular probe point depends on the File Scope probe point. Otherwise, it compiles without error. In this case, a compile error is expected during the test compile because the File Scope probe point is not included in the test compile. Or select the **Cancel** button to abort the operation.

Test Compile a Probe Point

A **Test Compile** button  is located in the upper right of the Probe Point Editor and the File Scope Probe Point Editor. This button performs a test compile of all active probe points in the unit. Note that deactivated probe points are not compiled.

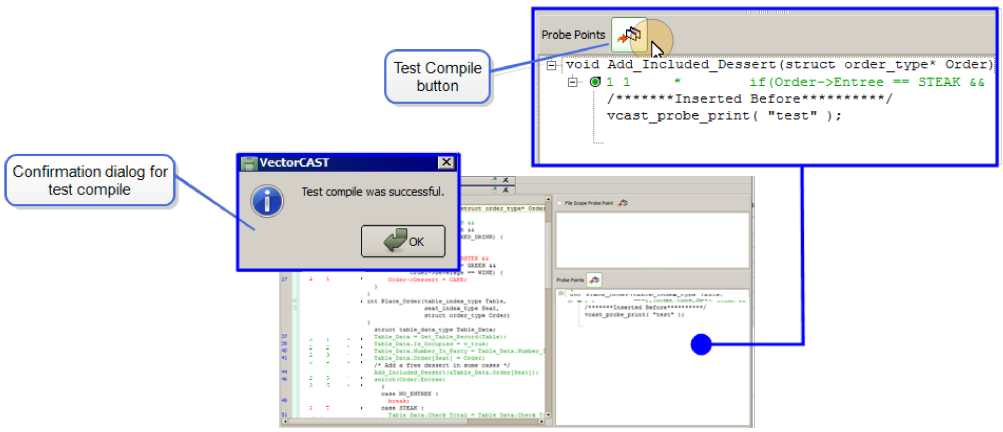
If you attempt to test compile a unit with a File Scope probe point in an old environment, you will receive an error message notifying you to re-instrument the unit. See "Using File Scope Probe Points

with Older Environments" on page 268.

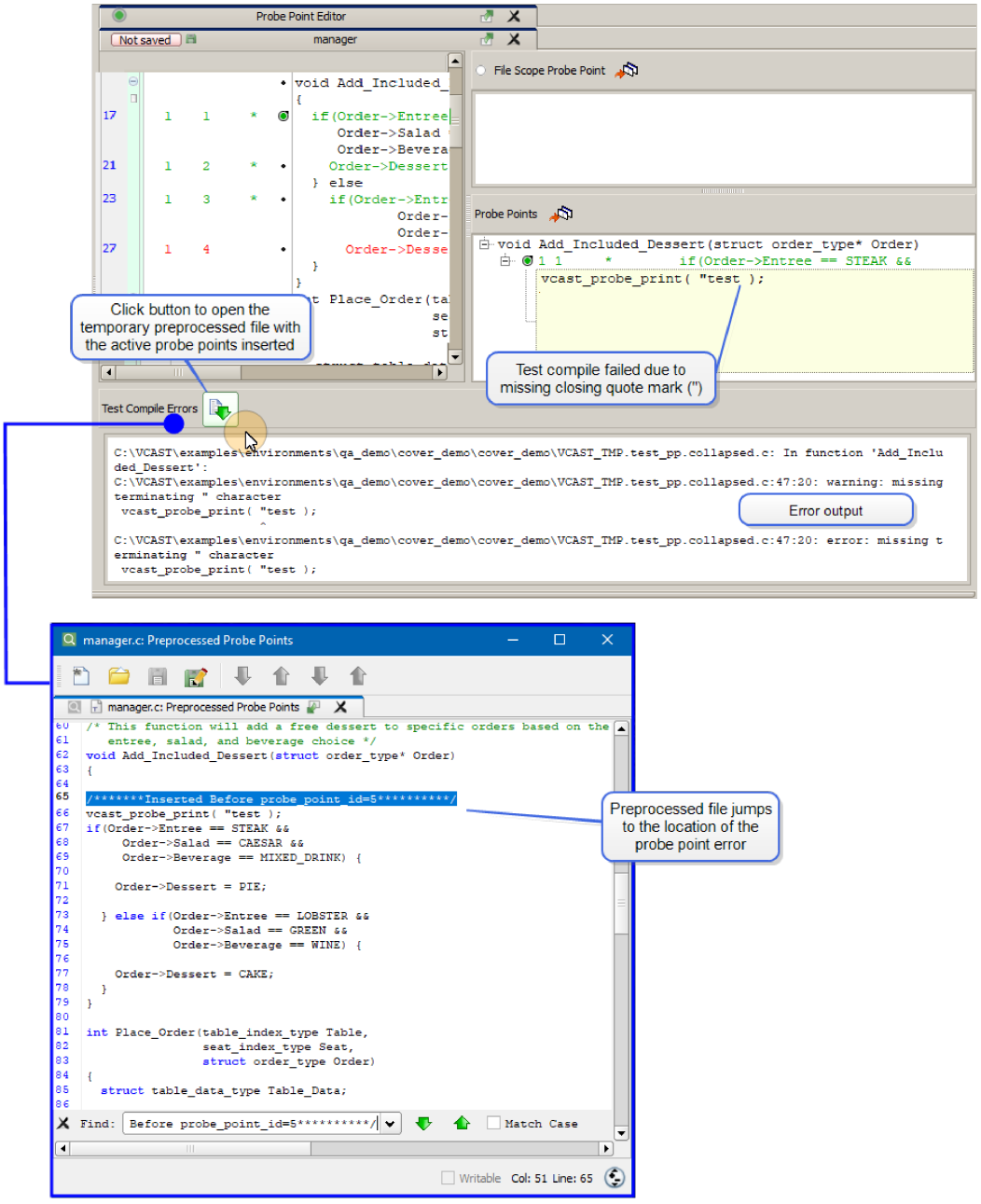
A test compile can be performed on a probe point in any status. It can be Not Saved, Not Applied, or Applied. See "Probe Point Status Buttons" on page 272 for more information.

The test compile process preprocesses the unit, inserts the probe points into the preprocessed file and compiles the file.

Click the **Test Compile** button to perform a test compile. Upon successful completion of the compile, a confirmation dialog is displayed.



Error output from this action is displayed at the bottom of the Probe Point Editor. Clicking the **File** button  located at the top of the Test Compile Errors pane opens the preprocessed file, enabling you to diagnose the compile error.



Saving and Applying Probe Points

When editing is complete, Save and Apply the probe point.

Edits and changes to probe points can be saved by doing any of the following:

- Clicking the **Not Saved** status button on the Probe Point Editor tab automatically saves and then applies the probe point by performing an incremental rebuild.
- Clicking the **Save** button on the Toolbar (which saves the changes in the Probe Point Editor with current focus).

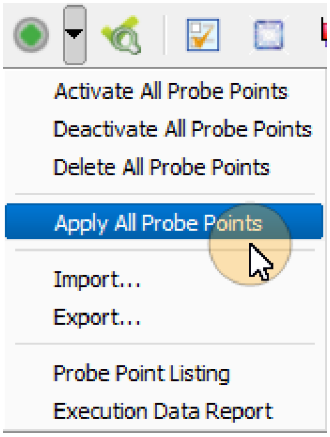
- Clicking the **Save All** button on the Toolbar (which saves all modified probe points in all units).
- Selecting **File => Save**.
- Closing the Probe Point Editor and selecting **Yes** in the confirm dialog.

Probe points can be applied to all units by selecting **Environment => Reinstrument All Source**. Once applied, the probe point text appears in the instrumented version of the UUT, in the environment directory.

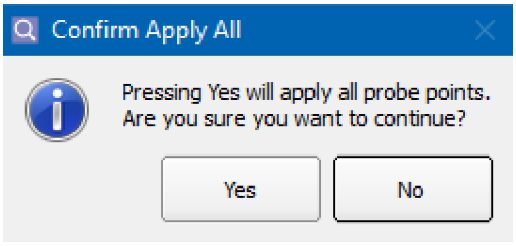
```
clicast -e <COVER ENV> [-u <unit>] Cover PROBE_point APPly

Applies the probe points. Causes the unit(s) with the probe point to be
reinstrumented.
```

Apply All Probe Points

VectorCAST provides the ability to apply all un-applied probe points in all units at the same time. From the Probe Point icon's drop-down menu, select **Apply All Probe Points** to save and apply the Probe Points in all modified units.

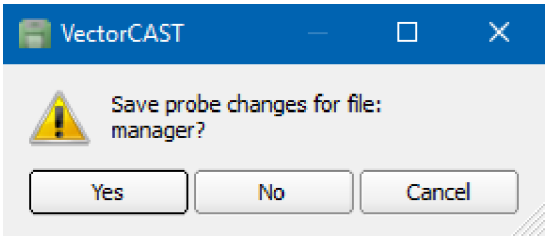


A confirmation dialog appears to confirm the action.



If **No** is selected, no units are applied and the action is aborted.

If **Yes** is selected, a confirmation dialog is shown for each modified unit, asking whether to save or not.



If **No** is selected, no changes are saved and the Probe Point Editor for the unit is closed.

If **Yes** is selected the changes are saved, the Probe Point Editor for the unit is closed, and the affected units are re-instrumented.

If **Cancel** is selected, no changes are saved and the Probe Point Editor remains open.

Probe Point Status Buttons

A status button and icon are shown on the left of the Probe Point Editor tab to indicate the status of existing probe points:

Empty	= Contains no probe points.
Not saved	= Probe point edits have been made and not yet saved. Clicking the button automatically saves and applies the probe points.
Not Applied	= Probe point edits have been saved but not inserted into the unit. Clicking the button applies the probe point and re-instruments the unit. Probe points can be applied to all units by selecting **Environment => Reinstrument All Source**.
Applied	= Probe points applied.

A right-click context menu is provided in the Coverage View pane allowing the user to quickly **Remove All Probe Points**, **Expand All Subprograms**, and **Collapse All** subprograms.

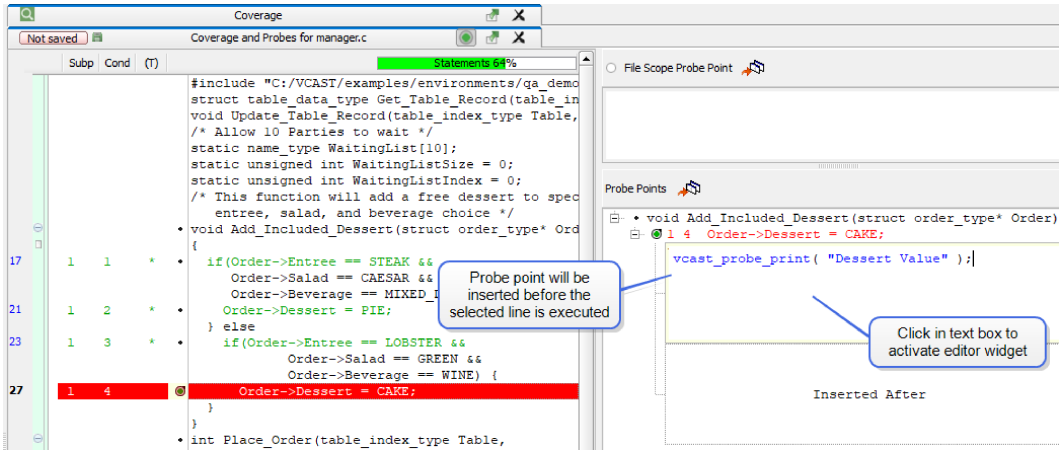
The keyboard shortcuts Select All (**Ctrl+A**) and Copy (**Ctrl+C**) are also available from the context menu in the Coverage View pane.

A similar right-click context menu is provided in the Probe Points editing pane allowing the user to **Expand All** nodes, **Collapse All** nodes or **Remove All Probe Points**.

Edit a Probe Point

To edit an existing probe point, right-click the source file in the Source File Tree and select **Edit Probe Points** from the context menu. Alternatively, select an instrumented source file in the Source Files pane and click the **Edit Probe Points** button on the Toolbar. The Probe Point Editor opens showing the nodes in the rightmost pane. Expand the probe point node to open the text editor and click to enter editing mode.

To insert and execute a probe point *before* the line of code is executed, click on the top text edit box to activate the editor widget. To insert and execute a probe point *after* the line of code is executed, click on the bottom text edit box and activate the editor widget.



When editing is complete, Save and Apply the probe point.

Built-in Functions and Macros

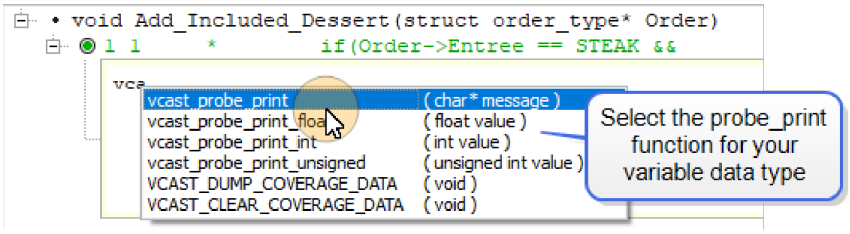
The Probe Point Editor provides built-in functions and macros for ease of use in creating probe points.

VectorCAST provides a replacement function for **`printf`** named **`vcast_probe_print()`**. This function, when included in a probe point that was reached during test execution, prints out the variables of string, integer, unsigned int, or floating point data types to the TESTINSS.DAT file for Cover environments.

Using **`vcast_probe_print()`** rather than **`printf`** ensures that data will be captured even when testing an embedded target.

Note that a warning is displayed whenever the **`printf`** command is entered in the editor. To turn off the warning, select the checkbox. To reinstate the warning at any time, from the Menu bar, select **View => Default Layout**.

The **`vcast_probe_print()`** functions will auto-complete when you begin typing "**`vcast_`**" into the Probe Point Editor. Select the appropriate function from the drop-down menu.



`vcast_probe_print();`

This function allows text output from the test to be captured to a file for inclusion in the Execution Results Report. Only character strings with double quotes are accepted.

`vcast_probe_print_float();`

This function allows floating point variable output from the test to be captured to

a file for inclusion in the Execution Results Report. Only floating point values are accepted.

```
vcast_probe_print_int();
```

This function allows integer output from the test to be captured to a file for inclusion in the Execution Results Report. Only integer values are accepted.

```
vcast_probe_print_unsigned();
```

This function allows unsigned int output from the test to be captured to a file for inclusion in the Execution Results Report. Only unsigned int values are accepted.

```
VCAST_DUMP_COVERAGE_DATA(void);
```

This function writes out the coverage data in the buffer to a file when the probe point is executed. The default file is TESTINSS.DAT.

Note that if the option "Enable the coverage clear API" (VCAST_ENABLE_DATA_CLEAR_API) is True, the user can also call VCAST_CLEAR_COVERAGE_DATA() in a probe point, which clears the buffer when the probe point is executed. Also, when the option "Dump buffered coverage data on exit" (VCAST_DUMP_COVERAGE_AT_EXIT) is True, any data remaining in the buffer is written to the TESTINSS.DAT file when the instrumented application finishes executing.

```
VCAST_CLEAR_COVERAGE_DATA(void);
```

This function clears collected coverage data for current execution when called. Requires VCAST_ENABLE_DATA_CLEAR_API to be defined at compile time.

Deactivate/Activate Probe Points

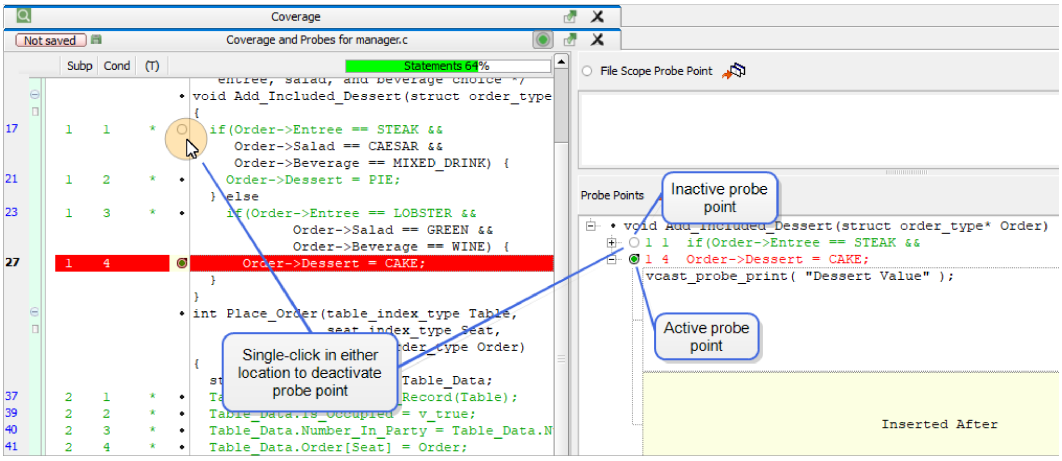
Clicking on the probe point icon in either the Coverage Viewer or the Probe Point Editor toggles the probe point between the active and inactive states. To deactivate an individual probe point, single-click on the active probe point icon . The inactive probe point icon is displayed in both the Coverage Viewer and the Probe Point Editor.

To deactivate all probe points in all units, select **Deactivate All Probe Points** from the drop-down menu next to the **Edit Probe Points** button on the toolbar. A Confirmation dialog appears to confirm that you wish to deactivate all probe points. Selecting the **No** button cancels the deactivation. Selecting the **Yes** button deactivates all probe points and performs an incremental rebuild of the environment.

Deactivated probe points cannot be edited. A probe point must be in an active state to edit.

To activate an individual probe point, single-click on the inactive probe point icon . The active probe point icon is displayed in both panes of the Editor.

To activate all probe points in all units, select **Activate All Probe Points** from the drop-down menu next to the **Edit Probe Points** button on the toolbar. A Confirmation dialog appears to confirm that you wish to activate all probe points. Selecting the **No** button cancels the activation. Selecting the **Yes** button activates all probe points and performs an incremental rebuild.



Remove Probe Points from Editor

To remove a single probe point from the Probe Point Editor, right click on the green probe point icon ◉ in either the Coverage Viewer or the Probe Point Editor and select **Remove Probe Point** from the context menu. Both active and inactive probe points can be removed. A Confirm Remove dialog appears, and selecting the **Yes** button removes the probe point and displays the available probe point icon ●.

To remove all active and inactive probe points from the unit, right-click within either the Coverage Viewer or the Probe Point Editor and select **Remove All Probe Points** from the context menu. A Confirm Remove dialog appears, and selecting the **Yes** button removes all probe points and displays the available probe point icon ●.

Note: Selecting "Remove" probe points temporarily removes them from the Probe Point Editor. "Removing" differs from "Deleting" in that it can be undone by closing the Editor and not saving. Probe points are not permanently removed (or "Deleted") until the file is saved.

Delete All Probe Points

To permanently delete all probe points, select **Delete All Probe Points** from the drop-down menu next to the Probe Point button on the toolbar.

Note: Selecting the **Delete All Probe Points** option permanently removes *all* probe points in *all* units in the environment, and cannot be undone.

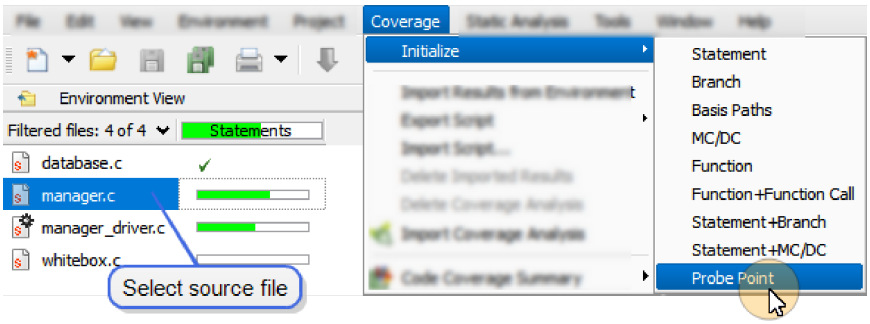
```
clicast -e <COVER ENV> Cover PROBE_point REMove_all

Remove all active and inactive probe points from the environment and
incrementally rebuild.
```

Probe Point-Only Instrumentation

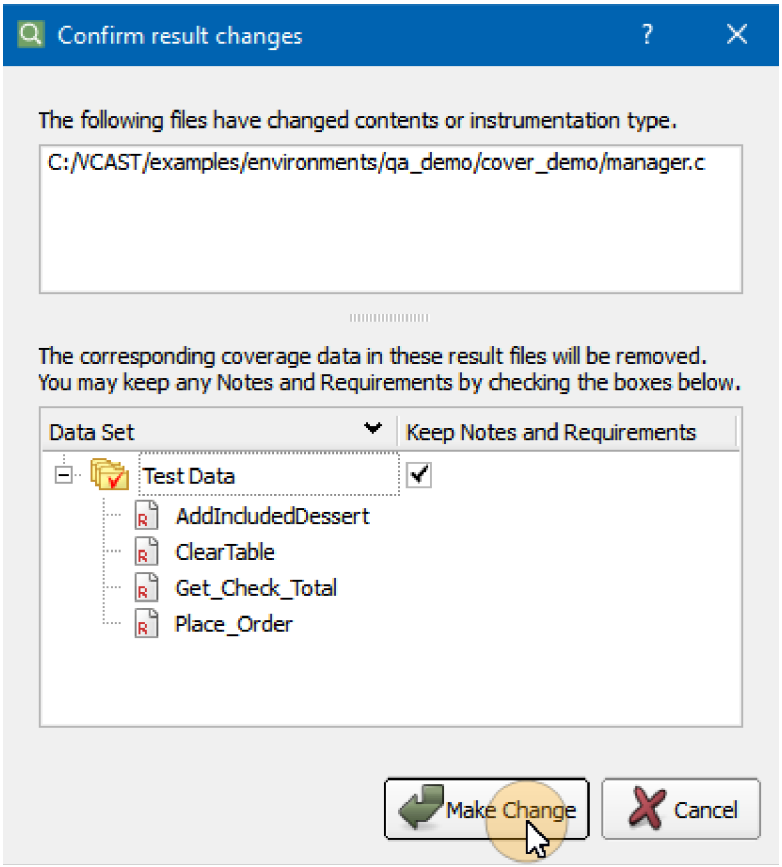
In Cover environments, VectorCAST Probe allows users to apply Probe Point-only instrumentation. This feature allows users to insert probe points without first instrumenting the environment for

coverage.

To initialize Probe Point-only instrumentation, first select a source file in the Source Files pane. From the Menu Bar, select **Coverage => Initialize => Probe Point**. Alternatively, right-click on the source file and select **Instrument => Probe Point** from the context menu.



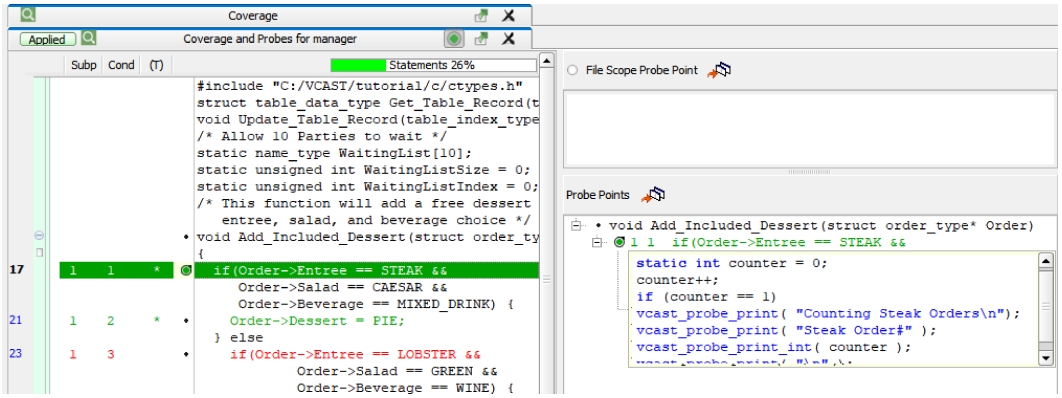
A Confirm Result Changes dialog may open, alerting the user to a change in instrumentation type for the source file. Note that the coverage data in the result files will be removed. To retain any Notes and Requirements, select the checkbox in the bottom pane. Select the **Make Change** button to confirm. A process dialog will appear as the file is instrumented for Probe Points.



Capture Local Variable Values

Probe points can be used to record the values of a parameter passed into a function on each invocation, or to count the number of times that the function is called.

In the example below, notice that we can insert variable declarations as well as executable statements like the call to **`vcast_probe_print`**.



Any text printed from a probe point is appended to the end of the Execution Data Report and can be used as a debugging aid. To access the Execution Data Report after the test result has been added to the Cover environment, choose the **Execution Data Report** option from the drop-down menu next to the **Probe Point** button on the Toolbar.

Event 12 - Returned from Place_Order

UUT: manager.c
Subprogram: Place_Order

Parameter	Type	Actual Value	Expected Value
Table	unsigned short	3	
Seat	unsigned short	4	
Order			
Soup	enum	CHOWDER	
Salad	enum	GREEN	
Entree	enum	STEAK	
Dessert	enum	NO_DESSERT	
Beverage	enum	MIXED_DRINK	
return	int	0	<match>

UUT Returned control to Driver, end of test case

Standard Output from Driver

```
Counting Steak Orders
Steak Order# 1
Steak Order# 2
Steak Order# 3
VCAST is Done!
```

Probe reveals order count of 3 for steak orders

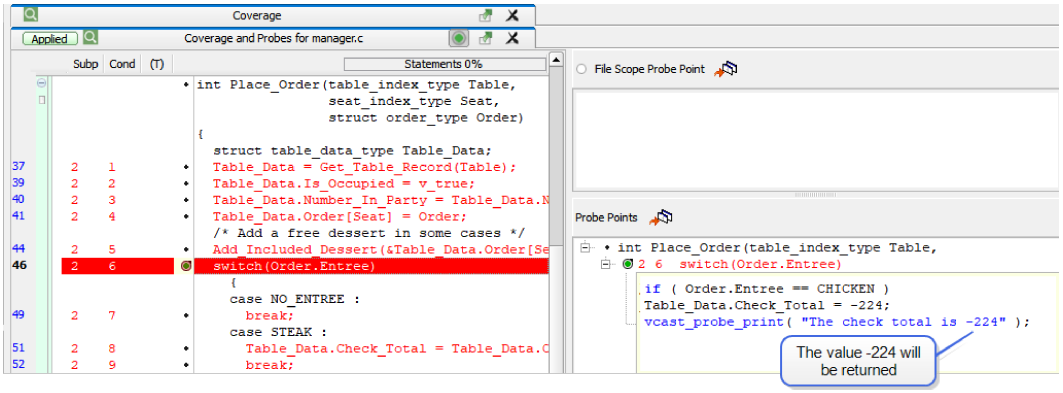
Result - PASS

Expected Results matched 100%	3/3	PASS

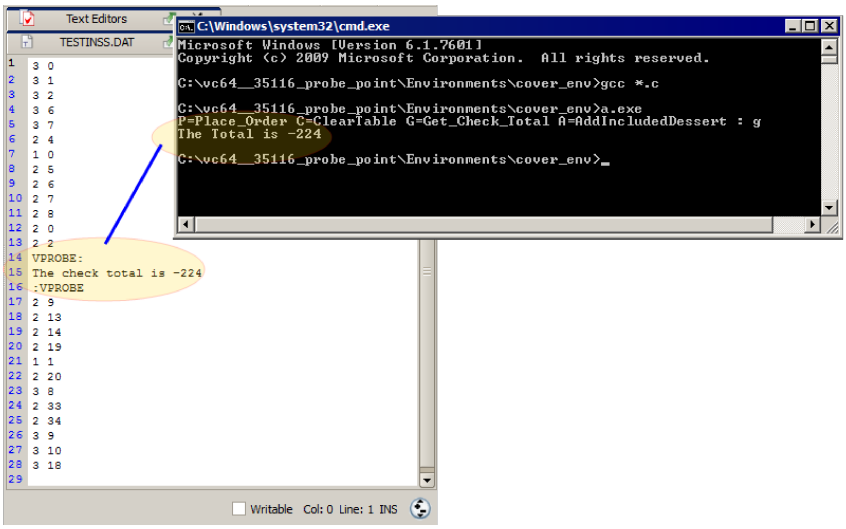
Inject Spurious Values

Probe points allow you to inject spurious values into a running system to test error handling. For example, you might have a routine that always returns a positive value, but you want to see how your system reacts in the event it receives a negative value.

In the example below, the routine will never return a negative value, but the return type is defined as **`int`**, so the value can theoretically be either positive or negative. By inserting a probe point we can cause the function to return the negative value -224 when it is called.



The probe point is saved and applied. Note that when the function is executed it returns a value of -224, and the that the probe point is included in the results file.

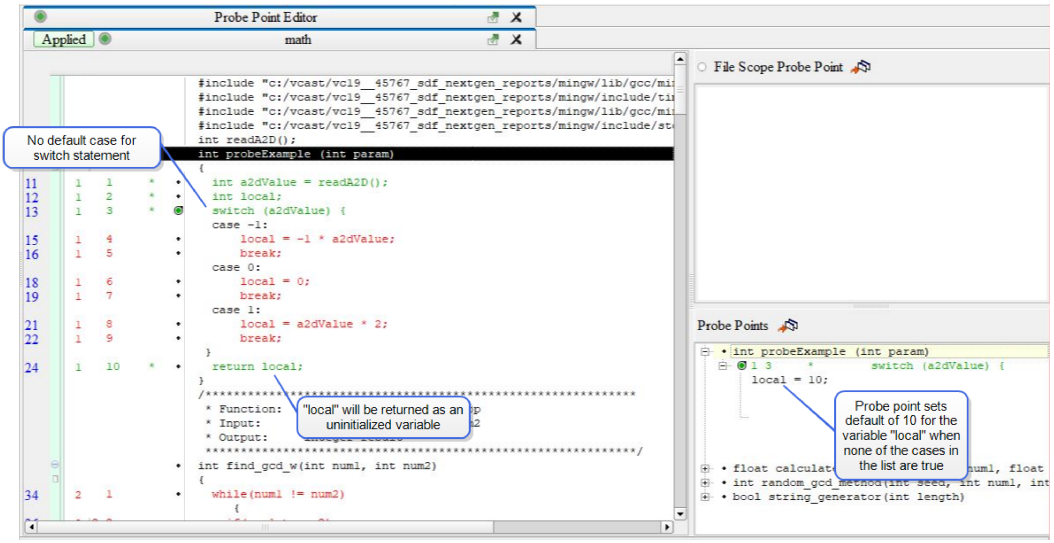


Patch Faulty Code

Probe points can be used to quickly apply a patch to faulty code. This is especially helpful when you want to test a fix prior to committing a code change.

In our code example below, note that there is no default case for the switch statement. If the value returned from **`readA2D()`** is any value other than -1, 0 or 1, **`local`** is then returned as an

uninitialized variable. This would be a likely source of bugs in the software.

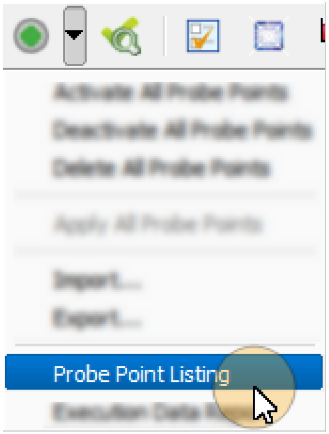


To fix this bug, we insert a probe point that initializes **`local`** to a known value of 10 and then verify that the value 10 is returned when the param is out of bounds. When the test is executed, the results show that the actual value for **`readA2D()`** is 40, which does not match the case list, so the default value of 10 is returned.

Now that the patch is verified, the changed code can be committed.

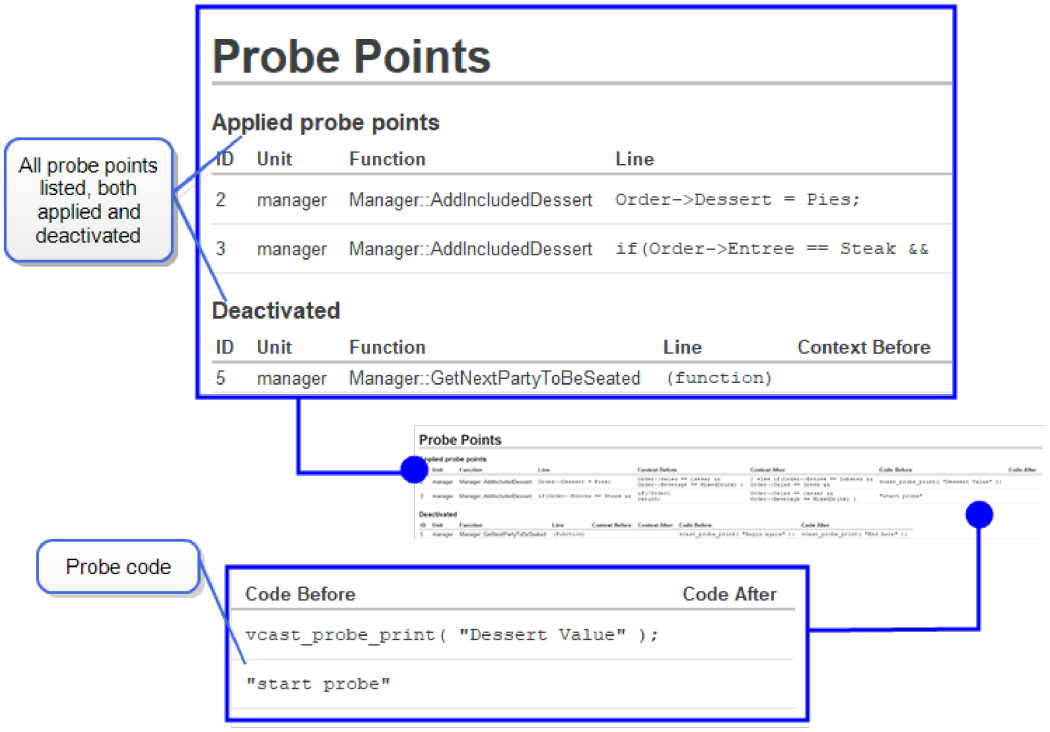
Probe Point Listing

To manually open the Probe Point Listing, select **Probe Point Listing** from the drop-down menu next to the Probe Point button on the toolbar.



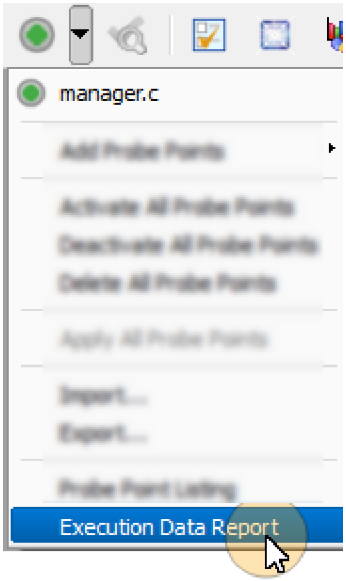
The Probe Point Listing lists all probe points for the environment. The report includes Saved, Applied, Dropped, and Deactivated probe points. For each probe point, the associated ID, Unit, Function and Line of source code are displayed. The Before and After context of the line of code is provided. The full source code for each probe point is shown, including an indication of whether the probe point is

inserted before or after the source code line.

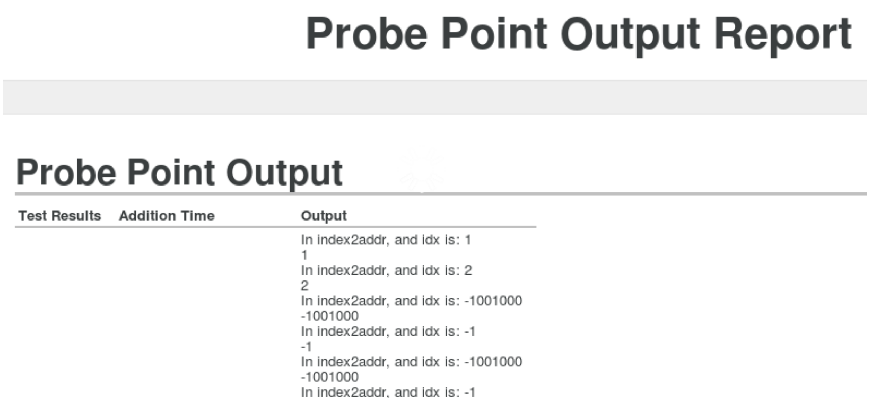


Probe Point Output Report

To open the Probe Point Output Report, select **Execution Data Report** from the drop-down menu next to the Probe Point button on the toolbar.



The Probe Point Output Report lists the Test Results, Addition Time and Output for the probe points.



Probe Point Output Report

Probe Point Output

Test Results	Addition Time	Output
		In index2addr, and idx is: 1 1 In index2addr, and idx is: 2 2 In index2addr, and idx is: -1001000 -1001000 In index2addr, and idx is: -1 -1 In index2addr, and idx is: -1001000 -1001000 In index2addr, and idx is: -1

```
clicast -e <COVER ENV> [-u <unit>] Cover REPort PRobe_point_output

Probe Point Output Report
```

Using the Probe Point API

VectorCAST Probe Point provides command line access to the data contained in the probe point database, and supports the ability to identify individual probe points by a serial number or ID.

The Probe Point API is used to export probe point data to third party tools. The functionality is implemented using CLICAST commands to enable, disable or remove specific probe points.

The Probe Point File

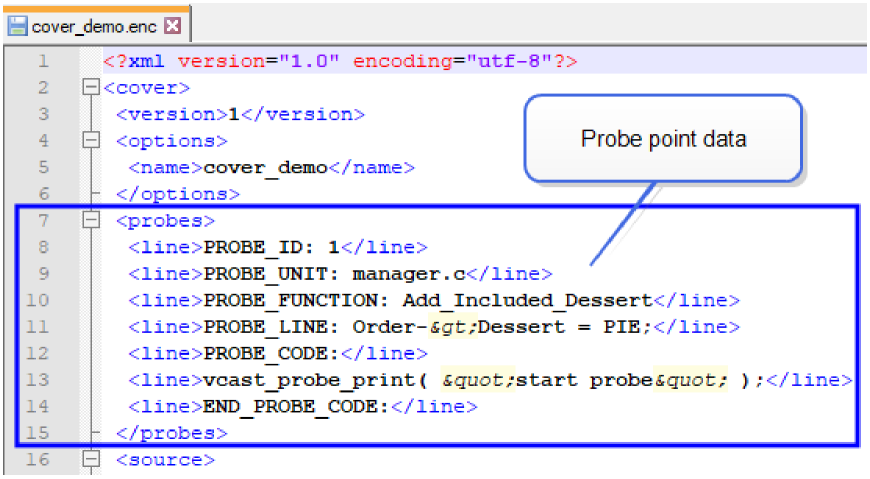
The probe point file is a result of building the regression script. A probe point is first created (either via the GUI or via CLICAST), and then saved and applied. Next, a regression script is generated by selecting **Environment => Create Regression Scripts...** from the Menu Bar.

The probe point file can reside anywhere. In our example, it resides in our environment directory: **`C:\VCAST\Examples\environments\qa_demo\cover_demo`**.

Once the regression script is generated, you will see two files in the selected directory:

- **`<env-name>.bat`**
- **`<env-name>.enc`**

The probe point data is contained within the **`<env-name>.enc`** file. An example **`.enc`** file is shown below.



The user can modify the probe point data contained in the file.

Probe ID Number

Upon creation of a probe point in the GUI or via CLICAST, the probe point is assigned a serial number or ID. The first probe point is assigned the number 1 by default, unless the probe point was added via CLICAST using the text:

PROBE_ID: <ID>

positioned after the definition of the probe point in the .env script or probe point file.

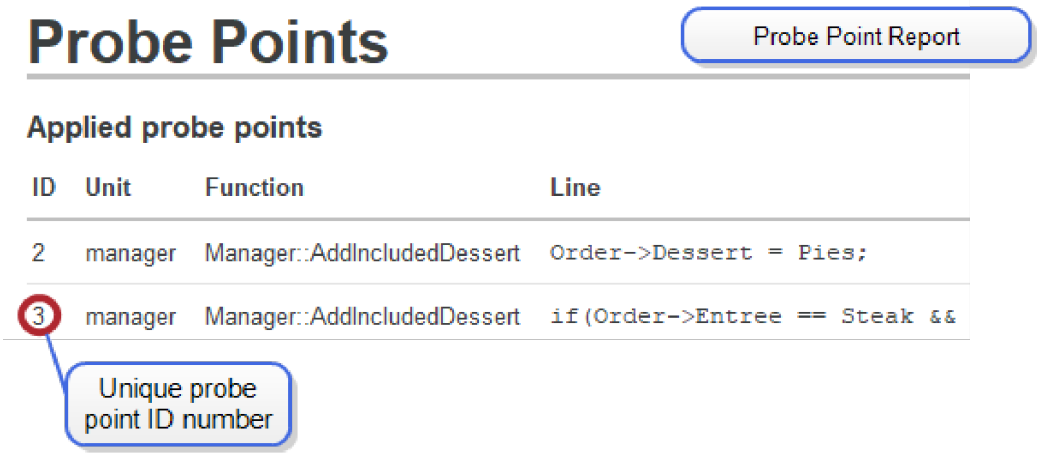


Note: If two probe points are given the same ID, the second one encountered is given the next available ID number that is unique.

The probe point file is user modifiable. For example, the probe point ID can be modified to reflect existing requirement numbers and improve traceability.

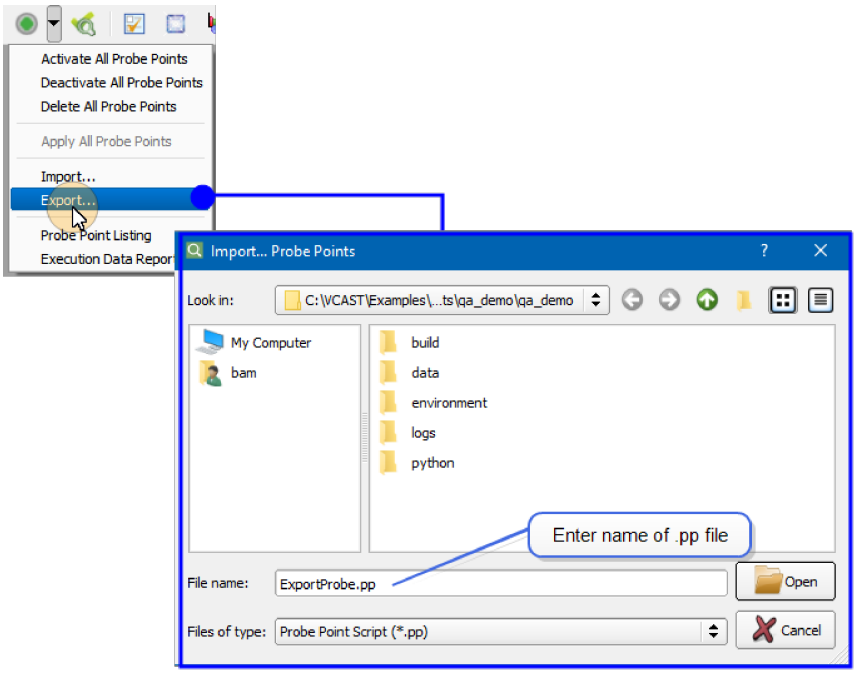
When a probe point is deleted or dropped due to changes in the source code, its ID is also deleted and can be re-used for another probe point.

Probe point IDs are displayed in the Probe Point Listing Report available from the GUI and in the probe point XML report generated with CLICAST.



Export a Probe Point

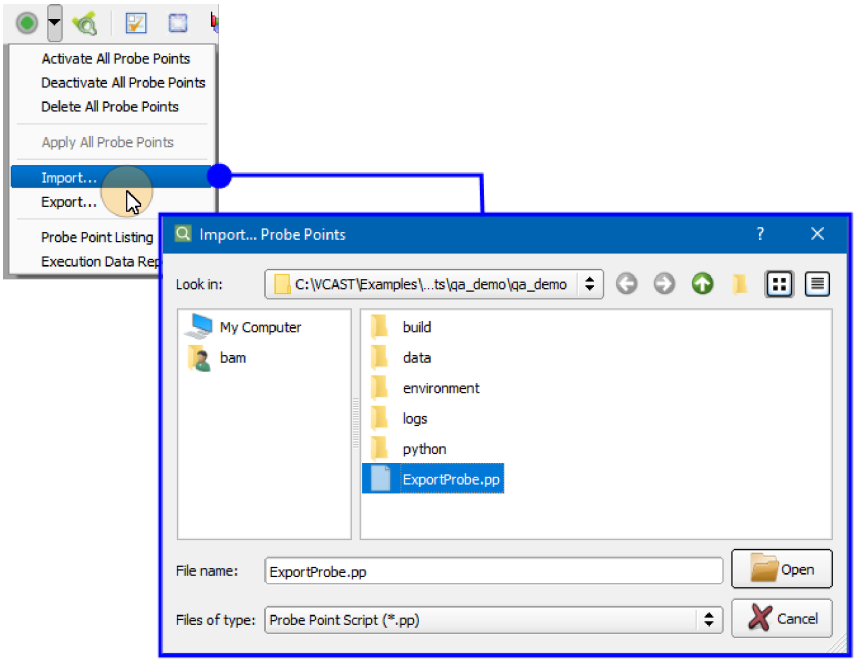
To export a probe point, select **Export...** from the drop-down menu next to the Probe Point button on the toolbar. Exporting probe points creates a .pp file, which is similar to the file created by regression scripts. Navigate to the location where you want to store the .pp file. Enter the file name and select the **Save** button.




```
clicast -lc -e <env> Cover PRobe_point EXport_file <probe_point_file>

Exports probe points specified in a .pp file. Requires 'clicast -e <env>
cover PRobe_point APPly' to be run in order to take effect.
```

Import a Probe Point

To import a probe point, select **Import...** from the drop-down menu next to the Probe Point button on the toolbar.



Navigate to the location of the .pp file and select the **Open** button. After importing, only the affected source files are reinstrumented and the Probe Point Import Log is displayed.

```
clicast -lc -e <env> Cover PRobe_point IMPort_file <probe_point_file>

Adds probe points specified in a .pp file. Requires 'clicast -e <env> cover
PRobe_point APPly' to be run in order to take effect.
```

Enable, Disable and Remove Probe Points

Using the following commands, specific probe points can be enabled, disabled or removed via the ID number.

Note: When any of the following commands are run, the action is merely scheduled to occur. The action doesn't take place until the Apply command is run.

If the environment is opened in the GUI before Apply has been performed, it is recommended that Apply be immediately performed by clicking the **Not Applied** button in the Probe Point Editor.

Enable Probe Point

```
clicast -lc -e <env> Cover PRobe_point ENAble_id <probe_point_id>

Enable (activate) the probe point specified by <probe_point_id> from the
environment. Requires 'clicast -e <env> cover PRobe_point APPly' to be run
in order to take effect.
```

Disable Probe Point

```
clicast -lc -e <env> Cover PRobe_point DISable_id <probe_point_id>

Disable (deactivate) the probe point specified by <probe_point_id> from the
environment. Requires 'clicast -e <env> cover PRobe_point APPly' to be run
in order to take effect.
```

Remove Probe Point

```
clicast -lc -e <env> Cover PRobe_point REMOVE_Id <probe_point_id>

Remove the probe point specified by <probe_point_id> from the environment.
Requires 'clicast -e <env> cover PRobe_point APPly' to be run in order to
take effect.
```

Create a Probe Point Report

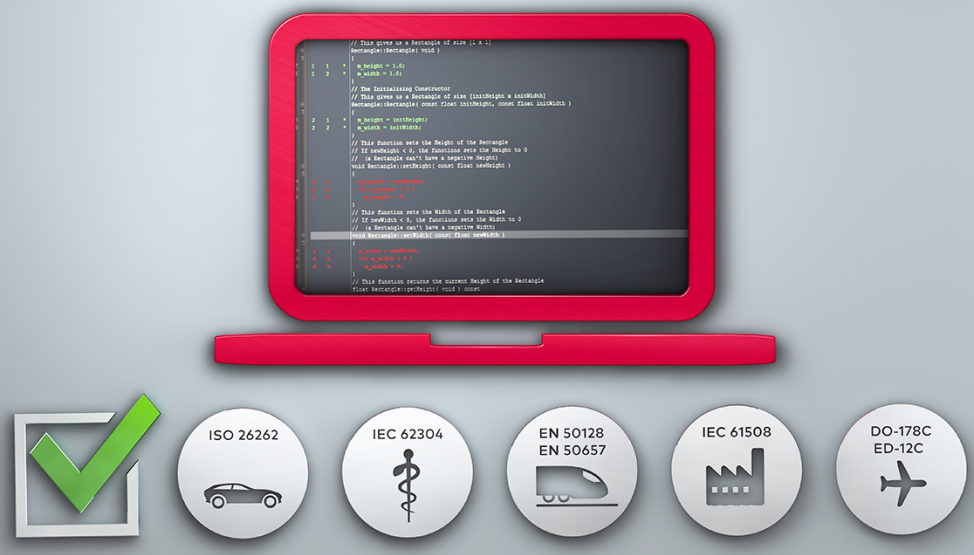
A Probe Point Report can be created in XML format using the following command:

```
clicast -lc -e <env> TOols PRobe_point Xml [<outputfile>]

Create a Probe Point Report in XML format.
```

The resulting probe_points.xml file produces a list of probe points which can be parsed and passed to a third party tool managing the probe points.

An example probe_points.xml file is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<probe_points version="1.0">
  <probe_point serial_number="1" unit="9" function="Add_Included_Dessert"
  matching_line="Order-&gt;Dessert = CAKE;" code="/*******Inserted
  Before**********/&#10;vcast_probe_print( &quot;Dessert value&quot; );"
  code_after="/*******Inserted After**********/&#10;vcast_probe_print(
  &quot;End probe&quot; );" enabled="1" applied="1" id="4" allowed_position=
  "2"/>
  <probe_point serial_number="2" unit="9" function="Add_Included_Dessert"
  matching_line="if(Order-&gt;Entree == STEAK &amp;&amp;" code=
  "/*******Inserted Before**********/&#10;vcast_probe_print( &quot;Start
  probe here&quot; );" enabled="1" applied="1" id="1" allowed_position="0"/>
  <file_scope unit="9" placement="0" tu_line="48"/>
</probe_points>
```



Using Coupling Verification

Understanding Control and Data Coupling

VectorCAST/Coupling provides built-in analysis and instrumentation for data coupling and control coupling verification for C and C++ source files and is used in conjunction with an existing VectorCAST/QA project or VectorCAST Cover environment. Coupling verification is mandatory for safety-critical avionics software development according to both RTCA DO-178B and DO-178C. The tools support the Coupling requirements of DO-178B/C by using a combination of static analysis to identify the couples in a code base and run-time verification of the couples during application execution.

Both the static analysis and run-time verification operate on software "components". A component is a user-defined collection of source files in one or more directories. DO-178C defines a component as "A self-contained part, combination of parts, subassemblies, or units that performs a distinct function of a system." (DO-178C ANNEX B, Glossary)

The intent of Coupling Analysis is to prove that the control and data flow between architectural components in the implementation match what was intended by the design, and to prove that these flows have been tested. DO-178B requires applicants to identify couples in the design, and to verify that those couples, and only those couples, exist in the implementation. DO-178C additionally requires applicants to verify that the couples have been exercised during functional requirements testing. VectorCAST's Component Report and Coupling Coverage Report provide this proof.

VectorCAST/Coupling will detect the read and write of data couples and capture calls to control couples. For data couples, the goal is to capture the order of access to each couple. For control couples, the goal is to ensure all control couples are tested by recording each time the control couple is called.

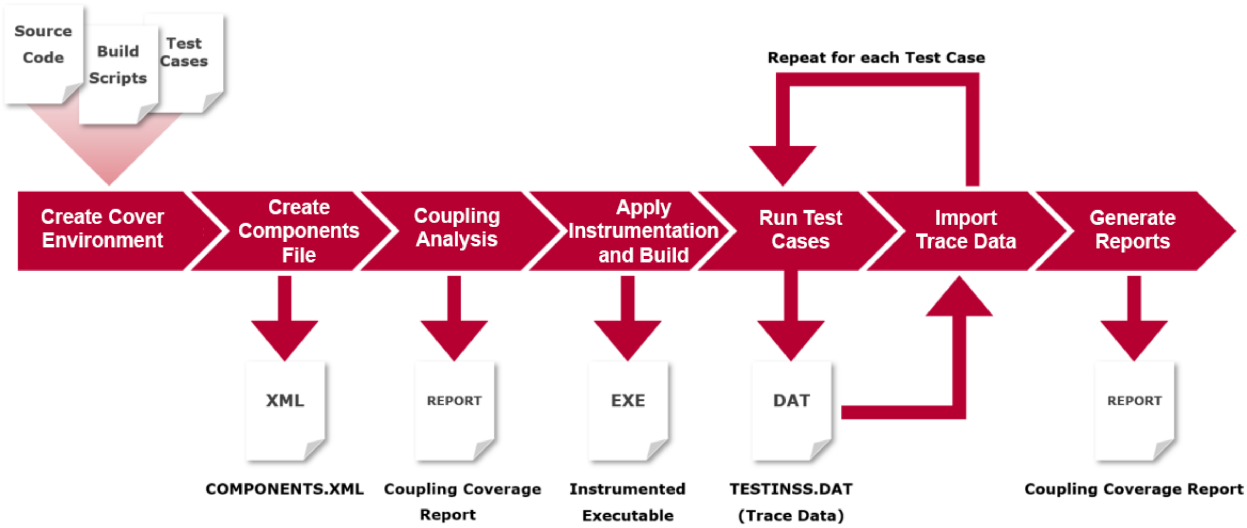
Consider the following example code where **`subSystem1.cpp`** and **`subSystem2.cpp`** are single file components. In a real application a component would be a collection of several, maybe hundreds, of files. Even though this is a simple example, it has data and control couples in both directions.



Using VectorCAST/Coupling

VectorCAST/Coupling works with C and C++ languages. To use the VectorCAST/Coupling tool, a VectorCAST Cover environment must first be created.

An illustration of the Coupling Verification process is provided below, and is discussed in the following sections.



The Cover Environment

For our example, we have created a VectorCAST Cover environment called **coupling_demo**. The instrumentation creates the symbol analysis files that the Coupling tool requires. All the intermediate files generated by the Coupling tool are stored in a **coupling** directory within our project.

Our example uses three source files: **callingSequences.cpp, subSystem1.cpp** and **subSystem2.cpp**. The **callingSequences.cpp** file has the code that drives the test cases for the verification. The three source files are split between the **Main, SS1** and **SS2** directories, and the VectorCAST Cover environment in our example exists in **VCAST\Examples\SystemTesting\CouplingExample\CouplingExample\coupling_demo** directory.

Create the Components File

The starting point for Coupling Analysis is the creation of the **components.xml** file, which defines the components and the list of source files and header files that belong to each component.

A component is generally a user-defined collection of source files and header files. Only source files and header files can be added to a component, so the smallest component would be a single file. Both the static analysis and the run-time verification operate on software components. The VectorCAST/Coupling tool automatically creates a component definition file, **components.xml**, which can be modified to match the definition of a component in your software architecture.

There is no specific guidance on how components should be defined, but it should be in line with your software architecture. Items such as subsystems, configurable items (CSCI) or modules are all possible choices for a component.

The **components.xml** file must be created before Coupling Analysis and Coupling coverage. If you already have a **components.xml** file from a previous run, this step can be skipped.

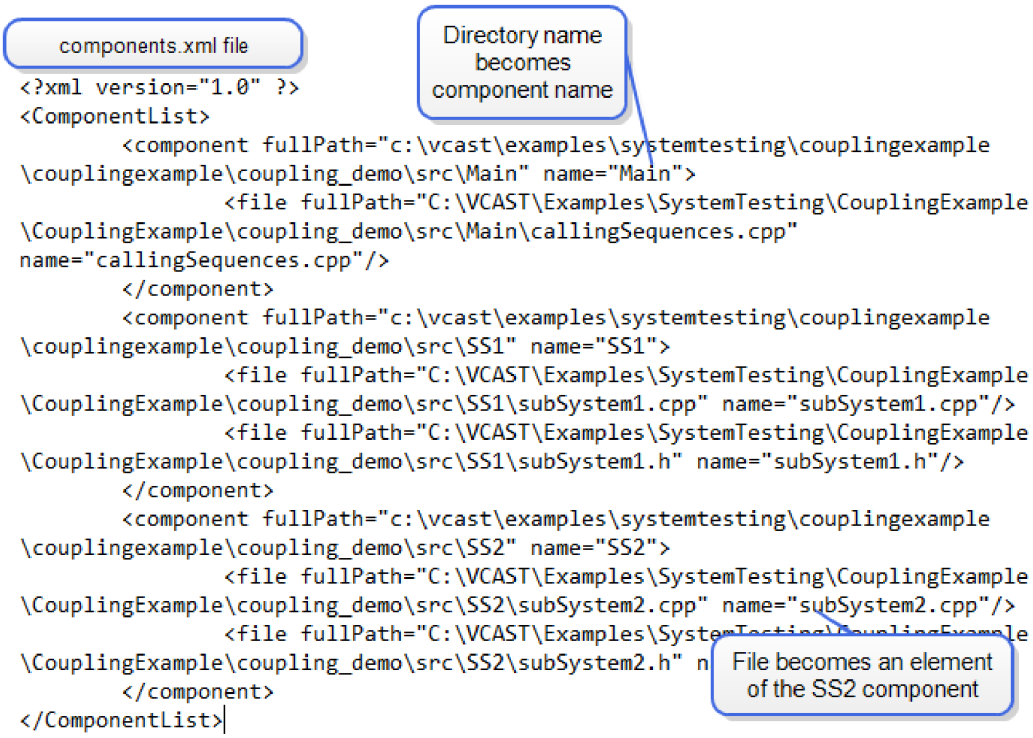
Once the Cover environment is created, the **components.xml** file is generated from the command line using the following command:

```
clicast -lc -e <COVER_ENV> COUpling COMponents

Run the coupling components creation script using the project source files.
```

Components are defined by the directories in which the Cover environment's source files reside.

The **components.xml** file is created and stored within the Cover environment's **coupling** sub-directory.

Using the directory structure, VectorCAST builds the **components.xml** file using each directory name as a component name and using the files within that directory as the elements of the component.




```
<?xml version="1.0" ?>
<ComponentList>
        <component fullPath="c:\vcast\examples\systemtesting\couplingexample
\couplingexample\coupling_demo\src\Main" name="Main">
                <file fullPath="C:\VCAST\Examples\SystemTesting\CouplingExample
\CouplingExample\coupling_demo\src\Main\callingSequences.cpp"
name="callingSequences.cpp"/>
        </component>
        <component fullPath="c:\vcast\examples\systemtesting\couplingexample
\couplingexample\coupling_demo\src\SS1" name="SS1">
                <file fullPath="C:\VCAST\Examples\SystemTesting\CouplingExample
\CouplingExample\coupling_demo\src\SS1\subSystem1.cpp" name="subSystem1.cpp"/>
                <file fullPath="C:\VCAST\Examples\SystemTesting\CouplingExample
\CouplingExample\coupling_demo\src\SS1\subSystem1.h" name="subSystem1.h"/>
        </component>
        <component fullPath="c:\vcast\examples\systemtesting\couplingexample
\couplingexample\coupling_demo\src\SS2" name="SS2">
                <file fullPath="C:\VCAST\Examples\SystemTesting\CouplingExample
\CouplingExample\coupling_demo\src\SS2\subSystem2.cpp" name="subSystem2.cpp"/>
                <file fullPath="C:\VCAST\Examples\SystemTesting\CouplingExample
\CouplingExample\coupling_demo\src\SS2\subSystem2.h" n
        </component>
</ComponentList>
```

Perform Coupling Analysis

Coupling Analysis performs static analysis on the source code and determines the couples between the components. Coupling Analysis requires the source files to be instrumented, so if the Cover environment is not instrumented, Coupling Analysis instruments it with the Coupling coverage type.

Coupling Analysis requires an existing **components.xml** file. By default, VectorCAST uses the **components.xml** file located in the Cover environment's **coupling** sub-directory. The user can provide an alternate file on the command line if desired.

To perform Coupling Analysis, enter the following command:

```
clicast -lc -e <COVER_ENV> COUpling ANalyze [<components-file>]

Perform the coupling analysis using the specified components file, or, if
no file is specified, use the file in the environment directory.
```

If there is not an exact match of source files between the Cover environment and the **components.xml** file, an error message is displayed. The user must correct the mis-match in order to perform the Coupling Analysis successfully. In our example below, the Cover environment contains a source file that is not in the **components.xml** file:

```
-----------------------------------------------------------------
ERROR: The following source files are not in any component...
-----------------------------------------------------------------
Missing-File:subsystem23.cpp
```

Two levels of analysis are performed. First, the symbols for each component are analyzed and a Component Report is generated. Second, the symbol analysis is used to generate a Coupling Coverage Report. Before proceeding to the next step, which is to apply Coupling to the source files, the reports should be reviewed by the user to verify that they match the design. For more information on the two reports, see "Coupling Reports" on page 295.

Apply Coupling to Instrumented Source Files

Coupling Analysis must be completed prior to applying Coupling to the instrumented source files. When Coupling is applied, each source file is re-instrumented to add the function calls necessary to gather control coupling data during execution.

To apply Coupling, enter the following command:

```
clicast -lc -e <COVER_ENV> COUpling APply

Add the function calls necessary to gather control coupling data during
execution.
```

The command builds an instrumentation script file containing the coupling instrumentation for each file, as well as the coupling data file that must be linked into the instrumented application.

The underlying data structures and the implementation of the couple functions are generated in the file **couplingData.c**. The instrumentation functions in this file use the same output mechanisms that VectorCAST uses in other Cover environments to dump coverage data. If you are configured for real-time coverage IO the ASCII data is generated as each instrumentation point is executed, otherwise the ASCII data is generated at the end of the test using the **vcastDumpCouplingData()** API.

This function must be called when the coverage I/O method is set to buffered. You will either need to add a call to your main program, or you will need to use the function **atexit** to register this function to be called when the application terminates.

Build the Instrumented Application

Coupling instrumentation must be applied prior to compiling and executing the instrumented source files. During execution of the instrumented application, data on the data couples and control couples

is collected and the information is used in the Coupling Coverage Report.

If the Cover environment is not using the Append Cover IO feature, then the following instructions should be followed.

Using a GNU compiler, the compile and link command for our example is:

```
g++ -o driver Main/callingSequences.cpp SS1/subSystem1.cpp
SS2/subSystem2.cpp -ICouplingExample__Coverage CouplingExample__Coverage/c_
cover_io.cpp CouplingExample__Coverage/coupling/couplingData.c
```



Note: The file **<cover env>/coupling/couplingData.c** (which implements the coupling functions) must be compiled and linked when compiling C or C++ source files. This is done in the same manner as compiling and linking the **c_cover_io.c** file. The user will receive a link error if the file is not compiled.

Run Test Cases and Import Trace Data

Once the application is built with instrumentation, the execution captures trace information. The coupling trace data is retrieved from the target using the same mechanism as is used for the code coverage data. If you set the macro: VCAST_COVERAGE_IO_REAL_TIME, the conversion of the coverage data to ASCII and the dumping or buffering of this string occurs in real time for each instrumentation point.

If this macro is not set (i.e. buffered I/O is being used), the coupling trace data is cached in RAM, and can be retrieved by calling the function **vcastDumpCouplingData()**, or by using a debugger to dump the C array: **vcastCouplingData[]**.

If you are configured for real-time IO, the coupling trace data is exported and shows the order in which the control couples and data couples are accessed.

If you are configured for buffered IO, you can cause the coverage data to be dumped by calling the target-side function **vcastDumpCouplingData()**.

If you have a debug connection to your target, you can use this connection to capture the raw data from the global data object **vcastCouplingData**, which is defined in the instrumentation file **couplingData.c**. In this case, you will need to convert the data from the debugger format to the: **"couple-data:0,1"** format in order for VectorCAST to perform the import. This can be easily accomplished with a script with logic similar to the functions **vcastDumpCouplingData()** and **vcastDumpOneCouplingCell()**, which are also part of the **couplingData.c** file.

Each test case is run and the coupling trace data file (TESTINSS.DAT) is saved for each test case.

The coupling trace data is imported into the Cover environment using the following command:

```
clicast -lc -e <COVER_ENV> Cover RESult Add TESTINSS.DAT [<optional-result-
name>] [-n <notes test | notes file>] [-r <requirements text | requirements
file>]

Adds the trace data to the Cover environment.
```

Generate Coupling Coverage Report

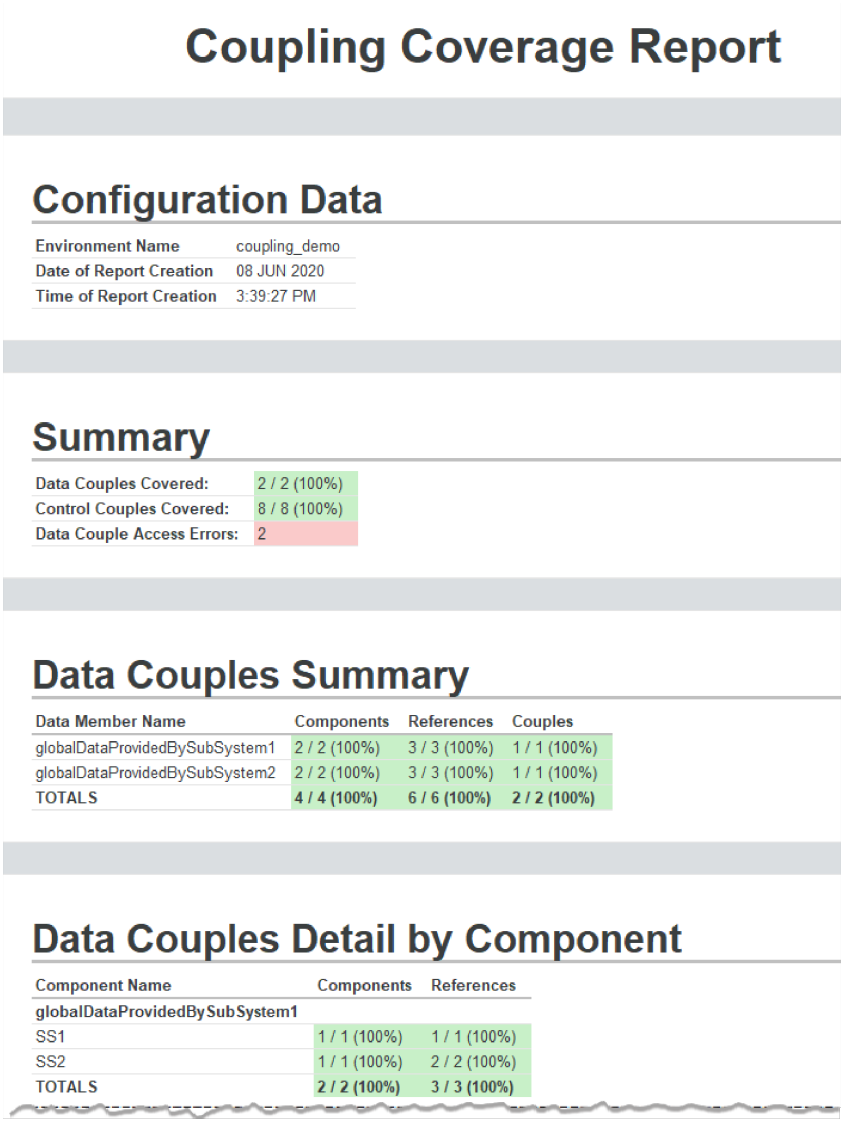
At the conclusion of the Coupling Analysis process the Coupling Coverage Report, showing all the

Data and Control Coupling Analysis, is available.

To open the Coupling Report, select **Environment => View => Coupling Coverage Report** from the Menu Bar, or enter the following command:

```
clicast -lc -e <COVER_ENV> COUpling REport COVerage [<output-file>]

Generate report listing the data and control couples, and details on which
couples are covered by which test result.
```

The figure below shows a partial example of a Coupling Coverage Report. For more information on the full report contents, see "Coupling Coverage Report" on page 297.



Coupling Coverage Report

Configuration Data

Environment Name	coupling_demo
Date of Report Creation	08 JUN 2020
Time of Report Creation	3:39:27 PM

Summary

Data Couples Covered:	2 / 2 (100%)
Control Couples Covered:	8 / 8 (100%)
Data Couple Access Errors:	2

Data Couples Summary

Data Member Name	Components	References	Couples
globalDataProvidedBySubSystem1	2 / 2 (100%)	3 / 3 (100%)	1 / 1 (100%)
globalDataProvidedBySubSystem2	2 / 2 (100%)	3 / 3 (100%)	1 / 1 (100%)
TOTALS	**4 / 4 (100%)**	**6 / 6 (100%)**	**2 / 2 (100%)**

Data Couples Detail by Component

Component Name	Components	References
globalDataProvidedBySubSystem1		
SS1	1 / 1 (100%)	1 / 1 (100%)
SS2	1 / 1 (100%)	2 / 2 (100%)
TOTALS	**2 / 2 (100%)**	**3 / 3 (100%)**

Re-analyze Coupling

Coupling information can be re-analyzed when changes occur to the source file. To start a new Coupling Analysis, select **Tools => Coupling => Analyze** from the Menu Bar.

```
clicast -lc -e <env> COUpling ANalyze [<components-file>]

Perform the coupling analysis using the specified components file, or, if
no file is specified, use the file in the environment directory.
```

Setting Coupling Options

The following coverage options are available for use with the Coupling tool:

VCAST_IGNORE_ERROR_WHEN_READ_FOLLOWS_ADDRESS_OF

When this option is enabled, VectorCAST ignores data couple access errors for a read following an address of operation. For full details on the use of this option, see "Ignore Access Error When Read Follows Address of" on page 204.

VCAST_TREAT_DC_INITIALIZATIONS_AS_WRITE

When this option is enabled, global data initialization expressions are treated as a write. For full details on the use of this option, see "Treat Initialization of Data Couple as a Write" on page 203.

Coupling Reports

VectorCAST's set of Coupling Reports provide proof that the control and data flow between architectural components in the implementation match what was intended by the design, and prove that these flows have been tested.

VectorCAST/Coupling provides the following reports, which are discussed in detail below:

- Component Report - lists the global data objects and functions for each component.
- Coupling Coverage Report - shows the static analysis and runtime analysis. This report is a high-level overview of the Coupling coverage in the application.
- Coupling Access Order Full Report - shows the access order by test result for all couples. This verifies that the architecture was correctly followed for all of the tests.
- Coupling Access Order Error Report - shows the access order by test result only for couples with errors. This report helps the user to efficiently track down any access errors without having to parse through all of the test results.

Coupling reports may display the following control couple access types:

- **`call`** - a function call
- **`fptr-call`** - a function pointer call
- **`vrtl-call`** - a virtual function call

Coupling reports also may display the following data couple access types:

- **`read`** - a variable was read
- **`(&)read`** - an address of a variable was read

- **write** - a variable was written
- **initialized** - a variable was initialized

Component Report

The Component Report is a text-only list of all the "needed" and "defined" global data objects and functions for each component. "Needed" means that the object or function is used by the current component. "Defined" means that the object or function is available to an external component. The report is a first step in understanding the inter-dependencies for data and control between any two components.

To generate the Component Report, the **components.xml** file must already be created.

Enter the following command to generate the report:

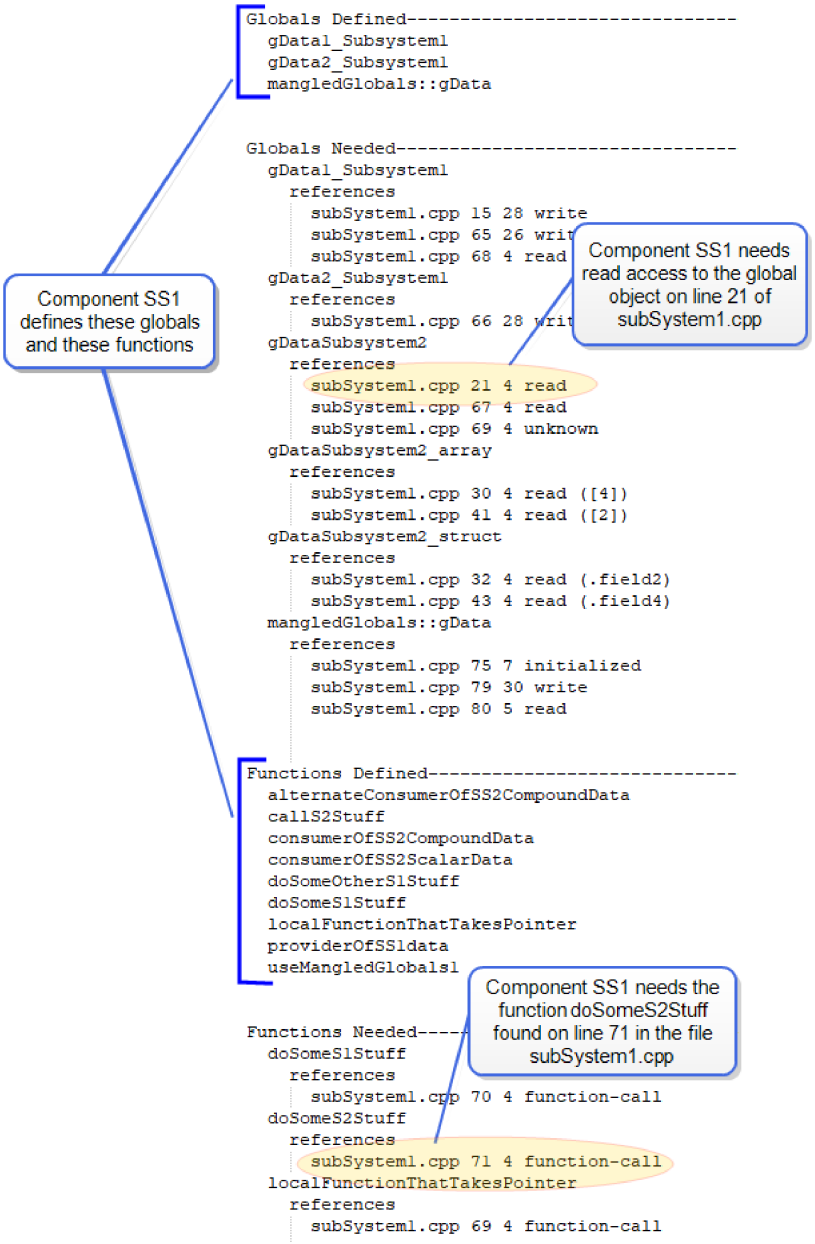
```
clicast -lc -e <COVER_ENV> COUpling REport COMponents [<component>]
[<output-file>]

Generate reports for the specified data/control coupling component, or all
components if none is specified.
```

The Component Report contains four sections: Globals Defined, Globals Needed, Functions Defined, and Functions Needed for each component in the report.

In the Globals Needed and Functions Needed sections, the filename and line number of the reference is provided. For Globals Needed, the Report also shows whether the reference requires read, write, or read/write access.

Note that when compilers mangle function names, you will see the mangled names displayed for the functions. Tracking mangled names is the only way to ensure that analysis is sound across multiple names.

An example Component Report for the Component SS1 is shown below:



Coupling Coverage Report

The Coupling Coverage Report provides run-time verification of the couples during application execution, as well as the static analysis. The Report shows a listing of all the Data Couples and Control Couples found in all components, with details on which couples are covered by each test result in the environment. Each sub-section of the report provides additional detail about the couples information arranged from a different perspective.

The Coupling Coverage Report consists of the following sections:

- Configuration Data - Contains environment name as well as the date and time of the report creation

- Summary - Shows metrics of data couples and control couples covered, and the number of data couple access errors
- Data Couples Summary- A list of the Data Members with its Component, References and Couples metrics
- Data Couples Detail by Component - A detailed view of each Data Member's components
- Data Couples Detail by Reference - A detailed view of each Data Member's components and where it was read and written to.
- Control Couples Summary - A list of function names along with Components, References and Couples metrics
- Control Couples Detail by Component - A detailed list of the Function Component name as well as Components and References metrics
- Control Couples Detail by Reference - A detailed view of each Function name and Component and location of the Access
- Data Couple Access Errors - A list of data couples where a read access occurred before a write access
- Function Pointer Analysis - Information on function pointer calls that require user analysis

To generate the Coupling Coverage Report, Coupling Analysis must be performed.



Note: VectorCAST allows the Coupling Coverage Report to be generated after Coupling Analysis, even if instrumentation has not been applied.

To open the report, select **Environment => View => Coupling Coverage Report** from the Menu Bar, or enter the following command:

```
clicast -lc -e <env> COUpling REport COVerage [<output-file>]

Generate report listing all the data/control couples and their references.
```

A partial example of the Coupling Coverage Report is shown below:

Coupling Coverage Report

Configuration Data

Environment Name	coupling_demo
Date of Report Creation	08 JUN 2020
Time of Report Creation	3:39:27 PM

Summary

Data Couples Covered:	2 / 2 (100%)
Control Couples Covered:	8 / 8 (100%)
Data Couple Access Errors:	2

Data Couples Summary

Data Member Name	Components	References	Couples
globalDataProvidedBySubSystem1	2 / 2 (100%)	3 / 3 (100%)	1 / 1 (100%)
globalDataProvidedBySubSystem2	2 / 2 (100%)	3 / 3 (100%)	1 / 1 (100%)
TOTALS	**4 / 4 (100%)**	**6 / 6 (100%)**	**2 / 2 (100%)**

Data Couples Detail by Component

Component Name	Components	References
globalDataProvidedBySubSystem1		
SS1	1 / 1 (100%)	1 / 1 (100%)
SS2	1 / 1 (100%)	2 / 2 (100%)
TOTALS	**2 / 2 (100%)**	**3 / 3 (100%)**
globalDataProvidedBySubSystem2		
SS1	1 / 1 (100%)	2 / 2 (100%)
SS2	1 / 1 (100%)	1 / 1 (100%)
TOTALS	**2 / 2 (100%)**	**3 / 3 (100%)**
GRAND TOTALS	**4 / 4 (100%)**	**6 / 6 (100%)**

Data Couples Detail by Reference

Data Member Name	Component	File	Line	Access	Hit
globalDataProvidedBySubSystem1					
	SS1	subSystem1.cpp	8	write	*
	SS2	subSystem2.cpp	14	read	E
	SS2	subSystem2.cpp	26	read	*
globalDataProvidedBySubSystem2					
	SS1	subSystem1.cpp	14	read	E
	SS1	subSystem1.cpp	26	read	*
	SS2	subSystem2.cpp	8	write	*

Coupling Access Order Full Report

The Coupling Access Order Full Report is a detailed Coupling verification report for each test.This report verifies that the architecture was correctly followed for all of the tests. The report details the test coverage in access order for each test result added.

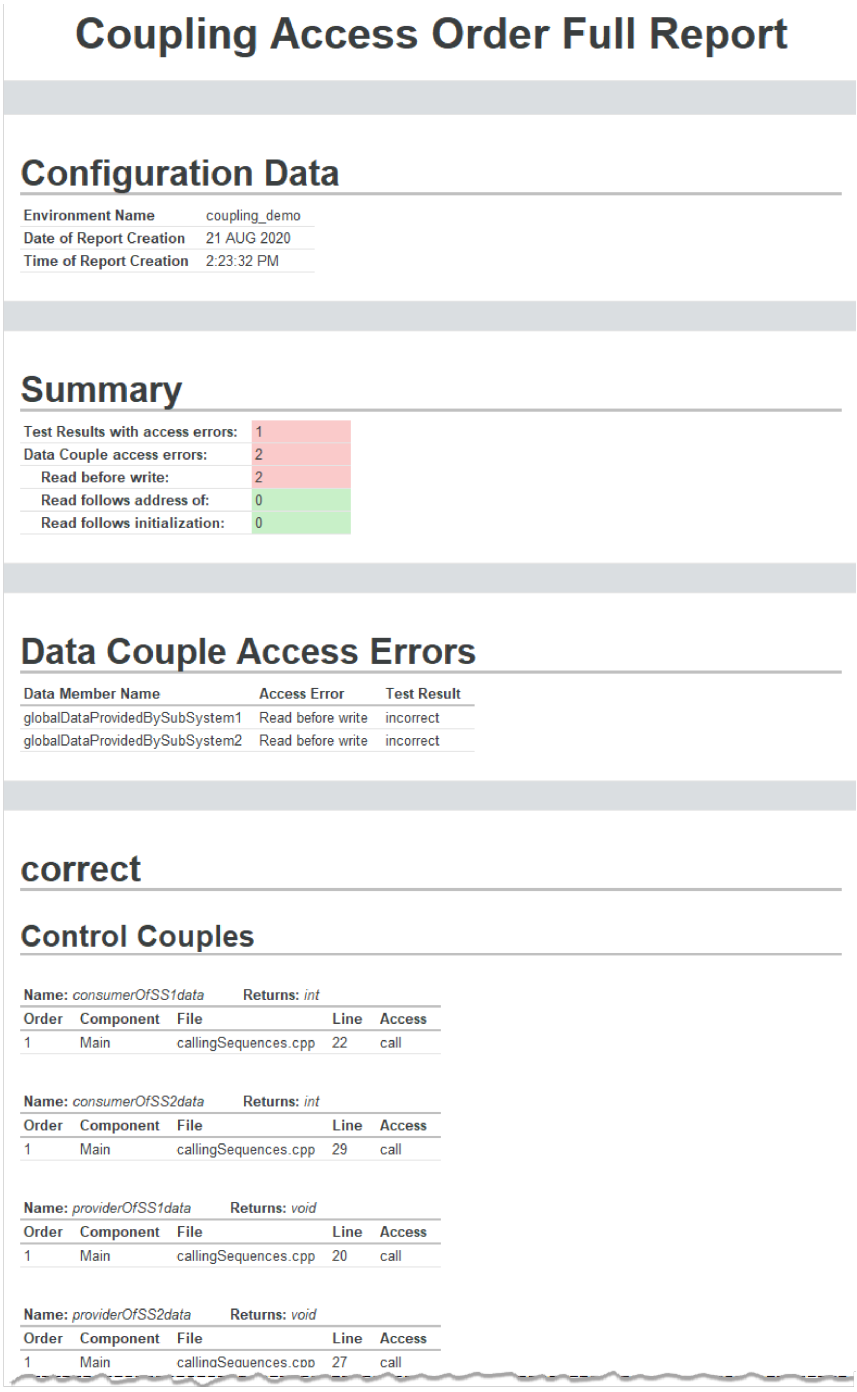
To open the report, select **Tools => Coupling => Access Order Report => View Full Report** from the Menu Bar, or enter the following command:

```
clicast -lc -e <env> COUpling REport ACCESS_ORDER_Full [<output-file>] [-t
<test>]

Generate the coupling access order full report. This shows the access order
per test result for each data and control couple.
```

The Coupling Access Order Full Report consists of the following sections:

- Configuration Data - Contains environment name as well as the date and time of the report creation
- Summary - Shows metrics for test results with access errors and for data couple access errors
- Data Couple Access Errors - Shows details of access errors for data couples
- Control Couples per Test Result - Shows the access order for each control couple. References with an error are marked.
- Data Couples per Test Result - Shows the access order for each data couple. References with an error are marked.

A partial example of the Coupling Access Order Full Report is shown below:



Coupling Access Order Full Report

Configuration Data

Environment Name	coupling_demo
Date of Report Creation	21 AUG 2020
Time of Report Creation	2:23:32 PM

Summary

Test Results with access errors:	1
Data Couple access errors:	2
Read before write:	2
Read follows address of:	0
Read follows initialization:	0

Data Couple Access Errors

Data Member Name	Access Error	Test Result
globalDataProvidedBySubSystem1	Read before write	incorrect
globalDataProvidedBySubSystem2	Read before write	incorrect

correct

Control Couples

Name: *consumerOfSS1data* **Returns:** *int*

Order	Component	File	Line	Access
1	Main	callingSequences.cpp	22	call

Name: *consumerOfSS2data* **Returns:** *int*

Order	Component	File	Line	Access
1	Main	callingSequences.cpp	29	call

Name: *providerOfSS1data* **Returns:** *void*

Order	Component	File	Line	Access
1	Main	callingSequences.cpp	20	call

Name: *providerOfSS2data* **Returns:** *void*

Order	Component	File	Line	Access
1	Main	callingSequences.cpp	27	call

Coupling Access Order Error Report

The Coupling Access Order Error Report details the test coverage in access order for each test result added that contains an error. Only couples with access errors are displayed in this report.

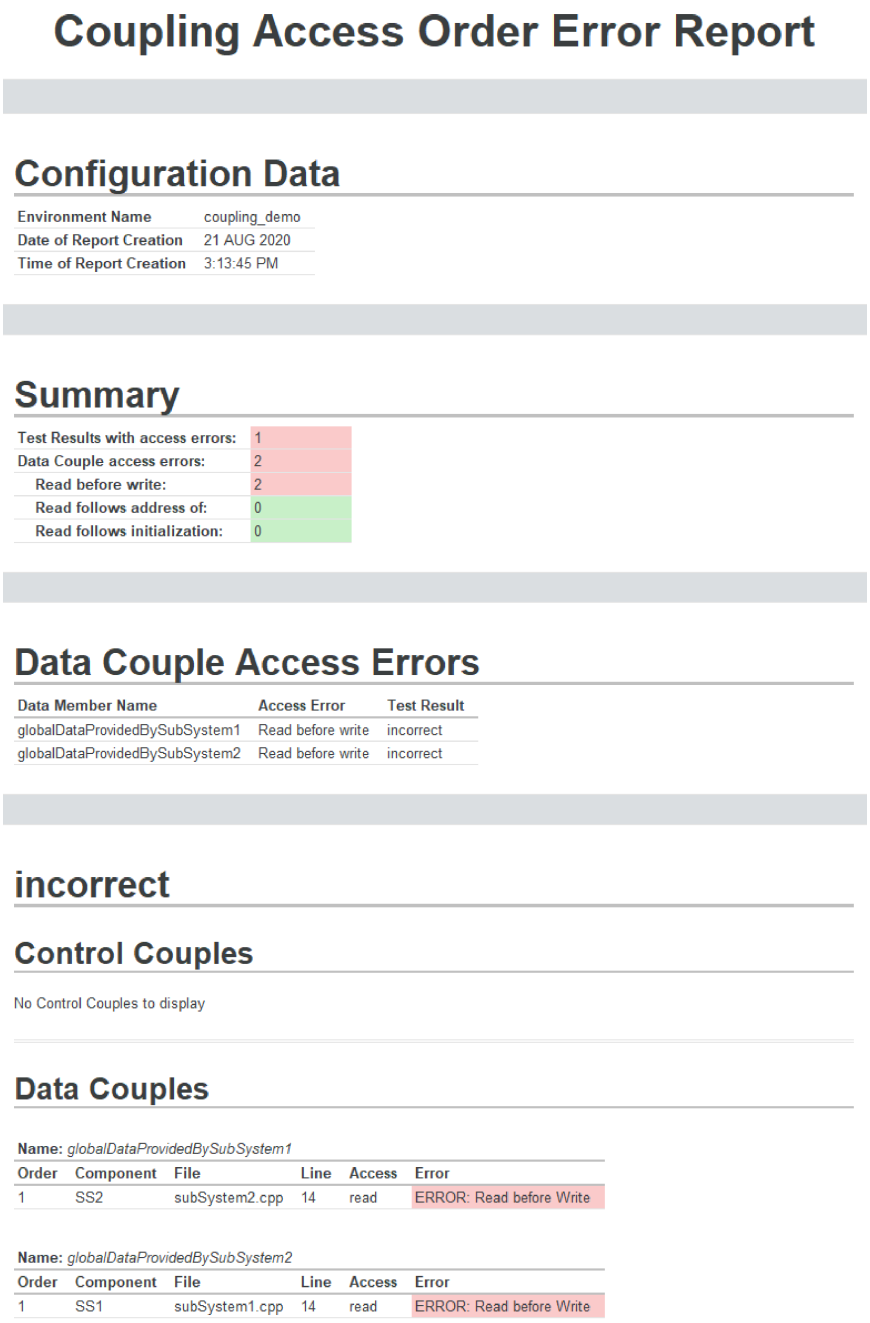
To open the report, select **Tools => Coupling => Access Order Report => View Error Report** from the Menu Bar, or enter the following command:

```
clicast -lc -e <env> COUpling REport ACCESS_ORDER_Errors [<output-file>] [-
t <test>]

Generate the coupling access order errors report. This shows the access
order per test result for only data and control couples with errors.
```

The Coupling Access Order Error Report consists of the following sections:

- Configuration Data - Contains environment name as well as the date and time of the report creation
- Summary - Shows metrics for test results with access errors and for data couple access errors
- Data Couple Access Errors - Shows details of access errors for data couples
- Incorrect Control Couples and Data Couples - Shows details of access errors of control and data couples.

A partial example of the Coupling Access Order Error Report is shown below:



Coupling Access Order Error Report

Configuration Data

Environment Name	coupling_demo
Date of Report Creation	21 AUG 2020
Time of Report Creation	3:13:45 PM

Summary

Test Results with access errors:	1
Data Couple access errors:	2
Read before write:	2
Read follows address of:	0
Read follows initialization:	0

Data Couple Access Errors

Data Member Name	Access Error	Test Result
globalDataProvidedBySubSystem1	Read before write	incorrect
globalDataProvidedBySubSystem2	Read before write	incorrect

incorrect

Control Couples

No Control Couples to display

Data Couples

Name: *globalDataProvidedBySubSystem1*

Order	Component	File	Line	Access	Error
1	SS2	subSystem2.cpp	14	read	ERROR: Read before Write

Name: *globalDataProvidedBySubSystem2*

Order	Component	File	Line	Access	Error
1	SS1	subSystem1.cpp	14	read	ERROR: Read before Write

View Coupling Logs

Coupling process output logs allow you to see the stdout from the analysis and are available for viewing when the Cover environment is open. To access the logs, select **Tools => Coupling => View Logs** from the Menu Bar and select one of the following logs from the context menu:

- **Analysis Log** - shows file `computeCouples.stdout.txt`
- **Instrumentation Log** - shows file `computeInstrumentationData.stdout.txt`

- **Coupling Report Log** - shows file **`generateReport.stdout.txt`**
- **Access Order Full Report Log** - shows file **`accessOrderFullReport.stdout.txt`**
- **Access Order Error Report Log** - shows file **`accessOrderErrorReport.stdout.txt`**

An example of each log is provided below.

Example - Analysis Log

```
VectorCAST/Coupling Analysis


Processing component: Main---------------------------------------

Begin processing XML files---------------------------------------
Processing file:
C:\VCAST\Examples\SystemTesting\CouplingExample\CouplingExample\coupli
ng_demo\coupling_demo\callingSequences.1.cpp.needed.xml
Processing Global Objects ...
Processing Functions ...
Processing Function Pointers ...
Done processing XML files----------------------------------------

The following 1 needed.xml file(s) were processed: ---------------
C:\VCAST\Examples\SystemTesting\CouplingExample\CouplingExample\coupli
ng_demo\coupling_demo\callingSequences.1.cpp.needed.xmll

Creating component report (XML): component.Main.xml---------------

Creating component report (text): component.Main.txt--------------
text file complete

Processing component: SS1----------------------------------------

Begin processing XML files---------------------------------------
Processing file:
C:\VCAST\Examples\SystemTesting\CouplingExample\CouplingExample\coupli
ng_demo\coupling_demo\subSystem1.2.cpp.needed.xml
Processing Global Objects ...
Processing Functions ...
Processing Function Pointers ...
Done processing XML files----------------------------------------

The following 1 needed.xml file(s) were processed: ---------------
C:\VCAST\Examples\SystemTesting\CouplingExample\CouplingExample\coupli
ng_demo\coupling_demo\subSystem1.2.cpp.needed.xml

Creating component report (XML): component.SS1.xml----------------

Creating component report (text): component.SS1.txt---------------
text file complete

Processing component: SS2----------------------------------------

Begin processing XML files---------------------------------------
Processing file:
C:\VCAST\Examples\SystemTesting\CouplingExample\CouplingExample\coupli
ng_demo\coupling_demo\subSystem2.3.cpp.needed.xml
Processing Global Objects ...
Processing Functions ...
```

```
Processing Function Pointers ...
Done processing XML files----------------------------------------

The following 1 needed.xml file(s) were processed: ---------------
C:\VCAST\Examples\SystemTesting\CouplingExample\CouplingExample\coupli
ng_demo\coupling_demo\subSystem2.3.cpp.needed.xml

Creating component report (XML): component.SS2.xml----------------

Creating component report (text): component.SS2.txt---------------
text file complete

Building Couples Report (master-couples-list)--------------------
 Text report saved to: master-couples-list.txt
 Saving XML report to: master-couples-list.xml
 Couples Report Complete
```

Example - Instrumentation Log

```
VectorCAST/Coupling Instrumentation -- computing instrumentation data

 Reading analysis file: analysis-needed.txt
Loading couples data ...
 Reading analysis file: analysis-needed.txt
 Reading analysis file: analysis-needed.txt
 Reading analysis file: analysis-needed.txt
Saving XML data to: couplingInstrumentation.xml
Creating instrumentation commands for file: callingSequences.cpp (1)
Creating instrumentation commands for file: subSystem1.cpp (2)
Creating instrumentation commands for file: subSystem2.cpp (3)

VectorCAST/Coupling Instrumentation - updating instrumentation data

Updating instrumentation data ...
 file: callingSequences.cpp, has: 8 probes, and 0 errors
 file: subSystem1.cpp, has: 4 probes, and 0 errors
 file: subSystem2.cpp, has: 4 probes, and 0 errors
Total files: 3, with: 16 probes, and 0 errors
```

Example - Coupling Report Log

```
VectorCAST/Coupling Reporting

Reading analysis file: analysis-needed.txt
Reading analysis file: analysis-needed.txt
Reading analysis file: analysis-needed.txt
Processing data for test: correct
Processing coupling data file:
C:\VCAST\Examples\SystemTesting\CouplingExample\CouplingExample\coupling_
demo\coupling_demo\results\correct
found: 8 trace items ...
Processing data for test: full
Processing coupling data file:
C:\VCAST\Examples\SystemTesting\CouplingExample\CouplingExample\coupling_
demo\coupling_demo\results\full
found: 14 trace items ...
Processing data for test: incorrect
Processing data file:
C:\VCAST\Examples\SystemTesting\CouplingExample\CouplingExample\coupling_
demo\coupling_demo\results\incorrect
found: 4 trace items ...
Checking for instrumentation errors ...
no instrumentation error(s) found
```

Example - Access Order Full Report Log

```
Reading analysis file: analysis-needed.txt
Reading analysis file: analysis-needed.txt
Processing data for test: correct
Processing coupling data file: .\results/correct
found: 8 trace items ...
Processing data for test: incorrect
Processing coupling data file: .\results/incorrect
found: 4 trace items ...
Processing data for test: full
Processing coupling data file: .\results/full
found: 14 trace items ...
Reading analysis file: analysis-needed.txt
```

Example - Access Order Error Report Log

```
Reading analysis file: analysis-needed.txt
Reading analysis file: analysis-needed.txt
Processing data for test: correct
Processing coupling data file: .\results/correct
found: 8 trace items ...
Processing data for test: incorrect
Processing coupling data file: .\results/incorrect
found: 4 trace items ...
Processing data for test: full
Processing coupling data file: .\results/full
found: 14 trace items ...
Reading analysis file: analysis-needed.txt
```

Data and Control Coupling Limitations and Error Checking

Combining Coupling and Coverage Instrumentation

Coupling Instrumentation can be performed on un-instrumented files. If you attempt to run the "coupling apply" command on a project that has files instrumented for some other type of coverage, such as Branch, MC/DC, Level A or Level B coverage, you will receive an error message prompting you to un-instrument the project. The following clicast command can be used to turn off coverage instrumentation:

```
clicast -lc -e <COVER_ENV> [-u <unit>] Cover INstrument None
```

Data Couples That Are Compound Types

When a data couple is a compound type such as an array or a structure, VectorCAST captures the field name or array element index expression for each access, and annotates the read and write lines of the Coupling Access Order Full Report with this information, as shown in this report fragment:

Name: *globalDataProvidedBySubSystem2_array*

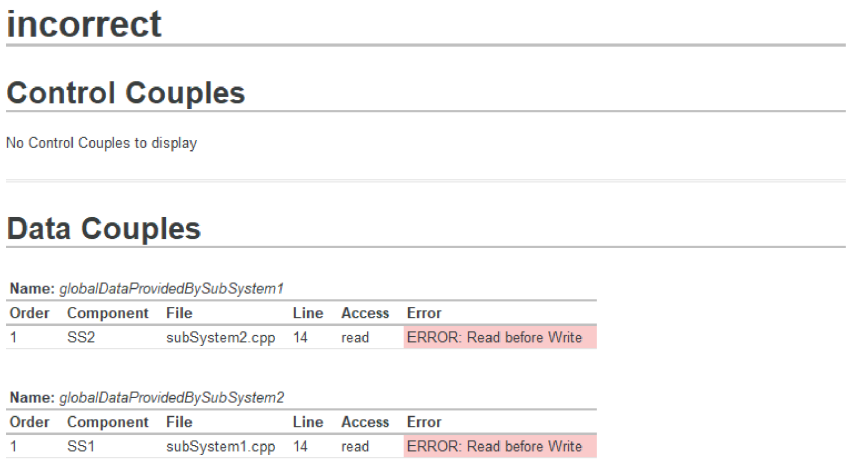
Order	Component	File	Line	Access
1	SS2	subSystem2.c	33	write ([4])
2	SS1	subSystem1.c	29	read ([4])

Name: *globalDataProvidedBySubSystem2_struct*

Order	Component	File	Line	Access
1	SS2	subSystem2.c	32	write (.field2)
2	SS1	subSystem1.c	31	read (.field2)

Data Couple Access Errors

The Coupling Access Order Error Report contains a summary section at the top of the report, and a Coupling Verification section for each test case. Data Couple Access errors are flagged in both places. If a data couple is "read" before it is written in any test case, VectorCAST flags that access in the Coupling Verification section for that test:



incorrect

Control Couples

No Control Couples to display

Data Couples

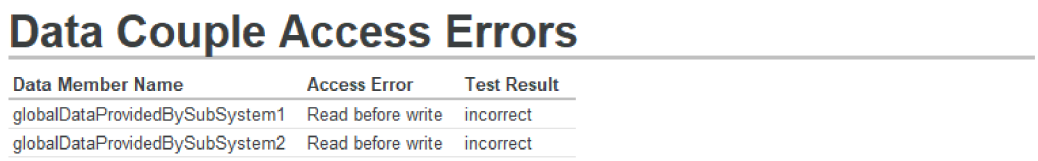
Name: *globalDataProvidedBySubSystem1*

Order	Component	File	Line	Access	Error
1	SS2	subSystem2.cpp	14	read	ERROR: Read before Write

Name: *globalDataProvidedBySubSystem2*

Order	Component	File	Line	Access	Error
1	SS1	subSystem1.cpp	14	read	ERROR: Read before Write

An entry is also added to the Data Couple Access Errors section of the report:



Data Couple Access Errors

Data Member Name	Access Error	Test Result
globalDataProvidedBySubSystem1	Read before write	incorrect
globalDataProvidedBySubSystem2	Read before write	incorrect

Note that for compound types, there can be an access error when a sub-component of that compound type (a field or array element) is written and a different sub-component is read. However, if an aggregate assignment is made to the data couple, then all read accesses of a sub-component are valid.

Prefixes used in Data Couple Access errors are:

- **ERROR** - Invalid access that needs to be corrected.
- **ANALYZE** - An invalid access potentially exists and needs further investigation to determine if the access was invalid or not.

Data Couple Access errors are described in detail below:

Error Message	Comments
`ERROR: Read before Write`	**Description:** A variable was read before ever being written. **Why it is relevant:** Catches invalid/undefined access of variables.
`ANALYZE: Read follows Initialization`	**Description:** A variable was initialized and then read, without an explicit write access in between. **Why it is relevant:** Catches accesses to variables that may have invalid/incomplete values. It is common to initialize variables to a default value, like 0, but in many cases these default values may not be valid when the data is later accessed. **Note:** In some code bases, variables may always be

Error Message	Comments
	initialized to valid values. In this case, flagging these accesses is unnecessary. To suppress this behavior, set the option VCAST_TREAT_DC_INITIALIZATIONS_AS_WRITE.
`ANALYZE: Read follows Address of`	**Description:** A possible Read before Write access error occurred; a variable's address was taken and then the variable was read, without an explicit write access in between. Once a variable's address is taken, VectorCAST cannot guarantee all further accesses will be reported. The user must determine if the variable was properly set before the read. **Why it is relevant:** Catches invalid/undefined access of variable data.

Function Pointers

Function pointer calls are not tracked for coverage by VectorCAST automatically. To start tracking the function pointer calls for coverage, manual analysis of the function pointer calls must be done.

As an example of this analysis process, consider the following function definitions:

```
189  int callViaFunctionPointerParameter (
190      int (*fPointerParam1)(void),
191      int (*fPointerParam2)(void)) {
192
193      int local1;
194      int local2;
195
196      local1 = fPointerParam1();
197      local2 = fPointerParam2();
198
199      return local1+local2;
200    }
201
202
203  int callViaFunctionPointers (int param) {
204
205      /* This will show up as called in the coverage report,
206         even though it is not called from this location. */
207
208      int (*fPointer1)(void);
209
210      fPointer1 = doSomeS1Stuff;
211
212      if (param<0)
213         return fPointer1();
214      else
215         return callViaFunctionPointerParameter (fPointer1, doSomeOtherS1Stuff);
216
217    }
```

In the example the functions **`doSomeS1Stuff()`** and **`doSomeOtherS1Stuff()`** are both control couples between components S1 and S2, and these functions are called via function pointers at lines 196 and 197. The VectorCAST Coupling Coverage report for this example looks like this:

```
------------------------------------------------------------
Function Pointer Analysis
------------------------------------------------------------

Function Pointer calls that require user analysis
   File: subSystem2.cpp line: 197 column: 14
   File: subSystem2.cpp line: 213 column: 15
   File: subSystem2.cpp line: 196 column: 14
```

Function pointer calls are listed within the “Function Pointer Analysis” section of the report. Function pointer calls listed in this section are “un-analyzed”, which indicates that VectorCAST is not tracking the calls for coverage until manual analysis is complete.

Control Couples Called via Function Pointers

When a function pointer is used to call a function, it is not possible to use static analysis to determine the actual function or functions that will be called at run time; and as a result, user analysis is required. As part of the static analysis for coupling, VectorCAST identifies every location in the application where function calls are made via a function pointer. For each function pointer call, the user must identify the possible functions that can be called by the function pointer. Once user analysis is done, VectorCAST performs coupling analysis on this information.

During execution, VectorCAST monitors the control couple established via the function pointer. If the function pointer calls the control couple, the function pointer call reference within the control couple is marked as covered. If the function pointer does not call the control couple and calls some other function, the function pointer call reference within the control couple is not marked as covered.

As an example, consider the following function, which makes two calls via function pointer:

```
189  int callViaFunctionPointerParameter (
190      int (*fPointerParam1)(void),
191      int (*fPointerParam2)(void)) {
192
193      int local1;
194      int local2;
195
196      local1 = fPointerParam1();
197      local2 = fPointerParam2();
198
199      return local1+local2;
200    }
```

During the normal analysis and instrumentation process for this example, these two locations are identified and reported in the Function Pointer section of the Coupling Coverage Report (see above examples).

The next step is to tell the tool which functions can be called by these function pointers. Do this by providing information to VectorCAST via the following steps:

1. Generate a **`user-analysis.txt`** file using the following clicast command:

```
clicast -lc -e <env> COUpling MAnual_analysis GEnerate_report <filename>

Create file to be used to perform manual analysis coupling data.
```

The following report is generated:

```
This file contains a list of coupling issues that require manual analysis

Function Pointer Analysis
Each FPTR line identifies a file, line, column where a call via function pointer is made.
If this FPTR has no relation to any Control Couple, change the status to: "ignore"
If this FPTR is used to call a Control Couple function, change the status to: "active"
and provide a list of space-separated function names for the target of this function pointer.
If the function is overloaded the mangled name must be used. A complete list of valid function
names has been created in the file: master-function-list.xmllocated in the "coupling"
sub-directory of the VectorCAST environment.

Example lines might look like this after your analysis:
...FPTR: foo.cpp:1, line: 234, col: 15, inst-line: 234, inst-col: 15, status: ignore, fp-targets:
...FPTR: bar.cpp:6, line: 123, col: 23, inst-line: 123, inst-col: 23, status: active, fp-targets: bozo bingo

FPTR: subSystem2.cpp:1, line: 196, col: 14, inst-line: 196, inst-col: 5, status: analyze, fp-targets:
FPTR: subSystem2.cpp:2, line: 197, col: 14, inst-line: 197, inst-col: 5, status: analyze, fp-targets:
FPTR: subSystem2.cpp:3, line: 213, col: 15, inst-line: 213, inst-col: 8, status: analyze, fp-targets:
```

These couples require manual analysis

2. Edit the **`user-analysis.txt`** file to update the status for each FPTR line, using one of the following:

- **Ignore** - Use if you don't want the FPTR to be tracked. For example, if the function pointer only calls functions from within the same file.
- **Active** - Use if you want the FPTR to be tracked. Note that you must provide a list of space-separated function names for the target of this function pointer. If the function is overloaded, the mangled name must be used. A complete list of valid function names is created in the file: **`master-function-list.xml`**, located in the **`coupling`** sub-directory of the VectorCAST environment.

Our example file can be updated as follows:

```
FPTR: subSystem2.cpp:1, line: 196, col: 14, inst-line: 196, inst-col: 5, status: active, fp-targets: doSomeS1Stuff
FPTR: subSystem2.cpp:2, line: 197, col: 14, inst-line: 197, inst-col: 5, status: ignore, fp-targets:
FPTR: subSystem2.cpp:3, line: 213, col: 15, inst-line: 213, inst-col: 8, status: ignore, fp-targets:
```

Function name provided for active FPTR target

All status updated

3. Pass the file back to VectorCAST using the following command:

```
clicast -lc -e <env> COUpling MAnual_analysis UPdate <filename>

    Import file to be used to perform manual analysis for coupling data.
```

4. Once the manual analysis is provided, use the following command to recompute and apply the coupling instrumentation logic and then re-run your tests.

```
clicast -lc -e <env> COUpling APply

    Add the function calls necessary to gather control coupling data
    during execution.
```

The Function Pointer Analysis section of the Coupling Coverage Report will now look like this:

Function Pointer Calls Needing Analysis

No un-analyzed calls in this application

The Coupling Coverage Details by Test Result section of the Coupling Access Order Full Report will now look like this:

Test Result: Example_Test

Control Couples

Name: *consumerOfSS1data* **Returns:** *int*

Order	Component	File	Line	Access
1	Main	callingSequences.cpp	24	call
2	Main	callingSequences.cpp	53	call

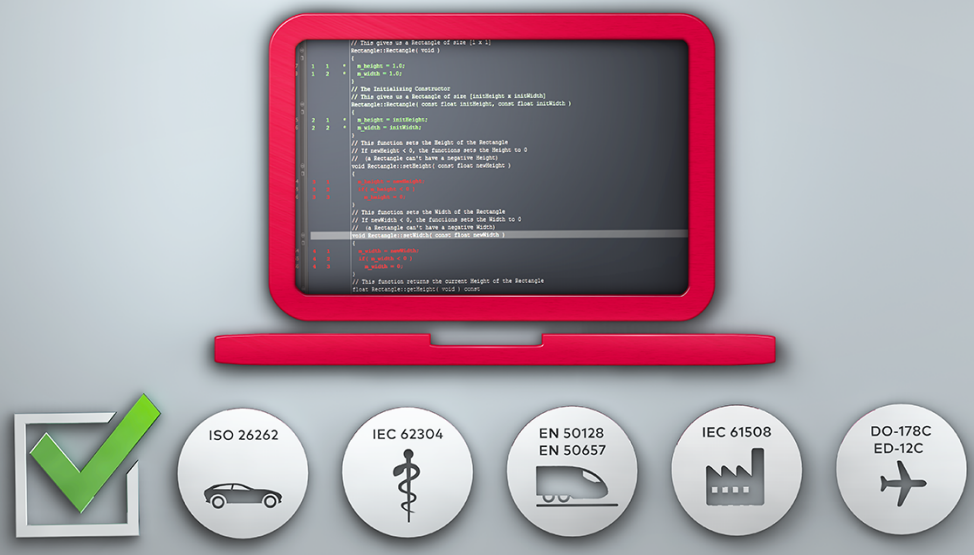
Name: *consumerOfSS2CompoundData* **Returns:** *int*

Order	Component	File	Line	Access
1	Main	callingSequences.cpp	36	call
2	Main	callingSequences.cpp	60	call

Name: *doSomeS1Stuff* **Returns:** *int*

Order	Component	File	Line	Access
1	SS2	subSystem2.cpp	63	call
2	SS2	subSystem2.cpp	196	fptr-call

Note that the **`fptr-call`** on line 196 is marked as covered. As you recall, we modified the analysis file in our example to set the status as "Active", and the function pointer called the target during execution.



VectorCAST/QA and CUDA® Integration

Using VectorCAST/QA with CUDA® Applications

VectorCAST/QA integrates with the NVIDIA® CUDA® development environment. VectorCAST is customized to handle both the CUDA C++ language extensions and the Multi-GPU compile and link workflow.



Note: You must have a graphics card that supports CUDA. For more information on the CUDA standard, see www.nvidia.com/cuda.

CUDA Language Extension Support

Most CUDA language extensions are handled with parser configuration flags that have been built into VectorCAST. The one exception is the **<<<kernel launch>>>** syntax, which is handled by a pre-processor script that converts these calls into a macro that can be parsed more easily by VectorCAST. The macro expansion that occurs when building the instrumented source restores the original syntax.

Multi-GPU Build Process Support

When the CUDA compiler (nvcc) compiles CUDA source, the user specifies one or more device (GPU) architectures for which object code generation is desired. There are several ways to accomplish this, but providing one or more **-gencode** options to nvcc is most often used.

The source file is then processed multiple times: once for each device (GPU) architecture, and once for the host (CPU) architecture. These code variants are passed to the underlying preprocessor/compiler with additional compilation flags to specify which variant is being compiled. In particular, nvcc sets the **__CUDA_ARCH__** macro to force processing for a particular device (the absence of this macro indicates host processing). The resulting object files are then linked together into a "fat binary" that contains object code for the host processor as well as multiple GPU architectures.

This process is typically handled by nvcc automatically, with little or no need for the developer to understand the mechanism. However, because each variant may contain different code (and thus different code coverage), VectorCAST must treat each architecture variant as an individual source file with an independent coverage set. This is accomplished by a set of Python scripts that are used to prepare (split) CUDA source for processing by VectorCAST. Prepared code variants are added to a VectorCAST project with unit options that mirror nvcc's internal pre-processing logic. This ensures that VectorCAST's instrumentation matches the pre-processing mechanism employed by nvcc.

CUDA Processing Scripts

The process of preparing CUDA source for VectorCAST involves the following operations:

1. Copy the original source code once for each architecture variant.
   - There will be one variant for each GPU device architecture and one for the CPU host.
   - Variants may be specified manually as script arguments, or can be derived automatically from compiler build options (e.g. **-gencode** flags).
2. Replace the original source file with an aggregator that uses **#ifdef** and **#include** logic to selectively include each variant copy during compilation.
   - This aggregator is designed such that pre-processing the aggregator produces equivalent

source code to the original.

- The original source is backed up prior to replacement.

Once this preparation is performed, each variant copy is added to a VectorCAST project with specific unit options. These options direct VectorCAST and the underlying nvcc pre-processor to process the variant for just a single architecture.

The overall effect is that the VectorCAST project will contain multiple variant copies of the original source, and each variant is processed for just one host/device architecture. This results in a single VectorCAST project that can track and report coverage from all architectures simultaneously.

Building

After CUDA source has been prepared with the VectorCAST CUDA processing scripts, the aggregator file may be built using the original build system. The aggregator file resides at the original source location; no changes to the build system are typically necessary. Note, however, that if you do not elect to append the **`c_cover_io.c`** source to one of your host-side source files (either a host variant of a CUDA source file, or a C or C++ file) then the build system will need to be updated to compile and link that additional source.

Partial Instrumentation

The aggregator unit is configured such that it may be built at any time, regardless of whether the source code variants have been instrumented or not.

Architecture variants may be independently instrumented without interfering with one another. If one or more variant copies have been instrumented, that instrumentation will be included in the object code for the associated architecture.

For example, if **`source.cu`** is built for CUDA architectures 3.5, 5.3 and 7.0 (as well as the CPU host), any combination of the four variant versions of the source may be instrumented without issue. During runtime, you will only receive code coverage data for execution on instrumented architectures.

VectorCAST CUDA® Example

This example provides an overview of using VectorCAST/QA with the CUDA programming language.

This example and its associated file (**`KB_1476_cuda-demo.zip`**) are located online in the Vector KnowledgeBase article VectorCAST CUDA Example .

To run the example:

1. Review and update **`setup_env.sh`**. This script contains environment variables and settings that configure the example for your specific environment.

   Verify that VECTORCAST_DIR and VECTOR_LICENSE_FILE are set appropriately for your system to access VectorCAST tools, and the CUDA_PATH variable is set to the base of your CUDA tools installation.

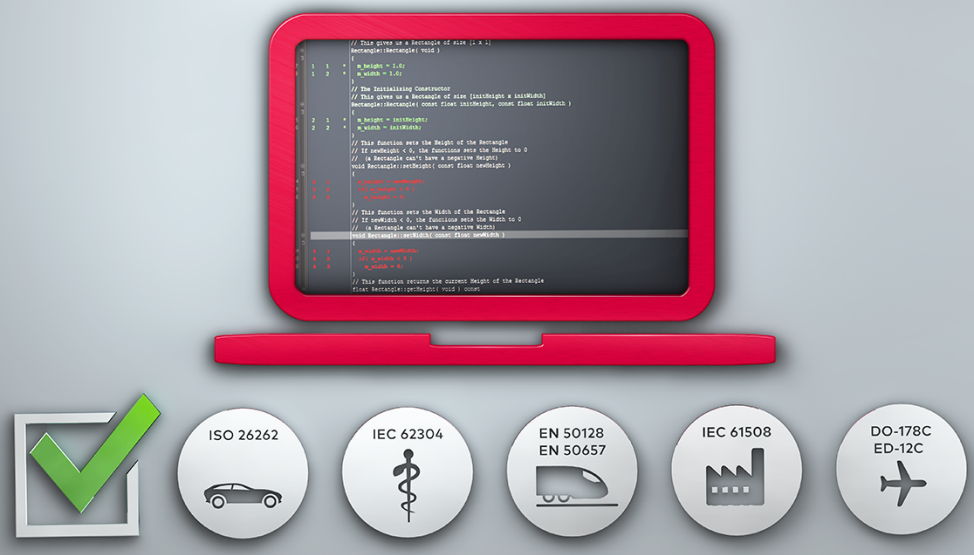
   The example is configured to produce code for GPUs with compute capabilities for CUDA architectures 3.5, 5.3, and 7.0. If your system's GPU uses a different architecture, then adjust the SMS variable appropriately.

   If you are executing on a remote ARM target (for example, using a Jetson device), then set TARGET_ARCH appropriately.
2. Execute the **`run_example.sh`** script to build and run the example. The script is set to provide detailed logging, and stop on any failures. If there are any issues, please review the execution log and script to determine the source of the failure.

The example execution script will:

1. Create a directory named **`example`** in the current working directory. All subsequent files will be created within **`example`**. (To clean the example, simply remove the **`example`** folder.)
2. Execute (source) **`setup_env.sh`** to import environment settings.
3. Create a CCAST_.CFG settings file.
4. Prepare build directory with source files.
5. Execute the build with the vcshell utility to capture build commands.
6. Locate and prepare CUDA source files using the CUDA workflow scripts (located in **`$VECTORCAST_DIR/DATA/cuda`**).
7. Create a coverage project for the prepared source variants.
8. Apply per-file unit options, coverage I/O library appending, and coverage capture probe points.
9. Instrument included source files.
10. Build the instrumented binary.
11. Execute the instrumented image.
12. Retrieve and import coverage results.
13. Produce a coverage report.

After the script completes, you may open the created coverage project (**`*.vcp`**) to inspect results, review the configuration, etc.



VectorCAST/QA and Jenkins Integration

VectorCAST/QA and Jenkins Integration

QA and Jenkins integration enables users to implement a Continuous Test process, where each code change is completely tested against all existing test cases prior to integration. This results in the shortest possible feedback loop for the developer, allowing bugs to be identified within minutes of code changes, rather than weeks.

Jenkins is an extensible, open source automation server for continuous integration and continuous delivery for any application. Two plugins are available from the Jenkins website for use with VectorCAST projects: VectorCAST Execution and VectorCAST Coverage.

The VectorCAST Execution plugin allows the user to create, delete and update jobs to build and run VectorCAST projects. Jobs can be created as a single job or split into multiple jobs for a VectorCAST project, with one job for each environment and an overall job to combine the results. The plugin adds a new top-level menu item to the Jenkins sidebar that provides job control for VectorCAST projects. To learn more about the Execution plugin, visit the Jenkins website at https://plugins.jenkins.io/vectorcast-execution.

The VectorCAST Coverage plugin allows the user to capture code coverage reports from VectorCAST projects. The VectorCAST Coverage plugin is added as a dependency to the VectorCAST Execution plugin and is automatically used to display coverage data. Jenkins automatically generates the trend report of coverage and displays a coverage trend graph at the top page of a job. To learn more about the Coverage plugin, visit the Jenkins website at https://plugins.jenkins.io/vectorcast-coverage.

General information on plugin installation is provided on the Jenkins' wiki at https://wiki.jenkins.io/display/JENKINS/Plugins.

Index

*
    indicating a statement covered 127
.enc 85
ADA_EXTENSIONS 229
adding multiple test results 119
aggregate coverage report
    instantiations 205
allowlist directory 62, 71
animated control flow 151
animation coverage type 190
animation play speed 194
apply coverage to source tree 40
    benefits 40
    changing option 42
    how it works 40
    strategies 40
apply instrumentation to source tree
    always option 41
    during build option 41
    never option 42
assembler command 186
ASSEMBLER_CMD 186
assembly file extensions 178
auto naming multiple test results 119
AUTO_NAME_RESULTS 227
background color for failing cells 238
background color for partially passing cells 238
background color for passing cells 237
background colors
    changing for coverage output 225
base directory
    add 59, 68
    allowlist 62, 71
    include headers 63, 72
    missing 92
    move 92
    path 61, 70
    remove 61, 70
    rename 60, 69
    working with 92
basis path
    viewing for subprogram 138
basis path analysis 139
basis path output
    building test cases from 139
basis paths coverage 103
Branch coverage 103
breakpoints
    removing 154
    setting in coverage animation 154
buffered coverage I/O 190
building environment, in wizard 65, 74
C compile command 184
C file extensions 178
C parser flags 184
C preprocessor command 183
C standard library
    link error using 167
C_ALT_COMPILE_CMD 184
C_ALT_EDG_FLAGS 184
C_ALT_PREPROCESS_CMD 183
C_COMPILE_CMD 164
C_COMPILE_CMD_FLAG 185
C_COMPILE_EXCLUDE_FLAGS 185
C_COMPILER_CFG_SOURCE 186
C_COMPILER_FAMILY_NAME 186
C_COMPILER_OUTPUT_FLAG 164
C_COMPILER_PY_ARGS 186
C_COMPILER_TAG 162
C_DEBUG_CMD 175
C_DEBUG_HELP_FILE 185

C_DEFINE_LIST 166
C_EDG_FLAGS 179
C_EXEC_HELP_FILE 185
C_EXTENSIONS 179
C_INCLUDE_FLAG 163
C_LINK_CMD 174
C_LINK_OPTIONS 174
C_OBJECT_EXT 174
C_OUTPUT_FLAG 173
C_PREPROCESS_CMD 163
C_PREPROCESS_FILE 164
C++ file extensions 178
cannot uninstrument 111
CBA
    aggregate coverage report 256
    associate imported files with source files 251
    change cba display color 252
    covered by analysis report 253
    edit analysis result 250
    export analysis files 253
    import analysis files 253
    metrics report 257
    re-instrumenting with cba data 253
    remove analysis files 253
    using coverage analysis with sfp 257
    view cba data in reports 253
    view cba in coverage viewer 251
cba editor
    branch coverage 248
    mcdc coverage 249
    statement coverage 247
change impact report 50
CLI
    execute all system tests 45
    execute component coverage 48
    list system tests 45
CLICAST
    coupling analyze 292, 295
    coupling apply 292, 312
    coupling components 291
    coupling report access order full 300, 302
    coupling report components 296
    coupling report coverage 294, 298
    cover base dir add 60, 69, 93
    cover base dir allowlist 62, 71
    cover base dir denylist 63, 72
    cover base dir remove 61, 70
    cover base dir rename 61, 70
    cover base dir set path 93
    cover base dir status 74-75, 82, 85
    cover base dir update 85
    cover environment create 68
    cover environment script create 86
    cover environment script run 86
    cover export cba 253
    cover instrument 100
    cover instrument basis_path 104
    cover instrument branch 103
    cover instrument mcdc 105
    cover instrument none 110
    cover options
        add_ada_extension 229
        add_c_extension 179
        add_cpp_extension 179
        auto_generate_result_name 227
        Starting_unit_number 228
    cover report cba 256
    cover report ram usage 109
    cover result add 119, 293
    cover result remove 133
    cover source reinstrument 112

environment delete 86
IF DOCPROPERTY Project COVER 158
IF DOCPROPERTY Project COVER 134
manual analysis generate report 311
manual analysis update 311
options
ASM_FUNCS_BEHAVE_AS_INLINES 214
ASSEMBLER_CMD 186
C_ALT_COMPILE_CMD 184
C_ALT_EDG_FLAGS 184
C_ALT_PREPROCESS_CMD 183
C_COMPILE_CMD 164
C_COMPILE_CMD_FLAG 185-186
C_COMPILE_EXCLUDE_FLAGS 185
C_COMPILER_OUTPUT_FLAG 164
C_COMPILER_PY_ARGS 186
C_DEBUG_CMD 176
C_DEFINE_LIST 167
C_EDG_FLAGS 179
C_EXEC_HELP_FILE 185-186
C_INCLUDE_FLAG 163
C_LINK_CMD 174
C_LINK_OPTIONS 174
C_OBJECT_EXT 174
C_OUTPUT_FLAG 174
C_PREPROCESS_CMD 163
C_PREPROCESS_FILE 164
CASE_FALLTHROUGH_BRANCH_COVERAGE 198
COVERAGE_ANIMATION_PLAY_SPEED_RATIO 194
COVERAGE_IO_TYPE 190
EXECUTABLE_EXTENSION 174
LIBRARY_INCLUDE_DIR 165
Max_mcdc_conditions 210
METRICS_TOTAL_LEVEL_DISPLAY 234
OPTIONS_Append 223
PRECOMPILE_CMD 187
PRECOMPILE_EXT 187
SOURCE_EXTENSION 177
TESTABLE_SOURCE_DIR 165
TYPE_HANDLED_SOURCE_DIR 166
VCAST_ASSEMBLY_FILE_EXTENSIONS 179
VCAST_AVOID_COMMA_OPERATOR 203, 208
VCAST_BASIS_PATHS_FOR_CONSTANT_BRANCHES 203
VCAST_C_FILE_EXTENSIONS 178
VCAST_CMD_AFTER_C_OPTIONS 216
VCAST_CMD_AFTER_INSTRUMENTATION 216
VCAST_COLLAPSE_STD_HEADERS 182
VCAST_COMMAND_LINE_DEBUGGER 176
VCAST_COVER_CATCH_AS_BRANCH 201
VCAST_COVER_SEPARATE_TYPES_FILES 228
VCAST_COVER_STATEMENTS_BY_BLOCK 197
VCAST_COVERAGE_DATABASE_CACHE_SIZE 213
VCAST_COVERAGE_FIELD_WIDTH 213

VCAST_COVERAGE_FOR_
TEMPLATE_
INSTANTIATIONS 205

VCAST_COVERAGE_POINTS_AS_
MACROS 203

VCAST_COVERAGE_RESULT_FILE_
NAME 221

VCAST_COVERAGE_RESULTS_
DIRECTORY 221

VCAST_COVERAGE_SOURCE_FILE_
PERSPECTIVE 190

VCAST_CPP_FILE_EXTENSIONS 178

VCAST_CUSTOM_REPORT_
FORMAT 238

VCAST_DISABLE_BOOLEAN_
CAST 212

VCAST_DISPLAY_EMPTY_
COVERAGE 231

VCAST_DISPLAY_UNINST_EXPR 231

VCAST_DUMP_BINARY_
COVERAGE 201

VCAST_DUMP_COVERAGE_AT_
EXIT 192

VCAST_EMPTY_STATEMENTS_
COVERABLE 222

VCAST_ENABLE_DATA_CLEAR 192

VCAST_ENABLE_FUNCTION_AND_
CALL_COVERAGE 197

VCAST_ENABLE_FUNCTION_CALL_
COVERAGE 196

VCAST_ENHANCED_FOR_LOOP_
COVERAGE 195

VCAST_ENVIRONMENT_FILES 183

VCAST_ESCAPE_LINE_
DIRECTIVES 181

VCAST_FILE_VERSION_
COMMAND 235

VCAST_GH_INTEX_CMD 175

VCAST_GH_INTEX_FILE 175

VCAST_GLOBAL_BUFFER_
OPTIMIZATIONS 199

VCAST_HANDLE_TERNARY_
OPERATOR 222

VCAST_HEADER_FILE_
EXTENSIONS 177

VCAST_IGNORE_ERROR_WHEN_
READ_FOLLOWS_ADDRESS_
OF 204

VCAST_INSTRUMENT_
ASSIGNMENTS 222

VCAST_INSTRUMENT_
PARAMETERS 222

VCAST_INSTRUMENT_
SEPARATES 223

VCAST_INSTRUMENT_
UNREACHABLE_CODE 196

VCAST_KEEP_INSTRUMENTATION_
DEBUG_FILES 215

VCAST_KEEP_INSTRUMENTATION_
INTERMEDIATE_FILES 214

VCAST_MAX_CAPTURED_ASCII_
DATA 191

VCAST_MAX_COVERED_
SUBPROGRAMS 193

VCAST_MAX_MCDC_
STATEMENTS 194

VCAST_NOTES_SECTION_
TEMPLATE 236

VCAST_OPTIMIZED_MCDC_
STORAGE_THRESHOLD 211

VCAST_PACK_INSTRUMENTATION_
STORAGE 199

VCAST_PARSE_FUNCTION_
TEMPLATES 180

VCAST_POST_PREPROCESS_COMMAND 182
VCAST_PREPROCESS_PREINCLUDE 181
VCAST_REMOVE_PREPROCESSOR_COMMENTS 180
VCAST_REPORT_ON_IMPLICIT_DEFAULT_CASE 198, 223
VCAST_RPTS_COMPLEXITY_COLUMN_WIDTH 241
VCAST_RPTS_COVERAGE_RESULT_COLUMN_WIDTH 241
VCAST_RPTS_CUSTOM_CSS 239
VCAST_RPTS_DATE_COLUMN_WIDTH 240
VCAST_RPTS_HEADER 236
VCAST_RPTS_NOTES_COLUMN_WIDTH 241
VCAST_RPTS_RESULT_COLUMN_WIDTH 240
VCAST_RPTS_SHOW_VERSION 235
VCAST_RPTS_SUBPROGRAM_COLUMN_WIDTH 240
VCAST_RPTS_TABLE_DATA_FAIL_BGCOLOR 238
VCAST_RPTS_TABLE_DATA_PARTIAL_BGCOLOR 238
VCAST_RPTS_TABLE_DATA_PASS_BGCOLOR 238
VCAST_RPTS_TESTCASE_COLUMN_WIDTH 240
VCAST_RPTS_UNIT_COLUMN_WIDTH 239
VCAST_SHOW_INLINES_COVERED_IN_ALL_UNITS 202
VCAST_SIMPLIFIED_CONDITION_COVERAGE 209
VCAST_SORT_METRICS_RPT_BY_DIR 230, 232
VCAST_STARTUP_FILE 188
VCAST_SUPPRESS_COVERABLE_FUNCTIONS 218
VCAST_TREAT_DC_INITIALIZATIONS_AS_WRITE 204
VCAST_UNCOVERED_LINE_INDICATOR 194
VCAST_UNSIGNED_LONG_MCDC_STORAGE 209-210
VCAST_USE_BUFFERED_ASCII_DATA 191
VCAST_USE_EDG_PREPROCESSOR 181
VCAST_USE_OPTIMIZED_MCDC_INSTRUMENTATION 211
VCAST_USE_STATIC_MEMORY 193
VCAST_USE_STDOUT 222
VCAST_USE_VCPP 181
VCAST_VCAST_MASKING_MCDC 209
VCAST_VERBOSE_MANAGEMENT_REPORT 234
probe point apply 271
probe point output 281
probe point remove all 275
reports custom equivalence_matrices 141
reports custom function call 137
set coverage type 99
set_source_dirs 166
template 162
tools basis_path 138, 144
tools delete imported results 159
tools import_results 158

clicast
options
VCAST_ENABLE_PROBE_POINTS 266
closing environment 80
code complexity 138
code coverage summary 147
collapse all subprograms 132
combined template instantiations 206
command line debugger 176
compile command 164
compile command flag 185
compile command line
parse for library includes and defined variables 167
compile flag to exclude 185
compiler
testing settings of 170
compiler family name 186
compiler python args 186
compiler template
choosing between C and C++ 162
compiler, setting in wizard 57, 66
compiling instrumented source files 113
component coverage 46
component report 296
configuration management 75
configure binary coverage 200
configure language 57, 66
control flow
animated 151
coupling
access order error report log 306
access order full report log 306
analysis log 304
apply coupling 292
build instrumented application 292
combining coupling and coverage instrumentation 307
components.xml 290
compound type data couples 307
control couples called via function pointers 310
coupling analysis 291
coupling options 295
coupling overview 288
coupling report log 306
cover environment 290
data couple access errors 307
define components 288, 290
function pointers 309
instrumentation log 305
limitations and error checking 307
logs 303
re-analyze coupling 295
reports 293
trace data 293
verification process 289
coupling access order error report 301
coupling access order full report 300
coupling coverage report 297
coupling reports 295
component report 296
coupling access order error report 301
coupling access order full report 300
coupling coverage report 297
cover environment script 85
coverage
animation play speed 194
changing the type of 109
compiling instrumented source files 113
control instrumentation with pragmas 113
deleting imported results 158
exporting 157

importing 158
importing from VectorCAST 154
instrument unreachable code 195
instrumenting basis paths 103
instrumenting Branch 103
instrumenting function 107
instrumenting Function+Function Call 108
instrumenting ISO-26262 ASIL A 101
instrumenting ISO-26262 ASIL A/B 107
instrumenting ISO-26262 ASIL B/C 101, 107
instrumenting ISO-26262 ASIL C/D 108
instrumenting ISO-26262 ASIL D 101, 106
instrumenting MC/DC 104
instrumenting statement 101
instrumenting Statement 101
instrumenting Statement+BRANCH 107
instrumenting Statement+MC/DC 106
not instrumenting sections of source code 112
opening test case which caused 132
pairs status 104, 135
show true and false in for loops 195
summary in upper right corner of Coverage tab 125
uncovered line indicator 194
un-instrumenting 110
viewing 133
coverage analysis editor 246
coverage animation
setting breakpoints 154
toolbar 152-153
Coverage menu
Export Script 157
Import Results 154
View Import Log 156
coverage options, setting in wizard 63, 72
coverage status bars 99
coverage viewer
annotation 126
branch coverage 128
customizing display 132
function coverage 130
mc/dc coverage 128
numbers along left-hand side 127
open 125
results navigation 131
statement coverage 127
turn on results 125
viewing cba lines 251
coverage wizard
building environment 65, 74
configure language 57, 66
coverage options 63, 72
defined variables 57, 67
locating source files 59, 68
name the environment 58, 67
search directories 57, 66
set compiler 57, 66
source file extensions 57, 66
standalone 65
summary 64, 73
COVERAGE_IO_TYPE 190
covered by analysis
see CBA 246
covered by analysis report 253
covered lines 223
covered-by-analysis lines 224
CPP_EXTENSIONS 179
create new environment
from a JSON Compilation Database 79
CUDA 314
example 316

custom end macro 167
customizing Coverage Viewer 132
debug help file 185
debugger command 175
default compiler options 186
default search list for wizard 162
defined variable 166
    parse out of compile command line 167
defined variables, setting in wizard 57, 67
deleting an environment 86
disable boolean cast in instrumentation 212
display coverage in reports (always) 231
display uninstrumented mcdc expressions in reports 231
EDG parser 179
enable the coverage clear api 192
environment
    closing 80
    deleting 86
    opening 87
    renaming 80
    source file summary 83
    update 81
    update source files 82
    view 133
environment coverage type 97
Environment menu
    Add Source Code File 87
    Add Test Result 118
    Delete All Result Files 133
    Delete All Source Files 92
    Reinstrument All Source 111
    View
        Metrics Report 104, 134
        Test Case Management Report 104
environment variables
    VCAST_COVER_SEPARATE_TYPES_FILES 228
    VCAST_MAX_LINE_LENGTH 113
equivalence matrices 141
escape backslashes in line directives 181
executable file extension 174
EXECUTABLE_EXTENSION 174
execute help file 185
expand all subprograms 132
extensions
    Ada 229
    Add/Remove/Default 179, 229
    C/C++ 179
file extensions, setting in wizard 57, 66
File menu
    Close Environment 80
    Delete Environment 86
    Open 80
    Print
        Aggregate Report 144
    Recent Environments 80, 87
    Set Working Directory 54, 80
file version command 235
files
    c_cover_io.c 114
    c_cover_io.cpp 114
    environment_name.cvr 75
    vcast_cover_io.adb 114
    vcast_cover_io.ads 114
    vcast_cover_io_ada83.adb 114
    vcast_cover_io_gmart.adb 114
    vcast_cover_io_mixed_lang.adb 114-115
    vcast_cover_io-restricted.adb 114-115
files tab
    function call report 137

filter Metrics report 232
fonts
    changing for coverage output 224
    default for coverage output 224
FUNCTION 107
function
    VCAST_CLEAR_COVERAGE_DATA 274
    VCAST_DUMP_COVERAGE_DATA 274
    vcast_probe_print 273
    vcast_probe_print_float 273
    vcast_probe_print_int 274
    vcast_probe_print_unsigned 274
function call report 137
function coverage 107
function entry probe points 262
function exit probe points 263
FUNCTION+FUNCTION CALL 108
Function+Function Call coverage 108
gcc 180
gnatchop instrumented source files 114
header file extensions 177
headers
    include in environment 63, 72, 89
Help menu
    Diagnostic
        Test Compiler Settings 173
importing coverage results 154
include directories 164
include flags 163
include headers 63, 72
incremental rebuild report 244
instrument template instantiations 204
instrument unreachable code 195
instrumentation
    ram usage report 109
instrumented files
    apply coverage to source tree option 40
instrumenting source files 97
jenkins
    coverage plugin 318
    execution plugin 318
    integration with enterprise testing 318
JSON Compilation Database
    create new environment in wizard 79
language mode 177
Level A coverage 101, 106
Level A/B coverage 107
Level B coverage 101, 107
Level C coverage 101
Level C/D coverage 108
library include directories
    parse out of compile command line 167
linker command 174
linker options 174-175
manual testing 37
map line selection 132
max size of ASCII coverage data buffer 191
maximum MC/DC Conditions 210
MC/DC 104
    maximum conditions 210
    understanding analysis 142
    viewing 144
messages
    after exporting coverage script 157
    after importing coverage results 155
metrics report 134
METRICS_TOTAL_LEVEL_DISPLAY 232
migrated cover environment 56
missing base directory 92
mixed-language environment 183-184
mixing C and C++ source files in one environment 178, 183

moving base directory
environment variable 96
full path 92-93
relative path 97
name the environment 58, 67
new generation preprocessor 181
no recursive calls
support for 115
NO_FILTERING 232
notes section template 235
Notes tab 234
object file extension 174
opening
environment 87
options
always display coverage in reports 231
animation coverage type 190
animation play speed 194
assembler command 186
assembly file extensions 178
buffered coverage I/O 190
C compile command 184
C compile command flag 185
C compile flag to exclude 185
C file extensions 178
C parser flags 184
C preprocessor command 183
C/C++ 161
compile command 164
linker options 174-175
C++ file extensions 178
change background color 225
change fonts 224
change text color 224, 226
command line debugger 176
compiler args 186
compiler family name 186
Compiler Integration tab 184
compiler template 162
coverage 188
default fonts 224
coverage field width 212
coverage I/O types 190
covered lines 223
covered-by-analysis lines 224
custom end macro 167
debug help file 185
debugger command 175
default compiler 186
defined variables 166
deprecated
C_INCLUDE_LIST 188
disable boolean cast in instrumentation 212
display uninstrumented mcdc expressions in reports 231
EDG flags 179
enable the coverage clear api 192
escape line directives 181
executable file extension 174
executehelp file 185
file version command 235
filter Metrics report 232
header file extensions 177
include flags 163
instrument unreachable code 195
language mode 177
Language tab 176
library include directory 165
linker command 174
Linker/Debug tab 173
max size of ASCII coverage data buffer 191
maximum covered subprograms 193

maximum MC/DC conditions 210
maximum MC/DC expressions 193
Misc tab 180
Mixed C/C++ tab 183
notes section template 235-236
object file extension 174
output file flags 173
parse function templates 179
partially covered lines 223
post-process instrumented files command 215
post-process vcast_c_options.h command 216
precompile command 187
precompile extension 187
preprocessor command 163
preprocessor file 164
Preprocessor/Compiler tab 162
real-time coverage I/O 190
reduce memory overhead using global buffers 199
remove preprocessor comments 180
report 229, 236
report complexity column width 240
report coverage results column width 241
report date column width 240
report format 238
report header 238
report notes column width 241
report results column width 240
report subprogram column width 239
report testcase column width 240
report unit column width 239
save data in ASCII format in memory 190
setting custom css 238
show inlines as covered in all unit 201
show true and false in for loops 195
show version of VectorCAST in reports 234
sort Metrics report by directory 230-231, 234
testable source directory 165
text report options 239
treat c++ catch blocks as branches 201
type handled source directory 165
uncoverable lines 224
uncovered line indicator 194
uncovered lines 224
use new generation preprocessor 181
use static memory allocation on the target 192
use Vector preprocessor 181
verbose management report 234
wrap text in Notes 241
original source code line numbers 132
output file flags 173
pairs status 104, 135
Parse Command Line button 167
parse function templates 179
parser 177
partially covered lines 223
pragma restrictions
support for no recursive calls 115
precompile command 187
precompile extension 187
PRECOMPILE_CMD 187
PRECOMPILE_EXT 187
preprocess file
diagnostic uses of 173
preprocessor command 163
preprocessor file 164
probe point listing 279
probe points 261
activate probe point 274
apply all 271

capture variable values 277
deactivate probe point 274
delete all probe points 275
disable 285
edit a probe point 272
enable 285
fault injection 278
function entry probe points 262
function exit probe points 263
function probe points 261
functions and macros 273
import a probe point 283-284
inject spurious values 278
insert a file scope probe point 267
insert a probe point 266
order of execution 264
output report 280
patch faulty code 278
probe id number 282
probe point api 281
probe point editor 265
probe point file 281
probe point only instrumentation 275
remove 285
remove probe point 275
saving and applying 270
status buttons 272
test compile 268
xml report 285
python configuration script 35
add test case 36
QA
change impact report 50
component coverage 46
enable/disable coverage 42
incremental build and execute 42
interactive execution of test cases 43
manual execution of tests 43
manual testing 37
python configuration script 35
run incremental build/execute 43
select system test cases dialog 43
ram usage report 109
real-time coverage I/O 190
regression test 75
remember last working directory 54
remove preprocessor comments 180
removing
breakpoints 154
removing source files 92
renaming
environment 80
report complexity column width 240
report coverage results column width 241
report date column width 240
report format 238
report header 238
report notes column width 241
report results column width 240
report subprogram column width 239
report testcase column width 240
report unit column width 239
reports
metrics 134
save to files 145
requirements
script 124
selecting 124
requirements gateway
importing .csv 123
requirements tab 123

results files
not corresponding 109
right-click menu
Flatten 90
Remove 92
Save Coverage Reports 145
uninstrument 110
Update Test Result 120
save coverage reports to files 145
save data in ASCII format in memory 190
scripting
exporting for regression test 157
search directories, setting in wizard 57, 66
set working directory 54
setting source directories 166
show inlines as covered in all units 201
show true and false in for loops 195
show version of VectorCAST in reports 234
sort Metrics report by directory 230-231, 234
source directories
setting, command line 166
source file extensions, setting in wizard 57, 66
source files
adding 87
compiling instrumented 113
hierarchical view of 90
instrumenting 97
removing 92
setting coverage type 97
viewing instrumented source code 101
viewing source code 91
source files, locating in wizard 59, 68
SOURCE_EXTENSION 177
standalone coverage environment 65
startup file 187
statement coverage 101
Statement coverage 101
STATEMENT+BRANCH 107
Statement+Branch coverage 107
STATEMENT+MC/DC 106
Statement+MC/DC coverage 106
SUBPROGRAM_DETAIL 232
suppress test code 218
directory 218
file 219
header only libraries 220
inlined 219
suppressed coverable functions 217
system testing environment 56
T
indicating a branch covered 128
template
for Notes tab 235
for user code 235
template instantiations 204
combined 206
requirements 204
view coverage 205
Test button 172
test cases
building from basis path analysis 139
test code coverage
suppressing 218
test execution
animated 151
test results
auto naming multiple when adding 119
customizing c_cover_io for your target 115
naming one when adding 119
using defined variables to change output of 116
test runner 38
Test Settings button 170

text colors
changing for coverage output 224, 226
toolbar
coverage animation 152-153
Open button 87
Tools
MC/DC
View Analysis Report 144
Tools menu
Basis Path
View Analysis Report 138
Coverage
Delete Imported Results 158
Import Script 158
MC/DC
View Equivalence Matrices 141
treat c++ catch blocks as branches 201
troubleshooting
error during uninstrumenting 111
exit is undefined 167
instrumented source file was locked 111
spaces in working directory 55
suppressing some statement in MC/DC coverage 105
true branch 128
uncoverable lines 224
uncovered line indicator 194
reinstrument 194
uncovered lines 224
undefined symbol VCAST_exit 167
UNIT_DETAIL 233
update
compiler 81
compiler settings 81
environment 81
locate source files 82
rules for updating environment 84
source file summary 83
user code template 235
V(g) 138
VCAST_APPEND_POSIX_PID 117
VCAST_APPEND_SECONDS_SINCE_EPOCH 118
VCAST_APPEND_WIN32_PID 117
VCAST_ASSEMBLY_FILE_EXTENSIONS 178
VCAST_ATTRIBUTE_CODE 117
VCAST_C_FILE_EXTENSIONS 178
VCAST_CLEAR_COVERAGE_DATA 274
vcast_clear_coverage_data() 192
VCAST_COMMAND_LINE_DEBUGGER 176
VCAST_COVER_SEPARATE_TYPES_FILES 228
VCAST_COVERAGE_FOR_DECLARATIONS 202
VCAST_COVERAGE_IO_ANIMATION 190
VCAST_COVERAGE_IO_BUFFERED 190
VCAST_COVERAGE_IO_REAL_TIME 190
VCAST_CPP_FILE_EXTENSIONS 178
VCAST_CUSTOM_FILE_OUTPUT 116-117
VCAST_CUSTOM_REPORT_FORMAT 238
VCAST_CUSTOM_STD_OUTPUT 115, 117
VCAST_DISPLAY_EMPTY_COVERAGE 231
VCAST_DISPLAY_UNINST_EXPR 231
VCAST_DO_MCDC 105
VCAST_DONT_DO_MCDC 105
VCAST_DONT_INSTRUMENT_END 112
VCAST_DONT_INSTRUMENT_START 112
VCAST_DUMP_CALLBACK 118
VCAST_DUMP_COVERAGE_DATA 274
VCAST_ENABLE_DATA_CLEAR_API 192
VCAST_ENABLE_PROBE_POINTS 266
VCAST_ESCAPE_LINE_DIRECTIVES 181

VCAST_exit, undefined symbol 167
VCAST_FILE_VERSION_COMMAND 235
VCAST_FLUSH_DATA 117
VCAST_GH_INTEX_CMD 175
VCAST_HEADER_FILE_EXTENSIONS 177
VCAST_MAX_COVERED_SUBPROGRAMS 117
VCAST_MAX_LINE_LENGTH 113
VCAST_MAX_MCDC_STATEMENTS 117
VCAST_MIN_STARTING_UNIT_NUMBER 228
VCAST_NOTES_SECTION_TEMPLATE 235
vcast_probe_print 273
vcast_probe_print_float 273
vcast_probe_print_int 274
vcast_probe_print_unsigned 274
VCAST_REMOVE_PREPROCESSOR_COMMENTS 180
VCAST_RPTS_COMPLEXITY_COLUMN_WIDTH 240
VCAST_RPTS_COVERAGE_RESULT_COLUMN_WIDTH 241
VCAST_RPTS_CUSTOM_CSS 238
VCAST_RPTS_DATE_COLUMN_WIDTH 240
VCAST_RPTS_HEADER 238
VCAST_RPTS_NOTES_COLUMN_WIDTH 241
VCAST_RPTS_RESULT_COLUMN_WIDTH 240
VCAST_RPTS_SHOW_VERSION 234
VCAST_RPTS_SUBPROGRAM_COLUMN_WIDTH 239
VCAST_RPTS_TESTCASE_COLUMN_WIDTH 240
VCAST_RPTS_UNIT_COLUMN_WIDTH 239
VCAST_RPTS_WRAP_NOTES 241
VCAST_SORT_METRICS_RPT_BY_DIR 230-231, 234
VCAST_STD_IO 116
VCAST_USE_EDG_PREPROCESSOR 181
VCAST_USE_FOPEN_S 114
VCAST_USE_GH_SYSCALL 117
VCAST_USE_PUTCHAR 117
VCAST_USE_STATIC_MEMORY 117
VCAST_USE_STD_STRING 117
VCAST_USE_STDOUT 117
VCAST_USE_VCPP 181
VCAST_VERBOSE_MANAGEMENT_REPORT 234
Vector preprocessor 181
verbose management report 234
version control 75
viewing
    coverage 133
visual studio secure mode 114
work flow
    append cover i/o 23
    build the instrumented code 28
    close environment 24
    create new system testing environment 16
    edit systems_tests.py script 25
    execute all tests 33
    instrument coverage 22
    instrument for coverage 22
    launch vectorcast 16
    open environment 22
    preliminary setup 14
    run a single test interactively 30
    set coverage type 22
    using the lua example 14
    view coverge results 32
working directory 87
    invalid 54
    remembering last 54
    setting 54
    uses of 54
    when starting from command line 54

when starting from Start menu 54
wrap text in Notes 241
writable checkbox
viewing source code 92