VECTOR



VectorCAST/C++

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: 6/8/2026, 8:08 PM

Rev: d2bd547

Part Number: VectorCAST/C++ User's Guide for VectorCAST 2026 SP2

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 **16**

VectorCAST Overview **17**
VectorCAST Automation 17
Key Terminology 17
Key Concepts 19
VectorCAST Components 21

Learning Your Way Around **23**

Starting VectorCAST **24**
To Start VectorCAST and Open an Environment 24
To Start VectorCAST, Build, and Open an Environment 24
To Start VectorCAST and Open an Options Dialog 24
To Start VectorCAST in C++ or Ada "Mode" 26
To Set the Product Mode 26
To Exit VectorCAST 27

VectorCAST Interface **28**
To Specify the Language for the VectorCAST GUI 29
Shift JIS Character Encoding 30
To Set the Industry Mode for Coverage 30
Changing Industry Mode 33
To Collapse / Expand the Message Window 33
To Hide the Message Window 34
To Clear or Copy Text in Message Window 35
To Hide the Environment View 35
To Move Docking Windows 36
To Float Docking Windows 37
To Return Docking Windows to Default Locations 38
To Save the Window Configuration 38
The Toolbar 39
Customize Icons 41
The Status Bar 43
The MDI Window: Groups 43
To Open a Group 44
To Close a Group 44
To Maximize a Window 45
To Close a Tab 45
To Arrange Groups in a Cascade 45
To Tile Groups 46
To Bring a Tab to the Top 48
To Close All Windows 48
VectorCAST Keyboard Shortcuts 48

Getting Help **51**
To Determine Available Licenses 51

To View Online User Guides ... 52
To Contact Technical Support ... 53
To Send a Test Environment to Technical Support ... 53
To Determine the VectorCAST Version and Customer Number ... 53

Using the Jobs Window and Jobs Monitor ... 55
The Jobs Window ... 55
Opening the Job Monitor ... 56
Job Status Viewer ... 57
Execution Status Viewer ... 58

Changing Application Preferences ... 62
To Set the Display Theme ... 62
To Set Up an External Text Editor ... 63
To Edit User Code with the External Editor ... 66
To Set Up an External Browser ... 66
To Toggle Gridlines in the Test Case Editor ... 68
To Alphabetize Test Cases ... 68
To Specify the Default String Display Mode ... 70
To Change the Main Window's Font ... 70
To Automatically Save Test Cases Before Execution ... 71
To Turn on a Reminder Before Closing Multiple Tabs ... 71
To Keep the Window Layout per Environment ... 71
Maximum Directories Added Recursively ... 72

Customizing Reports ... 72
Customizing Existing Reports ... 72
Customizing Templates ... 75
Customizing Report Sections ... 76
Customizing Report Style ... 78
Creating New Reports ... 81

Editing, Searching, and Printing ... 85
To Create a New Text File ... 85
To Open a Script or Report File ... 85
To Save a Script or Text File ... 86
To Save the Open Window As ... 86
To Save All Open Windows ... 86
Print Setup ... 86
To Print an Open Window ... 87
To Preview Before Printing ... 88
To Undo a Recent Change ... 88
To Redo ... 88
To Copy, Cut, and Paste ... 88
To Search for Text Using the Find Banner ... 89
To Apply a Search Filter to the Parameter Tree ... 92
To Apply a Search Filter to the Test Case Tree ... 92
To Goto a Line ... 98

Browsing Configuration Options ... 100
Using the Configuration Options Viewer ... 100

Creating a New Environment 102

Using the Wizard to Create an Environment 103
To Set the Working Directory 103
Troubleshooting the Working Directory 104
To Start the Wizard 104
To Save the Settings in the Wizard 105
Step 1: Choose Compiler 105
Step 2: Name the Environment 106
Step 3: Testing Method 107
Step 4: Build Options 108
To Use a JSON Compilation Database to Build an Environment 111
Step 5: Locate Source Files 114
Step 6: Choose UUTs & Stubs 123
Step 7: User Code (Optional) 143
Step 8: Summary 145
To Build the Environment 145
To Build an Environment for Object File Testing 146
To Build an Environment for Library Interface Testing 149
To Build an Environment for Test Driven Development 151
To View the Environment Overview Report 157
Files Created by the Environment 159
Troubleshooting Environment Creation 160
Tools to Aid Troubleshooting 164
To Create a System Testing Environment 166

Setting Compiler Options 168
Options: C/C++ Tab 168
Preprocessor/Compiler Tab 170
Setting Defaults for the Wizard 175
To Automatically Add Include Path and Defined Variables 184
To Test Your Compiler Settings 185

Setting Builder Options 205
Options: Builder Tab 205
Deprecated Builder Options 221

Working with a Test Environment 221
To Open an Environment 221
To Close an Environment 222
To Rename an Environment 222
To Update an Environment 222
To Create Regression Scripts for an Environment 224
To Post-Process the Regression Scripts 228
To Integrate Regression Scripts with ClearCase™ 230
To Save and Load a Post-Processing Script 231
To Incorporate Source Code Changes into Environment 232
To Rebuild the Environment 232
Troubleshooting Environment Rebuild 233
To Delete an Environment 234

To Re-create a Deleted Environment ... 235

Other Environment Tools ... 235
To Create an Environment Script ... 235
To Create an Environment by Running a Script ... 236
To Recompile the Test Harness ... 236
To Recompile the Instrumented Test Harness without Re-instrumenting It ... 237
To Create a Test Harness Recompile Script ... 237
Recompile the Test Harness by Running a Script ... 241
To Edit Link Options ... 242
To Relink an Environment ... 243
To Refresh Type Range Data ... 243
To Print Unit-specific Arguments ... 243

Building Test Cases ... 244

Using The Test Case Tree ... 245
The Test Case Tree Hierarchy ... 245
To Navigate the Test Case Tree ... 246
To Multi-Select Test Cases ... 247
Types of VectorCAST Test Cases ... 247
To Open a Test Case for Editing ... 248
Test Case Naming ... 248
To Rename a Test Case ... 248
To Sort Test Cases Alphabetically ... 248
To Sort the Test Case Tree ... 249
To Duplicate a Test Case ... 250
To Enable/Disable a Test Case ... 250
To Delete a Test Case ... 252
To Delete All Test Cases from a Subprogram ... 253
To Delete All Test Cases from a Unit ... 253
To Delete All Test Cases in the Environment ... 253
To Restore Deleted Test Cases ... 254
To View the Source for a Subprogram ... 254
To View Coverage for a Unit or Subprogram ... 254

Simple Test Cases ... 255
To Insert a Test Case for a Subprogram ... 255
To Insert an <<INIT>> Test Case ... 255
To Create a Min, Mid, or Max Test Case ... 255
To Insert Basis Path Test Cases ... 257
Troubleshooting Min, Mid, Max Test Cases ... 259

Compound Test Cases ... 261
Building a Compound Test Case ... 261
To Specify the Number of Iterations of a Test in a Compound ... 262
Compound as Automatic Initialization / Finalization ... 264
To Reorder Test Cases in a Compound ... 265
To Open a Test Case from a Compound ... 265
To Delete a Slot from a Compound ... 266
To View Coverage for a Test Case from a Compound ... 266

To Locate a Slot in the Test Case Tree ... 266
To Search for Text in the Compound Test Editor ... 266
Compound Only Test Cases ... 267
Nested Compound Test Cases ... 267
To Enable / Disable Reporting for Compound Slots ... 270
To Set Expected Values to Actual Values ... 271

Entering Test Case Data ... 272
The Parameter Tree ... 272
To Navigate the Parameter Tree ... 275
To Alphabetize Parameters in the Parameter Tree ... 276
To Enter Input and Expected Values ... 277
Understanding Input and Expected Values ... 278
Input and Expected Values in a Stubbed-by-Function Unit ... 279
To Enter Test Case Requirements or Notes ... 281
To Import the Contents of a File ... 281
To Use an External Editor ... 282
To Import a Template for User Code or Test Case Notes ... 282
To Instantiate a Class Object ... 282
To Enter Values for Global Data ... 283
To Enter a Range ... 284
To Enter a List ... 284

Data Types ... 284
To Enter Values for Enumeration Types ... 284
To Enter a Number for an Enumeration ... 285
To Enter Values for Numeric Types ... 285
To Enter Integer Values in Different Bases ... 286
To Enter Real Numbers using Scientific Notation ... 287
To Use Symbolic Constants ... 287
Using Test-Only Symbolic Constants ... 290
To Enter Values for Structure Types ... 291
To Enter Values for Union Types ... 292
To Enter Values for Class Types ... 292
Void Pointer Types ... 293
Character Types ... 296
To Enter a String ... 296
To Change a String Type into an Array of Characters ... 297
Support for C++ Standard Template Library (STL) types ... 298

Working with Arrays ... 306
To Enter Values for Constrained Array Types ... 306
To Expand All Elements of an Array ... 306
To Expand the First Element of an Array ... 307
To Collapse Unused Elements of an Array ... 307
To Expand Certain Elements of an Array ... 308
To Apply Values to an Array ... 309
To Clear Values from an Array ... 311
Using the Array Properties Dialog ... 312
Using Enumerals to Size and Index an Array ... 312
Unconstrained Arrays and Pointer Types ... 313

Entering Data with the Parameter Dialog Box **314**
To Access the Parameter Dialog 314
The Scalar Values Tab 314
To Apply Values to an Array 315
To Clear Values from an Array 316
The Range Values Tab 317
To Set Up an Input Range or List for More Than One Parameter 318
To Use Ranges as Expected Values 318
The List Values Tab 319
Controlling Values in the List 319
To Repeat Values in the List 320
To Use a Range Expression in an Expected Value List 321
To Use the <<Any>> Tag in a List 322
Range and List Example 322

Test Case Scripting **326**
To Export Test Cases to a Test Script 327
Test Script Organization 327
To Clear Unapplied Test Script Data 328
To Import Test Cases from a Test Script 328
To Create a Test Case Template 329
Troubleshooting Test Script Template 330
Automating Test Script Maintenance 331

Generating Test Cases Using Automatic Test Case Generation (ATG) **336**
To Generate Test Cases Using ATG 336
To Delete and Reload Test Cases Created by ATG 339
ATG Baselining Functionality 342

Generating Test Cases Using Coded Tests **344**
To Generate Test Cases Using Coded Tests 344

Generate Coded Tests with ATG **350**
Basic Use Case 351
Generated Output Files 351
Advanced Features 352
Generated Code Structure 353

Generating Test Cases Using CSV- or Tab-Delimited Files **355**
To Create a Template for a CSV- or Tab-Delimited File 356
To Create a Map from an Existing CSV or Tab-Delimited File 362
To Create Test Cases from a MAP Test Case 365
To Edit an Existing Map Test Case 366
To Create and Edit a Test Script from a Map File 367

Generating Test Cases Using the Vector Test Data Editor **369**
Make a New VCT MAP Test Case 369
Use the Test Data Editor to Set Input and Expected Values 370
Generate Automated Tests 374
Export/Import VCT MAP Test Case and .vct File 375

VectorCAST Tools **376**
To Set VectorCAST Options 376

Tools Menu ... 377
View Harness Source ... 377
Basis Path ... 378
MC/DC ... 379
Working With Custom Tools ... 380
To Create a Diagnostic Report ... 387
Troubleshooting VectorCAST ... 387

Executing Tests ... 389

Executing Test Cases ... 390
To Execute a Test Case and View Results ... 390
To Automatically Save Test Cases Before Execution ... 391
To Select the Next Passing or Failing Test Case ... 391
Compound Test Case Execution and Reports ... 398
To Execute Multiple Test Cases ... 400
Control Test Case Execution Using Variant Logic ... 400
To Abort Test Case Execution ... 405
To Execute a Test Case in the Debugger ... 405
Working with a Control Flow ... 405
To Save the Control Flow ... 409
To Clear the Control Flow ... 409
To Set Expected Values to Actual Values ... 409

Viewing Test Reports ... 410
View Reports in an External Browser ... 410
View a Test Case Data Report ... 411
View Test Execution Results ... 412
The Full Report ... 415
The Test Case Management Report ... 416
View Aggregate Coverage Report ... 417
View Metrics Report ... 418
View Function Call Report ... 418
View Covered By Analysis Report ... 419
View Test Data Summary ... 419
View the Test Comparison Report ... 423
View a Test Case's Raw Data Set ... 431

Setting Execution Options ... 431
Options: Execute Tab ... 431

Setting Report Options ... 443
Report Content Options ... 443
Report Format Options ... 457
Other Test Case Options ... 467

Incremental Rebuild ... 474

Performing an Incremental Rebuild ... 475

Using Code Coverage 477

Code Coverage 478
To Initialize Statement Coverage 479
To Initialize Branch Coverage 480
To Initialize MC/DC Coverage 481
Suppressing MC/DC Initialization on Compile Error 482
To Initialize Statement+MC/DC 483
To Initialize Function Coverage 483
To Initialize Function + Function Call Coverage 484
To Initialize Statement+Branch 485
To Initialize Coverage for Units Other than UUT 485
To Avoid Instrumenting Sections of Source Code 487
To Uninstrument an Environment 488
To Enable Coverage 488
To Disable Coverage 488

Viewing Coverage Results 489
To Turn on Coverage Results 489
The Coverage Viewer 490
To Customize the Coverage Viewer 496
Map Line Selection to Original Source View 496
To Open a Test Case That Covers a Line 497
To Remove All Coverage Results 499
To View the Aggregate Coverage Report 499
To View the Metrics Report 500
To View Code Coverage Summary 503
Understanding Basis Paths 506
To Build Test Cases from Basis Path Analysis 507
To View a Basis Paths Report 508
MC/DC Coverage 509
Understanding MC/DC Analysis 510
To View the Equivalence Matrices Report 511
To Insert MC/DC Test Cases 513

Viewing Execution Flow with Animated Coverage 515
To Activate Coverage Animation 515
To Play the Coverage Animation 516
The Animation Toolbar 517
To Set a Breakpoint 518

Setting Coverage Options 518
General Coverage Options 518
Instrumentation Options 524
MC/DC Options 533
Miscellaneous Options 537
Suppressed Coverable Functions Option 540
Coverage Viewer Options 542
To Format the Text in the Coverage Viewer 543

Importing Coverage Results 545

Preparing to Import Results 545
Importing the Results 545
To Export a Script of Coverage 547
To Import a Coverage Script 550
To Delete Imported Results 550
To View the Coverage Import Log 550

Using VectorCAST Probe Points 551

Probe Points 552

Function Probe Points 552
Function Entry Probe Points 553
Function Exit Probe Points 554
Probe Point Order of Execution 555

Using the Probe Point Editor 556
Open the Probe Point Editor 556
Insert a Probe Point 557
Insert a File Scope Probe Point 558
Test Compile a Probe Point 559
Saving and Applying Probe Points 561
Apply All Probe Points 562
Probe Point Status Buttons 563
Edit a Probe Point 563
Built-in Functions and Macros 564
Deactivate/Activate Probe Points 565
Remove Probe Points from Editor 566
Delete All Probe Points 566

Probe Point Events 567

Capture Local Variable Values 569

Inject Spurious Values 570

Patch Faulty Code 571

Probe Point Listing 572

Using the Probe Point API 573
The Probe Point File 573
Probe Matching Line Index 574
Probe ID Number 575
Export a Probe Point 576
Import a Probe Point 577
Enable, Disable and Remove Probe Points 578
Create a Probe Point Report 579

Using VectorCAST Covered By Analysis 580

Covered By Analysis (CBA) 581

To Add Coverage Analysis 581

Using the Coverage Analysis Editor ... 581
To Edit an Existing Analysis Result ... 584
Viewing CBA Coverage in the Coverage Viewer ... 585
To Change Covered-By-Analysis Display Color ... 585

Working With Analysis Files ... 586
To Import Analysis Files ... 586
To Export Analysis Files ... 586
To Remove Analysis Files ... 587

Re-Instrumenting With CBA Data ... 587

Viewing Analysis Data in Reports ... 587
Covered By Analysis Report ... 587
Aggregate Coverage Report ... 590
Metrics Report ... 591

Using Coverage Analysis With SFP ... 592
To Add Coverage Analysis ... 593
To Add or Remove Coverage Analysis for Multiple Lines ... 593
To Add or Remove Coverage Analysis for MC/DC ... 593
To Edit an Existing Result ... 594
Changes for the Covered By Analysis Report ... 594
The Aggregate Report ... 594

User Code ... 595

Understanding User Code ... 596
Types of User Code ... 596
Order of Execution ... 596
Editing User Code ... 597
User Code Tags ... 597
Features Common to All User Code ... 598

Environment User Code ... 600
Types of Environment User Code ... 600
To Edit Environment User Code ... 602
To Test Compile Environment User Code ... 603
User Globals ... 606
To Edit User Globals ... 606
To Save User Globals ... 608
To Recompile Environment User Code ... 609
Unit Appendix User Code ... 609
To Test Compile Unit Appendix User Code ... 610
To Save Unit Appendix User Code ... 611
To Recompile Unit Appendix User Code ... 612
To Remove Unit Appendix User Code ... 612
Example of Unit Appendix User Code ... 612
To Remove Environment User Code ... 614

Test Case User Code ... 614
To Enter Test Case User Code ... 615

To Test Compile Testcase User Code ... 615
To Clear Test Case User Code from Test Harness ... 616
Add Test Case User Code Back Test Harness ... 616

Parameter User Code ... 617
To Enter Parameter User Code ... 617
To Test Compile Parameter User Code ... 617
Input User Code Example ... 617
Expected User Code Example ... 619
Display Actual Value When Entering Expected User Code ... 620
Another User Code Example ... 621

Stub User Code ... 623
To Enter Configure Stub User Code ... 623
To Test Compile Stub User Code ... 625
To Save Stub User Code ... 626
To Recompile Stub User Code ... 627
To Remove Stub User Code ... 627
Example of Stub User Code ... 628

Using the Requirements Gateway ... 630

Requirements Gateway (RGW) ... 631

Using Requirements Gateway ... 631
Create a Requirements Repository ... 632
The Authentication Tab ... 632
The Import Tab ... 633
The Export Tab ... 635
The Requirements Tab ... 636
The Script Tab ... 637
Link Requirements to Test Cases ... 638
Tracking Requirement Change Impacts on Test Cases ... 640

Working With Supported Gateways ... 642

Example Work Flow for RGW Command Line ... 643
Step 1 - Set the Repository ... 644
Step 2 - Initialize the Gateway ... 644
Step 3 - Set Configuration Options ... 644
Step 4 - Import Requirements ... 646
Step 5 - Link Requirements to a Test Case ... 646
Step 6 - Run the Test Case ... 647
Step 7 - Configure the Export Settings ... 647
Step 8 - Export the Test Data ... 648

Using AI to Generate Test Cases from Requirements ... 648
CLI Example: Requirements to Tests ... 649
Command Reference for panreq ... 651

Appendix A: VectorCAST/C++ Messages ... 652

Start-up Messages 652
License Messages 652
Environment Building Informational Messages 652
Environment Building Error Messages 653
Test Case Messages 655
Test Case Variants Messages 656
General Messages 657
CLICAST Messages 660

Appendix B: Environment Script Language 661

Enviro Command List 661

Appendix C: Test Script Language 674

Script Commands 674
Setting Input and Expected Values 683
Numeric Types 684
Structure Types 684
Enumeration Types 684
Character and String Types 684
Pointer and Array Types 685
Structure Types 686
Function Return Parameters 686
Global Objects 686
Test Case Options 686
Min, Mid, or Max Values 686
Range of Values for Input Data 687
Range of Values for Expected Results 688
List of Values 688
Test Values Spanning Multiple Lines 688
Working with Overloaded Subprograms 689

Appendix D: Limitations and Restrictions 690

Appendix E: Environment Variables 691

Environment Variables 691

Appendix F: CLICAST - Command Line VectorCAST 696

Quick Start 696
Command Format 697
Console Report Format 698

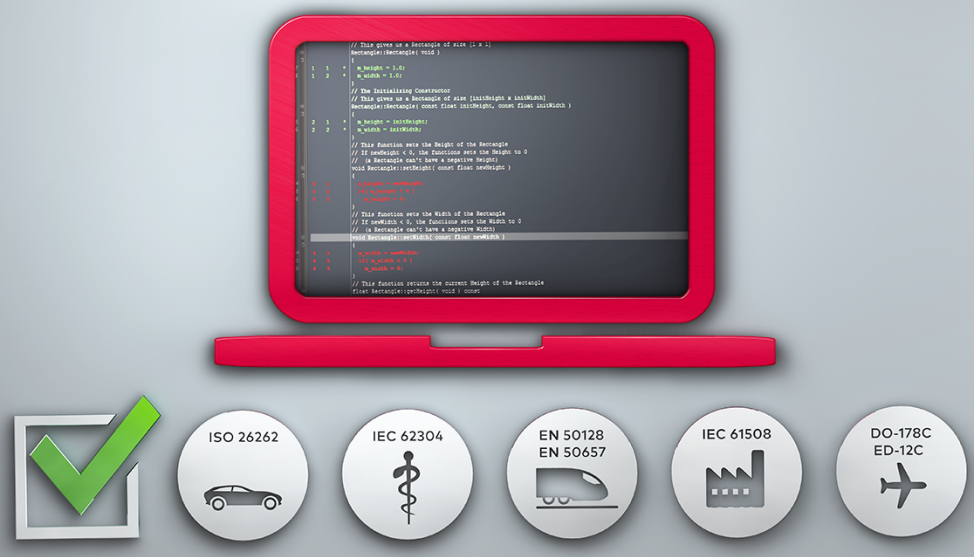
Appendix G: CLICAST - Tool Options Reference 702

Tool Options 702

Appendix H: RGW - Command Line Reference 784

RGW Commands 784
Configure Set 784
Export 785
Import 785
Initialize 785
Requirement Report 785
Set Gateway 786
Testcase Link 786
Testcase Mark as Reviewed 786
Testcase Migrate_links 786
Testcase Report 787

Index 788



Introduction

VectorCAST Overview

VectorCAST is a suite of tools for automating the entire process associated with conducting unit and integration testing:

- VectorCAST/C++ and VectorCAST/Ada automatically generate executable test harnesses for one or more units of application source code written in C, C++ or Ada. In addition, they generate and execute unit test cases and report on results and metrics.
- VectorCAST/QA is a code-coverage analysis tool that identifies which areas of an application have been exercised by test executions. It supports C, C++ or Ada, and supplements the unit coverage provided by VectorCAST/C++ and VectorCAST/Ada.

VectorCAST Automation

You can use VectorCAST to conduct unit or integration testing on applications written in the C, C++, or Ada programming languages. VectorCAST automates the following unit and integration test functions:

- Generating and compiling test stubs and driver programs
- Generating test cases based on boundary values
- Constructing user-defined test cases by interactive point-and-click, or by script
- Executing modified test cases without re-compiling the test harness (Exception: when running in stdout only mode, the test harness must be recompiled.)
- Conducting regression testing
- Generating standards-compliant test reports
- Conducting comprehensive branch and statement coverage analysis
- Conducting Modified Condition/Decision Coverage (MC/DC) analysis
- Conducting basis-path analysis and determining cyclomatic complexity
- Executing tests on host and embedded-target development systems

Three types of input data can affect the processing of a unit under test (UUT):

- The input parameters to the UUT
- The output parameters of dependent-unit subprograms (that is, the output of subprograms invoked by the unit)
- Global data objects

VectorCAST provides direct control over all three types of data. It does this by automatically creating a unique test harness for each software unit to be tested.

Key Terminology

You should be familiar with the following terms before you begin using the tutorials:

Environment – The term “Environment”, as it is used in this document, refers to a repository of information saved in a VectorCAST-created disk directory. This directory is used to store test harness source files, test-specific data files, etc.

Test Harness – A test harness is the executable program created when the test driver, the units

under test, and the dependent units or stubs of dependent units are linked together.

The test harness has three components:

- **Units Under Test (UUTs)**
  A Unit Under Test is a compilation unit to be tested. In C or C++, a UUT is a single source file. In Ada, it can be a standalone subprogram specification, subprogram body, a package specification, or package body. You can have more than one unit under test in a test environment (for integration testing).
- **Test Driver**
  The driver program is capable of invoking all subprograms of the unit under test with values provided from the test case data and capturing any returned data or exceptions that may be raised during test execution. The components of the test harness are automatically compiled and linked into an executable test harness, which is used by VectorCAST to effect all testing.
- **SmartStubs**
  When testing a software unit, it is often desirable to create alternate subprogram definitions for some or all dependent units. These “placeholders” used for testing purposes are known as *stubs*. Stubbed subprograms allow you to begin testing long before the actual system is built. They also allow you to exercise more exact control over the values passed back to the Unit Under Test than would be available with the actual dependent unit in place. VectorCAST automatically generates the code for those dependent units chosen to be stubbed.

Visible Subprogram – In C, a visible subprogram is a subprogram that is not declared as “static”; in C++, it is a member function that is not declared “private” or “protected”.

Dependent Unit – A file that contains a function called by a UUT, or data objects used by a UUT. A dependent unit is a C/C++ file that is included. Dependent units can be stubbed or used 'as is'.

Event – An event is a change in the flow of control that VectorCAST can monitor. Each of the following is an event:

- A call from the test harness to a UUT
- A call from a UUT to a stubbed dependent in the test harness
- A return from a UUT to the test harness
- A probe point that calls **`vcast_probe_assert()`**
- A function or regular probe point when the option "Consider probes as events" is True. Both the call-before and the call-after are events. File scope probes are not affected.

Test Case – A collection of input and output data, defined with the purpose of exercising specific software and verifying results. This data is commonly comprised of the formal input and output parameters of the dependent subprograms, and input and output parameters of the subprogram under test. The Test Data controls the execution of the unit under test as well as values for global objects visible in the unit under test or stubbed subprograms.

Test Driver – VectorCAST generates the main program, or *driver*, necessary to exercise all visible subprograms in all units under test. This consists of calls to each visible subprogram in a unit under test.

Search List (C/C++) – The directory that contains the units under test and dependent units. In C or C++ environments, the Search List can contain multiple directories.

Prototype Stubbing – In C and C++ environments, VectorCAST can generate stubs of dependent units using only the function's declaration. If VectorCAST cannot find the function's definition (the implementation of the function) in any C/C++ source file in the Search List, then it creates a stub of the function using the function declaration (from a header file). Prototype-stubs are grouped under the name **Stubbed Subprograms** instead of under the name of a unit.

Stub by Function Units – Prior to VectorCAST version 5.0, only subprograms in dependent units of the UUT were allowed to be stubbed. In fact, all subprograms in stubbed units had to be stubbed. In VectorCAST 5.0, you have the choice of making a Unit Under Test "stub by function (SBF)." When a UUT is made "SBF", this means that functions in the unit can be stubbed individually, while others in the unit are not stubbed.

Testable Class – In C++ environments, a testable class is one that has any member function defined in a UUT source file, or an inlined function defined in a header that is included by a UUT source file.

Key Concepts

You should be familiar with the following concepts before you begin using the tutorials:

Execution Closure – An execution closure consists of all the units required for an application to successfully link. Figure 1 shows a closure consisting of a single UUT, three direct dependent units, and several sub-dependents.

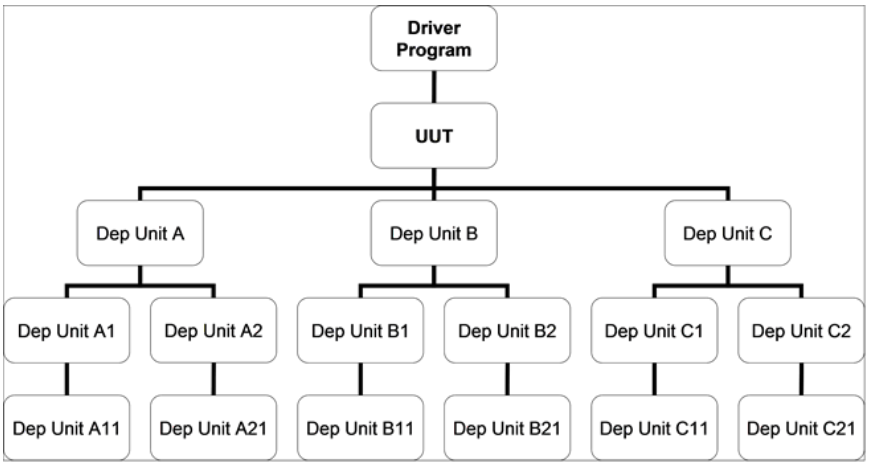


Figure 1. An example execution closure.

Stubbing Units in a Closure – When building a VectorCAST environment, you designate stubbing from the top-most level of dependency downward to the bottom-most level in each branch.

In the closure illustrated in Figure 1, for example, dependents Dep Unit A, Dep Unit B, and Dep Unit C would be the primary candidates for stubbing. If you were to leave one of these dependents unstubbed, the dependents immediately under it would then become the primary candidates for stubbing, and so on to the bottom of each branch.

Figure 2 shows the example closure after stubbing has been designated.

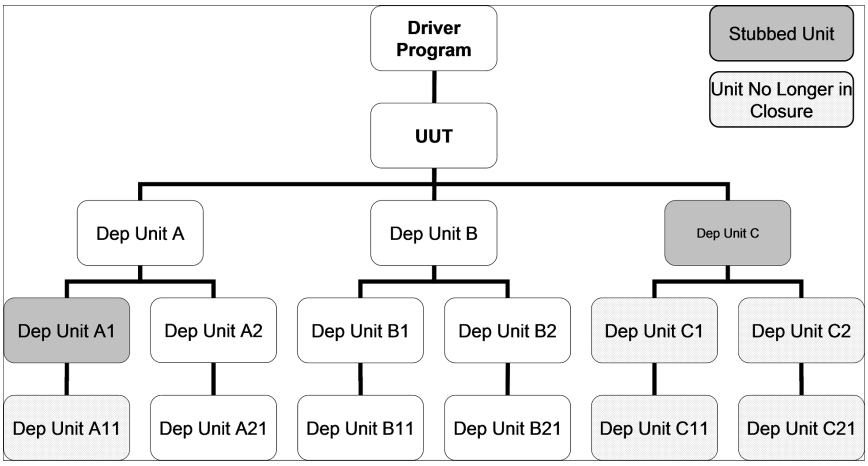


Figure 2. A typical VectorCAST closure.

The scheme allows a flexibility spanning from testing a single unit in a standalone environment in which every dependent is stubbed, to testing groups of units with only the dependents on the periphery (bottom) of the closure stubbed.

Ignoring Units in a Closure (Ada only) – As mentioned above, normally when you choose not to stub a unit, any dependents below this unit remain in the closure. In VectorCAST/Ada, however, you can choose to leave a unit unstubbed but effectively remove all the downstream dependents from the list of stub candidates; all downstream dependents will not be stubbed. You do this by directing VectorCAST/Ada to ignore the unit.

Note: In Ada, ignoring dependents applies only to the dependents of the 'body' of the ignored unit; it does not apply to the dependents of the 'spec'. The reason for this is that VectorCAST needs to parse the dependents of the 'spec' in order to resolve types derived from the dependents.

This option is useful when you want to avoid doing any stubbing beyond a certain point. For example, in Figure 3, suppose Unit C is an interface to a subsystem already tested. There is no need to stub beyond Unit C. By directing VectorCAST to ignore Unit C, it will use this entire branch as if you had chosen not to stub the units.

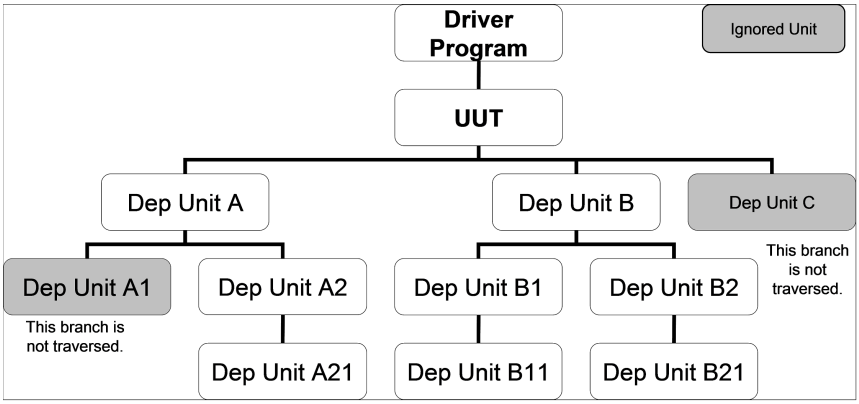


Figure 3. Closure with ignored units (Ada only)

This option can in some cases dramatically reduce the time it takes for VectorCAST to build an environment.



Note: In choosing to ignore a unit, downstream dependents will not be ignored if these units are brought into the closure through an alternate dependency.

VectorCAST Components

VectorCAST consists of four major functional components:

- Test Environment Builder
- Test Case Generator
- Execution Manager
- Report Generator

Test Environment Builder

The Test Environment Builder is central to VectorCAST and serves two related functions:

- It generates a framework (environment) consisting of all the necessary resources to build a test harness
- It generates (or regenerates) the test harness itself

To initiate a test environment, the user:

- Supplies the location of the source files to undergo test
- Designates the individual UUT(s) to be exercised
- Designates any dependent units to be stubbed

The Environment Builder parses the designated files and, from the information gathered, generates a complete test harness.

Unlike a manually generated test harness, a VectorCAST test harness can be automatically regenerated whenever changes are made to a UUT or stub included in the environment (for example, when a subprogram is fully coded).

In addition, test cases can be added to an existing environment without any need for manual coding, recompiling, etc.

Test Case Generator

The Test Case Generator uses static analysis and/or user input to build test cases for exercising the UUTs included in a test environment. These test cases (and their results) are stored within the test environment itself and are thereby available for use in future testing. This is an important feature for regression testing.

Execution Manager

The Execution Manager invokes the test harness created by the Environment Builder, and executes

the test cases built by the Test Case Generator.

Report Generator

The Report Generator produces a variety of reports based on information maintained in every environment database. This information includes:

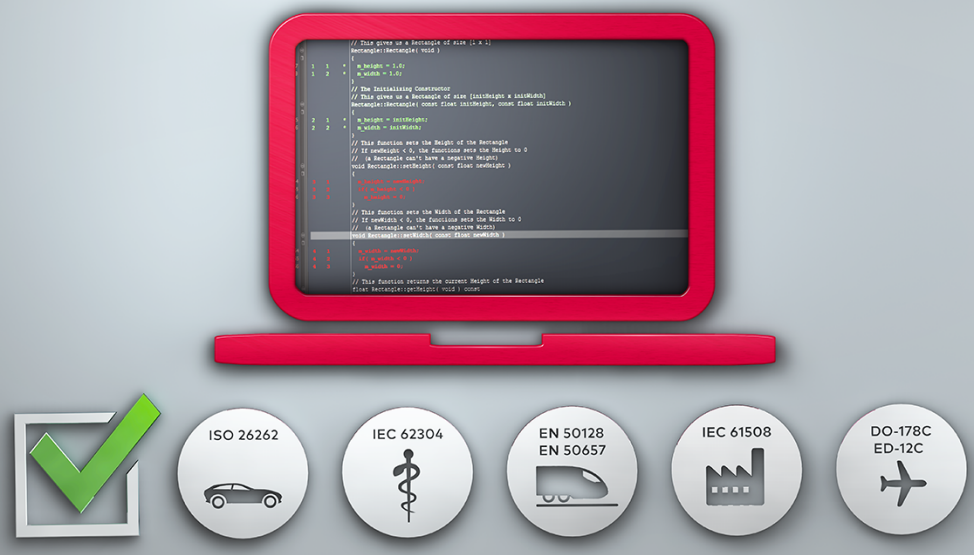
Test configurations – Unit names, test names, dates and times.

Test case data – Listings of inputs and expected results.

Execution histories – Listings of execution results; comparisons of expected and actual results; comparisons of control flow; pass/fail indications.

Coverage – Source listings annotated and color-coded to indicate code coverage.

Results – Tables summarizing pass/fail status and coverage results.



Learning Your Way Around

Starting VectorCAST

Prior to starting VectorCAST, you should ensure that VectorCAST is installed. See the *VectorCAST Installation Guide* for details. Also, the *VectorCAST Quick Start Guide* has additional information for starting VectorCAST.

To Start VectorCAST and Open an Environment

To start VectorCAST and open an environment, first change directories to the directory where the environment exists, then use the **-e** command line option, as in the following examples.

Windows Command Prompt

```
%VECTORCAST_DIR%\vcastqt -e <environment_name>
```

Linux

```
$VECTORCAST_DIR/vcastqt -e <environment_name>
```

To Start VectorCAST, Build, and Open an Environment

To start VectorCAST, build an environment and open it, use the **-b** command line option, as in the following examples.

Windows Command Prompt

```
%VECTORCAST_DIR%\vcastqt -b <env script>
```

Linux

```
$VECTORCAST_DIR/vcastqt -b <env script>
```

The path to **<env script>** can be relative to the directory from which VectorCAST is invoked, or a full path. The filename can be specified with or without the **.env** extension.

To Start VectorCAST and Open an Options Dialog

To start VectorCAST and open an Options dialog from the command line use the **-o** command line option, as in the following examples.

Windows Command Prompt

```
%VECTORCAST_DIR%\vcastqt -lc  -o <path to working directory>
```

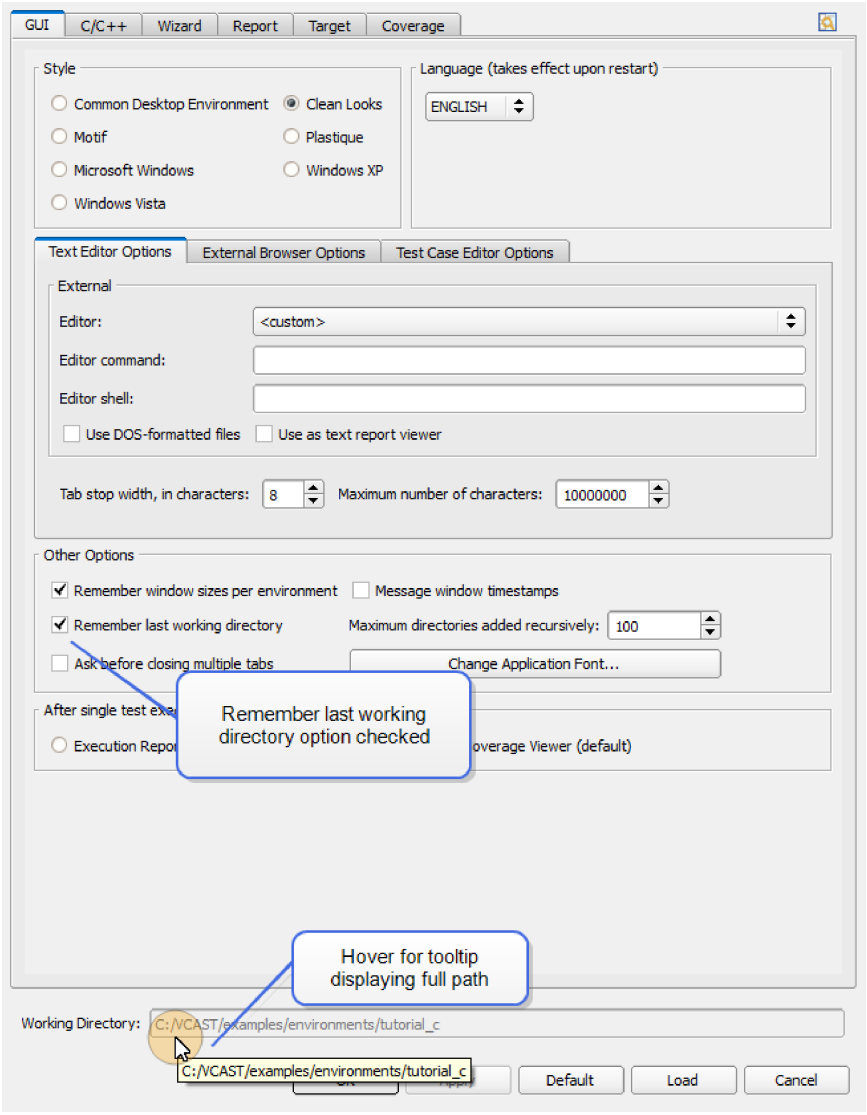
Linux

```
$VECTORCAST_DIR/vcastqt -lc  -o <path to working directory>
```

Users can specify a path to a working directory immediately after the `-o` command line option. The Options Dialog uses this path to set the current working directory from which configuration options are loaded.

When the `-o` option is used without specifying a path, the behavior of the Options Dialog depends on two factors:

- If the option to remember the last working directory is not checked, or if the last working directory is not set, the Options Dialog defaults to the current directory.
- If the option to remember the last working directory is checked and the last working directory is valid, the Options Dialog defaults to using that directory.



The `-o` option accepts the following path formats:

- Absolute paths
- Relative paths
- Paths with tilde (~) expansion
- Paths containing spaces
- Paths derived from environment variables set to a directory

Note that invalid paths are ignored. In such cases, the Options Dialog reverts to using the current directory.

To Start VectorCAST in C++ or Ada "Mode"

If you have both VectorCAST/C++ and VectorCAST/Ada licensed, use the following command to start VectorCAST in “C/C++ mode” or “Ada mode.” Thereafter, when you click the New button, the wizard for the specified language opens, instead of giving you a choice.

Windows Command Prompt

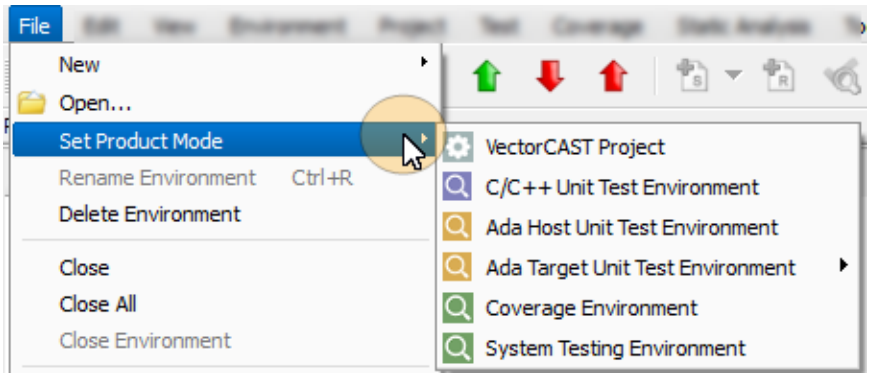
```
C:\> %VECTORCAST_DIR%\vcastqt -lc (to open VectorCAST/C++)
C:\> %VECTORCAST_DIR%\vcastqt -lada (to open VectorCAST/Ada)
```

Linux

```
$> $VECTORCAST_DIR/vcastqt -lc (to open VectorCAST/C++)
$> $VECTORCAST_DIR/vcastqt -lada (to open VectorCAST/Ada)
```

To Set the Product Mode

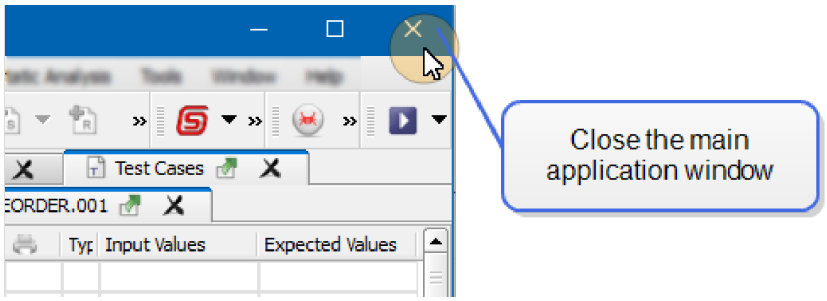
You can set or change the current product mode without having to open an environment. To do this, choose **File => Set Product Mode => <mode>**. The available options for <mode> depend on the VectorCAST products your organization is licensed to use.



If you already have an environment open, the **File => Set Product Mode** menu item is dim.

To Exit VectorCAST

The **File => Exit** command enables you to exit the VectorCAST application. You can also exit by using the shortcut of clicking the **"X"** in the top right-hand corner of the main window.



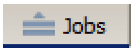
VectorCAST Interface

When you start VectorCAST, the VectorCAST interface (main window) appears. This window can vary slightly in detail on the basis of the licenses held. For users with only VectorCAST/Ada licensed, the VectorCAST/Ada window opens. For users with only VectorCAST/C++ licensed, the VectorCAST/C++ window opens. For users with licenses for multiple VectorCAST products, a product-neutral VectorCAST window opens.

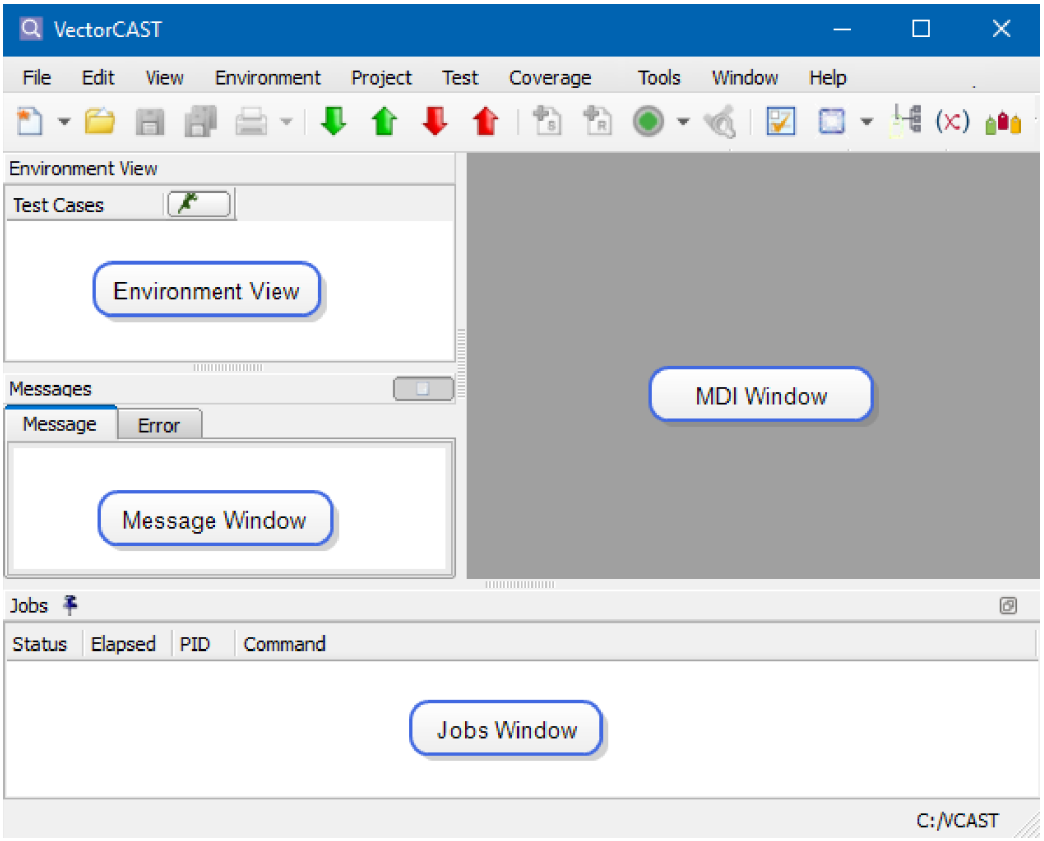
The VectorCAST main window is divided into four panes:

- The **Environment View** is located on the left-hand side of the main window. It provides a high level view of the project structure. The Environment View pane contains the Test Case Tree.
- The **Message Window** is located along the bottom left of the main window. It contains tabs for informational messages and for error messages. The Message window can be expanded and collapsed using the small  button located on the upper right corner of the message window.
- The **Multiple Document Interface (MDI) Window** is located to the right of the Project Tree. It displays a variety of windows, including Test Case editors, Coverage Viewers, Report Viewers and Text Editors. Windows are collected into groups.
- The **Jobs Window** is located on the bottom of the main window. It displays the status of jobs as they execute and exposes the associated back-end commands.

The **Environment View** and the **Message Window** are docking windows: That is, you can move these from the context of the main window to any other location on your desktop.

The **Jobs Window** is an auto-hide window. Hover over the Jobs tab 

to open it and click on the pin icon  to keep it open.



See also "To Move Docking Windows" on page 36.

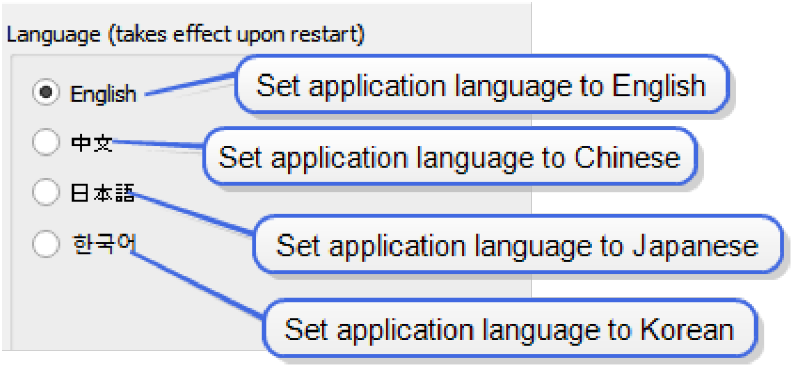
The VectorCAST tools and functions are available from a menu bar and a toolbar located at the top of the main window.

To Specify the Language for the VectorCAST GUI

Choose **Tools => Options**, and click the **GUI tab**.

By default, the VectorCAST application displays text in English, using the Latin1 character encoding. You can set the application to display in Chinese, Japanese or Korean by selecting the radio button for the desired language. Available languages are displayed in their native script for better recognition and usability. Note that you must exit VectorCAST and restart it for the change to take effect.

This information is saved to a file named **`.vcast-qt`** in the user's HOME directory and persists across sessions for a consistent user experience.



Shift JIS Character Encoding

If source code files use the character encoding Shift JIS, select the option **Source code files use Shift JIS**, located in the **Options** dialog. This option is saved to the CCAST_.CFG file.

Choose **Tools => Options**, and click the **C/C++** tab, and **Language** sub-tab.

```
clicast -lc option VCAST_MULTIBYTE_CHARACTERS True | False

Enable this option if source code files are encoded using Shift JIS.
```

To Set the Industry Mode for Coverage

Setting an Industry Mode provides familiar industry specific choices when selecting a coverage type. The supported Industry Modes and their associated coverage types are listed below.

- Avionics (DO-178 B/C)
  - Level A (Statement, Branch, MC/DC),
  - Level B (Statement, Branch)
  - Level C (Statement)
- Automotive (ISO-26262)
  - Architectural Level ASIL A/B (Function)
  - Architectural Level ASIL C/D (Function, Function Call)
  - Unit Level ASIL A (Statement)
  - Unit Level ASIL B/C (Statement, Branch)
  - Unit Level ASIL D (Statement, Branch, MC/DC)
- Industrial (IEC-61508)
  - SIL4 (Statement, Branch, MC/DC)
  - SIL3 (Statement, Branch)
  - SIL 1/2 (Statement)
- Railway (EN-50128)
  - SIL4 (Statement, Branch, MC/DC),
  - SIL3 (Statement, Branch)

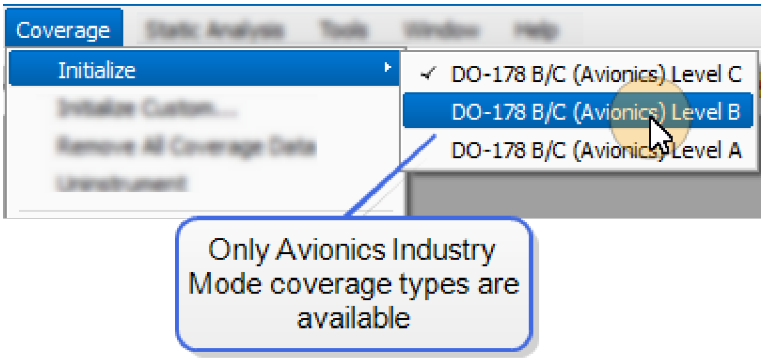
- SIL 1/2 (Statement)

- Medical (IEC-62304)
  - Class C (Statement, Branch, MC/DC)
  - Class B (Statement, Branch)
  - Class A (Statement)

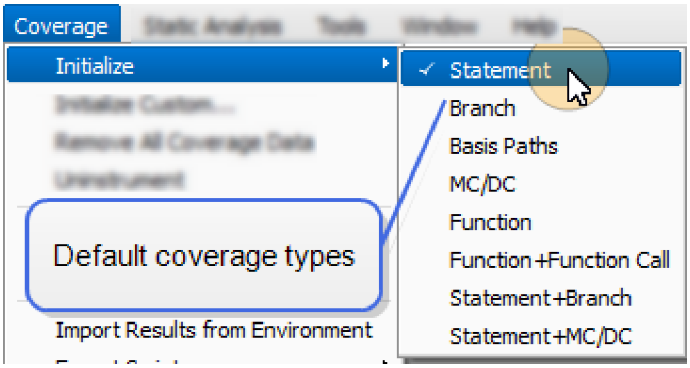
- Default
  - Statement
  - Branch
  - Basis Paths
  - MC/DC
  - Function
  - Function + Function Call
  - Statement + Branch
  - Statement + MC/DC

To select the industry mode, select the **Tools=>Industry Mode** command and then select the specific Industry Mode from the context menu.

When initializing or selecting a coverage type, the industry appropriate coverage types are shown.

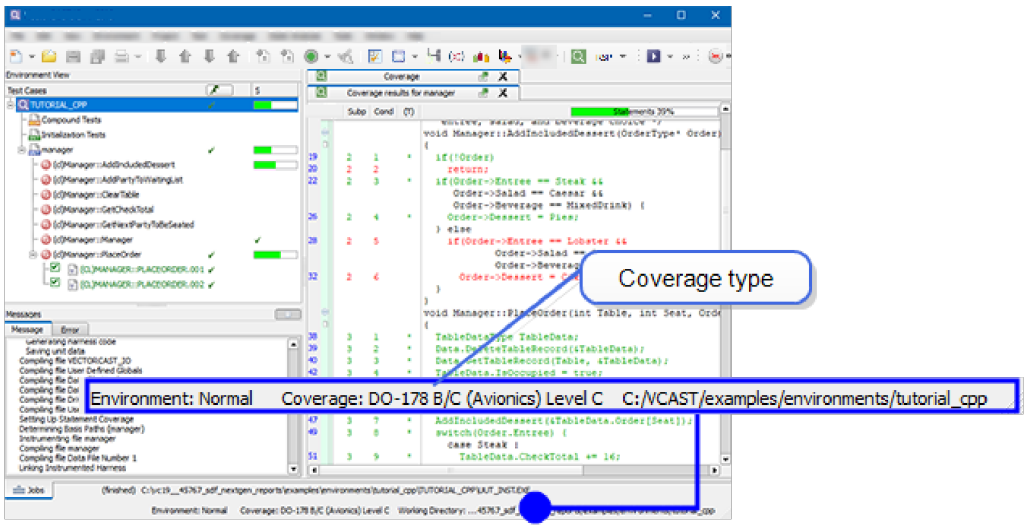


If you do not set an Industry Mode, the Default mode is used and you will be presented with the Default coverage types when choosing or initializing coverage.

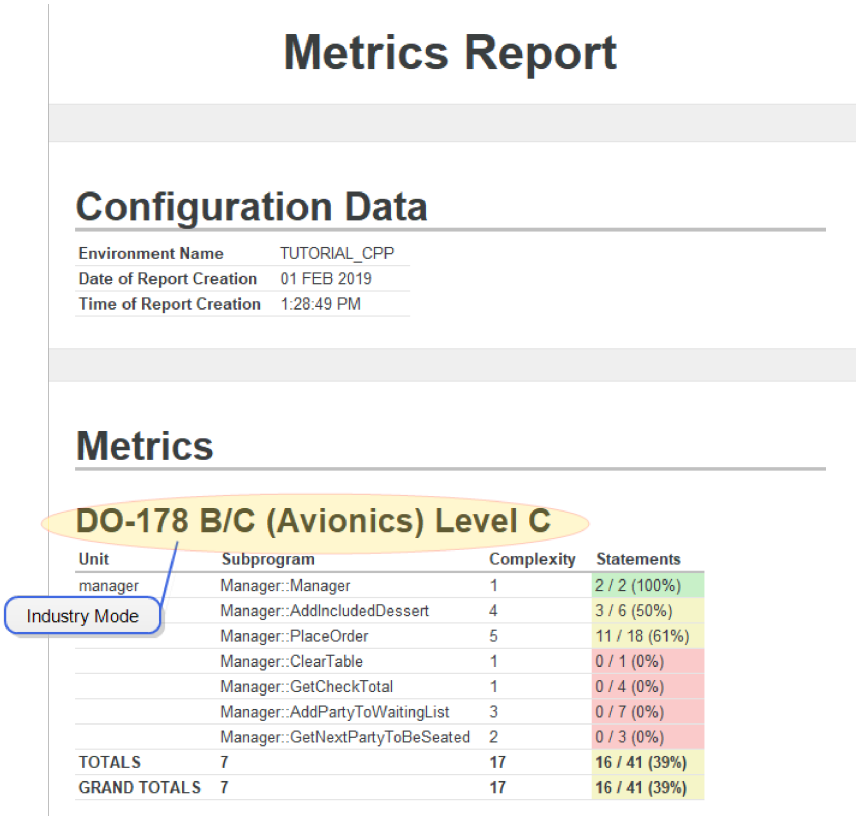


Once an Industry Mode is selected, it is stored in the **`.vcast-qt`** file and the chosen Industry Mode remains in effect until a different Industry is selected.

When coverage is initialized, the industry appropriate coverage type is displayed in the status bar at the bottom.



Coverage reports reflect the associated coverage categories, Statements, Branches, MC/DC Pairs, for the coverage type chosen. In addition, the Industry Mode will be shown on the report.



Metrics Report

Configuration Data

Environment Name	TUTORIAL_CPP
Date of Report Creation	01 FEB 2019
Time of Report Creation	1:28:49 PM

Metrics

DO-178 B/C (Avionics) Level C

Unit	Subprogram	Complexity	Statements
manager	Manager::Manager	1	2 / 2 (100%)
	Manager::AddIncludedDessert	4	3 / 6 (50%)
	Manager::PlaceOrder	5	11 / 18 (61%)
	Manager::ClearTable	1	0 / 1 (0%)
	Manager::GetCheckTotal	1	0 / 4 (0%)
	Manager::AddPartyToWaitingList	3	0 / 7 (0%)
	Manager::GetNextPartyToBeSeated	2	0 / 3 (0%)
TOTALS	7	17	16 / 41 (39%)
GRAND TOTALS	7	17	16 / 41 (39%)

Industry Mode

Changing Industry Mode

If the current Industry Mode is Default, and the current coverage type is Branch, Basis Path, MC/DC, Function, or Function + Function Call, when the Industry Mode is changed the current coverage type is retained. However, the coverage choices in **Coverage => Initialize** or in the Build Options step of the Update Wizard will reflect the industry specific coverage types

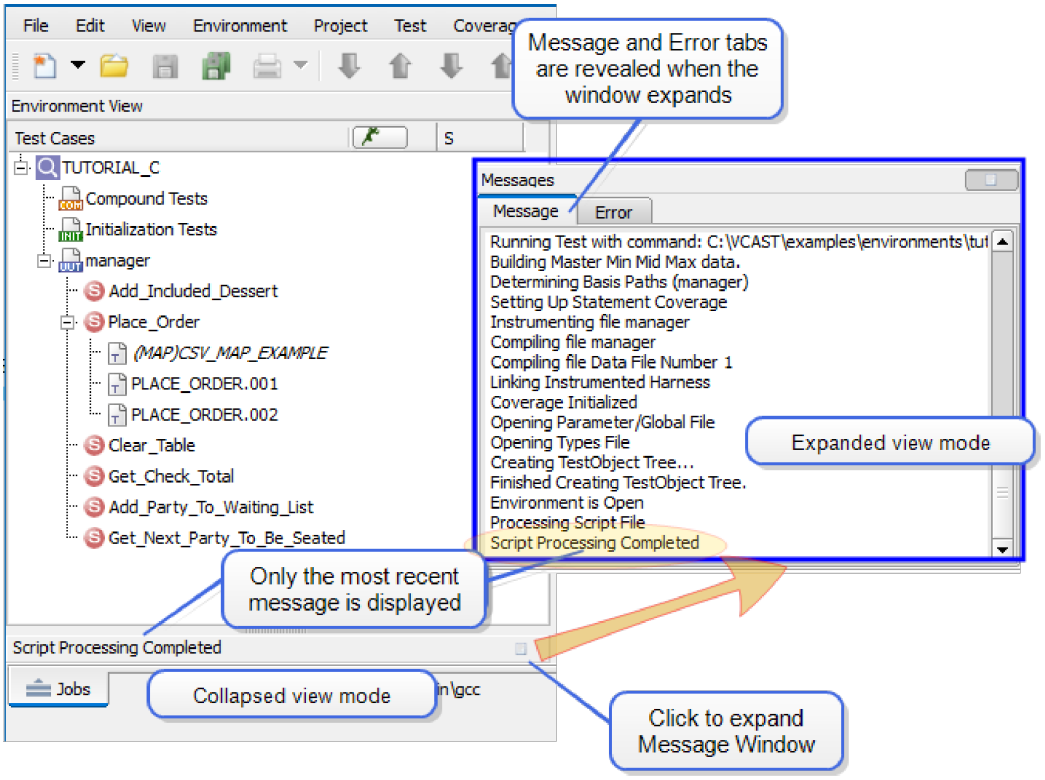
If the environment is rebuilt (**Environment => Rebuild Environment**) after changing the Industry Mode from default, the Branch, Basis Path, MC/DC, Function, or Function + Function Call coverage type is retained.

If the environment is updated (**Environment => Update**) after changing the Industry Mode from default, the Branch, Basis Path, MC/DC, Function, or Function + Function Call coverage type is set to None.

For environments that were created with a version of VectorCAST prior to 6.0, Level A, Level B, and Level C coverage types will be mapped as if the Industry Mode was set to Avionics prior to the Industry Mode change.

To Collapse / Expand the Message Window

The Message window can operate in a collapsed, single line mode. When the Message window is "collapsed" to show only a single line, it takes less space vertically. Only the most recent message is displayed, and the Message and Error tabs are hidden until the window is expanded.

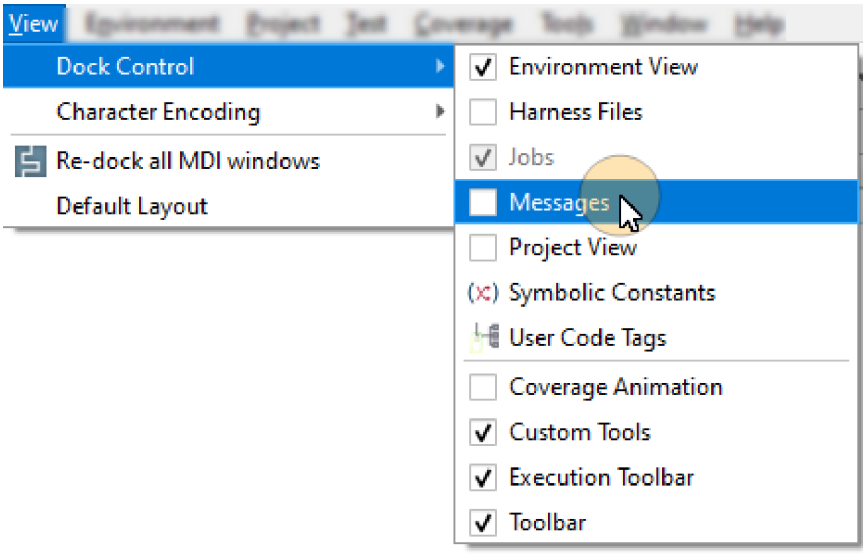


To expand the Message window to see all of the text in the Message window, click the small  button located on the right side of the single line message. This action can be done almost at any time, and is specifically permitted between execution of multiple tests.

The expanded Message window consists of the following tabs: **Message** and **Error**. The **Message** tab displays user status messages. The **Error** tab displays VectorCAST diagnostic messages.

To Hide the Message Window

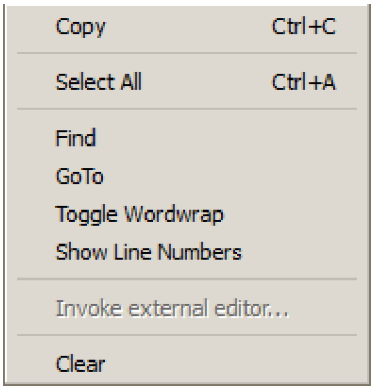
To hide the Message window in the main window, choose **View => Dock Control** and uncheck **Messages**.



To make it visible again, choose **View => Dock Control => Messages**, or **View => Default Layout**.

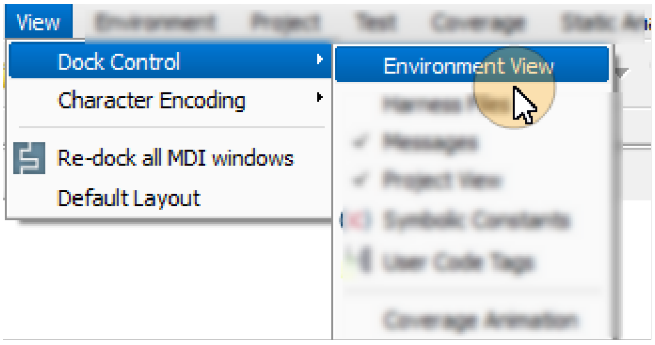
To Clear or Copy Text in Message Window

You can Copy, Select All, or Clear the text in the Message window by right-clicking in the Message window.



To Hide the Environment View

To hide the Environment View in the main window, choose **View => Dock Control** and uncheck **Environment View**.



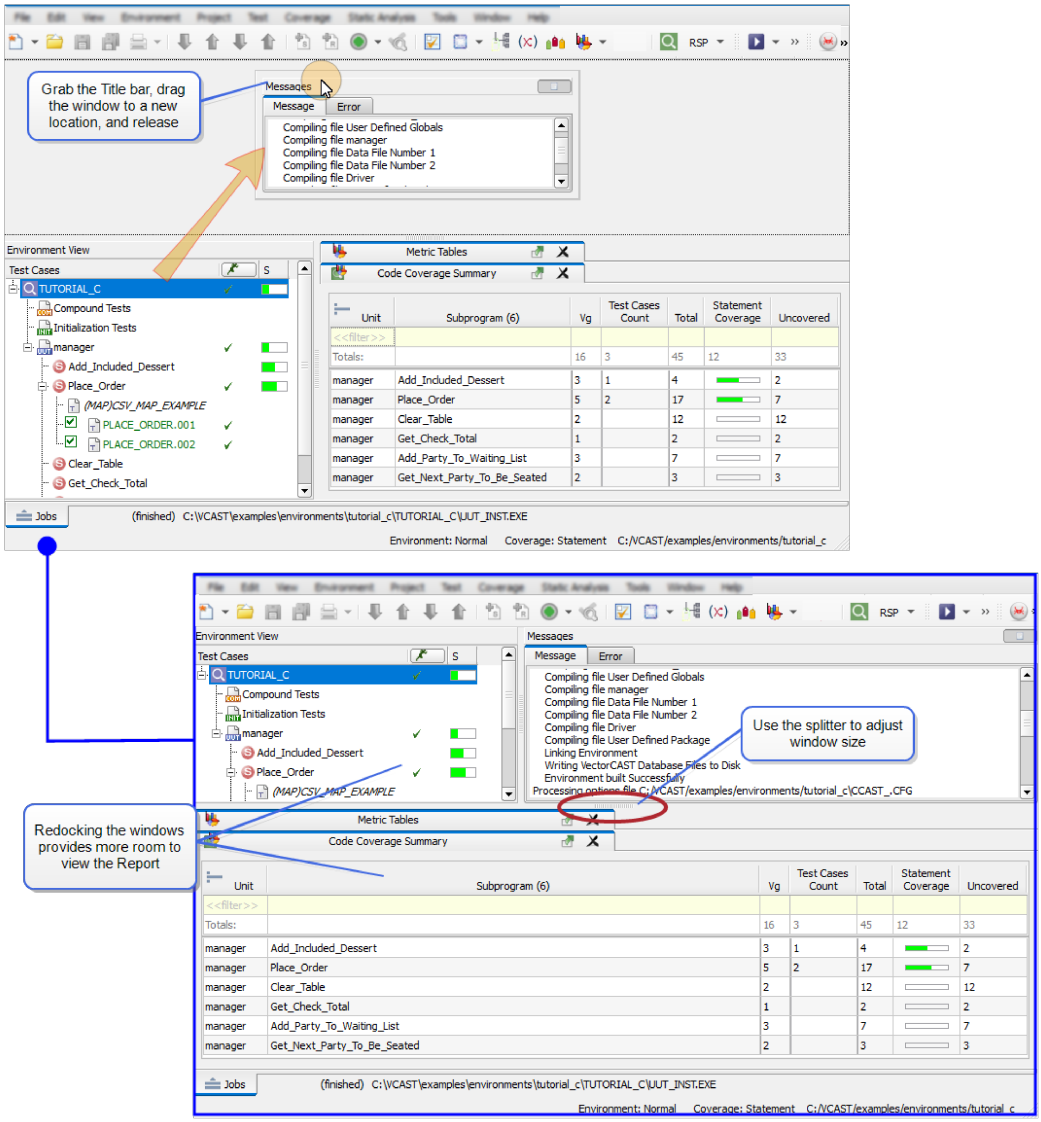
To make it visible again, choose **View => Dock Control => Environment View**, or **View => Default Layout.**

To Move Docking Windows

The Test Case Tree and Message Window are docking windows, meaning they can be moved within the main application window or float outside of the application window.

To move the Message Window or the Test Case Tree to a new location, grab the window by its top Title bar and drop the window within the boundaries of the VectorCAST main window.

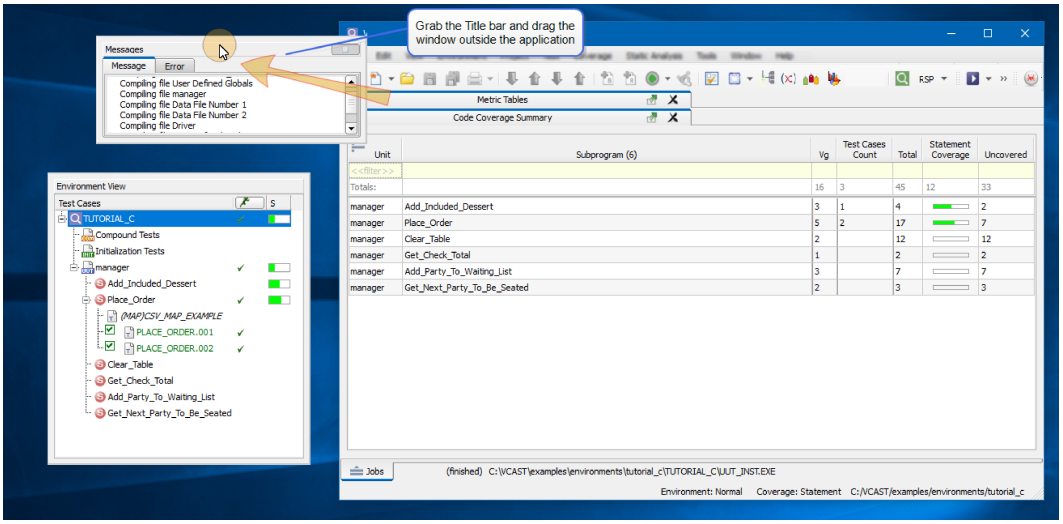
For example, to make more room in the MDI window, you can drag the Message window and Test Case Tree and dock them above the MDI window. To do this, drag the Title bars of the Message window and Test Case Tree and drop them above the MDI window. Use the splitter to adjust the window size.



Use **View => Default Layout** to return the main window to its default layout, if desired.

To Float Docking Windows

To float the Message Window or the Test Case Tree on your desktop, grab the window by its Title bar and move it outside the boundaries of the main window. You can then resize the window, as shown below.



Use **View => Default Layout** to return the window to its default layout, if desired.

To Return Docking Windows to Default Locations

Both the Message Window and the Test Case Tree can be moved from their default locations by dragging them by their Title bars. To return them to their default locations, choose **View => Default Layout**. If the Message Window appears as a column in the middle, just make the main application window a little larger.

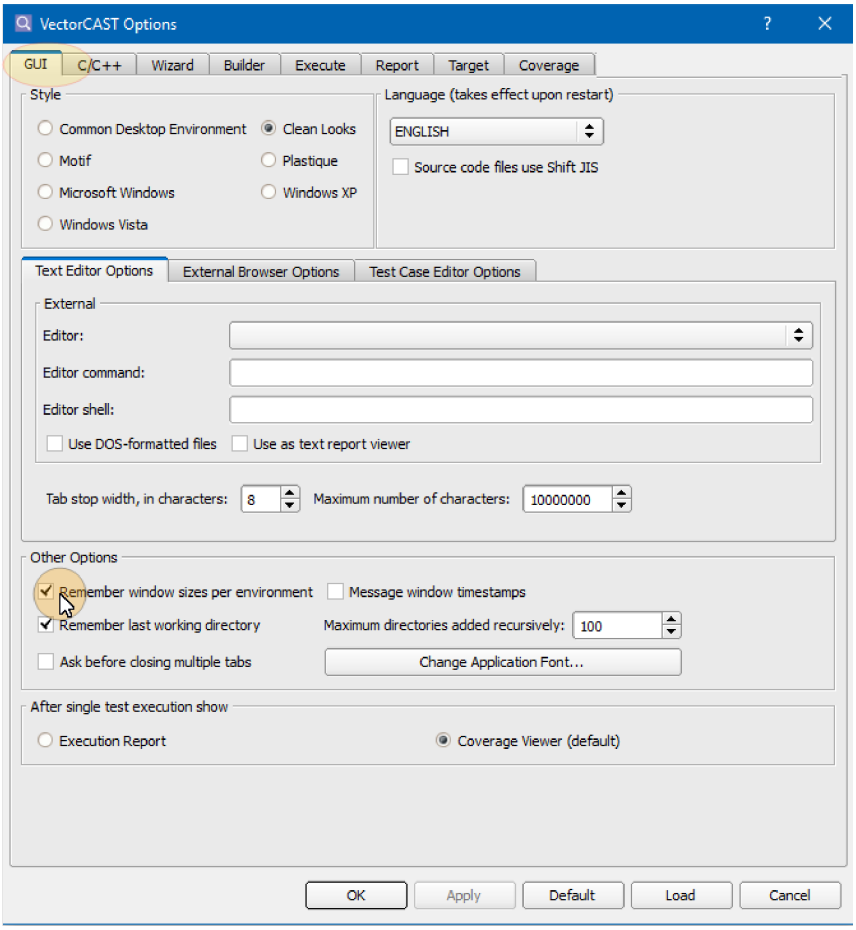
To Save the Window Configuration

When you change the current VectorCAST window configuration, you can save the new configuration for use in all subsequent sessions, or you can save it for use only in sessions involving the same environment.

By default, VectorCAST saves the current configuration for use in all subsequent sessions. In other words, if you change the window configuration, exit VectorCAST, and then return, the window configuration will be the same as it was when you exited the previous session.

To save the current window configuration for use only in subsequent sessions involving the same environment, choose **Tools => Options => GUI tab**, then check **Remember window sizes per environment**.

This information is saved to a file named `.vcast-qt` in the user's HOME directory.



The Toolbar

The Toolbar shows all icons associated with the various VectorCAST commands. The Toolbar includes icons for third-party tools integrated with VectorCAST. The icons are functional only if you are licensed for these features.

By default, the Toolbar is visible. If you move the Toolbar you can return it to its default position by choosing **View => Default Layout**.

Toolbar Icon	Description	Comment
	Create new environment	
	Open new file or environment	
	Save the current file in MDI	Ctrl+S
	Save all	
	Print	
	Next matched result	Ctrl+Down

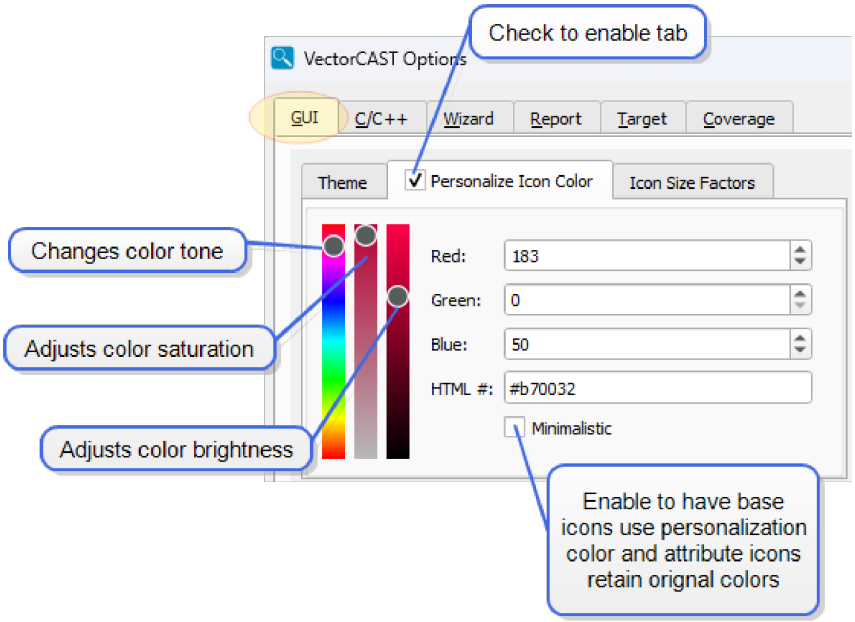
Toolbar Icon	Description	Comment
	Previous matched result	Ctrl+Up
	Next unmatched result	Shift+Down
	Previous unmatched result	Shift+Up
	Add source file	
	Add test result	
	Edit Probe Points	
	Add Coverage Analysis	
	Tool Options	
	Open Requirements Gateway	
	User Code Tags	
	Symbolic Constants	
	Create and assign Variant Logic to test cases	
	Data Summary Report	
	View Coverage	
	RSP documentation	
	Execute test case	
	ATG	
	Bug Tracking	
	Command Prompt	
	Generate CSV Execution Report	
	Settings Database	
	RSP Configurator	
	RSP Installer	

Customize Icons

Users can personalize the color and size of icons within the graphical user interface. This ability to adjust icon colors and to scale subsets of icons relative to the application font size improves accessibility and usability for the user. These settings are saved to the `.vcast-qt.ini` file, located in the user's home directory, and persist across sessions for a consistent user experience.

Personalize Icon Color

To change the color of the icons, choose **Tools => Options**, and select the **GUI** tab. Select the **Personalize Icon Color** tab and insert a checkmark to enable the tab. Once the tab is enabled, customize the colors by adjusting RGB values or using a HEX code. Enter the desired values directly in the fields provided. Move the associated sliders to adjust the color as needed. The slider in the first position changes the color tone. The slider in the second position adjusts color saturation. The slider in the third position adjusts color brightness. Enabling the Minimalistic checkbox will have all base icons use the selected personalization color, while attribute icons, such as exclamation marks or checkmarks, will retain their original colors.

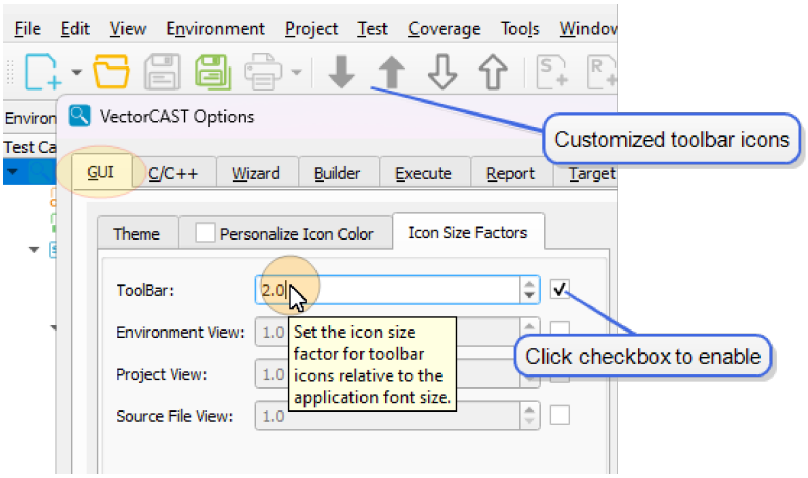


Icon Size Factors

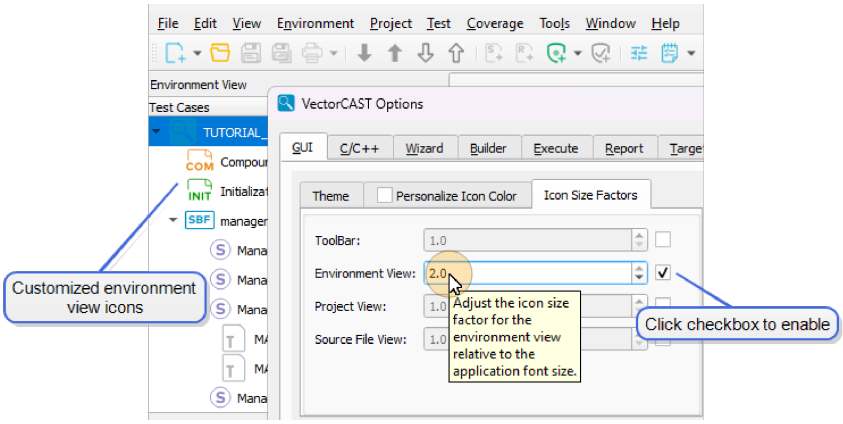
To change the size of the icons, choose **Tools => Options**, and select the **GUI** tab. Then select the **Icon Size Factor** tab. Customized icon sizing is enabled and disabled by selecting the checkboxes. Insert a checkmark to enable editing of the desired view. Enter the desired size factor in the field provided for the selected view. Icon size is determined by multiplying the application font size by the selected factor.

Icon size customization is available for four views:

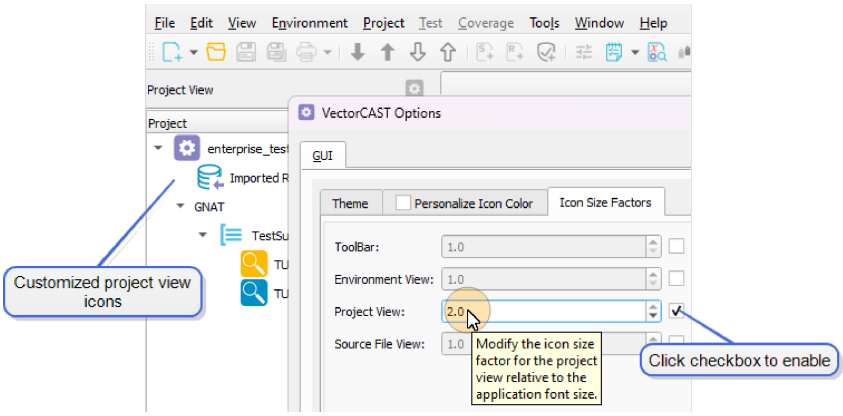
1. Toolbar



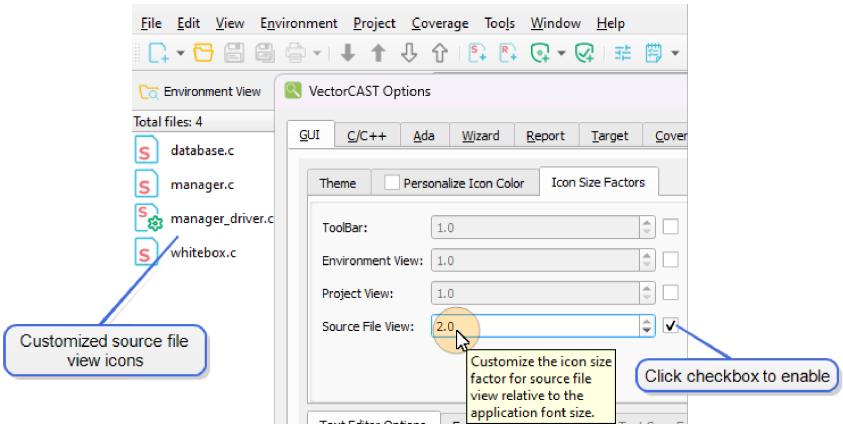
2. Environment View



3. Project View

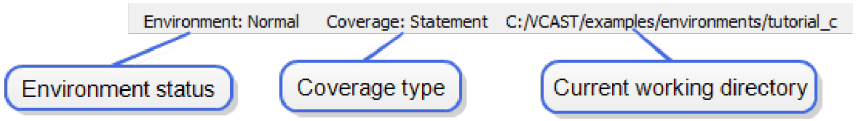


4. Source File View



The Status Bar

When an environment is open, the Status bar shows the status of the Environment, the type of Coverage initialized, and the current working directory.



The environment's status should be "Normal"; if an error occurs when building or rebuilding an environment, the status changes to one of the following:

- Compile Error – An error occurred while compiling the test harness. You will need to correct the problem and recompile the environment. Choose **Environment => View => Compile Errors** to view the compile errors.
- Link Error – An error occurred while linking the test harness. You will need to correct the problem and relink the environment. Choose **Environment => View => Link Errors** to view the linker errors.
- Data Interface Error – The test harness linked but generated a catastrophic error during program elaboration.

If coverage has not been initialized, then the Coverage status is "None." If you have initialized any of the types of coverage (statement, branch, etc.), then the status is changed to "*Type*," where *Type* is one of the types of coverage. Note that if you disable coverage, the status changes to "*Type* (disabled)". Enabling coverage removes "disabled" from the status.

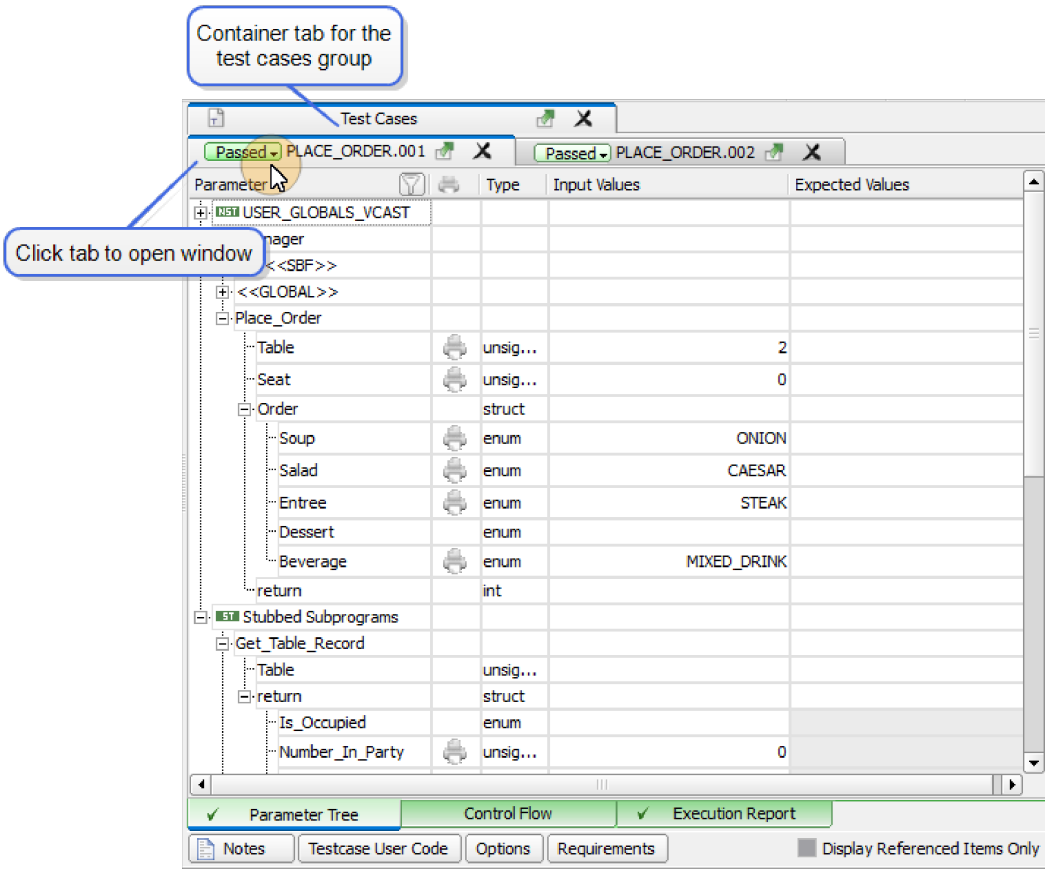
The right-most section of the status bar shows the full path to the current working directory. It is the default directory in the File Save dialog when saving a report or exporting a script to a file, and the default directory in the File Open dialog when opening a file or importing a script.

The MDI Window: Groups

The MDI window is always visible. When an environment is open, it is used to display a variety of windows, including Test Case editors, Coverage Viewers, Report Viewers, and Text Editors. The various types of windows are collected into *groups;* as you open a new window it is added as a tab to

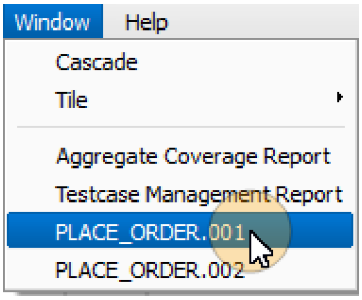
an existing group, or a new group is created.

You can switch between different windows in the group by clicking the named tab at the top.



To Open a Group

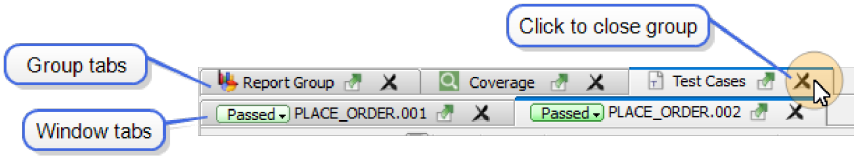
The lower half of the Window menu contains a list of open windows. To switch between windows, choose **Window => <window name>**.



To Close a Group

To close a group, click the X in the upper right corner of the group tab. Closing a group closes all the

tabs in that group. If any tabbed window requires saving before it can be closed, you are asked if you want to save it.

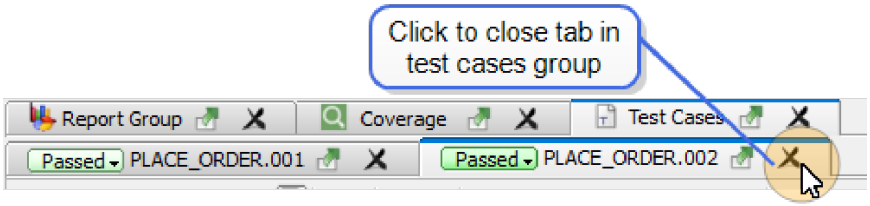


To Maximize a Window

You can maximize a window by un-docking it and clicking the maximize button in the title bar of the window.

To Close a Tab

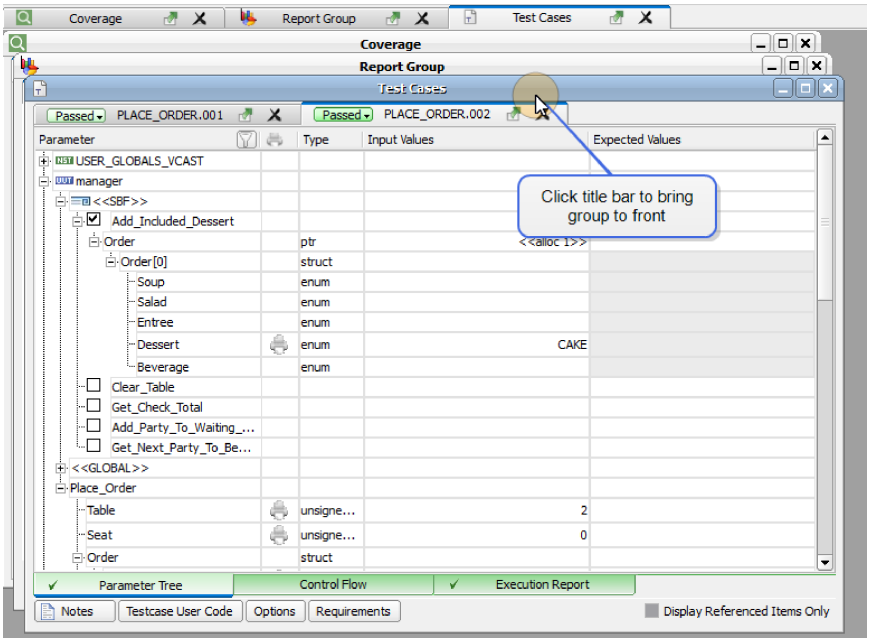
To close a tab in a group, click the **X** in the tab. Alternatively, right-click a tab and choose **Close** from the context menu.



Alternatively, selecting the **File => Close** command from the Menu Bar enables you to close the current active tab in the MDI window. Note that It does not close the environment, only the tab.

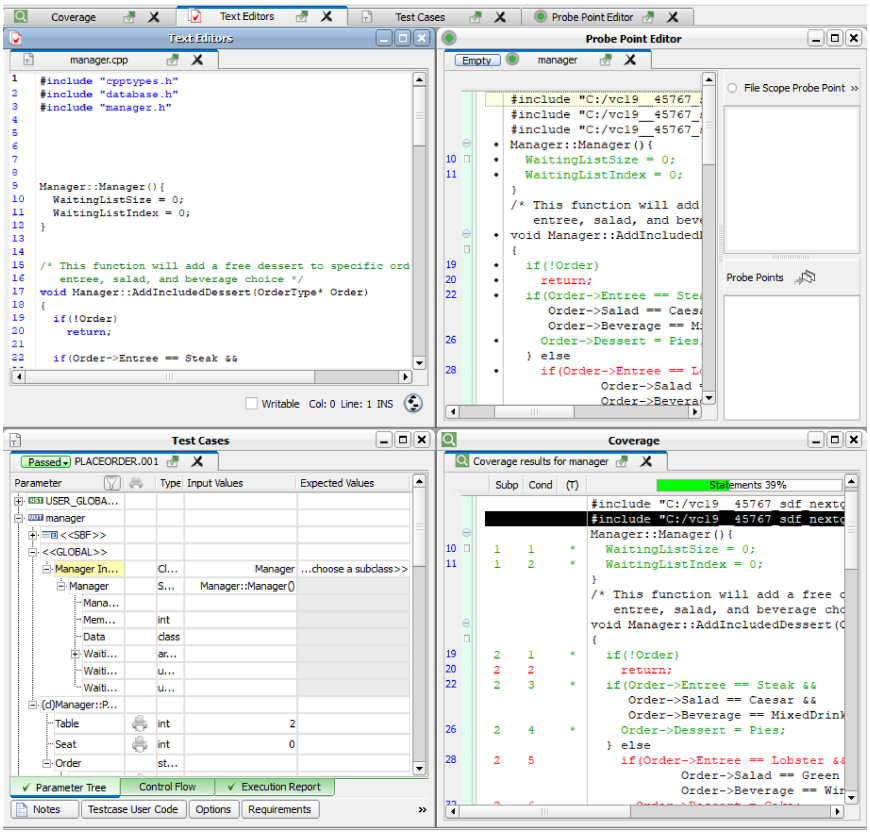
To Arrange Groups in a Cascade

Choose **Window => Cascade** to layer all open groups, showing each title bar. To bring a group to the front, click its title bar.

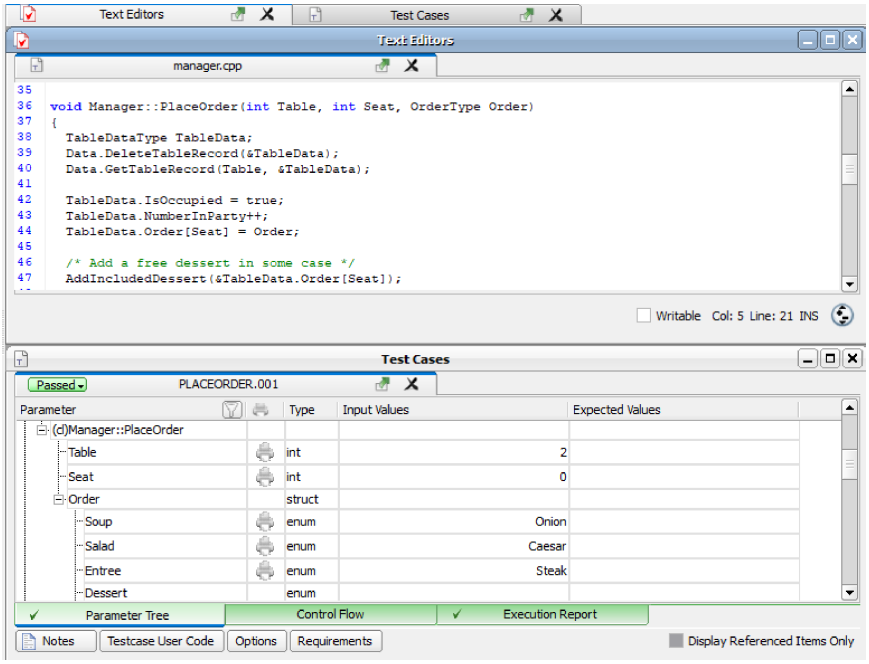


To Tile Groups

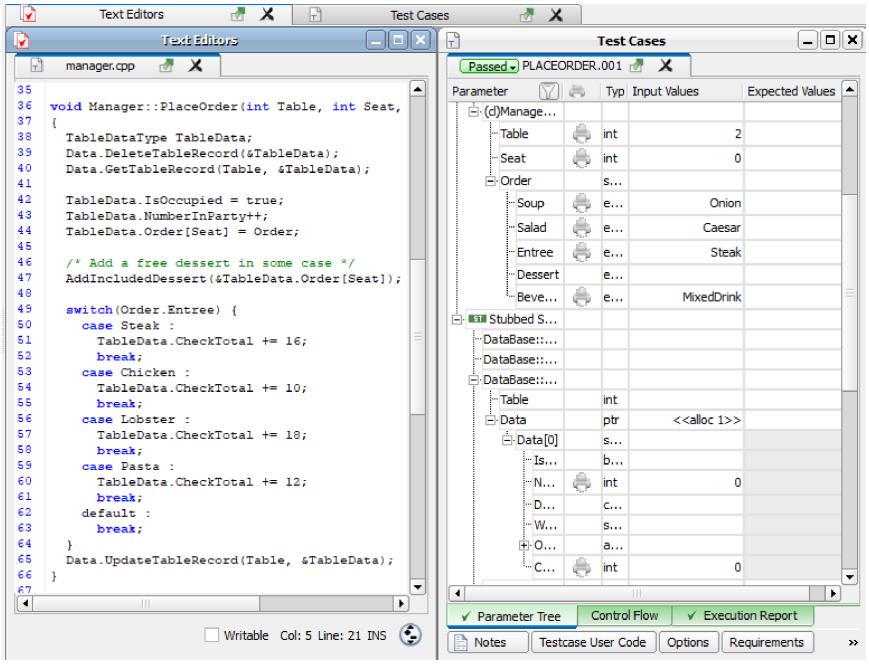
The **Window => Tile => Grid** menu item arranges all open windows in a grid pattern. Each window is reduced to a rectangle, and set next to another window. There is no overlap or layering. The keyboard shortcut is Ctrl+Alt+G.



Choose **Window => Tile => Horizontal** to arrange the groups horizontally. The figure below shows two groups tiled horizontally. The keyboard shortcut is Ctrl+Alt+H.

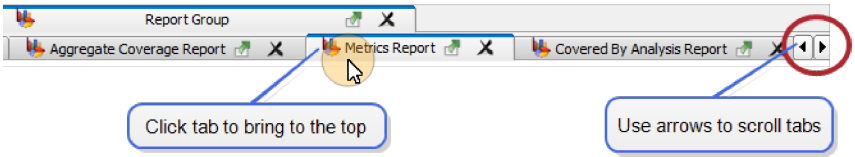


Choose **Window => Tile => Vertical** to arrange the groups vertically. The keyboard shortcut is Ctrl+Alt+V.



To Bring a Tab to the Top

A group may have many tabs open in it. To bring a particular tab to the top, you can dlick its tab. When a window is in focus, its tab is highlighted. If a tab is not visible, use the left and right arrows to scroll the tabs to the left or right.



To Close All Windows

To close all open tabs and groups in the MDI, choose **File => Close All**. This menu item does not close the environment.

VectorCAST Keyboard Shortcuts

Tiling Window Groups	
To Tile Groups Horizontally	**Ctrl+Alt+H**
To Tile Groups Vertically	**Ctrl+Alt+V**
To Tile as a Grid	**Ctrl+Alt+G**

Editing, Searching and Printing

Create a New Text File	**Ctrl+T**
Go to a Line	**Ctrl+G**
Select All	**Ctrl+A**
Delete	**Ctrl+Del**
Undo	**Ctrl+Z**
Redo	**Ctrl+Y**
Cut	**Ctrl+X**
Copy	**Ctrl+C**
Paste	**Ctrl+V**
Save	**Ctrl+S**
Find	**Ctrl+F**
Repeat a Search	**F3**
Print	**Ctrl+P**

Expand / Collapse Windows

Available in a Unit Test environment, Cover environment, or VectorCAST project:

Expand / Collapse Message Window	**Ctrl+Alt+m**
Expand / Collapse Jobs Window	**Ctrl+Alt+j**

Available in a Unit Test environment or Cover environment that has been opened from within a VectorCAST project:

Expand / Collapse Project View	**Ctrl+Alt+p**

Building Test Cases

Available from Test Case Tree:

Insert a Test Case	**Ctrl+Insert**
Open a Test Case	**Ctrl+Return**
Open Source	**Ctrl+Return**
Rename a Test Case	**Ctrl+Shift+R**
Rename an Environment	**Ctrl+R**
Expand a collapsed node	**Right Arrow**
Collapse an expanded node	**Left Arrow**

Available from Compound Test Case Tree:

Open an Individual Test Case from a Compound Test Case	**Ctrl+Shift+O**
Delete a Test Case from a Compound	**Del**

Move Slot Up	**Ctrl+Shift+Up**
Move Slot Down	**Ctrl+Shift+Down**
Find Slot in the Test Case Tree	**Ctrl+Shift+F**

Available from the Test Case Editor:

Expand a collapsed node	**Right Arrow**
Collapse an expanded node	**Left Arrow**
Display list of enum items for a selected parameter	**Alt+Down Arrow**
Switch between Input Value and Expected Value	**Tab**

Executing Test Cases

Batch Execute	**F6**
Edit Properties	**Shift+P**

Available from the Test Case Tree:

Execute	**F5**

Working With Control Flow

Available from the Control Flow tab in the Test Case Editor:

Move Up	**Ctrl+Shift+Up**
Move Down	**Ctrl+Shift+Down**
Remove Subprogram	**Del**

Working With Coverage

Available from the Test Case Tree:

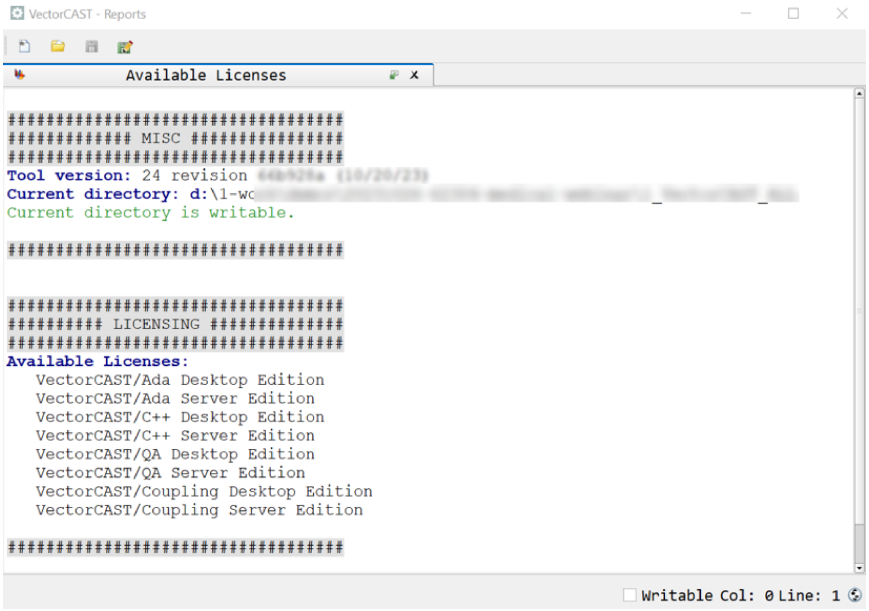
View Coverage	**Shift+Return**
Select Coverage	**Ctrl+Shift+S**
Deselect Coverage	**Ctrl+Shift+D**

Exit VectorCAST	**Ctrl+Q**

Getting Help

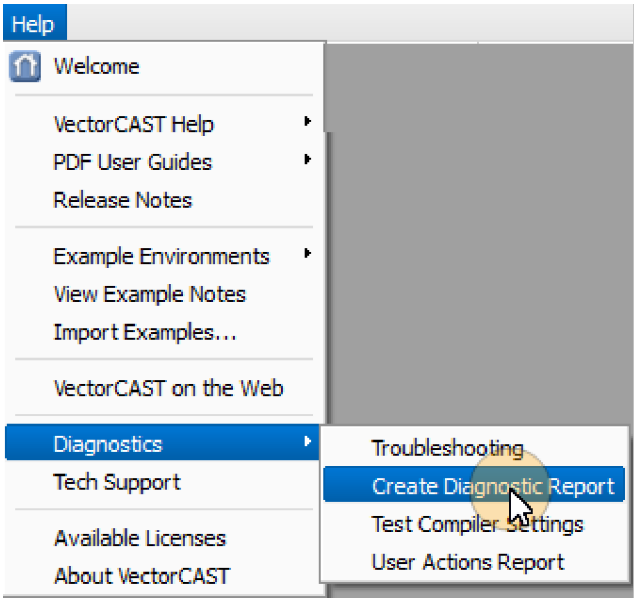
To Determine Available Licenses

To determine which VectorCAST products your organization is licensed to use, choose **Help => Available Licenses**:

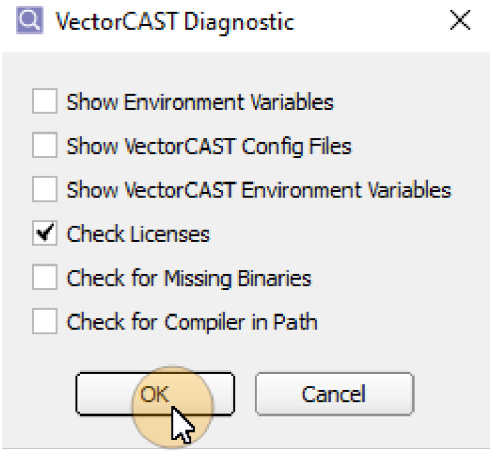


An Available Licenses Report is displayed in the MDI window, listing all available licenses.

Alternatively, to determine which features and targets are licensed, choose **Help => Diagnostics => Create Diagnostic Report**:



Check **Check Licenses** on the Diagnostics popup, click **OK**, and then scroll down the report to the Licensing section.




```
clicast -lc REports DIagnostic [<outputfile>]

Generates diagnostic report to standard output or the specified output
file.
```

To View Online User Guides

The Help menu gives you access to the various VectorCAST User Guides in PDF and HTML format. Adobe Acrobat Reader must be available on your system in order to view the PDF files. The following documents are available:

- Release Notes, which include the changes and improvements to VectorCAST since the last released version.
- User Guides for all VectorCAST products.

The User Guide .PDF files are located in the VectorCAST installation directory, **`/docs/PDF`** sub-directory, and can be accessed by selecting **Help =>PDF User Guides** from the Menu Bar. Note that only User Guides for licensed products are available from the menu.

.PDF filename	User Guide
vcast_quick_start.pdf	*Quick Start Guide*
vcast_enterprise.pdf	*Enterprise Testing With VectorCAST*
vecchelp.pdf	*VectorCAST/C++ User's Guide*
veachelp.pdf	*VectorCAST/Ada User's Guide*
vcqa.pdf	*VectorCAST/QA User's Guide*
vcast_utilities.pdf	*VectorCAST Utilities User's Guide*

.PDF filename	User Guide
vcast_install_guide.pdf	*VectorCAST Installation Guide*
fnp_LicAdmin.pdf	*License Administration Guide*

In addition to the PDF versions of the User Guides, VectorCAST includes HTML Help. Use the **Help =>VectorCAST Help** menu item to access the HTML guides. After selection, the guide opens in a VectorCAST browser window which is detached from the main VectorCAST GUI. If multiple help guides are opened, each will exist in a separate tab in the browser window.

To Contact Technical Support

Choose **Help => Tech Support** to find contact information for VectorCAST Technical Support.

Email: **support@vector.com**
Web: **https://support.vector.com**

To Send a Test Environment to Technical Support

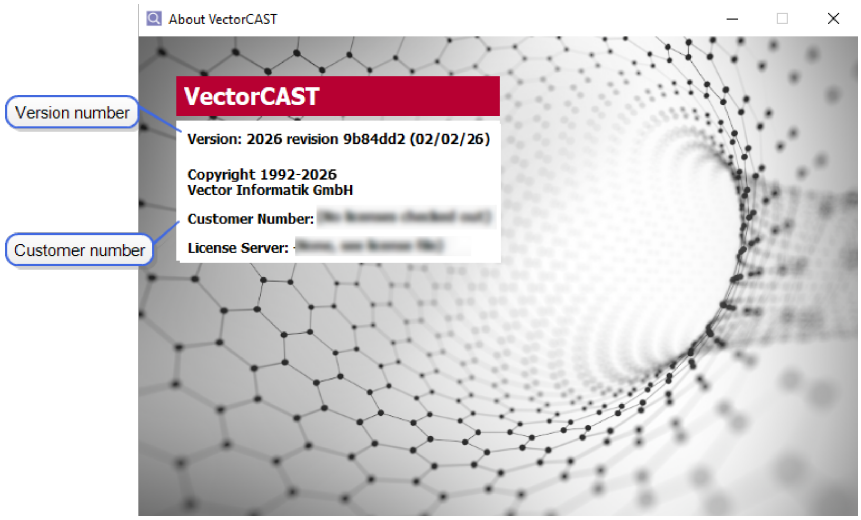
Sometimes, Tech Support may need to review your test environment to answer a question. The easiest way to send your test environment is to compress it (zip, tar, or gzip) and email it to **support@vector.com**.

If your environment is named TEST, for example, you should send the entire contents of the directory named TEST along with the file TEST.vce. Under Windows, you can use PKZip or WinZip; under Linux, you can use tar and compress or gzip to create an archive file that can be emailed to Tech Support.

If the size of the compressed file is greater than 5 MB, please send an email to support@vector.com and they will instruct you on how to send the file.

To Determine the VectorCAST Version and Customer Number

Choose **Help => About VectorCAST** to determine VectorCAST's version number and build date. Your Customer Number is also displayed.

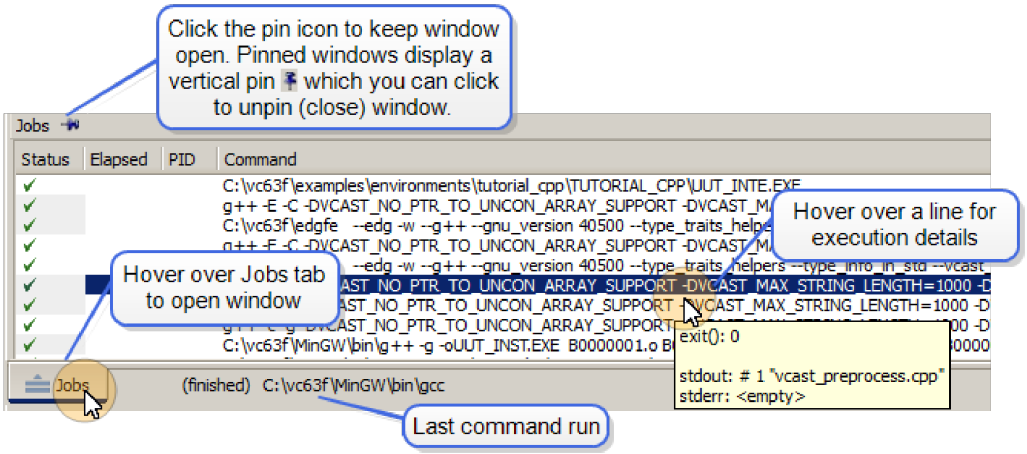

About VectorCAST
VectorCAST
Version: 2026 revision 9b84dd2 (02/02/26)
Copyright 1992-2026
Vector Informatik GmbH
Customer Number:
License Server:
Version number
Customer number

Using the Jobs Window and Jobs Monitor

It is often desirable when running a target test to view the output in real time as the commands are running. The Jobs window displays jobs as they execute and the Jobs Monitor displays the status of each job as it executes and exposes the toolchain processes. The Jobs Monitor also allows the user to diagnose failing commands and test a possible fix.

The Jobs Window

The Jobs window is an auto-hide window located at the bottom of the main window. If the Jobs window is hidden, the status bar displays the current or last command run. Hover over the window to open it, and use the pin icon to keep the window open.



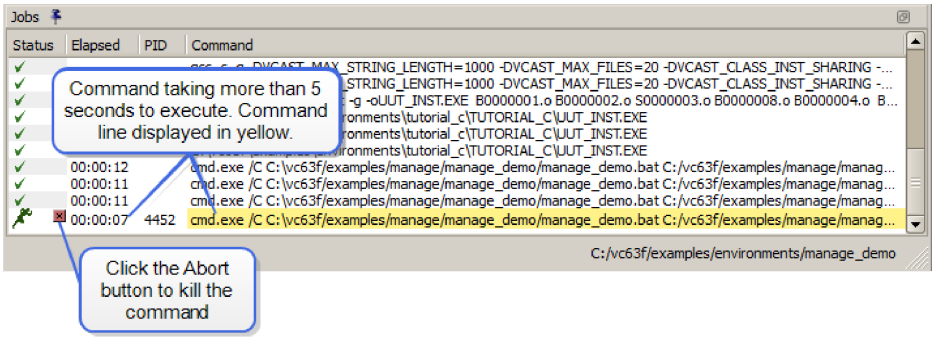
The Jobs window displays the full command line for every invocation of an executable by VectorCAST's back end. Such executables include the compilers, .bat files, python and any other commands called while building the environment.

Single-clicking on a command line highlights the associated line in the Messages window. Hovering over a command line shows the exit code and stdout and stderror for that line.

For each command, the status, execution time elapsed and Process ID (PID) is displayed. Icons in the Status column represent the following:

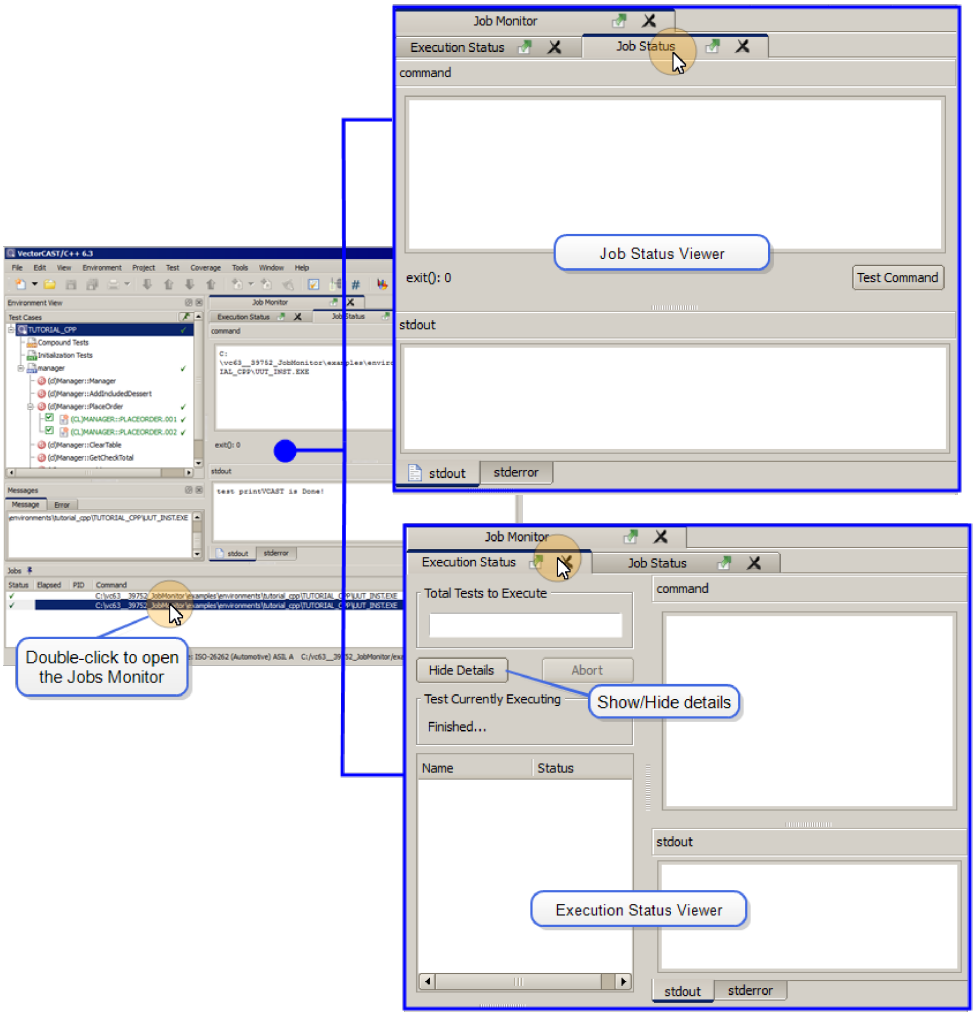
- = exit() code is 0
- = exit() code is non-zero
- = Click to abort job
- = Executing
- = Error
- = Monitored sub-process (used in Verbose mode)

A command taking more than 5 seconds to execute displays a yellow background. Click the Abort button to kill the command if desired.



Opening the Job Monitor

The Job Monitor is accessed via the Jobs window. The Job Monitor contains two panes: the Job Status Viewer and the Execution Status Viewer. The Job Status Viewer displays details about a command. The Execution Status Viewer provides real-time test case execution information.

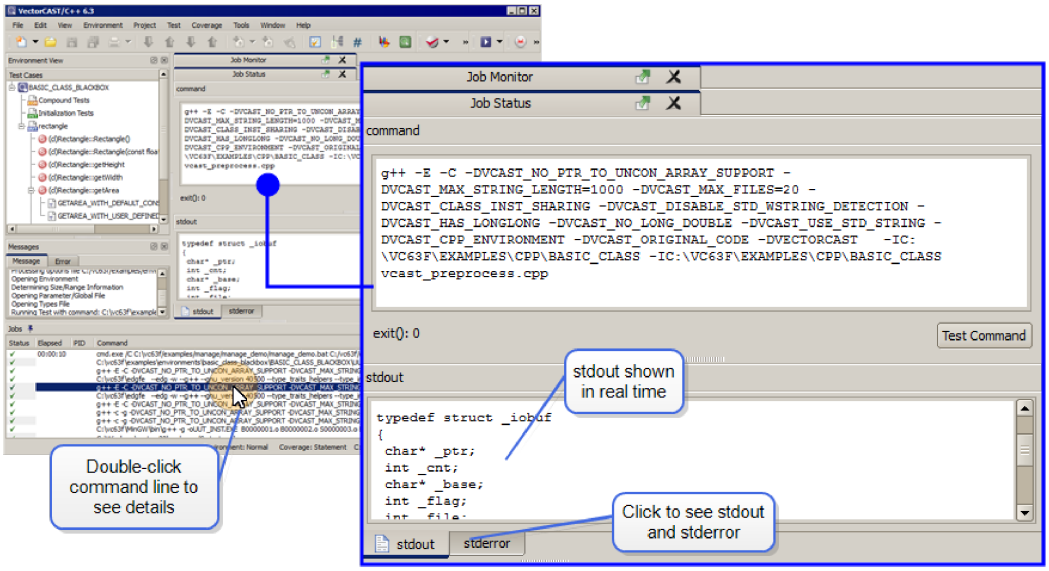


To open the Job Monitor and the Job Status viewer, double-click a command line in the Jobs window or right-click a command line and select **Job Status** from the context menu. The Job Status Viewer opens in the MDI window.

Job Status Viewer

The Job Status viewer displays details about a command and provides a way to diagnose failing commands (or "jobs") and test a possible fix. Each job is an invocation of a program, such as the VectorCAST test harness, the compiler, or linker.

The Job Status viewer is primarily used to diagnose how VectorCAST makes calls to the Target compiler, and provides the user with helpful information about what a command is and where it is run from.



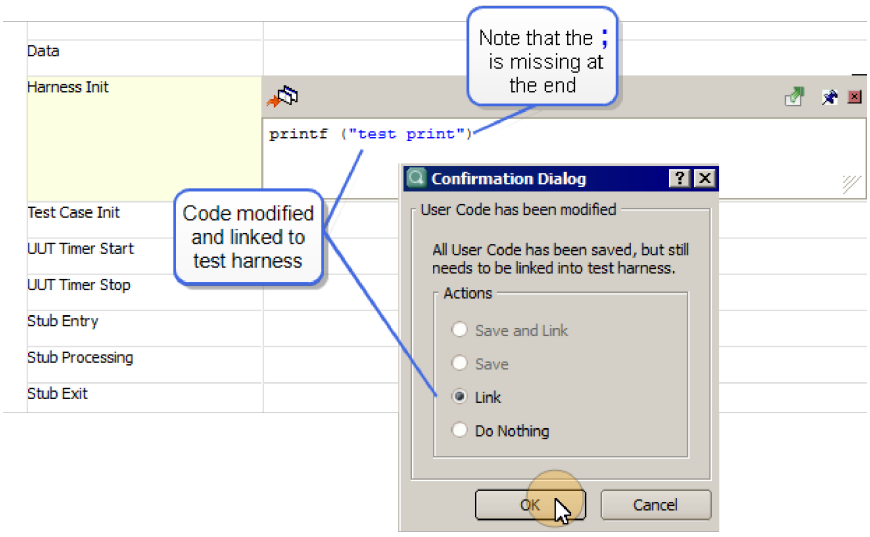
The Job Status viewer displays the executable called, any arguments and full stdout and stderror information. Click a running command to see the stdout and stderror in real time.

Debugging Using the Job Status Viewer

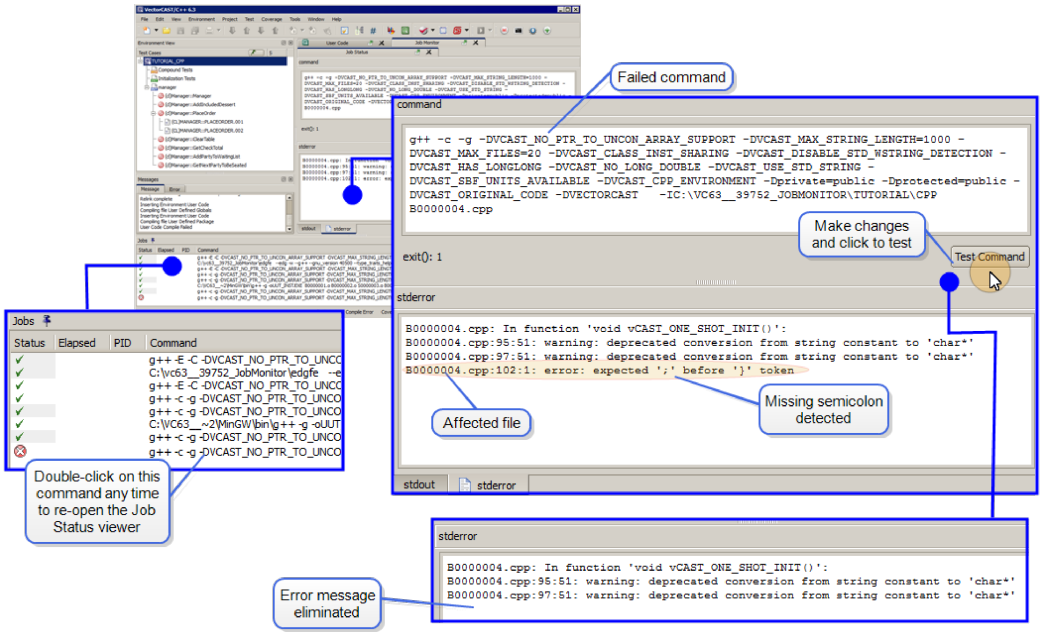
When a compile or link error occurs in user code during test execution, the Job Status viewer opens automatically, showing the command that failed and the standard output and standard error. The command displayed in the Job Status window is editable, and a **Test Command** button is provided to re-run the command.

This debugging feature provides a way to diagnose failing commands. It does not affect the state of the environment or resolve errors caused by the original command failing to run.

For example, say the Environment User Code is modified and a coding error introduced. When the change is saved and linked, the coding error is detected and the Job Status viewer automatically opens.



You can use the information in the standard error tab to determine which test harness file has the problem, make changes to the file, save, and then click the **Test Command** button to try the command again.



Once satisfied that you have the solution to the problem, you still need to make the change in the user code or Compiler template before attempting the test execution again.

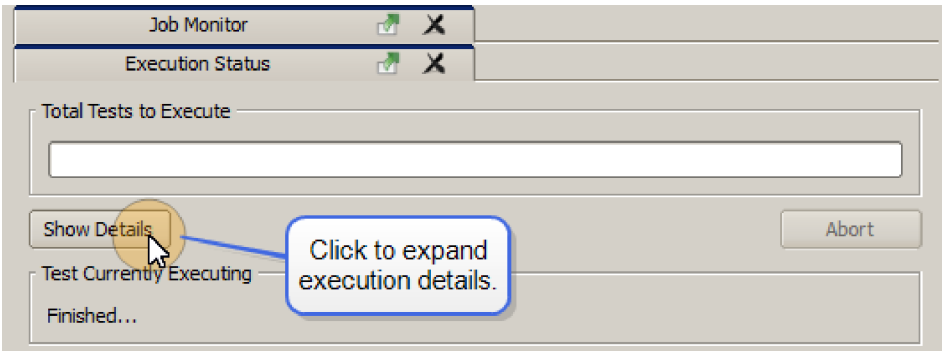
Execution Status Viewer

The Execution Status viewer is the counterpart to the Job Status viewer. The Execution Status viewer

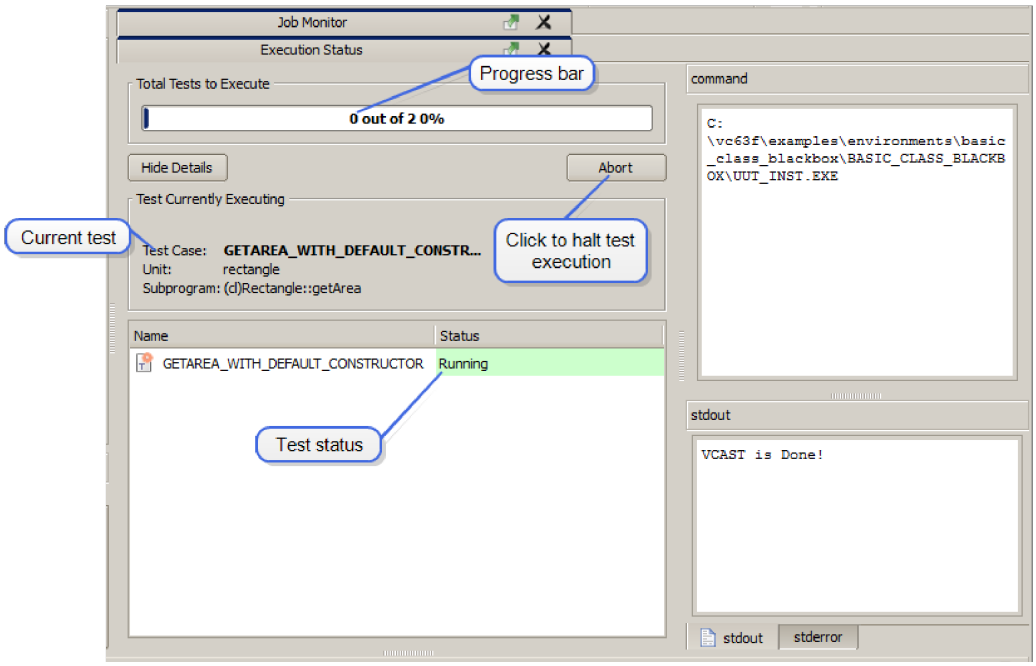
provides detailed real time test case execution information.

The Execution Status viewer opens automatically when a test case is executed. Click the **Show Details** button to display the full details of the execution in real time.

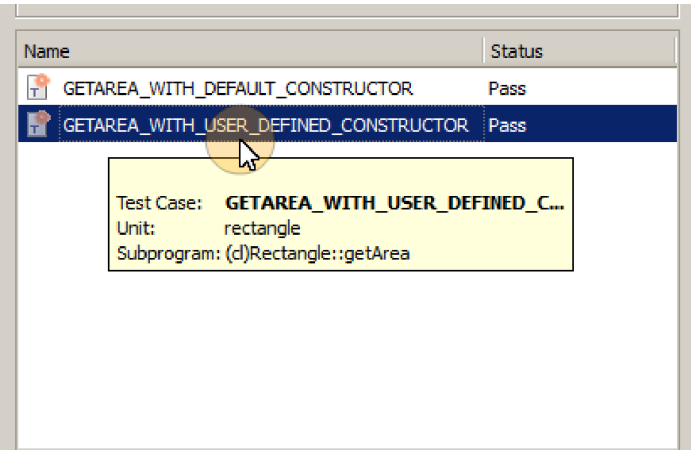
Note: When running hundreds of tests, hiding details can reduce total execution time.



A progress bar is displayed on the top left showing the number of tests remaining to execute. The name of the currently executing test is displayed beneath. An **Abort** button is available to halt the test case execution if desired and return control to VectorCAST.



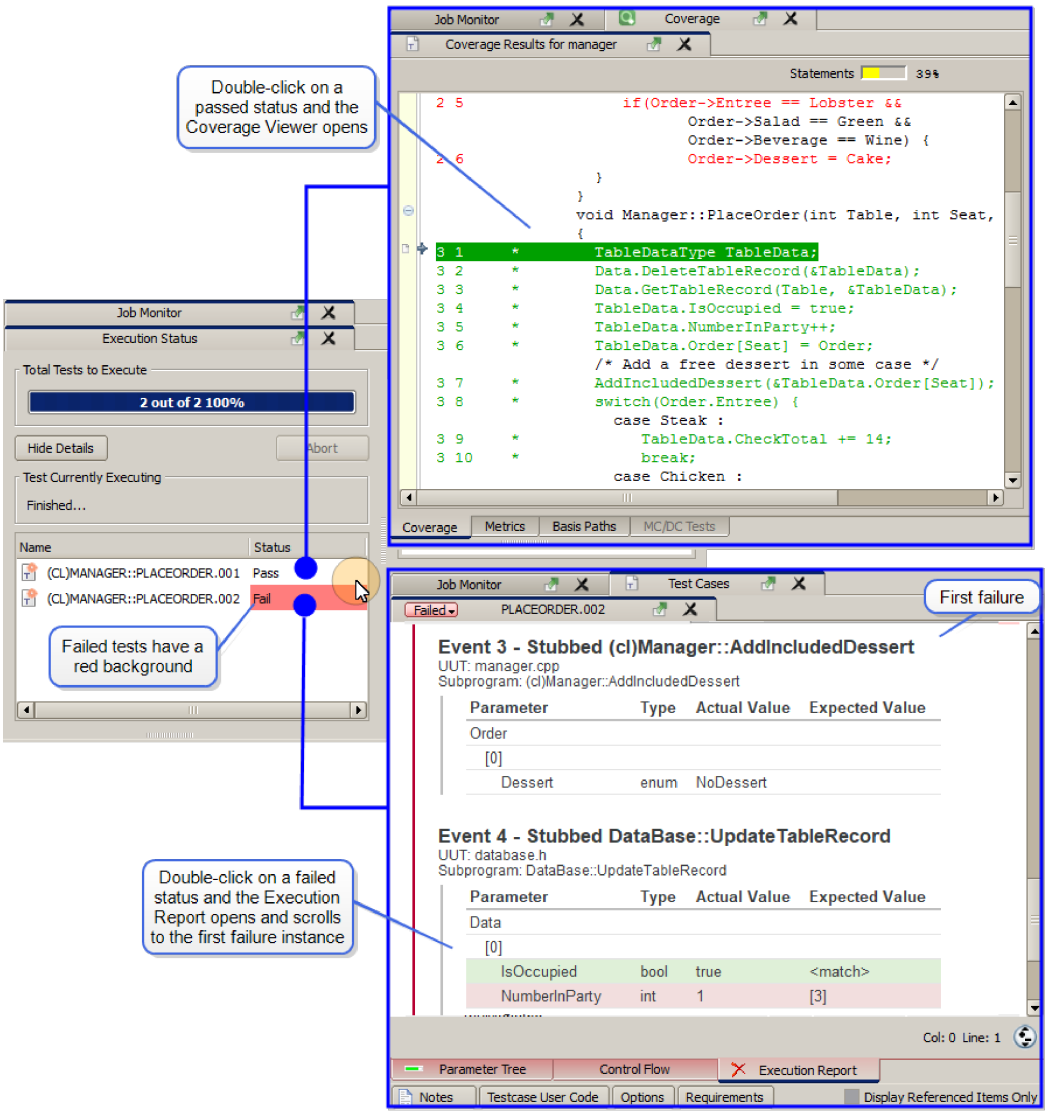
When viewing full execution details, as each test completes it is listed on the bottom left along with its pass/fail status. Hover over the name of a test case to see the name, unit and subprogram.



A deleted test displays an icon to the left of its execution status indicating that it is an invalid result now.

Hover over the status of a test case to see details of the execution. If a test aborts during execution, that information is displayed in the tooltip.

If the environment has coverage and the test passed, double-clicking on a passed status opens the Coverage Viewer. If the environment has no coverage and the test passed, double-clicking on a passed status opens the Execution Report.



Failing test cases are displayed on a red background. Double-clicking on a fail status opens the Execution Report, and the report scrolls to the first failure instance.

The full command line and the stdout and stderror details are displayed in the right column of the Execution Status viewer.

Changing Application Preferences

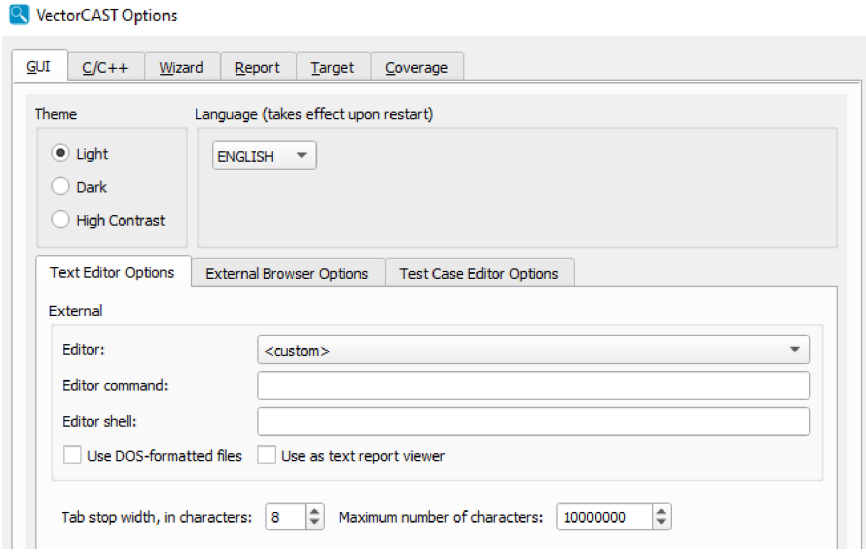
To Set the Display Theme

To select a display theme, choose **Tools => Options**, and click the **GUI** tab. Select the radio button for the desired theme and click **Apply** or click **OK** to close the dialog.

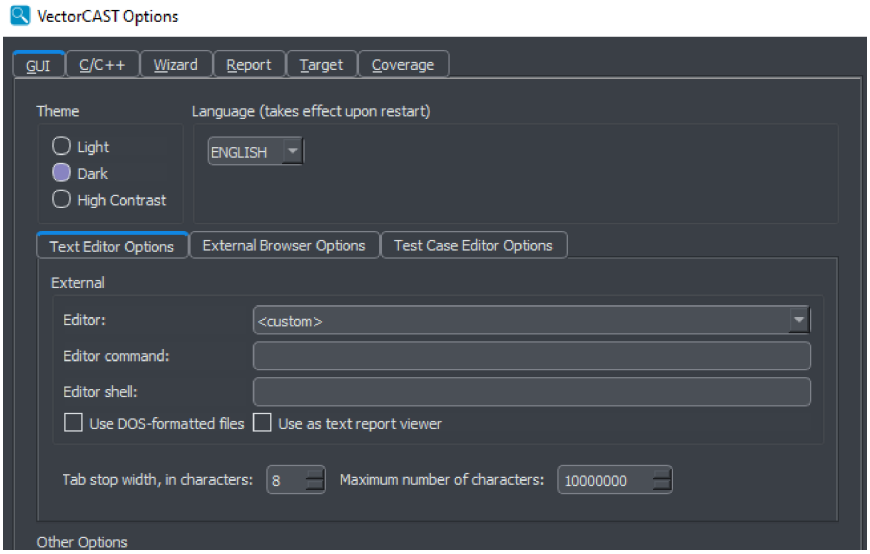
VectorCAST offers three visual display themes for selection:

- Light
- Dark
- High Contrast

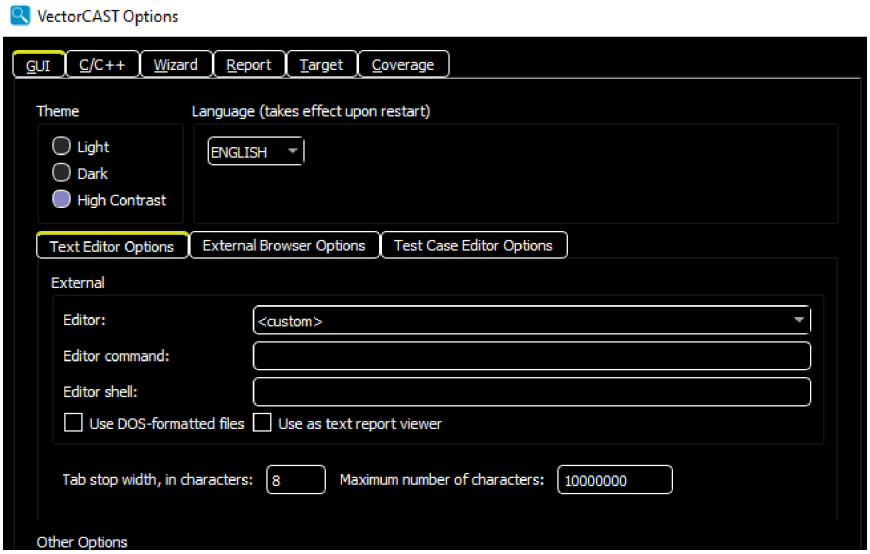
Light mode is the default display theme.



Dark mode reduces eye strain and improves visibility for certain users, particularly those with visual impairments or sensitivity to light.



High Contrast mode improves text legibility and overall visibility for those with low vision or color blindness. High contrast colors are easier to read when paired together.



To Set Up an External Text Editor

By default, all text and HTML files opened in VectorCAST are displayed in the MDI window. If you want to use an alternative editor for text files and text reports, you may configure this by selecting **Tools => Options** from the Help Menu, then selecting the **GUI** tab, **Text Editor Options** sub-tab.

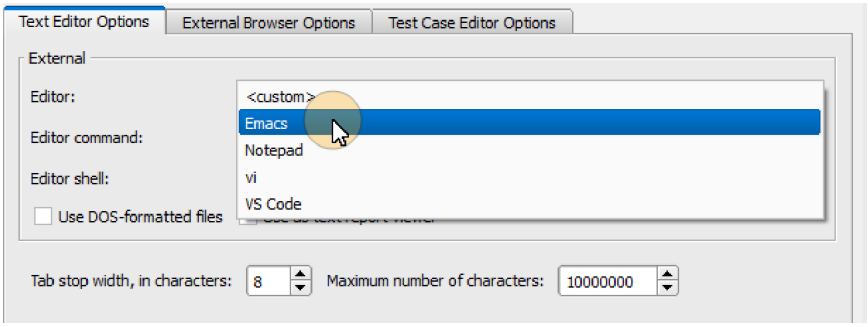
Note: These options are for text files and text reports only. To use an external browser for HTML reports, see "To Set Up an External Browser" on page 66.

The **Editor** drop-down list contains a list of predefined commonly used text editors which can be

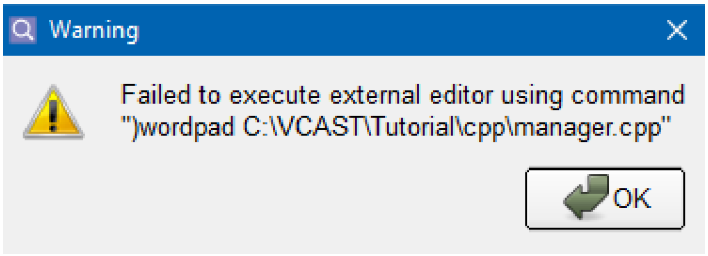
selected as a VectorCAST external editor for text files. Optionally, based on the setting of the "Use as text report viewer" option, the external editor can be used to view text reports.

Text editor options include:

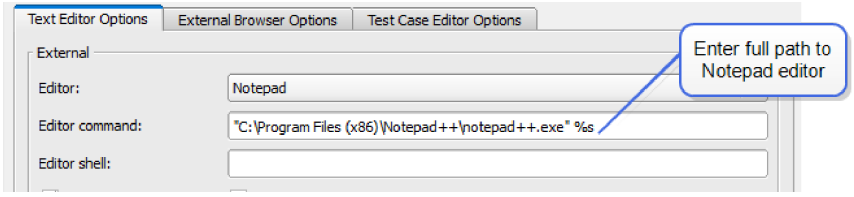
- <custom> (this option allows users to specify their own external editor)
- VS code (Visual Studio Code)
- Emacs
- Notepad
- vi



The executable file for the editor must exist on your PATH environment variable. In cases where the executable of the editor is not on your path, a warning appears:



In this case, you can either modify your PATH environment variable or configure VectorCAST with the full path to the editor executable using the Editor Command option.



Click **Use DOS Formatted Files** if your choice of editor requires that all lines be terminated with a CR + LF (i.e. Notepad).

If you want to use the same editor for viewing text reports, click the checkbox next to **Use as text report viewer**. If this option is set, and you have chosen “Text” as the report format in the Reports tab, then any time you choose a “View” action, the file chosen is opened in the external editor.

When you have made your selection(s), click **Apply** or click **OK** to close the dialog.

This information is saved to a file named .vcast-qt in the user's HOME directory.

Set an External Editor to Scroll to First Line of Function

When an external editor is used to open source files, a script can be used to enable the editor to scroll to the line number of a selected function in a source file. The script uses the environment variable VCV_EDIT_LINE_NUMBER, which is set to the line number of the selected function. The external editor can then be set to a script command that uses the environment variable VCV_EDIT_LINE_NUMBER to go to a specific line number in the external editor.

When opening a file using an external editor in a context where no particular location is requested, such as having selected a unit rather than a subprogram, VCV_EDIT_LINE_NUMBER is set to 0, which opens the file and sets the cursor to the top.

Linux Example

On Linux, a vi user creates a bash script named **`my_start_editor.sh`**, whose contents include:

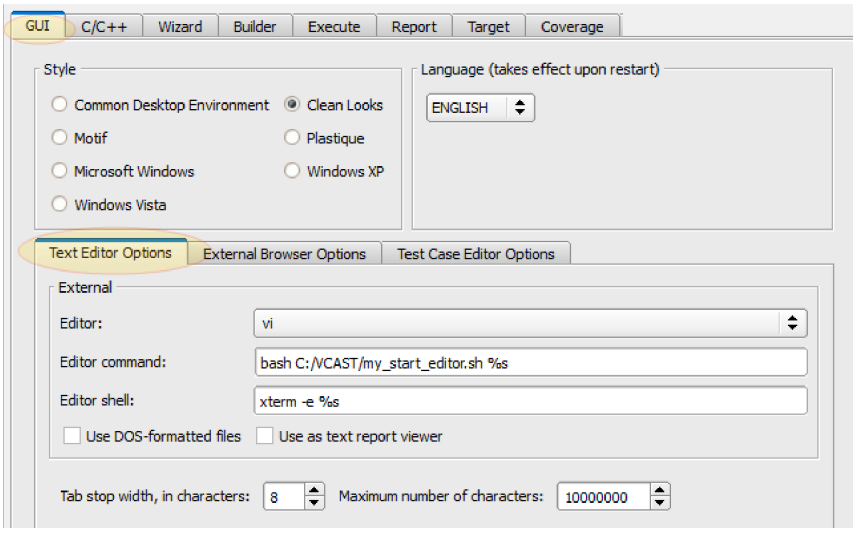
```
#!/bin/bash
vi "$@" +$VCV_EDIT_LINE_NUMBER
```

In VectorCAST's Options dialog, **GUI** tab, **Text Editor Options** sub-tab, the user sets the option **Editor command** to:

```
bash <path to>/my_start_editor.sh %s
```

and sets the option **Editor shell** to:

```
xterm -e %s
```



Then, in a VectorCAST unit test environment, when right-clicking on a UUT's subprogram name and

choosing **Open Source**, **`xterm`** opens with the vi editor correctly scrolled to the first line of the selected subprogram.

Windows Example

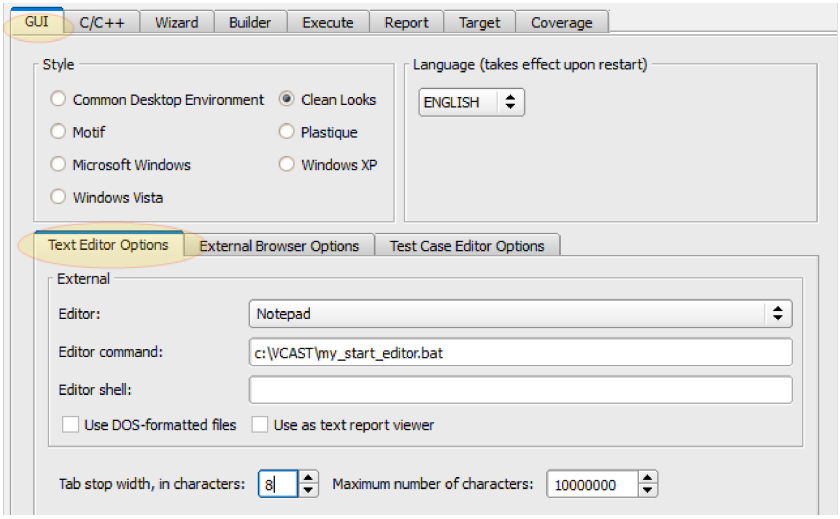
On Windows, a Notepad++ user creates a batch file named **`my _start_editor.bat`**, whose contents include:

```
"<path to>\notepad++.exe" -n%VCV_EDIT_LINE_NUMBER% %1
```

where **`<path to>`** is written similar to **`"c:\Program Files (x86)\Notepad++"`** and enclosed in quotes.

In VectorCAST's Options dialog, **GUI** tab, **Text Editor Options** sub-tab, the user sets the option **Editor command** to:

```
<path to>\my_start_editor.bat
```



Then, in a VectorCAST unit test environment, when right-clicking on a UUT's subprogram name and choosing **Open Source**, Notepad++ opens with source file correctly scrolled to the first line of the selected subprogram.

To Edit User Code with the External Editor

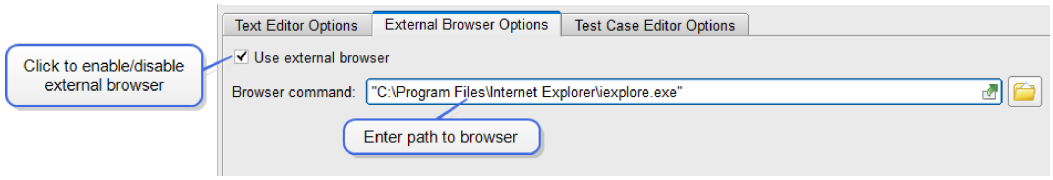
When you are editing user code (in the Configure Stubs dialog, the Test Case User Code tab, Parameter User Code dialog, or Environment User Code dialog), right-click in the text area and select **Invoke External Editor** from the popup menu. The external editor opens and you can edit user code there. When you save and exit the editor, the temporary file is imported into the user code text edit box.

To Set Up an External Browser

By default, all HTML reports opened in VectorCAST are displayed in the MDI window. If you want to use an alternative browser for HTML reports, you may configure this by selecting **Tools => Options**

from the Help Menu, then selecting the **GUI** tab, **External Browser Options** sub-tab.

First, click the checkbox next to **Use External Browser** to enable the feature. Note that checking this box without specifying a browser causes VectorCAST to display all reports, including the test case Execution Results, in a tab in the Report group.



If you would like all reports to be displayed in a separate browser window, enter the path to your HTML browser in the edit box next to **Browser Command**. If none is specified, VectorCAST's internal browser is used. You can set up a browser but temporarily turn it off by un-checking the **Use External Browser** checkbox.

When you have made your selection, click **Apply** or **OK** to close the dialog.

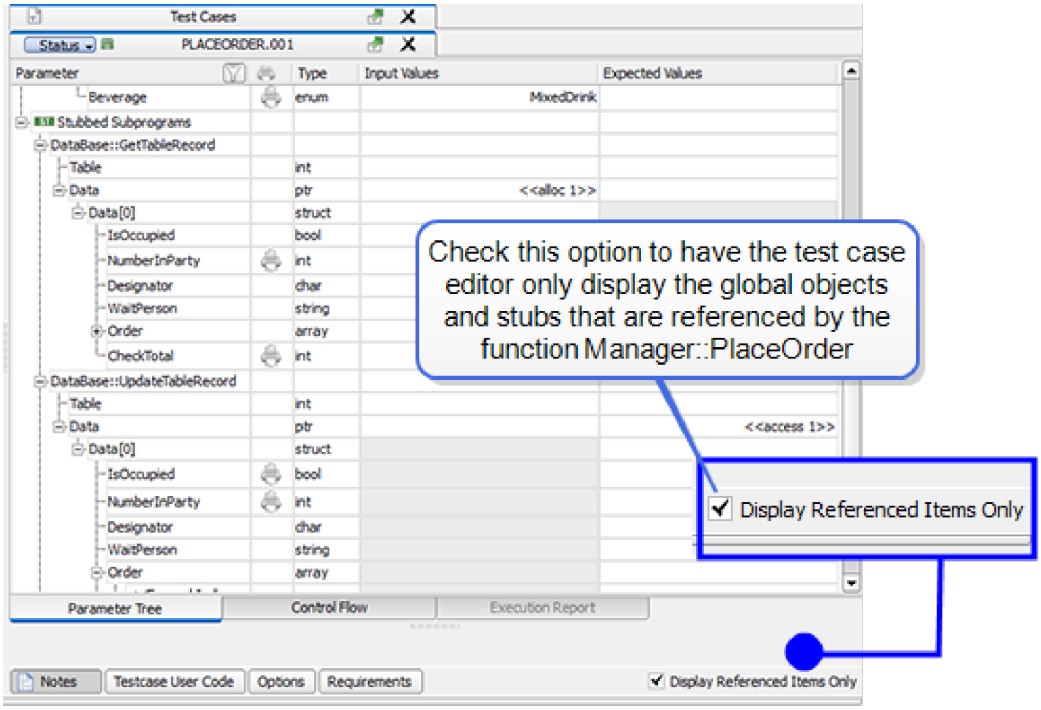
This information is saved to a file named `.vcast-qt` in the user's HOME directory.

Display Reference Items Only in the Test Case Editor

A checkbox labeled "**Display Referenced Items Only**" is located in the lower right-hand corner of the Test Case Editor. When this option is set, the Parameter Tree filters out and hides any Global variable or Stubbed Subprogram that is not called directly by the subprogram under test.

Filtering an item only affects the display in the Parameter Tree; its Input and Expected Values are used in test execution.

Note: <<SBF>> subprograms are not filtered.



Each test case can have the filter on, off, or set to the default environment-wide setting, which is located on the **Tools => Options => GUI tab => Test Case Editor Options sub tab.**

To set this option in the CCAST_.CFG file so that it affects all environments in the directory, use:

```
clicast -lc option VCAST_REFERENCED_GLOBALS True | False

When this option is on, the VectorCAST test case editor will display only
those global objects and stubs that are referenced by the function under
test. The default value is False.
```

In a test case script, this option is: TEST.VALUE:<<OPTIONS>>.REFERENCED_GLOBALS: TRUE | FALSE. The default value is unspecified, meaning inherit the value of the environment-wide value of the option.

To Toggle Gridlines in the Test Case Editor

Choose **Tools => Options**, and click the **GUI** tab and click the **Test Case Editor Options**sub- tab.

The **Show gridlines in test case** option enables you to view the Test Case Editor either with gridlines (if you prefer a spreadsheet look to the editor) or without gridlines. By default, the Test Case Editor displays with gridlines.

When you have made your selection, click **Apply** or **OK** to close the dialog.

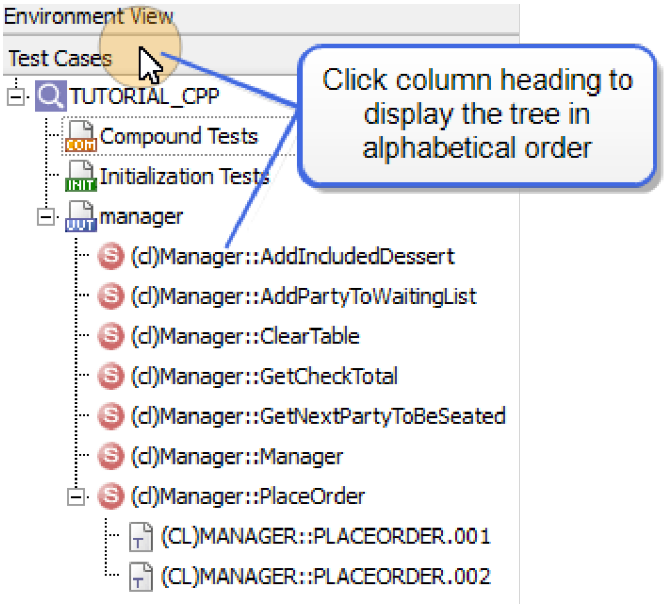
This information is saved to a file named **`.vcast-qt`** in the user's HOME directory.

To Alphabetize Test Cases

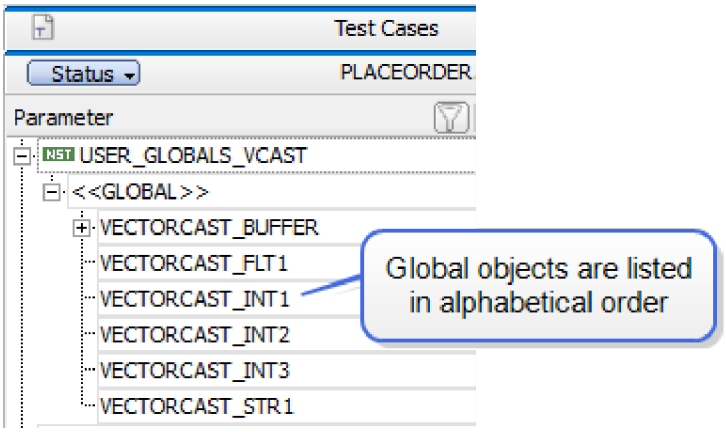
Users can set the order in which the items in the Test Case Tree and in the Test Case Editor

Parameter Tree are displayed.

In the Test Case Tree, clicking on the title bar of the Test Cases column will toggle the display from an alphabetical listing of Unit, Subprogram, and Test Case names to a listing based on the order of definition of those items. "Order of definition" means the order that subprograms are defined in your source file under test, or the order in which a test case is added to the environment.



To alphabetize in the Test Case Editor Parameter Tree, choose **Tools => Options**, and click the **GUI** tab and click the **Test Case Editor Options** sub-tab. Select the **Alphabetize parameters** option. When this option is set, the Unit names, Subprogram names, Global Object names and Parameter names are alphabetized in the Parameter Tree. When this option is cleared, the Parameter Tree items are displayed in the order they are defined in the source code.

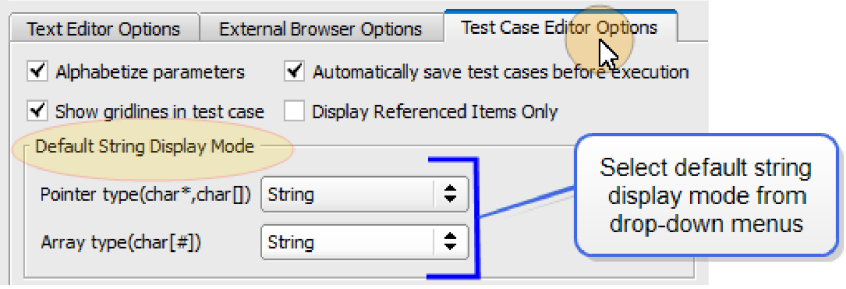


Note: If you turn this option on or off, you must close and re-open any existing Test Case Editor windows for the change to take effect.

This information is saved to a file named `.vcast-qt` in the user's HOME directory.

To Specify the Default String Display Mode

Choose**Tools => Options**, and click the **GUI** tab and click the **Test Case Editor Options** sub-tab.



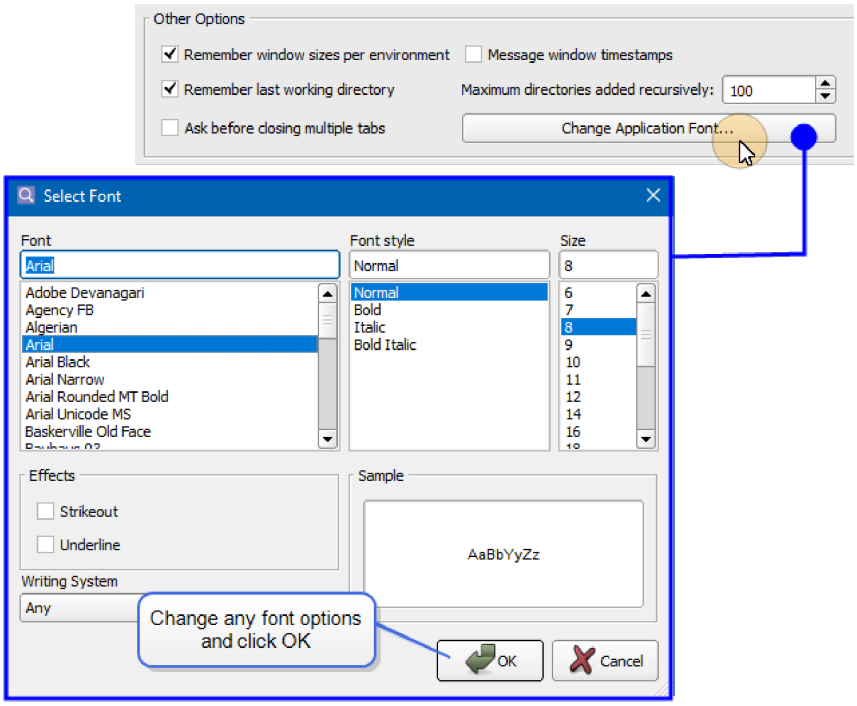
For pointer-type strings (defined as char* or char[]), you can set the default string display mode as String or Pointer in the Parameter Tree by choosing String or Pointer from the drop-down menu. For array-type strings (defined as char[#]), you can set the default string display mode as String or Array in the Parameter Tree.

You can still change the display mode for a string parameter on the fly in the Parameter Tree. The option on the **GUI** tab enables you to set how you want strings displayed by default in the Parameter Tree.

This information is saved to a file named `.vcast-qt` in the user's HOME directory.

To Change the Main Window's Font

To change the font used by the VectorCAST application, choose**Tools => Options**, and click the **GUI** tab. Click the **Change Application Font** button in the Other Options panel. The Font dialog opens. Select from any of the provided font options. Click **OK** in the font dialog.



Click either **OK** or **Apply** in the **Tools => Options** dialog, and the new font appears.

To Automatically Save Test Cases Before Execution

Choose **Tools => Options**, and click the **GUI** tab and the **Test Case Editor Options** sub-tab.

The **Automatically Save Test Cases Before Execution** option saves the current test case prior to execution. If this option is not selected, you are prompted to save a modified test case when **Execute** is chosen. By default, modified test cases are automatically saved before executing.

When you have made your selection, click **Apply** or **OK** to close the dialog.

This information is saved to a file named `.vcast-qt` in the user's `HOME`directory.

To Turn on a Reminder Before Closing Multiple Tabs

Choose **Tools => Options**, and click the **GUI** tab and the **Test Case Editor Options** sub-tab.

If you would like VectorCAST to remind you that more than one tab is being closed, turn on the **Ask before closing multiple tabs** option. This option is off by default, which means that when you click the close X of a tag group, all the tabs close without a reminder.

When you have made your selection, click **Apply** or **OK** to close the dialog.

This information is saved to a file named `.vcast-qt` in the user's HOME directory.

To Keep the Window Layout per Environment

Choose **Tools => Options**, and click the **GUI** tab.

By default, VectorCAST remembers how you have changed the layout of the various windows and

opens every environment with the dockable windows in the same location.

If you would rather have VectorCAST use one layout for every environment, uncheck the **Remember window sizes per environment** option. VectorCAST then opens each environment with the dockable windows in the same location.

When you have made your selection, click **Apply** or **OK** to close the dialog.

This information is saved to a file named `.vcast-qt` in the user's HOME directory.

Maximum Directories Added Recursively

Choose **Tools => Options**, and click the **GUI** tab.

By default, VectorCAST searches through 100 sub-directories when adding search directories recursively in the Create New C/C++ Environment wizard. When the limit is reached, you are given the option to continue looking through 100 more, stopping there, or canceling the operation. You can change the limit using the **Maximum directories added recursively** option.

Unlike other options on the GUI tab, this option is saved to the CCAST_.CFG file.

```
clicast -lc option VCAST_RECURSIVE_DIRECTORY_ADD_LIMIT <integer number>

The maximum allowed increment of directories that can be added recursively
in the Create New Environment wizard. The default value is 100 directories.
```

Customizing Reports

VectorCAST's reporting system allows reports to be customized to contain any information with any layout. Reports can be produced as either HTML or text. To generate a PDF of a report, use the HTML version.

Note that the full data API into all statistics is only available with the VectorCAST/C++ Enterprise edition.

Customizing Existing Reports

VectorCAST supports customizing existing reports. The following sections discuss how to customize sections of reports (add, remove, replace, and reorder sections), and also how to configure VectorCAST to use custom templates.

The option VCAST_RPTS_CUSTOM_CONFIG is used to customize the sections used in reports and to replace the contents of a template wihout needing a customization directory. The option can point to a file containing the configuration or the JSON configuration itself.

```
clicast -lc Option VCAST_RPTS_CUSTOM_CONFIG <config text> | <config file>

Custom report configuration. If this option points to an existing file,
then the contents of that file is used as the custom report configuration,
otherwise the contents of the option is used.
```

Adding and Removing Report Sections

To add or remove a section:

```
{
  "reports": {
    "<report_name|*>" : {
       "<HTML|TEXT>": {
        "<sections|extra_sections|remove_section":[<data>]
      }
    }
  }
}
```

Where:

`report_name` - the report to customize, or all reports (**`*`**)

`HTML|TEXT` - report format to apply the sections to

`sections` - specify a list of sections for the report

`extra_sections` - allows the addition of sections without having to list out all of the sections. Use the option '**+**' to add the new section after the existing section. Use the option '**-**' to add the new section before the existing section.

`remove_section` - remove one or more sections.

The example below shows adding the Overall Results and Metrics tables after the Configuration Data in the Execution Report:

```
{
  "reports": {
    "<ExecutionResultsReport>" : {
       "HTML": {
        "extra_sections":
          "[["CONFIG_DATA", "+OVERALL_RESULTS"],{"OVERALL_
RESULTS","+METRICS"]]
      }
    }
  }
}
```

The following example shows removing the Metrics Table from the Full Report:

```
{
  "reports": {
    "<FullReport>" : {
       "HTML": {
        "remove_sections": ["METRICS"]
      }
    }
  }
}
```

Specifying Report Sections to Replace

To replace a section:

```
{
  "sections": {
    "<section_name>" : {
      "<HTML|TEXT_FILE|TEXT>":"<path to file|text>"
    }
  }
}
```

Where:

section_name - the name of the section

HTML | TEXT_FILE | TEXT - specifies whether the following string is the actual HTML or text template or a file name containing the template.

The following example shows replacing the Metrics Table with a text file:

```
{
  "sections": {
    "<METRICS>" : {
      "TEXT_FILE": "C:\template.txt"
     }
   }
 }
```

Specifying Report Section Order

To specify the order of the sections:

```
{
  "reports": {
    "<report_name |*>" : {
      "<HTML | TEXT>" : {
        "<sections | extra_sections | remove_section": [<data>]
      }
    }
  }
}
```

Where:

report_name - the report to customise, or all reports (*****)

HTML | TEXT - report format to apply to the sections

sections - specifies a list of sections for the report

extra_sections - allows the addition of sections without having to list out all of the sections.

For example,
['EXISTING_SECTION', '<option>NEW_SECTION']
– where **option** is '**+**' or '**-**' to add the new section after or before the existing section

remove_section - remove one or more sections

The example below shows adding a "new section" after Configuration Data and "another section" before the Metrics Table to all reports:

```
{
  "reports": {
    "*" : {
      "TEXT" : {
        "extra_sections" : [["CONFIG_DATA", "+NEW_SECTION"], ["METRICS", "-
ANOTHER_SECTION"]]
      }
    }
  }
}
```

Customizing Templates

All report templates reside in the product installation directory located at:
%VECTORCAST_DIR%\python\vector\apps\ReportBuilder\templates*

Templates can be customized. It is important to note the following:

- Any template changes will apply to ALL VectorCAST projects.
- It is possible that you may not have permission to modify the installation directory. This means that each user who wants to run reports must modify the files.

Best practice is to use a Configuration Directory to override the default templates. If a template exists in the Configuration Directory, it will be used over the template located in the installation directory.

The Configuration Directory can also be the location to create new reports.

To create a new Configuration Directory:

```
%VECTORCAST_DIR%\clicast REPORTS EXTRACT <new_dir>

This command copies all the templates and adds a .orig extension.
```

Configuring VectorCAST to Use Custom Templates

To configure VectorCAST to use the Custom Templates in the Configuration Directory, use the following command:

```
clicast -lc Option VCAST_RPTS_USER_REPORTS_DIR <directory>
```

Template files are located in the **templates\<section_name>** directory. All template files have a **.orig** extension, so by default, they are not used. To use the template in the custom report directory, you must remove the **.orig** extension.

Template files use a full-featured template engine for Python called Jinja (http://jinja.pocoo.org/), which allows:

- Easy decoupling of layout from data,
- Easy modification of report structure, and the
- Ability to modify the Jinja code to change appearance.

Customizing Report Sections

VectorCAST supports adding customized headers and footers. By default, every report includes a custom header and a custom footer, but they are empty. You must define a custom header or a custom footer for them to appear.

To accomplish this, users need to configure VectorCAST to use custom templates. The following steps are used to set up the configuration:

1. First, to override the default report templates, it is recommended that you create a configuration directory to store and create new reports. For our example, we are creating a new configuration directory named **`CustomDirectory`**. This will copy all the default report template directories into our new directory and add a **`.orig`** extension to each file.

```
clicast REPORTS EXTRACT CustomDirectory
```

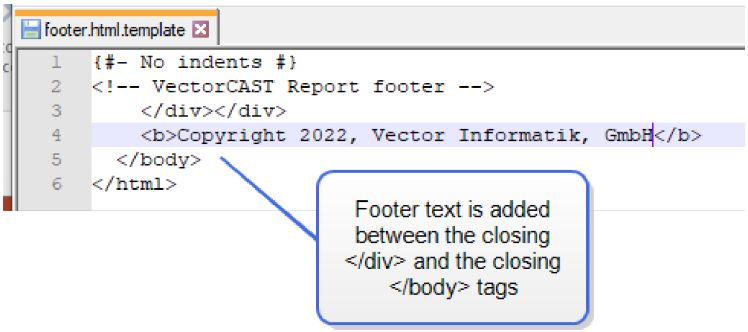
2. Next, configure VectorCAST to use the new configuration directory, **`CustomDirectory`**.

```
clicast -lc Option VCAST_RPTS_USER_REPORTS_DIR
C:\VCAST\examples\environments\enterprise_testing_demo\CustomDirectory
```

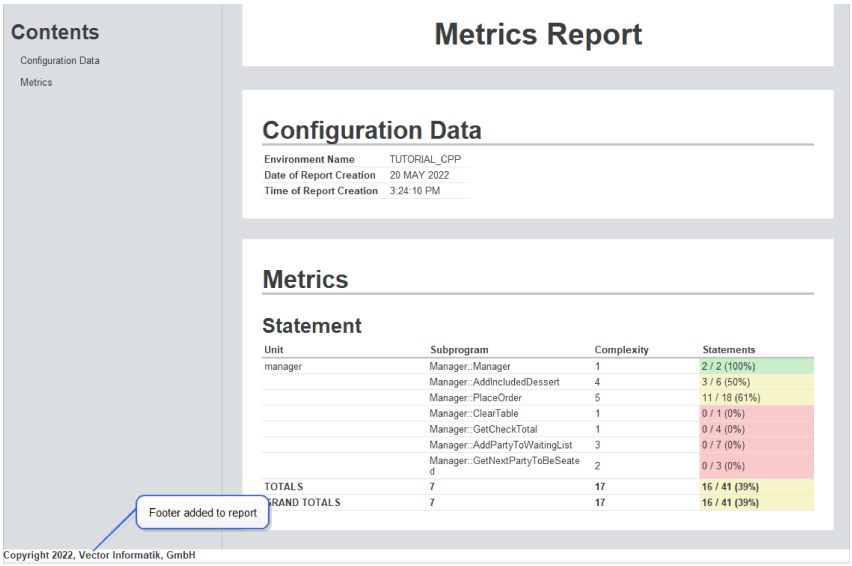
To customize reports, locate the report template in the configuration directory, remove the **`.orig`** extension and modify the file as needed.

Add a Custom Footer

To add a custom footer to the end of all reports, you will customize the **`footer.html.template`** file located in your configuration directory under **`<configuration directory>\templates`**. Open the file in your editor and add the desired footer text after the closing tag, and before the closing </body> tag, as shown below.



Save the changes and generate a report. The footer is displayed at the bottom of the report.



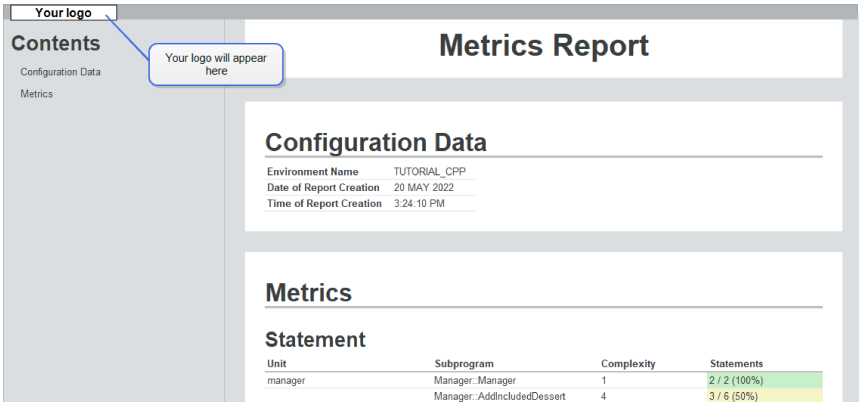
Add Your Logo to Reports

To add your custom logo to reports, you will customize the **`main.html.template`** file, located in your configuration directory under **`<configuration directory>\templates\ReportTitle`**.

Open the file in your editor and override the default VectorCAST logo with your own Base64 encoded image.


Replace with your Base64 encoded logo image
main.html.template
<!-- ReportTitle -->

{%- if report.is_self_contained %}
<img alt="Vector"
src="



Customizing Report Style

VectorCAST uses a cascading style sheet (CSS) to control the appearance of reports. The default CSS is located at: **`%VECTORCASTDIR%\python\vector\apps\ReportBuilder\css\default-style.css`**.

The styles contained in this style sheet are embedded into the **`<style>`** section of the HTML VectorCAST reports when they are generated.

Creating a Custom Style Sheet

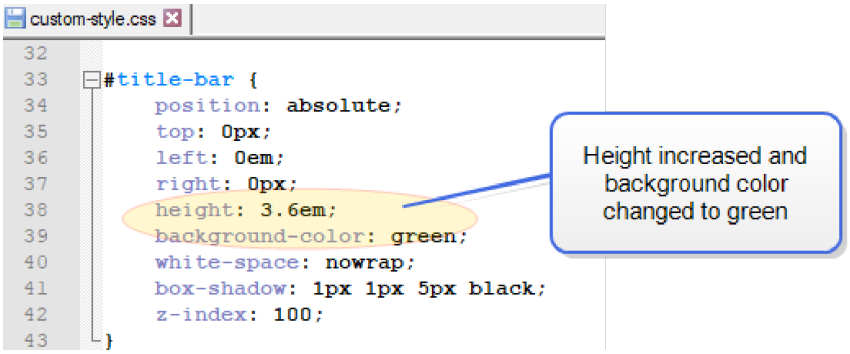
While you can modify the default stylesheet, best practice is to make a copy of the default stylesheet, store it in your configuration directory (you can rename the stylesheet), and customize that file as needed.

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, as shown in the example below:

```
<!DOCTYPE html>
<!-- VectorCAST Report header -->
<html lang="en">
  <head>
    <title>Report</title>
    <meta charset="utf-8"/>
    <style>
       default styles...
       user's custom styles...
    </style>
  </head>
...
```

Any user defined styles will follow standard cascading style sheets rules and will correctly override the defaults, meaning that only the specific attributes of a style must be specified.

In the example below, we are modifying our custom style sheet to change the report header background color to green and make it twice as wide.

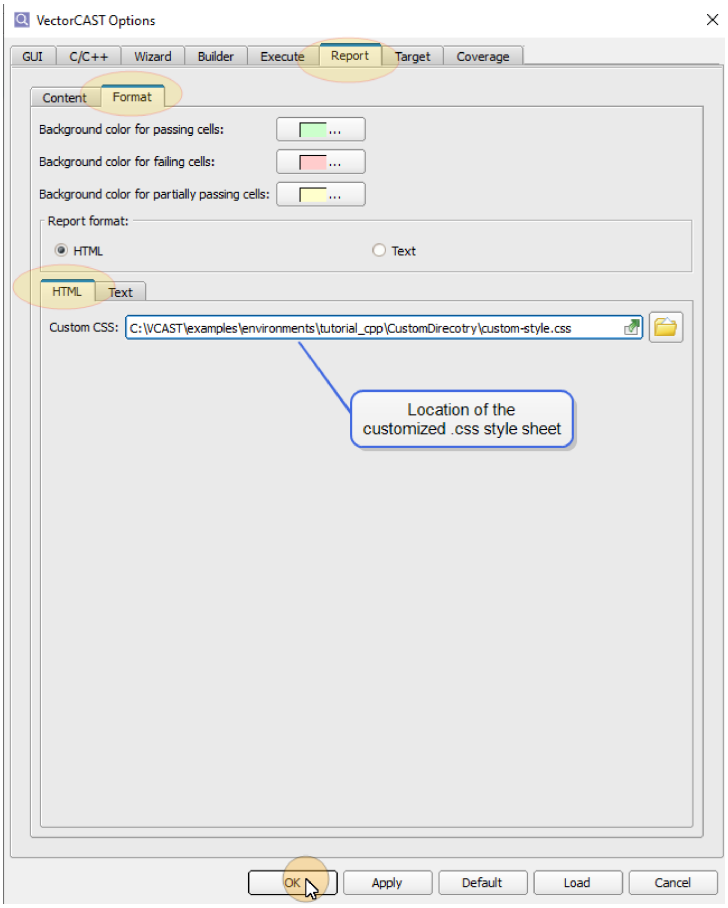


Next we add our custom style sheet to VectorCAST.

Adding a Custom Style Sheet

From the GUI:

To add a custom style sheet, Choose **Tools => Options** and click the **Report** tab => **Format** sub-tab => **HTML** sub-tab. Use the browser button to select the location of your custom style sheet. Environment variables in the path to the custom style sheet are supported, using the syntax **$(ENV_VAR)** . Click **OK** to apply the custom style sheet.

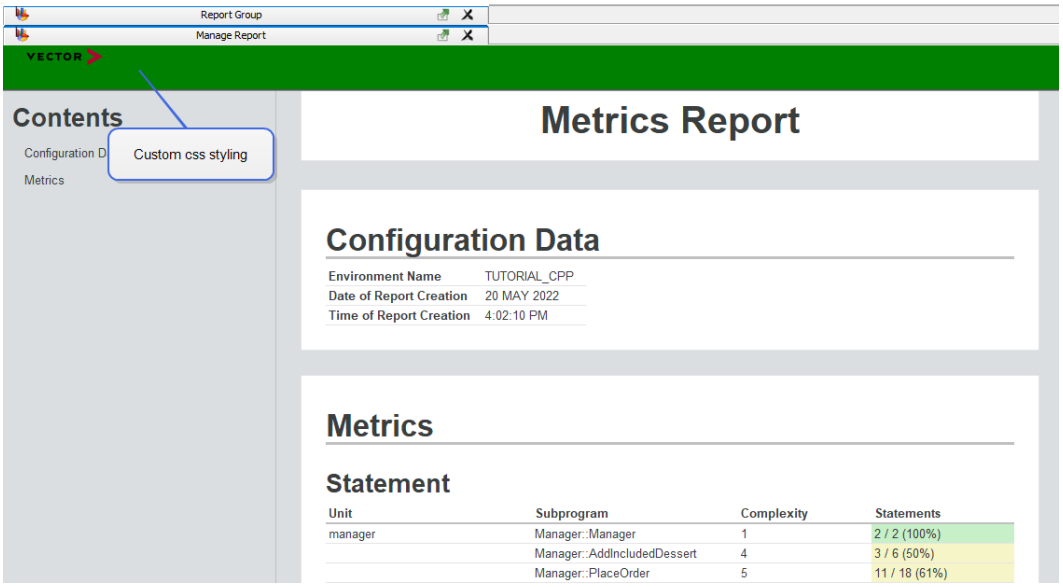


From the command line:

```
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 the 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.
```

Once the custom style sheet is added, it is applied when generating a report. In the example below, we see our custom-style.css is applied with its modifications to the title bar.



Embedding Style Sheets

VectorCAST provides the clicast-only option VCAST_RPTS_SELF_CONTAINED, which controls how reports are generated. By default, the option is set to True. This means that each HTML report is created as a single, self-contained file with an embedded style sheet and embedded images (for all built-in, non user-modified reports). This is useful when you are planning to email reports.

Embedding can cause problems with some web servers. To address this, VectorCAST provides the ability to set the option to False. When the option is set to False, the styles contained in the default style sheet (and any custom style sheet) are generated into a local style file referenced by the report. Style sheets and images are stored in the same directory as the report, which is useful when serving reports from a web server.

```
clicast -lc option VCAST_RPTS_SELF_CONTAINED True | False

When True, HTML reports are created as a single, self-contained file with
```

```
an embedded stylesheet and embedded images (for all built-in, non user-
modified reports). Set the option to True (default value) when planning to
email reports. When False, HTML reports are created as multiple files,
storing stylesheets and images in the same directory as the report. Set the
option to False when serving reports from a webserver.
```

Creating New Reports

VectorCAST uses a cascading style sheet (CSS) to control the appearance of reports. The default CSS is located at:
%VECTORCASTDIR%\python\vector\apps\ReportBuilder\css\default-style.css

Report Components

There are three components to writing a new report: a main report file, a section file, and a template layout file. The table below lists the files used to create an example of a new report, named My Report, along with the location of each file and its purpose. Each component is discussed in more detail in the following sections.

File Name	Location	Purpose
my_report.py	custom_rep\reports	The 'main' report file. This defines the report name and overall sections. In this report there are some built in sections and a new section called EXAMPLE_SECTION. The file and class name should match up, so the file is my_ report.py with a class called MyReport. The report name is 'my_report'.
example_ sections.py	custom_rep\sections	Uses the data API to extract data from VectorCAST and put it into a data structure so it can be formatted into the report file. The file/class/section names are all related: example_sections.py contains an ExampleSections class to implement the EXAMPLE_SECTIONS section. The prepare_ data function is called and this adds data to the self.section_context object. In this case we use all the test cases and iterate over them, getting the result.
main.html.template	custom_ rep\templates\ExampleSection	Contains the presentation format for the section. Note that this file has a fixed name, but is in a directory corresponding with the section name. This is a Jinja template which allows you to control the HTML formatting of the Python data. This example just adds the report name and then a simple table of the test case results, coloured using built in styles.

Report File

The report file, **my_report.py**, is the main entry point for the report. This file sets up any initial data, defines the report name, and describes the sections of the report and the order in which they appear.

```
import os
from vector.apps.ReportBuilder.custom_report import CustomReport

class MyReport (CustomReport):
    default_sections = ['CUSTOM_HEADER', 'REPORT_TITLE', 'TABLE_OF_
CONTENTS', 'EXAMPLE_SECTION', 'METRICS', 'CUSTOM_FOOTER']

    def initialize(self, **kwargs);
       pass
```

Section File

The section file, **example_sections.py**, contains the data that goes into the report. Report data is typically extracted using the Manage Data API (although it is possible to retrieve the data from elsewhere).

```
from __future__ import absolute_import
from vector.apps.ReportBuilder.report_section import ReportSection
from vector.apps.DataAPI.api import Api

class ExampleSection(ReportSection):
# Define the title (this shows in the config data and the Table of
Contents)
    title = 'Results Section'
# Sections can be conditional on type of environment - in this case, this
will be used for both
supported_environments = ('COVER', 'UNIT')

# prepare_data is called to setup any data required for generating the
report
       def prepare_data(self):
           self.section_context['rep']="Execution Results Summary"
           self.section_context["results"]=[]
           api=Api('.')

           a=api.TestCase.all()
           for test in a:
                result={}
                result["name"]=test.name
                result["verdict"]="PASS" if test.passed else "FAIL"
                self.section_context["results"].append(result)
```

Layout Template File

The layout template file, **main.html.template**, is a Jinja template that describes the layout of the data. Note that:

- Elements in double curly braces **{{}}** are Python data objects from the section file
- Can also contain regular HTML
  - But do not include styles. Presentation should be placed in the CSS file.

```
{# This is a comment! My Report template #}
<h1>{{rep}}</h1>

<table style="width:100%">
    {%- for result in results %}
       {%- if result.verdict == "PASS" %}
            <tr class="success">
       {%- else %}
            <tr class="danger">
       {%- endif %}
       <td>{{result.name}}</td>
       <td>{{result.verdict}}</td>
       </tr>
    {%- endfor %}
</table>
```

Running New Reports

From the command line:

```
clicast -e <environment> report user <report_name>
```

For example:

```
clicast -e my_env report user my_report
```

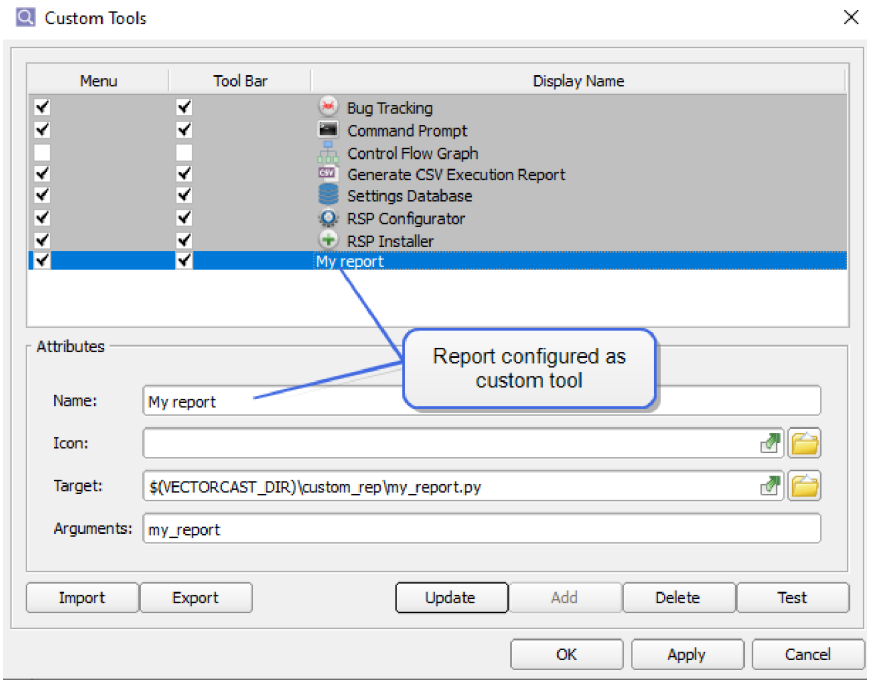
From the GUI:

To generate new reports from the GUI, the report can be configured as a custom tool which will run the clicast command.

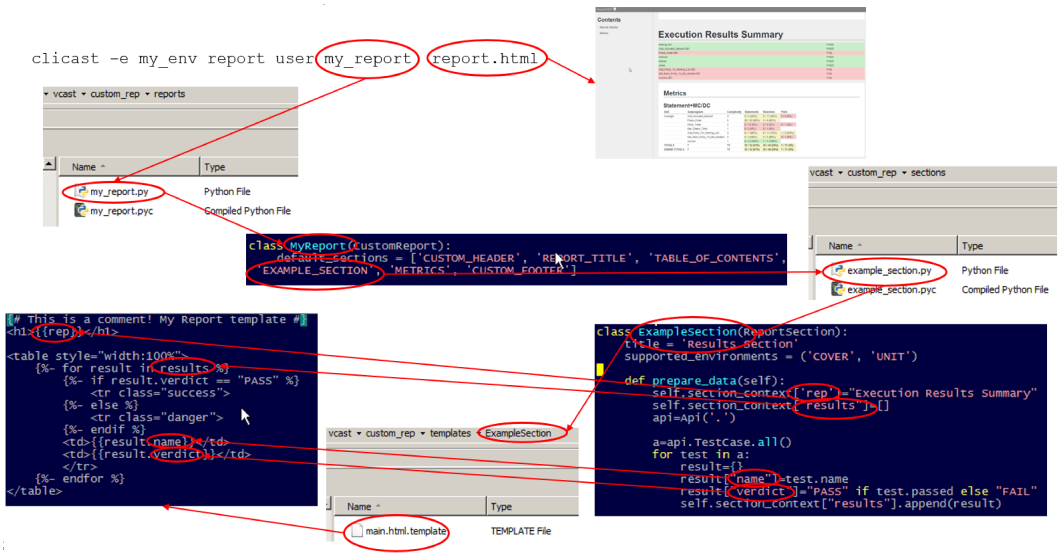
Create a helper script which will do the following:

- Change up a directory out of the environment directory
- Remove any previous stale report file
- Run the report
- Open the report in the GUI

The script can accept any report name, so it can be used for any customised report.



The following figure illustrates how the components used to create a custom report work together to produce the final custom HTML report.



Editing, Searching, and Printing

To Create a New Text File

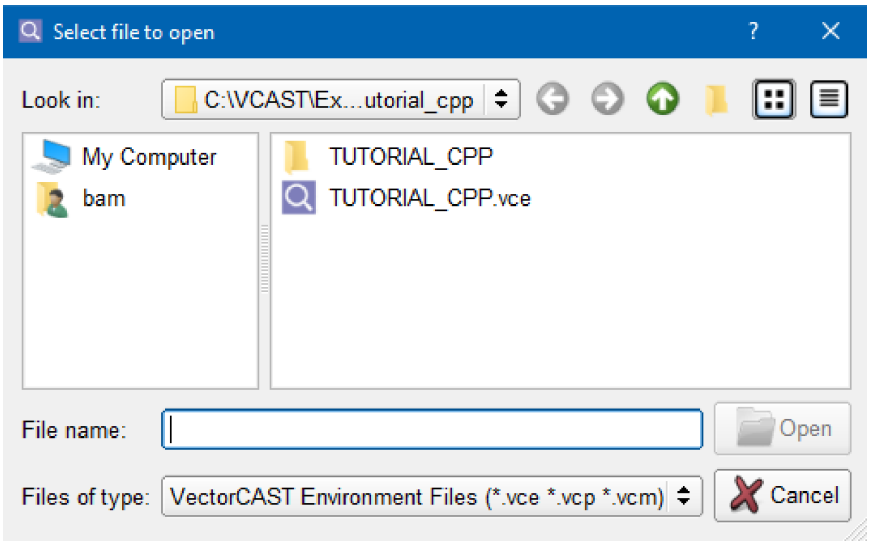
Use **File => New => New Document** to create a new text file. The keyboard shortcut is Ctrl+T. When selected, a new, untitled text window opens in a Text Editor in the MDI window. The file is set as writeable by default.

If you have specified an external editor on the **Tools => Options** dialog, GUI tab, then the text file opens in that editor.

To Open a Script or Report File

File => Open... enables you to open an existing environment. A standard open file dialog box is used. By default, the file extension "***.vce *.vcp *.vcm**" is selected as the file type.

The command may also be launched by clicking on the Open button  in the toolbar.



By changing the "Files of Type" selection, you can also use **File => Open...** to open other kinds of files such as source files. Files are opened in a Text Editor or the external editor, if specified.

To open a(n)	Choose Files of Type
Unit test environment	Environment Files (***.vce** *.vcp *.vcm)
Coverage environment	Environment Files (*.vce ***.vcp** *.vcm)
Manage project	Environment Files (*.vce *.vcp ***.vcm**)
Test case script	Script Files (*.scr, *.env, *.**tst**)
Environment script	Script Files (*.scr, *.**env**, *.tst)
Windshell script	Script Files (***.scr**, *.env, *.tst)

To open a(n)	Choose Files of Type
C or C++ source code file	C/C++ Source Files (*.cpp, *.cc, *.cxx, *.c, *.c++)
C or C++ header file	C/C++ Header Files (*.h, *.hxx, *.h++)
Ada Source code files	Ada Source Files (*.adb, *.ada, *.ads)
Configuration file	VectorCAST Config Files (*.CFG)
HTML or Text file	Report Files (*.html, *.txt)
CSV file	CSV Map files (*.csv, *.tab)
a file of any type	All Files (*)

To Save a Script or Text File

File => Save enables you to save the contents of the current active text document in the Text Editor.

You can save a changed test case if it is in the MDI window. If a source code file, test script, or configuration file is currently open in the MDI window, then it is saved. If a results file report, data file, or anything listed in the **Environment => View** or **Test => View** menu is saved, then the Save As dialog opens, allowing you to give the file a name.

The command may also be invoked by clicking on the **Save** button on the Toolbar  .

To Save the Open Window As

File => Save As... enables you to save the text or HTML data in the current open window to a filename of your choice. Choose **File => Save As...** from the Menu Bar. You are prompted for the name of the file where the extracted data should be stored.

The command may also be invoked by clicking on the **Save All** button  in the toolbar.

To Save All Open Windows

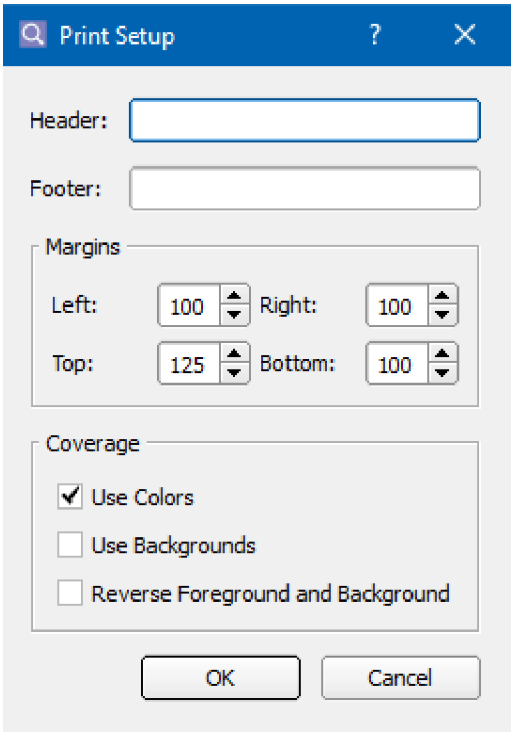
File => Save All applies the Save command to all unsaved, visible windows. For windows in the Test Case Editor, it saves unsaved parameter data. For reports and scripts in the Text Editor, it saves the information to a file. If more than one unsaved text file is open, you are prompted to save each one, starting with the top-most file.

The command may also be invoked by clicking on the **Save All** button  in the toolbar. When you use the **Save All** button, each tabbed window is saved. If you have several different windows open and tiled or cascaded, then each tab in each visible window is saved. In cases where a filename is required, you are prompted once for each tabbed window that needs a filename.

Print Setup

File => Print Setup... enables you to customize your printing options, such as headers, footers and

margins.



Header and Footer: Add a header and/or footer text string to your printed report.

Margins: Customize the size of the margins in the report output. The margins options are: Left, Right, Top and Bottom. You can enter the desired margin value for each option. You can also use the spin arrows to select the margin values.

Use Colors: Print coverage reports with the colors chosen in **Tools => Options, Coverage** tab, **Coverage Viewer** sub-tab. Colors can be chosen for covered lines, partially covered lines, uncovered lines, and uncoverable lines.

Use Backgrounds: Print the background color specified in **Tools => Options dialog, Coverage tab, Coverage Viewer sub-tab**. The default is to print using the foreground color only. Printing with background colors enabled can use a lot of ink or toner.

Reverse Foreground and Background: Reverse foreground and background colors when printing coverage reports. It is useful when the selected coverage foreground colors are light and the background colors are dark, saving on ink or toner.

To Print an Open Window

This command enables you to print out the contents of any text file in a Text Editor or Coverage Viewer. The **File => Print...** command applies to reports, results, scripts, and coverage information.

Select **File => Print...** or click on the **Print** button on the toolbar  .

To Preview Before Printing

The **File => Print Preview…** command enables you to preview any print job before sending it to the printer. Selecting **Print Preview...** will open a separate Print Preview window.

This command can also be accessed by clicking the arrow on the right of the Print button  on the Toolbar and then selecting **Print Preview…** from the drop-down menu.

To Undo a Recent Change

Edit => Undo applies to entered text and enables you to undo the last text stored in the text buffer.

This command can also be accessed by pressing **Ctrl+Z**.

To Redo

Edit => Redo reverses the action of the **Undo** button. If you select **Undo** by mistake, **Redo** will correct the error.

This command can also be accessed by pressing **Ctrl+Y**.

To Copy, Cut, and Paste

You can copy sections of text by copying and pasting. This means you can select text from one location and insert it into another place. To copy and paste do the following:

1. Select the text you want to copy.
2. Perform one of the following actions to save a copy of the text to the clipboard:
   - Choose **Edit => Copy** from the Menu Bar,
   - Press **Ctrl+C**,
   - Or right-click and choose **Copy** from the context menu.
3. Position the cursor where you wish to insert the text.
4. Perform one of the following actions to insert the text :
   - Choose **Edit => Paste** from the Menu Bar,
   - Press **Ctrl+V**,
   - Or right-click and choose **Paste** from the context menu.
5. To undo the paste, choose **Edit => Undo** from the Menu Bar.

You can also move text using cutting and pasting. This means you can remove text from one place and move it to another. To cut and paste do the following:

1. Select the text you want to move.
2. Choose **Edit => Cut** from the Menu Bar.
3. Position the cursor where you wish to insert the text.
4. Choose **Edit => Paste** from the Menu Bar to insert the text.

To Search for Text Using the Find Banner

The Find Banner is used to search for text within the following areas:

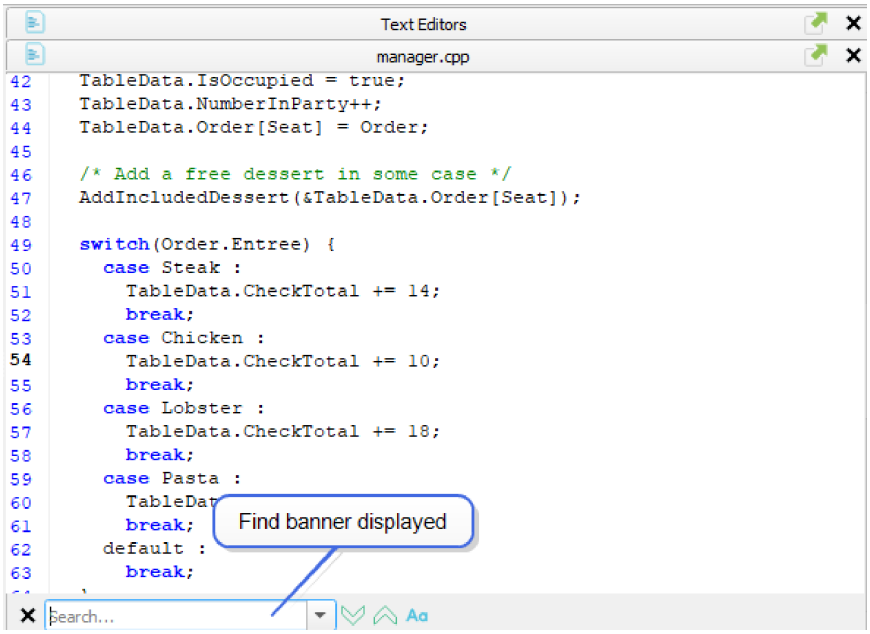
- Text Editor
- User Code Editor
- Coverage Viewer
- Parameter Tree
- Test Case Tree
- Report Viewer

The Find Banner appears at the bottom of the window, and is off by default.

Open The Find Banner

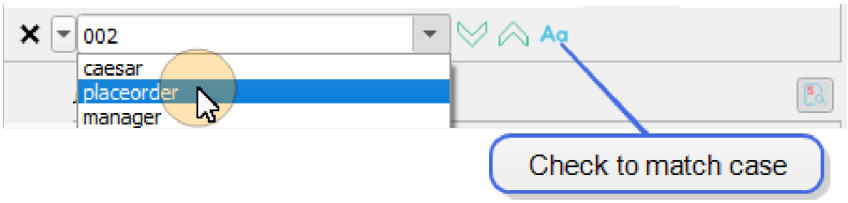
To open the Find Banner and search for text, follow these steps:

1. Open a file in an Editor, a report in a Viewer, or open a Parameter or Test Case Tree.
2. Click the mouse within the window where you wish to perform the search.
3. Choose **Edit => Find...** from the Menu Bar (or press **Ctrl+F**) and a Find Banner appears at the bottom of the Text Editor.

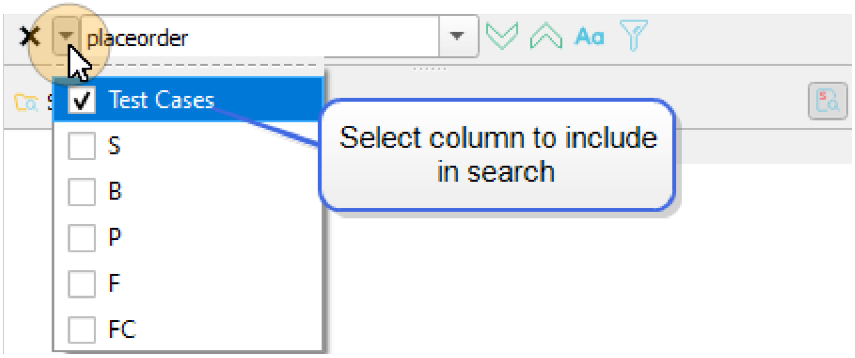


Execute a Search

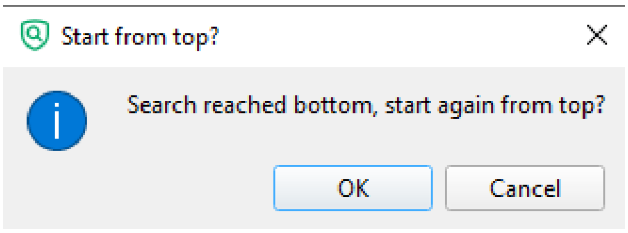
1. Type the text you are looking for in the **Find** text edit box. To choose text you have typed before, click the drop-down arrow, and select text from the list.



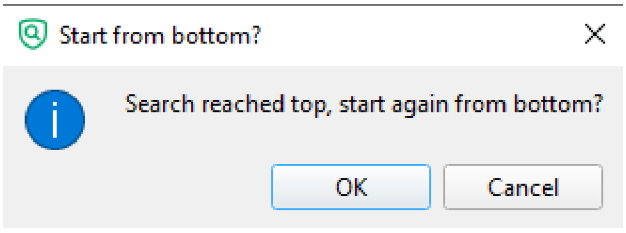
2. To specify that the case is matched exactly as you typed it, click the **Match Case** button.
3. When executing a search in the Test Case Tree, you must first select the columns to be included in the search. Click the drop-down arrow to the left of the search entry field, and from the menu, select the column names.



4. Press **Enter** to execute the search. All instances of the search string are highlighted, making it easy to locate search results.
5. To search for the next instance of the text, click the **Down** arrow.
   > In the Coverage Viewer, Report Viewer and User Code Editor, if the search text between the current line and the end of the file is not found, you are prompted to search from the top. Click **OK** to continue searching, or **Cancel** to quit the search.

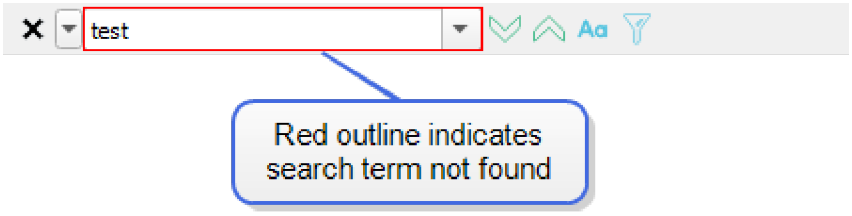


   > In the Text Editor, Parameter Tree and Test Case Tree, the search will continue to cycle through from the beginning.
6. To search for the previous instance of the text, click the **Up** arrow.
   > In the Coverage Viewer, Report Viewer and User Code Editor, if the search text between the current line and the top of the file is not found, you are prompted to search from the bottom. Click **OK** to continue searching, or **Cancel** to quit the search.

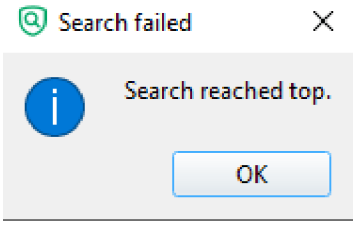


> In the Text Editor, Parameter Tree and Test Case Tree, the search will continue to cycle through from the bottom.

7. In the Parameter Tree, Text Editor, and Test Case Tree, if the text is not found, the text entry field displays a red outline.



8. In the Coverage Viewer, if the text is not found you are notified that the search failed.

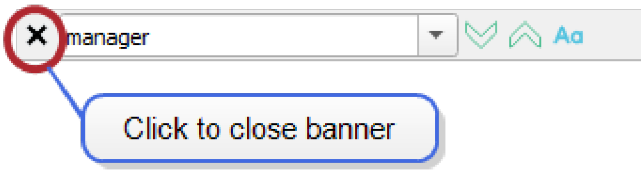


Repeat a Search

To search again without using the Find dialog, choose **Edit => Find Again** from the Menu Bar, or press the **F3** key.

Close the Find Banner

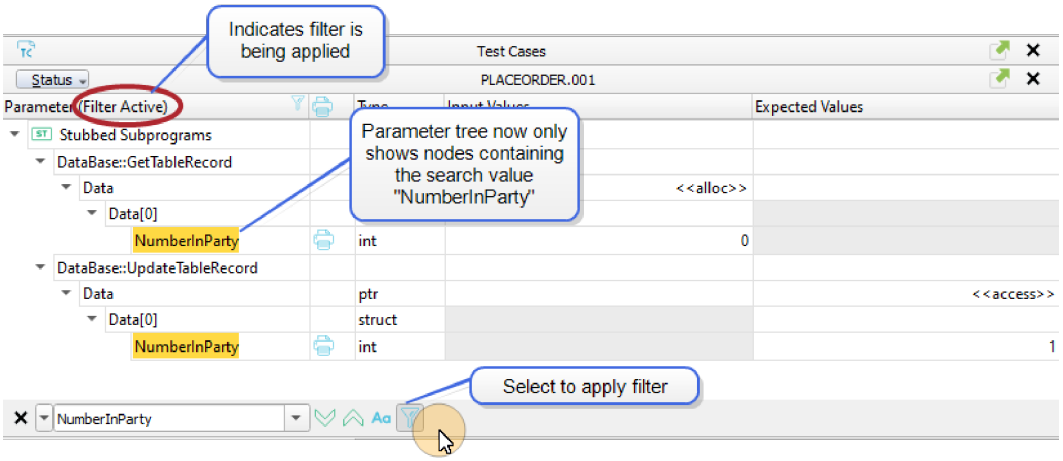
To dismiss the Find Banner, click the **Close** button located on the banner.



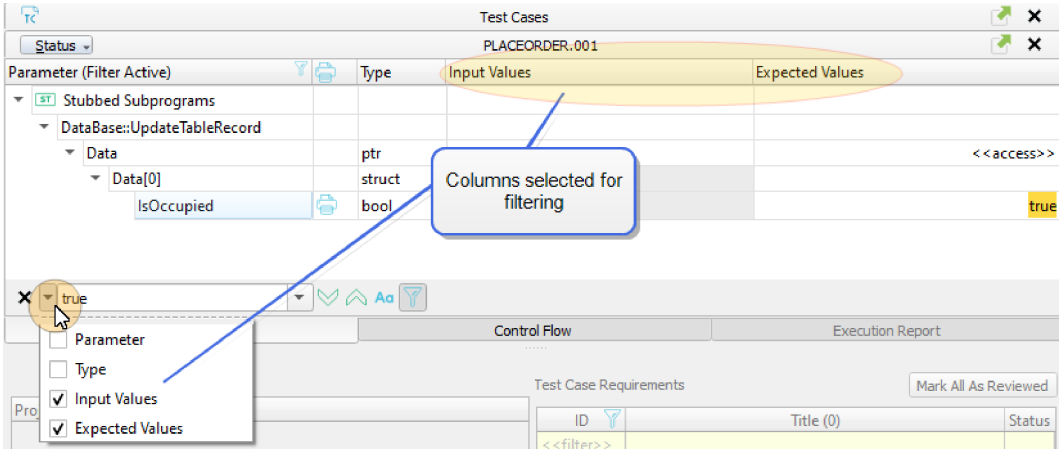
To Apply a Search Filter to the Parameter Tree

A search filter can be applied to the Parameter Tree. Filters are useful for pruning the tree and showing only those items of current interest. A filter is applied to all parameters in the tree, regardless of whether they are in an expanded part of the tree or not.

To apply a filter to the Parameter Tree, select the **Filter** button on the Find Banner. Enter the search value and press the **Enter** key. The Parameter Tree will display the unit node(s) associated with the filtered results. Note that **(Filter Active)** is displayed next to the Parameter column heading, making it easy to identify when a filter is being applied.



A filter can be applied to any column or combination of columns in the Parameter Tree. To select column filters, click the drop-down arrow to the left of the search entry field, and from the menu, select the column names.

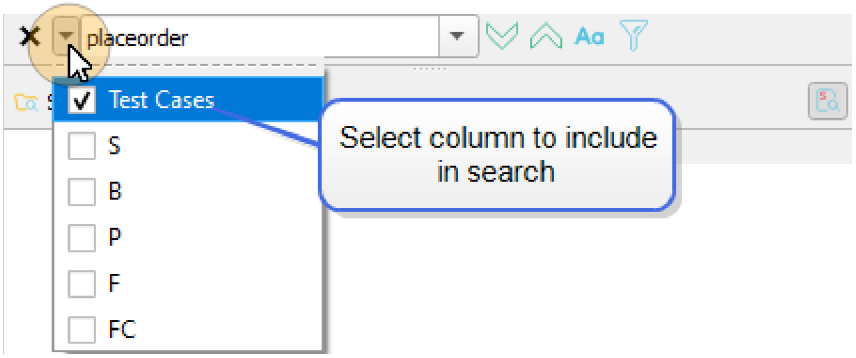


To Apply a Search Filter to the Test Case Tree

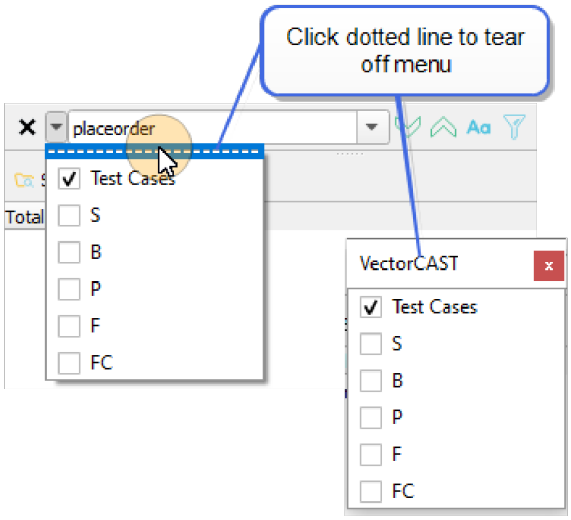
A search filter can be applied to the Test Case Tree. Filters are useful for pruning the tree and showing only those items of current interest. A filter is applied to all nodes in the tree, regardless of

whether they are in an expanded part of the tree or not.

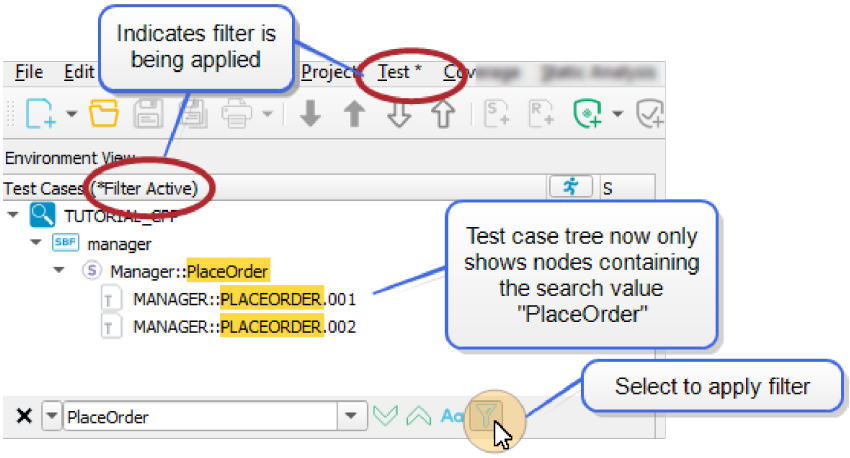
To apply a filter to the Test Case Tree, first select the columns to be included in the search. Click the **Find** button drop-down menu, and from the menu, select the column names.



The Find drop down menu is a tear off menu. Click once on the dotted line, and a floating menu stays on the screen.



Next, select the **Filter** box on the Find Banner. Notice that after selecting **Filter,** two filter indicators are displayed. The menu bar Test item is now displayed as **Test ***, and the Test Cases header in the Test Case Tree is now displayed as **Test Cases (*Filter Active)**. These indicators make it easy to detect if a filter has been applied to the tree.



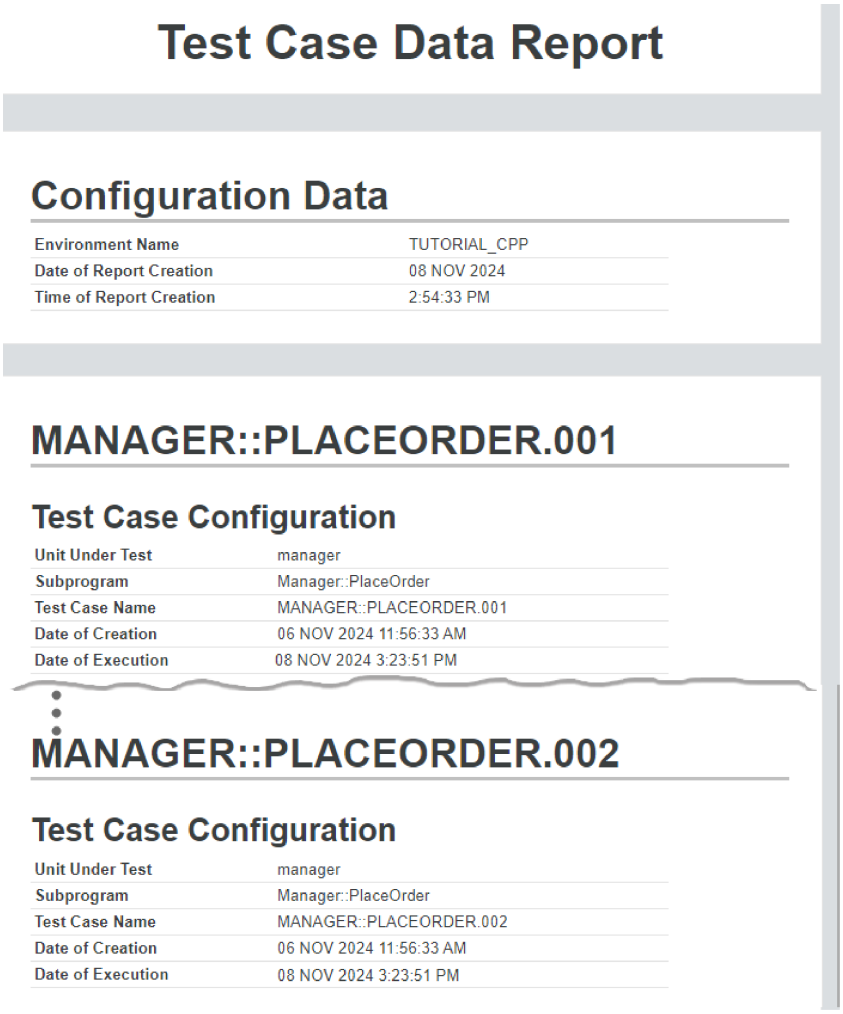
Enter the search value and press the **Enter** key. The Test Case Tree will now filter to only show nodes containing the search value.

To change the search term and reapply the filter, enter the new term in the search text box, click **Filter** to uncheck, and click **Filter** again to check it. The filter for the new term is applied and the results will be displayed.

To Apply a Search Filter to Reports

Test Case Tree filters can be applied to Test Case Data Reports, Execution Results Reports and Full Reports.

In this example, the TUTORIAL_CPP environment is executed and a Test Case Data Report is generated (**Test => View => Test Case Data**). A portion of the Test Case Data Report is shown below. The report contains information for all test cases.



Test Case Data Report

Configuration Data

Environment Name	TUTORIAL_CPP
Date of Report Creation	08 NOV 2024
Time of Report Creation	2:54:33 PM

MANAGER::PLACEORDER.001

Test Case Configuration

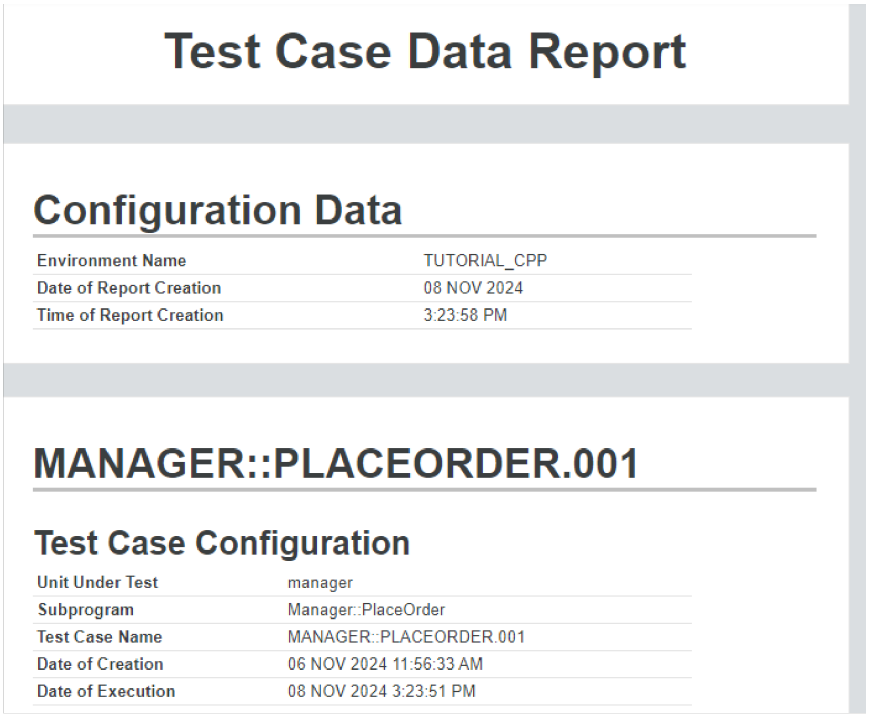
Unit Under Test	manager
Subprogram	Manager::PlaceOrder
Test Case Name	MANAGER::PLACEORDER.001
Date of Creation	06 NOV 2024 11:56:33 AM
Date of Execution	08 NOV 2024 3:23:51 PM

MANAGER::PLACEORDER.002

Test Case Configuration

Unit Under Test	manager
Subprogram	Manager::PlaceOrder
Test Case Name	MANAGER::PLACEORDER.002
Date of Creation	06 NOV 2024 11:56:33 AM
Date of Execution	08 NOV 2024 3:23:51 PM

A filter is applied for PLACEORDER.001 in the Test Case Tree, and the report is re-executed. (**Test => View => Test Case Data**). The report now shows information for only the test case that is displayed for the filter.



Test Case Data Report

Configuration Data

Environment Name	TUTORIAL_CPP
Date of Report Creation	08 NOV 2024
Time of Report Creation	3:23:58 PM

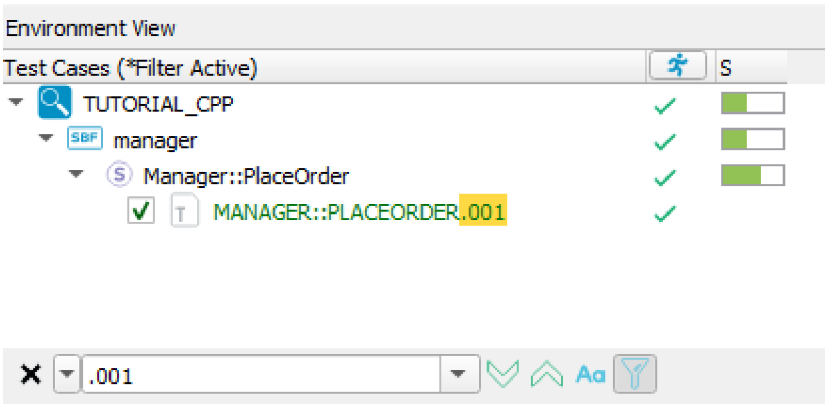
MANAGER::PLACEORDER.001

Test Case Configuration

Unit Under Test	manager
Subprogram	Manager::PlaceOrder
Test Case Name	MANAGER::PLACEORDER.001
Date of Creation	06 NOV 2024 11:56:33 AM
Date of Execution	08 NOV 2024 3:23:51 PM

To Apply a Search Filter to Test Scripts

The Test Case Tree filters can also be applied in exporting test scripts. The tree below has been filtered to search the Test Cases for **.001**.



A test script is then exported from the TUTORIAL_CPP environment level of the tree. TUTORIAL_CPP is selected, then, **Test => Scripting => Export Script** is selected. The following test script is generated. Note that the only test case in the test script is for the test case returned by the filter. In an unfiltered tree, the test script would contain all test cases for the environment.

```
-- VectorCAST 25 (11/04/24)
-- Test Case Script
--
```

```
-- Environment : TUTORIAL_CPP
-- Unit(s) Under Test: manager
--
-- Script Features
TEST.SCRIPT_FEATURE:C_DIRECT_ARRAY_INDEXING
TEST.SCRIPT_FEATURE:CPP_CLASS_OBJECT_REVISION
TEST.SCRIPT_FEATURE:MULTIPLE_UUT_SUPPORT
TEST.SCRIPT_FEATURE:REMOVED_CL_PREFIX
TEST.SCRIPT_FEATURE:MIXED_CASE_NAMES
TEST.SCRIPT_FEATURE:STANDARD_SPACING_R2
TEST.SCRIPT_FEATURE:OVERLOADED_CONST_SUPPORT
TEST.SCRIPT_FEATURE:UNDERSCORE_NULLPTR
TEST.SCRIPT_FEATURE:FULL_PARAMETER_TYPES
TEST.SCRIPT_FEATURE:STRUCT_DTOR_ADDS_POINTER
TEST.SCRIPT_FEATURE:STRUCT_FIELD_CTOR_ADDS_POINTER
TEST.SCRIPT_FEATURE:STRUCT_BASE_CTOR_ADDS_POINTER
TEST.SCRIPT_FEATURE:STATIC_HEADER_FUNCS_IN_UUTS
TEST.SCRIPT_FEATURE:VCAST_MAIN_NOT_RENAMED
--

-- Unit: manager

-- Subprogram: Manager::PlaceOrder

-- Test Case: MANAGER::PLACEORDER.001
TEST.UNIT:manager
TEST.SUBPROGRAM:Manager::PlaceOrder
TEST.NEW
TEST.NAME:MANAGER::PLACEORDER.001
TEST.NOTES:

This test is the the same test case that should be created
by following all of the steps in the first part of the
"C Tutorials -> Basic Tutorial" from the VectorCAST
Getting Started manual.

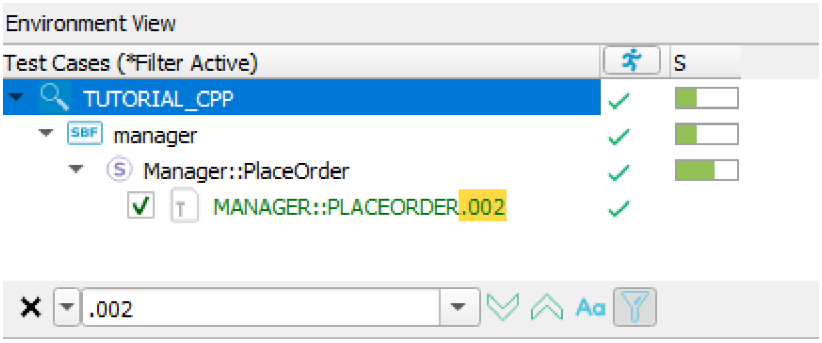
It shows the basic concepts associated with setting input and
expected values for both the Unit Under Test and Stub Functions.

TEST.END_NOTES:
TEST.VALUE:manager.<<GLOBAL>>.(cl).Manager.Manager.<<constructor>>.Manager
().<<call>>:0
TEST.VALUE:manager.Manager::PlaceOrder.Table:2
TEST.VALUE:manager.Manager::PlaceOrder.Seat:0
TEST.VALUE:manager.Manager::PlaceOrder.Order.Soup:Onion
TEST.VALUE:manager.Manager::PlaceOrder.Order.Salad:Caesar
TEST.VALUE:manager.Manager::PlaceOrder.Order.Entree:Steak
TEST.VALUE:uut_prototype_stubs.DataBase::GetTableRecord.Data
[0].NumberInParty:0
TEST.VALUE:uut_prototype_stubs.DataBase::GetTableRecord.Data
[0].CheckTotal:0
TEST.EXPECTED:uut_prototype_stubs.DataBase::UpdateTableRecord.Data
[0].IsOccupied:true
TEST.EXPECTED:uut_prototype_stubs.DataBase::UpdateTableRecord.Data
[0].NumberInParty:1
TEST.EXPECTED:uut_prototype_stubs.DataBase::UpdateTableRecord.Data
```

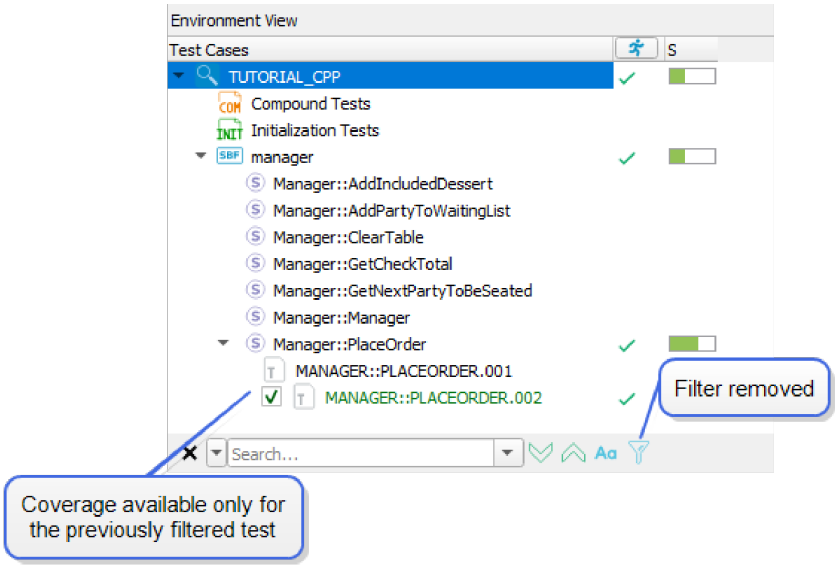
```
[0].CheckTotal:12..16
TEST.END
```

To Apply a Search Filter to Enable Coverage

A Test Case Tree filter can also be used to selectively enable coverage. In this example, the environment is initially built and executed without coverage. The tree below has been filtered to search the Test Cases for **.002** and Statement coverage is initialized (**Coverage => Initialize => Statement**) at the environment level.



The environment level is executed and the filter is then removed to display the entire Test Case Tree. Notice that coverage is available only for the test that was selected by filtering. This is indicated by a Check Box next to the previously filtered test case name.

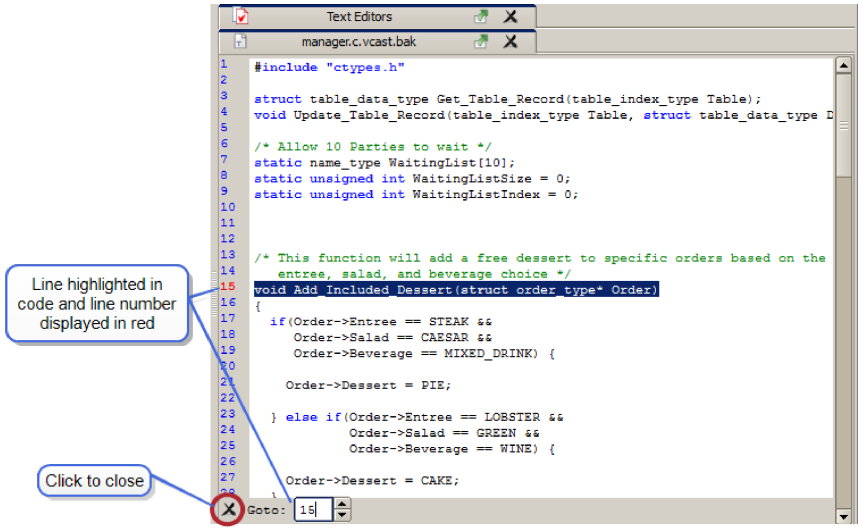


To Goto a Line

The **Edit => Goto Line** command enables you to go to a specific line number in User Code and source code in Text Editors. The file line numbers start at 1. This command is available in any text file: script files, source files, text reports, and results files.

The shortcut for the **Edit => Goto Line** command is **Ctrl+G**.

At the bottom of the Text Editor window a Goto dialog opens. Insert the line number and hit Enter. The specific line is highlighted in the code.

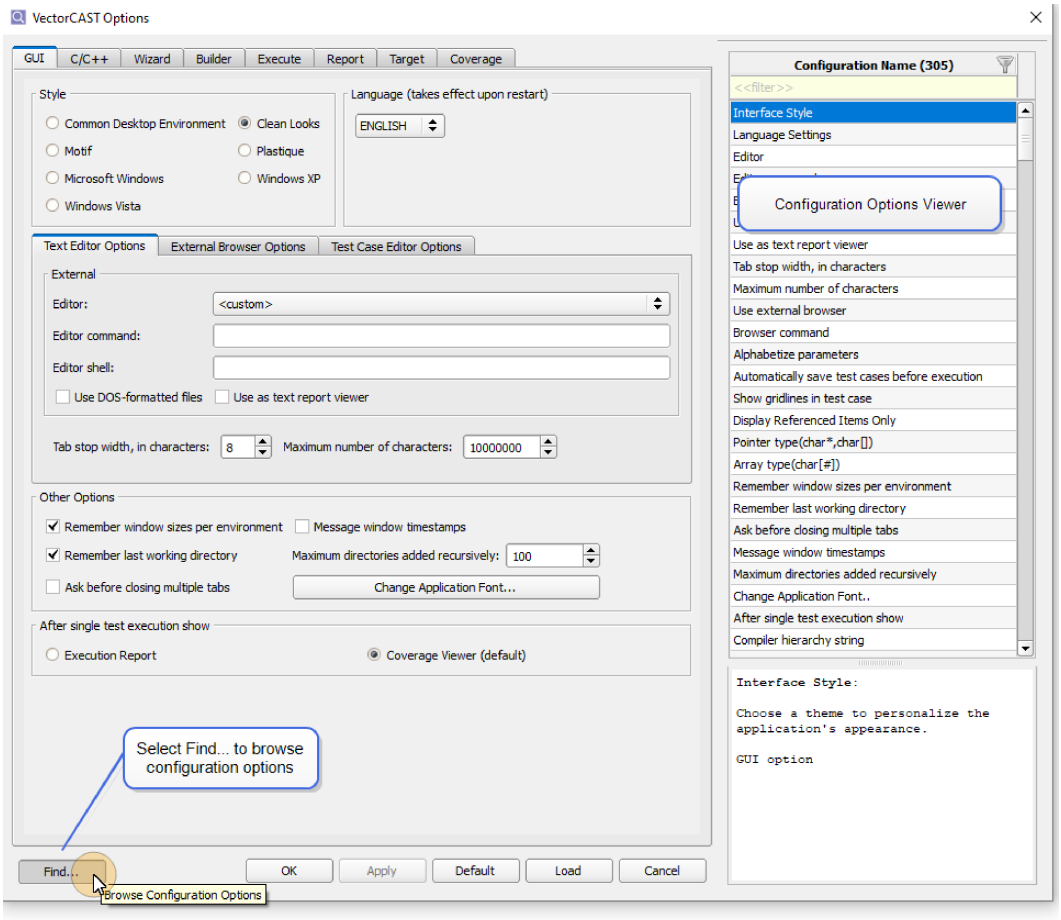


Note: The file line number is the source file line number, not the executable source code line number.

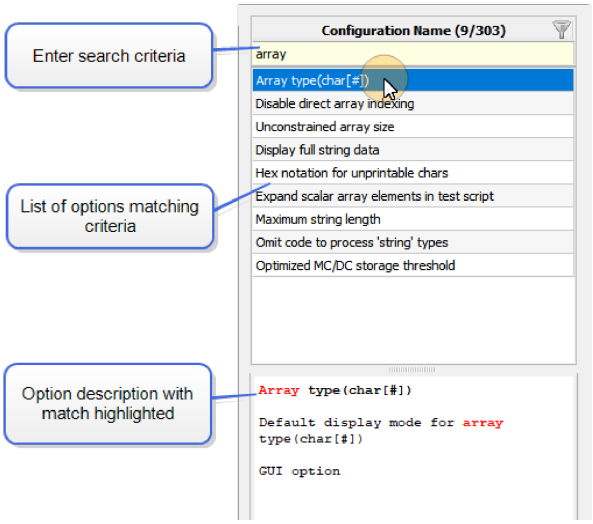
Browsing Configuration Options

Using the Configuration Options Viewer

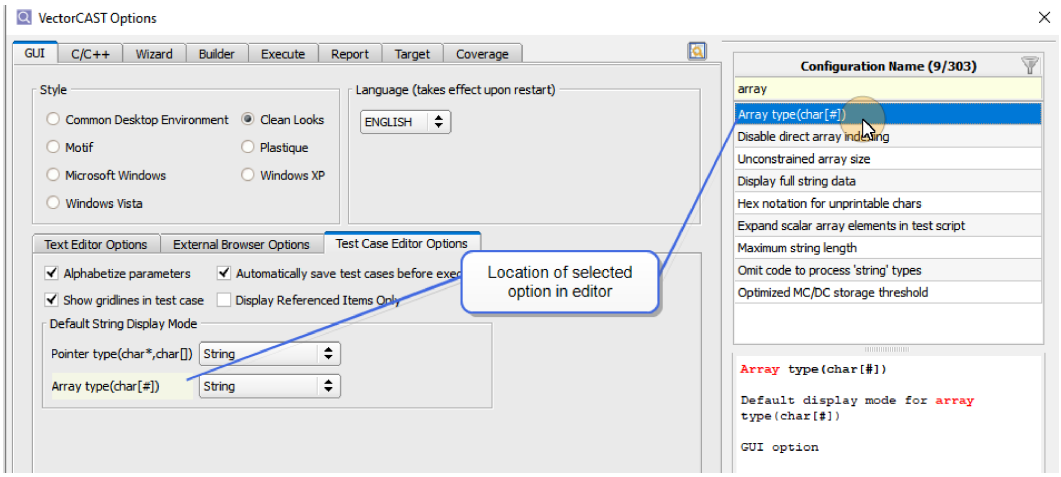
The Configuration Options Viewer provides users with the ability to easily locate and manage configuration settings located in the Options dialog. The viewer is accessed by toggling the **Find...** button located in the lower left corner of the Options dialog.

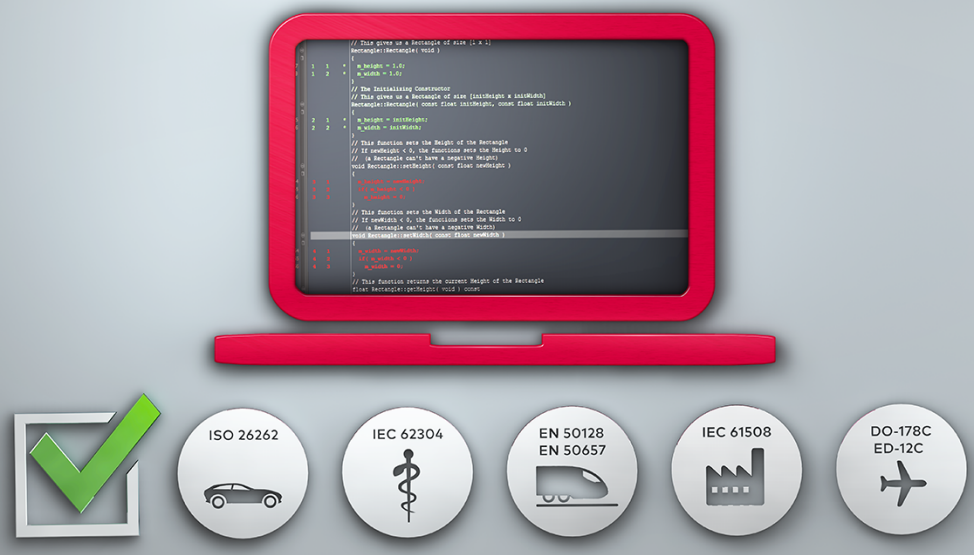


The Configuration Options viewer provides a filter to search for relevant options. As the user types, the viewer dynamically filters the list of settings, presenting only those configuration options that match the input criteria. Highlighting an option in the list displays the description of the selected option in the lower pane, where the matches to the search criteria are highlighted in red.



Upon selection of an option in the list, the corresponding Options editor interface is immediately brought into focus and the option field is highlighted in yellow. This allows the user to edit or review the current value of an option without the need to navigate through multiple layers of nested tabs.





Creating a New Environment

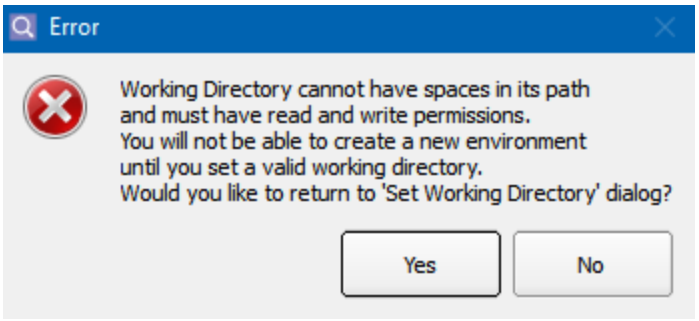
Using the Wizard to Create an Environment

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.

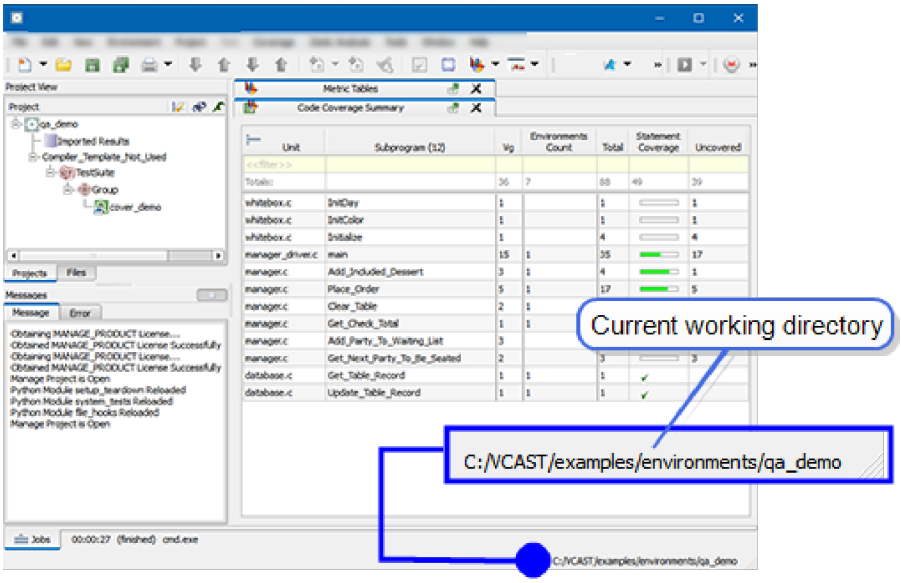
When you start the VectorCAST application from the command line, the working directory is the directory from which you invoked the application, unless the “Remember last working directory” option is set.

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.

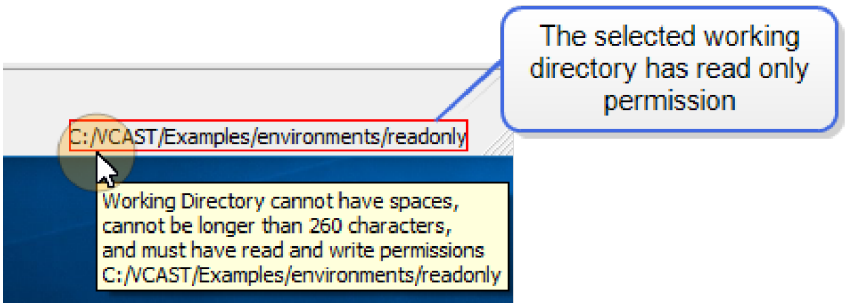
The current working directory is indicated in the status bar in the main application window.



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.)



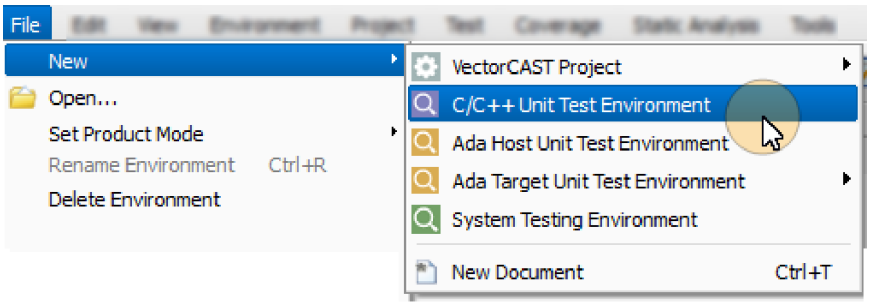
If you start VectorCAST from such a directory, the status bar has a red outline. Open the Set Working Directory dialog by selecting **File => Set Working Directory** from the Menu Bar and enter a valid directory.

To Start the Wizard

The **File => New** menu item is used to begin a new VectorCAST Project, C/C++ Unit Test Environment, Ada Host Unit Test Environment, Ada Target Unit Test Environment or System Testing environment. The menu changes depending on what products you have licensed from Vector.

Alternatively, the **New** button on the toolbar  has a small arrow to the right. Clicking this arrow causes a popup menu to appear, which lists the types of environments that can be built.

In the figure below, the user has licensed: VectorCAST/Project, VectorCAST/C++, VectorCAST/Ada, one or more Ada targets and VectorCAST/System Testing. Your menu may differ, depending on the products you have licensed from Vector.



If you have only one product licensed from Vector, then clicking the **New** button on the toolbar brings up the wizard to create a new environment of that product type.

Tip: The Create New Environment wizard reflects the current settings on the **Tools => Options** dialog, **Builder** tab. If you consistently want to create environments with Whitebox checked or Coverage on, for example, choose **Tools => Options**, and go to the **Builder tab** to set these options. Then, the next time you choose **File => New => Environment**, those settings are already specified, saving you time.

To Save the Settings in the Wizard

In the Create New Environment wizard, the **Save** button is used to create an environment script that reflects the data entered in the environment wizard. The name of the environment script is the environment name entered in the dialog with an ".env" extension. Once an environment is built, a script is automatically created. The **Save** button is used to create a script while working through the Create New Environment wizard.

If saving the environment script would modify the existing environment script of the same name, VectorCAST asks if you want to overwrite the original environment script with the changed version.

To create an environment script from a currently open environment, use **Environment => Scripting => Create**.

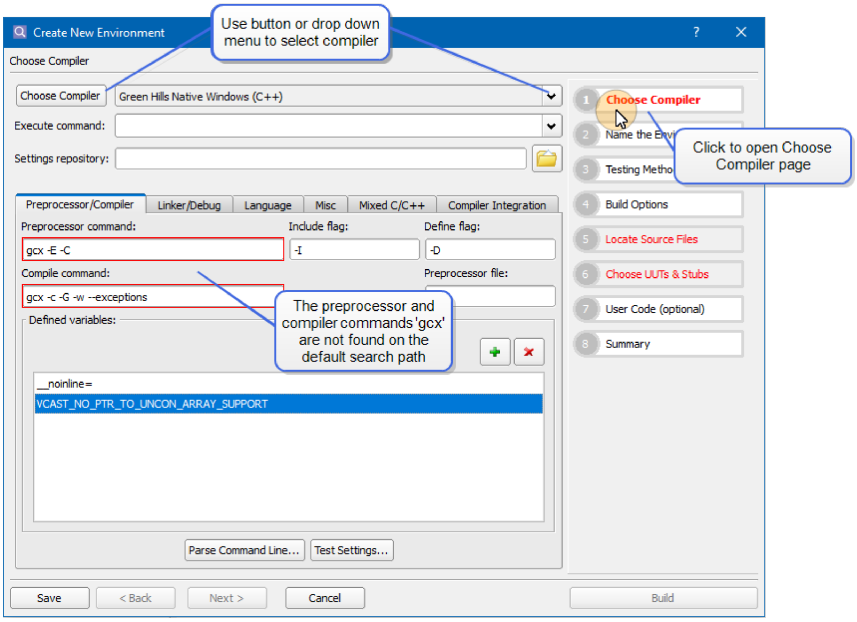
```
clicast -e <env> ENvironment SCript Create <scriptfile>

Create an environment script file from an existing environment. If no
extension is provided, .env is used.
```

Step 1: Choose Compiler

If you have already set your compiler, then the wizard skips over Step 1 and opens on **Step 2: Name the Environment.** You can click **Step 1: Choose Compiler** to go directly to Step 1.

On this page, you select the compiler. The preprocessor and compiler commands must be on your default system PATH. If you select a compiler and the preprocess or compile command is not on your PATH, then these commands are outlined in red. In this case you are not able to proceed to Step 2.



Select your compiler either from the dropdown menu next to the **Choose Compiler** button or click the **Choose Compiler** button itself.

- If you are building an environment with C source code files, choose a template name ending with “C”.
  Example**: GNU Native => 3.3 => C**
- If you are building an environment with C++ source files (or a mixture of C++ and C files), choose a template name ending with “C++”.
  Example: **GNU Native => 3.3 => C++**

If you find that the compiler you are using is not on your default search path, click **Cancel**, exit VectorCAST, and set up your compiler. Once you have selected a compiler that is on your PATH, the **Next** button enables.

```
clicast -lc TEMplate <compiler name>

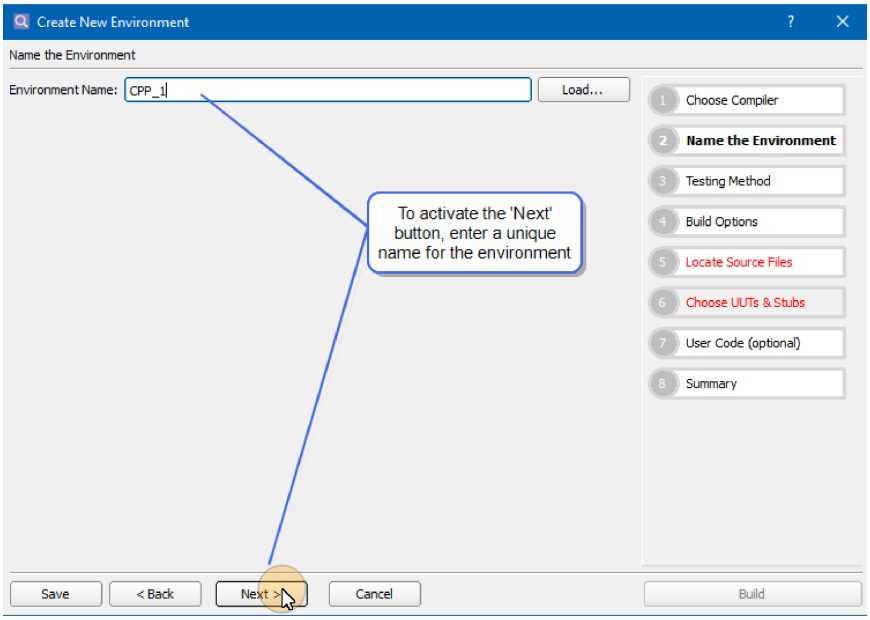
Initialize C/C++ Compiler options based on standard values for known C and
C++ compilers. Selecting a template automatically sets other options, such
as C_COMPILER_NAME, C_COMPILE_CMD, C_EXECUTE_CMD, and any other options
that are needed for the compiler chosen.
```

See also " Options: C/C++ Tab" on page 168 for more information on the various options and buttons on this page.

Step 2: Name the Environment

On this page you perform the following actions:

- Give the environment a name, which becomes a directory
- Load in an existing environment script (.env), which fills in the settings in the whole wizard (optional)



Environment Name: Enter a unique name you wish to give to the new environment. Any space characters you type are converted to "_" characters. A directory is created with this name, and a file is created with this name and a ".vce" extension. If a directory with that name already exists in the working directory, the name entry is outlined in red. The tool-tip asks you to choose a unique name for the environment.

To Load an Environment Script

Load... Button: Loads an existing environment script into the Create New Environment dialog to create an environment similar to an existing one. Click the **Load...** button, select an environment script (.env) and click the **Open** button. The wizard's pages are filled in according to the specifications in the environment script. At this point you can make any changes and then build the environment.

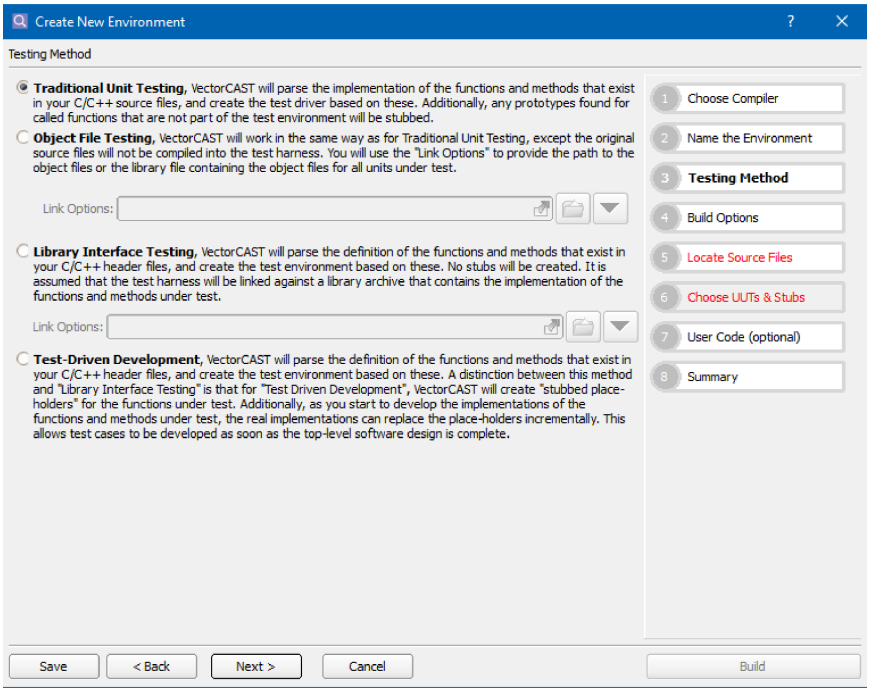
Tip: When loading an environment script that was saved with Relative Paths, ensure that your current working directory is the same as it was when the script was saved.

If you need to modify an *open* environment, say, to turn on Whitebox, add a Search directory/Parent Library, or change the stubbing strategy, use **Environment => Update Environment**. See also, "To Update an Environment" on page 222.

Step 3: Testing Method

On this page, you select the testing method:

- **Traditional Unit Testing** – VectorCAST parses your C/C++source files to create the test harness. In addition, any prototypes found for called functions that are not part of the test environment are stubbed.
- **Object File Testing** – VectorCAST works the same way as for Traditional Unit Testing except instead of compiling the original source files into the test harness, existing object or library files are used. When you select Object File Testing the **Link Options** text entry box becomes active. Specify an object file or a library archive to be linked to the VectorCAST test harness. Relative paths should be relative to the environment directory.
- **Library Interface Testing** – VectorCAST builds the test environment by parsing your C/C++ header files for function and method definitions. No stubs are created and it is assumed that the test harness will be linked to a library that provides method and function implementations. When you select Library Interface Testing, the **Link Options** text entry box becomes active. Specify the path to the object file, library archive, or DLL to be linked into the test environment.
- **Test Driven Development** – VectorCAST builds the test environment by parsing your C/C++header files for function and method definitions and creates stubbed place holders for the functions under test. This allows test cases to be developed as soon as the top-level software design is complete. As development proceeds and code implementations become available, the place holder stubs can be replaced with the actual implementation.



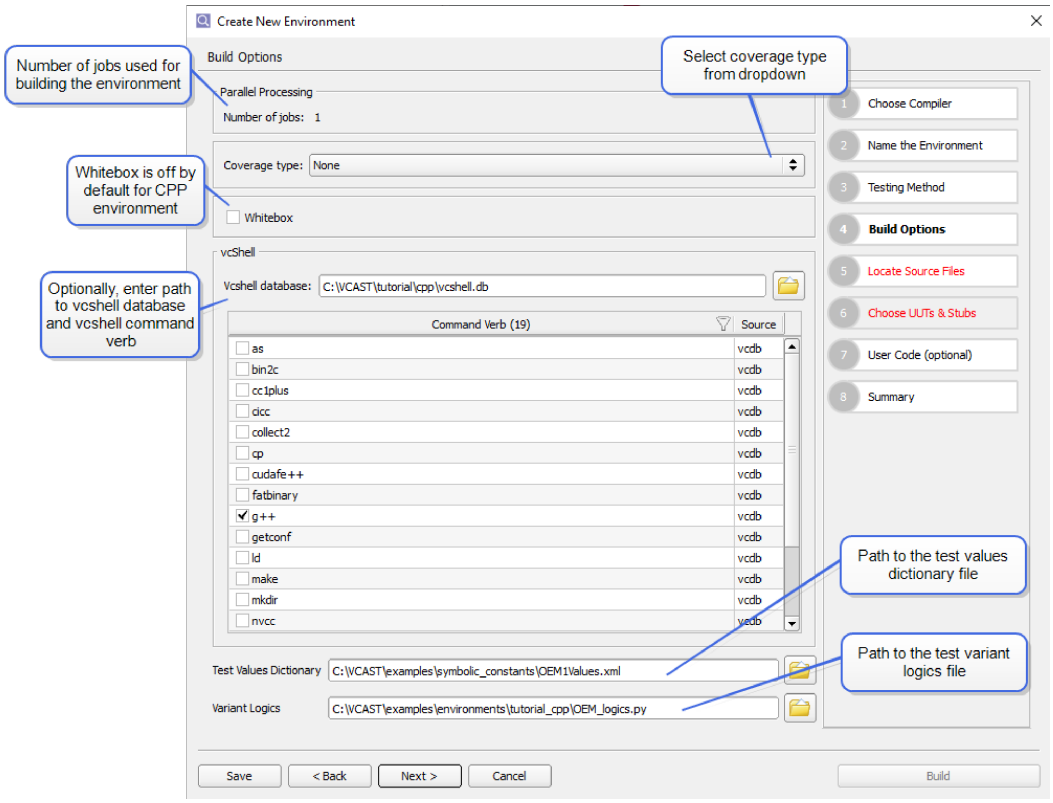
Step 4: Build Options

At the top of the Build Options page, the Parallel Processing pane displays the number of jobs used for building the environment.

Note: The value for number of jobs is for display purposes only. To change the value, set the VCAST_NUM_JOBS environment variable.

On this page, you select:

- **Coverage type** (optional) - Although coverage can be added at any time after the environment has been built, it is recommended that you select coverage here in the wizard for increased speed and efficiency.
- **Whitebox** (optional) - A Whitebox environment provides visibility to all internally defined entities. For C, this would be any function or object declared static. For C++, all private or protected members will be promoted to public.
- **vcShell database file** (optional) - Specify the path to the vcshell database (vcshell.db)
- **vcShell command verb** (optional) - Select the command verb used for the source files to be tested in the environment.
- **Test Values Dictionary file** (optional) - Specify the path to the Test Values Dictionary file.
- **Variant Logics file** (optional) - Specify the path to the Variant Logics file.



To Turn on Coverage

Coverage Type: Choose a type of code coverage for the environment from the dropdown menu, if desired. To build an environment without code coverage, set the Coverage Type to NONE.

To turn off coverage in an *open* environment, use the **Environment => Update Environment** command.

To set a Coverage Type as the default setting when you open the Create New Environment wizard, use the **Tools => Options** dialog, **Wizard** tab, **Coverage Type** option.

You can initialize code coverage at any time after the environment is built.

To Turn on Whitebox

Whitebox: If you wish to perform Whitebox testing, check the box marked **Whitebox**. The unchecked state is the default, unless you have checked "Whitebox" on the **Tools => Options** dialog, **Wizard** tab.

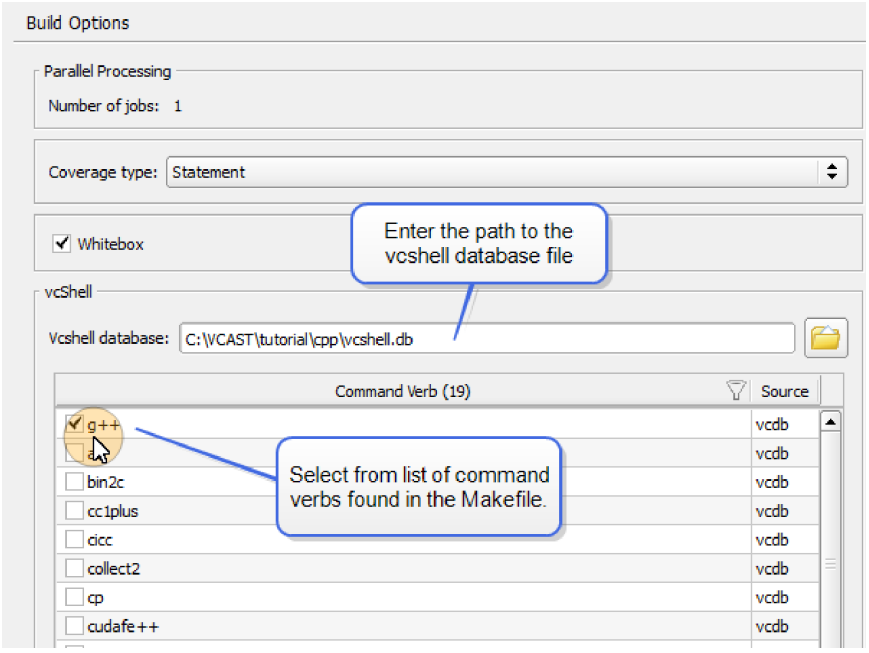
Whitebox enables you to test static-linkage functions and objects defined in the unit under test. For C++, it also gives you visibility to member functions and variables explicitly declared as private or protected.

To Use a vcshell Database to Build an Environment

VectorCAST provides users with the option to use a vcshell database to build an environment. When this option is selected, the wizard uses the information in the database to determine file locations, directory locations and unit -specific options.

vcShell database file: Specify the path to the vcshell database. The wizard uses the information in the vcshell database (vcshell.db) to determine the search directories, library include directories and type-handled directories, as well as any unit-specific options and file locations.

vcShell command verb: A list of command verbs found in the Makefile are listed in the Command Verb panel. Select the command verb used for the source files that you intend to test in that environment.



When the environment is built, the values of these options are written to the environment script (`.env`):

```
ENVIRO.VCDB_FILENAME: <path to vcshell database>
```

ENVIRO.VCDB_CMD_VERB: <command verb>

Once the vcshell database and command verb are specified, the directories are loaded into the Source Directories list on Step 5 of the wizard. Note that when using a vcshell database, the ability to add or remove source directories on Step 5 of the wizard is disabled.

Proceed through the wizard as usual, selecting the UUT(s) on Step 6. Here you can see the unit-specific options that were loaded if you switch to the "Unit Options" sub-tab.

The vcshell database and command verb used to build the environment are listed in the Overview report.

```
clicast -lc option VCDB_FILENAME <path to vcshell database file>

clicast -lc option VCDB_CMD_VERB <command verb>

clicast -lc option VCAST_VCDB_FLAG_STRING <list of flag strings>, such as
"-I=1, -D=1, -isystem=1" etc.,
```

For more information on **vcshell** and **vcdb**, see the *VectorCAST Utilities User's Guide*.

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 112.

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 Unit Test GUI.

- Go to “Build Options” 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.

To Select a Variant Logic File

To select a Variant Logics file, select a file path using the **File** Button or specify a known path using the line edit.

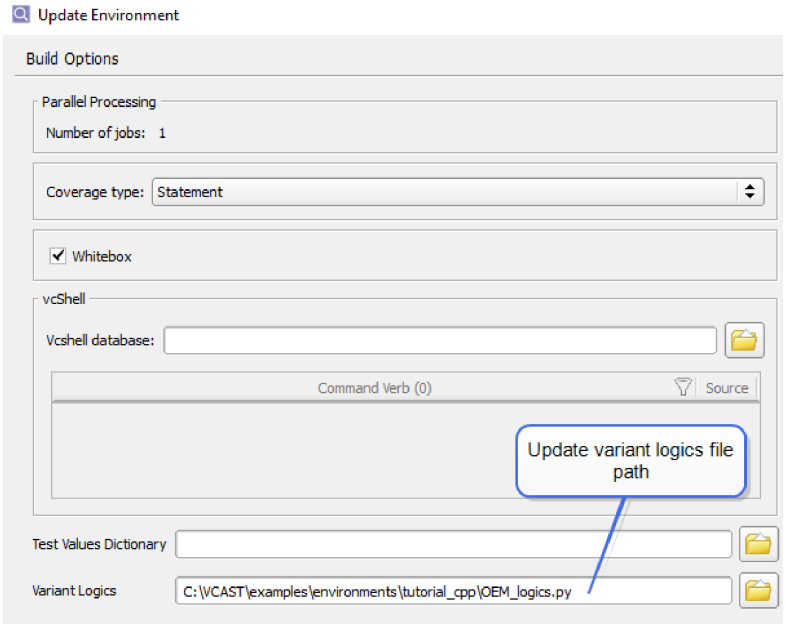
The path can be absolute or relative, and can contain environment variables, using the syntax **`$(ENV_VAR)`** or **`%ENV_VAR%`**.

To set a Variant Logics file as the default setting when you open the Create New Environment wizard,

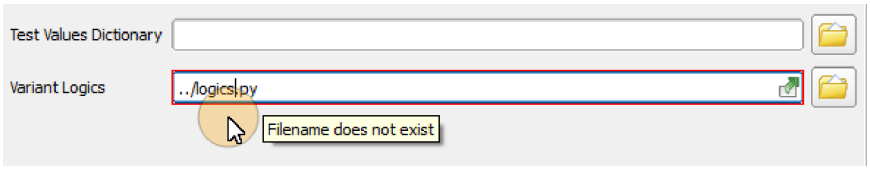
use the **Tools => Options** dialog. Select the **Wizard** tab, and then select the **Variant Logics** option.

Note: Loading an environment script containing a Variant Logics file using the Create New Environment Wizard (done by selecting **File => C/C++ Unit Test Environment** from the Menu Bar, and then selecting **Name the Environment** and the **Load…** button) will overwrite the current Variant Logics file. This is the default setting unless otherwise modified in the New Environment Wizard's Build Options Page.

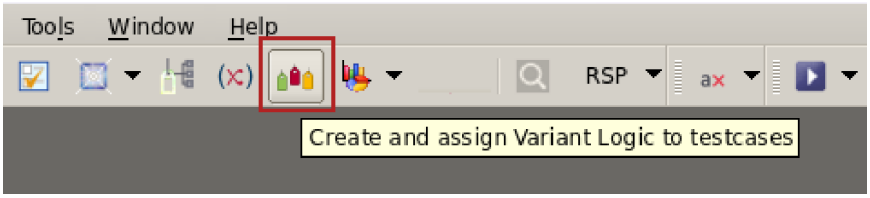
To replace or remove the Variant Logics file specified in an *open* environment, select **Environment => Update Environment...** from the Menu Bar and open the **Build Options** page.



If a Variant Logics file is specified, both the Create New Environment Wizard and the Update Environment Wizard require the file to exist at the specified path in order to build. When the specified file does not exist, it must be corrected prior to proceeding.



Once an environment is built and opened in the GUI, the Variant Logic File can be modified using the Logic Editor, which is accessed by selecting the Variant Logic button located in the VectorCAST toolbar.



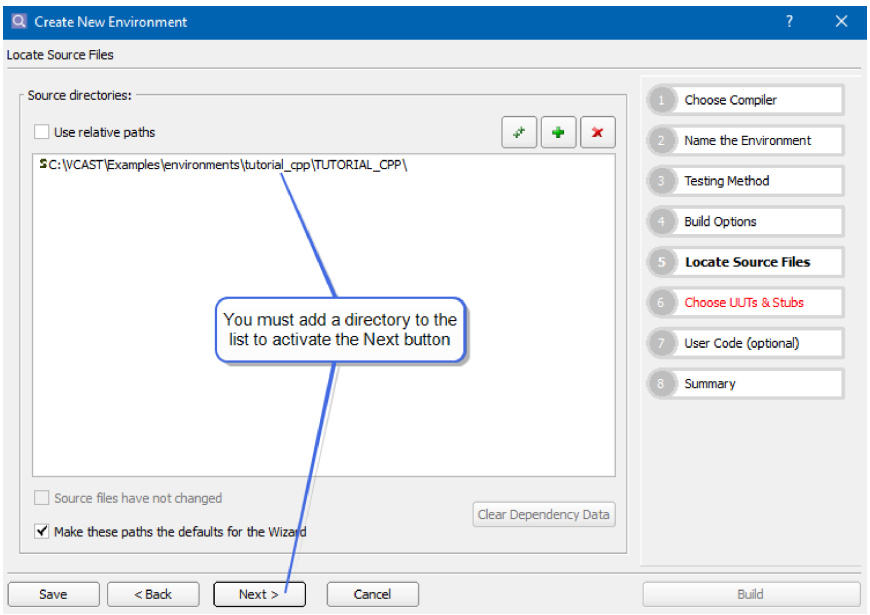
For more information on using Variant Logic, see "Control Test Case Execution Using Variant Logic" on page 400.

Step 5: Locate Source Files

On this page you perform the following:

- Add one or more directories (called *source directories*), in which VectorCAST looks for units to test or stub and header files
- Add Library Include directories, in which the compiler will look for library headers (optional)
- Indicate that the Source directories should be specified relative to the working directory (optional)
- Clear the cached parse data (optional)
- Indicate that the source files have not changed since the last parse (optional)
- Indicate that these source directories should become the default paths for the Create New Environment wizard (optional)

Note: When using a vcshell database, the ability to add or remove source directories is disabled.



Source Directories: These are the directories that VectorCAST searches when looking for source

files, header files, and system libraries.

A *source directory* can be one of three types:

- Search directory – units in this type of directory are available to be UUTs. VectorCAST parses the units in these directories to make them testable or stubbable.
- Library include directory – units in this type of directory are typically third-party libraries. VectorCAST does not test, stub, or parse units for data types.
- Type-handled directory – units in this type of directory are parsed for data types, but are not made testable or stubbable.



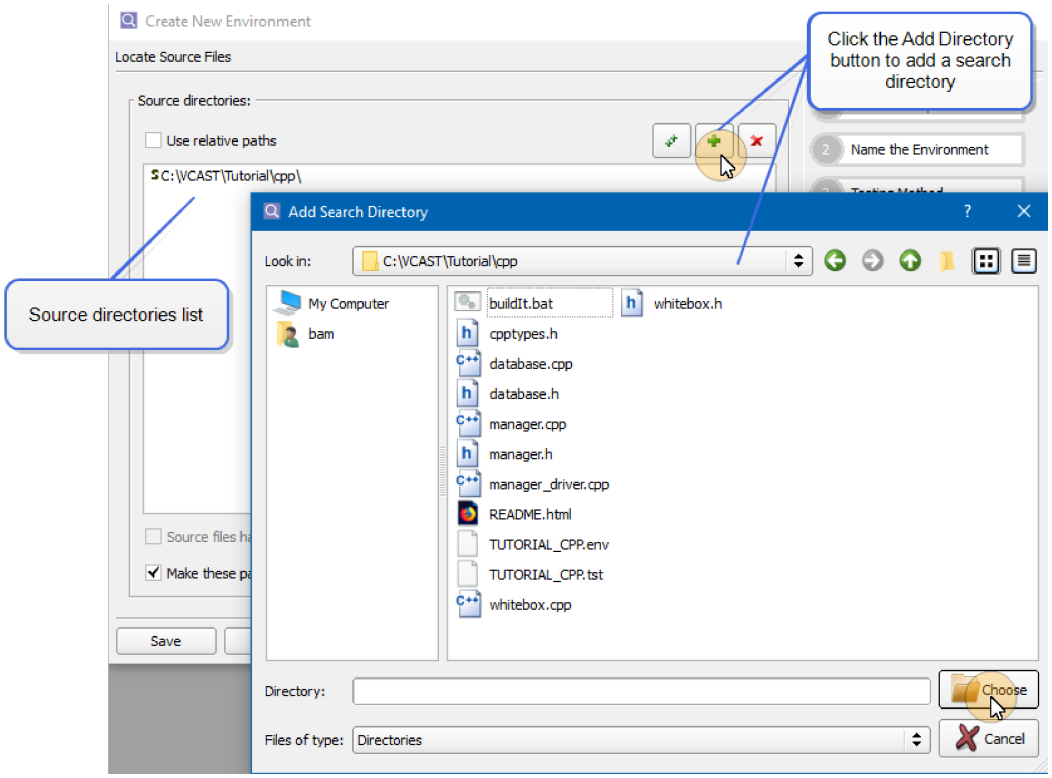
Note: When using a vcshell database, changes to the type of a search directory made in the wizard are written to the vcshell database during the environment build process.

Source directories may be already present in the wizard if you have:

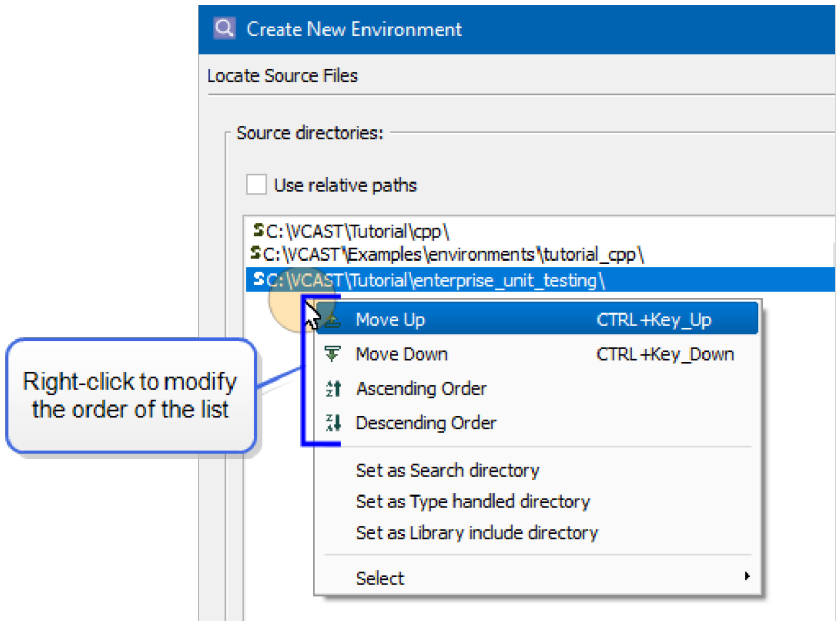
- Specified default source directories on the **Tools => Options** dialog, **C/C++** tab or you used clicast to specify default source directories
- Added directories via the compiler template
- Loaded build settings from a repository
- Loaded an environment script (.env)
- Loaded build settings from a specified vcshell database

To Add a Search Directory

To add a search directory to the Source directories list, click the **Add Directory** button  . An Add Search Directory dialog appears, enabling you to locate a directory containing source files. On Windows, you can also drag and drop a path from Windows Explorer to the Source directories list.

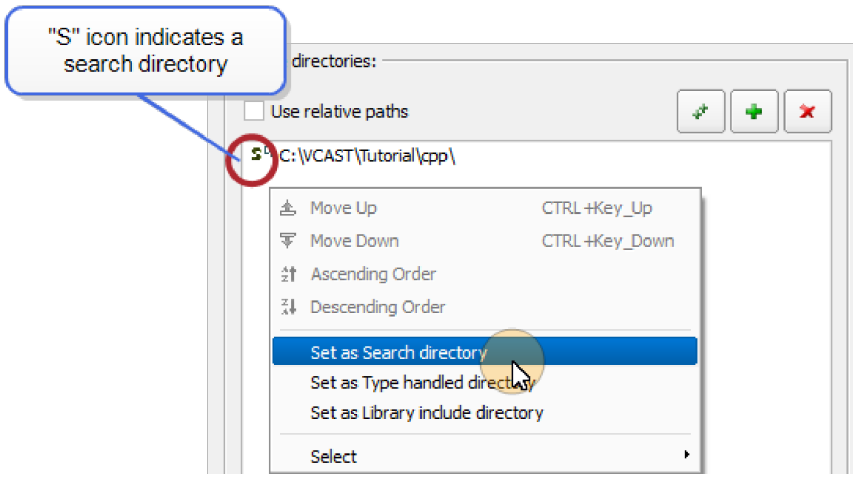


The directory paths are sorted alphabetically in the list by default. The order of the list is the order in which VectorCAST searches them. Modify the order of the list by right-clicking and selecting from the context menu. You can move directories up and down the list, and sort the list in alphabetical and reverse-alphabetical order.



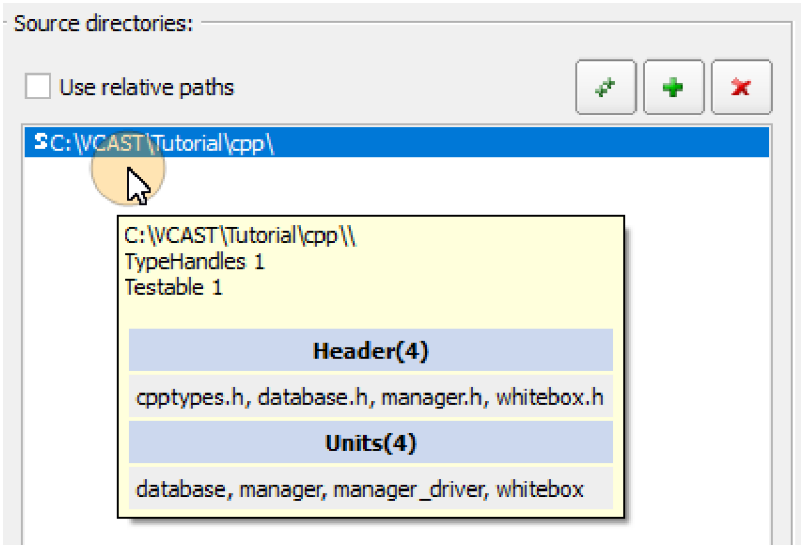
When a directory path is added to the Source directories list, it is set as a Search directory, by default.

To set a directory path as a Search directory, select the path, right-click the path and choose Set as Search directory from the menu. The S icon indicates a Search directory.



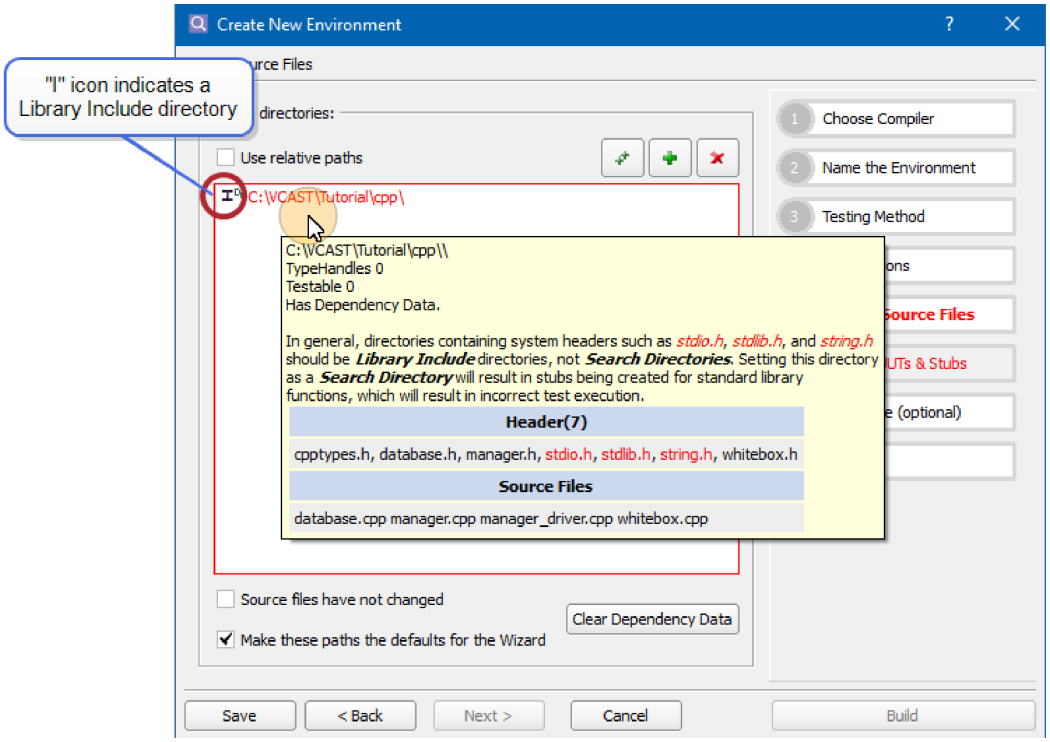
All source code files with an extension .c or .cpp (and others listed on the **Misc** sub-tab) are located.

The tool-tip for each directory lists the units found in that Search directory.



When you add or remove directories from the Search List, VectorCAST updates the “Unit Names” list on the next page of the wizard (Step 6: Choose UUTs & Stubs).

Adding a directory containing system header files can cause problems during test execution, because standard library functions might be unintentionally stubbed. When adding a search directory, if a directory is detected to contain standard library functions, the entry in the Wizard is set to red to alert the user, and the search directory is automatically changed to a Library Include directory (see "To Add a Library Include Directory" on page 118). Hover over the entry and the tool tip provides further detail.



See also ENVIRO.SEARCH_LIST in "Environment Script Language" on page 661 for the syntax used in environment scripts to specify a search directory.

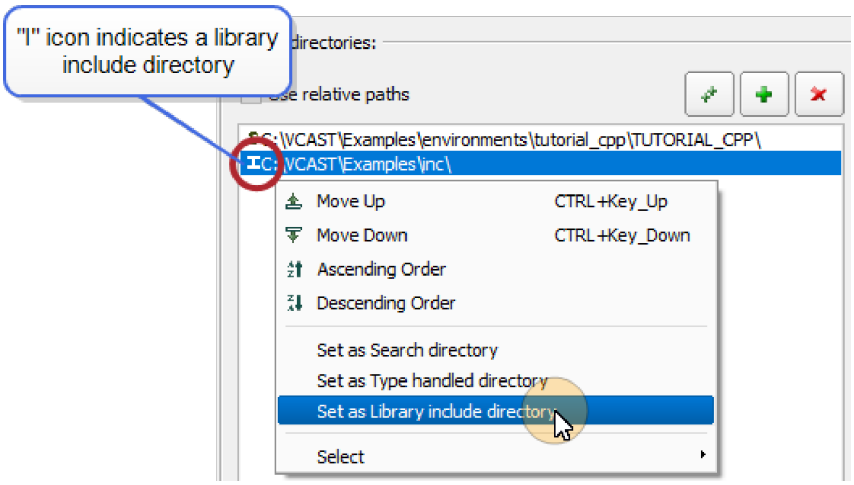
See also "Default Source Directories for Wizard" on page 172 for the clicast command used to specify that a directory path be used as a default search directory.

To Add a Library Include Directory

If you normally add an object file or library on your link command line, then you need to add that directory to the Library Include directories list. Header files that exist in a Library Include directory are considered system headers. VectorCAST does not stub any units found in the Library Include directories, and does not create extern variables for symbols found. Furthermore, unless you add them as a library stub, the object file for any symbol needed from a Library Include directory must be linked in by you. You can do this by editing the Linker options on **Step 1: Choose Compiler**, **Linker/Debug** sub-tab.

To add a Library Include directory to the Source directories list, click the **Add Directory** button . A directory navigation dialog appears, enabling you to locate a directory containing the library includes or system headers. On Windows, you can also drag and drop a path from Windows Explorer to the Source directories list.

When a directory path is added to the Source directories list, it is set as a Search directory, by default. To set a directory path as a Library Include directory, select the path, right-click the path, and choose **Set as Library include directory** from the menu. The icon indicates a Library Include directory.



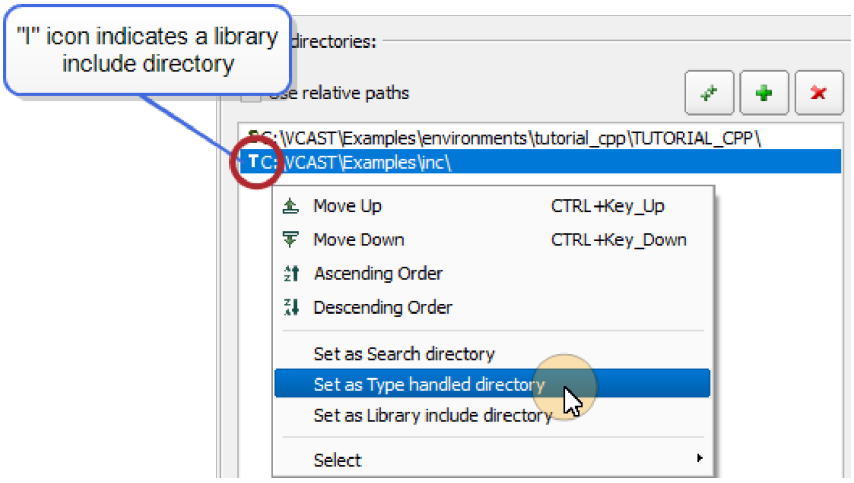
See also “ENVIRO.LIBRARY_INCLUDE_DIR in "Environment Script Language" on page 661 for the syntax used in environment scripts to specify a Library Include directory.

See also "Default Source Directories for Wizard" on page 172 for the clicast command used to specify that a directory path be used as a default Library Include directory.

To Add a Type-Handled Directory

To add a Type-Handled directory to the Source directories list, click the **Add Directory** button . A directory navigation dialog appears, enabling you to locate a directory. On Windows, you can also drag and drop a path from Windows Explorer to the Source directories list.

When a directory path is added to the Source directories list, it is set as a Search directory, by default. To set a directory path as a Type-Handled directory, select the path, right-click the path, and choose **Set as Type handled directory** from the menu. The T icon indicates a Type-Handled directory.

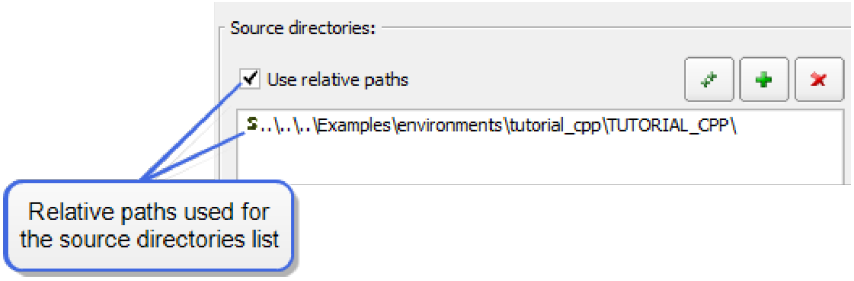


See also ENVIRO.TYPE_HANDLED_SOURCE_DIR in "Environment Script Language" on page 661 for the syntax used in environment scripts to specify a Type-Handled directory.

See also "Default Source Directories for Wizard" on page 172 for the clicast command used to specify that a directory path be used as a default Type-Handled directory.

To Use Relative Paths for the Source Directories

Check the **Use relative paths** box to change the full paths in the Source directories list to shortened paths, relative to the current working directory. The relative path is saved to the environment script when the environment is finished building and when you click the **Save** button .



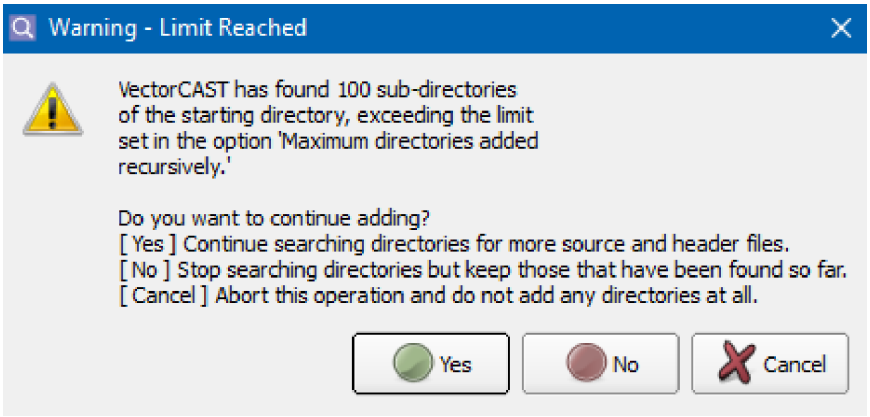
This feature is useful when several users are working with an environment, and each has a different “home” directory, but the directory structure below that is the same.

To set this option every time you build a new environment, check the box for **Use Relative Paths for search list** in the **Tools => Options** dialog, **Wizard** tab.

To Add Search Directories Recursively

To add a whole tree of directories to the Source directories list, click the **Add Search Directory Recursively** button . A directory navigation dialog appears, enabling you to locate the top-most, or starting, directory. All directories below this starting directory are searched for source and header files. If it has either, it is added to the Search List.

By default, VectorCAST looks through 100 directories for import sources. When this limit is reached, a message appears:



If you chose the wrong starting directory and want to abort the operation, click **Cancel**. If you want to continue searching the next 100 directories for environments or regression scripts, click **Yes**. If you want to stop the operation but retain the directories already found, click **No**. This limit, called "Maximum directories added recursively," is configurable. It is located on the **Tools => Options** dialog, **GUI** tab.

Only directories that contain VectorCAST environments, or VectorCAST regression scripts are added to the list. While determining if a directory should be added to the list, the Wizard:

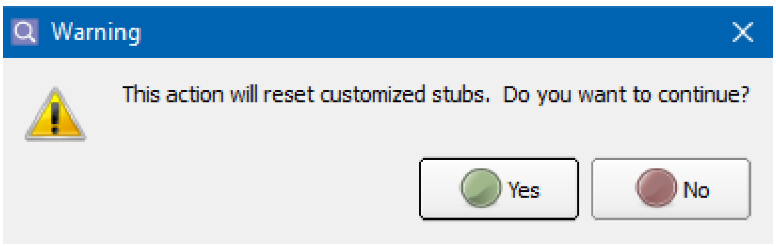
- Disregards VectorCAST backup (.BAK) environments
- Disregards directories that do not contain valid VectorCAST environments (.env and .vce files with matching names) or valid VectorCAST regression scripts (.env and .bat/.sh files with matching names)

To Remove a Search Directory

To remove a directory from the Source directory list, first click the directory, then click the **Remove Search Directory** button  .

When you add or remove a search directory, VectorCAST updates the "Unit Names" list on the next page of the wizard (Step 6: Choose UUTs & Stubs).

If you have previously chosen "Custom" as your stubbing method, and selected "Show Dependency View", VectorCAST will prompt to reset the stubbing method to the default mode. These updates are necessary because modifying directories to the Search List results in different dependency data being computed for UUTs. The following warning dialog appears. "This action will reset customized stubs. Do you want to continue?" Click **Yes** to reset the stubbing method to the default, or **No** to abandon the operation.



To Clear the Dependency Data

When VectorCAST parses the UUTs to determine dependencies, the parser caches that information in a file named VCAST.QIK. This file is written to each Search directory or in an alternative location you specify.

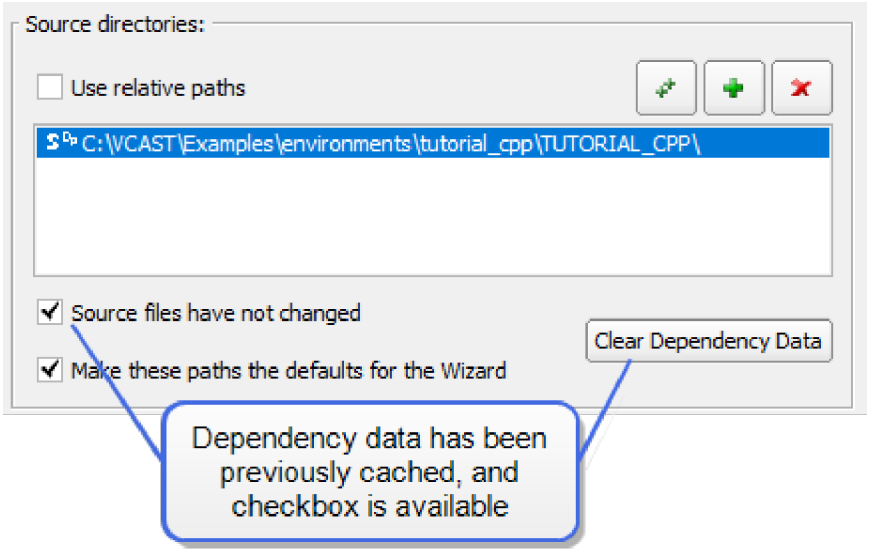
Each time you build an environment, if any files have changed, the dependency cache is updated. VectorCAST determines if a file has changed by computing a checksum for the preprocessed file. If this file is corrupted or if you switch compilers, you should remove any existing parse data. To do this, click the **Clear Dependency Data** button. The cached data is removed from each directory in the list.

Note: Clearing the dependency cache does not apply to QIK files created when using the Stub None (Object-Files) option. Only legacy QIK files generated through the **Show Dependency View** button are removed.

To Skip Re-Parsing of Source Files

By default, the **Source Files Have Not Changed** checkbox is unchecked, which causes VectorCAST to check the time tag and compute a checksum for each file in the Search Directories. If you know that your source files have not changed, you can save time when building an environment by checking this option. When this option is checked, the parser only parses the Units Under Test.

The **Source files have not changed** checkbox is dimmed if there is no dependency data already cached.



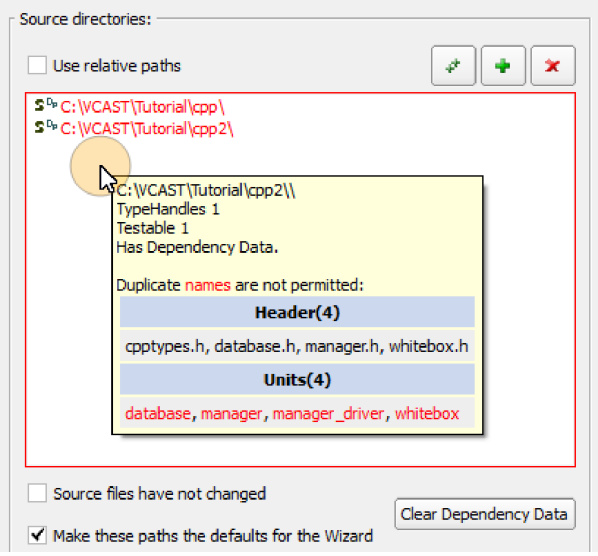
When using this option, there are several points to be aware of:

- If a source file has changed yet the “Source Files Have Not Changed” option is checked, then the dependency data used to create the environment may be invalid.
- If, for some reason, a VCAST.QIK file has been deleted from a Search directory, then you will see an error when you click **Custom** followed by the **Show Dependency View** button, or when you build. The error message that appears in the Message window is similar to: “The filename '<unit name>' could not be found in any of the VCAST.QIK files. Can't proceed without quickparse data for this file.”
- If you specify a directory for the Builder option “Dependency data directory” and are building a new environment and you have turned on “Source Files Have Not Changed,” then you will see an error when you click Custom and the Show Dependency View button, or when you build.

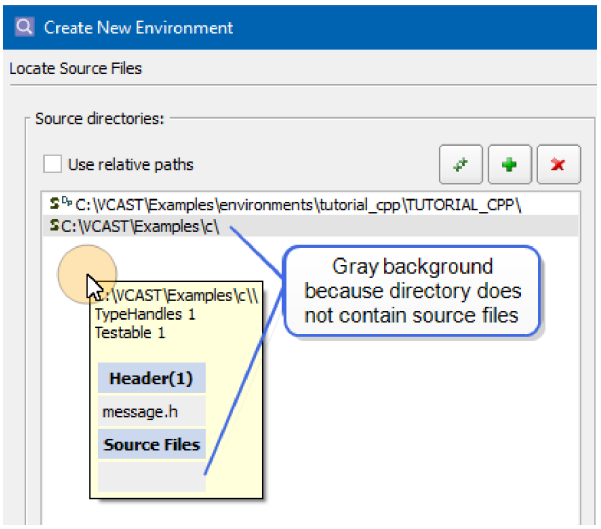
Troubleshooting Search Directories

If a Search directory contains any unit names that are duplicated in another Search directory, then both paths turn red, and the tool-tip provides information. Before going to the next step in the wizard,

remove one of the paths.



If you add a Search directory that does not contain source code units (it might contain header files), then the Search directories path is colored with a gray background, and the tool-tip provides information. Before proceeding to the next step in the wizard, add a Search directory that contains at least one source code unit. Any file with an extension matching those listed in “Unit Extensions” is recognized (Step 1: Choose Compiler, click the Misc tab).

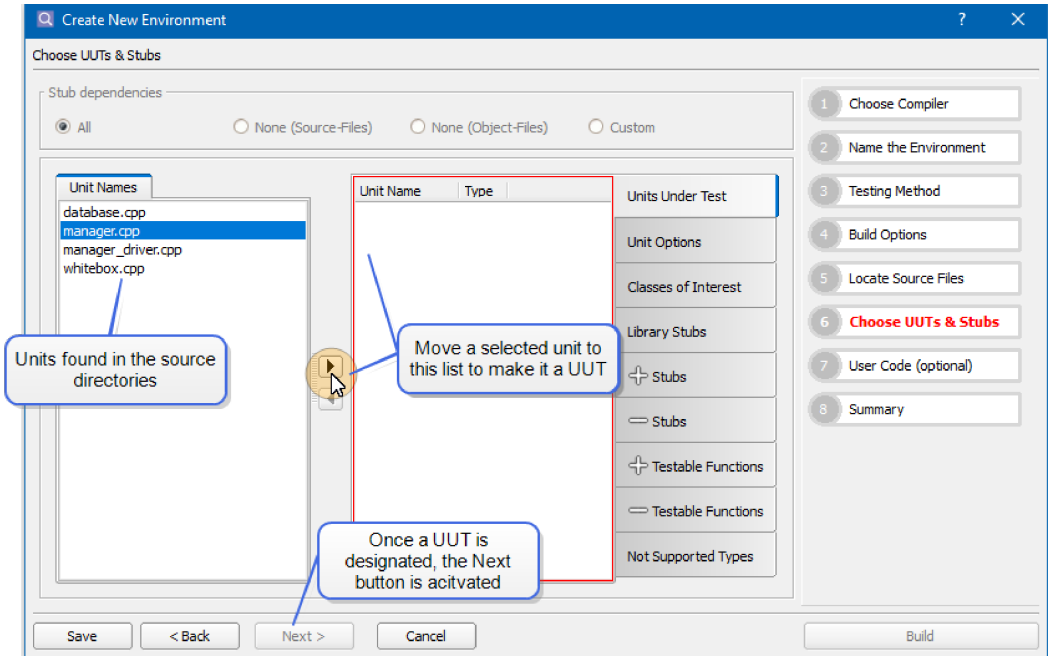


Step 6: Choose UUTs & Stubs

This page of the wizard, called **Choose UUTs & Stubs**, is used to specify the Units Under Test (UUTs) and their optional compilation arguments, stubs, and Classes of Interest.

On this page you perform the following:

- Specify the Unit(s) Under Test (UUT)
- Choose stubbing type for all dependents or customize the stubbing for individual units (default is to stub all dependents by prototype (sbf), unless specified otherwise on the **Tools => Options, Wizard tab**)
- Enter compilation options for individual units (optional)
- For C++ environments, choose classes of interest other than the default (optional)
- Specify library functions to stub in this environment (optional)
- Specify Additional Stubs (optional)
- Specify Suppressed Stubs(optional)
- Specify Additional Testable Functions (optional)
- Specify Suppressed Testable Functions (optional)
- Specify a type that you do not want supported in the environment (optional)



Specifying Unit(s) Under Test: The units found in the directories specified in the Source directories list are displayed on the left, under the tab labeled “Unit Names.” Removing or adding a directory to the Source directories list changes the items in this list. These units are “testable.”

You may specify more than one Unit Under Test.

To specify a unit as a Unit Under Test, do one of the following:

- select a unit name and click the move-to-right button 
- drag and drop a unit name into the Units Under Test tab
- double-click a unit name.

To remove a unit from the Unit Under Test list, do one of the following:

- select the name and click the move-to-left button 
- drag a UUT from the Units Under Test tab and drop it in the Unit Names list
- double-click a UUT name.

Stubbing Type for Dependencies: When unit testing, it is often desirable to stub functions called by the unit you are testing. Stubbing allows you to test without waiting for dependent units to be completed and tested. VectorCAST stubs enable you to capture parameters passed in, control parameters passed out, and raise exceptions.

When you first open the Create New Environment wizard, the default stubbing strategy is to stub All of the dependent units by their prototype. (You can change the default by going to the **Tools => Options dialog, Wizard tab**, and changing it to None. Thereafter, when you open this dialog, it will default to None or Custom None dependent units to be stubbed.)

There are four choices for the stubbing strategy:

- **All** – Stub all dependent units of the UUT(s) without parsing any other units in the Search directories. VectorCAST doesn't determine any of the dependencies between the various source files. VectorCAST assumes that the prototypes for any functions called by the UUT(s) exist in header files found in the Search directories.
- **None (Source-Files)** -- Do not stub any dependent units of the UUTs. Compiles the dependent units serially.
- **None (Object-Files)** -- Do not stub any dependent units of the UUTs. Compiles the dependent units in parallel, using previously compiled files found in a vcshell database. See "To Stub None Using Object Files" on page 126 for more information.
- **Custom** – Specify particular units to be stubbed by prototype, stubbed by implementation, or not stubbed.

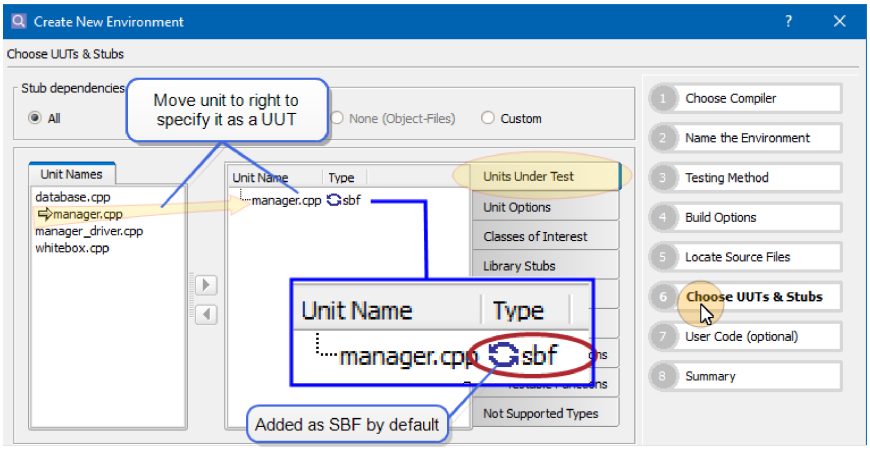
See also “ENVIRO.UUT” in "Environment Script Language" on page 661 for syntax used in the environment script.

To Stub a UUT by Function

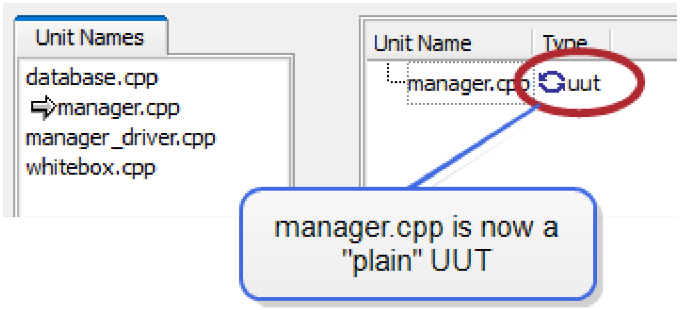
VectorCAST has the capability to stub individual functions within a UUT on a per-function basis. In the Create New Environment wizard, UUTs are set for stubbing by function by default. After the environment is built and a test case inserted, you then indicate which functions, if any, you want to stub when the test case is executed.

To stub a UUT by function, go to **Step 6: Choose UUTs & Stubs** of the Create New Environment wizard and do the following:

1. Select the Units Under Test tab.
2. Select a unit from the Unit Names column on the left and move it to the right to specify it as a unit under test. By default, it is a stubbable-by-function (SBF) unit.



To change a unit to a “plain” UUT, without the capability of stubbing individual functions at test case execution, click the **SBF icon** sbf . The icon will change to the **UUT icon** uut indicating that the unit is now a "plain" UUT.



See also ENVIRO.STUB_BY_FUNCTION in"Environment Script Language" on page 661 for syntax used in the environment script.

The default setting, SBF, is specified on the **Tools => Options** dialog, **Wizard** tab, **Unit Type** option.

To Stub None Using Object Files

Selecting the option **Stub None (Object-Files)** compiles the dependent units in parallel using information found in a vcshell database. Environment building with this method is much faster because it uses the previously compiled files.

Note: The **Stub None (Source-Files)** option parses and compiles each dependent source file serially, making it slower than the **Stub None (Object-Files)** method which compiles the dependent units in parallel.

Environments built with **Stub None (Object-Files)** do not permit Custom Coverage to be applied to the dependent units, as is allowed in environments built with **Stub None (Source-Files)**.

How to Use the Stub None (Object-Files) Method

1. A vcshell database (by default named **`vcshell.db`**) must be created. This is accomplished by calling VectorCAST's vcshell command which uses the "make" or "compile" command for your existing code base. Be sure to use the same compiler with which you will build the environment.

   ```
   vcshell <make or compile command>
   ```

   For more information see "To Use a vcshell Database to Build an Environment" on page 110. Also see the *VectorCAST Utilities User's Guide*.

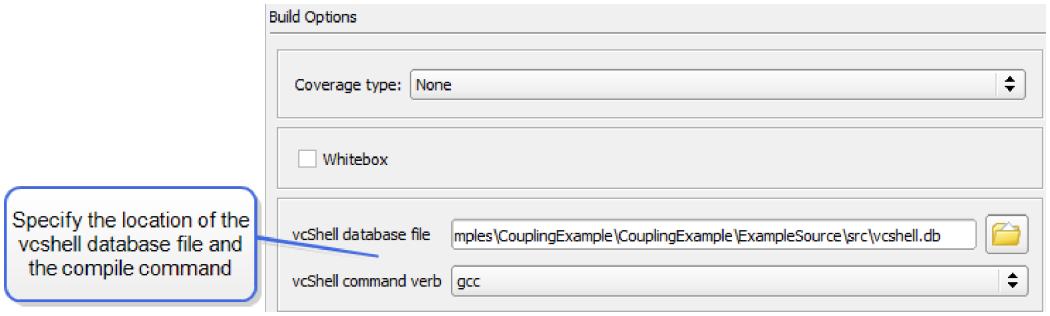
2. Set the compiler template for VectorCAST. See "Compiler Template" on page 168 for more information.

   ```
   clicast -lc TEMPLATE <template_name>
   ```

3. Using VectorCAST's Parser utility, parse the source files in parallel and compile them, placing both the object files and .QIK files in a directory named **`vc-parse`**. One .QIK file is created for each dependent unit, named **`<uni>.<unit number>.QIK`**.

   ```
   vcutil parse
   ```

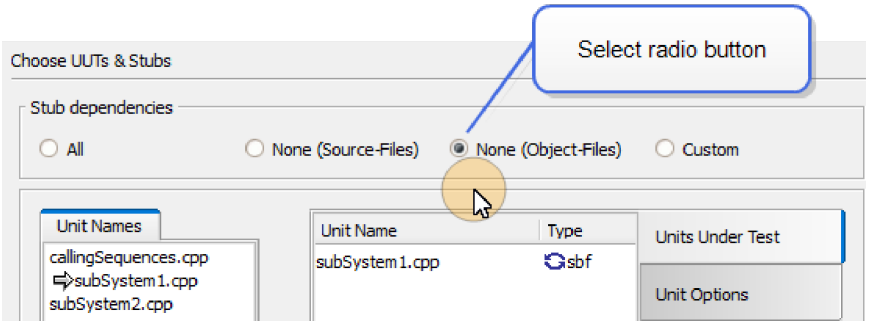
4. On **Step 4 Build Options** of the Create New Environment wizard, specify the location of the vcshell database file. Note that the directory containing the vcshell database file must also have a vc-parse directory containing valid quickparse files from running **`vcutil parse`**.


   ```
   clicast -lc option VCDB_FILENAME <path to vcshell.db file>
   clicast -lc option VCDB_CMD_VERB <compile command used>
   ```

   Note that the **None (Object-Files)** option will be unavailable for selection on **Step 6 Choose UUTs & Stubs**, if a path isn't provided for the vcshell database file here on **Step 4 Build Options**.

5. On **Step 5 Locate Source Files** of the wizard, the Search Directories and Unit Options are filled in as specified in the vcshell database.
6. On Step 6, Choose UUTs & Stubs, select a unit as the UUT. Then select the radio button **None (Object-Files)**.



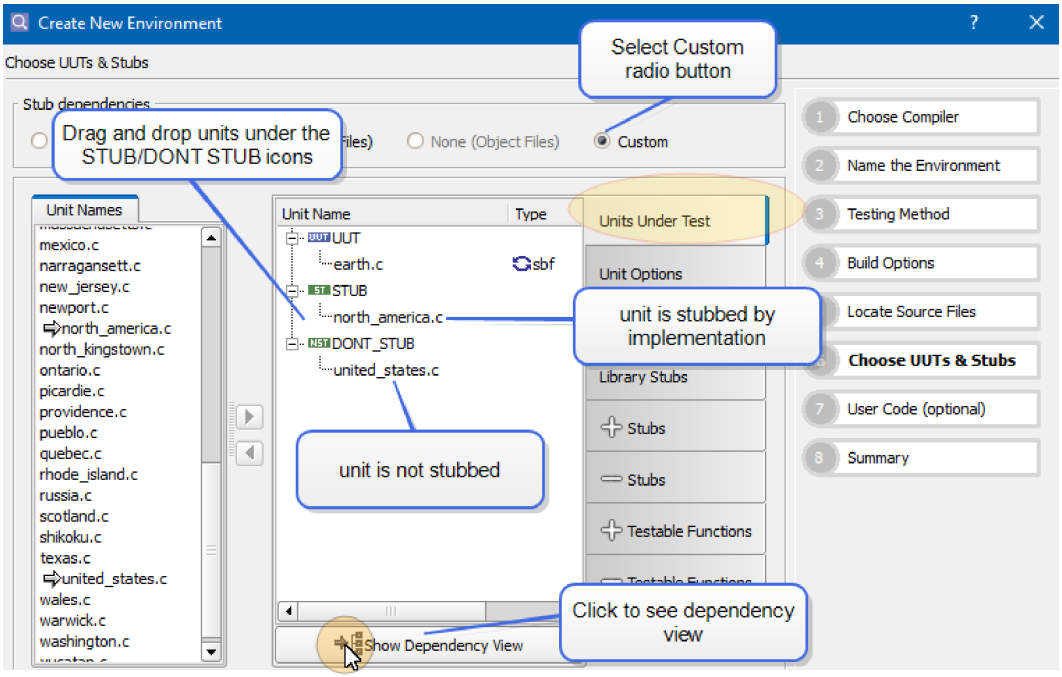
7. Select the **Build** button to finish building the environment.

To Specify Customized Stubbing

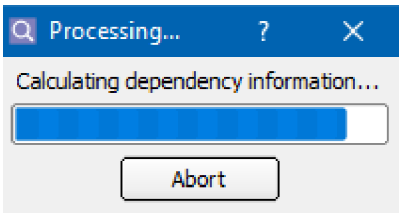
If you want to specify that some dependent units be stubbed and some not, click the radio button next to **Custom**. The Units Under Test tab changes to a "list mode", with the UUT listed under the icon UUT UUT . By dragging and dropping units onto the STUB (ST STUB) or DONT_STUB (NST DONT_STUB) icons, you can specify that some units be stubbed by implementation and some not stubbed. Unspecified units are stubbed by prototype. These stubbing types are explained below:

- not stubbed – Drag and drop a unit under the DONT_STUB icon (NST DONT_STUB). VectorCAST will not stub this unit; any dependents are parsed and can have their own setting. When building begins, VectorCAST will parse this unit's source file to determine its dependents.
- stubbed (by prototype) – This is the default for units not in the list. Stub this unit by parsing header files only; any dependent units are not testable.
- stubbed (by implementation) – Drag and drop a unit under the STUB icon (ST STUB). When building begins, VectorCAST will stub this unit by parsing the unit's source file. Stubbed (by implementation) has been deprecated and has been replaced by Stubbed (by prototype).

In the following figure, **earth.c** is the UUT, **north_america.c** is stubbed (by implementation), and **united_states.c** is not stubbed. All others are stubbed by prototype.

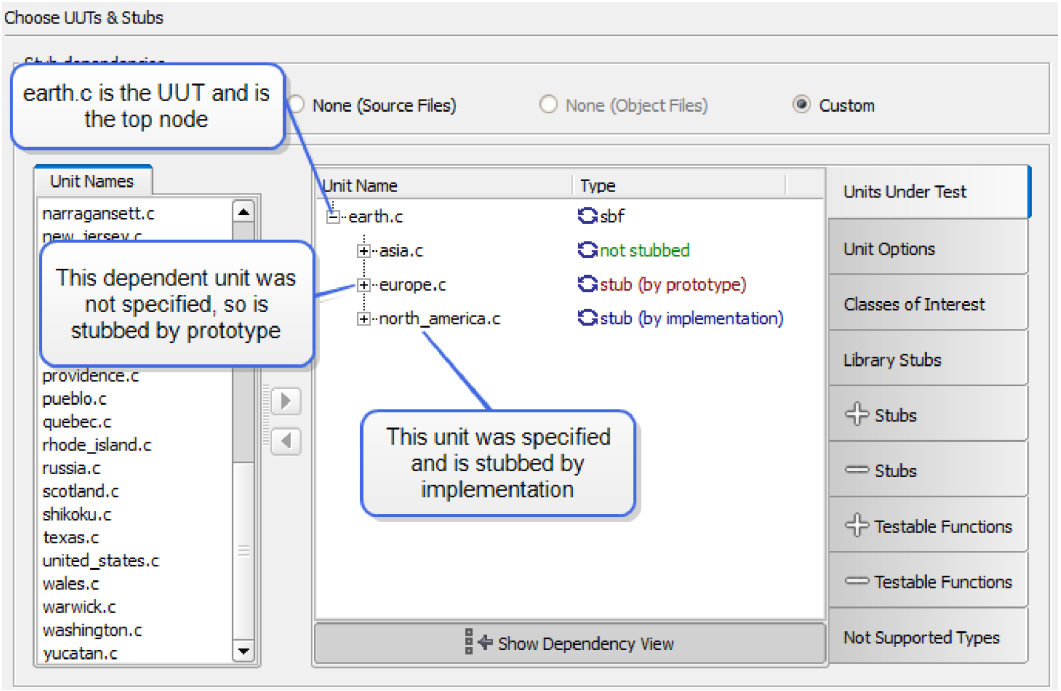


To see a hierarchical display of the dependencies, click the **Show Dependency View** button. This toggle button has the icon  . Once you click this button, VectorCAST parses all source code files it finds in the Source directories list to determine the dependents of UUTs. Note that this operation is expensive and may take some time to complete. While it calculates the dependency information, you see the following status dialog:

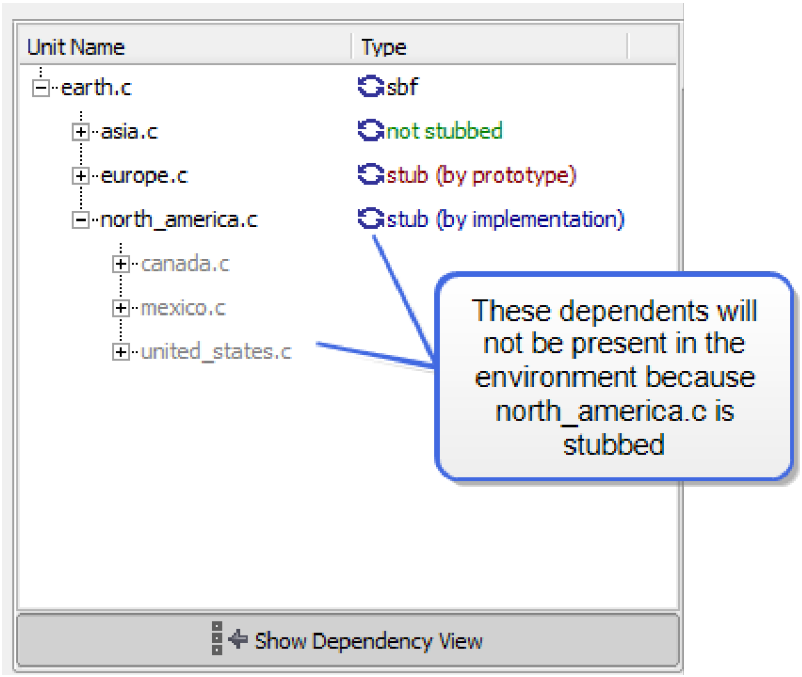


After the dependency information is generated, the Units Under Test tab shows a hierarchical tree of the source files in the environment. Units Under Test are top nodes of the tree, so their names appear all the way to the left.

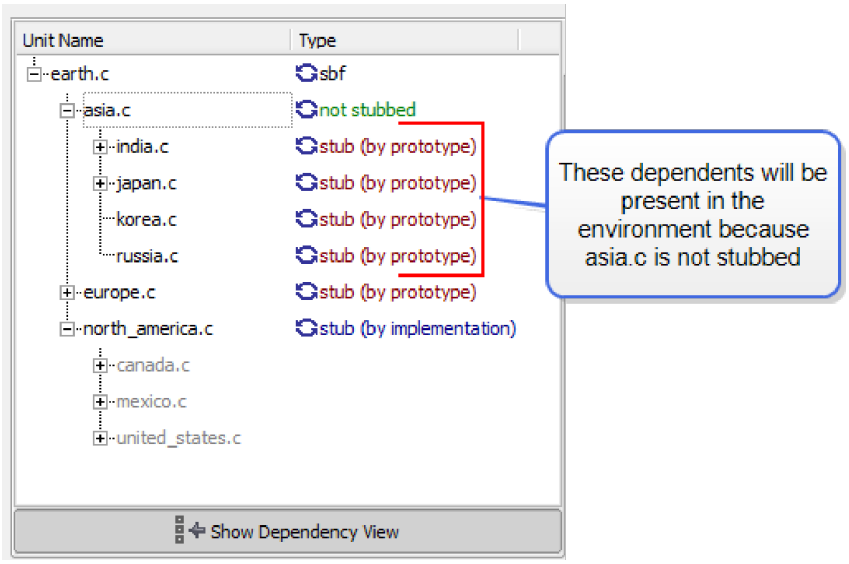
The figure below shows that **north_america.c** is stubbed (by implementation), and **asia.c** is not stubbed. **europe.c** is a dependent of **earth.c** and is stubbed (by prototype) because it was not specified otherwise.



Dependents of stubbed units (by prototype or by implementation) are displayed in the tree as gray, so that you may be aware of their relationship to other units, even though they are not part of the environment. In the figure below, **`canada.c`**, **`mexico.c`**and **`united_states.c`** were found to be dependents of **`north_america.c`**, but since **`north_america.c`** is stubbed, they are not going to be present in the environment when it is built.

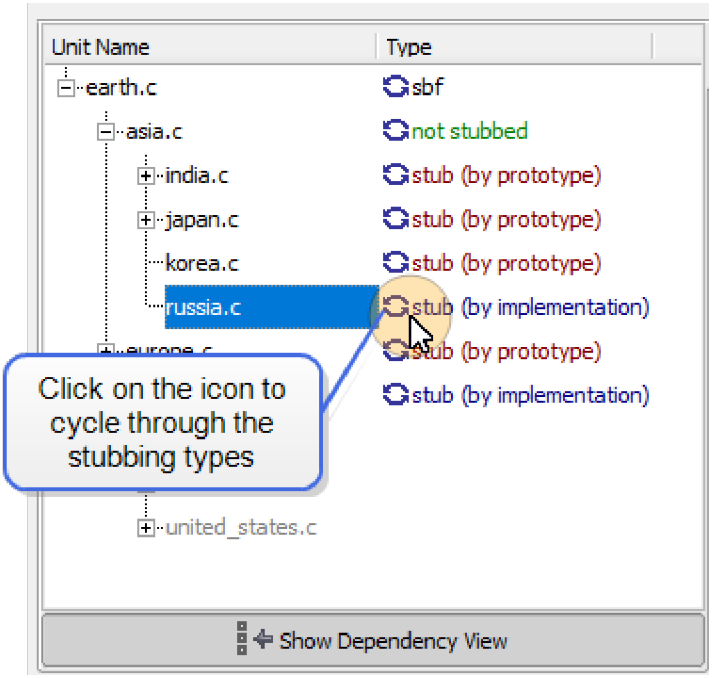


By contrast, the dependents of **`asia.c`** will be present in the environment, because **`asia.c`** is not stubbed.

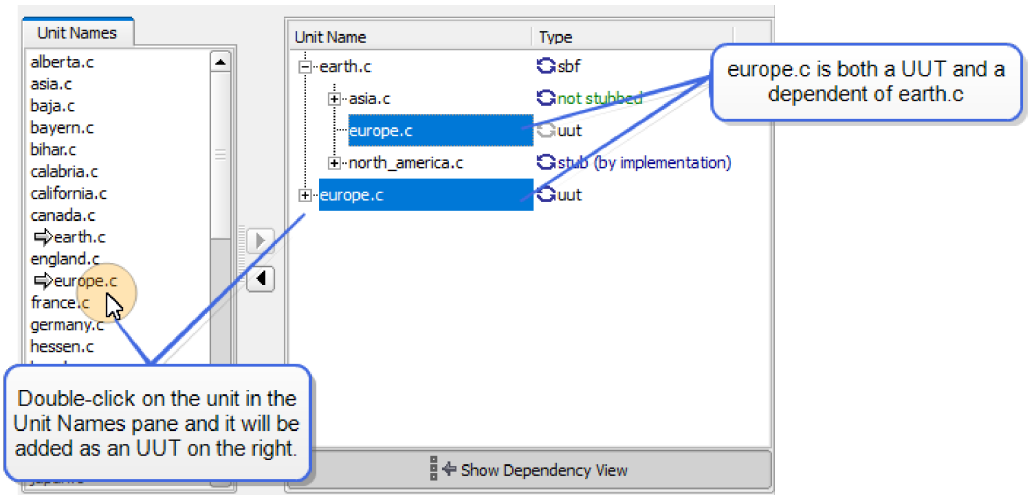


The cycle icon, when clicked, rotates through **stub (by prototype)** (default), **stub (by implementation)**, and **not stubbed**.

To specify a unit be stubbed, click the icon once until you see “stub (by implementation),” as shown below. To specify a unit should be not stubbed, click it again until you see “not stubbed.” You can cycle back to “stub (by prototype)” with one more click. Customize each dependent unit in this way, before clicking the **Build** button.



More than one unit can be made a UUT. We can make **`europe.c`** a UUT by double-clicking it in the Unit Names column on the left. If a unit is both a UUT and a dependent of another unit, then it appears in both places in the tree. In the example below, **`europe.c`** is a UUT so it appears along the left edge, and it is also a dependent of **`earth.c`**.

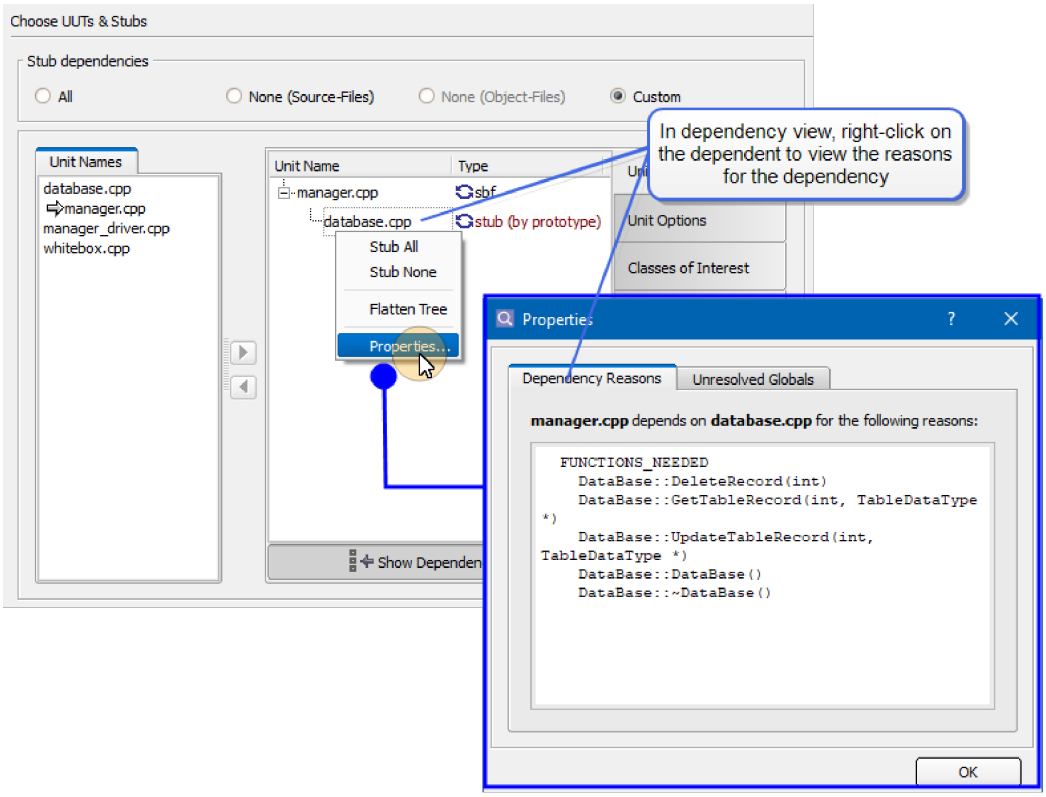


To Get Properties of a Dependent

If you have selected “Custom” in the Stub Dependencies group and you are viewing the dependency hierarchy, you can right-click the name of a dependent unit to see the reasons for dependency.

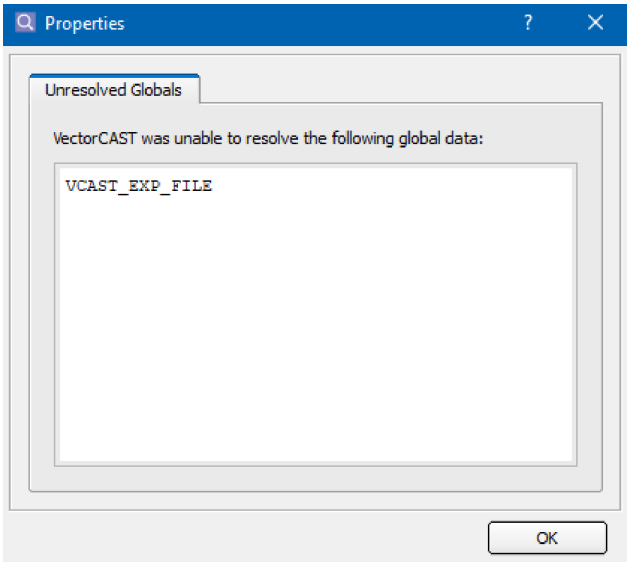
The Properties dialog lists the functions defined in the unit that are called by the unit that is above it in the dependency tree.

In the example below, the UUT **`manager.cpp`** calls several functions in dependent **`database.cpp`**. By right-clicking on **`database.cpp`** and choosing **Properties**, we see the reasons for **`manager.cpp`**'s dependency on**`database.cpp`**: the function definitions for **`DeleteRecord`**, **`GetTableRecord`**, **`UpdateTableRecord`**, and **`DataBase`** are provided by **`database.cpp`**.



If a unit is displayed in the dependency tree more than once, then the reasons for dependency refer only to the unit above it in the tree.

The Unresolved Globals tab lists the global data for which VectorCAST was unable to find the definition, and will attempt to create its definition upon environment creation. When clicking a UUT, this tab is the only tab displayed the Properties dialog. In the following example, `B0000007.cpp` had a declaration for VCAST_EXP_FILE, but no definition. VectorCAST automatically generates a definition for these entities.



To Enter Compilation Arguments for a Unit

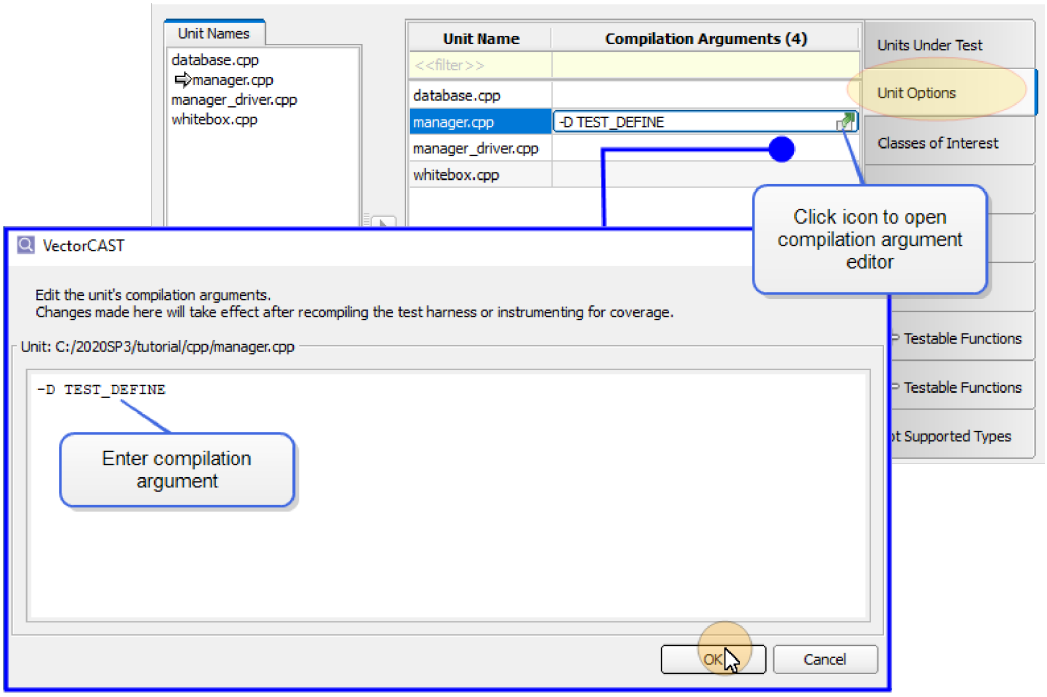
Compilation arguments for preprocessing and compiling *all* units in the environment are specified in **Step 1: Choose Compiler**, or on **the Tools => Options** dialog, **C/C++** tab.

If you want to specify a compilation argument for an individual unit, use the **Unit Options** tab in the wizard. This tab lists the units found in the Search directories on the left, and that unit's arguments on the right.

To enter a compilation argument, first select a unit name in the list. Open the text editor by clicking the icon on the right of the compilation argument field, or double-clicking on the unit name, or right-click and select **Edit...** from the context menu. Enter the command line compilation argument for that unit in the dialog and select **OK**.

Any changes made in the dialog will take effect after recompiling the test harness or instrumenting for coverage. The arguments are combined with those on the **Tools => Options** dialog**, C/C++** tab when preprocessing and compiling the unit.

Note: If your environment is built using a vcshell database, all vcdb options are treated as default unit options. Gray text indicates a default vcshell database value. These options can be modified or replaced, but they cannot be removed.



This option is specified in the environment script as:

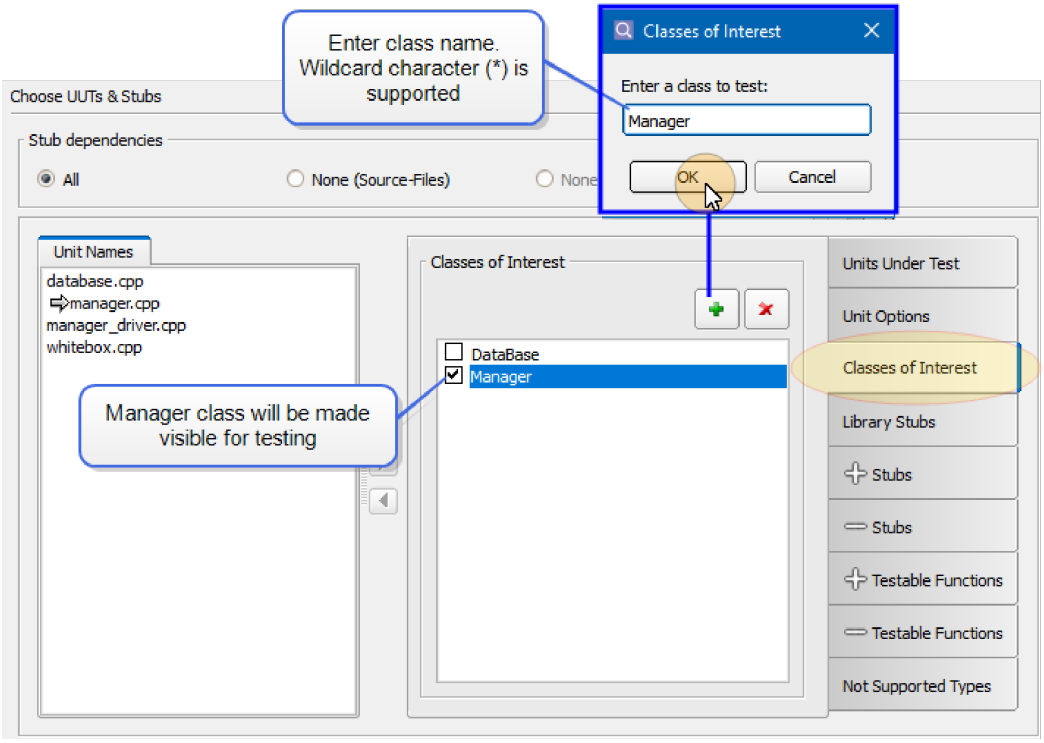
```
ENVIRO.UNIT_COMPILATION_ARGUMENTS: unit : compilation arguments
```

To Enter Classes of Interest

The Classes of Interest tab is used in C++ testing only. By default, a C++ environment is built with any classes that have a member function defined in a UUT made visible for testing. You may have additional classes that have inlined functions that are **`#included`** by a UUT.

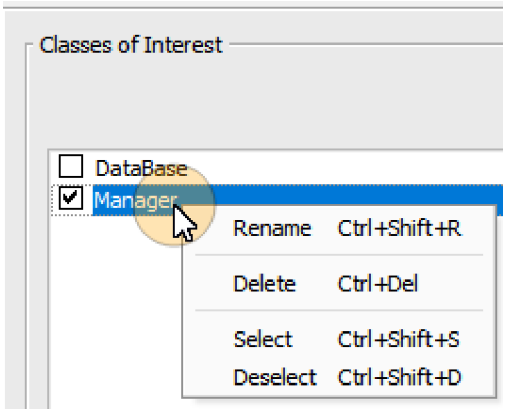
In the Classes of Interest tab, you can specify classes to override the set of classes whose member functions are considered testable. If no classes are specified, the set of testable classes is mostly defined as the set of all classes with member function defined in a UUT.

In Step 6 of the Wizard, click the **Classes of Interest** tab, and select the **Add** button . The Classes of Interest dialog opens. Enter the name of the class and select the **OK** button. The use of the wildcard character (*) is supported.



A checkmark next to a class name indicates that it will be made visible for testing when the environment is built.

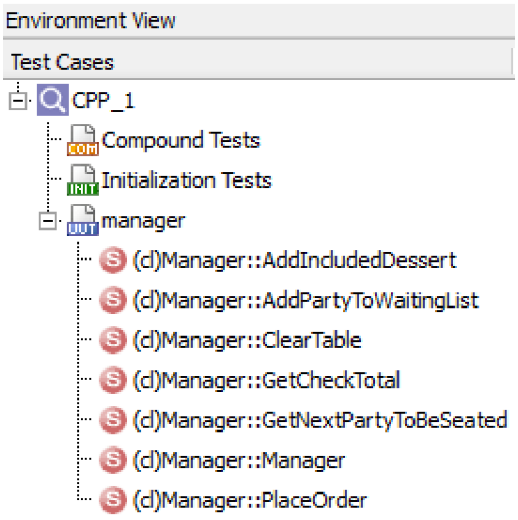
The right-click menu enables you to select or de-select all the UUT's classes, and rename or delete the selected class of interest.



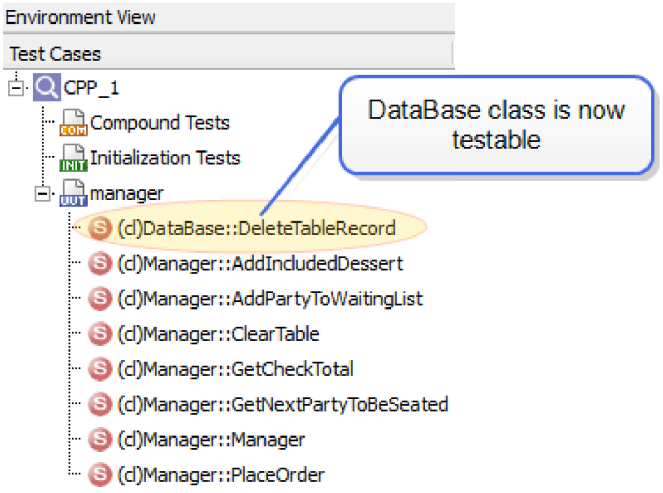
Note: If you are always interested in testing all the inlined functions, then it is easier to enable the option "Test all member inlined functions" on the **Tools => Options** dialog, **Builder** tab. With this option set, all new environments are built as though you enabled all classes in the **Classes of Interest** tab.

Once an environment is built, member functions belonging to the classes that were selected appear in the Test Case Tree, in VectorCAST's main window. The figure below shows the environment built

with only the Manager class selected.



In the figure below, both the Manager and DataBase classes were selected. Note the additional member function DeleteTableRecord in the Test Case Tree, making it testable as well.



See also "Key Terminology" on page 17 for a discussion of what makes a class testable.

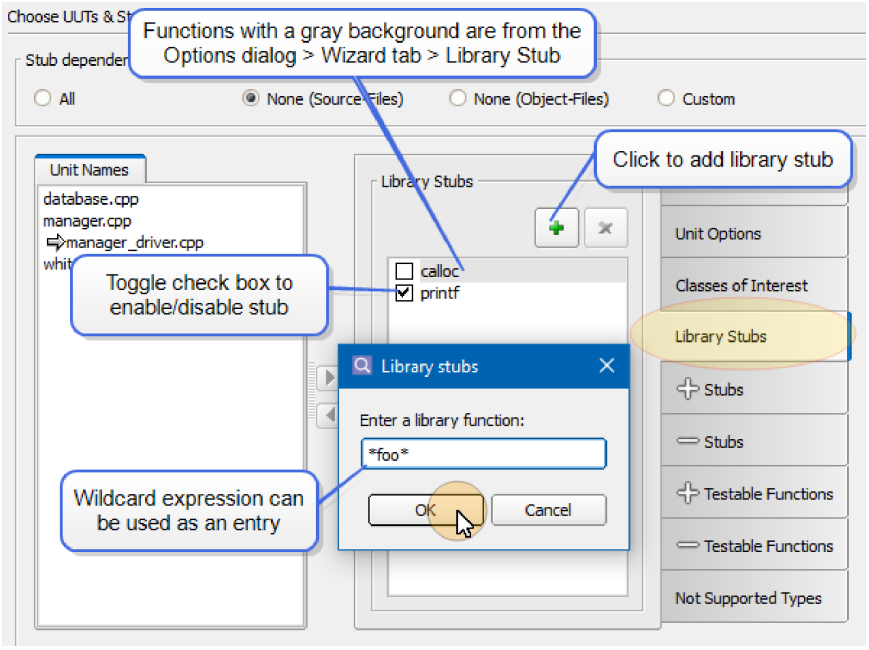
This option is specified in the environment script as:

```
ENVIRO.CLASS_OF_INTEREST: class name
```

To Stub a Library Function

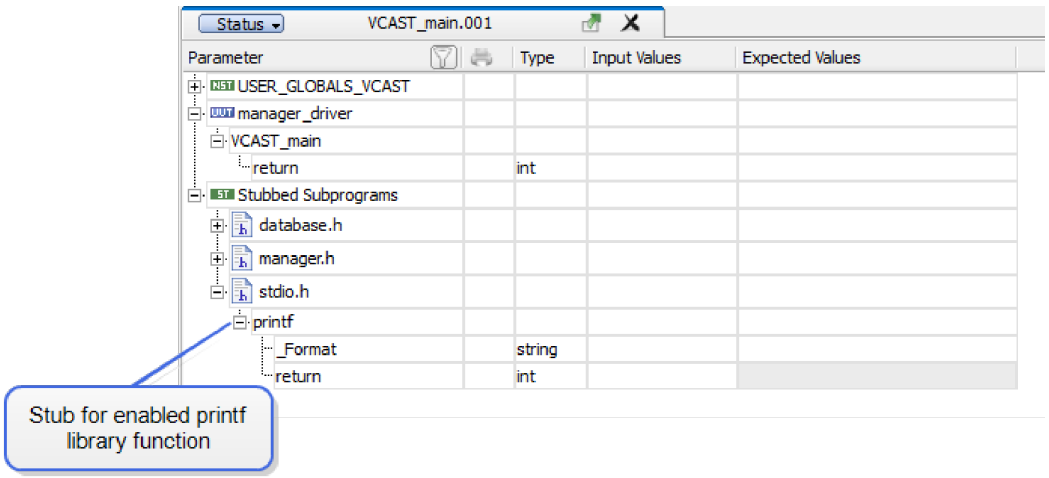
This feature is useful when you need to change behavior of a library function for testing purposes. For example, you may wish to have a call to the malloc function return a failure.

To create a stub for a library function, select the **Library Stubs** tab. Click the **Add** button, enter the function name in the Library Stubs dialog, and click **OK**. A wildcard expression (`*foo*`) can be used as an entry. The function appears in the Library Stubs list in the dialog.



If a stub name is already displayed on this screen with a gray background, the library stub was added in the **Tools => Options** dialog, **Wizard** tab. See the section "Library Stubs" on page 183 for further information.

To enable or disable any library stub, check or uncheck the box next to the name of the function. After building the environment, the Parameter Tree contains stubs for all enabled library functions referenced by the UUT. The stubs are also available to be edited from the **Environment => Configure Stubs => Edit** menu.



This option is specified in the environment script as:

```
ENVIRO.LIBRARY_STUBS:<function>
```

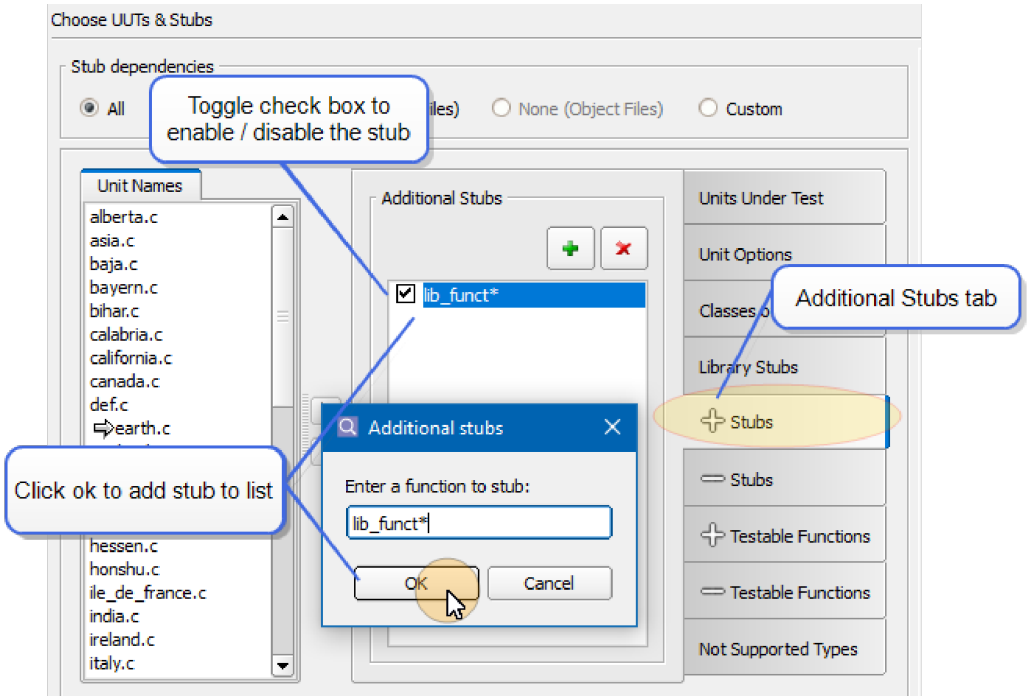
Additional Stubs

VectorCAST automatically creates a stub when a function is both referenced by a UUT *and* is located within a Search directory.

However, in a situation where this criteria is not met, for example, if you have added test case user code that calls a function not referenced by the UUT source code, and you wish to modify the behavior of that function, the Additional Stubs feature will create a stub for you.

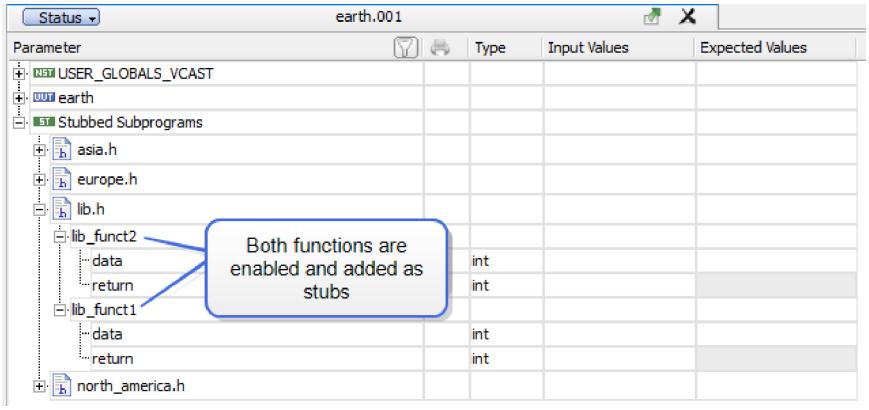
For this example, we wish to create stubs for the functions **`lib_funct1`** and **`lib_funct2`**, which are defined in a header file called **`lib.h`**.

In step 6 of the Wizard (**Choose UUTs & Stubs**), select the **Additional Stubs** tab. Click the Plus button ( ) and enter the function name in the Additional Stubs dialog, and click OK. Wild card entries are supported. For our example, we will use the wild card entry **`lib_funct*`**.



The function appears in the STUBS section of the dialog. To enable or disable any Additional stub, check or uncheck the box next to the name of the function.

After building the environment, the parameter tree will contain stubs for all enabled Additional stubs if the test case references the additional stub function. The stub is also available to be edited from the **Environment => Configure Stubs => Edit** menu.

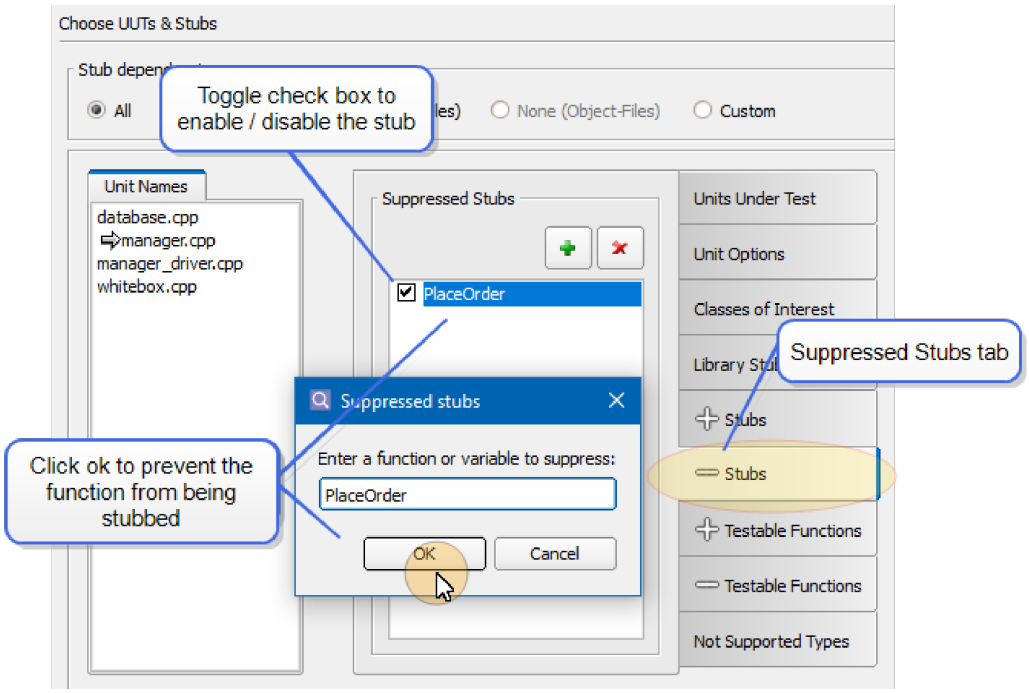


This option is specified in the environment script as:

```
ENVIRO.ADDITIONAL_STUB:<function>
```

Suppressed Stubs

If you wish to prevent VectorCAST from stubbing a function, select the Suppressed Stubs tab. Click the **Plus** button,  enter a function name in the Suppressed Stubs dialog, and click **OK**. A wildcard expression (`*foo*`) can be used as an entry. The function name appears in the STUBS section of the dialog. To enable or disable any Suppressed stub, check or uncheck the box next to the name of the function. After building the environment, a selected suppressed stub function will not appear in either the parameter tree or in the **Environment => Configure Stubs => Edit** menu.



Note: To prevent a linker error when using this option, you must provide a function definition elsewhere, i.e., via **Linker Options** located in **Step1 Choose Compiler**, **Linker / Debugger** tab.

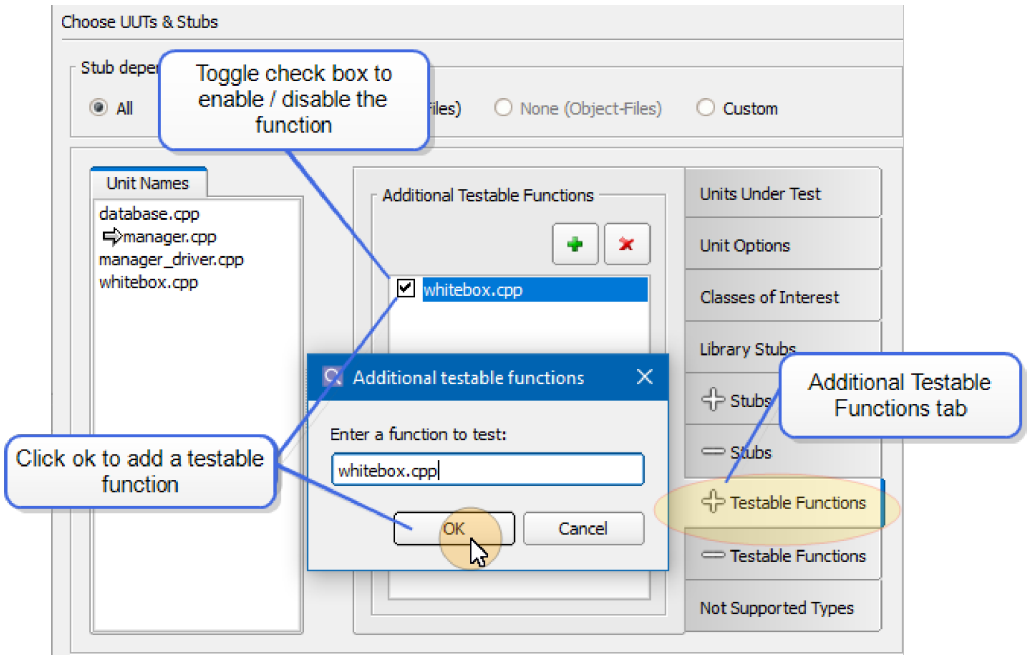
This option is specified in the environment script as:

```
ENVIRO.SUPPRESS_STUB:<function>
```

Additional Testable Functions

Test functions named here specify additional testable functions, even if they would not normally be testable. Functions named here are testable even if they are also listed under Suppressed Testable Functions. Note that at least a function prototype for the additional function is required to exist in the source.

To add a testable function, select the **Additional Testable Functions** tab. Click the **Plus** button, enter a function name in the Additional Testable Functions dialog, and click **OK**. A wildcard expression (**`*foo*`**) can be used as an entry. To enable or disable any additional function, check or uncheck the box next to the name of the function. After building the environment, a selected additional testable function will appear in the parameter tree.



This option is specified in the environment script as:

```
ENVIRO.ADDITIONAL_TESTABLE_FUNCTION:<function-name>
```

This command is useful in the situation where the configuration options **`VCAST_TEST_ALL_INLINES`** and /or **`VCAST_TEST_ALL_NON_MEMBER_INLINES`** are set to false. Setting **`VCAST_TEST_ALL_INLINES`** to false prevents testing of inline member functions in header files. Setting

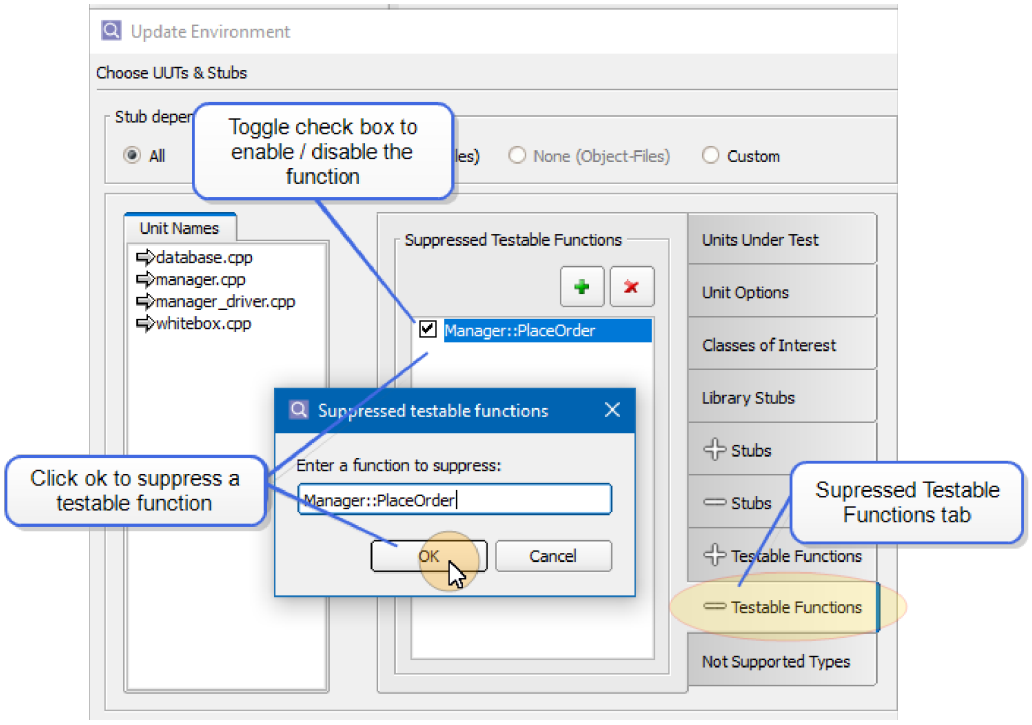
VCAST_TEST_ALL_NON_MEMBER_INLINES to false prevents testing of non-member inline functions in header files.

The **ENVIRO.ADDITIONAL_TESTABLE_FUNCTION** command takes precedence over the **ENVIRO.SUPPRESS_TESTABLE_FUNCTION** command. If **ENVIRO.ADDITIONAL_TESTABLE_FUNCTION** is specified for a function, then it is testable even if **ENVIRO.SUPPRESS_TESTABLE_FUNCTION** is also specified for that function.

Suppressed Testable Functions

You can prohibit VectorCAST from making a function in a UUT testable by using the **Suppress Testable Functions** tab in the Create New Environment Wizard. Functions named here do not appear in either the Test Case Tree or in the <<SBF>> node of the Parameter Tree.

To suppress a testable function, select the **Suppressed Testable Functions** tab. Click the **Plus** button, enter a function name in the Suppressed Testable Functions dialog, and click **OK**. A wildcard expression (***foo***) can be used as an entry. To enable or disable any Suppressed function, check or uncheck the box next to the name of the function. After building the environment, a selected suppressed testable function will not appear in the parameter tree.



This option is specified in the environment script as:

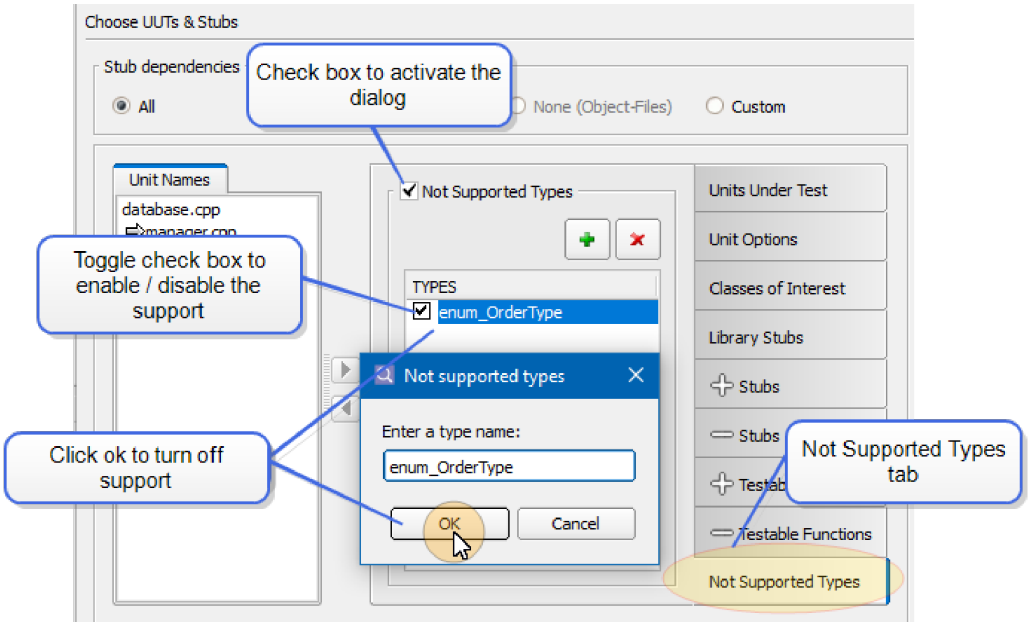
```
ENVIRO.SUPPRESS_TESTABLE_FUNCTION: <function>
```

To Turn Off Support for a Data Type

When instrumenting a very large system, you may wish to limit the size of the generated test harness.

By suppressing support for a specific data type, an enum for example, the overall size of the generated test harness is reduced.

Select the **Not Supported Types** tab. Check the **Not Supported Types** checkbox to activate the dialog. Click the **Plus** button ( ), enter a type name in the Not supported types dialog, and click **OK**. The type name appears in the TYPES section of the dialog. To enable or disable any Not supported type, check or uncheck the box next to the name of the type.

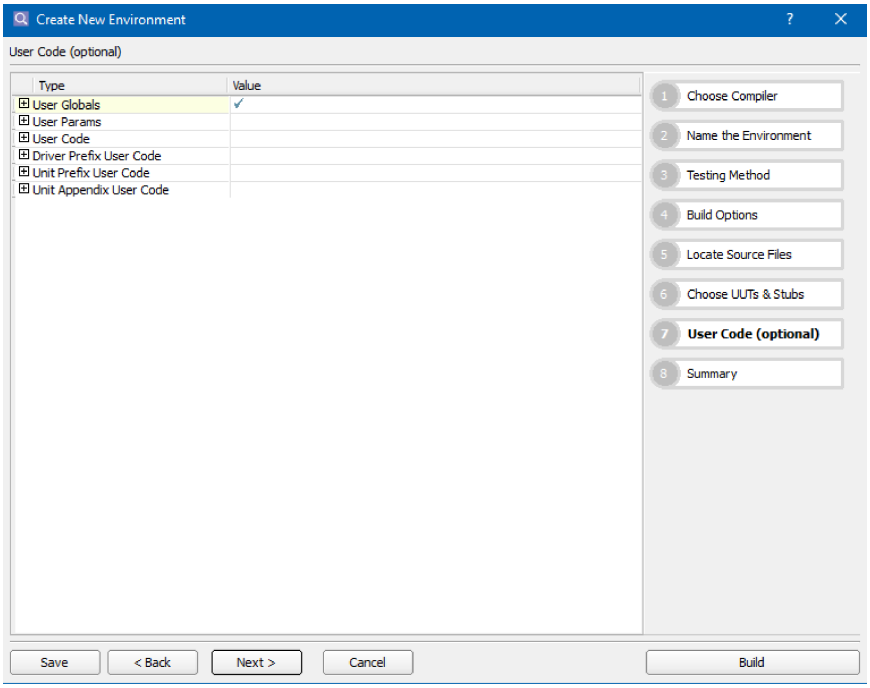


Step 7: User Code (Optional)

Step 7 is used to optionally specify environment user code, user globals, user parameters, driver prefix user code, unit prefix user code and unit appendix user code. If you have loaded in an environment script having any of these types of user code defined, then you can view and modify the user code here.

On this page you can perform the following, if desired:

- Add environment user code
- Create your own global variables
- Create an object to represent a parameter
- Insert user code to be appended to a UUT
- Insert user code to be prepended to the beginning of the test harness driver for the UUT



This section briefly describes the types of environment user code. For a complete description, see "User Code" on page 595.

User Globals - User Globals provide a mechanism for user-defined types and objects to be included in the test harness. By default, there are five integer objects, five floating point objects, five string objects, and an array of 200 integer elements. All data objects that are defined in User Globals when the environment is built can be manipulated as test data when building a test case.

Preface any variables defined here with VCAST_USER_GLOBALS_EXTERN to ensure that only one definition of the variable is created in the test harness. They will be listed in the Parameter Tree under USER_GLOBALS_VCAST.

User Params - User Params are used to instruct VectorCAST to use a user-defined object for a parameter, rather than a VectorCAST-generated object. User Params are used to provide the association between the user-defined parameter and the subprogram's parameter. A syntax example is:

```
<<manager.Place_Order.Table>> VECTORCAST_INIT1
```

User Code - User Code is best suited for operations that relate to the harness as a whole; loading data from a file or initializing a database unit are two examples. It enables you to write source code to handle some of the harness tasks that are not easily accomplished with static data. With user code, you can write code to read data from a file, call initialization routines, or assign and verify data objects based on dynamic criteria.

Driver Prefix User Code - Driver Prefix User Code is user code that is appended to the beginning of the driver code for the specified UUT (and is therefore added to the test harness). It is not parsed or preprocessed during header expansion. Driver Prefix User Code is useful for **`#undef`**-ining something that should not be **`#defined`** in the test harness, as well as many other purposes.

Unit Prefix User Code - Unit Prefix User Code is user code that is placed at the beginning of the

specified UUT (and is therefore added to the test harness). Because Unit Prefix User Code is considered part of the specified UUT, it is useful for **`#undef`**-ining something that should not be **`#defined`** in the UUT, as well as many other purposes.

Unit Appendix User Code - Unit Appendix User Code is user code that is appended to the end of the specified UUT (and therefore added to the test harness). Because Unit Appendix User Code is considered part of the specified UUT, it is useful for **`#including`** a concrete subclass for an abstract class, as well as many other purposes.

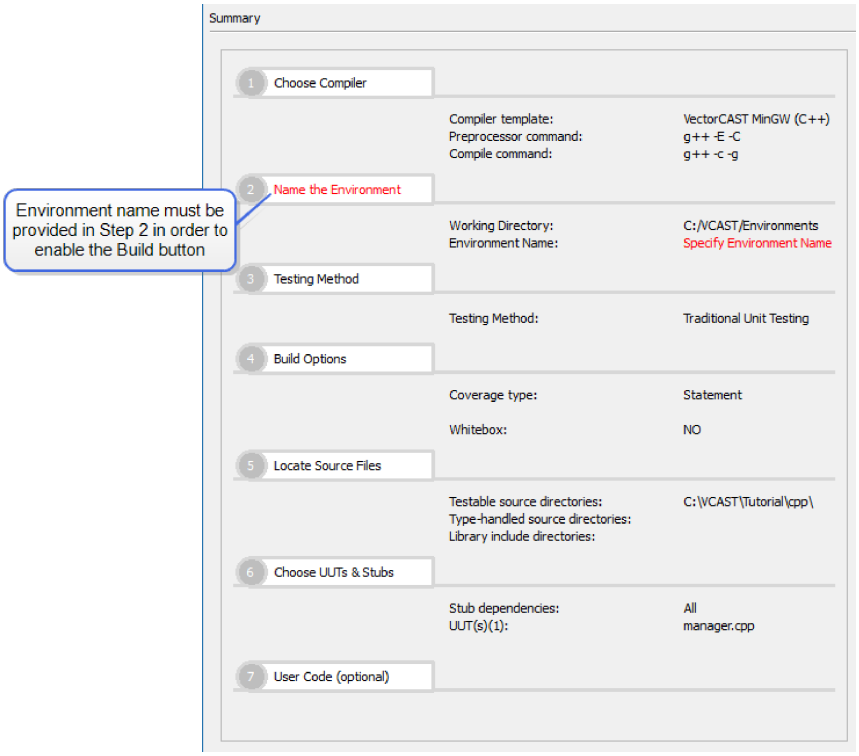
You have access to all types of environment user code after the environment is built by choosing **Environment => User Code => Edit.**

See also "Environment Script Language" on page 661.

If you are not familiar with Environment User Code, see "Types of Environment User Code".

Step 8: Summary

This page of the wizard, called Summary, displays a summary of your settings. If there is any missing information, the problem is displayed in red text. In particular, it checks that the compiler and preprocessor are on the system PATH and that there are no empty fields. Missing information must be supplied before the **Build** button is enabled.



To Build the Environment

Once you have completed data entry in the Create New Environment wizard, click the **Build** button. The environment builder determines if the source for the UUT has any compile errors, and determines the dependencies.

If there is an error, a dialog is presented with a description of the error and steps required to resolve the error. Once errors are resolved, VectorCAST proceeds with environment creation.

See "Troubleshooting Environment Creation" on page 160 for more information.

```
clicast -lc ENvironment Build <scriptfile>

Build an environment from <scriptfile>. The environment script
<scriptfile> must be in the current working directory; it cannot be a full
path or a relative path to a file.

    You can also use *.env for <scriptfile>. CLICAST is passed a command
    line with each <filename>.env found in the directory. On Linux, to
    avoid having the command line get too long after expansion, you can
    quote the argument to CLICAST, using "*.env" for <scriptfile>. This
    form passes CLICAST the actual wildcard string for it to expand.
```

If the compiling or linking of the test harness component fails, the diagnostic messages for the compiler are displayed, and the Message Window displays “Environment Built but not compiled”, or “Environment Built but not linked” or “Environment Built but run-time errors exist.” After correcting the errors, choose **Environment => Rebuild Environment**.

To Build an Environment for Object File Testing

Object File Testing allows you to create tests from previously compiled object modules. VectorCAST works in the same manner as for traditional Unit Testing, except that the original source files are not compiled into the test harness. The user provides the path to the object files or the library file containing the object files for all units under test.

To illustrate this feature, an object file for the source file shown below is created using an external compiler. The object file is linked into the test harness at build time.

```
/* FourFunctions.c

    This file contains code for the four
    arithmetic functions – add, subtract,
    multiply and divide
*/

int add(int a, int b)
{
   return a+b ;
}

int sub(int a, int b)
{
   return a-b ;
}

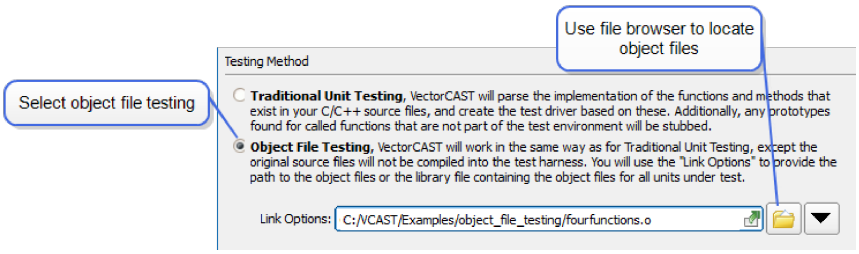
int mul(int a, int b)
{
```

```
    int i = 0 ;
    int result = 0 ;

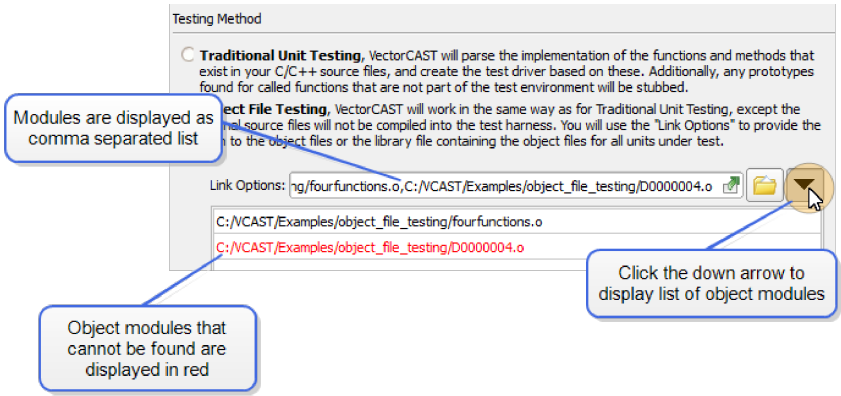
    for( i = 0 ; i < a ; i++)
    {
       result += b ;
    }
    return result ;
}

int divd(int a, int b)
{
    return a/b ;
}
```

1. Create a new environment using the Environment Wizard. In step 3, Testing Method, select Object File Testing. Use the file browser to locate previously compiled object modules or library archives. In this example, the name of the object module is FourFunctions.o.



Object modules are displayed as a comma separated list in the **Link Options** text box. To edit the list, click the **down arrow** next to the file browser button to display a list of object modules. If an object module cannot be found, its path is indicated in red.

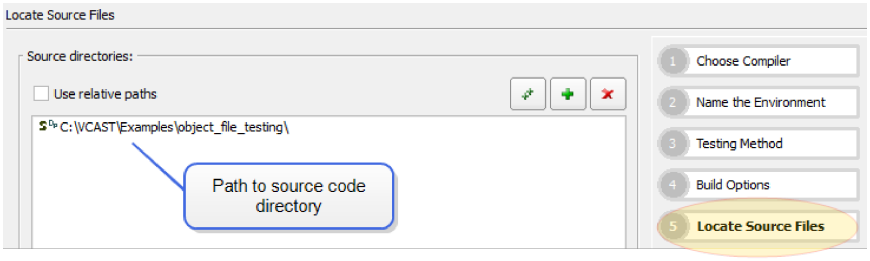


To edit an entry in the list, double-click the entry and enter the changes. To remove an entry in the list, right-click the entry and select **Delete** from the context menu. If you wish to use relative paths, the path is specified relative to the environment directory. At build time, the object module

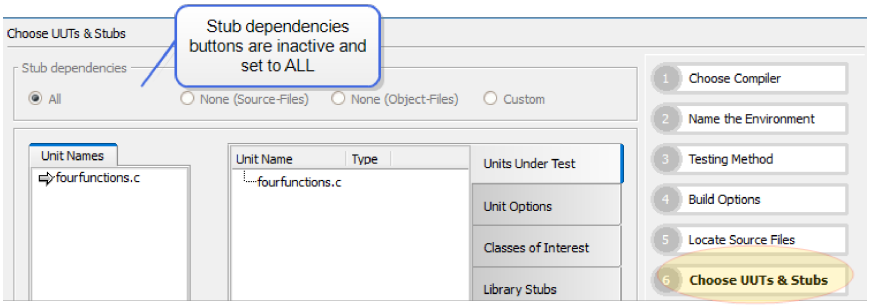
or library archive is linked into the test harness.

Note: If you select Code Coverage in step 4, VectorCAST will NOT use the previously compiled object file when executing the instrumented harness.

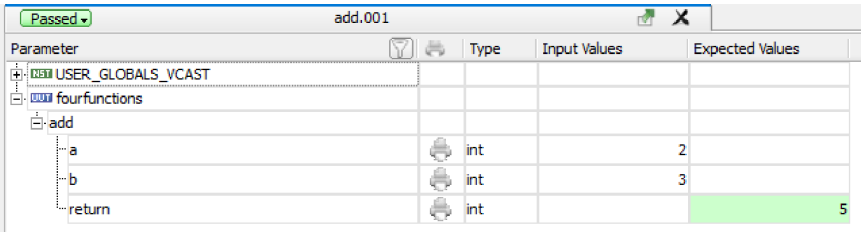
2. In step 5 of the Wizard, Locate Source Files, add a path to the directory containing the source code for the object module you wish to test.



3. In step 6 of the Wizard, Choose UUTs and Stubs and add the unit to Units Under Test. Notice that the radio buttons in the Stub dependencies panel are inactive and that the UUT Type is not user selectable. For Object File Testing, the stub dependencies selection is automatically set to **ALL** and the UUT Type is automatically set to UUT. Click **Build,** and a test harness is created using the specified previously compiled object module.



4. Add test cases for the desired subprograms. We will add a test case for the add function. In the parameter tree, we add Input Values and an Expected return value. When the test is executed, the linked object module code is executed and the test results are shown in the Parameter Tree.



Looking at the VectorCAST environment file OBJECT_FILE_TESTING.env, the link options path ENVIRO.LINK_OPTIONS, is set to the path and file name specified in step 3 of the Wizard as shown above.

```
ENVIRO.NEW
ENVIRO.NAME: OBJ_FILE
ENVIRO.LINK_OPTIONS: C:/VCAST/Examples/object_file_testing/fourfunctions.o
ENVIRO.OBJECT_FILE_UUT: fourfunctions
ENVIRO.COVERAGE_TYPE: NONE
ENVIRO.LIBRARY_STUBS:
ENVIRO.STUB: ALL_BY_PROTOTYPE
ENVIRO.COMPILER: CC
ENVIRO.TYPE_HANDLED_DIRS_ALLOWED:
ENVIRO.SEARCH_LIST: C:\VCAST\Examples\object_file_testing
ENVIRO.END
```

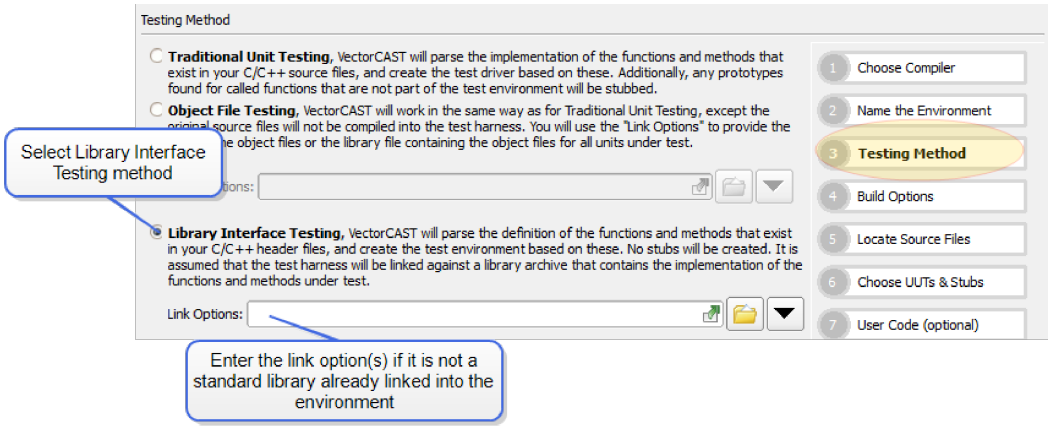
To Build an Environment for Library Interface Testing

Library Interface Testing allows you to create tests for an existing library or DLL without having the source code available. The library header file is all that is needed to create the test environment.

A common use of this feature is to allow testing of a third-party library that is delivered to as a object file library, and an API definition (header files). You can create test cases for the API and validate the correctness of the library functions for your application, without needing access to the source code.

In this example, we will build test cases for the string library as defined in string.h.

1. Create a new environment using the Environment Wizard. In Step 3, Testing Method, select Library Interface Testing.



For this example, the string.h is part of the standard C library and it is already linked into the environment.

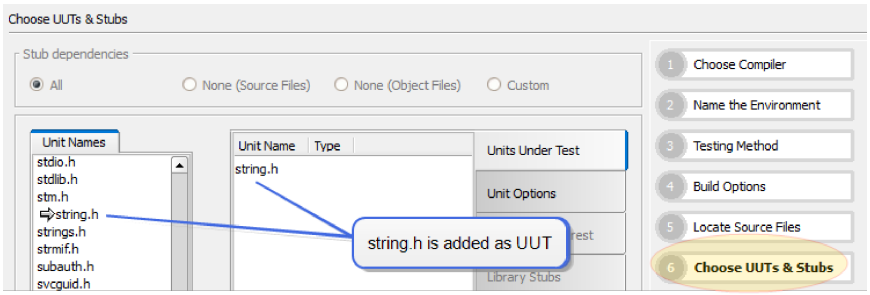
To link custom libraries to the environment, use the file browser next to the Link Options text box to locate the library modules. If multiple library modules are selected as Link Options, they are displayed as a comma separated list in the **Link Options** text box. To edit the list, click the

down arrow next to the file browser button to display a list of object modules. If an object module cannot be found, its path is indicated in red. To edit an entry in the list, double-click the entry and enter the changes. To remove an entry in the list, right-click the entry and select **Delete** from the context menu. If you wish to use relative paths, the path is specified relative to the environment directory. At build time, all specified library modules are linked into the test harness.

In Step 5 of the Wizard, Locate Source Files, add a path to the library's header file.



2. In Step 6 of the Wizard, Choose UUTs and Stubs, add the header files for the units you wish to test. When you have completed choosing the units, click **Build**. The environment will be built and linked to the library.

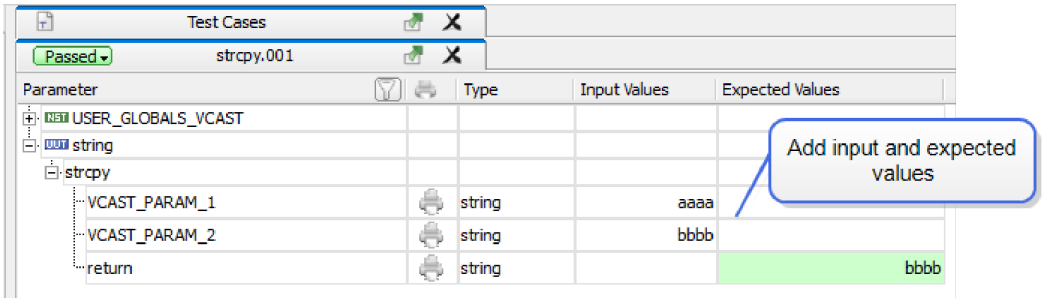


3. Add test cases for the desired subprograms. We will add a test case for the **`strcpy`** function. The header prototype for this function is as follows:

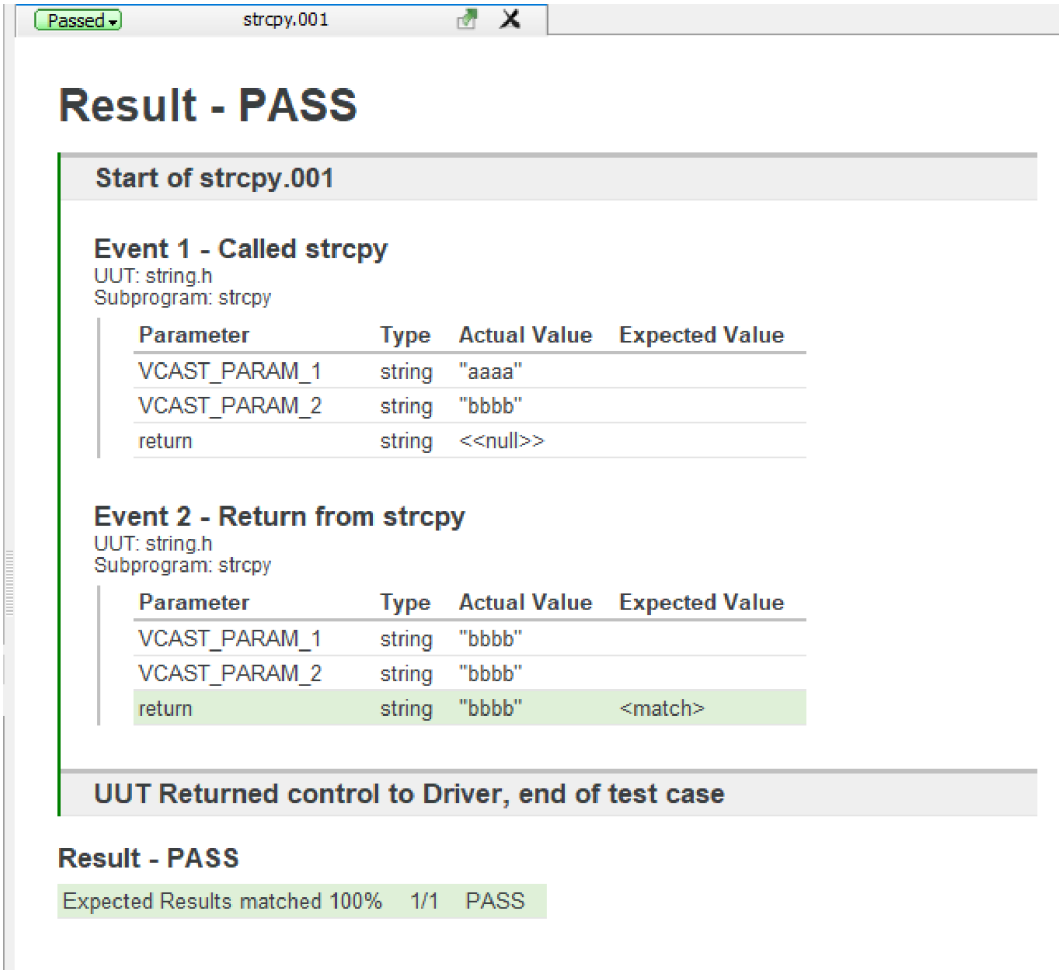
```
char *strcpy (char *, const char *);
```

This function copies the **`_Source`** string to the **`_Dest`** string, and returns the **`_Dest`** string.

4. In the parameter tree, we add Input Values and the Expected return value.



5. When we execute the test, the linked library code is executed and the Execution Report reflects success.



To Build an Environment for Test Driven Development

With Test Driven Development (TDD) the test cases are developed as soon as the function prototypes are generated (header files), and before any function implementations are generated (.c and .cpp files). When using the TDD feature in VectorCAST, you will choose one of more header files

as the units under test. VectorCAST will automatically create empty function place holders for the functions defined in the header files, which will allow you to build and execute tests. Once some of the function implementations are developed, you can simply rebuild the test environment, and VectorCAST will add the "real code" that it finds in the .c/.cpp file into the test environment. In this way, you can incrementally build the function implementations for your unit.

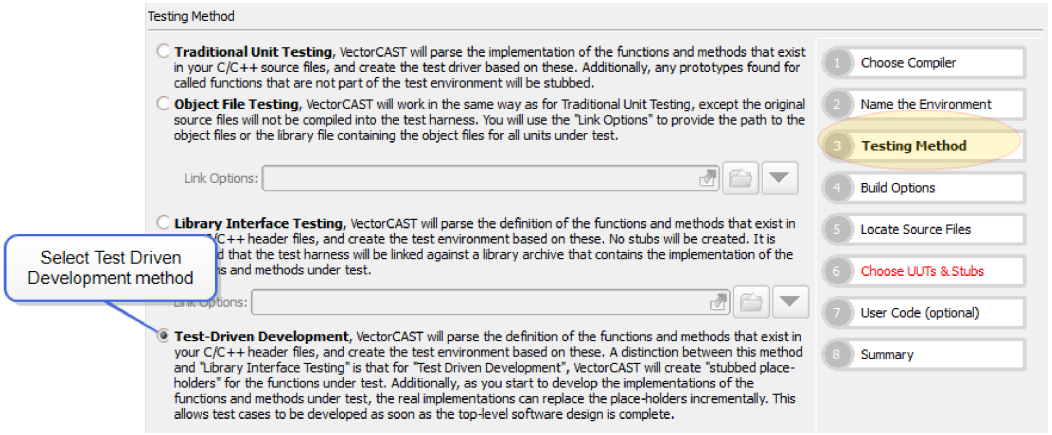
The following header file, **`subSystem2.h`**, will be used to demonstrate creating a Test Driven Development environment:

```
extern int globalDataProvidedBySubSystem1;

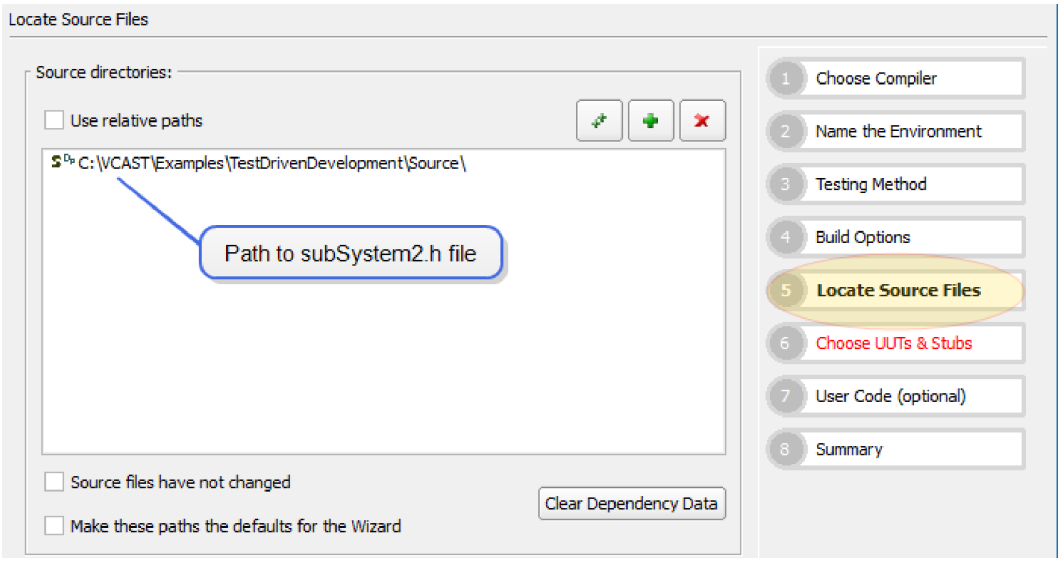
void providerOfSS2data (int setValue);
int consumerOfSS1data (int command);
int doSomeS2Stuff (int state);
int callS1Stuff (int state);
```

The implementation file is not available, but using only the header we can begin designing our test cases.

1. Create a new environment using the Environment Wizard. In step 3, choose the Testing Method **Test Driven Development**.

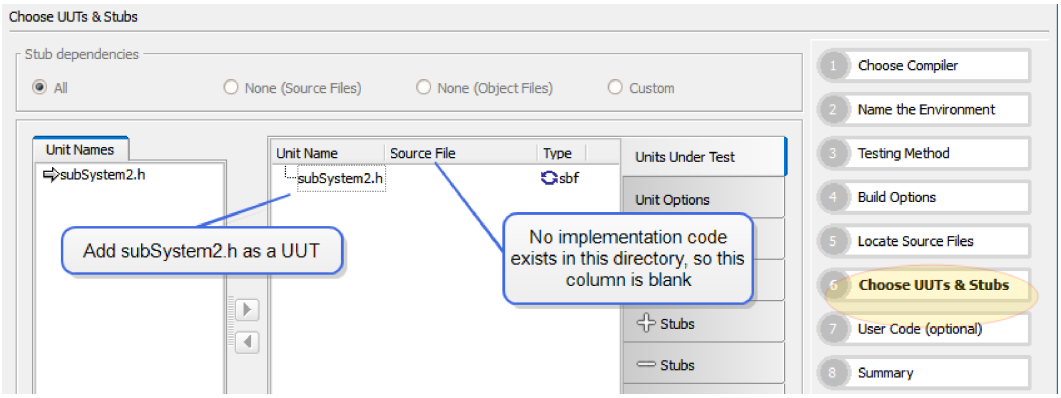


2. In step 5 of the Wizard, specify the path to the header file under test and all of its dependencies.

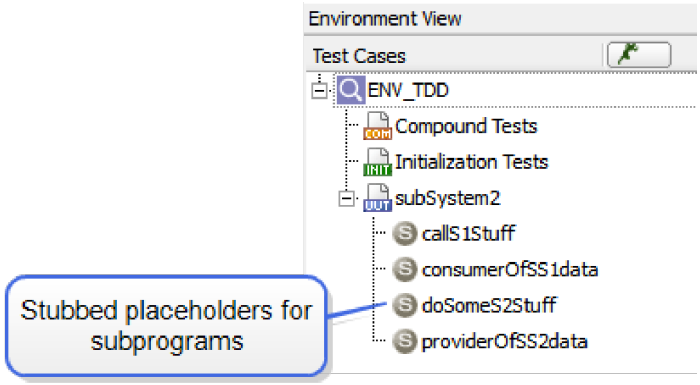


3. In step 6 of the Wizard, add the header file as a UUT. For the purposes of our example below, our implementation code resides in a different directory and therefore our Source File column is empty.

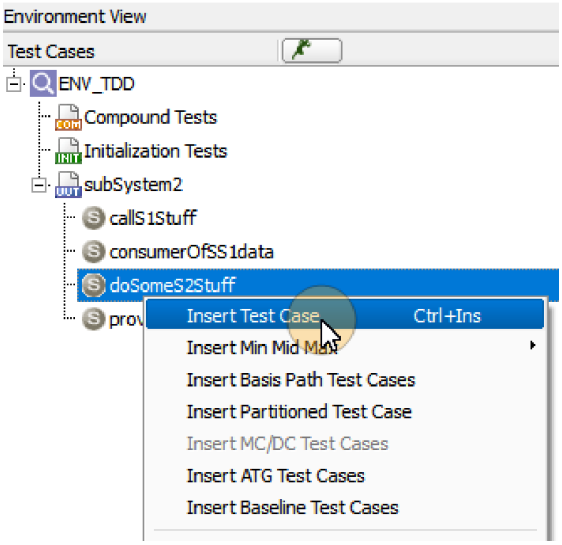
   Note that if a source file is listed in the Source File column, with a checkbox next to the file name, then implementation code has been located by VectorCAST. By checking the box, the implementation code is added to the test environment after **Build** is clicked. If the source file contains partial implementation of the interfaces described in the header file, the implemented methods will be built as standard VectorCAST subprograms, and the unimplemented methods will be built as placeholder stub subprograms. See item 9 below.



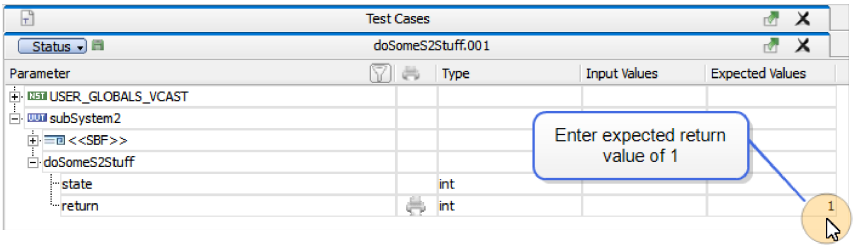
4. Build the environment.
5. VectorCAST creates stubbed place holders for each subprogram specified in the header file. This is indicated with a gray subprogram icon in the Test Case Tree. Note that if a constructor is specified in a class definition, VectorCAST automatically generates a call to the default constructor and a placeholder stub is created for it.



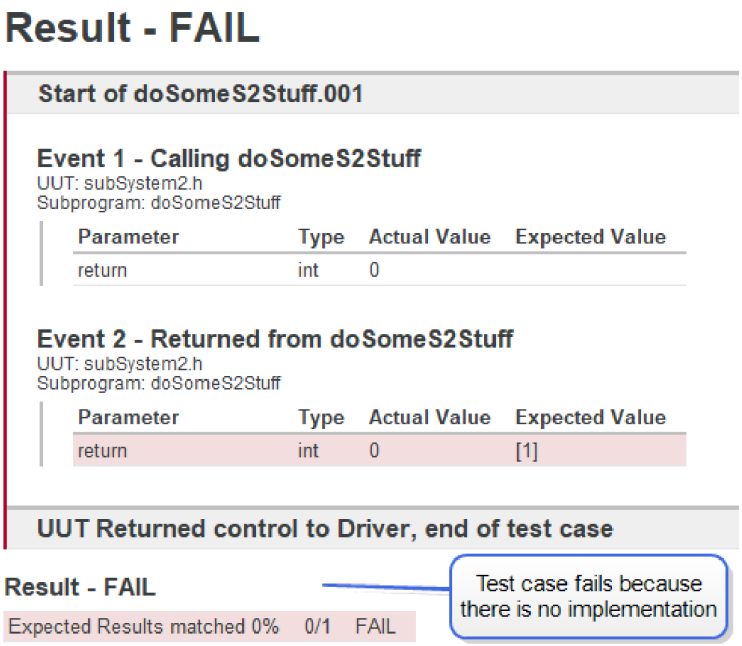
6. We are now ready to begin specifying test cases. Right-click any of the place holder subprograms and select **Insert Test Case**.



7. In the Parameter Tree, enter test data for each test case that is created. By creating test cases using only the interface definitions, the test cases are not dependent upon the underlying implementation. The example below creates a test case for **`doSomeS2Stuff`**, specifying an expected return value of 1.



8. Note that if the test case is executed at this point, we expect the test case to fail. This is because there is no implementation available for it yet.



The following steps show how to add an implementation file to the environment. Once added, the test case can be re-run to verify that the test case passes.

Here is source code for the partial implementation of **`subSystem2`**. It implements the **`doSomeS2Stuff`** and **`callS1Stuff`** functions but **`providerOfSS2data`** and **`consumerOfSS1data`** are not yet available.

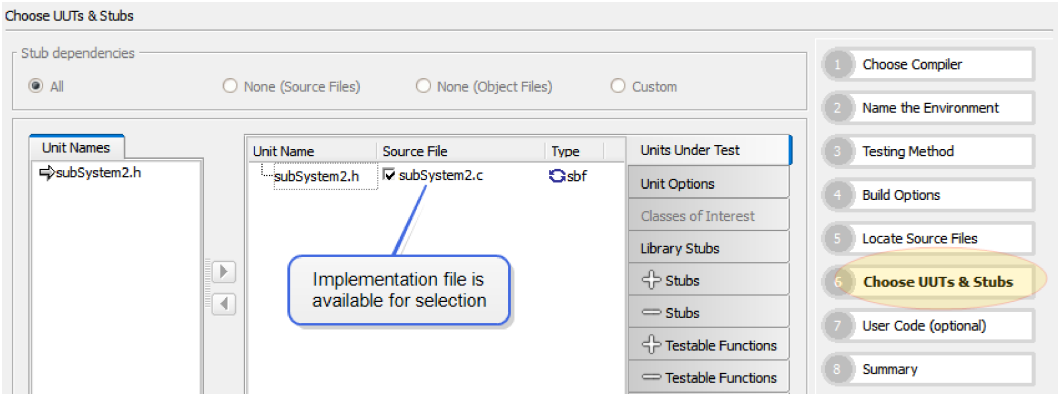
```
#include "subSystem2.h"

int globalDataProvidedBySubSystem2;

int doSomeS2Stuff (int state) {
    if (state == 1) return 222222;
    else return 1;
}

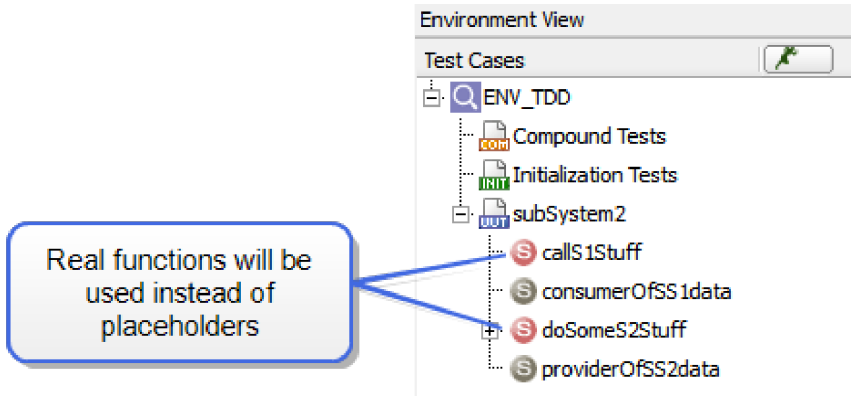
int callS1Stuff (int state) {
    globalDataProvidedBySubSystem1++;
    if (state == 1) return doSomeS2Stuff (1);
    else return doSomeS2Stuff (0);
}
```

9. Add the implementation file, **`subSystem2.c`** to the test environment by selecting **Environment => Update**. In step 5 **Locate Source Files**, add the path to the .c file. If source files are located across several different directories, add the path to each directory in step 5. Next, go to step 6, **Choose UUTs and Stubs**.

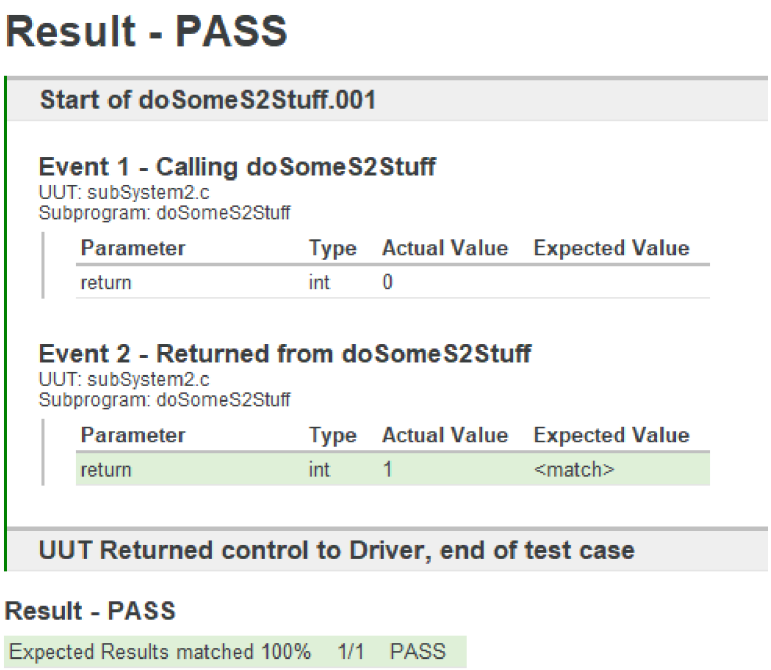


Notice there is a checkbox next to **`subSystem2.c`** in the Source File column in the Units Under Test tab. Click on the checkbox to select it, and then Update the environment.

When the update completes, notice that the subprogram icons in the Test Case tree for **`callS1Stuff`** and **`doSomeS2Stuff`** are now red, indicating that they are no longer stubbed place holders and the real functions will be used.



10. Execute the test for **`doSomeS2Stuff`** created above. The execution report indicates that the test passes.



To View the Environment Overview Report

The Overview report is a configuration report on the environment. To open the Environment Overview Report, select **Environment => View => Overview** from the Menu Bar. The report contains the following items:

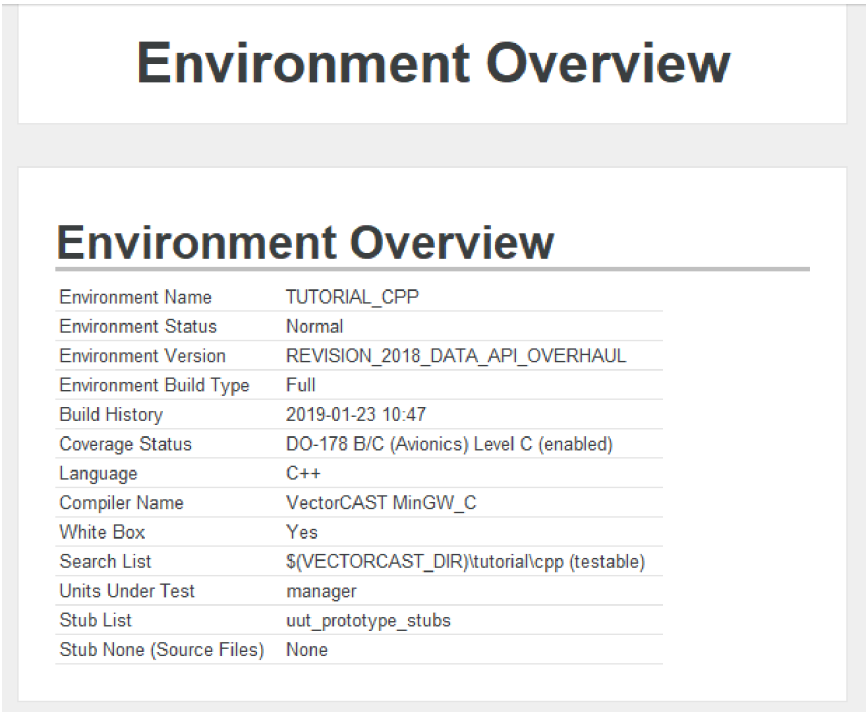
- **Environment Name** – the name of the environment, the VectorCAST internal version number, and the environment status.
- **Environment Status** – Normal, Compile Error, or Link Error.
- **Environment Version** – the version of the VectorCAST environment builder. For internal use.

- **Environment Build Type** – Full, Incremental, Incremental Rebuild Failed
- **Build History**: –The original build entry is listed at the top of the list, followed by later builds.
- **Coverage Status** –The type of coverage initialized, and whether it is enabled or disabled.
- **Language** – the source code language: Ada, C, C++.
- **Compiler Name** – the name of the compiler used to create the test harness.
- **White Box**: Yes – this line is present in the report only if the environment was created with Whitebox checked.
- **Search List** – one or more absolute or relative paths to directories in the Source directories list, used to find source files.
- **Units Under Test** – the name(s) of the unit(s) under test.
- **Stub List** – the names of the units that were stubbed, if any. If units were stubbed by prototype, then the unit name is listed as “uut_prototype_stubs”.
- **Stub None** – If the stub type Stub None (Source Files) is used, a list of the not stubbed units is displayed. If the stub type Stub None (Object Files) is used, a list of all the dependent object files is displayed.
- **Additional Environment Information** – This line is not present unless the Builder option 'multiunit whitebox' is turned on.

```
clicast -e <env> ENvironment Extract Overview [<outputfile>]

Extract environment overview report to standard output or to a file if
specified. If VCAST_CUSTOM_REPORT_FORMAT is HTML, the standard output is
saved to <env>_overview_report.html.
```

A sample Environment Overview report is shown below.



Environment Overview

Environment Overview

Environment Name	TUTORIAL_CPP
Environment Status	Normal
Environment Version	REVISION_2018_DATA_API_OVERHAUL
Environment Build Type	Full
Build History	2019-01-23 10:47
Coverage Status	DO-178 B/C (Avionics) Level C (enabled)
Language	C++
Compiler Name	VectorCAST MinGW_C
White Box	Yes
Search List	$(VECTORCAST_DIR)\tutorial\cpp (testable)
Units Under Test	manager
Stub List	uut_prototype_stubs
Stub None (Source Files)	None

A sample Environment Overview report in Text format is shown below.

```
------------------------------------------------------------
Environment Overview
------------------------------------------------------------
  Environment Name:    TUTORIAL_CPP
  Environment Status:  Normal
  Environment Version: REVISION_2018_DATA_API_OVERHAUL
  Environment Build Type: Full
  Build History:       2019-01-23 10:47
  Coverage Status:     Statement (enabled)
  Language:            C++
  Compiler Name:       VectorCAST MinGW_C++
  White Box:           Yes
  Search List:         $(VECTORCAST_DIR)\tutorial\cpp (testable)
  Units Under Test:    manager
  Stub List:           uut_prototype_stubs
  Stub None (Source-Files): None
```

Files Created by the Environment

The following table lists some of the files that are contained in the working directory. Under normal conditions, you will not need to open or edit these files.

Note: These files do not need to be maintained for configuration management or version management. The **Environment => Create Regression Scripts** command creates the three files that need to be archived for each environment (unless you have imported coverage results, which adds one more).

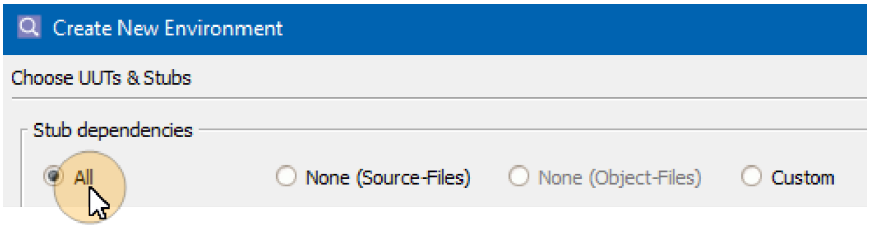
File name	Description
Env.vce (where *Env* is the name you gave the environment.)	The VectorCAST environment file. In Windows, double-clicking this file or its icon launches VectorCAST and opens the environment.
Env.env	The environment script file.
CCAST_.CFG	Configuration file created and read by the **Tools => Options dialog box**.
VCAST.QIK	Quickparse file. This file is located in each Search Directory.
A directory named *ENV*	The environment directory, which contains the test harness source files, the harness executable, user code files, and environment data files. None of the files in the environment directory should be modified or opened by the user! If they are modified, unpredictable tool performance and/or data loss could result.

Troubleshooting Environment Creation

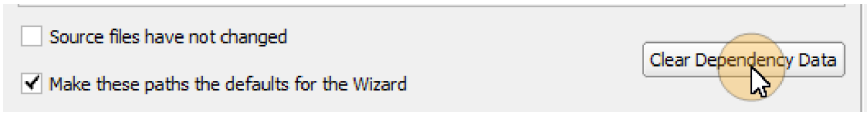
The QuickParse File

When VectorCAST builds an environment, it needs to determine which files are called by the Unit(s) Under Test, and which files are called by those files, and so on. To determine this information, VectorCAST invokes a utility called QuickParse to perform a simplistic parsing of the source files. When QuickParse is called for the first time in a specific directory, it parses all C/C++ files in that directory and creates a data file to store a list of functions defined and called, and data objects defined and called. This file is stored in the parsed directory as VCAST.QIK (if write-permission is granted) as well as in the environment directory. By saving the file in the parsed directory, future invocations of QuickParse can use the existing VCAST.QIK file and parse only the source files that have been modified since the last quick-parse rather than re-parsing the directory every time an environment is built.

To create an environment using only the prototype definitions, thus avoiding quick-parsing, select the **All** radio button in the Stub Dependencies pane of the Choose UUTs & Stubs window of the Create New Environment wizard.

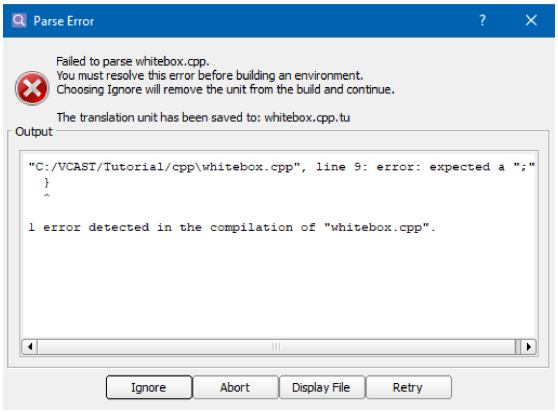


To delete this file prior to building a new environment, click the **Clear Dependency Data** button in the Locate Source Files window of the Create New Environment wizard.



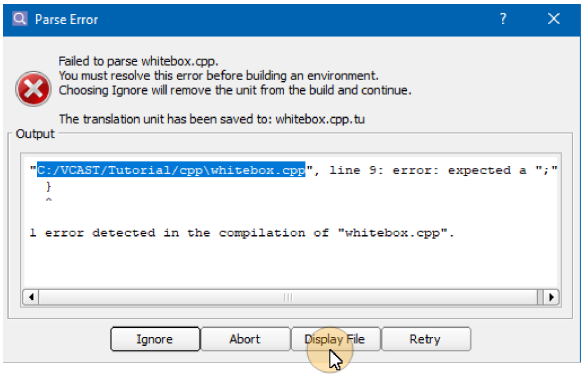
QuickParse Failures

If there are source files unrelated to your project in any of the directories in the Source directories list, it is possible that the Language Mode or compile options may not be appropriate for those files. In that case, QuickParse will fail, and VectorCAST shows an informational dialog similar to the following.

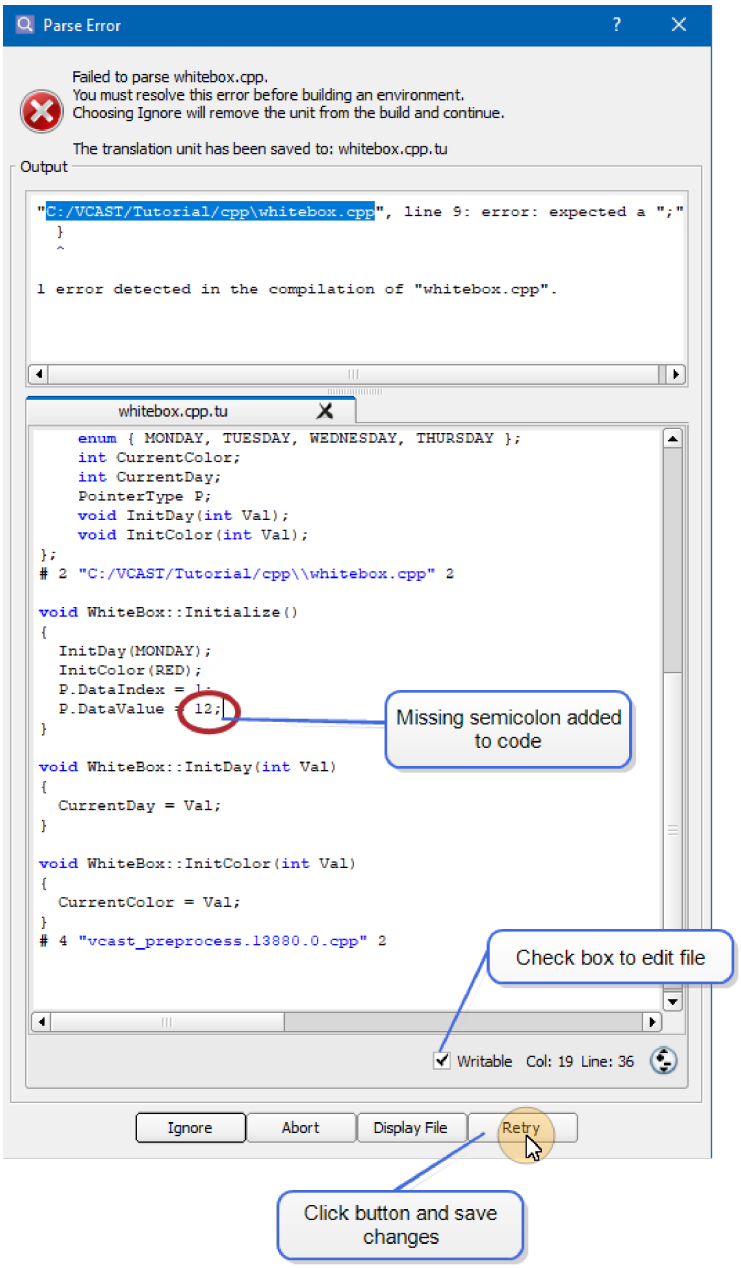


This dialog shows the error(s) found while pre-processing or parsing the source files found in the directories specified on the Source directories list. There are several courses of action you can take:

- Choosing **Ignore** will remove the unit from the build and automatically continue the build for all other units.
- If the error indicates a compilation error in a UUT source file, you can edit the file directly from this dialog box. Highlight the file name in the Output panel by left-clicking and dragging the mouse. Right-click the highlighted name and then select the **Display File** button.


Parse Error
Failed to parse whitebox.cpp.
You must resolve this error before building an environment.
Choosing Ignore will remove the unit from the build and continue.
The translation unit has been saved to: whitebox.cpp.tu
Output
"C:/VCAST/Tutorial/cpp\whitebox.cpp", line 9: error: expected a ";"
}
^
1 error detected in the compilation of "whitebox.cpp".
Ignore
Abort
Display File
Retry

A text editor appears in the lower panel. Check the **Writeable** control and then make the necessary changes in the file. When editing is complete, click the **Retry** button. You will be prompted to save the changes. After saving the changes, the build proceeds.



> Clicking the **Abort** button returns you to the Wizard for any modifications you wish to make.

Tools to Aid Troubleshooting

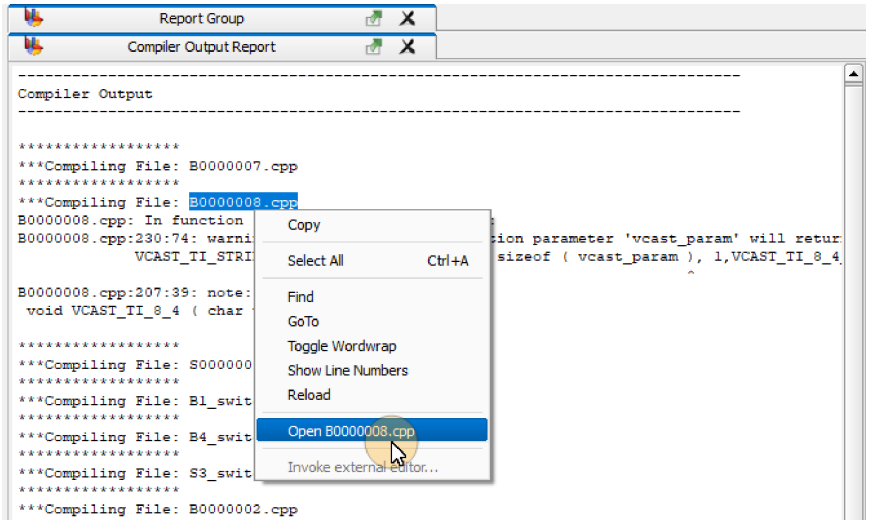
Compile Log File

All compiler diagnostic messages that are generated during the initial environment construction or during the recompile of an existing environment are saved in the Compile Log File (ACOMPILE.LIS) located in the environment directory, and can be viewed using the **Environment => View => Compile Errors** command.

```
clicast -e <env> ENvironment Extract Compile_errors [<outputfile>]

Extract compile errors to standard output or to a file if specified. If
VCAST_CUSTOM_REPORT_FORMAT is HTML, the file is saved to <env>_compiler_
output.html.
```

For example, in the TEXT Compiler Output listing shown below, we have highlighted filename B0000008.c and right-clicked on it. Selecting **Open B0000008.c** from the popup menu opens that file. (This feature does not work for HTML reports on Windows, because VectorCAST has an embedded IE browser.)

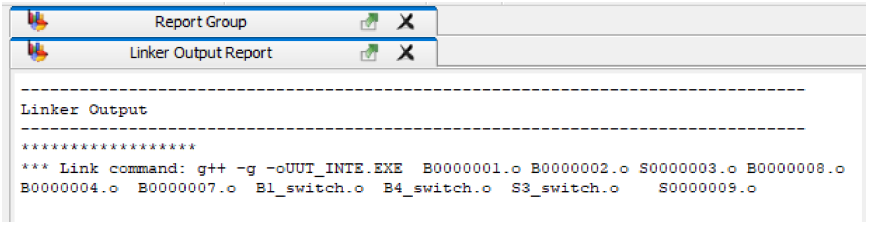


Binder/Linker Log File

All linker diagnostic messages that are generated during environment linking or during the re-linking of an existing environment are saved in the Binder/Linker Log File (AALINKER.LIS) located in the environment directorty, and can be viewed using the **Environment => View => Link Errors** command.

```
clicast -e <env> ENvironment Extract Link_errors [<outputfile>]

Extract link errors to standard output or to a file if specified. If VCAST_
CUSTOM_REPORT_FORMAT is HTML, the file is saved to <env>_linker_
output.html.
```



Environment Build Log File

All messages generated by the environment builder during environment creation or rebuilding are captured in the Environment Build Log file.

This file can be viewed from within VectorCAST using the **Environment => View => Environment Build Log** command. The log data includes information written to the Message window.

```
clicast -e <env> ENvironment Extract Build_log [<outputfile>]

Extract the environment build log to standard output or to a file if
specified. If VCAST_CUSTOM_REPORT_FORMAT is HTML, the file is saved to
<env>_env_build_report.html.
```

Unstubbed Entities Log

The Unstubbed Entities Log enables you to view the listing of all functions or global data objects that are used by the Unit(s) Under Test, but are not defined in any other source file from the Source directories list. This report is very useful in diagnosing link errors.

This file can be viewed using the **Environment => View => Unstubbed Entities Log**. It is not an error log; your environment could be compiled and linked successfully without having to define any undefined entities on this list.

The Unstubbed Entities Log is divided into sections by directory. For each directory, undefined entities are listed, and for each entity, detailed information about whether the entity was created, and if not, why it was not created.

This report is useful for diagnosing link errors that occur when you build an environment. Many times, link errors are caused by an entity whose directory was not in the Source directories list. To correct this problem, you may add additional directories to the Source directories list in the Create New Environment wizard. If the entity is in the Source directories list and VectorCAST was not able to provide its definition, you may add your own definition to the test harness via Environment User Code.

```
clicast -e <env> ENvironment Extract Undefined_entities [<outputfile>]

Extract the undefined entities log to standard output or to a file if
specified. If VCAST_CUSTOM_REPORT_FORMAT is HTML, the file is saved to
<env>_undefined_entities_report.html.
```

An example of an undefined entity that cannot be completed by VectorCAST is an extern to an

anonymous class. The source code is as follows:

uut.cpp

```
#include "struct.h"

int uut() {
   return A.a;
}
```

struct.h

```
#ifndef EXTERN
#define EXTERN extern
#endif

EXTERN struct {
   int a;
} A;
```

A link error occurs when this environment is built, because the struct is declared but not defined. Thus, the Unstubbed Entities Log is displayed.

Unstubbed Entities Log

```
*** This is not an error log ***

    The existence of entities in this log does not signify that an error
    has occurred. The purpose of this log is to provide a reason to you
    when a symbol has not been stubbed by VectorCAST. When a link error
    occurs, the 'Reason Not Stubbed' column provides an explanation. In
    certain cases you may be required to provide a definition for certain
    entities in user code. Please consult the User's Guide for more
    information.



Directory: L:/ENV/inc
Symbol Name  Type      Reason Not Stubbed
-------------------------------------------------------------------------
A            variable  cannot define an extern to an anonymous class
```

Once you know the reason the entity cannot be defined by VectorCAST, you can solve this link error by adding the following to Environment User Code (Header section), which will define the struct:

```
#define EXTERN
#include "struct.h"
```

To Create a System Testing Environment

The **File => New => System Testing Environment** command enables you to build a System

Testing environment, if you have VectorCAST/QA licensed from Vector. The command may also be launched by clicking the arrow ( ) to the right of the New button  on the toolbar.

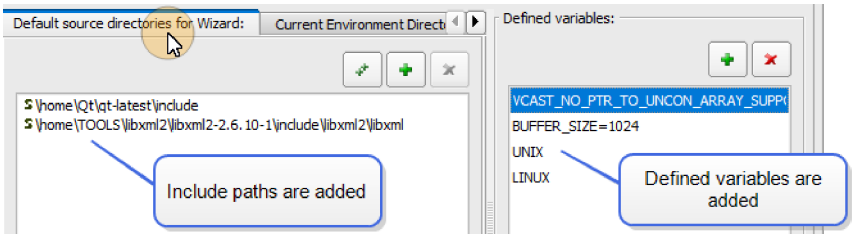
Refer to the *VectorCAST/QA User's Guide* for information on using VectorCAST/QA.

Setting Compiler Options

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.



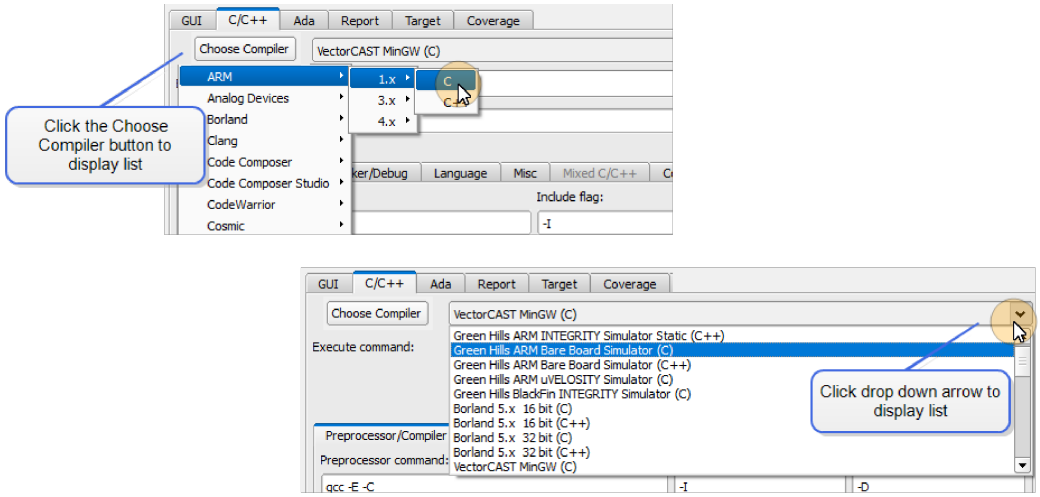
Tip: Selecting a Compiler Template loads in settings for other edit boxes automatically. You can modify these settings if necessary.

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.
```

Execute Command

When performing embedded testing, choose the method of executing on the target or simulator. A default will be provided if you chose a compiler template. A blank line indicates that the target is the host machine. To change this command, type directly into the text box. In general, the value that is set for this option when you choose a compiler template is the appropriate command. If you need to add a parameter to the default command, simply edit the displayed command.

VectorCAST provides some special case entries for certain targets, indicated by <<cmd>>. If you are using a target, refer to the applicable Application Note for your particular RSP for more information on the functionality provided by these special commands. RSP Application Notes can be accessed on the vector.com Download Center page. They can also be accessed by using the RSP icon on the main Tool Bar; however, the documentation displayed in the dropdown menu is dependent on the compiler template selected in the environment's compiler configuration.

```
clicast -lc option C_EXECUTE_CMD <command>

Command (or target flag) used to indicate method of executing harness. Its
default value is set by the compiler template.
```

RGW Database Location

To specify the location of an RGW database whose contents you want to be loaded by default whenever you open an environment, add the database's path to the **RGW database location** option. The path to the database is typically specified as a full path, but can be a path relative to the working directory.

Clicking the browse button (

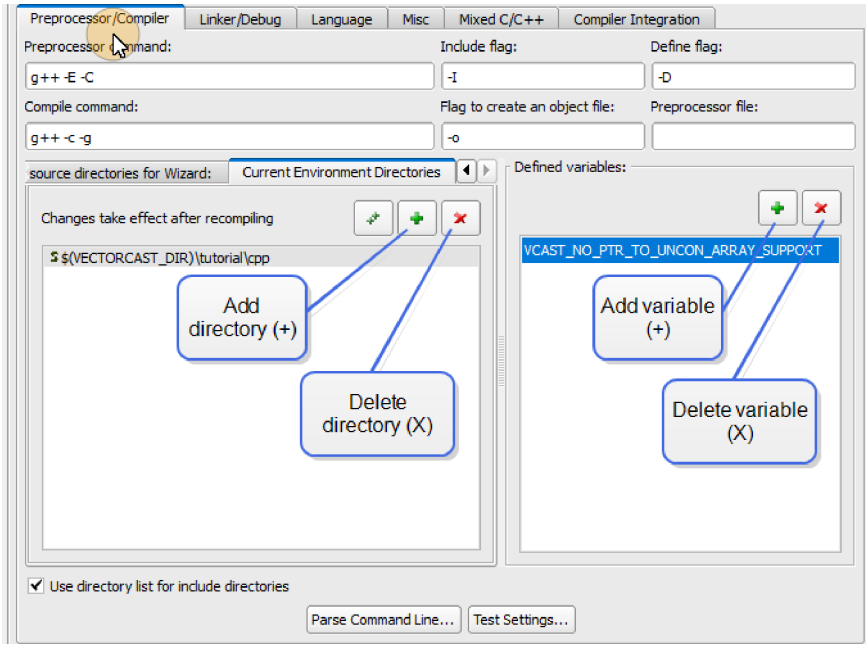
) opens the standard Choose a Directory dialog, from which you can navigate to the directory that contains the RGW database.

```
clicast -lc option VCAST_REPOSITORY <directory path>

This is the path to the directory that contains the Requirements Gateway
Database.
```

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

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.

```
clicast -lc option C_DEFINE_FLAG <flag>

Command line option used to specify values for C/C++ preprocessor. <flag>
is a quoted string, such as "-D", "/D". 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.
```

Default Source Directories for Wizard

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Preprocessor/Compiler tab**.

This option provides a method for you to specify search directories, library include directories, and type-handled directories as the default list for the Create New Environment Wizard. Each time you select **File => New => C/C++ Environment**, the Source directories list will contain those listed here.

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  .

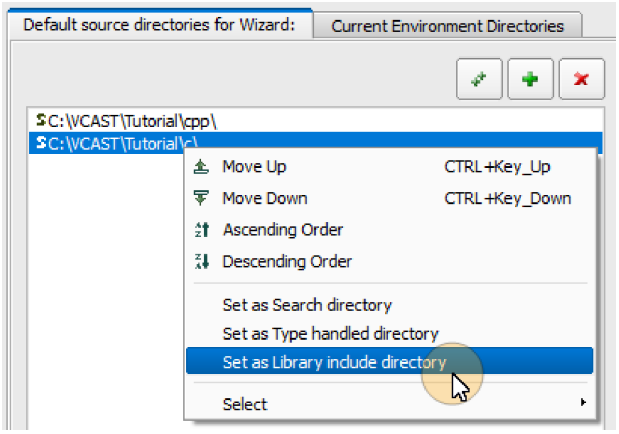
Once a directory is added, you can change the type. Right-click and choose:

- **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.



Note: Changes made here do not have any effect on an existing environment. This listing makes it more convenient to build a new environment.

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 -lc set_default_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.

clicast -lc option clear_default_source_dirs

Remove all instances of TESTABLE_SOURCE_DIR:<path>, LIBRARY_INCLUDE_DIR:<path>, and TYPE_HANDLED_SOURCE_DIR:<path> from the CCAST_.CFG file, which represent the default source directories that are loaded into the Create New Environment wizard.

Current Environment Directories

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Preprocessor/Compiler tab**.

This option provides a method for you to add a Library Include directory to resolve problems due to a missing compiler, system, or third-party header file.

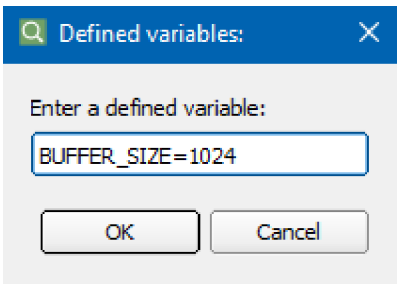
First, add your Library Include directory. Then choose **Environment => Recompile => Automatic** from the Menu Bar. Changes take effect after recompiling.

Note: You cannot delete Source directories or add new Source directories here. To do that, use the Update Environment wizard.

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.

Use Directory List for Include Directories

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Preprocessor/Compiler** tab.

This option controls whether directories specified in the source directory list are used as include directories when invoking a preprocessor or compiler.

Source directories might be specified via CCAST_.CFG, vcshell.db, or an .env file, depending on tool configuration. Disable this option when include directories should only be specified via unit-specific options, such as when the source directory list is very long, or when different units require different orders in the arguments.

```
clicast -lc option VCAST_USE_DIR_LIST_FOR_INCLUDES TRUE|FALSE

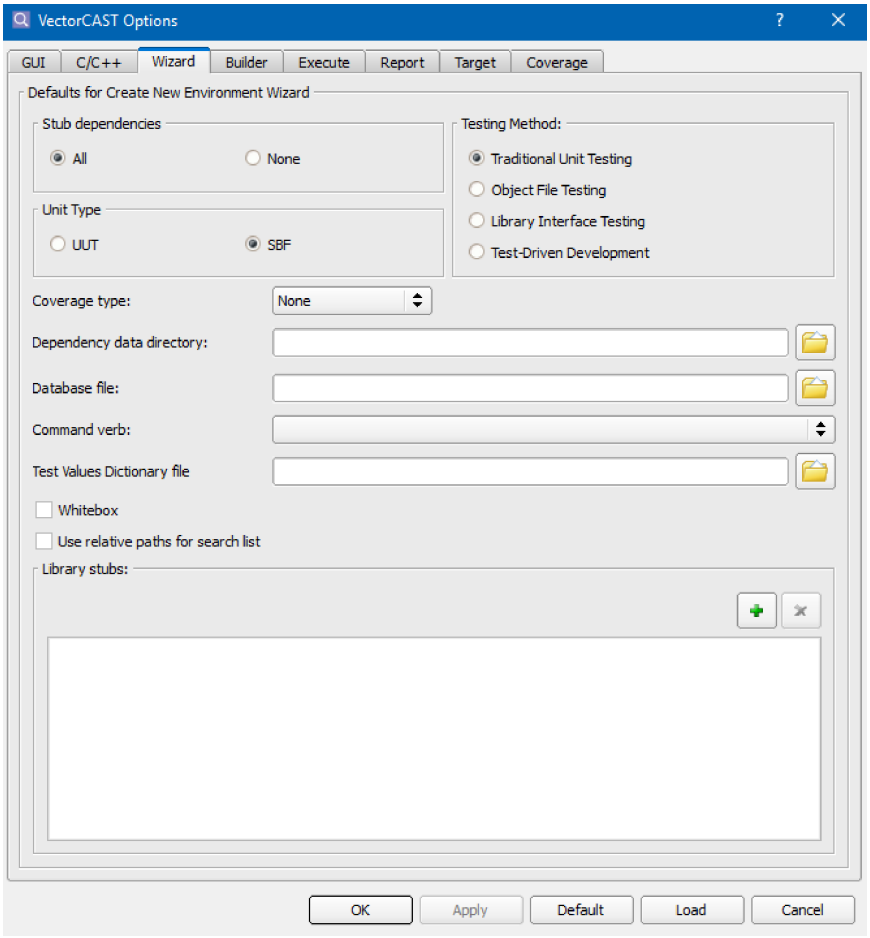
Use items from the source directory list as include directories when
invoking the preproprocessor or compiler. The default value is True.
```

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**).

Setting Defaults for the Wizard

This section of the Tools Options enables you to choose a default value for some of the Create New Environment wizard settings. In all cases, values you choose here can be overridden in the wizard. A full description of each of these options and how it affects the building of an environment is discussed in the section Using the Wizard to Create an Environment.



Once the environment is built, the environment script reflects the choices made in the Create New Environment or Update Environment wizard.

Stub Dependencies

Choose **Tools => Options**, and click the **Wizard** tab.

The Stub Dependencies feature provides the following choices for the default stubbing type displayed in the Create New Environment wizard:

- All, that is, stub all by prototype (default)
- None, that is, stub none

```
clicast -lc option STUB_DEPENDENCIES Yes | No | Custom

Default setting for Stub Dependencies option in the environment builder.
ALL means stub all dependencies, NONE means do not stub any dependencies.
Its default value is YES (All).
```

The default, “All,” refers to the stubbing of all dependent units of the unit under test. “None” specifies that no units are to be stubbed during environment creation. If a particular dependent unit is not

stubbed then the actual source code will be linked into the test harness.

Once the environment is built, the environment script reflects the choices made in the Create New Environment or Update Environment wizard.

Unit Type

Choose **Tools => Options**, and click the **Wizard** tab.

The default setting for Unit type in the Create New Environment wizard. Choose one of the following:

- UUT – allows stubbing only for functions external to the unit being tested
- SBF (default) – allows stubbing of functions within the unit being tested

```
clicast -lc option VCAST_UNIT_TYPE UUT | SBF

The default setting for Unit type in the Create New Environment wizard.
UUT allows stubbing only for functions external to the unit being tested.
Stub By Function (SBF) allows stubbing of functions within the unit being
tested. The default value is SBF.
```

Testing Method

Choose **Tools => Options**, and click the **Wizard** tab.

The Testing Method provides the following choices for the default testing method displayed in the Create New Environment wizard:

- **Traditional Unit Testing** (default) - VectorCAST will parse the implementation of the functions and methods that exist in your C/C++ source files, and create the test driver based on these. Additionally, any prototypes found for called functions that are not part of the test environment will be stubbed.
- **Object File Testing** - VectorCAST will work in the same way as for Traditional Unit Testing, except the original source files will not be compiled into the test harness. You will use the "Link Options" to provide the path to the object files or the library file containing the object files for all units under test.
- **Library Interface Testing** - VectorCAST will parse the definition of the functions and methods that exist in your C/C++ header files, and create the test environment based on these. No stubs will be created. It is assumed that the test harness will be linked against a library archive that contains the implementation of the functions and methods under test.
- **Test-driven Development** - VectorCAST will parse the definition of the functions and methods that exist in your C/C++ header files, and create the test environment based on these. A distinction between this method and Library Interface Testing is that for Test Driven Development, VectorCAST will create "stubbed place-holders" for the functions under test. Additionally, as you start to develop the implementations of the functions and methods under test, the real implementations can replace the place-holders incrementally. This allows test cases to be developed as soon as the top-level software design is complete.

```
clicast -lc option LIBRARY_TESTING Source| Library

Default setting for Library Testing Method option in environment builder
```

When using Traditional Unit Testing method, the environment script contains:

ENVIRO.SBF or ENVIRO.STUB_BY_FUNCTION
ENVIRO.STUB
ENVIRO.UUT

When using Object File Testing method, the environment script contains:

ENVIRO.OJBECT_FILE_UUT

When using Library Interface Testing method, the environment script contains:

ENVIRO.LIBRARY_UUT

When using Test-driven Development, the environment script contains:

ENVIRO.TDD_HEADER or ENVIRO.TDD_SBF_HEADER
ENVIRO.TDD_SOURCE or ENVIRO.TDD_SBF_SOURCE

Coverage Type

Choose **Tools => Options**, and click the **Wizard** tab.

This option sets the type of coverage instrumentation to perform immediately after building the environment. By setting this option to a value other than None, coverage will be initialized each time you build a new environment.

The choices provided in the context menu are dependent upon the current Industry Mode (see "To Set the Industry Mode for Coverage" on page 30 of this manual). If no Industry Mode is set, the Default set of coverage types are used.

Here is a list of coverage types listed by Industry Mode and the clicast commands to set them:

Avionics (DO-178 B/C)

- Level A (Statement, Branch, MC/DC),
- Level B (Statement, Branch)
- Level C (Statement)

```
Clicast -lc option COVERAGE_TYPE None | LEVEL_A | LEVEL_B | LEVEL_C

Type of coverage instrumentation to perform immediately after building
environment. Its default value is NONE. By setting this option to a value
other than None, coverage will be initialized each time you build a new
environment.
```

Once the environment is built, the environment script has the line:

ENVIRO.COVERAGE_TYPE: <type>
where **<type>** is None | LEVEL_A | LEVEL_B | LEVEL_C

Automotive (ISO-26262)

- Unit Level ASIL D (Statement, Branch, MC/DC)
- Unit Level ASIL B/C (Statement, Branch)
- Unit Level ASIL A (Statement)
- Architectural Level ASIL C/D (Function, Function Call)
- Architectural Level ASIL_A/B (Function)

```
clicast -lc option COVERAGE_TYPE None | ASIL_D | ASIL_B/C | ASIL_A | ASIL_
C/D | ASIL_A/B

Type of coverage instrumentation to perform immediately after building
environment. Its default value is NONE. By setting this option to a value
other than None, coverage will be initialized each time you build a new
environment.
```

Once the environment is built, the environment script has the line:

ENVIRO.COVERAGE_TYPE: <type>
where **<type>** is None | ASIL_D | ASIL_B/C | ASIL_A | ASIL_C/D | ASIL_A/B

Industrial (IEC-61508)

- SIL4 (Statement, Branch, MC/DC),
- SIL3 (Statement, Branch)
- SIL 1/2 (Statement)

```
clicast -lc option COVERAGE_TYPE NONE | SIL_4 | SIL_3 | SIL_1/2

Type of coverage instrumentation to perform immediately after building
environment. Its default value is NONE. By setting this option to a value
other than None, coverage will be initialized each time you build a new
environment.
```

Once the environment is built, the environment script has the line:

ENVIRO.COVERAGE_TYPE: <type>
where **<type>** is None | SIL_4 | SIL_3 | SIL_1/2

Railway (EN-50128)

- SIL4 (Statement, Branch, MC/DC),
- SIL3 (Statement, Branch)
- SIL 1/2 (Statement)

```
clicast -lc option COVERAGE_TYPE None | SIL_4 | SIL_3 | SIL_1/2

Type of coverage instrumentation to perform immediately after building
environment. Its default value is NONE. By setting this option to a value
other than None, coverage will be initialized each time you build a new
environment.
```

Once the environment is built, the environment script has the line:

ENVIRO.COVERAGE_TYPE: <type>
where **<type>** is None | SIL_4 | SIL_3 | SIL_1/2

Medical (IEC-62304)

- Class C (Statement, Branch, MC/DC)
- Class B (Statement, Branch)
- Class A (Statement)

```
clicast -lc option COVERAGE_TYPE None | CLASS_C | CLASS_B | CLASS_A

Type of coverage instrumentation to perform immediately after building
environment. Its default value is NONE. By setting this option to a value
other than None, coverage will be initialized each time you build a new
environment.
```

Once the environment is built, the environment script has the line:

ENVIRO.COVERAGE_TYPE: <type>
where **<type>** is None | CLASS_C | CLASS_B | CLASS_A

Default

- Statement
- Branch
- Basis Paths
- MC/DC
- Function
- Function+Function Call
- Statement+Branch
- Statement+MC/DC

```
clicast -lc option COVERAGE_TYPE
    None | STATEMENT+MC/DC | STATEMENT+BRANCH | FUNCTION+FUNCTION_CALL |
FUNCTION | MC/DC | BASIS_PATHS | BRANCH | STATEMENT

Type of coverage instrumentation to perform immediately after building
environment. Its default value is NONE. By setting this option to a value
other than None, coverage will be initialized each time you build a new
environment.
```

Once the environment is built, the environment script has the line:

ENVIRO.COVERAGE_TYPE: <type>
where **<type>** is None | STATEMENT+MC/DC | STATEMENT+BRANCH | FUNCTION+FUNCTION_CALL | FUNCTION | MC/DC | BASIS_PATHS | BRANCH | STATEMENT

Dependency Data Directory

This option sets the location where you want VectorCAST to store the output from QuickParse (*.QIK). It is this directory that is cleared when you click the **Clear Dependency Data** button in the Create New (or Update) Environment wizard. If this option has no specification, QuickParse stores the dependency data in the search directories. Use this option if you do not have write permissions for the search directories.

```
clicast -lc option VCAST_DEPENDENCY_CACHE_DIR <directory path>

Location where the output from QuickParse is stored (*.QIK). QuickParse
determines the dependencies relative to the UUTs. Specify this option if
you do not have write access to the search directories. Its default value
is empty, meaning use the search directories specified by ENVIRO.SEARCH_
LIST in the environment script (.env) to store the VCAST.QIK files.
```

Database File

This option sets the path to the compiler settings database created by vcshell (vcshell.db). The path can be absolute or relative, and can contain environment variables, using the syntax **$(ENV_VAR)** or **%ENV_VAR%**.

```
clicast -lc option VCDB_FILENAME <vcdb filename>

 path to the compiler settings database created by vcshell (vcshell.db).
The path can be absolute or relative, and can contain environment
variables, using the syntax $(ENV_VAR) or %ENV_VAR%.
```

Command Verb

Set this option to the compile command verb that vcshell "sees" when it processes your build settings. For example, the command line compile command may be **gcc** but the underlying executable that the system sees might be **cc1**. In this case, set this option to be **cc1**.

```
clicast -lc option VCDB_CMD_VERB <vcdb replacement verb>

Set this option to the compile command verb that vcshell "sees" when it
processes your build settings.
```

Test Values Dictionary File

This option sets the path to the test values dictionary file.

```
clicast -lc option VCAST_TEST_VALUES_DICTIONARY <path to test values
dictionary file>

Path to the test values dictionary file.
```

Variant Logics Path

Choose **Tools => Options**, and click the **Wizard** tab.

This option sets the path to the variant logics file. The option value is used as the default value for the Variant Logics path selector in the Create New Environment wizard.

```
clicast -lc option VARIANT_LOGICS_PATH <path to variant logics file>

Path to .py file containing test case variant logic definitions written in
Python. A Logic from this file can be assigned to one or more testcases,
and is evaluated prior to execution to determine if the test is eligible
for test execution. Path can be absolute or relative, and can contain
environment variables, using the syntax $(ENV_VAR) or %ENV_VAR%.
```

Whitebox

Choose **Tools => Options**, and click the **Wizard** tab.

Default settings for the **Whitebox** checkbox in the Create New Environment wizard.

```
clicast -lc option WHITEBOX Yes | No

Default settings for the "Whitebox" checkbox in the environment builder.
Its default value is YES (whitebox) for C environments, NO (blackbox) for
other environments.
```

Once the environment is built, the environment script has the line: ENVIRO.WHITE_BOX: YES | NO.

Use Relative Paths for Search List

Choose **Tools => Options**, and click the **Wizard** tab.

Default setting for the **Use relative paths** checkbox in the Create New Environment wizard. The paths are relative to the working directory.

```
clicast -lc option VCAST_USE_RELATIVE_PATHS True | False

If set, the new environment dialog will use relative paths. The target I/O
directory will also use relative paths.The default value is False.
```

If checked, the wizard uses paths relative to the working directory in the Search Directories list. If unchecked, the Search Directories list uses fully qualified paths. Once the environment is built, the environment script has the line: ENVIRO.SEARCH_LIST:<path>, where <path> is one of the paths in the Search Directories list, and is either written relative to the current working directory, or is a full path.

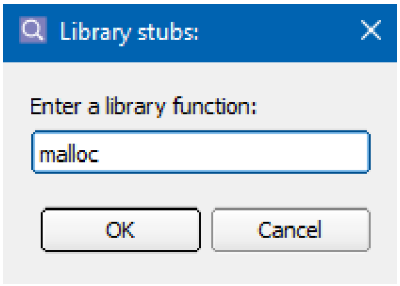
Library Stubs

Choose **Tools => Options**, and click the **Wizard** tab.

This is a list of library functions to be stubbed by default in the Create New Environment wizard.

Stubbing a library function allows you to simulate failures that would be difficult or impossible to achieve with the actual functions being used. For example, by stubbing malloc, you can force a bad status to be returned from the malloc call.

To specify a standard C library function to be stubbed by default, click the **Add Function** button  . In the dialog, type a function name and click **OK**.



When you create a new environment, this function is listed on the “Library Stubs” tab on Step 6 of the Create New Environment wizard.

If you add a library function name to this list when an environment is open, you must enable the stubbing of the library function *explicitly*. To do this, go to the **Update Environment wizard**, Step 6. Select the **Library Stubs** tab (scroll to the right to find it) then check the box next to the library function you want to stub in this existing environment.

See the example provided with VectorCAST in the /Examples/C/stubbing_c_lib directory for an example on how to use a non-stubbed version of a library stub on a testcase-by-testcase basis.

```
clicast -lc option VCAST_LIBRARY_STUBS <list of library functions>

List of functions from libraries to stub (e.g. malloc) by default in the
Create New Environment wizard.
```

Once the environment is built, the environment script has the line: ENVIRO.LIBRARY_STUBS:

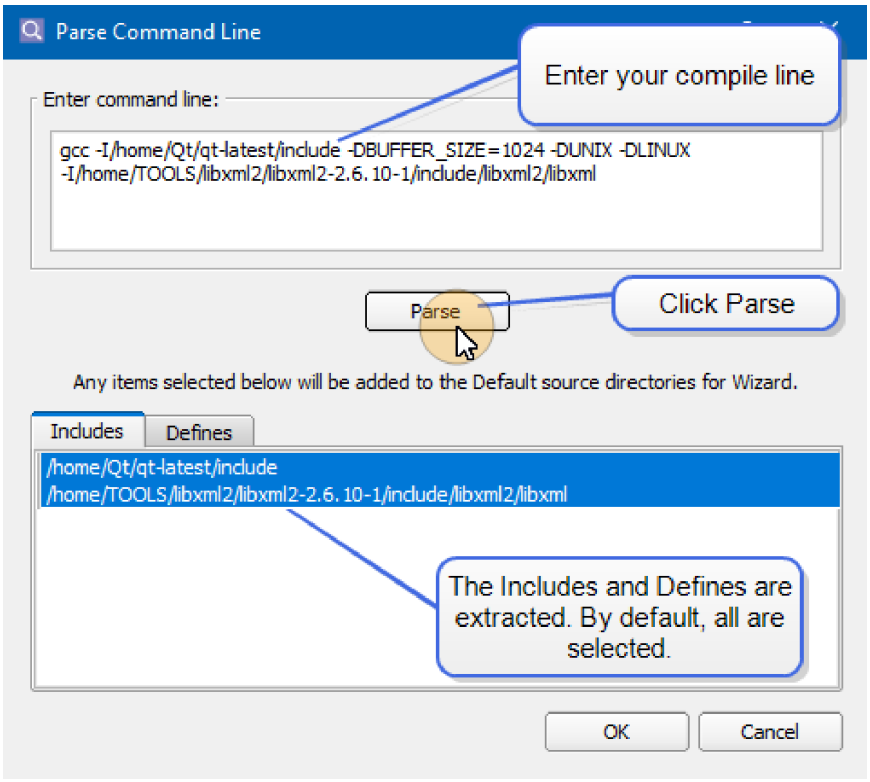
<function1> [<function2>] … , where <function> is the name of a library function to stub (such as malloc).

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 enter 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 our example we are using the gcc compiler, and the normal compile command is:

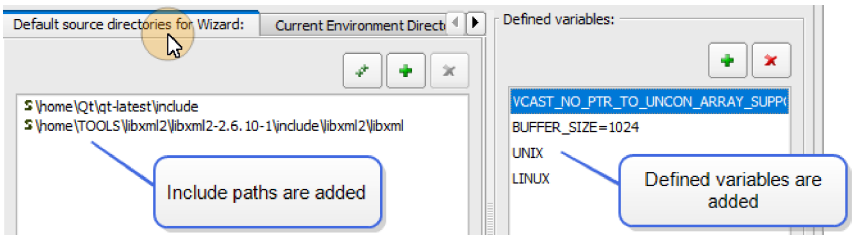
```
gcc
-I/home/Qt/qt-latest/include
-DBUFFER_SIZE=1024
-DUNIX
-DLINUX
-I/home/TOOLS/libxml2/libxml2-2.6.10-1/include/libxml2/libxml
```

We enter the command line and 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.

By default, each item is selected. To eliminate an item, unselect it.

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.

The Parse Command Line dialog closes, and the Include paths and Defined variables are now present in their respective edit boxes.



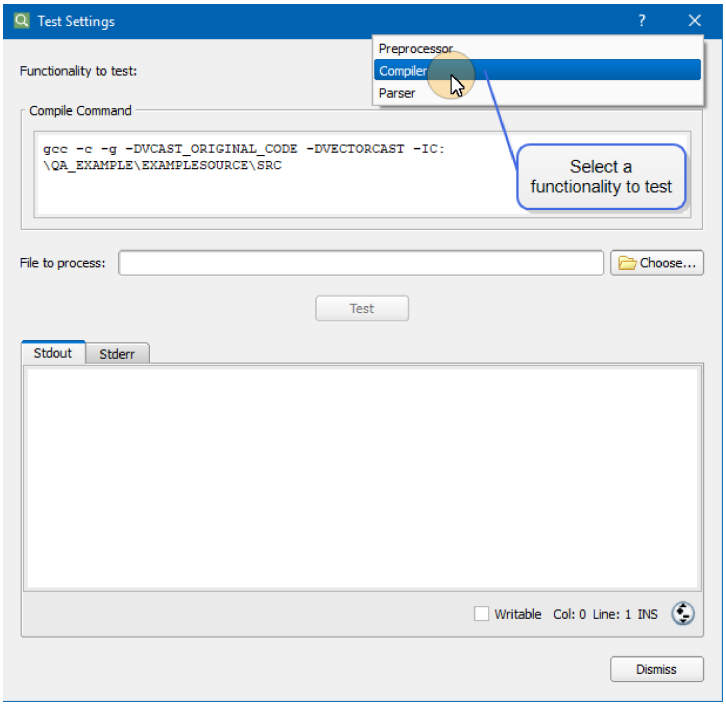
To Test Your Compiler Settings

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.

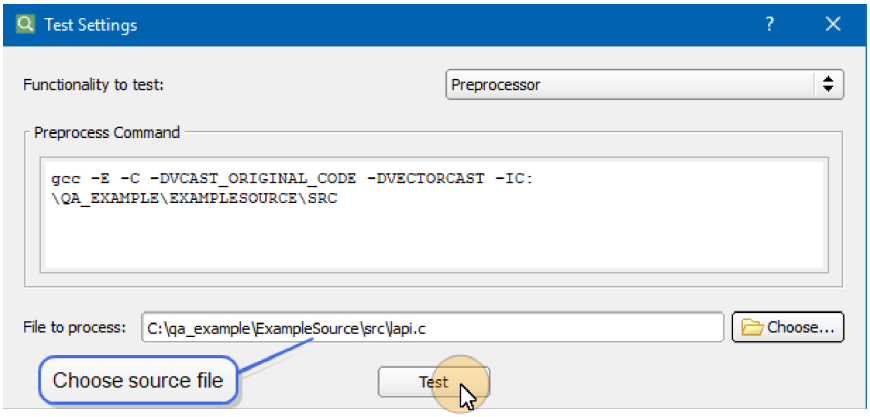


When **Preprocessor** is selected, the preprocess command, with any defines or includes, is displayed in the “Preprocess Command” box. When **Compiler** is selected, the compile command is displayed in the “Compile Command” box. When **Parser** is selected, the Parser Flags are displayed in the “Parser 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.

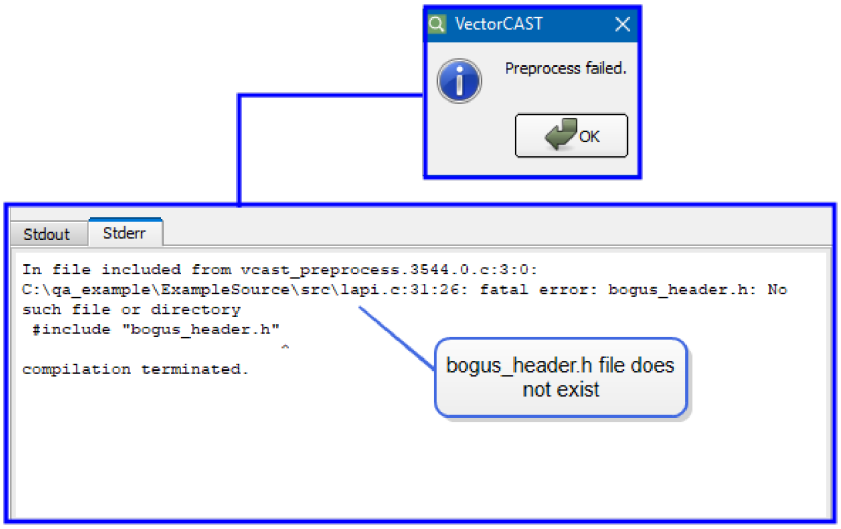
Some defined variables may be added by VectorCAST based on options set in the Builder tab. If you have an environment open, then all paths in the Source directories are listed here as -I<search dir>. When Parser is selected, the Parser flags are displayed in the “Parser Flags” box.

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.

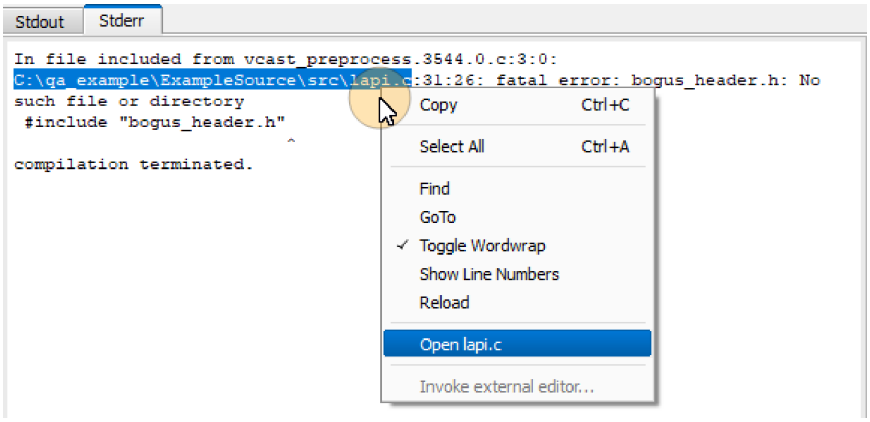


Click the **Test** button. VectorCAST uses the displayed command to preprocess or compile the file. A message dialog appears indicating whether the command was successful or failed.

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>**.

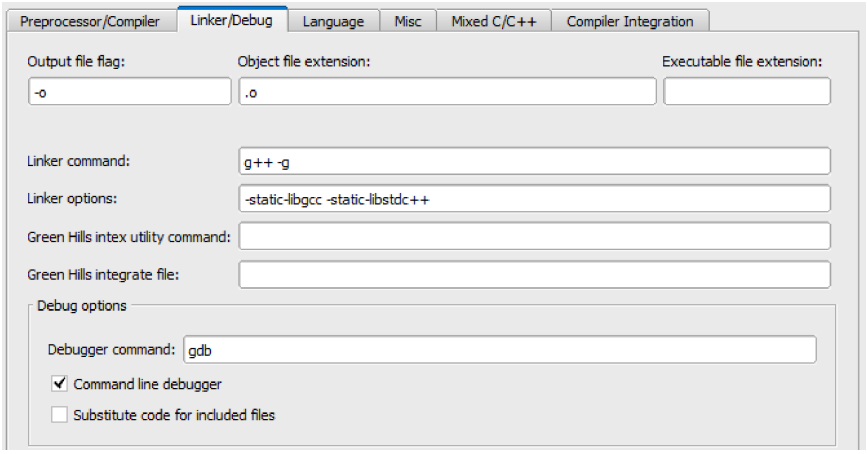


When you are done testing the compiler settings, click the **Dismiss** button. You return to the **Tools => Options dialog, C/C++** tab. If you made any changes to the command being tested, then you must transfer these changes to the C/C++ tab manually.

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.
```

Substitute Code For Included Files

When constructing the test harness, some VectorCAST-generated source files may **`#include`** other source files that contain function definitions. Some debuggers are not able to show source code for functions defined in **`#included`** files.

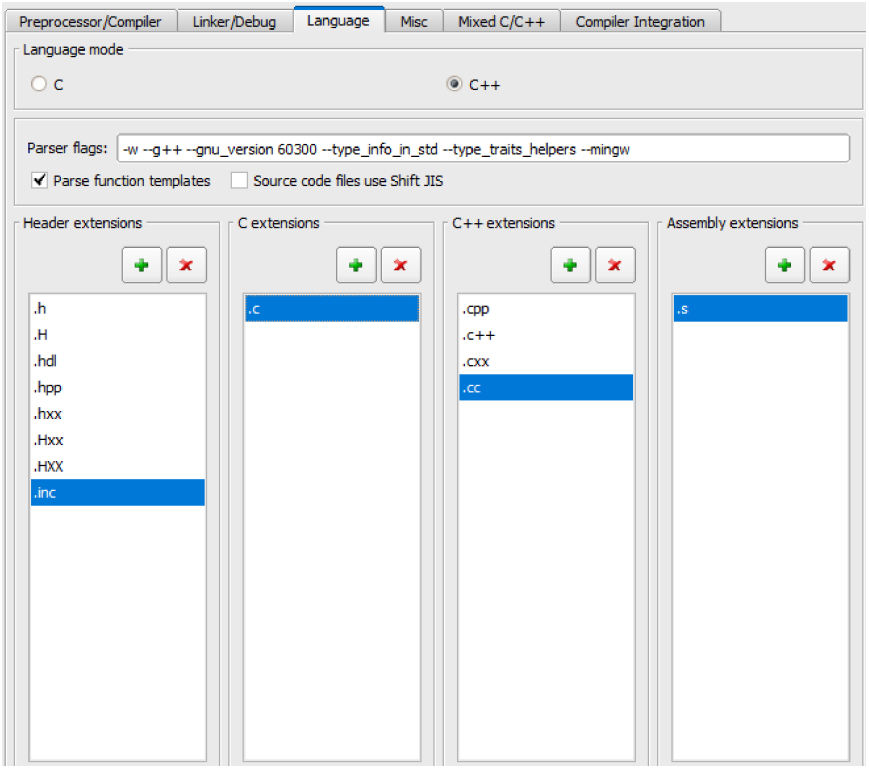
Enable this option to direct VectorCAST to replace such **`#includes`** with the content of the **`#included`** file before compiling. Enabling this option requires environment rebuild.

```
clicast -lc Option SUBSTITUTE_CODE_FOR_C_FILE True | False

When constructing a test harness, some VectorCAST-generated source files
may #include other source files that contain function definitions. Some
debuggers are not able to show source code for functions defined in
#included files. Enable this option to have VectorCAST replace such
#includes with the content of the included file before compiling. Rebuild
the environment after enabling this option. Its default value is false.
```

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.
```

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.
```

Source Code Files Use Shift JIS

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Language** tab.

This option is used if source code files use the Shift JIS character encoding.

```
clicast -lc option VCAST_MULTIBYTE_CHARACTERS True | False

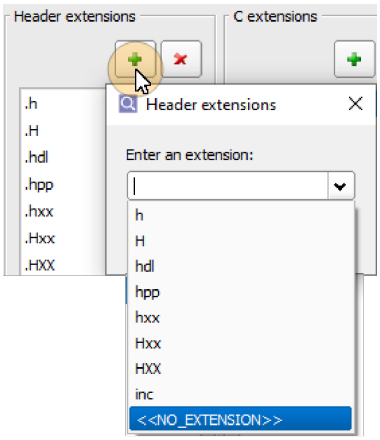
Enable this option if source code files are encoded using Shift JIS. The
default value is False.
```

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.

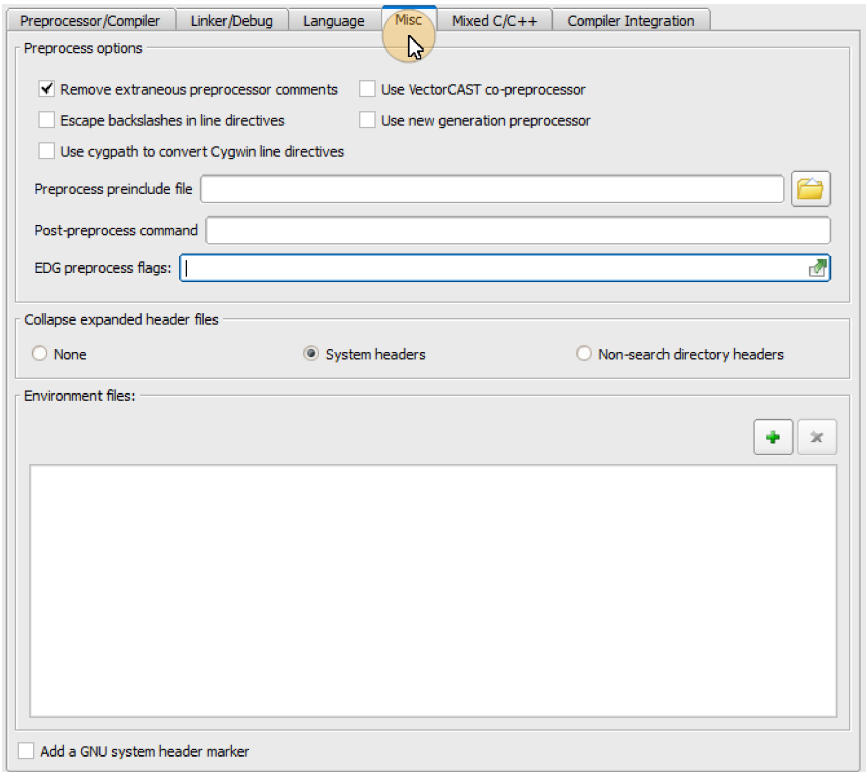
When entering a new extension, VectorCAST adds a "." character if one is not provided.

```
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.
```

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.
```

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 cygpath to Convert Cygwin Line Directives

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Misc** tab.

This option controls whether line directives in preprocessor output are converted from Cygwin-style paths to Windows-style paths. Enable this option when using a compiler that produces Cygwin-style paths. This option requires that the cygpath executable can be found via the PATH environment variable.

This option is enabled by default when selecting a compiler template for GNU Native compilers version 5.0 or greater on Windows platforms. Previously, the VCAST_USE_VCPP option was enabled for correcting paths with such compiler versions, but is no longer needed for that purpose.

```
clicast -lc Option VCAST_CYGPATH_LINE_DIRECTIVES True | False

This option controls whether line directives in preprocessor output are
converted from Cygwin-style paths to Windows-style paths. Enable this
option when using a compiler that produces Cygwin-style paths. This option
requires that the cygpath executable can be found via the PATH environment
variable. The default value is True for GNU Native compilers version 5.0 or
greater on Windows. For all other compilers, the default value is False.
```

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.
```

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.
```

EDG Preprocess Flags

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Misc** tab.

This option allows you to pass additional command-line arguments to the EDG preprocessor used when VCAST_USE_EDG_PREPROCESSOR is TRUE or when VectorCAST performs macro detection. This option is mainly useful for specifying **`--include_dirs_file`** arguments when needed.

For example:

```
clicast -lc option VCAST_EDG_PREPROCESS_FLAGS --include_dirs_
file=C:\path\to\file\include.txt
```

where **`include.txt`** contains the include directory, such as: **`C:\path\to\inc`**

```
clicast -lc Option VCAST_EDG_PREPROCESS_FLAGS <flags>

Use this option to pass additional command-line arguments to the EDG
preprocessor used when VCAST_USE_EDG_PREPROCESSOR is TRUE or when
VectorCAST performs macro detection.
```

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.



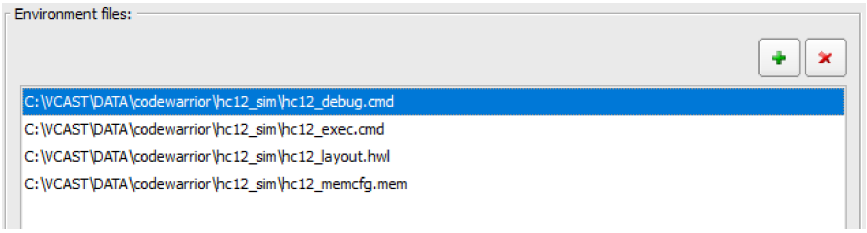
Note: Contact VectorCAST Technical Support before changing the value of this option.

```
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 or rebuild. 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 or rebuild. Multiple paths
are separated by a comma. Its default value is set by the compiler
template.
```

Add a GNU System Header Marker

Choose **Tools => Options**, and click the **C/C++ tab**. Then click the **Misc tab**.

This option controls whether VectorCAST inserts a GNU-style system header line marker at the start of instrumented files and other files containing expanded header content. Enable this option when using GNU-based or clang-based compilers and the compiler issues errors in VectorCAST-generated files for unmodified code expanded from a system header.

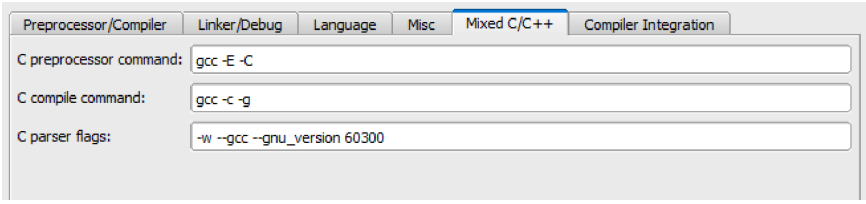
```
clicast -lc option VCAST_GNU_SYSTEM_MARKER True|False

This option controls whether VectorCAST inserts a GNU-style system header
line marker at the start of instrumented files and other files containing
expanded header content. Enable this option when using GNU-based or clang-
based compilers and the compiler issues errors in VectorCAST-generated
files for unmodified code expanded from a system header. The default value
is False.
```

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.

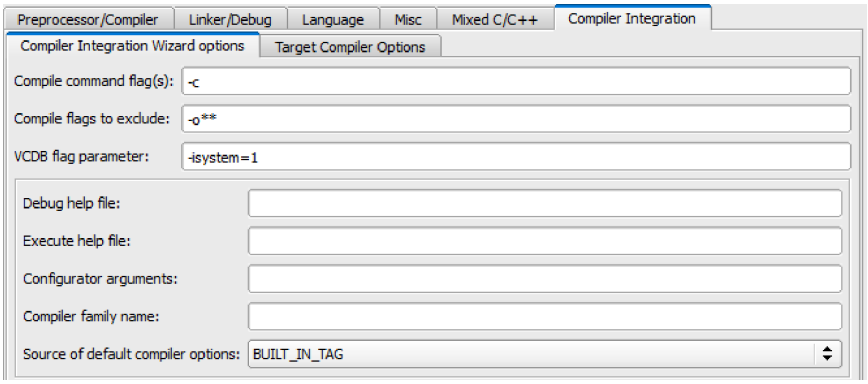
The "C parser flags" option specifies the flags to pass to the QuickParser when parsing C source files in a C++ environment.

```
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.
```

VCDB Flag Parameter

Choose **Tools => Options**, and click the **C/C++** tab. Then click the **Compiler Integration** tab and the **Compiler Integration Wizard options** sub-tab.

List of extra flags to be passed to vcdb as the **--flags** parameter. By default, the **--flags** are set to **-I=1,-D=1**. This option lets you add additional flags such as **-isystem=1**.

```
clicast -lc option VCAST_VCDB_FLAG_STRING <flag string such as -I=1,-D=1>

List of extra flags to be passed to vcdb as the --flags parameter. By
default, the --flags are set to -I=1,-D=1. This option lets you add
additional flags such as -isystem=1.
```

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.

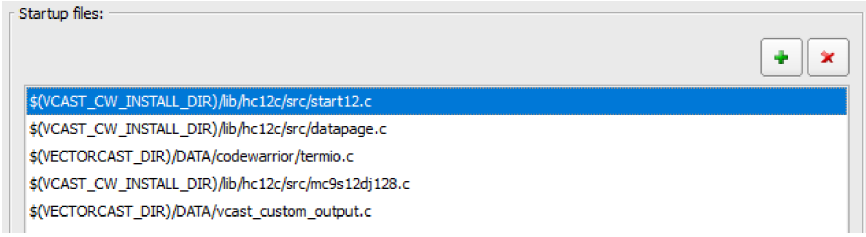
```
clicast -lc option PRECOMPILE_EXT <command>

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.

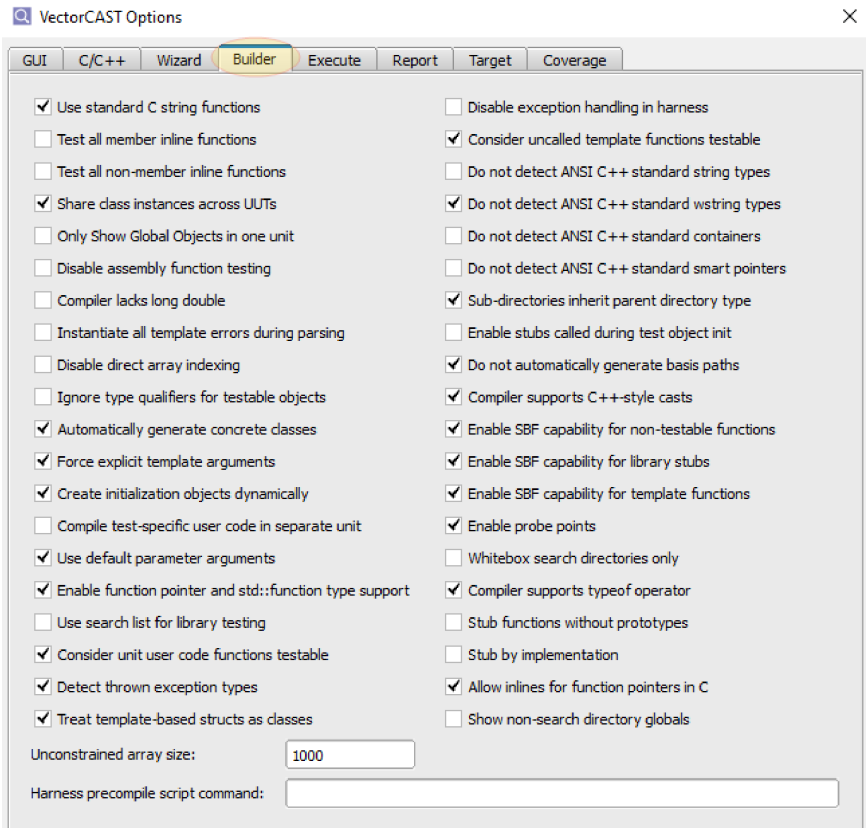
Setting Builder Options

Options: Builder Tab

Choose **Tools => Options** and click the **Builder** tab.

The Builder options relate to building a new environment or rebuilding an environment.

These options affect an open environment only after the environment is rebuilt. If you change an option before creating an environment (in the same directory as the CAST_.CFG file), then the option will affect new environments.



See also "To Set VectorCAST Options" on page 376.

Use Standard C String Functions

Choose **Tools => Options**, and click the **Builder tab**.

If checked, this option causes VectorCAST to use the standard C library string functions instead of VectorCAST versions of these functions. If your C/C++ runtime library provides implementations of the standard string functions, set this option.

```
clicast -lc option VCAST_USE_STD_STRING True | False

This build option will cause the standard C library string functions to be
used instead of their respective VCAST versions. The default value is True.
```

See also "Key Terminology" on page 17 for a discussion of what makes a class testable.

Test All Member Inline Functions

Choose **Tools => Options**, and click the **Builder** tab.

If checked, this option causes all inlined member functions that are found while parsing the C++ source files in the Search List to be made testable. Therefore, they appear in the Parameter Tree and they appear selected in the Classes of Interest tab of the Create New Environment wizard. If this option is not checked, inlined member functions can still be made testable using the Classes of Interest tab in the Create New Environment wizard.

```
clicast -lc option VCAST_TEST_ALL_INLINES True | False

This option is used to tell VectorCAST all inlined member functions found
in a header file on the search list should be testable. (C++ only.) The
default value is False.
```

Test All Non-Member Inline Functions

Choose **Tools => Options**, and click the **Builder** tab.

If checked, this option causes all non-member inlined functions that are found while parsing the C++ source files in the Search List to be made testable.

```
clicast -lc option VCAST_TEST_ALL_NON_MEMBER_INLINES True | False

This option is used to tell VectorCAST all non-member inlined functions
found in a header file on the search list should be testable. (C++ only.)
Its default value is False.
```

Share Class Instances Across UUTs

Choose **Tools => Options**, and click the **Builder** tab.

This option enables VectorCAST to share class instances across multiple units under test. Only objects that are legal to declare are shared. That is, if a unit doesn't have visibility to a class type, then that class won't be shared in that unit. This option is useful when multiple units are under test and you want to use a class instance constructed in one unit as a parameter to a function in another unit.

```
clicast -lc option VCAST_CLASS_INST_SHARING True | False

Use this option to share class instances across multiple units under test.
Only units that are legal to declare will be shared. Its default value is
True.
```

Only Show Global Objects in One Unit

Choose **Tools => Options**, and click the **Builder** tab.

By default, global objects are displayed in the Parameter Tree of the Test Case Editor under every unit in which they are accessible. To help avoid accidentally setting a global object twice, enable this

option which causes global objects to be displayed in the Parameter Tree under the first unit in which they are referenced.

```
clicast -lc option VCAST_ONLY_SHOW_GLOBAL_OBJECTS_IN_ONE_UNIT True | False

By default, global objects are shown in the GUI under every unit in which
they are accessible. When this option is enabled, global objects will only
be displayed from the first unit in which they are referenced.The default
value is False.
```

Disable Assembly Function Testing

Choose **Tools => Options**, and click the **Builder** tab.

If checked, assembly functions in the source code are made untestable. Set this option if driver calls to assembly functions in the harness are causing a compile error during environment build.

```
clicast -lc option VCAST_DISABLE_ASM_FUNC_TESTING True | False

When this option is selected, assembly functions will not be considered
testable. The default value is False.
```

Compiler Lacks Long Double

Choose **Tools => Options**, and click the **Builder** tab.

If checked, this option indicates that your compiler does not support the long double data type.

```
clicast -lc option VCAST_NO_LONG_DOUBLE True | False

This option indicates that your compiler does not support the long double
data type. Its default value is set by the compiler template.
```

Instantiate All Template Functions During Parsing

Choose **Tools => Options**, and click the **Builder** tab.

If checked, this option instructs the VectorCAST parser to attempt to instantiate template functions that have been referenced and all member functions of template classes that have been referenced when parsing source files. It is useful for catching problems that would be compile errors when the environment is built.

```
clicast -lc option VCAST_INSTANTIATE_ALL_TEMPLATE_FUNCTIONS True | False

Set this option to cause VectorCAST to instantiate template functions that
have been referenced and all member functions of template classes that have
been referenced when parsing source files. The default value is False.
```

Disable Direct Array Indexing

Choose **Tools => Options**, and click the **Builder** tab.

For some arrays, VectorCAST is not able to determine the index type for the array. These cases usually involve constants or expressions in the definition of the array in the package. Therefore, the array indices are presented in the pos notation. This notation is used when this option is on.

For example, if you have an array subscripted from -10 to 10, if this option is True, then VectorCAST will create the script with subscripts 1 to 21. If this option is False, VectorCAST will output the subscripts from -10 to 10.

```
clicast option VCAST_DISABLE_DIRECT_ARRAY_INDEXING True | False

This option is used to allow VectorCAST to know how to display the
subscripts of an array in the Parameter Tree and in the test script. For
example, if you have an array indexed with an enumerated value, if this
option is set, then the array subscripts are displayed with the enumerated
constants. If this option is not set (default), VectorCAST displays the
subscripts with the enumerations. The default value is False.
```

Ignore Type Qualifiers for Testable Objects

Choose **Tools => Options**, and click the **Builder** tab.

This option allows the testing of objects whose types normally prohibit testing because of qualifiers (i.e. **`const`** or **`volatile`**, etc.).

When this option is enabled, type qualifiers are ignored in the test harness, allowing the testing of objects that use these prohibitive types. When the option is disabled, the objects that use the prohibitive type require user code to test.

```
clicast option VCAST_TI_IGNORE_QUALIFIERS True | False

This option allows the testing of objects whose types normally prohibit
testing because of qualifiers (i.e. const or volatile, etc.). When this
option is enabled, type qualifiers are ignored in the test harness,
allowing the testing of objects that use these prohibitive types. When the
option is disabled, the objects that use the prohibitive type require user
code to test. The default value is False.
```

Automatically Generate Concrete Classes

Choose **Tools => Options**, and click the **Builder** tab.

When this option is enabled, VectorCAST automatically creates a concrete class for each abstract class found in the source code under test. In the Parameter Tree, the dropdown menu listing the constructors for the class includes the generated concrete class, prefixed with **`vcast_concrete_`**.

```
clicast option VCAST_AUTO_CONCRETE_CLASS_GENERATION True | False

When this option is set, VectorCAST will generate concrete subclasses of
abstract classes found in the code under test. The default value is True.
```

Force Explicit Template Arguments

Choose **Tools => Options**, and click the **Builder** tab.

This option controls whether the test harness explicitly specifies template arguments when invoking template-based functions. When set, template arguments are always explicitly specified. When not set, template arguments are explicitly specified only if they were explicitly specified when the template was instantiated in the original translation unit.

Note that if a template-based function is not instantiated in the original translation unit but is testable, then with this option off the template arguments will not be explicitly specified, which will cause a compile error if the template arguments cannot be deduced from the function arguments. If the option must remain off to handle the problem with template-based member functions in Visual C++ 6, the compile errors can be eliminated by instantiating the template in unit appendix user code and rebuilding the environment.

```
clicast option FORCE_EXPLICIT_TEMPLATE_ARGS True | False

When VectorCAST generates a call to a template-based function, explicit
template arguments are used if calls to the function in the original code
used explicit template arguments. If there were no calls to the function in
the original code, then generated calls to the function will use explicit
arguments only if this option is enabled. The default value is True.
```

Create Initialization Objects Dynamically

Choose **Tools => Options**, and click the **Builder** tab.

This option causes VectorCAST to create initialization objects dynamically, rather than define them at file scope before **`main()`** is invoked. The option is True by default.

Note that this option applies only to initialization objects used for stubbed constructor member initializers. Initialization objects for stubbed global variables are never constructed dynamically.

```
clicast option VCAST_CREATE_INIT_OBJECTS_DYNAMICALLY True | False

When this option is enabled, VectorCAST will create initialization objects
that are constructed dynamically. When disabled, initialization objects are
defined at file scope and will be constructed at static initialization time
(before main is invoked). Note this option applies only to initialization
objects used for stubbed constructor member initializers. Initialization
objects for stubbed global variables are never constructed dynamically. The
default value is True.
```

Compile Test-Specific User Code in Separate Unit

Choose **Tools => Options**, and click the **Builder** tab.

This option helps to reduce test harness compilation time. When an environment is built with this option set, VectorCAST places Testcase and Parameter user code into a separate unit. Prior to testcase execution, only this separate, "user code unit" is compiled and linked into the test harness, instead of compiling the entire UUT every time.

To allow visibility to UUT code, the separate unit #includes an interface file that VectorCAST generates based on the original UUT code with function and object definitions transformed into declarations. This increases the complexity of the harness and causes the environment building process to slow somewhat. This option is compatible with "Traditional Unit Testing" and "Test-Driven Development" Testing Methods only.



Note: If your work in the environment involves making edits to test-specific user code, test compiling, and executing tests, then you should set this option on. If you are modifying source code and rebuilding the environment frequently, you should set the option off.

```
clicast -lc option VCAST_SPLIT_UC_FILES True | False

Enabling this option causes test-specific user code to be inserted into a
separate unit in the harness. To allow visibility to UUT code, the separate
unit #includes an interface file that VectorCAST generates based on the
original UUT code with function and object definitions transformed into
declarations. This increases the complexity of the harness and the
environment building process but can speed up user code compilation in
large environments. This option is compatible with "Traditional Unit
Testing" and "Test-Driven Development" Testing Methods only. Note that when
user code is inserted into a separate unit, user code references to
entities with internal (static) linkage will be references to different
instances than are used by a UUT. The default value is False.
```

Use Default Parameter Arguments

Choose **Tools => Options**, and click the **Builder** tab.

When the option is set to true, the harness sets the variables passed as parameters for test functions to their default arguments.

```
clicast -lc option VCAST_USE_DEFAULT_ARGS True | False

When creating the harness, VectorCAST will set up functions to be called
with their default arguments. The default value is True.
```

Enable Function Pointer and std::functionType Support

Choose **Tools => Options**, and click the **Builder** tab.

In the Parameter Tree, the type for function pointer parameters is type "FunctionPointer." The Input and Expected Values provide a dropdown menu that lists all global functions that match the signature

of the parameter. If only <<null>> is shown in the list, then no compatible functions were found.

When this option is disabled, the parameter type in the Parameter Tree is shown as "users" and you must set values using Parameter User Code.

```
clicast -lc option VCAST_FUNCTION_POINTER_SUPPORT True | False

Set this option to allow setting and checking function pointer and
std::function variables and parameters in the test case parameter tree.
When this option is disabled, such items can be set and checked only via
user code. The default value is True.
```

Use Search List for Library Testing

Choose **Tools => Options**, and click the **Builder** tab.

This option allows functions in header files that are **`#included`** by a UUT header file to be testable, assuming it is also on the Search List.

For example, suppose the UUT header file, **`uut.h`**, **`#includes a.h`** and **`b.h`**. Suppose **`uut.h`** and **`a.h`** are in the same directory, while **`b.h`** is in a different directory. The Search List has both directories. When this option is False (default value), only functions declared in **`uut.h`** are testable and displayed in the Test Case Tree. When the option is True, functions in **`uut.h`** as well as **`a.h`** and **`b.h`** are made testable, because they are #included by the UUT and also on the Search List.

```
clicast -lc option VCAST_USE_SEARCH_LIST_FOR_LIBRARY_TESTING True | False

Use the search list instead of a header file name to determine testability
in Library Test Mode. The default value is False.
```

Consider Unit User Code Functions Testable

Choose **Tools => Options**, and click the **Builder** tab.

This option controls whether functions defined in unit appendix and unit prefix user code show as testable in the test case tree. Note that regardless of the value of this option, such functions do not get instrumented for coverage or SBF stubbing.

```
clicast -lc option VCAST_TEST_UNIT_USER_CODE TRUE | FALSE

Enable this option to allow functions defined in unit prefix user code or
unit appendix user code to be testable. Note that functions in such user
code do not get coverage instrumentation, even when this option is enabled.
The default value is True.
```

Detect Thrown Exception Types

Choose **Tools => Options**, and click the **Builder** tab.

This option, enabled by default, allows test cases to set expected exceptions of any type that are thrown directly from the body of the function under test.

If a function definition uses an exception specification, this option is ignored and only the types from the specification can be expected exceptions. When this option is False, only functions with exception specifications can have test cases with expected exceptions.

```
clicast -lc option VCAST_DETECT_THROWN_EXCEPTION_TYPES TRUE | FALSE

Enable this option to allow test cases to specify expected exceptions of
any type thrown directly from the body of the function. If a function
definition uses an exception specification, this option is ignored and only
the types from the specification can be expected exceptions. When this
option is disabled, only functions with exception specifications can have
test cases with expected exceptions. The default value is True.
```

Treat Template-based Structs as Classes

Choose **Tools => Options**, and click the **Builder** tab.

This option, enabled by default, treats all template-based **`struct`** types like classes that use 'new' to allocate and for which generated harness objects are always pointers.

When this option is False, template-based **`struct`** types are treated like classes only when they contain declared or compiler-generated constructors, destructors, or virtual functions, or when VectorCAST detects an instantiation will not compile.

```
clicast -lc option VCAST_TEMPLATE_STRUCT_AS_CLASS TRUE | FALSE

When this option is TRUE, all template-based struct types will be treated
like classes that use 'new' to allocate and for which generated harness
objects will always be pointers. When this option is FALSE, template-based
struct types will be treated like classes only when they contain declared
or compiler-generated constructors, destructors, or virtual functions, or
when VectorCAST detects an instantiation will not compile. The default
value is True.
```

Disable Exception Handling in Harness

Choose **Tools => Options**, and click the **Builder** tab.

If this option is set, VectorCAST does not generate any code in the test harness that makes use of exception throwing or catching. This option should be used if your C/C++ runtime does not support exceptions.

```
clicast -lc option VCAST_DISABLE_CPP_EXCEPTIONS true | false

Disable the catching of unhandled exceptions in the test harness. Use this
option if your compiler does not support C++ exceptions. The default value
is False.
```

Consider Uncalled Template Functions Testable

Choose **Tools => Options**, and click the **Builder** tab.

When checked, this option instructs VectorCAST to consider member functions of a testable template class to be testable, even if they are never called. It is on by default. To remove uncalled template functions from the environment (if there is uncompilable code inside the template function), turn off this option and rebuild the environment.

In the following example, the **`TemplClass<int>::uncalled`** function is only considered testable when this option is enabled.

```
template <typename T>
class TemplClass {
  public:
    void called () {}
    void uncalled () {}
};

void uut () {
  TemplClass<int> t;
  t.called ();
}
```

`clicast -lc option VCAST_CONSIDER_UNCALLED_TEMPL_FUNCTIONS_TESTABLE True | False`

This option will consider uncalled template member functions of an already testable template class to be testable. Consider a class A<T> which contains two template functions A<T>::b and A<T>::c. If A<int>::b is instantiated, then A<int>::c would become testable as well. The default value is True.

Do Not Detect ANSI C++ Standard String Types

Choose **Tools => Options**, and click the **Builder** tab.

This option is unchecked by default, enabling VectorCAST to detect the use of ANSI C++ standard string types. As a result, you can type strings directly into the Parameter Tree for parameters of the type **`std::string`**. When checked, this option causes VectorCAST to not detect ANSI C++ standard string types, and so you must use parameter user code for those parameters. Check this option if your compiler does not support the ANSI C++ standard string type.

`clicast -lc option VCAST_DISABLE_STD_STRING_DETECTION true | false`

This option is used to allow VectorCAST to detect ANSI C++ standard string types. Normally, VectorCAST allows the user to enter a string directly. When this option is on, User Code must be used. The default value is False.

Do Not Detect ANSI C++ Standard Wide String Types

Choose **Tools => Options**, and click the **Builder** tab.

This option enables VectorCAST to detect the use of ANSI C++ standard wide-string types. When

unchecked, you can type strings directly into the Parameter Tree for parameters of the type **`std::wstring`**.

This option is checked by default. When checked, this option causes VectorCAST to not detect ANSI C++ standard **`wstring`** types, and so you must use parameter user code for those parameters. Check this option if your compiler does not support the ANSI C++ standard **`wstring`** type.

```
clicast -lc option VCAST_DISABLE_STD_WSTRING_DETECTION True | False

This option is used to allow VectorCAST to detect ANSI C++ standard wstring
types. Normally, VectorCAST allows the user to enter a wstring directly.
When this option is on, User Code must be used. The default value is True.
```

Do Not Detect ANSI C++ Standard Containers

Choose **Tools => Options**, and click the **Builder** tab.

This option enables VectorCAST to detect the use of ANSI C++ standard container types. Normally, VectorCAST allows the user to enter a container directly. When this option is on, User Code must be used.

```
clicast -lc option VCAST_DISABLE_STD_CONTAINER_DETECTION True | False

This option is used to allow VectorCAST to detect ANSI C++ standard
container types. Normally, VectorCAST allows the user to enter a container
directly. When this option is on, User Code must be used. The default value
is False.
```

Do Not Detect ANSI C++ Standard Smart Pointers

Choose **Tools => Options**, and click the **Builder** tab.

C++11 Dynamic Memory Management pointers **`(std::unique_ptr, std::shared_ptr`**) are natively supported in the Parameter Tree. Users can allocate and expect values from these smart pointer types.



Note: Environments built with VectorCAST version 2019 SP3 and earlier and using User Code that deals with smart pointers must be modified in one of the following ways prior to rebuilding:

- To continue using User Code, set the configuration option VCAST_DISABLE_STD_SMART_PTR_DETECTION to True
- Or, edit the User Code to remove a layer of indirection
- Or, edit the test Input Values and Expected Values completely, using the new functionality available in the Test Case Editor

```
clicast -lc option VCAST_DISABLE_STD_SMART_PTR_DETECTION True | False

This option is used to allow VectorCAST to detect ANSI C++ standard smart
pointer types. Normally, VectorCAST allows the user to initialize smart
pointers with the test case parameter tree. When this option is on, User
```

```
Code must be used. The default value is False, meaning detect smart
pointers in source code.
```

Sub-directories Inherit Parent Directory Type

Choose **Tools => Options**, and click the **Builder** tab.

When this option is enabled, a sub-directory inherits the type of its parent directory, unless it is explicitly given its own type. A sub-directory of a library include directory is considered a library include directory, a sub-directory of a type-handled directory is a type-handled directory, and a sub-directory of a testable source directory is a testable source directory.

```
clicast -lc option VCAST_SUBDIRS_OF_SEARCH_LIST_CONSIDERED True | False

When enabled, sub-directories of each type of source directory are the same
type as their parent, unless explicitly set to its own type. The default
value is True.
```

Enable Stubs Called During Test Object Init

Choose **Tools => Options**, and click the **Builder** tab.

When this option is set, stubs called during test object initialization (such as stubbed constructors) will perform all stub-related processing. Note that when the option is enabled, an execution report is more likely to show stub events before the call to the function under test.

```
clicast -lc option VCAST_ALWAYS_DO_STUB_PROCESSING_IN_TI True | False

This option controls whether stubs called during test object initialization
are fully enabled. Test object initialization refers to the initialization
of objects as designated in the parameter tree. When this option is
enabled, stubs called during test object initialization (such as stubbed
constructors) will perform all stub-related processing. When this option is
disabled, stubs called during test object initialization will skip most
stub-related processing, but configure stubs user code and object
initialization user code can still be executed. Note that when the option
is enabled, an execution report is more likely to show stub events before
the call to the function under test. The default value is False.
```

Do Not Automatically Generate Basis Paths

Choose **Tools => Options**, and click the **Builder** tab.

When this option is set, the environment builder does not generate the basis paths information automatically, saving some time during environment build. The Complexity column in the Metrics report indicates “(not computed)” and all basis path report items are gray. Once the environment is built, the basis paths can be computed at any time by choosing **Tools => Basis Path => Generate**.

```
clicast -lc option VCAST_DISABLE_AUTO_BASIS_PATH_GEN True | False

When this option is set, VectorCAST will not generate Basis Path
information unless specifically requested by the user. The default value is
False.
```

Compiler Supports C++ Style Casts

Choose **Tools => Options**, and click the **Builder** tab.

This option controls use of **`static_cast`** (and **`dynamic_cast`** in cases where that would be used). Some compilers do not support the **`static_cast`** now used by the harness when casting from a base to a derived type. For these compilers, the option is disabled.

```
clicast -lc option VCAST_COMPILER_SUPPORTS_CPP_CASTS True | False

Enabling this option allows VectorCAST to use C++-style cast in a C++ test
harness. When this option is disabled only C-style casts are used. The
default is set by the compiler template.
```

Enable SBF Capability for Non-Testable Functions

Choose **Tools => Options**, and click the **Builder** tab.

VectorCAST allows non-testable inlined functions to be stubbed dynamically, on a testcase-by-testcase basis. The inlines appear in the Parameter Tree grouped by header file, under "Stubbed Subprograms," each with a checkbox similar to stubbed UUT functions under the <<SBF>> node.

```
clicast -lc option VCAST_SBF_NONTESTABLE_FUNCTIONS True | False

Enabling this option allows dynamic stubbing of functions which are defined
in an SBF unit but are not testable. For example, depending on other
options, these may include functions that are inline, static, or private.
The default value is True.
```

Enable SBF Capability for Library Stubs

Choose **Tools => Options**, and click the **Builder** tab.

When set, the option provides a checkbox next to each Library Stub, allowing it to be dynamically stubbed or not stubbed per testcase. When cleared, this option disables the dynamic stubbing capability, causing the Library Stub to be always stubbed for every test case.

This option applies even without SBF units. Note that library stubs for member functions and variadic functions are not affected by this option, and are never dynamically stubbable.

Note that when set to True, inline functions specified as library stubs are stubbed if they are in headers which are not collapsed. For more information on setting the configuration option VCAST_COLLAPSE_STD_HEADERS, see "Collapse Expanded Header Files" on page 199.

```
clicast -lc option VCAST_SBF_LIBRARY_STUBS True | False

Enable this option to allow some library stubs to be dynamically stubbable.
This option applies even without SBF units. Note that library stubs for
member functions and variadic functions are not affected by this option,
and are never dynamically stubbable. The default value is True.
```

Enable SBF Capability for Template Functions

Choose **Tools => Options**, and click the **Builder** tab.

VectorCAST supports stubbing-by-function (<<SBF>>) of all instantiations, explicit or implicit, of each template function or member of a template class. The template functions appear in the Parameter Tree under the <<SBF>> node.

By default, environments with template functions in the UUT are built with stub-by-function capability. To disable this functionality, set this option to False.

```
clicast -lc option VCAST_SBF_TEMPLATES True | False

Enabling this options allows dynamic stubbing of template functions. The
default value is True.
```

Enable Probe Points

Choose **Tools => Options**, and click the **Builder** tab.

Set this option to enable the use of probe points.

```
clicast option VCAST_ENABLE_PROBE_POINTS True | False

Set this option to enable the use of probe points. The default value is
True.
```

Whitebox Seach Directories Only

Choose **Tools => Options**, and click the **Builder** tab.

The normal C++ whitebox method affects all classes regardless of their origin. Enable this option to apply whitebox instrumentation only to classes for which the class declaration is in a file from a search directory. Regardless of the value of this option, class members that are implicitly private will remain blackbox.

```
clicast -lc option VCAST_WHITEBOX_SEARCH_DIRS_ONLY True | False

When whitebox processing is on and this option is enabled, C++ access
specifiers 'private' and 'protected' are transformed to 'public' only in
files located in search directories. With whitebox processing on and this
option disabled, C++ access specifiers 'private' and 'protected' are
transformed via preprocess and compilation arguments that apply to the
entire translation unit. This option has no effect on C files. The default
```

```
value is False.
```

Compiler Supports typeof Operator

Choose **Tools => Options**, and click the **Builder** tab.

This option indicates whether the compiler accepts the **_typeof_** operator.

```
clicast option VCAST_TYPEOF_OPERATOR True | False

This option indicates that your compiler supports the typeof operator. The
default value is determined by the compiler template.
```

Stub Functions Without Prototypes

Choose **Tools => Options**, and click the **Builder** tab.

This option allows automatic stubbing of called functions that are not declared with a prototype.

Since the function return and parameter types can only be inferred, the resulting stub might have a different prototype than the real function. More significantly, undeclared functions normally defined by external libraries could inadvertently be stubbed, resulting in either multiple definition errors or unexpected behavior.

Disable this option to prevent automatic stubbing of these functions, or **#include** the proper header so that VectorCAST can identify the declaration location. With this option disabled, the best way to stub undeclared functions is to declare or **#include** the proper prototype, either in the original code or in Unit Appendix User Code. Alternatively a function can be specified in Additional Stubs in the Environment Wizard.

```
clicast option VCAST_STUB_WITHOUT_PROTOTYPE True | False

Enable this option to allow automatic stubbing of called functions that are
not declared with a prototype. Since the function return and parameter
types can only be inferred, the resulting stub might have a different
prototype than the real function. More significantly, undeclared functions
normally defined by external libraries could inadvertently be stubbed,
resulting in either multiple definition errors or unexpected behavior.
Disable this option to prevent automatic stubbing of these functions, or
#include the proper header so that VectorCAST can identify the declaration
location. With this option disabled, the best way to stub undeclared
functions is to declare or #include the proper prototype, either in the
original code or in Unit Appendix User Code. Alternatively a function can
be specified in Additional Stubs in the Environment Wizard. The default
value is False.
```

Stub by Implementation

Choose **Tools => Options**, and click the **Builder** tab.

In VectorCAST the default stubbing method is to stub-by-prototype. In the stub-by-implementation

method, VectorCAST parses the actual source code files of dependent units, rather than their function prototypes. Enable this option if you prefer the behavior of the stub-by-implementation method.

```
clicast -lc option VCAST_STUB_BY_IMPLEMENTATION True | False

Create stubs based on actual source code files instead of just their header
files. The default value is False.
```

Allow Inlines for Function Pointers in C

Choose **Tools => Options**, and click the **Builder** tab.

This option causes inlined functions to be testable as usual. In situations where an inlined function does not have a standalone definition whose address can be taken, set this option to False to avoid a link error.

```
clicast -lc option VCAST_FUNC_PTR_INLINES_C TRUE | FALSE

Enable this option to allow inline functions to be selectable for setting
or checking function pointer parameters in C units when VCAST_FUNCTION_
POINTER_SUPPORT is enabled. Disable this option if inline functions do not
have a standalone definition whose address can be taken. The default value
is True.
```

Show Non-Search Directory Globals

Choose **Tools => Options**, and click the **Builder** tab.

This option directs VectorCAST/C++ to display global variables declared in non-search directory header files in the Parameter Tree. When the option is False, such variables are not shown, but can be accessed via user code.

```
clicast -lc option VCAST_ACCESS_NON_SEARCH_GLOBALS TRUE | FALSE

When this option is TRUE, global variables declared in non-search directory
headers will be shown in the parameter tree. When the option is FALSE, such
variables will not be shown, but can be accessed via user code. The default
value is False.
```

Unconstrained Array Size

Choose **Tools => Options**, and click the Builder tab.

This option sets the default size of “extern” array objects in which the index range is not specified. For example, if VectorCAST encounters an extern (e.g. **`extern int A[];`**) and does not find the actual definition of A, it generates a definition of the size specified in this option.

```
clicast -lc option UNCON_ARRAY_SIZE <integer number> | <integer
number>:<integer number>

Default size for unconstrained arrays where the index range is unknown or
very large. May be specified as [lower bound]:[upper bound] or just [upper
bound]. The default value is 10 for upper bound.
```

Harness Precompile Script Command

Choose **Tools => Options**, and click the Builder tab.

This command is executed prior to full compilations of the test harness. It can be used to make modifications to harness files before the initial compilation. Note that this command may also be called for subsequent harness compilations, but it is not called before every compilation.

```
clicast -lc option VCAST_PRECOMPILE_SCRIPT <script>

This command is executed prior to full compilations of the test harness. It
can be used to make modifications to harness files before the initial
compilation. Note that this command may also be called for subsequent
harness compilations, but it is not called before every compilation.
```

Deprecated Builder Options

VCAST_FORCE_NO_USERGLOBALS was deprecated in VectorCAST 4.2a

VCAST_STUB_ALL_SUBPROGRAMS was deprecated in version 4.0.

VCAST_TORNADO_CONSTRUCTOR_CALL_FILE was deprecated in VectorCAST version 5.0.

Working with a Test 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.

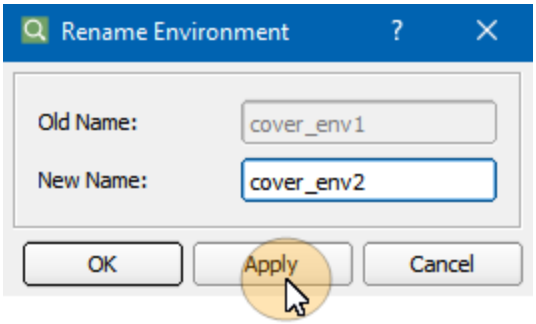
Because VectorCAST/QA writes data to the .vcp file on closing the environment, you are restricted from opening a System Testing environment that is write-protected.

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.

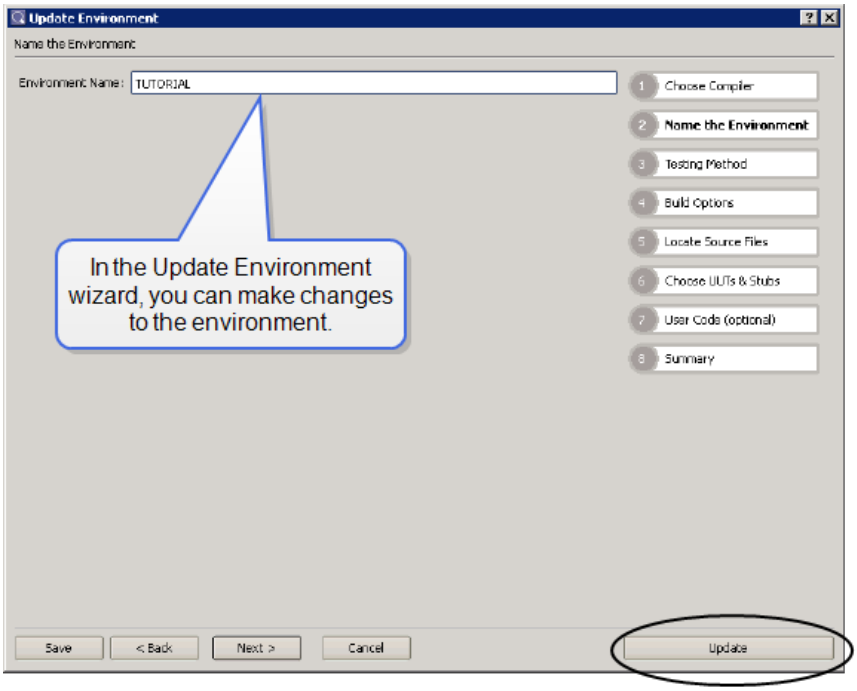
Note: The environment script is not changed when an environment is renamed.

To Update an Environment

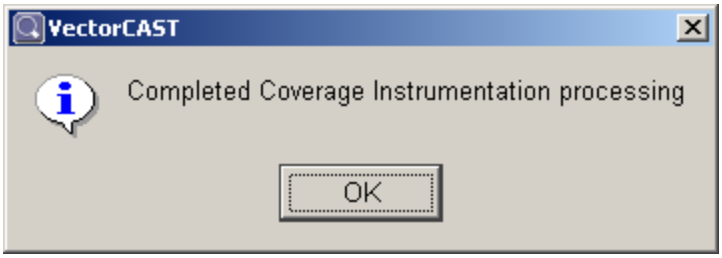
The Update Environment command provides the opportunity to easily make changes to an open environment, such as adding another UUT, adding a source directory, making the environment Whitebox, or changing the stubbing strategy. Updating the environment rebuilds the environment, and re-initializes coverage, if applicable.

When you choose **Environment => Update Environment**, VectorCAST first creates environment and test case scripts for the existing environment. It then moves the existing environment to a backup directory. The Update Environment dialog appears, with all the information filled in for the current environment. Here you can enter new information or click to a different step. Note that the Load

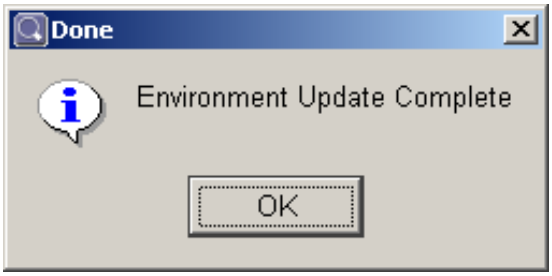
button is not available.



Click the **Update** button to begin building the environment with the changed settings. If coverage was initialized in the former environment, it is initialized in the updated environment, and the following message appears:

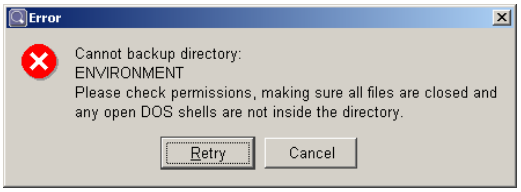


The following message indicates that the process finished successfully:



If there is some reason that VectorCAST cannot create the backup directory, then the user is alerted

and offered the choice to **Retry** or **Cancel**.



Reasons that VectorCAST cannot copy the environment directory include:

- there is a DOS shell in the environment directory
- there is a file open in the environment directory
- the directory does not have write permissions
- there already is a file or directory(for rename) with the name *environment_name*.bak in the working directory
- the environment is on a mapped drive, and the copy process isn't quite finished when VectorCAST begins to rebuild

Once you have corrected the problem, click the **Retry** button and the rebuild process continues. Click the **Cancel** button to stop the rebuild process.

See also "Using the Wizard to Create an Environment" on page 103 for more information on the various steps in the wizard.

To Create Regression Scripts for an Environment

The **Environment => Create Regression Scripts** command creates a set of regression scripts in the directory you choose (excluding the environment directory).

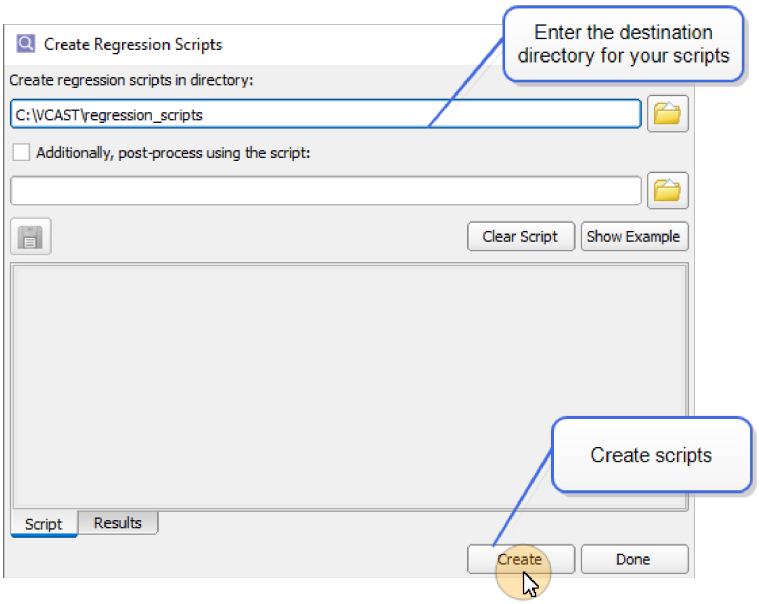
The files created are:

- a batch file *environment_name*.bat or shell script *environment_name*.sh (depending on your platform)
- an environment script with instructions on how to build the environment, *environment_name*.env
- the test script file *environment_name*.tst

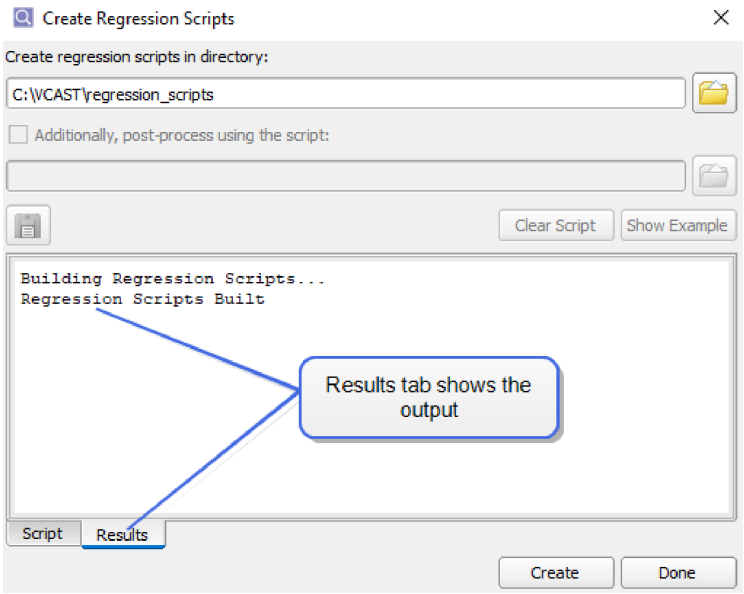
  If you have imported coverage results present in the environment, then an additional file is created:
- *environment_name*.cvr, which includes the imported coverage results themselves

This command enables you to generate the minimum set of files that allow for the regeneration of the tool options, test environment, and all test cases. 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 rebuilds the environment, loads and runs the test cases, and creates a test case management report (by default).

When you choose **Environment => Create Regression Scripts**, the following dialog appears.



The destination directory is relative to the current working directory. You can use relative or full paths. For example, entering "." causes the scripts to be created in the current working directory. Once a valid directory is entered, the Create button enables. Click **Create** to write the regression script files to the directory specified. The Create button is dimmed if you choose the environment directory. The Results tab shows that the scripts have been built successfully. Click **Done** to close the dialog.




```
clicast -e <env> TOols Regression_test [<post-processing script>]

Build the environment script, test script, and shell script (Linux) or
batch file (Windows) used to perform a complete regression test. The
```

```
optional post-processing script is invoked after the regression scripts
have been created.
```

When the regression script is run to recreate the environment, a temporary file is created, named **commands.tmp** This file contains the list of commands to build the environment, run all tests, and to create a management report. The last line of the regression script uses **commands.tmp** as an argument for the clicast command shown below. This clicast command runs all of the commands in a single invocation, using a single license checkout, and single "open" of the underlying project, resulting in enhanced performance.

```
clicast -lc -e <env> TOols EXEcute_commands <command file> [True|False]

Run multiple commands in one invocation. If the last argument is 'false',
then the commands will run to completion. Default behavior is to halt on
error.
```

The following are excerpts from the regression scripts for a Windows environment called TUTORIAL.

tutorial.bat (Windows)

```
del commands.tmp

echo options WHITEBOX NO >> commands.tmp

...

echo options C_COMPILER_TAG GNU_CPP_34 >> commands.tmp

...

echo environment build TUTORIAL.env >> commands.tmp

echo /E:TUTORIAL tools script run TUTORIAL.tst >> commands.tmp

echo /E:TUTORIAL execute batch >> commands.tmp

echo /E:TUTORIAL reports custom management TUTORIAL_management_report.html
>> commands.tmp

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

commands.tmp

```
options WHITEBOX NO

...

options C_COMPILER_TAG GNU_CPP_34

...

environment build TUTORIAL.env
```

```
/E:TUTORIAL tools script run TUTORIAL.tst

/E:TUTORIAL execute batch

/E:TUTORIAL reports custom management TUTORIAL_management_report.html
```

tutorial.env

```
ENVIRO.NEW

ENVIRO.NAME:TUTORIAL

ENVIRO.COVERAGE_TYPE: NONE

ENVIRO.STUB_BY_FUNCTION:database

ENVIRO.STUB_BY_FUNCTION:manager

ENVIRO.STUB_BY_FUNCTION:manager_driver

ENVIRO.STUB_BY_FUNCTION:whitebox

ENVIRO.WHITE_BOX:NO

ENVIRO.MAX_VARY_RANGE: 20

ENVIRO.STUB: ALL_BY_PROTOTYPE

ENVIRO.SEARCH_LIST: C:\VCAST\Environments\

ENVIRO.TYPE_HANDLED_DIRS_ALLOWED:

ENVIRO.LIBRARY_STUBS:

ENVIRO.LINK_OPTIONS:

ENVIRO.END
```

tutorial.tst

```
-- VectorCAST 5.3 (07/18/11)

-- Test Case Script

--

-- Environment : TUTORIAL

-- Unit(s) Under Test: database manager manager_driver whitebox

--

-- Script Features

TEST.SCRIPT_FEATURE:C_DIRECT_ARRAY_INDEXING
```

```
TEST.SCRIPT_FEATURE:CPP_CLASS_OBJECT_REVISION

TEST.SCRIPT_FEATURE:MULTIPLE_UUT_SUPPORT

TEST.SCRIPT_FEATURE:STANDARD_SPACING_R2

TEST.SCRIPT_FEATURE:OVERLOADED_CONST_SUPPORT

--

-- Test Case: (CL)MANAGER::PLACEORDER.001

TEST.UNIT:manager

TEST.SUBPROGRAM:(cl)Manager::PlaceOrder

TEST.NEW

TEST.NAME:(CL)MANAGER::PLACEORDER.001

TEST.VALUE:manager.(cl)Manager::PlaceOrder.Table:1

TEST.VALUE:manager.(cl)Manager::PlaceOrder.Seat:4

TEST.VALUE:manager.(cl)Manager::PlaceOrder.Order.Soup:Onion

TEST.VALUE:manager.(cl)Manager::PlaceOrder.Order.Salad:Caesar

TEST.VALUE:manager.(cl)Manager::PlaceOrder.Order.Entree:Steak

TEST.VALUE:manager.(cl)Manager::PlaceOrder.Order.Dessert:Pie

TEST.VALUE:manager.(cl)Manager::PlaceOrder.Order.Beverage:Wine

TEST.VALUE:whitebox.<<GLOBAL>>.
(cl).WhiteBox.WhiteBox.<<constructor>>.WhiteBox().<<call>>:0

TEST.END

--

... (many other test cases follow)
```

To Post-Process the Regression Scripts

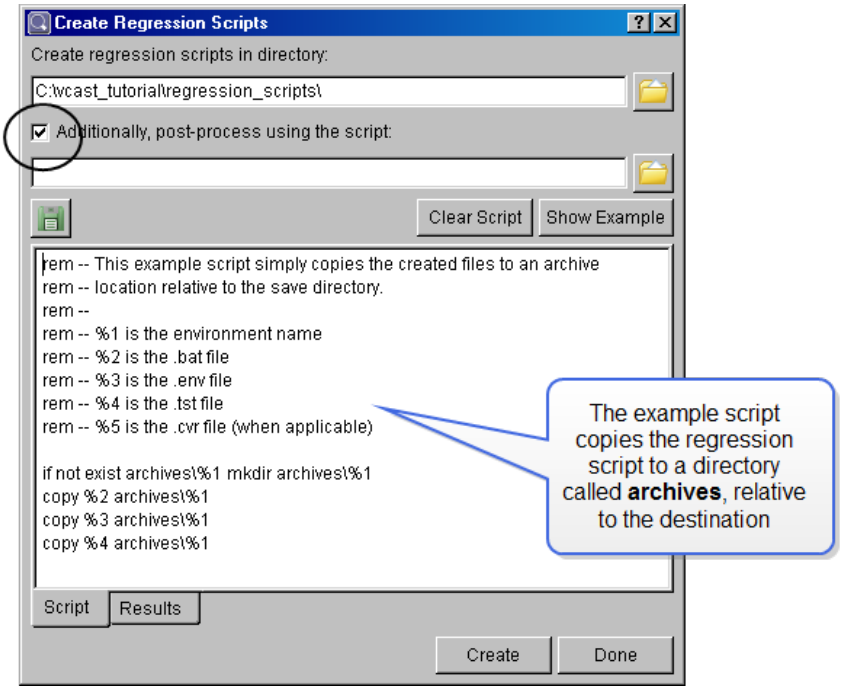
The Create Regression Scripts dialog offers the ability to copy, archive, or otherwise post-process the regression scripts. This feature is useful in cases where you want to check the regression script files into your configuration management (CM) system, or do some other sort of post-processing.

You can then save these post-processing steps to a script or just perform them once. The following instructions explain how to perform post-processing on your regression scripts.

This functionality is controlled by the checkbox "**Additionally, post-process using the script**." After checking this option, you can then type or load a script into the Script tab. When you click the Create button, the regression scripts are created and then manipulated according to your instructions in the script.

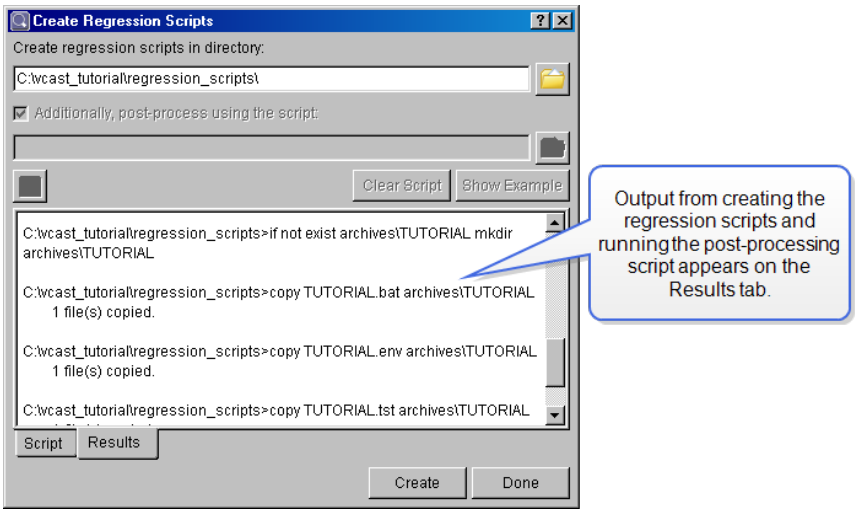
To see an example script, click the **Show Example** button. An example batch file (Windows) or shell

script (Linux) is displayed in the Script tab.



You can use this script, modify it, or clear it (click **Clear Script**) and write your own.

When you provide a post-processing script and click the **Create** button, the Results tab displays the standard output generated by post-processing the regression scripts. Note that the script itself is not saved by the Create action.




```
clicast -e <env> TOols Regression_test [<post-processing script>]

Build the environment script, test script, and shell script (Linux) or
```

```
batch file (Windows) used to perform a complete regression test. The
optional post-processing script is invoked after the regression scripts
have been created.
```

See also "To Save and Load a Post-Processing Script" on page 231 for information on saving to a script and loading in a script.

To Integrate Regression Scripts with ClearCase™

To integrate VectorCAST's regression scripts with IBM ClearCase, you use the Create Regression Scripts feature to create a script. Within the script, copy the regression script files to the directory from which you want to commit them, and then run the ClearCase commit call on these files.

1. With an environment open, choose **Environment => Create Regression Scripts**.
2. In the dialog box, choose the destination directory for the regression scripts.
3. Click the checkbox next to **Additionally, post-process using the script.**
4. Click the **Show Example** button to get started.
   The example script currently reads:

```
rem -- This example script simply copies the created files to an
rem -- archive location relative to the save directory.
rem --
rem -- %1 is the environment name
rem -- %2 is the .bat file (or .sh file on Linux systems)
rem -- %3 is the .env file
rem -- %4 is the .tst file
rem -- %5 is the .cvr file (when the environment has imported coverage
results)

if not exist archives\%1 mkdir archives\%1
copy %2 archives\%1
copy %3 archives\%1
copy %4 archives\%1
```

5. Change the script so that it runs the ClearCase commit call on the regression script files:

```
rem -- This example script simply copies the created files to an
rem -- archive location relative to the save directory.
rem --
rem -- %1 is the environment name
rem -- %2 is the .bat file
rem -- %3 is the .env file
rem -- %4 is the .tst file
rem -- %5 is the .cvr file

DEST_DIR = \home\user_name\project\regression\%1
if not exist $DEST_DIR mkdir -p $DEST_DIR
copy %2 $DEST_DIR
copy %3 $DEST_DIR
copy %4 $DEST_DIR
cd $DEST_DIR
cleartool commit *
```

6. To create the regression scripts and commit them using ClearCase, click the **Create** button.
   VectorCAST creates the three (or four) regression script files for the environment and post-processes them using the script you entered. You can save this script by clicking the **Save** button.
7. Close the dialog by clicking **Done**.

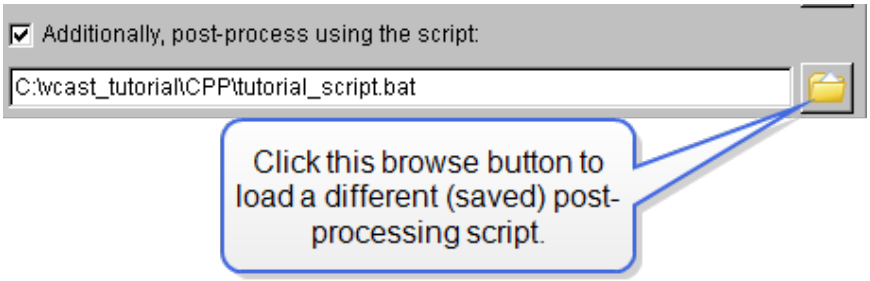
To Save and Load a Post-Processing Script

Once you have written a script in the Create Regression Scripts dialog, you can save it to use it again.

To do this, click the Save icon  . In the Save Script dialog, give the file a name and extension appropriate for your platform. Once saved, the name appears in the edit box below **Additionally, post-process using the script**.

The script name and its path is preserved in the registry (Windows) or in the .vcast-qt file (Linux). This file name is loaded in the Create Regression Scripts dialog as the default post-processing script. Once a post-processing script is saved, you can create regression scripts and post-process them with one click.

You can load an existing script file (such as a project-level file created by someone else) by using the browse button  to the right of the edit box for the script. The post-processing script you load in becomes the new default script.



See "To Post-Process the Regression Scripts " on page 228 to learn how to create a script that post-processes your regression script files.

To Incorporate Source Code Changes into Environment

Use **Environment => Recompile => Automatic** to incorporate small source code changes into your test environment. “Small” source code changes refer to any modification that does not affect a parameter or global data. If you change the interface between subprograms, that is, the type of a parameter, the number of parameters, a data structure, etc., then you would want to use **Environment => Rebuild** to be able to see the new data in the Parameter Tree.

This command recompiles the instrumented test harness as well, if you have initialized coverage.

```
clicast  -e <env> ENvironment RECompile Auto

Recompile and link all of the elements of an existing environment.
```

See also the section To (Re)Generate Basis Path Tests in "Basis Path" on page 378.

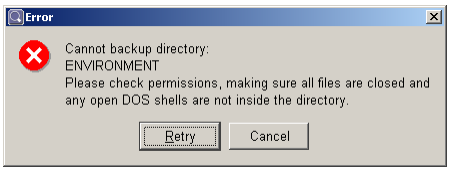
See also "To Rebuild the Environment" on page 232

See also "To Recompile the Test Harness" on page 236

To Rebuild the Environment

The **Environment => Rebuild** command completely rebuilds the test harness and associated data files (i.e. the environment). If you have changed any Builder options, VectorCAST uses the new settings. VectorCAST first creates environment and test case scripts for the existing environment. It then moves the existing environment to a backup directory, and then rebuilds the environment from the script and reloads the test cases. The backup environment directory name is *environment_name*.bak, where *environment_name* is the original environment name.

If there is some reason that VectorCAST cannot create the backup directory, then the user is alerted and offered the choice to Retry or Cancel.



Reasons that VectorCAST cannot copy the environment directory include:

- there is a DOS shell in the environment directory
- there is a file open in the environment directory
- the directory does not have write permissions
- there already is a file or directory(for rename) with the name *environment_name*.bak in the working directory
- the environment is on a mapped drive, and the copy process isn't quite finished when VectorCAST begins to rebuild

Once you have corrected the problem, click the **Retry** button and the rebuild process continues. Click the **Cancel** button to stop the rebuild process.

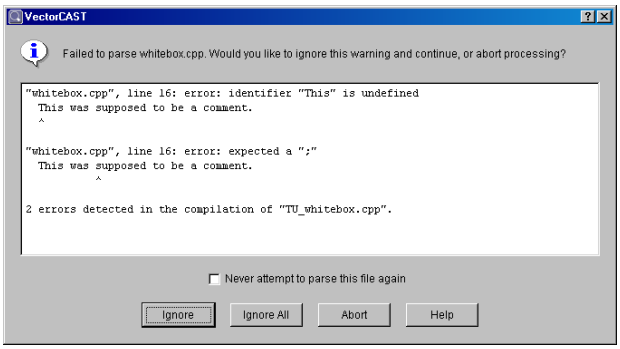
```
clicast -lc -e <env> ENvironment RE_Build

Rebuilds a previously built environment.
```

Troubleshooting Environment Rebuild

Parse Error: Source File has Error

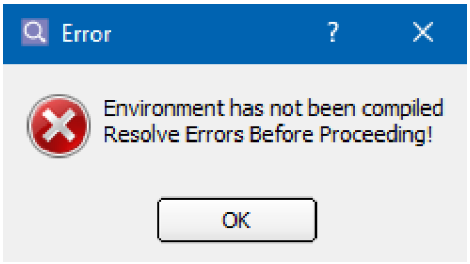
If, for some reason, you have introduced a source code error before rebuilding, you will get a parse error while rebuilding. In the example below, a comment was added to the source file but the comment marker was inadvertently left out.



To solve this problem, click the Abort button. Edit the source file to correct the error. Because the environment has already been renamed to the backup directory, you won't be able to reopen it. Instead, choose **Environment => Scripting => Run** and open the VCAST_REBUILD_TEMP.env file. When you click **OK**, the environment rebuilds successfully.

Parse Error: Compiler Set Incorrectly

If, for some reason, your compiler template is changed between the time you created the environment and the time you rebuild the environment, you may get a parse error message when you try to rebuild the environment. VectorCAST automatically reverts to the original environment for the rebuild.



Possible problems that may be encountered are:

- Changing the compiler to a C version when using CPP source files.
- Introducing a compile error in one of the source code units.

Some Other Application is Accessing the Environment Directory

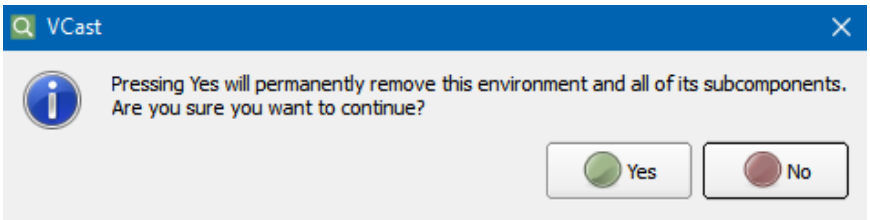
On the Windows platform, if you have a Command Prompt (DOS) window open and in the environment directory, or if any other application is accessing files in the environment directory, then the error message "Cannot backup directory" appears. Change directories in the Command Prompt, or quit the application, and then try rebuilding the environment.

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 Re-create a Deleted Environment

To rebuild a deleted environment, choose **Environment => Scripting => Run**. Find the *environment_name*.env file of the environment you deleted. Click **Open**. VectorCAST rebuilds the environment. The test case(s) are no longer present, unless you previously exported a test case script or created Regression Scripts.

```
clicast -lc ENvironment SCript Run <scriptfile>

Build a new environment from the specified script file. You can also use
*.env for <scriptfile>. CLICAST is passed a command line with each
<filename>.env found in the directory. On Linux, to avoid having the
command line get too long after expansion, you can quote the argument to
clicast, using "*.env" for <scriptfile>. This form passes CLICAST the
actual wildcard string for it to expand.
```

Other Environment Tools

To Create an Environment Script

When an environment is built, an environment script is automatically generated. It is named *environment_name*.env, and stored in the working directory.

If you delete the environment script, it can be re-created using **Environment => Scripting => Create**. The menu item is dimmed unless you have an environment open. This command enables you to create a script file that has all the necessary information to rebuild the current environment, including user globals, stubbed units, and whitebox indication. To create a script, choose **Environment => Scripting => Create**. A Save File dialog appears, prompting you for a file name. Regardless of the file name, the ENVIRO.NAME line in the script reflects the name of the currently open environment.

```
clicast -e <env> ENvironment SCript Create <scriptfile>

Create an environment script file from an existing environment. If no
extension is provided, .env is used.
```

```
clicast -lc ENvironment SCript Quick "<wildcard specification>"

Build multiple environment script files based on a wildcard specification.
For example, using the wildcard expression "*.c" (with the quotes) results
in an environment script being built for each file in the current directory
with a .c extension. This command is very powerful for quickly building
multiple test environment scripts.
```

For each unit indicated by the wildcard specification, the environment script contains the following by default:

```
ENVIRO.NEW
ENVIRO.NAME:ENV_unitname
ENVIRO.UUT:unit
ENVIRO.COVERAGE_TYPE:NONE
ENVIRO.STUB:ALL_BY_PROTOTYE
ENVIRO.SEARCH_LIST:full path to current directory
ENVIRO.TYPE_HANDLED_DIRS_ALLOWED:
ENVIRO.END
```

To Create an Environment by Running a Script

Running an environment script enables you to specify all environment build information in a text file, and then build the environment from that file.

If an environment is currently open when you choose this command, it is closed and the new one is built and opened.

When you choose **Environment => Scripting => Run**, the Open File dialog appears, prompting you to open an environment script file (.env). These files are created by VectorCAST, but sometimes you may want to copy one and slightly modify it as a text file. This command enables you to build an environment by running an environment script.

If you attempt to build an environment from a script and a directory already exists with the same name as the ENVIRO.NAME line in the script, the environment will not be overwritten and the build process will terminate.

In clicast, the error says “The directory <dir> already exists – cannot overwrite.” In the GUI, nothing happens except the last-opened env re-opens.

```
clicast -lc ENvironment SCript Run <scriptfile>

Build a new environment from the specified script file. You can also use
*.env for <scriptfile>. CLICAST is passed a command line with each
<filename>.env found in the directory. On Linux, to avoid having the
command line get too long after expansion, you can quote the argument to
clicast, using “*.env” for <scriptfile>. This form passes CLICAST the
actual wildcard string for it to expand.
```

To Recompile the Test Harness

The **Environment => Recompile => Automatic** command is used to recompile and link all of the elements of an existing VectorCAST test harness (and instrumented test harness if you have code coverage initialized). This command is useful when a compiler macro definition has changed, causing different preprocessor behavior. The test harness needs to be recompiled to reflect this new behavior.

```
clicast  -e <env> ENvironment RECompile Auto

Recompile and link all of the elements of an existing environment.
```

To Recompile the Instrumented Test Harness without Re-instrumenting It

If you have initialized code coverage for the environment, VectorCAST creates a modified version of the test harness with coverage instrumentation, and does so with every rebuild. If VectorCAST Technical Support instructs you to modify this instrumented file, the only way to recompile this file is with the Recompile Instrumented Code option (Recompile Automatic and Relink will cause the instrumentation to be performed again, thus removing your changes).

```
clicast -e <env> ENvironment RECompile Instrumented

Re-compile instrumented unit(s) and relink instrumented harness without re-
initializing coverage. This command is useful if you have made changes to
the instrumented harness that you do not want deleted.
```

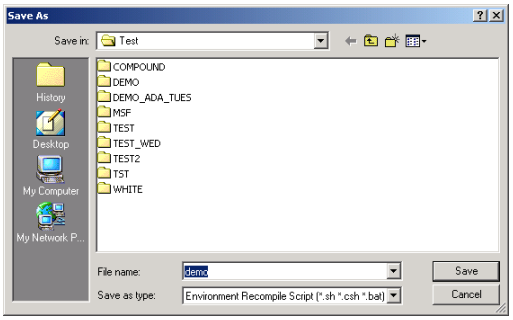
To Create a Test Harness Recompile Script

The **Environment => Recompile => Create Script** command is used to create a compile script for all of the elements of an existing VectorCAST environment. These compile scripts are operating system dependent and can always be run outside of VectorCAST if desired. You may also add any additional commands or conditional logic to the scripts to accommodate your specific requirements. In addition, this feature enables you to view the exact compile and link commands that VectorCAST uses to build the harness executable; this is especially useful to diagnose when specific compile or link options are incorrect.



Note: No actual compiling is performed by this command. The commands that would normally be executed are dumped to a file. Execution of the script is performed with the **Environment => Recompile => Run Script** command.

Select **Environment => Recompile => Create Script** from the Main menu. Enter a filename to store the compile script. If you are running under Windows, give the filename an extension **`.bat`**. If you are running under Linux, give the filename the extension **`.sh`**. Click **Save**.



You will see this dialog box when the script has finished building.




```
clicast -e <env> ENvironment RECompile Create <scriptfile>

Create a script of compile and link commands used to build harness
executable. <scriptfile> is a required argument, specifying the filename.
```

Sample Recompile Script (Windows, using MS VisualC++)

```
cd c:\cygwin\home\jml\temp\tutorial

rem # VECTORCAST_IO

CL /c /Zi /DVCAST_MAX_STRING_LENGTH=1000

/DVCAST_CLASS_INST_SHARING=VCAST_CLASS_INST_SHARING

/DVCAST_HAS_LONGLONG=VCAST_HAS_LONGLONG

/DVCAST_MICROSOFT_LONG_LONG=VCAST_MICROSOFT_LONG_LONG

/DVCAST_USE_STD_STRING=VCAST_USE_STD_STRING /Ic:\VCAST\Tutorial\cpp

B0000007.cpp

rem # USER_GLOBALS_VCAST

CL /c /Zi /DVCAST_MAX_STRING_LENGTH=1000

/DVCAST_CLASS_INST_SHARING=VCAST_CLASS_INST_SHARING

/DVCAST_HAS_LONGLONG=VCAST_HAS_LONGLONG

/DVCAST_MICROSOFT_LONG_LONG=VCAST_MICROSOFT_LONG_LONG

/DVCAST_USE_STD_STRING=VCAST_USE_STD_STRING /Ic:\VCAST\Tutorial\cpp

B0000008.cpp

rem # manager

CL /c /Zi /DVCAST_MAX_STRING_LENGTH=1000

/DVCAST_CLASS_INST_SHARING=VCAST_CLASS_INST_SHARING

/DVCAST_MICROSOFT_LONG_LONG=VCAST_MICROSOFT_LONG_LONG

/DVCAST_USE_STD_STRING=VCAST_USE_STD_STRING /Ic:\VCAST\Tutorial\cpp
```

```
S0000009.cpp
CL /c /Zi /DVCAST_MAX_STRING_LENGTH=1000
/DVCAST_CLASS_INST_SHARING=VCAST_CLASS_INST_SHARING
/DVCAST_HAS_LONGLONG=VCAST_HAS_LONGLONG
/DVCAST_MICROSOFT_LONG_LONG=VCAST_MICROSOFT_LONG_LONG
/DVCAST_USE_STD_STRING=VCAST_USE_STD_STRING /Ic:\VCAST\Tutorial\cpp
B1_switch.cpp
CL /c /Zi /DVCAST_MAX_STRING_LENGTH=1000
/DVCAST_CLASS_INST_SHARING=VCAST_CLASS_INST_SHARING
/DVCAST_HAS_LONGLONG=VCAST_HAS_LONGLONG
/DVCAST_MICROSOFT_LONG_LONG=VCAST_MICROSOFT_LONG_LONG
/DVCAST_USE_STD_STRING=VCAST_USE_STD_STRING /Ic:\VCAST\Tutorial\cpp
B4_switch.cpp
CL /c /Zi /DVCAST_MAX_STRING_LENGTH=1000
/DVCAST_CLASS_INST_SHARING=VCAST_CLASS_INST_SHARING
/DVCAST_HAS_LONGLONG=VCAST_HAS_LONGLONG
/DVCAST_MICROSOFT_LONG_LONG=VCAST_MICROSOFT_LONG_LONG
/DVCAST_USE_STD_STRING=VCAST_USE_STD_STRING /Ic:\VCAST\Tutorial\cpp
S3_switch.cpp
rem # DATA_IF_ADACAST_PKG
CL /c /Zi /DVCAST_MAX_STRING_LENGTH=1000
/DVCAST_CLASS_INST_SHARING=VCAST_CLASS_INST_SHARING
/DVCAST_HAS_LONGLONG=VCAST_HAS_LONGLONG
/DVCAST_MICROSOFT_LONG_LONG=VCAST_MICROSOFT_LONG_LONG
/DVCAST_USE_STD_STRING=VCAST_USE_STD_STRING /Ic:\VCAST\Tutorial\cpp
S0000001.cpp
rem # DATA_PKG_ADACAST
CL /c /Zi /DVCAST_MAX_STRING_LENGTH=1000
/DVCAST_CLASS_INST_SHARING=VCAST_CLASS_INST_SHARING
/DVCAST_HAS_LONGLONG=VCAST_HAS_LONGLONG
```

```
/DVCAST_MICROSOFT_LONG_LONG=VCAST_MICROSOFT_LONG_LONG
/DVCAST_USE_STD_STRING=VCAST_USE_STD_STRING /Ic:\VCAST\Tutorial\cpp
B0000002.cpp
rem # DATA_IF_ADACAST_PKG
CL /c /Zi /DVCAST_MAX_STRING_LENGTH=1000
/DVCAST_CLASS_INST_SHARING=VCAST_CLASS_INST_SHARING
/DVCAST_HAS_LONGLONG=VCAST_HAS_LONGLONG
/DVCAST_MICROSOFT_LONG_LONG=VCAST_MICROSOFT_LONG_LONG
/DVCAST_USE_STD_STRING=VCAST_USE_STD_STRING /Ic:\VCAST\Tutorial\cpp
B0000001.cpp
rem # UUT_INTERFACE_ADACAST
CL /c /Zi /DVCAST_MAX_STRING_LENGTH=1000
/DVCAST_CLASS_INST_SHARING=VCAST_CLASS_INST_SHARING
/DVCAST_HAS_LONGLONG=VCAST_HAS_LONGLONG
/DVCAST_MICROSOFT_LONG_LONG=VCAST_MICROSOFT_LONG_LONG
/DVCAST_USE_STD_STRING=VCAST_USE_STD_STRING /Ic:\VCAST\Tutorial\cpp
S0000003.cpp
rem # USER_CODE_ADACAST
CL /c /Zi /DVCAST_MAX_STRING_LENGTH=1000
/DVCAST_CLASS_INST_SHARING=VCAST_CLASS_INST_SHARING
/DVCAST_HAS_LONGLONG=VCAST_HAS_LONGLONG
/DVCAST_MICROSOFT_LONG_LONG=VCAST_MICROSOFT_LONG_LONG
/DVCAST_USE_STD_STRING=VCAST_USE_STD_STRING /Ic:\VCAST\Tutorial\cpp
B0000004.cpp
LINK /FORCE:MULTIPLE /DEBUG /OUT:UUT_INTE.EXE B0000001.OBJ
B0000002.OBJ S0000003.OBJ B0000008.OBJ B0000004.OBJ B0000007.OBJ
B1_switch.OBJ B4_switch.OBJ S3_switch.OBJ S0000009.OBJ
rem # manager
CL /c /Zi /DVCAST_MAX_STRING_LENGTH=1000
/DVCAST_CLASS_INST_SHARING=VCAST_CLASS_INST_SHARING
```

```
/DVCAST_HAS_LONGLONG=VCAST_HAS_LONGLONG

/DVCAST_MICROSOFT_LONG_LONG=VCAST_MICROSOFT_LONG_LONG

/DVCAST_USE_STD_STRING=VCAST_USE_STD_STRING /Ic:\VCAST\Tutorial\cpp

I0000009.cpp

rem # DATA_PKG_ADACAST

CL /c /Zi /DVCAST_MAX_STRING_LENGTH=1000

/DVCAST_CLASS_INST_SHARING=VCAST_CLASS_INST_SHARING

/DVCAST_HAS_LONGLONG=VCAST_HAS_LONGLONG

/DVCAST_MICROSOFT_LONG_LONG=VCAST_MICROSOFT_LONG_LONG

/DVCAST_USE_STD_STRING=VCAST_USE_STD_STRING /Ic:\VCAST\Tutorial\cpp

B0000002.cpp

LINK /FORCE:MULTIPLE /DEBUG /OUT:UUT_INST.EXE B0000001.OBJ

B0000002.OBJ S0000003.OBJ B0000008.OBJ B0000004.OBJ B0000007.OBJ

B1_switch.OBJ B4_switch.OBJ S3_switch.OBJ I0000009.OBJ

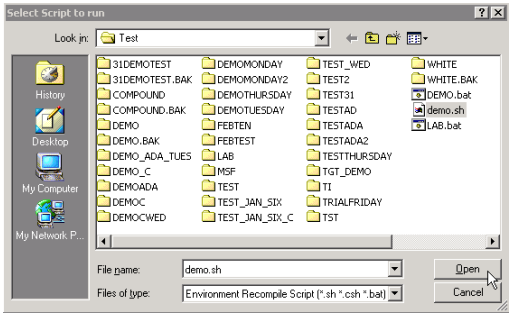
c:

cd \cygwin\home\jml\temp\tutorial
```

Recompile the Test Harness by Running a Script

The **Environment => Recompile => Run Script** command is used to execute a recompile script to update a VectorCAST environment.

If an environment is currently open, select **Environment => Recompile => Run Script**. The following dialog window opens, in which you may select the script you wish to run.



Once you have selected the script you wish to run, click Open. If you are on the Windows platform, VectorCAST opens a Command Prompt (DOS box) and runs the batch file. If you are on the Linux platform, VectorCAST runs a shell script. The Recompile is complete when you see the following text in the Message window:

```
Executing script <path to script file>
Script processing complete
```

The Scripting Log is displayed in the MDI window, and it shows the output from the recompile operation.

```
clicast -e <env> ENvironment RECompile Run <scriptfile>

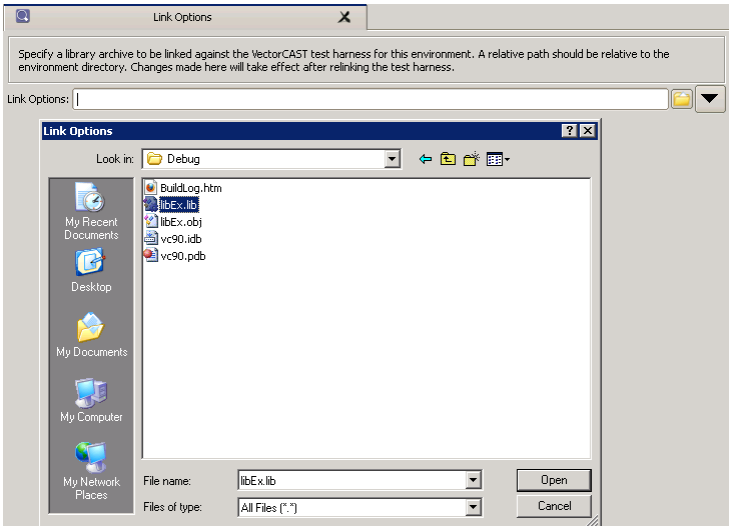
Run script of compile and link commands to build harness executable.
```

See "To Create a Test Harness Recompile Script" on page 237 for information on how to create a recompile script.

To Edit Link Options

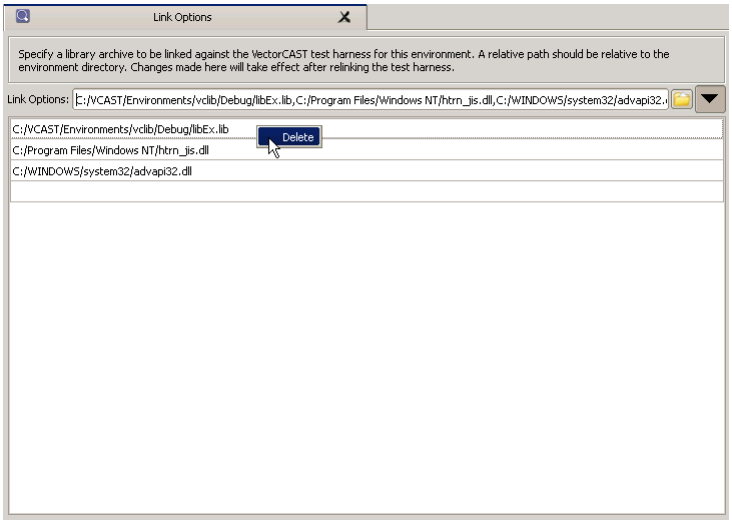
The **Environment => Edit Link Options** command enables you to specify one or more library archives to link against the VectorCAST test harness.

Click the **File Browser** button next to the **Link Options** text box. Using the Open File dialog, locate the library and then click. **Open**. To add additional libraries repeat for each library.



As each library is added, the comma separated list of selected files is updated in the **Link Options** text box.

To edit an entry in the list, double-click the entry and enter the changes. If a path is entered that is not found, the entry will be in red. To remove an entry in the list, click the **Down Arrow** located next to the File Browser button. Right-click on any entry and select **Delete** from the context menu.



Any changes made to the Link Options will take effect after relinking the test harness.

To Relink an Environment

The **Environment => Relink** command enables you to relink the object files of a previously created test environment. This command should be used any time modifications are made to the source code of the unit under test. You would then recompile your source files, and choose **Environment => Relink** to link the test harness with the new object files.

```
clicast -e <env> ENvironment RELink

Relink a previously created test environment.
```

To Refresh Type Range Data

The **Environment => Rebuild Range Data** command is used to re-calculate the ranges for all types in the code under test. This command is used when changes have been made to the original unit under test, or any type it depends on.

```
clicast -lc -e <env> ENvironment REBuild_range_data

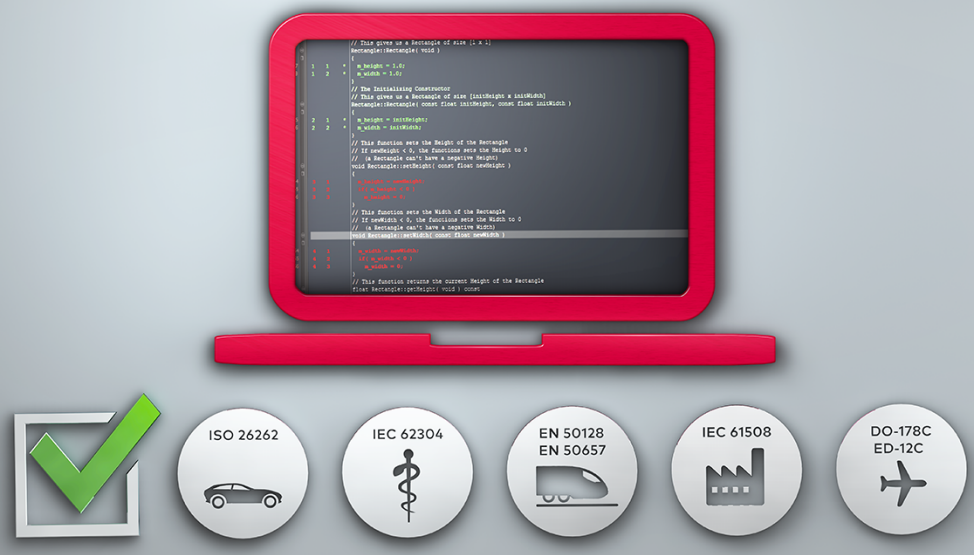
Regenerate typemark range data.
```

To Print Unit-specific Arguments

The Get_unit_options command is used to print unit-specific arguments.

```
clicast -lc -e <env> -u <unit> ENvironment Get_unit_options

Get unit-specific options.
```



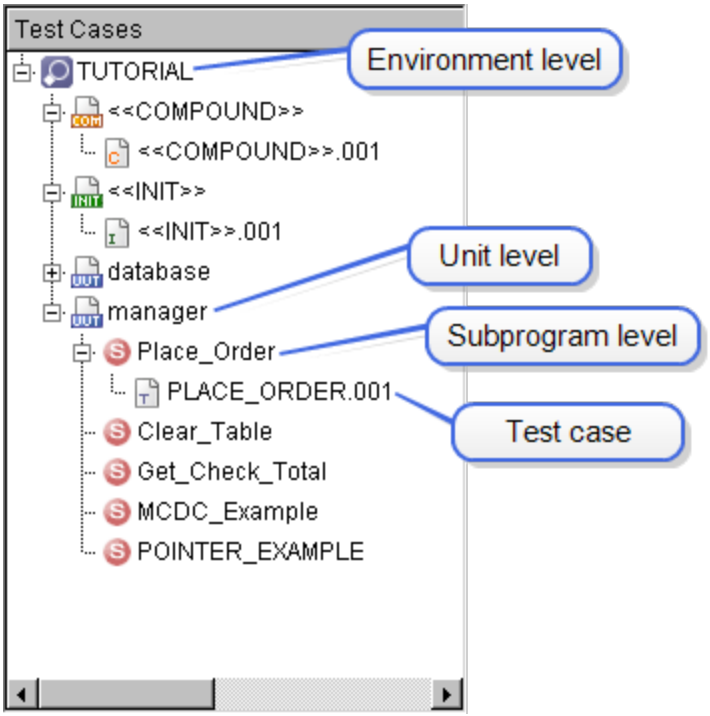
Building Test Cases

Using The Test Case Tree

The Test Case Tree Hierarchy

When an environment is opened in VectorCAST, the panel on the left (the *Test Case Tree*) displays the units, subprograms, and test cases in the environment in a hierarchy view.

Many of the menu items in the Test menu, Tools menu, and of course the right-click menu, operate on the item or items that are selected in the Test Case Tree.



At the top of the hierarchy is the environment name. Click this node if you want your next action to affect the whole environment.

Next are the <<COMPOUND>> and <<INIT>> items. Click one of these special items if you want your next action to affect only the COMPOUND test cases or INIT test cases, respectively.

The Units Under Test are listed at the next level of the hierarchy. They are listed alphabetically. Click this a unit node if you want to your next action to affect all subprograms in the unit.

The subprograms for the unit are listed below the unit in the hierarchy, in the order they are specified in the source code. Click a subprogram node if you want your next action to apply to that subprogram only.

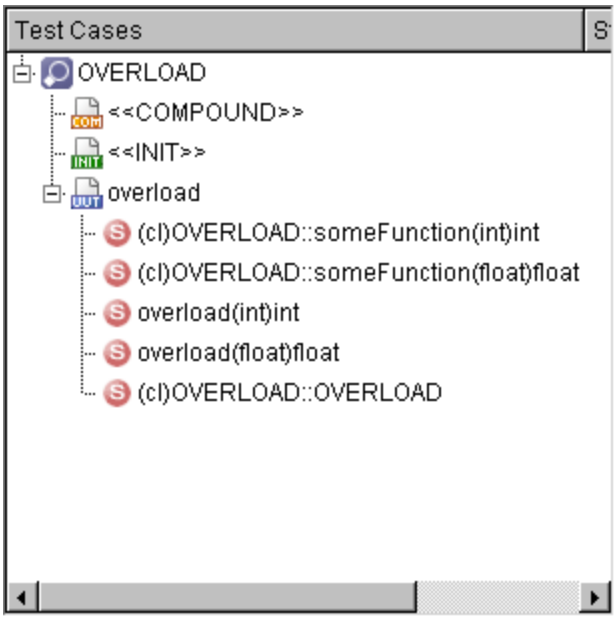
The lowest level of the hierarchy is the test case level. Click a test case if you want your next action to apply only to that test case.

When selecting items in the Test Case Tree, you can combine items at different levels using

Shift+click or Ctrl+click actions.

Naming of Overloaded Subprograms

For overloaded subprogram names, the list of parameter types will be appended to the subprogram name to differentiate between overloaded subprograms. Example: store (integer), store (float), store (boolean).



To Navigate the Test Case Tree

In the Test Case Tree, the units, subprograms, and test cases are displayed hierarchically.

To expand a node:

- Mouse: click the ⊞ icon to the left of the node
- Keyboard: after placing the keyboard focus on that node, press the right-arrow key
- Menu: right-click the node and choose **Expand All Children**

To collapse a node:

- Mouse: click the ⊟ icon to the left of the node
- Keyboard: press the left-arrow key
- Menu: right-click the node and choose **Collapse All Children**

Additionally, use the keyboard's up- and down-arrow keys traverse the expanded nodes. You can also type the first few letters of an expanded node name to jump to that node.

To Multi-Select Test Cases

To select one test case in the Test Case Tree, simply click it. To select more than one, select one, then, holding down the Ctrl key, select the other test cases you want to execute, duplicate, delete, or modify. You can select a range of test cases by holding down the Shift key while clicking the second test case. Or, you can drag your cursor over a span.

The same process is used to multi-select subprograms, units, or a mixture of test cases, subprograms, and units.

Types of VectorCAST Test Cases

There are two types of VectorCAST test cases: simple test cases and compound test cases. A *simple test case* corresponds to a single invocation of a UUT. Simple test cases are primarily used to test the processing of a single subprogram within a UUT. The simple test case data is loaded, the subprogram being tested is invoked and results are captured. <<INIT>> test cases are a type of simple test case. They invoke the test harness, but do not call any subprograms. They are used primarily to perform initialization or to check final values of global data.

A *compound test case* is a collection of simple test cases that perform a series of calls. A compound test case provides the ability to invoke a UUT multiple times with a variety of data within a single execution. Each test case in a compound test case can be executed one or more times. Compound test cases can be used to test processing which is dependent on multiple calls to different procedures of a UUT. For example, you may have processing that implements a database with one subprogram to write to the database, and another subprogram to read from the database. In this case a compound test could be developed to invoke a call to the "write" subprogram, followed immediately by a call to the "read" subprogram. In this way you can verify the integrity of the stored data.

When executing a compound test case the Event header contains information about the compound test case that is executing, the slot number executing, its name, and its iteration. For example, the following Event header is displayed for event 1 when <<COMPOUND>>.001 is executed:

```
Event 1
<<COMPOUND>>.001 Slot 1 ((CL)MANAGER::PLACEORDER.001) Iteration 1
```

All simple test cases are associated with a subprogram of a Unit Under Test. Prior to building simple test cases, it is necessary to select a subprogram to test. The test cases built for this subprogram are applicable to this subprogram only.

When building a new test case, you may assign values to parameters of the subprogram under test, any parameters of the stubbed dependent subprograms, and any global data objects.

Values are entered for individual scalar data items. VectorCAST takes you through a simple-to-use tree expansion of the type mark for each data item until a scalar data type is encountered. You are then able to enter a value for the item directly, or double-click to bring up a dialog box that is specific to that scalar type.

VectorCAST can generate the following types of test cases:

- **Min Mid Max** - Min Mid Max tests stress a function at the bounds of the input data types. C and C++ code often will not protect itself against out-of-bound inputs. The engineer often has some functional range in mind and does not protect against out of range inputs.

- **Partitioned** - Partitioned tests create "partitions" for each data type and test the min and max value from each partition. The assumption is that values from the same partition will stimulate the application in a similar way.
- **MC/DC** - MC/DC tests use the MC/DC analysis to examine the unique paths that exist through a procedure. MC/DC tests can automatically create a high level of path coverage.
- **Basis Path** - Basis Path tests use the basis path analysis to examine the unique paths that exist through a procedure. Basis Path tests can automatically create a high level of branch coverage.

To Open a Test Case for Editing

To open a test case in the TestCase Editor, select a test case name, right-click, and choose **Open Test Case**. Or, double-click the test case name.

The keyboard shortcut for Open Test Case is **Ctrl+Return**.

Test Case Naming

When building a new test case, a name is automatically created based on the name of the subprogram and a unique number:

Name of the Subprogram under test.number

For example, PLACE_ORDER.008 is the eighth test case for the subprogram Place_Order.

The two portions of the automatically generated name are separated by a period ('.'). There is no limit for the length of test case names.

The automatically generated name may be edited or completely replaced by right-clicking on the test case name and choosing **Rename**, as described in "To Rename a Test Case" on page 248.

To Rename a Test Case

To rename a test case, select a test case, right-click and choose **Rename**. An edit box surrounds the old name. You can type or paste a new name in. There is no limit for the length of test case names.

Test case names must contain only alphanumeric characters or punctuation (i.e., hyphens, underscores, or periods). Percent, backslash and single or double quotes are not supported and cause a message to pop up and the new name to be discarded.

Press **Enter** to finalize the name change.

If you rename a test case that is part of a compound test case, VectorCAST renames it in the compound test automatically.

To Sort Test Cases Alphabetically

When a new test case is inserted, by default it is positioned as the last test case for that subprogram. When importing a test script, the test cases are positioned according to the order in the test script.

To sort the test cases alphabetically (for each subprogram), choose the **Tools => Options dialog, GUI**, and set the option “Alphabetize test cases” on. As a result, as each test case is created or imported, it is positioned in the list alphabetically, for each subprogram.

To Sort the Test Case Tree

The Test Case Tree can be optionally sorted as originally processed by VectorCAST or it can be sorted in alphabetical ascending order.

The following UUT source code will be used to illustrate sorting the Test Case Tree.

```
#include "FourFunctions.h"

int add(int a, int b)
{
   return a+b ;
}

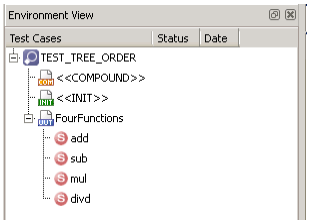
int sub(int a, int b)
{
   return a-b ;
}

int mul(int a, int b)
{
   int i = 0 ;
   int result = 0 ;

   for( i = 0 ; i < a ; i++)
   {
      result += b ;
   }
   return result ;
}

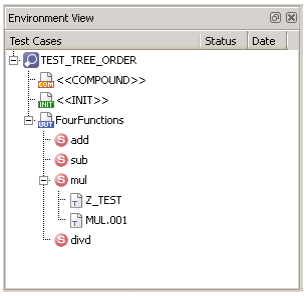
int divd(int a, int b)
{
   return a/b ;
}
```

Subprograms in the Test Case Tree are initially ordered by VectorCAST in the same order as they appear in the source file.

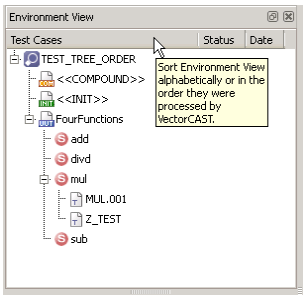


When the tree is sorted as originally processed by VectorCAST and a new test case is inserted, by default it is positioned as the last test case for that subprogram. In the example shown below, a test case is added to mul and renamed to Z_TEST. Another test case is then added and has the default

name, MUL.001. The last test case created is the last test case in the tree for that subprogram.



To order both subprograms and their associated test cases alphabetically, click the **Test Cases** column heading of the Test Case Tree



When the Test Case Tree is sorted alphabetically and a test case is renamed, test cases will be resorted in alphabetical order after Enter is pressed.

The sort order can be toggled back to the original VectorCAST processing order by clicking **Test Cases** again. When an environment is closed, the last sort order selected is remembered and will be restored when the environment is reopened.

To Duplicate a Test Case

Once a test case is built and saved, the data that is contained in it can be copied to another test case. This allows you to clone an existing test case as the basis for a new test.

Select a test case to duplicate, right-click, and choose **Duplicate Test Case**. The new test case is automatically named based on the name of the original test case.

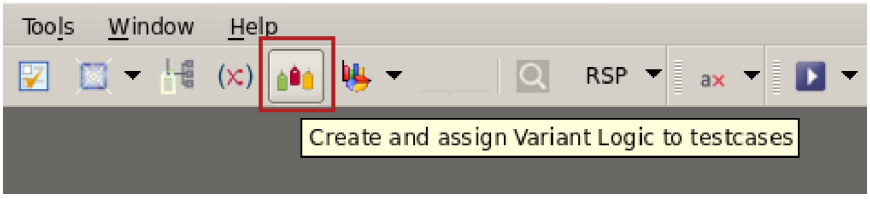
For example, if you duplicate a test case for the subprogram Place_Order named STEAK$14, then the new test case takes the name STEAK$14.001.



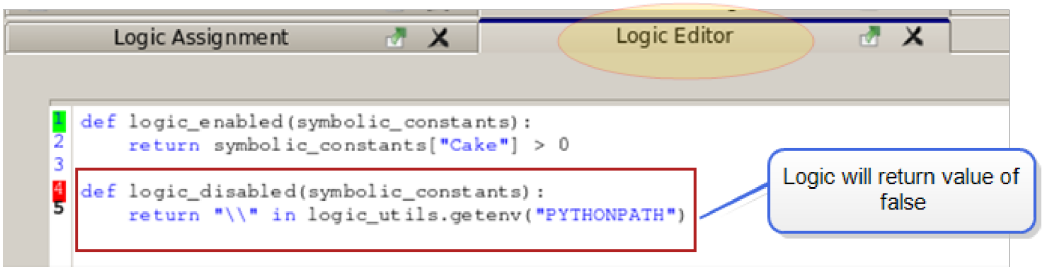
To Enable/Disable a Test Case

Test cases are enabled by default. To disable a test case, perform the following actions:

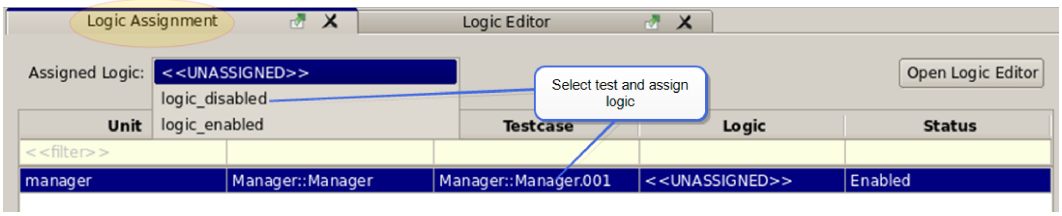
1. Select the Variant Logic toolbar button.



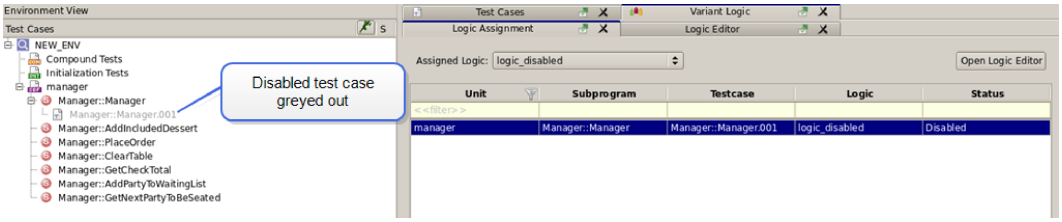
2. Click inside the Logic Editor and write a logic which will result in a return value of False. An example logic is shown below.



3. Click the **Save** toolbar button and select the Logic Assignment tab. Click on the Assignment Table row which contains the desired test case. Click on the Assigned Logic selector and choose Logic.



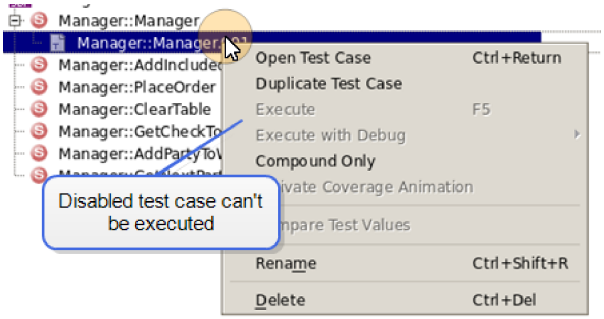
4. Click the **Save** toolbar button.



Disabled test cases are greyed out in the Test Case tree and they cannot be executed using any of the following methods:

- the Test Case tree right-click menu,
- the **Execute** toolbar button,

> Selecting **Test => Batch Execute All** from the Menu Bar, or
> CLICAST execute commands.



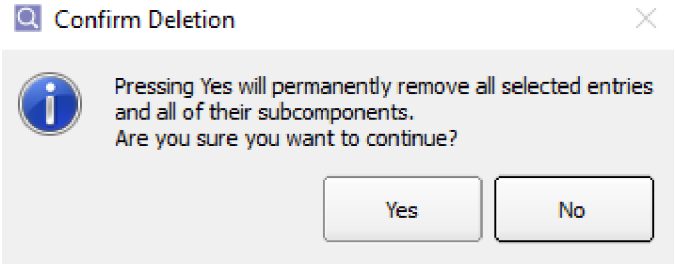
Note that the Test Case Editor for a disabled test case can still be opened and is fully usable. Furthermore, when at least one test case is enabled in the environment, the **Test => Batch Execute All** and **Execute** Menu bar options are enabled.

If a disabled test is re-assigned to an enabled logic or removes its assignment, akin to <<UNASSIGNED>>, upon saving assignment changes, the test is enabled and executable.

To Delete a Test Case

To delete a test case, select a test case in the Test Case Tree. Right-click and choose **Delete**.

The following confirmation dialog box appears:



Note that if a test case is used in a compound test case, then deleting the test case also affects the compound test case. Click **Yes** to delete the test case and remove it from any compound test cases. Click **No** to cancel the delete operation.

```
clicast -lc -e <env> -u <unit> -s <sub> -t <test> TESt Delete [ALL]

Delete the test case(s) specified via the unit, subprogram, and testcase
command line parameters. If no parameters are specified, the user must
specify ALL to confirm deletion of all testcases in the environment. If a
test being deleted is contained in a compound test, the user must specify
ALL to confirm deletion of the test in all locations.
```

To Delete All Test Cases from a Subprogram

To delete all test cases in a subprogram, select a subprogram in the Test Case Tree. Right-click and choose **Delete**.

```
clicast -lc -e <env> -u <unit> -s <sub> -t TESt Delete [ALL]

Delete the test case(s) specified via the unit and subprogram command line
parameters. If a test being deleted is contained in a compound test, the
user must specify ALL to confirm deletion of the test in all locations.
```

To Delete All Test Cases from a Unit

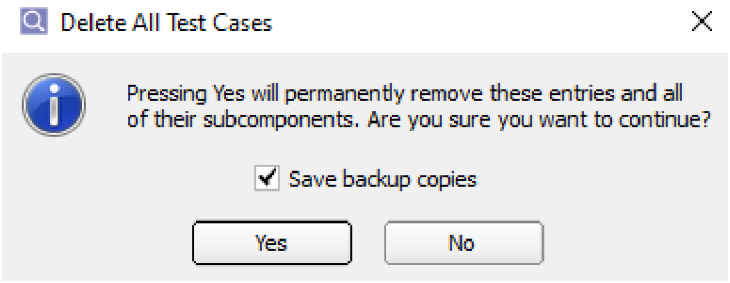
To delete all test cases from a unit, select a unit in the Test Case Tree. Right-click and choose **Delete**.

```
clicast -lc -e <env> -u <unit> TESt Delete [ALL]

Delete the test case(s) specified via the unit command line parameters. If
a test being deleted is contained in a compound test, the user must specify
ALL to confirm deletion of the test in all locations.
```

To Delete All Test Cases in the Environment

If you wish to delete all test cases from all subprograms, choose **Test => Delete All**. The following confirming dialog appears.



The “Save backup copies” option is selected by default as an added failsafe to prevent accidental deletion of test cases. This “backup script” is saved in the environment directory and named TEST_DELETE.tst.

Alternatively, right-click the top node of the Test Case Tree hierarchy (the environment name), and choose **Delete**. Doing so deletes the test cases from the environment level, which includes all test cases. You are asked to confirm the delete operation, but a backup file is not created.

```
clicast -lc -e <env> TESt Delete [ALL]

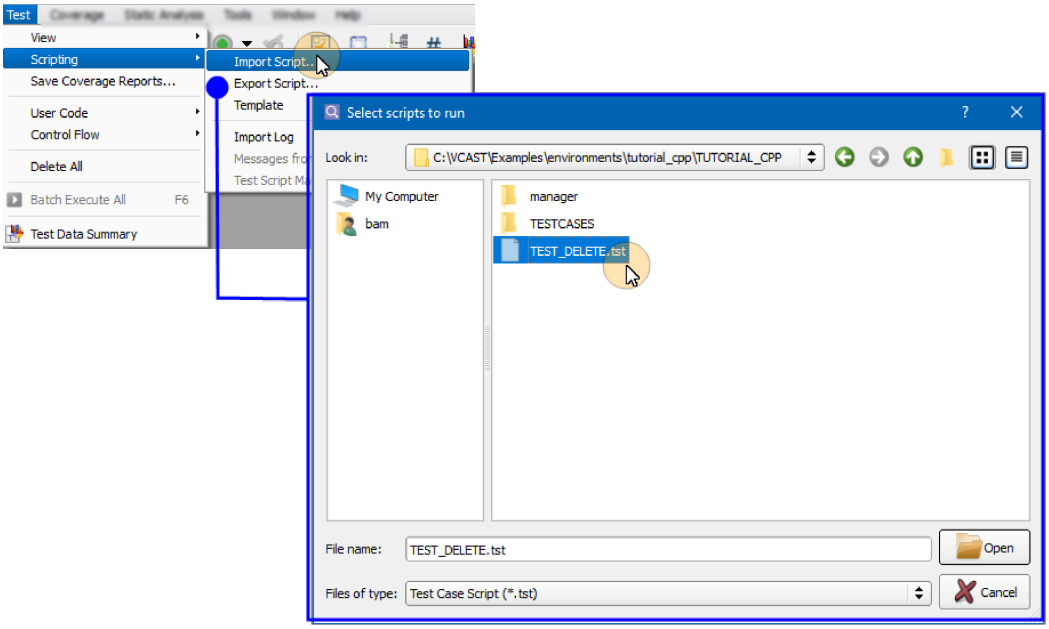
Delete all test cases in the environment. The user must specify ALL to
confirm deletion of all testcases in the environment.
```

See "To Restore Deleted Test Cases" on page 254 for information on restoring test cases.

To Restore Deleted Test Cases

Only test cases that were deleted with the **Test => Delete All** command can be restored.

The Delete All Test Cases dialog has a checkbox labeled “Save backup copies.” By default, this checkbox is selected. When selected, a backup test case script is automatically generated to preserve the deleted test cases. This test script file is called **`TEST_DELETE.tst`** and is created inside the environment directory. If you need to retrieve the deleted test cases, simply import the test script (**Test => Scripting => Import Script => env_dir/TEST_DELETE.tst**) and all deleted test cases will be restored.



To View the Source for a Subprogram

To open the source file for a unit or subprogram, right-click the unit or subprogram node, and choose **Open Source**. The source file opens in a tab in the Text Editor group. If this action is performed on a subprogram node, then the cursor is placed at the beginning of the subprogram.

The keyboard shortcut for Open Source is **Ctrl+Return.**

To View Coverage for a Unit or Subprogram

To open the Coverage Viewer for a unit or subprogram in the Test Case Tree, right-click the unit or subprogram node, and choose **View Coverage**. The Coverage Viewer opens. If this action is performed on a subprogram node, then the Coverage Viewer is scrolled to the top of the subprogram.

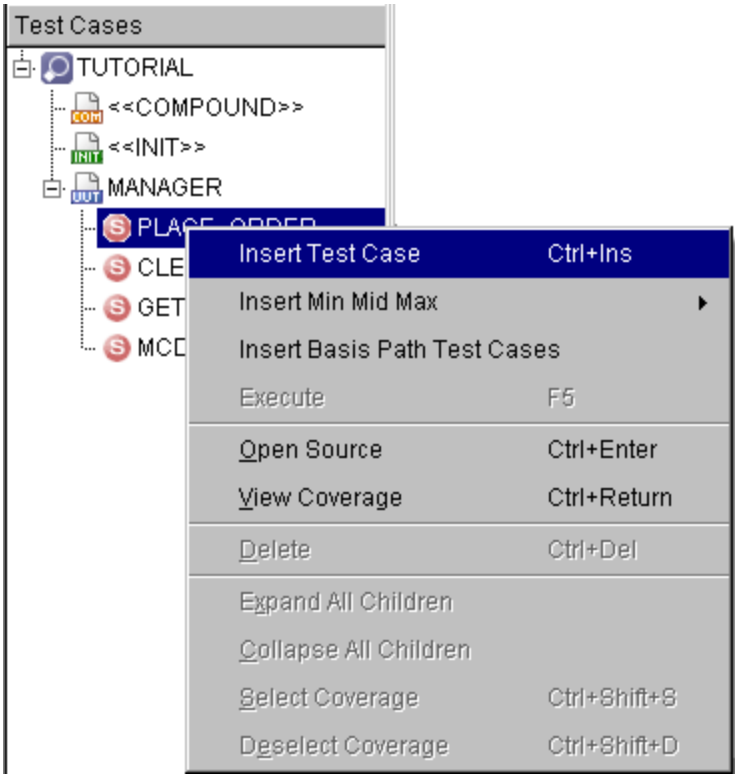
The keyboard shortcut for View Coverage is **Shift+Return**.

Simple Test Cases

To Insert a Test Case for a Subprogram

Select a subprogram, or multi-select several subprograms. Right-click, and choose **Insert Test Case**. A test case is inserted for each subprogram selected. By default, its name is "*subprogram*.001", where *subprogram* is the name of the subprogram.

The keyboard shortcut for Insert Test Case is **Ctrl+Insert**.



See"To Rename a Test Case" on page 248 on page for information on renaming a test case.

To Insert an <<INIT>> Test Case

The <<INIT>> test case enables you to initialize global data within the Unit(s) Under Test. This test case invokes the test harness without calling any subprograms. Any initialization (global data initialization, class object construction) will take place before the test driver "main" is called.

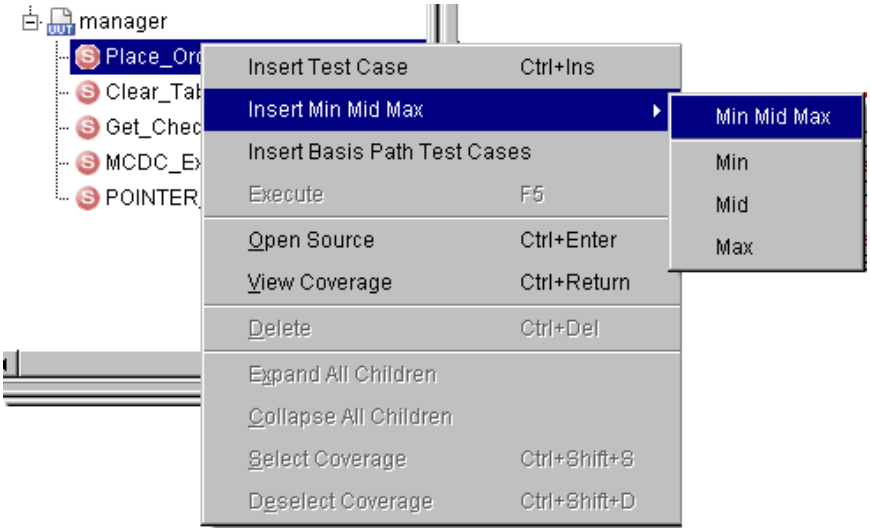
Another useful feature for the <<INIT>> test case is as a setup step for compound test cases. You can set the <<INIT>> test case to initialize global data, and then include that test case as the first slot in a compound test. Doing so enables large amounts of initialization data to be entered one time for multiple compound tests.

To Create a Min, Mid, or Max Test Case

The Min-Mid-Max command provides an automated test case generation capability for creating test

cases that set all parameter and global data values (excluding User Globals) to their minimum, maximum, or mid-point values. Min, Mid, and Max test cases are useful when unit testing because they often uncover hidden assumptions present in the code under test. For example, using an integer-typed data item to index into an array with only 10 elements without checking that the index value is within range.

To create all three types, click a node in the Test Case Tree (subprogram, unit, or environment). Right-click and choose **Insert Min Mid Max** from the drop-down menu.

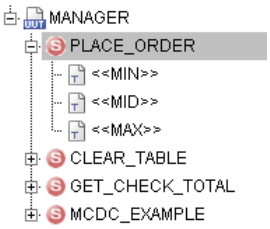



```
clicast -e <env> [-u <unit> [-s <sub>]] TESt Minmidmax

Build the <<MIN>>, <<MID>>, and <<MAX>> special test cases for the
environment, a specified unit, or a specified subprogram.
```

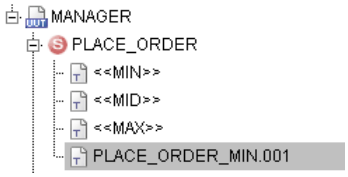
To create only one type (min, mid, or max), right-click and choose **Insert Min Mid Max** from the popup menu, and then choose **Min** (or **Mid** or **Max**) from the submenu.

Min-Mid-Max test cases are given the special names <<MIN>>, <<MID>>, and <<MAX>> in the Test Case Tree. These special test cases cannot be edited, but you can derive new test cases from them.



To derive a test case from one of the special <<MIN>>, <<MID>>, or <<MAX>> test cases, double-click on it. A new test case is created. For example, double-clicking on <<MIN>> that appears below the subprogram Place_Order creates the test case PLACE_ORDER_MIN.001. In this test case, all input values for the subprogram and stubs are already set to the minimum value, but you can override

specific values by entering data.



When you view the Test Case Data Report, it indicates that all *input* data values are set to min, mid, or max, as shown in the following example.

Test Case Data

Test Case	(cl)Manager::PlaceOrder_MIN.001
File Name	C-000006.DAT

All Data Values Set to: "<<MIN>>"

Input Test Data

This test has no input data.

Expected Test Data

This test has no expected data.

Test Case / Parameter Input User Code

This test has no input user code.

Test Case / Parameter Expected User Code

This test has no expected user code.

To Insert Basis Path Test Cases

To automatically generate the test cases needed to fulfill all the basis paths, click a node in the Test Case Tree (subprogram, unit, or environment). Right-click and choose **Insert Basis Path Test Cases**.

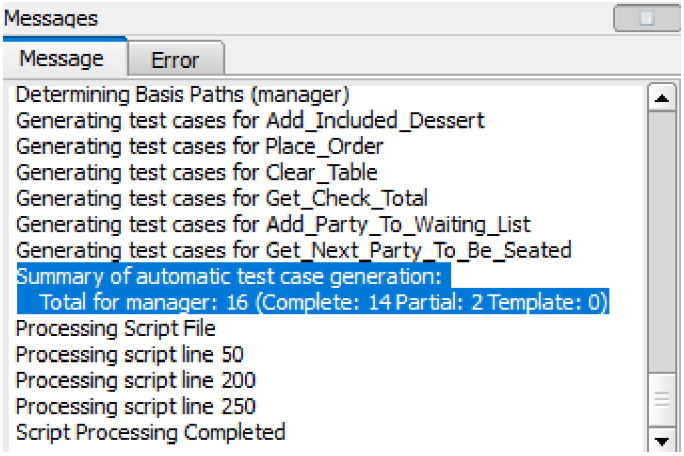
Test cases are added to the environment, named *BASIS-PATH-001,* -002, -003, etc., which correspond to the basis paths for the subprogram. Because of unreachable paths and because some conditions may be based on intermediate computations, the test cases that VectorCAST creates may not be complete. After the basis path test cases are created, a Test Script Log is displayed, showing the test case processing.

The Message window displays a summary of the automatic test case generation. The generated test cases are complete, partial, or template tests.

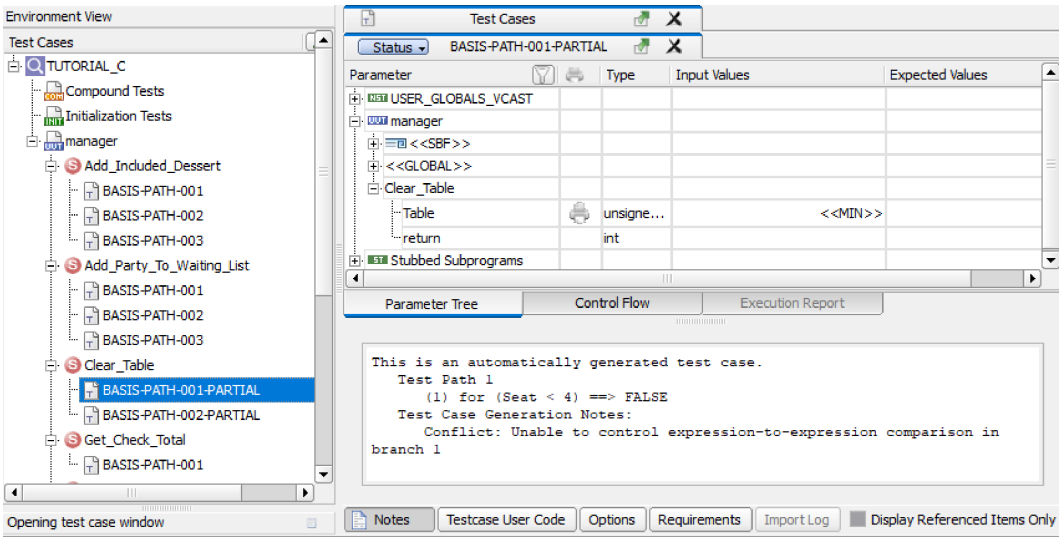
- Complete – the test case was completely generated, with an input value for each data item necessary to traverse the test path
- Partial – the test case was partially generated. The test case's name has "-PARTIAL" appended to it. The test case's Notes tab contains a full description of which portions of the test case are

not complete and why. Example: a dependency on an intermediate computation (`c=foo(); if(c)`).

> Template – VectorCAST was unable to determine any information necessary to traverse the test path. The test case's name has "-TEMPLATE" appended to it. As with partial test cases, the test case Notes tab contains a full description of the reasons.



There are a number of reasons why VectorCAST may not be able to determine input values: computations embedded in the conditional, local variables, etc. In cases where complete test cases cannot be generated, the test case Notes tab describes any missing data that you must `set` up to implement the test.



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. In cases where a conditional is based on an intermediate computation, VectorCAST is not able to control the decision point. The following is an example of this case:

Example 1:

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

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 at least once. The basis path calling for the loop to not be entered cannot be satisfied.

```
clicast -e <env> [-u <unit> [-s <sub>]] TOols Auto_Test_Generation
[<outputfile>]

Generate a test script containing one testcase for each basis path in the
environment for the specified unit, subprogram in that unit, or whole
environment.
```

See also "Understanding Basis Paths" on page 506.

Troubleshooting Min, Mid, Max Test Cases

Segmentation Violation Caused by Min, Mid, or Max Input Values

When creating Min, Mid, or Max test cases, be aware that *all input* values are automatically set by VectorCAST. In the Tutorial code, for example, the test case PLACE_ORDER_MAX.001 causes a constraint error because the maximum value for TABLE, SEAT, CHECK_TOTAL, and NUMBER_IN_PARTY are too large when they are used as array indices.

Event 2 - Stubbed Get_Table_Record
UUT: manager.c
Subprogram: Get_Table_Record

Parameter	Type	Actual Value	Expected Value
Table	unsigned short	65535	
return			
Is_Occupied	enum	v_true	
Number_In_Party	unsigned short	65535	
Designator	char	'\177'	
Wait_Person	string	"""	
Order			
[0]			
Soup	enum	CHOWDER	
Salad	enum	GREEN	
Entree	enum	PASTA	
Dessert	enum	FRUIT	
Beverage	enum	SODA	
[1]			
Soup	enum	CHOWDER	
Salad	enum	GREEN	
Entree	enum	PASTA	
Dessert	enum	FRUIT	
Beverage	enum	SODA	
[2]			
Soup	enum	CHOWDER	
Salad	enum	GREEN	
Entree	enum	PASTA	
Dessert	enum	FRUIT	
Beverage	enum	SODA	
[3]			
Soup	enum	CHOWDER	
Salad	enum	GREEN	
Entree	enum	PASTA	
Dessert	enum	FRUIT	
Beverage	enum	SODA	
Check_Total	float	3.40282e+038	

Event 3 - Signal Raised: Segmentation Violation
This event causes a failure.

UUT Returned control to Driver, end of test case

Result - FAIL

Signals 1 FAIL

To solve this problem, click the Parameter Tree tab to return to the Test Case Editor. If you know that the subprogram under test would never get a value that is too large or too small, then enter your own input values for any parameters that may be getting values that are too large or too small. On the other hand, you may want to modify your source code to handle values that are too large or too small.

See also "To Incorporate Source Code Changes into Environment " on page 232.

Compound Test Cases

Building a Compound Test Case

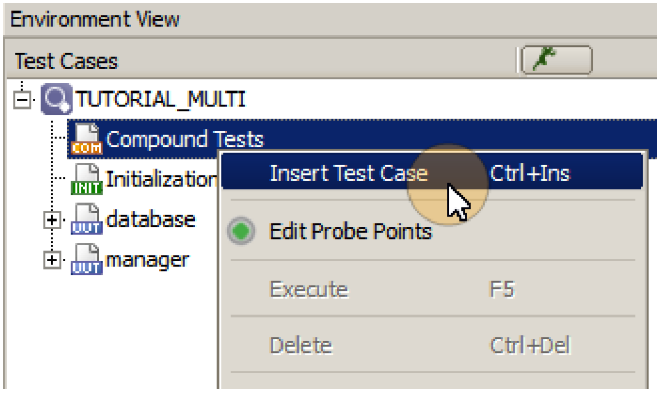
The following sections describe how to build compound test cases. Because compound test cases are a sequence of simple test cases linked together, at least one simple test case must be defined before a compound test case can be built.

A compound test case consists of a series of "slots." Each slot contains the following information:

- A slot number
- The name of the unit
- The name of a subprogram
- The name of a simple test case
- The number of times to invoke the subprogram with the test case data

To create a new compound test case:

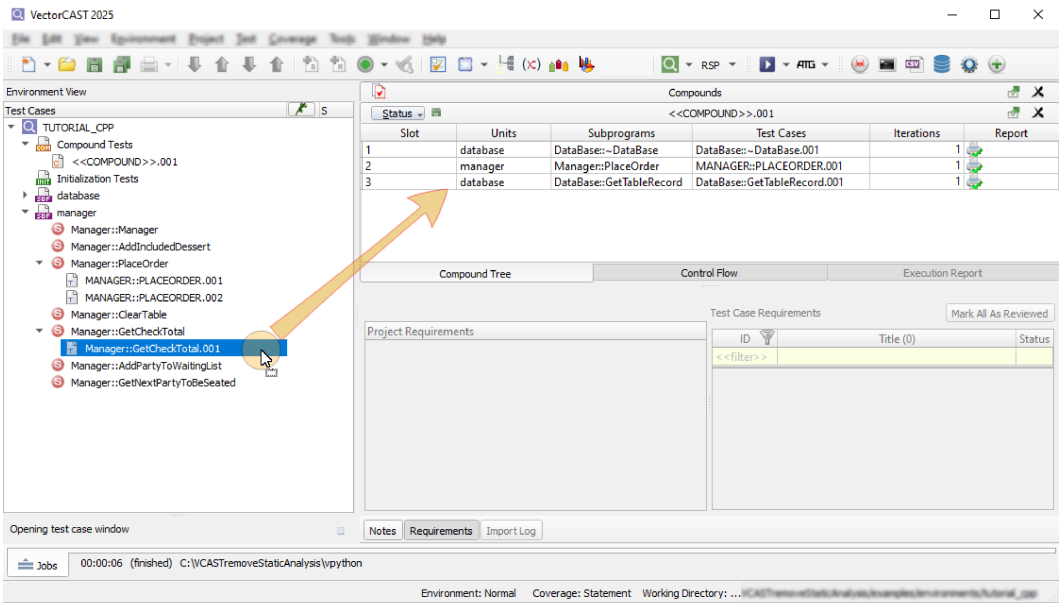
1. Right-click on the Compound Tests node to insert a new test case



   The name defaults to <<COMPOUND>>.001. You can change this by right-clicking on the test case name at any time and selecting **Rename** from the context menu.

2. With your mouse, click-and-drag one or more test cases to the Compound Test Editor located in the MDI area. You can drag and drop a test case to a new position within the Compound Test

Editor.



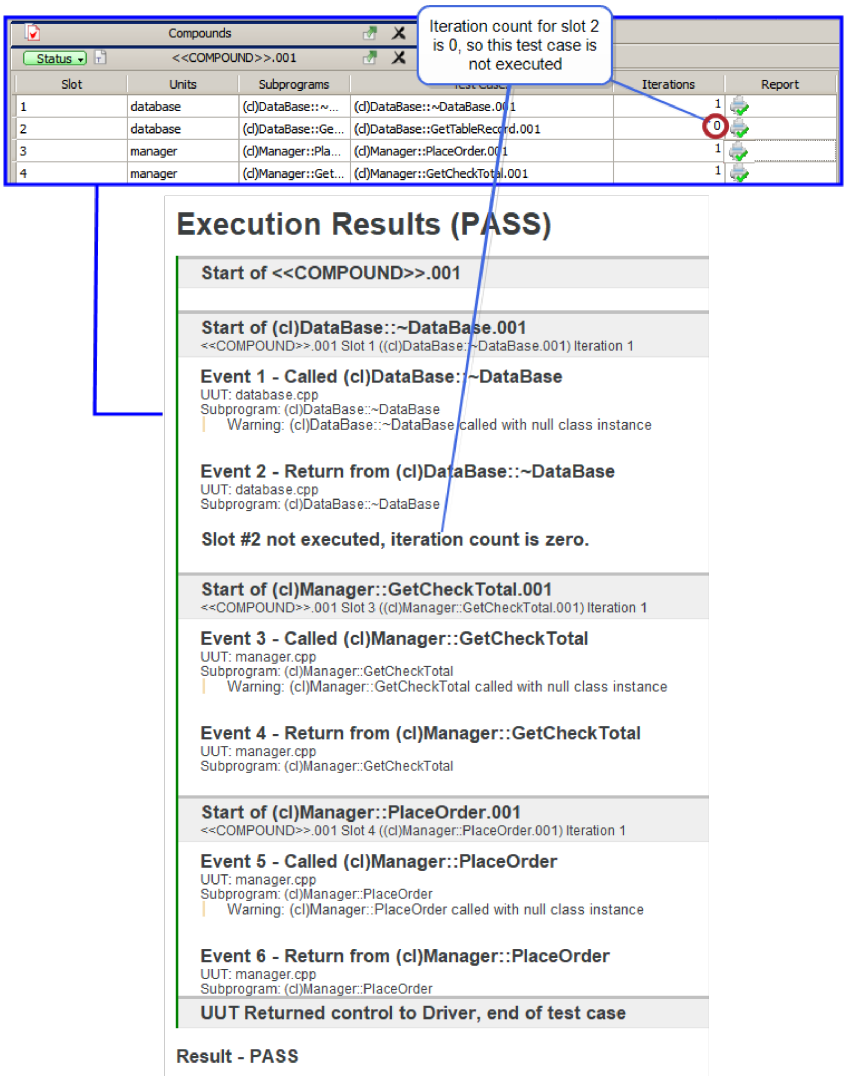
Each test case in a compound is called a *slot*. The unit name, subprogram name, test case name, number of iterations, and report status is displayed for each slot. The slots are executed in order when the compound test case is executed. See " To Reorder Test Cases in a Compound" on page 265 to change the execution order.

To Specify the Number of Iterations of a Test in a Compound

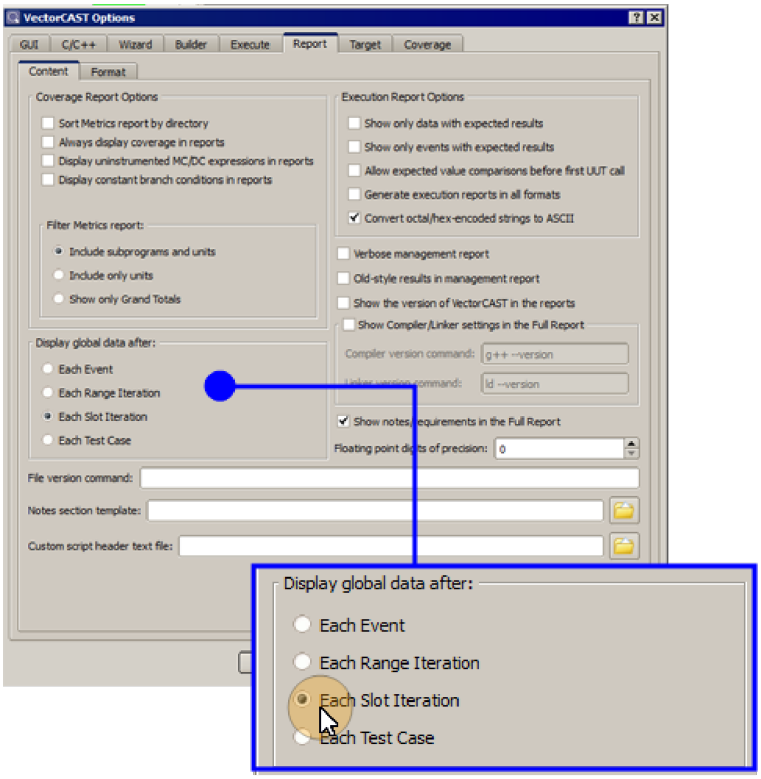
By default, each test case in a compound is executed once. Increasing the number of iterations causes the test case to execute repeatedly before going to the next test case in the compound.

The maximum number of iterations in a compound slot is 999,999.

A setting of 0 Iterations causes the test case to be skipped during execution of the compound. A notation, "Slot <#n> not executed, iteration count is zero", is added to the Execution Report.



If you want global data to be displayed in the Execution Report after each slot iteration, select **Tools => Options** from the Menu Bar to open the Options dialog. Select the Report tab, Content sub-tab and go to the **Display global data after** group. Click the radio button next to **Each slot iteration** and select the **Apply** button.



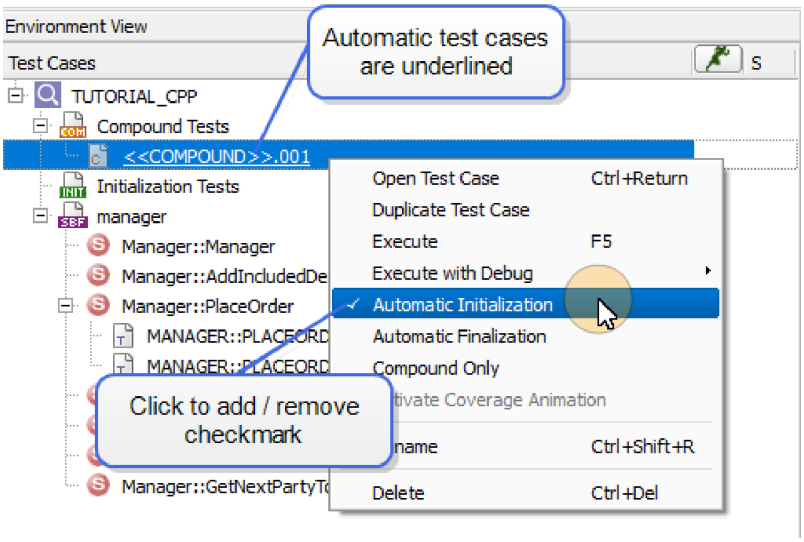
See also "Compound Test Case Execution and Reports" on page 398.

Compound as Automatic Initialization / Finalization

A compound test can be set as an Automatic Initialization or Finalization test which runs either before or after other test cases. The Automatic Initialization test is invoked at the beginning of each test execution and is useful for setting up data for the test case that follows it. The Automatic Finalization test is invoked at the end of test case execution and is useful for checking data that the preceding test may have changed.

See the sections "TEST.AUTOMATIC_FINALIZATION" on page 674 and "TEST.AUTOMATIC_INITIALIZATION" on page 675 for further information.

In this example, <<COMPOUND>>.001 has been marked as an Automatic Initialization test. Tests set as automatic tests have their name underlined in the Test Case Tree.



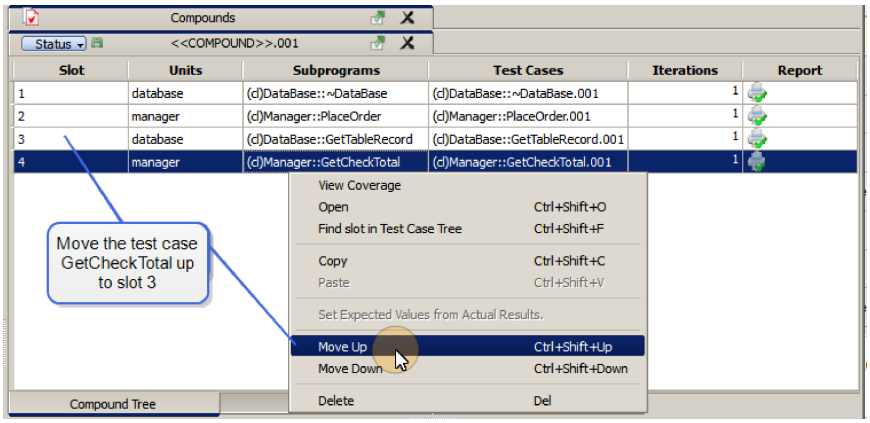
To Reorder Test Cases in a Compound

In a compound test case, the individual test cases, or *slots*, are executed in order from top to bottom. To change the order in which the tests are executed, you change the order that they are listed in the editor. To do this, right-click a test case, and choose **Move Up** or **Move Down**.

The keyboard shortcut for Move Up is **Ctrl+Shift+Up Arrow**.

The keyboard shortcut for Move Down is **Ctrl+Shift+Down Arrow**.

Alternatively, you can drag and drop a test case to a new position within the Compound Test Editor. Multi-selection drag and drop is supported. When you select multiple slots and drop them in a new position, the slots are dropped in the order in which they were selected. In this way you can either duplicate slots or duplicate and re-order slots at the same time.



To Open a Test Case from a Compound

To open an individual test case from a Compound test case, double-click the test case. Alternatively, right-click the test case and choose **Open**.

The keyboard shortcut for Open is **Ctrl+Shift+O**.

The test case opens in the Test Case Editor, located below the Compound Test Editor. Once changes are made to the individual test case and saved, they are part of the Compound too.

To Delete a Slot from a Compound

To delete a test case from a Compound, right-click the test case, and choose **Delete**. Selecting Delete removes the test case entirely from the compound test case; it does not delete the test case from the environment.

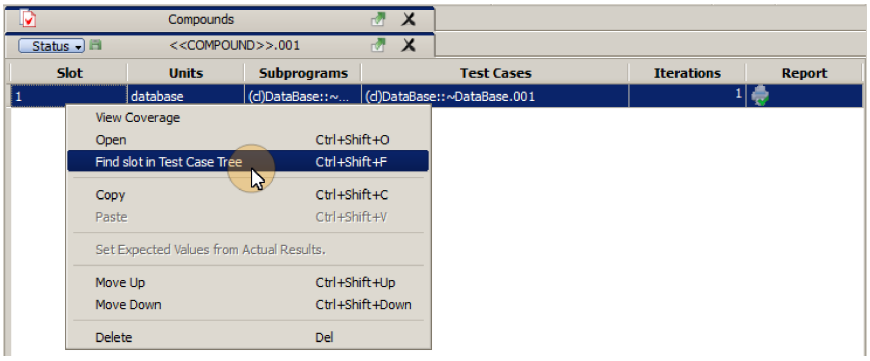
The keyboard shortcut for Delete is **Del**.

To View Coverage for a Test Case from a Compound

To open the Coverage Viewer for a particular subprogram from a Compound, right-click the test case and choose **View Coverage**. The Coverage Viewer opens, scrolled to the top of the particular subprogram referenced by that test case.

To Locate a Slot in the Test Case Tree

To find the test case in the Test Case Tree that corresponds to the slot selected in the Compound Test Case Editor, right-click on the slot in the Compound Test Editor and then select **Find slot in Test Case Tree** from the context menu.



The keyboard shortcut to locate a slot is **Ctrl+Shift+F**.

The slot is highlighted in the Test Case Tree.

To Search for Text in the Compound Test Editor

The Compound Test Editor supports Search to find text within a slot. To search within the Compound Test Editor, select **Edit => Find** or press **Ctrl+F**. A Find Banner appears at the bottom of the Editor.

Type the text you are looking for in the **Find** text edit box and press **Enter**.

The search looks in all columns (slot number, unit name, subprogram name, test case name, and iterations). So searching for the number 3 will find both slot 3 and any slot with Iterations set to 3, for example.

For more details, see "To Search for Text Using the Find Banner" on page 89.

Compound Only Test Cases

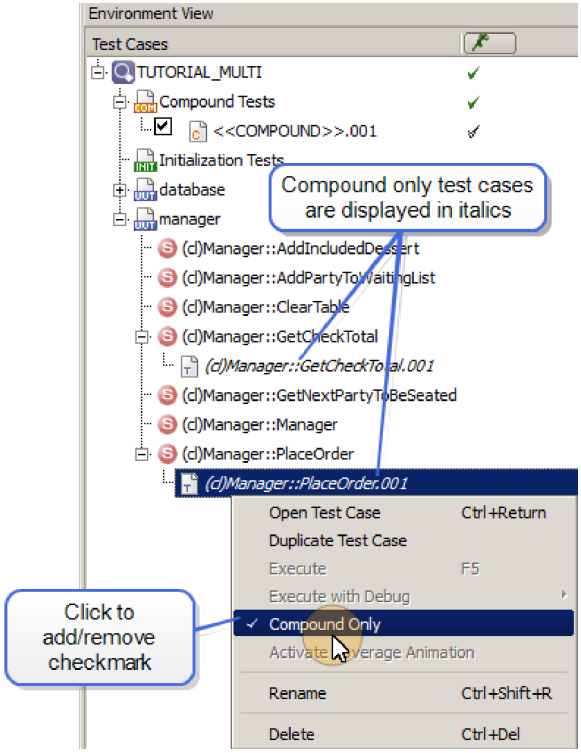
Compound Only test cases are only executed from within Compound tests, and are not executed individually in normal or batch mode. Both individual test cases as well as compound test cases can be set to "Compound Only". To specify that a test case is "Compound Only," right-click the test case and choose **Compound Only** from the context menu.

```
clicast -lc -e <env> -u <unit> -s <sub> -t <testcase> TESt Compound_only
true | false

Change the specified test case to a "Compound Only" test case or back
again. "Compound Only" test cases are not executed in normal or in batch
mode, but only as part of a compound test case.
```

When a test case is "Compound Only" its name is italicized, and there is a check mark next to Compound Only in the right-click context menu.

To specify that a test case is no longer Compound Only, right-click the test case and choose **Compound Only** again, such that the check mark is removed.



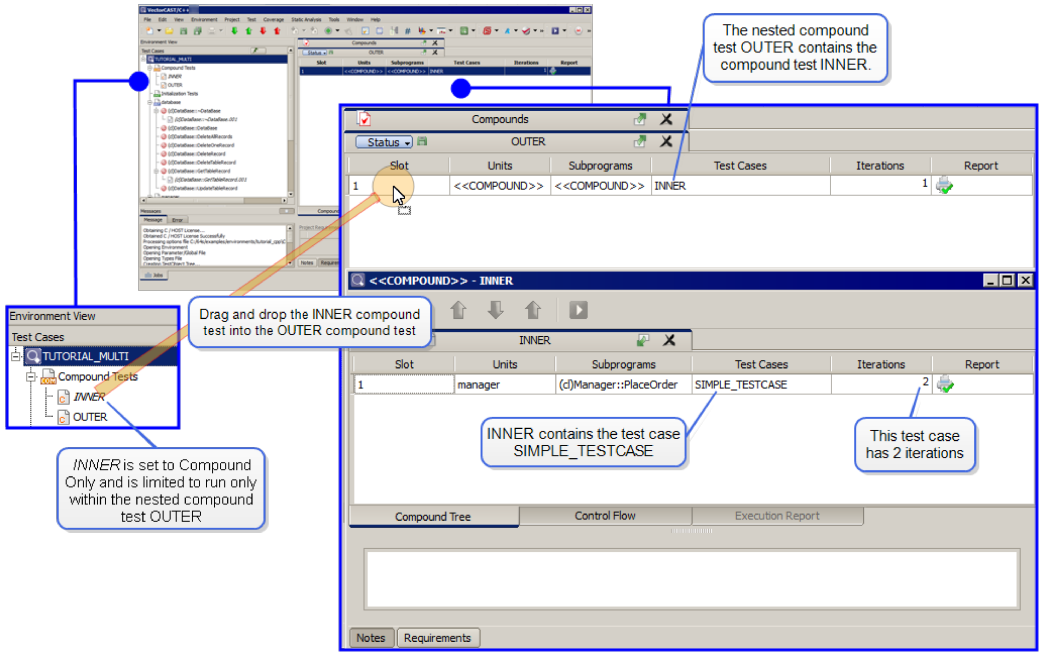
Nested Compound Test Cases

A compound test can be added as a slot to an existing compound test creating a *nested compound test*. To do so, simply drag an existing compound test case into a new compound test case. A compound test can be limited to run only within a nested compound test case by setting it to

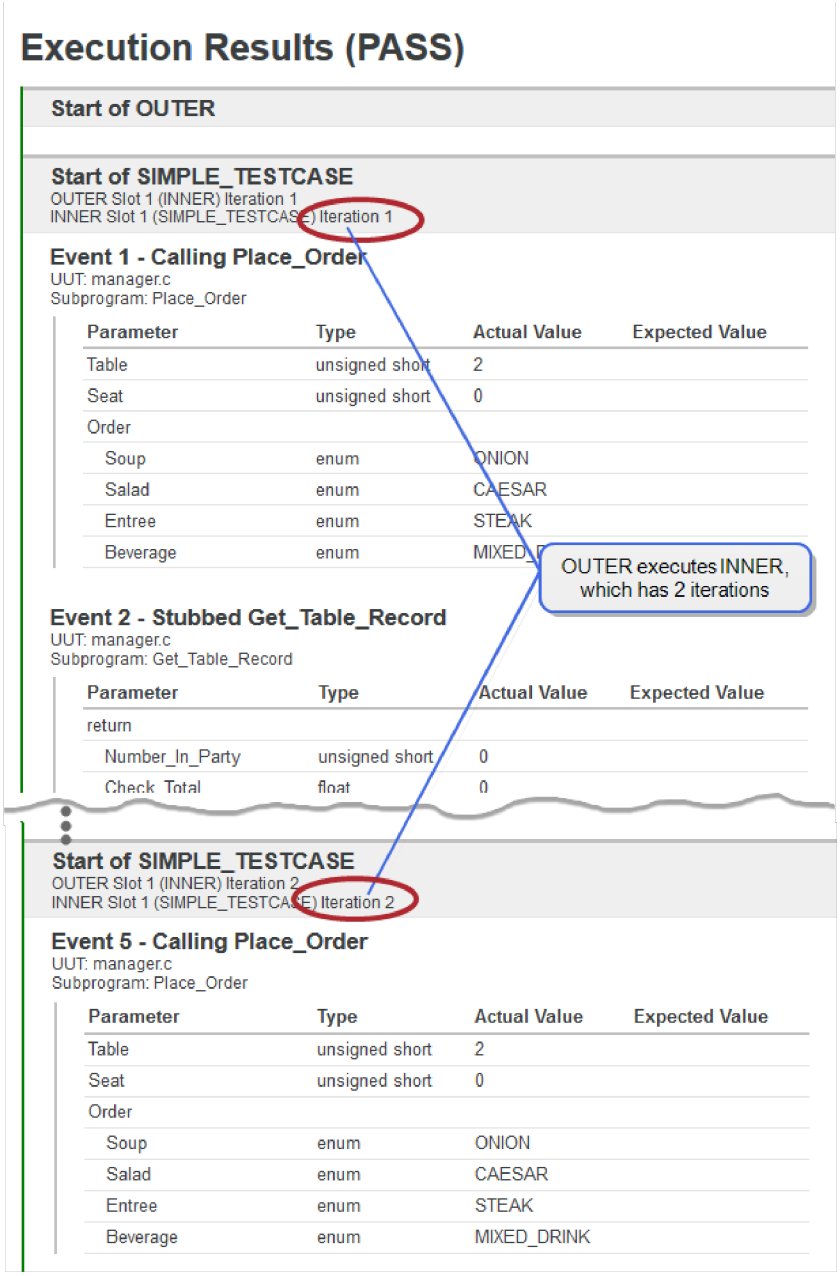
"Compound Only." See "Compound Only Test Cases" above.

In this example, SIMPLE_TESTCASE is the only slot in the compound test case named INNER. The number of iterations for SIMPLE_TESTCASE is set to 2. Another compound test case is inserted in the Test Case Tree, named OUTER, and then INNER is dragged into OUTER.

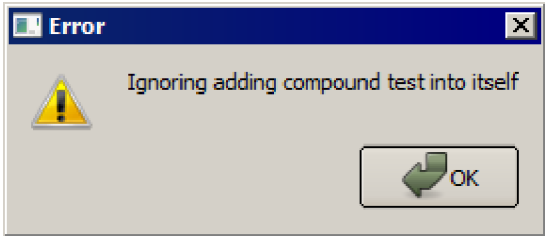
Double-clicking on INNER in the OUTER Compound Tree displays the Compound Tree for INNER.



Execution report event headers for a nested compound event reflect the nesting relationship. Below, Event 1 of the execution report for OUTER shows that OUTER executes INNER and INNER executes the first iteration of SIMPLE_TESTCASE. Scrolling down the report shows that in Event 9, the second iteration of SIMPLE_TESTCASE in INNER is executed.

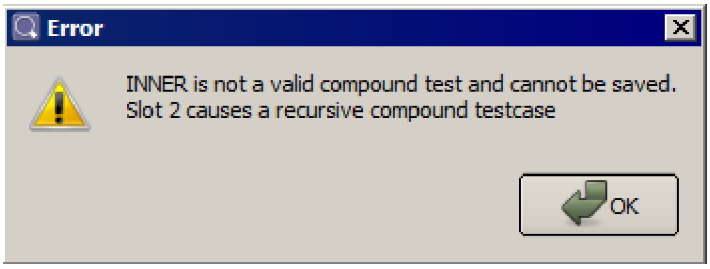


A compound test cannot be a nested slot of itself. If you drag a compound onto itself, the following error dialog appears:



Nested compound tests do not support test recursion. For example, if OUTER nests INNER, and INNER nests OUTER, the relationship is a recursive one.

The recursive relationship is detected when the change to the compound test is saved, not when the nested compound is created. When a recursive relationship is detected, an error dialog appears when you save the compound test case:



To specify a compound test case as a slot in a compound test script, use syntax similar to the following examples:

```
-- COMPOUND TESTS

TEST.SUBPROGRAM:<<COMPOUND>>
TEST.NEW
TEST.NAME:INNER
TEST.COMPOUND_ONLY
TEST.SLOT: "1", "manager", "(cl)Manager::PlaceOrder", "2", "SIMPLE_TESTCASE"
TEST.END
--

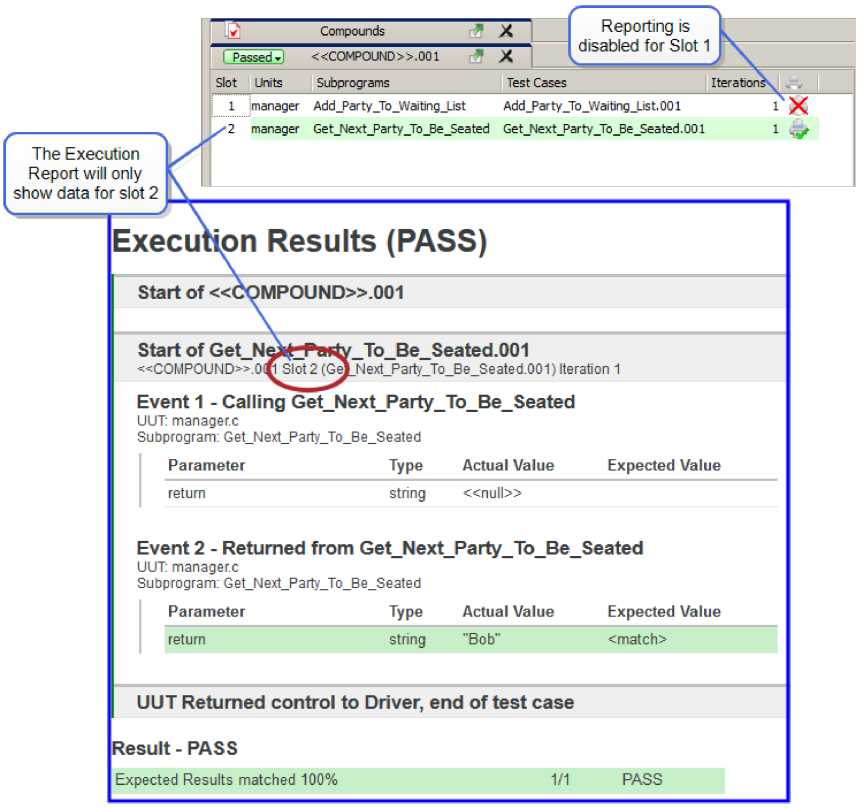
-- COMPOUND TESTS

TEST.SUBPROGRAM:<<COMPOUND>>
TEST.NEW
TEST.NAME:OUTER
TEST.SLOT: "1", "<<COMPOUND>>", "<<COMPOUND>>", "1", "INNER"
TEST.END
--
```

To Enable / Disable Reporting for Compound Slots

VectorCAST provides the ability to prevent the harness from generating Execution Report data for selected slots in a compound test case (coverage data will still be gathered). This is useful in instances of large compound test cases with numerous iterations where, for example, only the first and last iterations of a test are of interest.

By default, reporting is enabled and the **Enable Reporting** icon is displayed in the far-right column of the Compound Editor. To prevent the harness from reporting on executing data in the Execution Report, click on the icon to toggle to the **Disable Reporting** icon .



From a test script, disable reporting by adding "PRINT=FALSE" to the end of the TEST.SLOT line. For example:

```
TEST.SLOT: "1", "manager", "Add_Party_To_Waiting_List", "1", "Add_Party_To_
Waiting_List.001", "PRINT=FALSE"
```

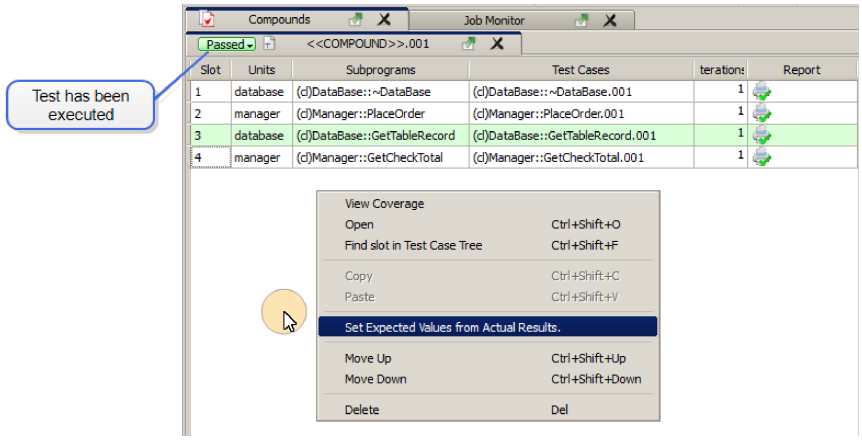
Note: If the slot data contains expected results (either for the particular test case, or, in the case of a nested compound, within the execution of the slot), then the compound test will fail to run and will notify the user of the reason.

Control Flow and Slots With Disabled Reporting

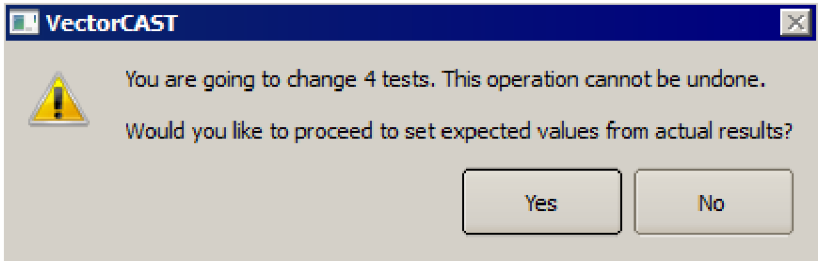
When the Print flag is set to False for one or more slots in a compound test, the slots execute but the test harness is not aware of them. Therefore, the Control Flow for the compound excludes the events for the "hidden" slots.

To Set Expected Values to Actual Values

After executing a test case, Expected Values can be set to the actual execution result values. Right-click anywhere in the Compound Test Editor and select **Set Expected Values from Actual Results** from the context menu.



A warning dialog appears indicating the number of test cases that will be changed by the action. You have the opportunity to abort the action by selecting the **No** button. Selecting the **Yes** button will allow the operation to proceed and modify the test cases. Note that this action cannot be undone.



All Expected Values listed in the Execution Report are set to their actual values.

See Controlling Report Information for more information on capturing output for variables.

```
clicast -l<lang> -e <env> [-u <unit> [-s <sub> [-t <testcase>]]] TESt
Actuals_to_expected

This will set the actual values for a test case as the expected values.
Thus 100% of the expected results will match for that test case.
```

Entering Test Case Data

There are two kinds of test case data: Input Values and Expected Values. The *input values* are values to be provided as stimulus for the test. The *expected values* are values to be verified during the test.

There are two ways to enter test case data. Enter test data directly into the Parameter Tree by selecting the appropriate spot in the “Input Value” or “Expected Value” column next to the parameter name. Alternatively, double-click on any parameter or global object from within the Parameter Tree to bring up the Parameter dialog box for the data object. The two methods are described below.

The Parameter Tree

Once you have created a test case, it opens in the Test Case Editor, in the MDI window. One of the

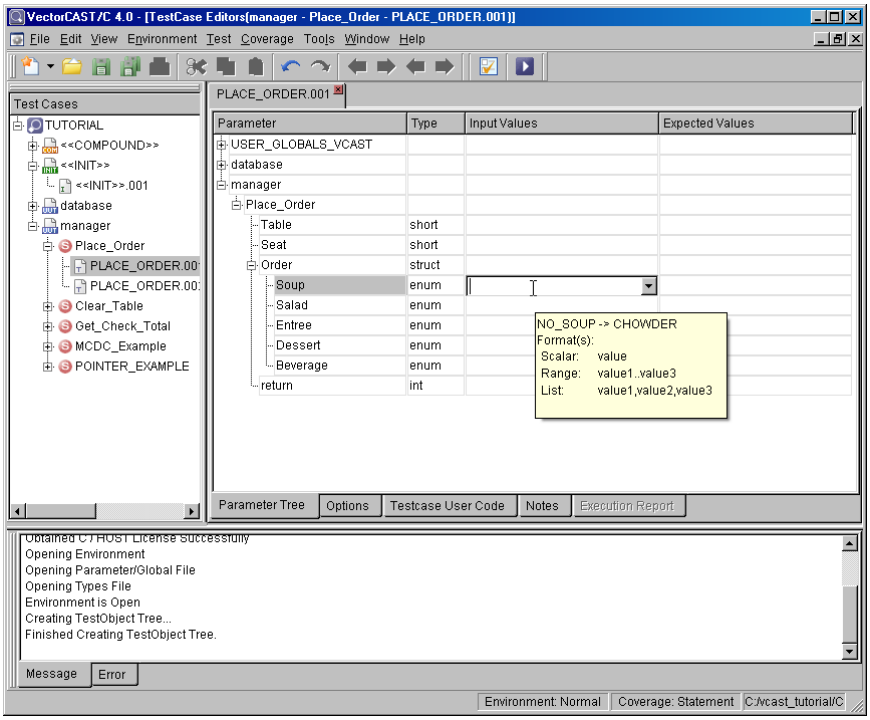
tabs of Test Case Editor, the Parameter Tree tab, displays the Unit(s) Under Test and their subprograms in a tree-like hierarchy. Using this hierarchy, you can edit the Input and Expected values for the test case.

When entering parameter values for the subprogram being tested, all parameter values that will be accessed during subprogram execution must be set. If no value is provided for a parameter or object, its value is indeterminate. Some compilers zero the stack heap and therefore parameter values before allocating data but this should not be relied upon.

Note: If you hover the mouse cursor over a cell in the Parameter Tree, VectorCAST displays the legal range of values for that parameter in a tool-tip.

The Parameter Tree also contains a column to control which parameters are to be captured in the Test Execution Report. This column is indicated by the  icon. For more information on using this control, see "To Enter Input and Expected Values" on page 277and "To Execute a Test Case and View Results" on page 390

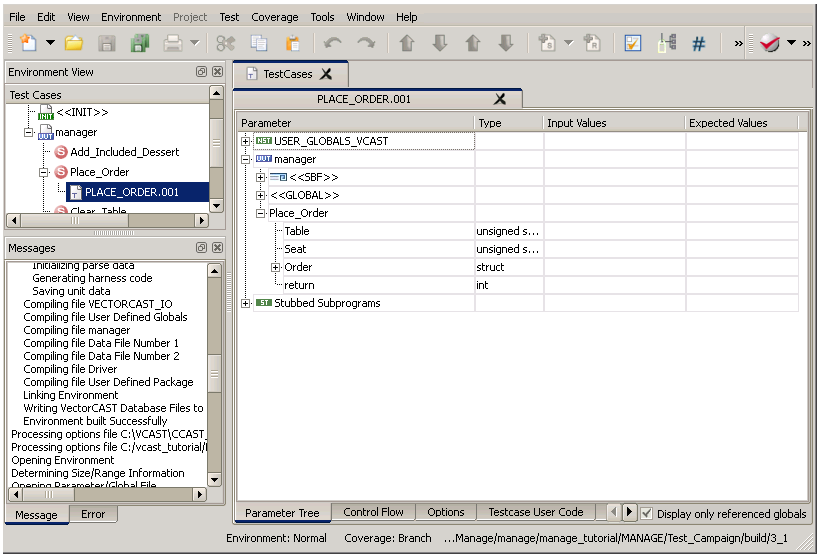
Using the Tutorial environment as an example, the UUT is **`manager`**, and one of its subprograms is **`Place_Order`**. **`Place_Order`**'s parameters consist of an integer type for the Table, an integer type for the Seat, and a structure type for the Order. The dependent unit **`database`** is also a UUT. In this case, the Parameter Tree looks similar to the following screen shot:



See also "To Enter Values for Global Data" on page 283 for information on entering Input and Expected data for specific data types.

If the environment were built with **`manager`** as a stubbed-by-function unit, then the parameter tree

would look similar to the following screen shot:

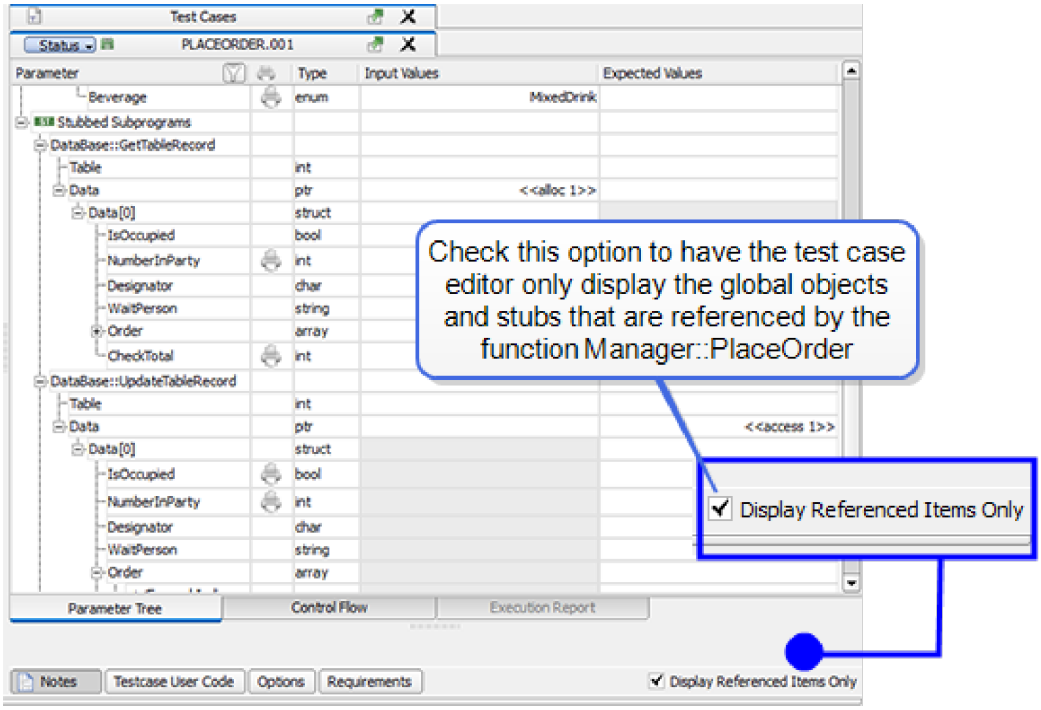


Display Reference Items Only in the Test Case Editor

A checkbox labeled "**Display Referenced Items Only**" is located in the lower right-hand corner of the Test Case Editor. When this option is set, the Parameter Tree filters out and hides any Global variable or Stubbed Subprogram that is not called directly by the subprogram under test.

Filtering an item only affects the display in the Parameter Tree; its Input and Expected Values are used in test execution.

Note: <<SBF>> subprograms are not filtered.



Each test case can have the filter on, off, or set to the default environment-wide setting, which is located on the **Tools => Options => GUI tab => Test Case Editor Options sub tab.**

To set this option in the CCAST_.CFG file so that it affects all environments in the directory, use:

```
clicast -lc option VCAST_REFERENCED_GLOBALS True | False

When this option is on, the VectorCAST test case editor will display only
those global objects and stubs that are referenced by the function under
test. The default value is False.
```

In a test case script, this option is: TEST.VALUE:<<OPTIONS>>.REFERENCED_GLOBALS: TRUE | FALSE. The default value is unspecified, meaning inherit the value of the environment-wide value of the option.

To Navigate the Parameter Tree

In the Parameter Tree, the unit names, <<GLOBALS>>, subprograms, and parameters are displayed hierarchically.

To expand a node

Mouse: click the icon to the left of the node

Keyboard: after placing the keyboard focus on that node, press the right-arrow key

To collapse a node

Mouse: click the icon to the left of the node

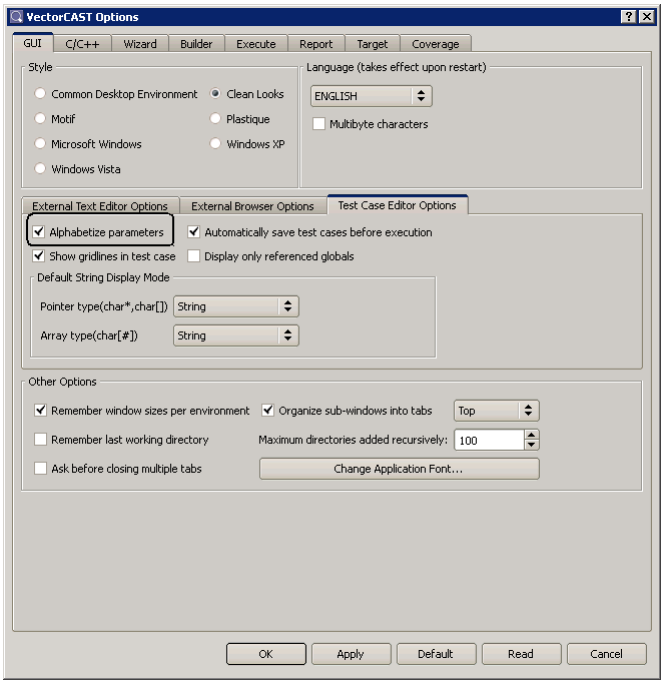
Keyboard: press the left-arrow key

Additionally, use the keyboard's up- and down-arrow keys traverse the expanded nodes. You can also type the first letter of an expanded node name to jump to that node.

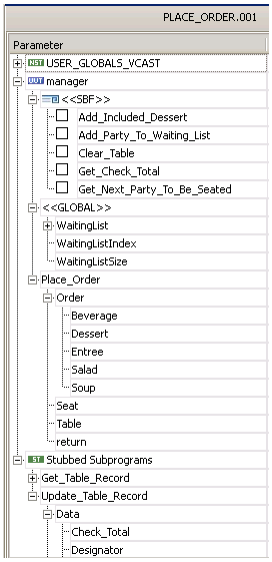
When you arrive at the node you would like to edit, simply press Enter. This will edit either the Input or Expected value (depending on which can be edited at the time).While editing a value, press the Tab key to continue to the next value, or use the up- and down-arrow keys to continue to the next editable node in the same column.

To Alphabetize Parameters in the Parameter Tree

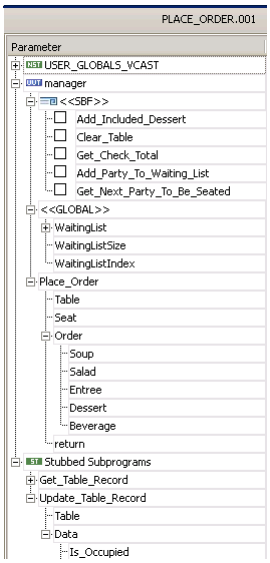
To alphabetize the items for all nodes in the Parameter Tree, select **Tools => Options => GUI**. Then click the Test Case Editor Options tab and click **Alphabetize parameters** in the *Other Options* panel. Click **Apply** or **OK.**



Close any open test cases. Open a test case and all items for each node of the Parameter Tree are displayed alphabetically, with "return" always appearing last.



To restore the order as originally processed by VectorCAST, uncheck Alphabetize Parameters in **Tools => Options => GUI**. Close any open test cases. Open a test case and all items for each node of the Parameter Tree are displayed as originally processed by VectorCAST.



To Enter Input and Expected Values

To enter values for particular parameters, simply type the value in the appropriate column (Input Values column or Expected Values column) next to the parameter name in the Parameter Tree.

Input Values to a Unit Under Test are “in” parameters; Expected Values are returns or pointers.

Parameters passed to a subprogram in a stubbed unit are Expected Values; values returned from a stub are Input Values.

Entering an Input Value or Expected value into the Parameter Tree automatically enables the data to

be displayed in the Test Execution Report. This is indicated by the  in the parameter tree. For Input Values, the data may be optionally hidden (See Controlling Report Information). Entered expected values are always shown on the Test Execution Report and may not be hidden.

Understanding Input and Expected Values

When inserting a test case, you focus on the subprogram you want the test harness to call. Keeping in mind the functionality of the subprogram in your source code, you enter values for various parameters in the Input Values column. Doing so sets up the inputs with which you want the test harness to call the subprogram. These values define a set of circumstances for the test case. Other test cases under the subprogram may be similar.

If the subprogram under test calls a subprogram in a stubbed unit, you normally would set values for that subprogram, so that the flow of control is complete. Parameters passed into a stubbed subprogram are expected values. Think of “what does the stub expect the values coming in to be?” You enter data in the Expected Values column for parameters coming into a stubbed subprogram.



Tip: Input Values for a stub’s parameters are not meaningful.

Values returned from a stubbed subprogram are passed back to the test harness, which are then passed to the calling subprogram. Think of “what does the stub send back (input) to the calling program?” You enter values to be returned from a stubbed subprogram in the Input Values column for any access parameter, In a simple case where the subprogram under test calls one stubbed subprogram, these values from the stub are passed back to the subprogram under test.



Tip: Expected Value for a stub’s return parameter is not meaningful.

The test harness can then expect certain values to be passed back from a UUT. You can enter an Expected Value for a parameter in the subprogram under test that is being passed back to the test harness.

For example, suppose you are testing a unit with source code as follows:

```
#include "UUT.h"
#include "STUB.h"

int g_UUT = 0;

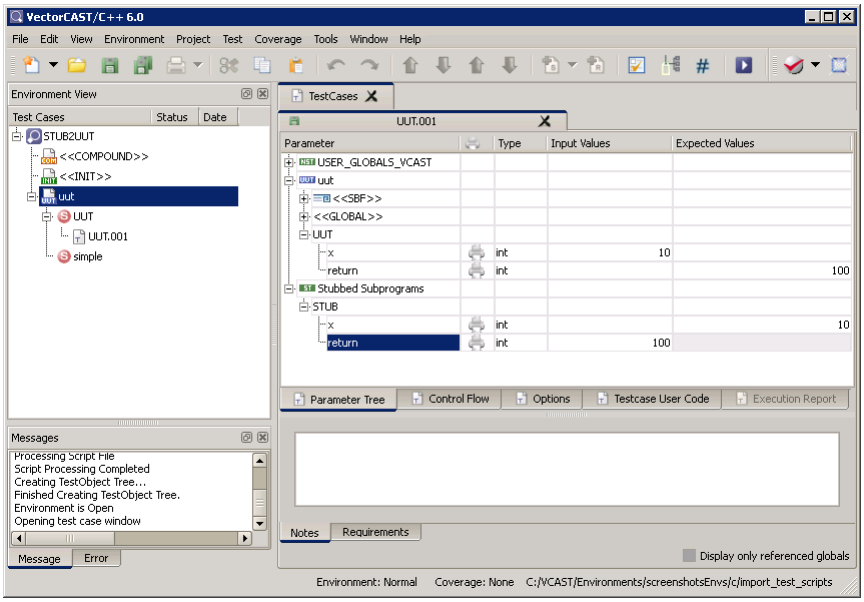
int UUT( int x ) {
   int result = x;
   result = simple ( x );
   result = STUB( result );
   return result;
}

int simple ( int x ) {
  return x;
```

```
}
```

A test case is inserted for the subprogram **`UUT()`** in the unit UUT.c. As an input value for the test harness to use when calling the subprogram **`UUT()`**, 10 is entered. The stubbed subprogram **`STUB()`** receives this value. Because STUB.c is a stubbed unit, we can make the subprogram **`STUB()`** return anything, such as 100. As a result, 100 is returned from **`STUB()`**, so **`UUT()`** expects 100 as its return value. During the call to the UUT, the subprogram **`simple()`** is called and just returns the value of its input parameter. It is not part of this Parameter Tree because it is not the subprogram under test (i.e. the subprogram for which a test case was inserted).

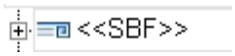
The Parameter Tree looks like this:



Input and Expected Values in a Stubbed-by-Function Unit

In the case where a UUT has been designated as a stubbed-by-function unit prior to environment build, then it is possible to insert a test case for the subprogram under test, **`UUT()`**, and stub the **`simple()`** subprogram that is called.

To Specify that a Subprogram be Stubbed

In the Parameter Tree, expand the node next to <<SBF>> 

, then check the box next to the subprogram to stub. The parameters expand to enable you to specify input or expected values.

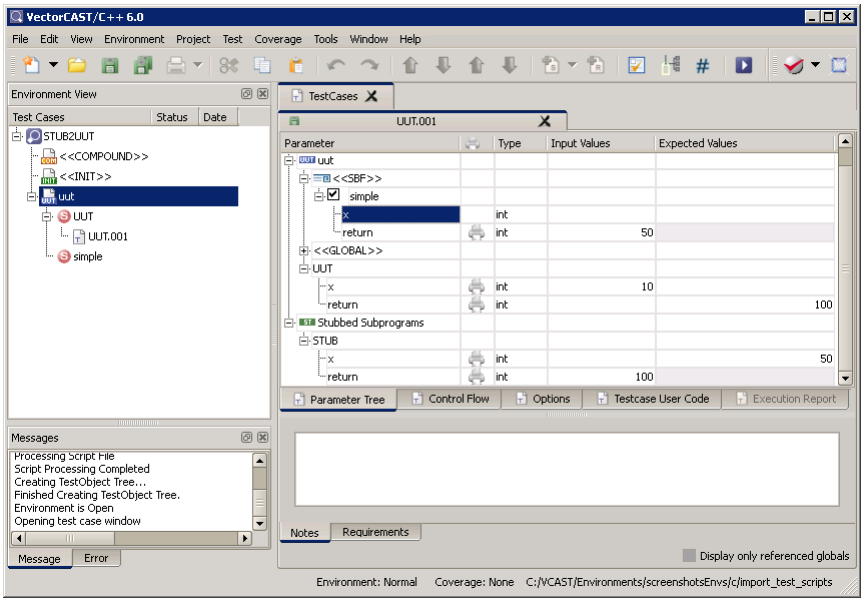
To set all other subprograms in the UUT to be SBF, right-click the <<SBF>> node in the Parameter Tree and choose **Select**. To unset all subprograms, right-click the <<SBF>> node in the Parameter Tree and choose **Deselect**.

Keyboard Shortcuts:
To select all, click the <<SBF>> node and enter **Ctrl+Shift+S**. To deselect all, click the <<SBF>>

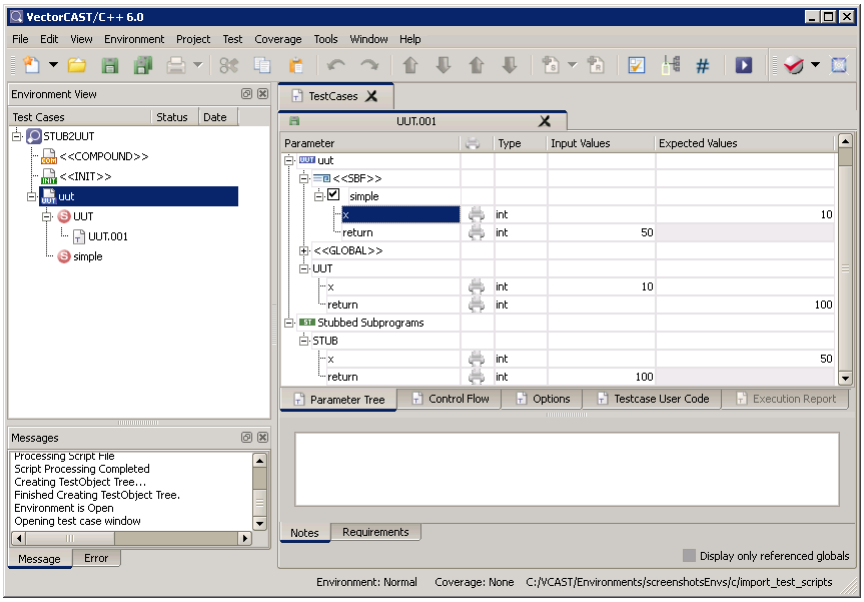
node and enter **Ctrl+Shift+D**.

As an input value for the test harness to use when calling the subprogram UUT(), 10 is entered. **`UUT()`** calls **`simple()`**, and we want to have **`simple()`** return, say, 50. After **`UUT()`** calls **`simple()`**, it calls **`STUB()`**, which is expecting the value 50 coming in. Because STUB.c is a stubbed unit, we can make the subprogram **`STUB()`** return anything, such as 100. As a result, 100 is returned from **`STUB()`**, so **`UUT()`** expects 100 as its return value.

The Parameter Tree looks like this:

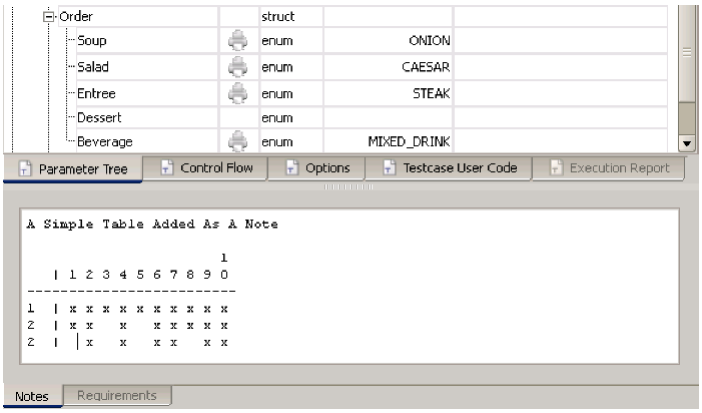


We could go one step further, and enter 10 as the value that the stubbed **`simple()`** function is expecting. Because **`simple()`** is stubbed, we get to specify what it returns. Ordinarily, **`simple()`** returns the value of its input, x.



To Enter Test Case Requirements or Notes

The Notes tab enables you to specify text that will be associated with a specific test case. This information can be an actual test requirement or test rationale. The information from this window is retained in the test case data. In the figure below, some notes are entered for the test case. The Notes tab uses a monospaced font allowing the creation of a properly spaced “graphical” table.



See also “Display/Check Global Data After” on page 449.

To Import the Contents of a File

In the Test Case Notes tab, Parameter User Code, and Test Case User Code, you can import the contents of a file.

To import the contents of a text, first click the checkbox next to Enable. Right-click in the blank text area and choose **Import file contents**. An Open File dialog appears, enabling you to choose a file of any type. Click Open, and the text from the file is displayed in the tab’s text area. Note that the Enable checkbox must be checked for the menu items to appear on the pop-up menu.

To Use an External Editor

You can edit the contents of the Test Case Notes tab, Parameter User Code, and Test Case User Code with an external editor if you've specified an external editor in the Options dialog, GUI tab. To activate the external editor, right-click in the text area of any these edit areas, and choose **Invoke External Editor**. The specified editor opens. When you are done entering text, save the file in text-only format and exit the editor. The text you edited appears in the text area.

If "Invoke External Editor" is dim, specify an external editor on the **Tools => Options** dialog, **GUI** tab.

To Import a Template for User Code or Test Case Notes

In the Test Case Notes tab, Parameter User Code, and Test Case User Code, 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. Enter the full path to the template file for the option "Notes section template."

To import the template at the cursor location in a User Code editor, first click the checkbox next to Enable. Right-click in the blank text area and choose **Import template**. The text from the template file appears in the text area.

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 -lc option VCAST_NOTES_SECTION_TEMPLATE <Path to Template text
file>

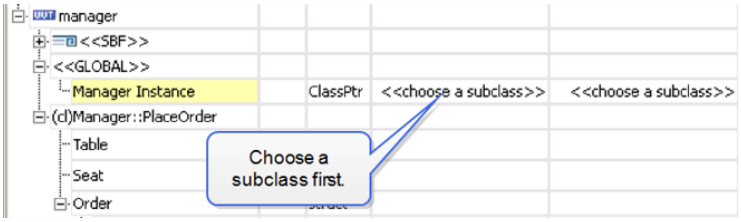
Path to a text file that contains a template for the Notes section of test
cases.
```

To Instantiate a Class Object

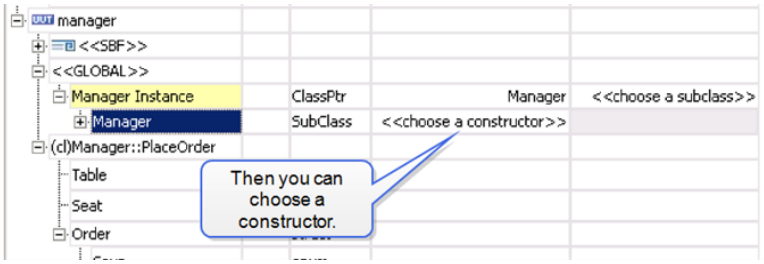
In C++ environments, in order to test a class member function, you need to instantiate the class first. You can instantiate it in the test case itself, or you can create a Compound Test Case with the first slot as a test case for the constructor. The class instance that VectorCAST uses to call the class methods is listed in the unit's <<GLOBAL>> node in the Parameter Tree.

When instantiating a class object, use the *yellow-shaded* ClassPtr, as it is the one that is appropriate for the test case you are editing.

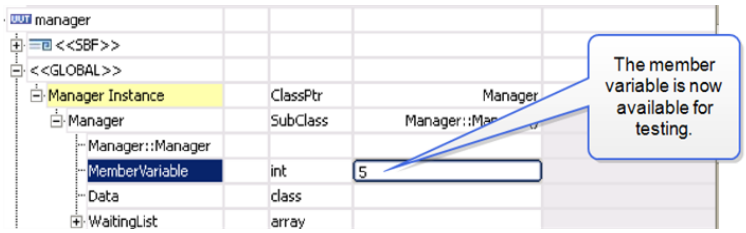
In the example below, we will construct a class instance for the manager class. First, click <<choose a subclass>> in the Input Values column. Choose a subclass name.



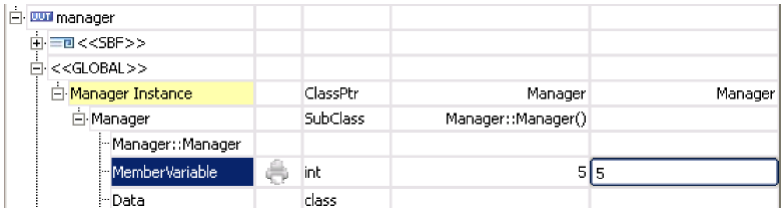
Once you choose a subclass, another row is added to the Parameter Tree and you can now choose a constructor.



Once you choose a constructor from the dropdown list, parameters for that constructor and the class member variables appear. You can set values for the member variables exactly as you would set values for all other data items.



To enter an expected value for a member variable, choose the subclass in the Expected Values column, in the same way that you do for the Input Values column.



Tip: Don't enter Input Values for the Class Instance in a constructor's test case, because they are overridden when the constructor is called in a member function's test case.

To Enter Values for Global Data

In the Parameter Tree, if any unit has global data objects, the first entry in the tree below that unit's

name is <<GLOBAL>>. By expanding <<GLOBAL>> you see a list of global data objects which are defined in that unit. All processing associated with setting values for global objects is exactly the same as setting data for a parameter.

To Enter a Range

For any data type, you can enter a range for an Input or Expected Value in the Parameter Tree. In the Input Values column, the range syntax is ***low..high/delta***. The range causes the test case to iterate over the input values starting at *low*, going to *high*, by *delta*.

In the Expected Values column, the range syntax is ***low..high*** and defines a range of values that will match an input. The acceptable range includes ***low*** and ***high***.

See also "The Range Values Tab" on page 317.

To Enter a List

For any data type, you can enter a list for an Input or Expected Value in the Parameter Tree. In the Input Values column, the list syntax is ***value-1,value-2,value-n***. The list causes the test case to iterate over the input values starting with ***value-1***, and continuing to ***value-n***.

The special input value **<<KEEP>>** indicates that the input object should keep its current value from the previous iteration, rather than being modified. This special input value is useful when an item needs an initial value but an iteration is occurring and you don't want a new value set on each iteration. For example, the input might look like:

```
123,<<KEEP>>,<<KEEP>>
```

In the Expected Values column, the list syntax is ***value-1,value-2,value-n*** and defines a list of values that will match an input, one at a time. There is no space between the items. To specify that an item be repeated, use parentheses around the number of times to repeat the value: (***repeat***)***value-1,(repeat)value-2,(repeat)value-n***

Repeating and non-repeating list items can be mixed in the list. A range can be an item in an Expected Values list.

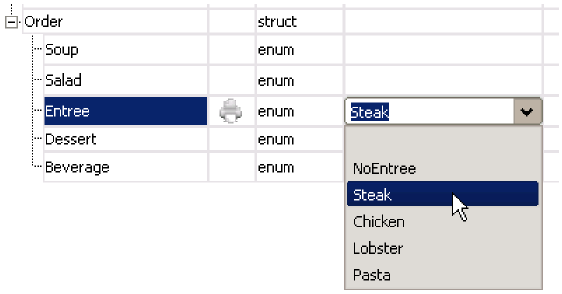
See also “The List Values Tab” on page 319.

Data Types

The parameter types referred to in this user's guide are defined as "Scalar" and “Composite", with Scalar types being broken down into Integer and Floating Point types. Integer types can be further subdivided into Enumerated and Numeric types. Composite types can be subdivided into array, structure (including class), and pointer types.

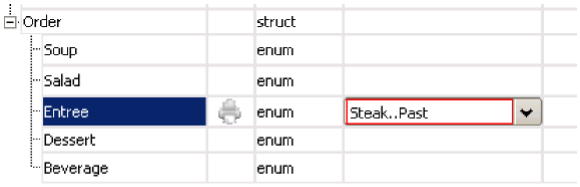
To Enter Values for Enumeration Types

For enumeration types, values are set by clicking in the Input Value space and selecting from the drop-down list of enumerals. You may also type the enumeral value. VectorCAST auto-completes the name as soon as the entered characters uniquely identify an enumeral.



If you enter an illegal value, VectorCAST outlines the cell in red and the tool-tip provides information.

When entering a range for an enumeration type, the syntax is *value1..value2*, where *value1* and *value2* are enumerals defined in the type. VectorCAST auto-completes the first value as you type it. After you type ".." a red outline appears, until you finish typing the second value. VectorCAST does not auto-complete the second value in the range.

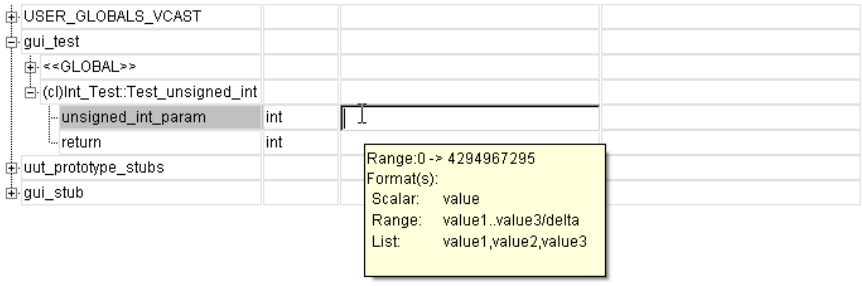


To Enter a Number for an Enumeration

In some cases you may want to enter a number in place of the enumeral. VectorCAST permits integer values to be assigned to enumerated types. In this way, you can force an out-of-range value for an enumeration type parameter.

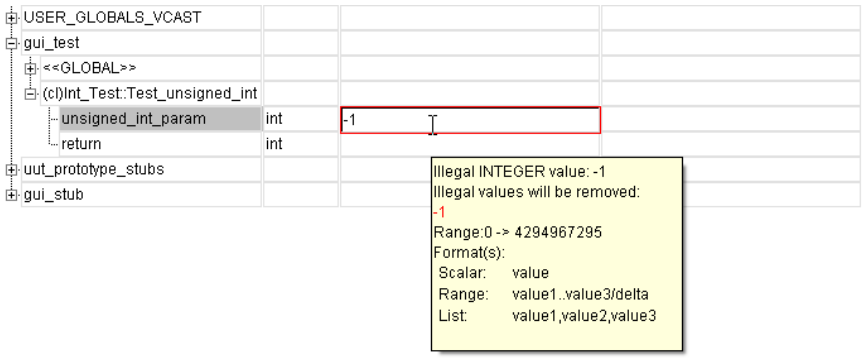
To Enter Values for Numeric Types

For numeric types (i.e. integer, float), the parameter value is set by typing the number into the value cell.



Out of Range or Illegal Values

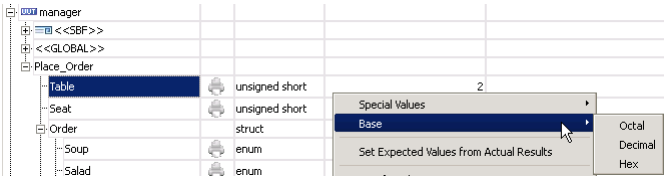
If, based on the data attributes of the selected parameter, the numeric value is out of range or the value entered is inconsistent with the type, a red outline appears, and a notification of illegal value will be presented in a tool-tip.



To Enter Integer Values in Different Bases

The assumed "base" for entered values is base 10. For integer types you may also select alternate bases.

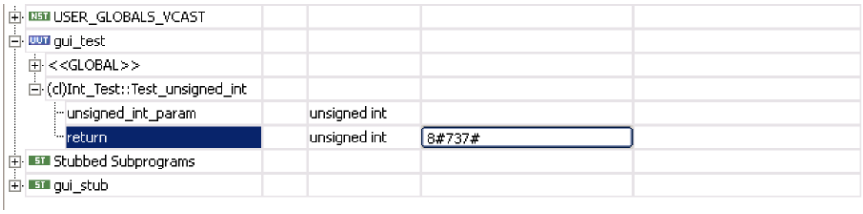
To change the base, right-click an input or expected value in the parameter tree and select the desired base from the context menu.



A base may also be specified as part of the value, using the standard C syntax of a “0” prefix for octal or a “0x” prefix for hexadecimal, or Ada syntax of “8#” prefix for octal, “16#” prefix for hexadecimal, or “2#” prefix for binary. When using the Ada syntax, any value between 2 through 16 is valid.

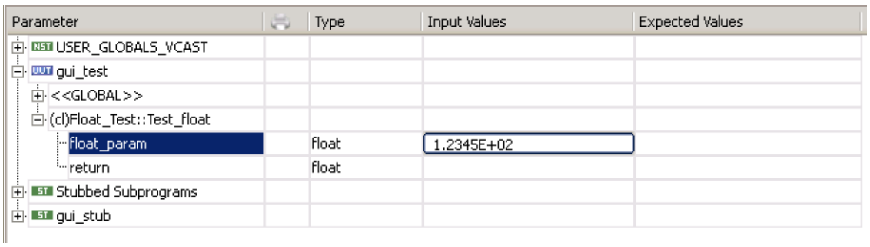
For example:

- Hex: 16#3A0# or 0x3A0
- Octal: 8#737# or 0737
- Binary: 2#1010#
- Base 5: 5#12# (equivalent to the value of 7 for the decimal base).



To Enter Real Numbers using Scientific Notation

Real numbers may be entered using scientific notation (i.e. 1.2345E+02).

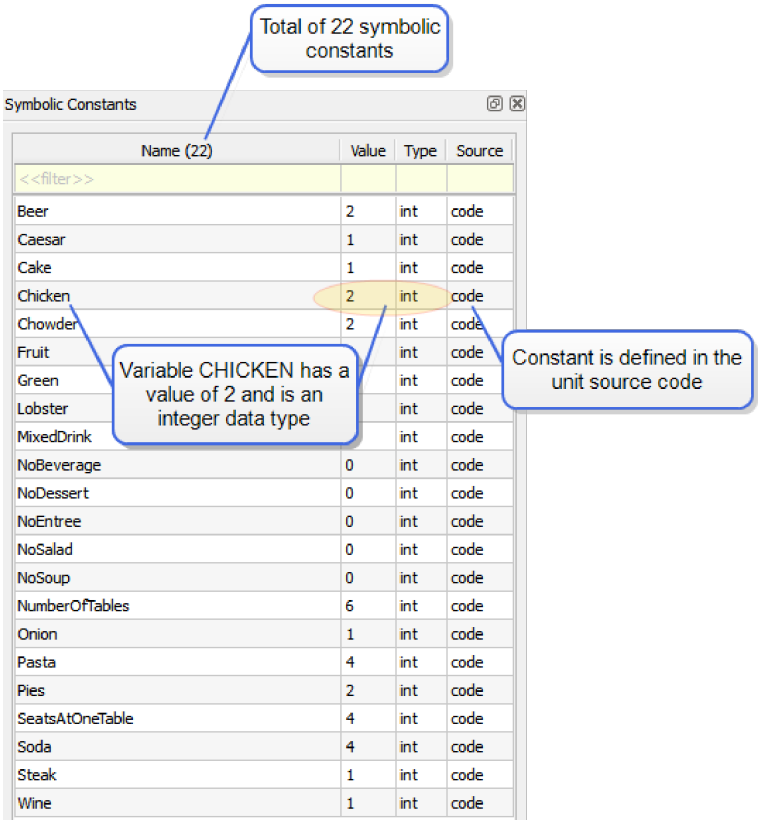


To Use Symbolic Constants

VectorCAST creates a data dictionary for numeric macros or global constants that are defined in any of the Units Under Test or their header dependencies. These integer and floating-point symbolic constants are detected during the environment build process and are available in the Symbolic Constants window. These values can be used to control test case execution through the Variant Logic feature. For more information on the Variant Logic feature, see "Control Test Case Execution Using Variant Logic" on page 400.

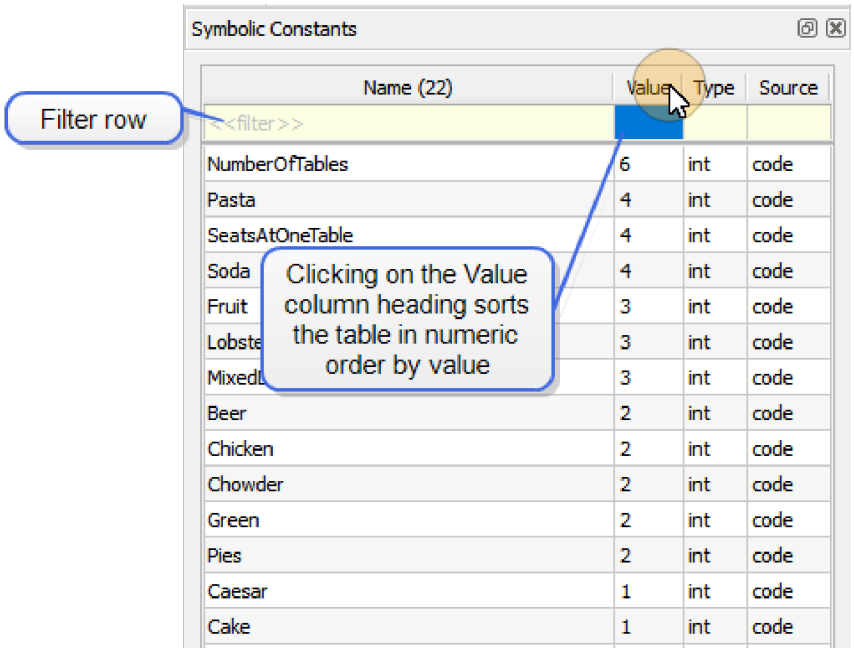
The Symbolic Constants window opens automatically if the units under test contain enumerations, #defines, or other symbolic constants. The user can override the default behavior by clicking the **Symbolic Constants** button on the Toolbar. By default, the window is displayed in the right hand pane of the MDI area, and may be undocked.

The Symbolic Constants window remains open as long as there is at least one test case open in the Test Case Editor.



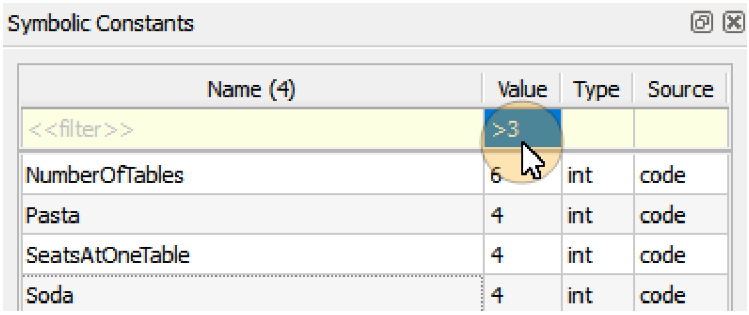
The Symbolic Constants window contains four columns: the macro or variable **Name**, the **Value**, the base Data **Type** (int or float) and the **Source** of the constant (code or user defined).

Sorting and filtering is available to locate symbolic constants 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.



Symbolic constants can be accessed all the way down to the individual constant 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 Symbolic Constants window has been filtered to only show constants with a value greater than 3.



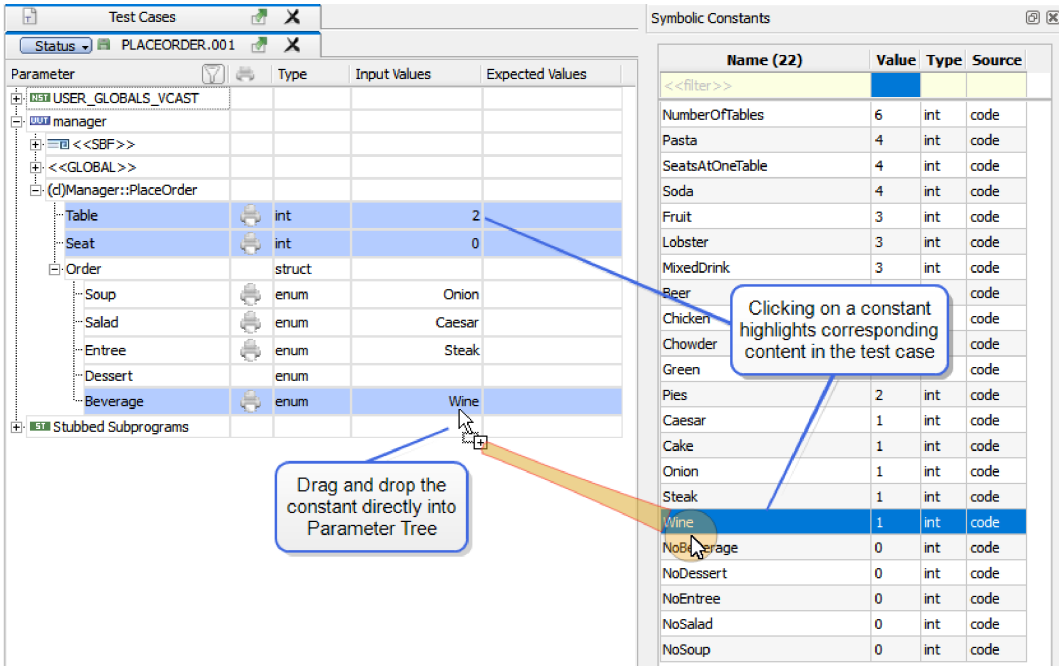
Filtering supports the following symbols: <, >, =. For example, the window can be filtered to only show constants with a value equal to 3. Other examples of filtering inputs are:

- **10** - lists symbolic constants matching the specific value of "10" in the selected column
- **>5** - lists symbolic constants with a value greater than the value of "5" in the Value column
- **<9** - lists symbolic constants with a value less than the value of "9" in the Value column
- **=12** - lists symbolic constants matching the specific value of "12" in the Value column
- **<** - lists only duplicate values in the Value column
- **>** - excludes duplicate values in the Value column

You can use a symbolic constant as an Input or Expected Value by dragging it from the Symbolic

Constants window and dropping it into the Parameter Tree, or by typing its name in the Parameter Tree.

When you select a particular Symbolic Constant, the Test Case Editor window highlights the data items that match the type of the selected constant. Use **Shift + Esc** keys to remove the highlighting.



Duplicate constant names with different values are not supported. Duplicates are disabled and displayed in gray text in the Symbolic Constants window. They cannot be dragged and added to the Parameter Tree.

Using Test-Only Symbolic Constants

The Test-Only Symbolic Constants feature allows the user to supply test-only constants. This makes test cases more portable by replacing hard-coded scalar values with symbols that can be controlled independently from tests.

The benefits of this approach are:

- Improved test readability - replacing hard-coded values with symbolic names increases readability and understanding.
- Decreased test maintenance - values of symbolic names can be easily changed in one location. When a value is changed, all test cases are updated automatically.
- Better support for variant testing - symbolic names can be set to different values for different test contexts.

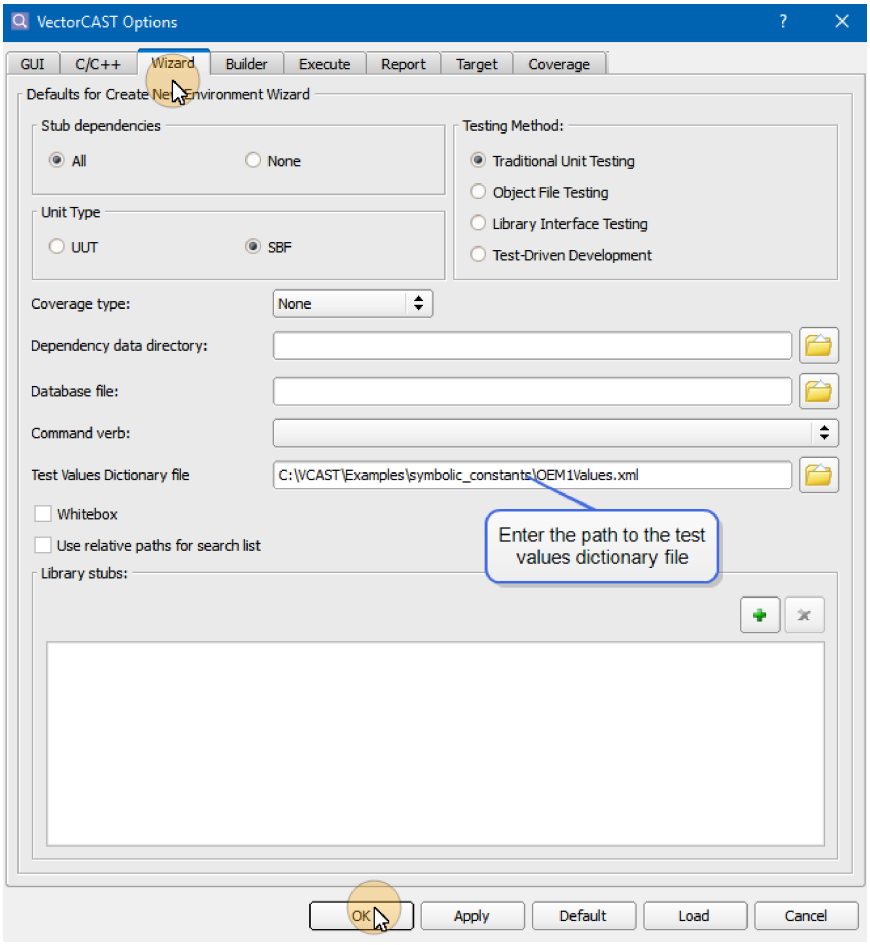
Test Values Dictionary

The Test Values Dictionary is an .xml file containing the variables for test values. Variables can be set to have a single value, a list of values, or a range of values. An example Test Values Dictionary is

shown below:

```
<test-values-dictionary version="1">
  <test-value name="dataSet" type="string" value="OEM1"/>
  <tes--value name="maximumSpeed" type="int" value="200"/>
  <test-value name="speedRange" type="int" value="0..200/20"/>
  <test-value name="keySpeedValues" type="int" value="0,25,55,75"/>
  <test-value name="saferatio" type="float" value="0.920"/>
</test-values-dictionary>
```

To enter the path to the Test Values Dictionary, select **Tools => Options...** and select the **Wizard** tab. Using the browser dialog, enter the path to the location of the dictionary. Click the **OK** button.




```
clicast option VCAST_TEST_VALUES_DICTIONARY <path to test values dictionary
file>

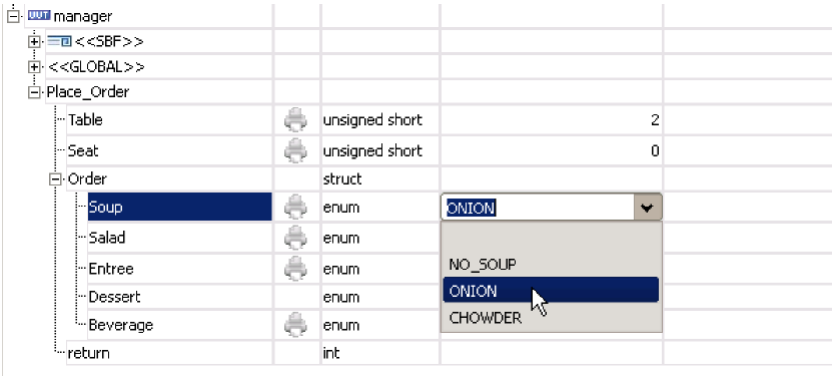
Path to the test values dictionary file.
```

To Enter Values for Structure Types

To enter values for a Structure, use the plus ⊞ button next to the parameter name to see the fields

defined for the structure type.

As a Structure Type is a composite data type, you do not enter values for the structure directly. You choose a field of the structure by first expanding out the list of fields and then setting a value for any scalar fields. If a field is a structure, you would expand that field using the plus ⊞ button again next to the field.



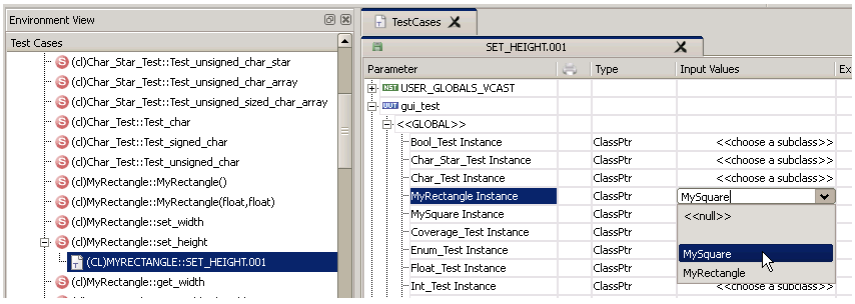
To Enter Values for Union Types

Union Types are handled exactly like structures, with the caveat that only one part of the union can be active at a time. As a result when you set data for a field in a union, any existing data for any other field in that union is discarded by VectorCAST.

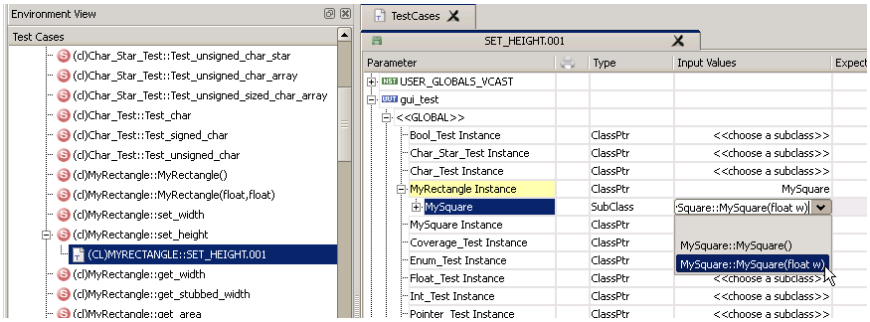
To Enter Values for Class Types

When setting values for parameters that are classes, or global objects that are class pointers, you must first construct a specific instance of the class. To do this, select a subclass, a constructor, and, if applicable, constructor parameter values.

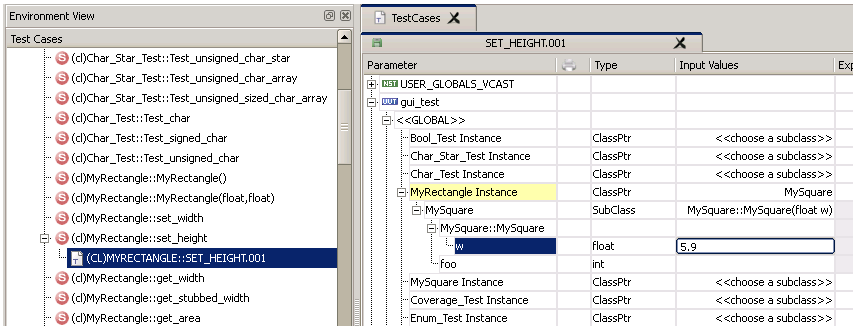
The list of subclasses contains the class name of the base class as well as the name of all classes derived from this class. The following screen shot shows an example for a shapes application, with a rectangle base class and a square subclass.



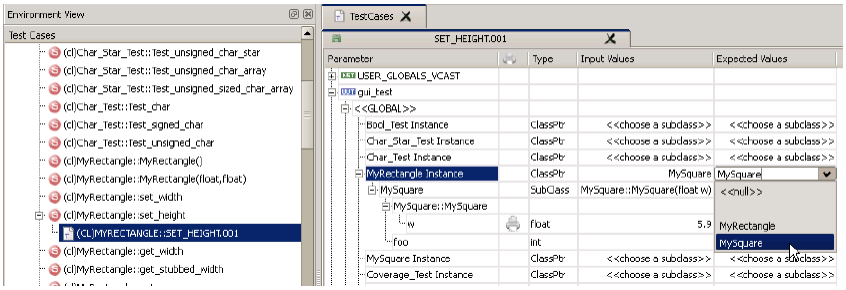
After selecting MySquare as the subclass, you may choose the constructor that VectorCAST will use to instantiate this class. The constructors are listed in a drop-down box for the class.



Once the subclass and constructor are chosen, the values for the constructor parameters and public data members are entered the same way as for any other data types.



When setting expected values for class variables, you use the same steps as for input values: first choose a subclass, then choose a constructor, and finally set an expected value for a particular member variable.



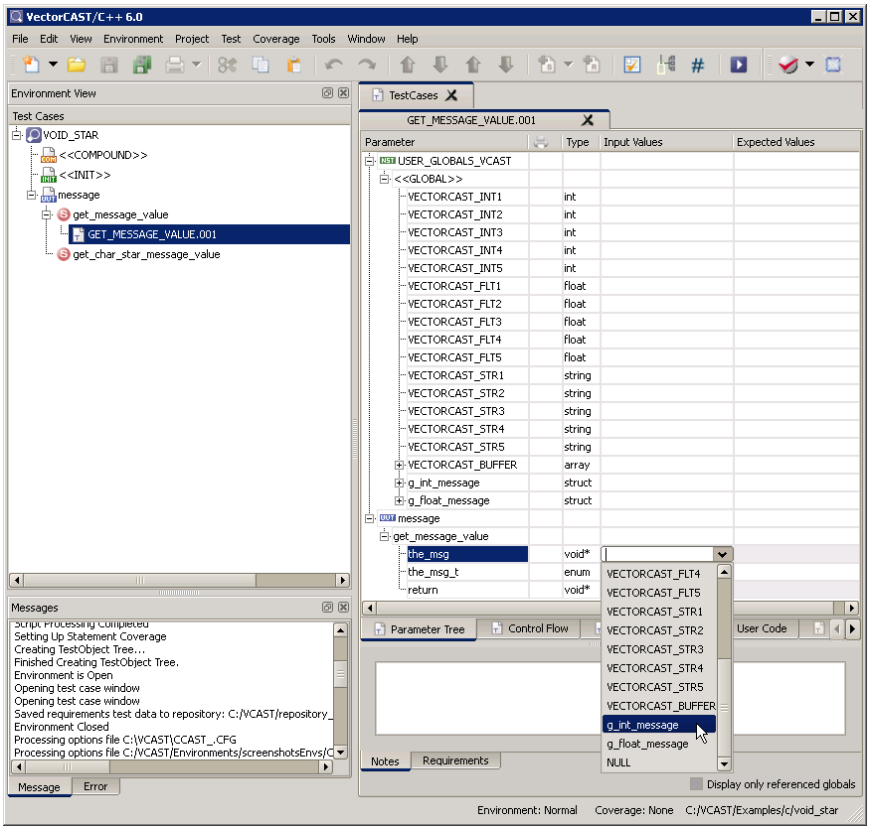
Void Pointer Types

Since the type of pointed-to data is not known when a pointer is declared as “void*”, VectorCAST enables you to set a void pointer value to be NULL or to set the value to be the address of a global data object that is declared in the User Globals environment user code. You may add your own types and objects to this file to allow you the flexibility to use any type you wish for a void pointer.

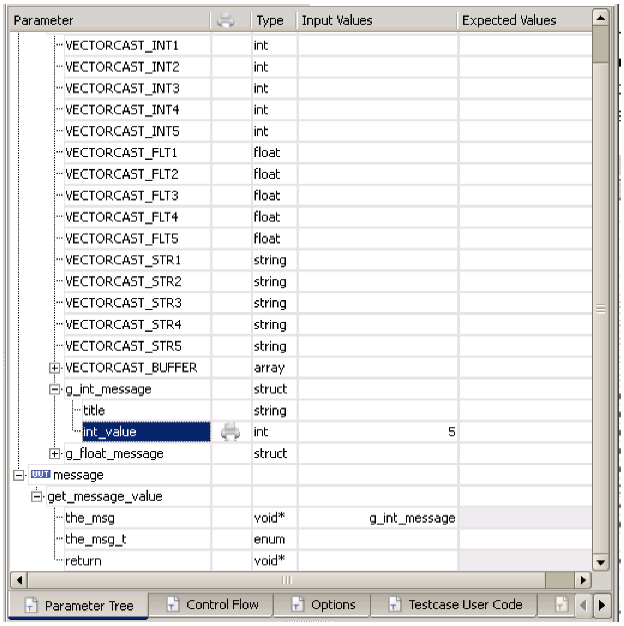
Consider the following function prototype (taken from the C example named void_star, located in the VectorCAST installation directory):

```
void* get_message_value ( void * the_msg, MESSAGE_STRUCT_TYPE the_msg_
t );
```

To set a value for the **`the_msg`** parameter you would choose from the drop down list of USER_GLOBALS objects presented to you in the Parameter Tree, as in the following example:

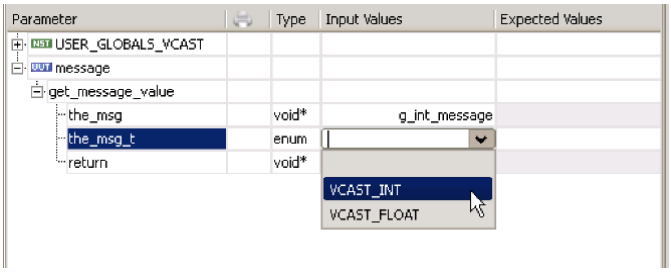


Once you have chosen an object to use for the void pointer, you then set the actual data by setting the value of this pointed-to object. For our example, if you selected the **`int_value`** member of the **`g_int_message`** structure for the value of the void pointer, you would then set the value of the object **`int_value`** as shown here:

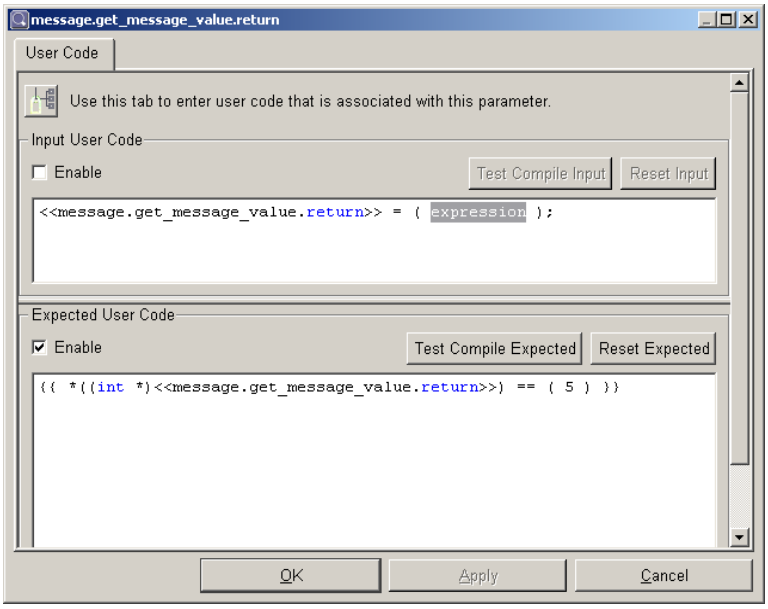


If you add additional user global objects to the User Globals file, those object names also appear in the drop-down list.

Because this **`g_int_message`** is an **`INT_MESSAGE`** type, select the enum value INT as **`the_msg_t`** input parameter.



Finally, for the expected results, we must use usercode to convert that value by typecasting it as an integer pointer and dereferencing it for comparison to our expected value like so:



See "User Globals" on page 606 for information on adding user global objects.

Character Types

If the parameter chosen is of “char” type, you can enter the character or value, or enter a numeric hex or octal value using the standard syntax (\ for octal, \x for hex). For example:

- c
- \0 for the null character (0)
- \x0A (hex) or \12 (octal) for the ASCII LF character

To Enter a String

To enter a value for parameter of type string, type the string without quotes into the Input Values or Expected Values column. To include a comma character in the string, use the escape character before it. For example, the string **`bob\,tom`** is a string of one item.

To type a list of strings, use the comma character to separate the items of the list. For example, entering **`bob,tom`** in the Input Values column for a string parameter enters a list of two string items, causing the test case to iterate with “bob” as the first input and “tom” as the second input.

You can enter the following special characters without escaping them:

~ ! ` @ # $ ^ & * ( ) - _ + = { } [ ] | : ; “ ‘ < > ? / <space>

To enter the character \ (backslash), escape it with another \ first. To enter the comma, escape it with a backslash first.

To enter the % character, use the syntax \045, as the % character itself is used internally by VectorCAST as a string delimiter.

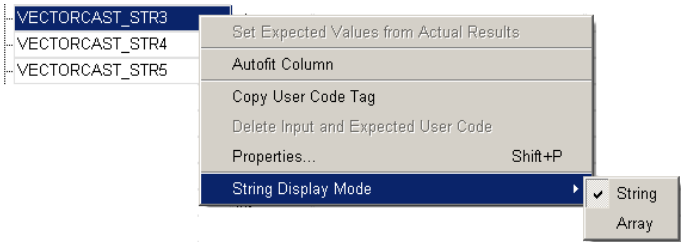
You can include unprintable characters in your string by using the standard C-style backslash syntax.

For example:

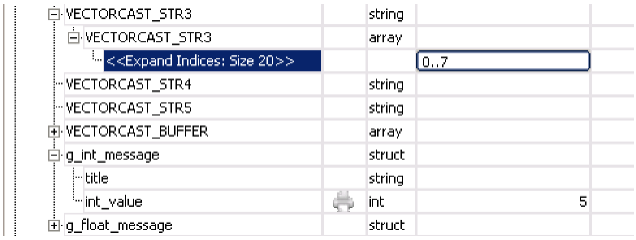
- \000\123 for octal values
- \xff\x13 for hexadecimal values
- \n for newline

To Change a String Type into an Array of Characters

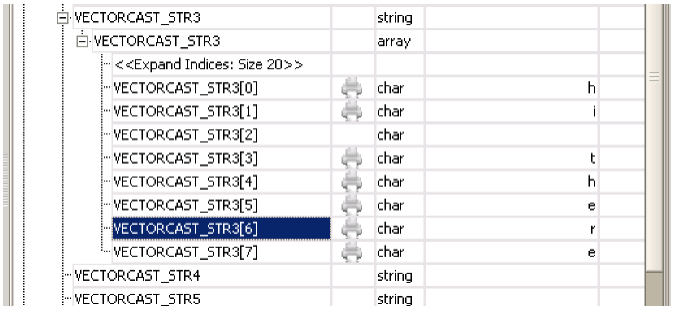
If the parameter chosen is of “char*” type, VectorCAST treats the parameter as either a string (default) or as a pointer to an array of characters. To change modes, right-click on the parameter name, and choose **String Display Mode**. Choose **Array** to change the string into an array or characters.



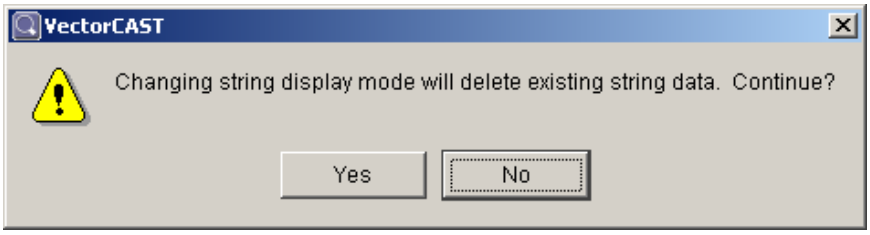
The parameter is displayed as an array. To expand the array indices, enter the range of array elements in the Input Values column.



The following figure shows the string “hi there” in array mode.



If the display mode is changed and there is already data in the parameter, VectorCAST asks for confirmation before deleting the data.



Support for C++ Standard Template Library (STL) types

The VectorCAST test harness and test case editor provide easy-to-use abstractions for the C++ Standard Template Library (STL) containers, such as vectors, maps, lists, pairs, etc. All of the STL containers are supported.

An example has been added to the VectorCAST distribution. This example is located in the VectorCAST installation directory, sub-directory: "examples/cpp/STL". In the example directory, you will find a source file called: stl.cpp, an environment script called: STL_CONTAINERS.env, and a test case script called: STL_CONTAINERS.tst. If you build the environment and load the test case script you will be able to run test cases for the sample code that demonstrates interaction with the following STL containers:

- list
- vector
- map
- set

The following containers are also supported but not illustrated in the example:

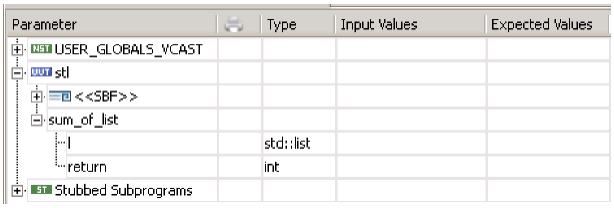
- pair
- queue
- dequeue
- stack
- priority queue
- multiset
- multimap
- bitset

The following function accepts an STL **`list`** of integers as its input parameter, adds all of the integer values, and then returns the sum as an integer.

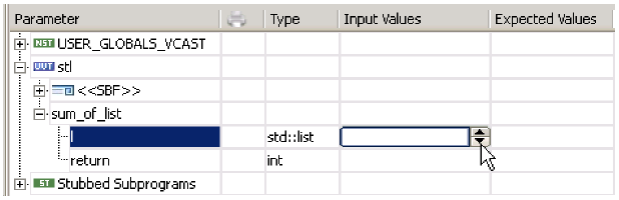
```
int sum_of_list(const std::list<int>& l)
{
   int sum = 0;
   for (std::list<int>::const_iterator i = l.begin(); i != l.end(); ++i)
      sum += *i;

   return sum;
}
```

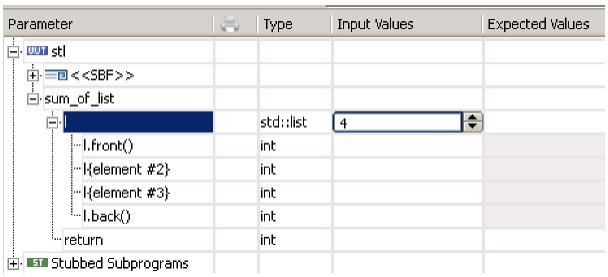
After adding a test case for this subprogram, the parameter tree contains an empty STL list for the input parameter, “l”. Note that VectorCAST uses the std:: namespace scope resolution identifier when creating STL data items.



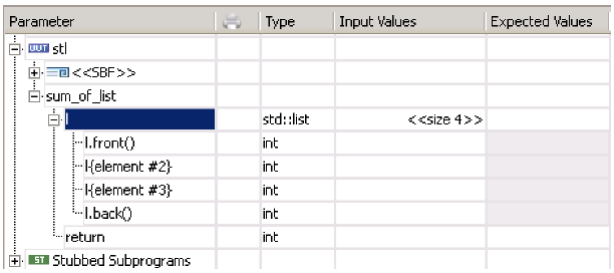
To add input test values to the STL list, first click the Input Values cell and a spinner control will appear.



Use the spinner control, or type a value directly into the cell for the number of values that the STL list contains. For this example, the list size is set to 4.



After pressing Enter, or clicking on another cell, the list expands to contain the number of elements specified.



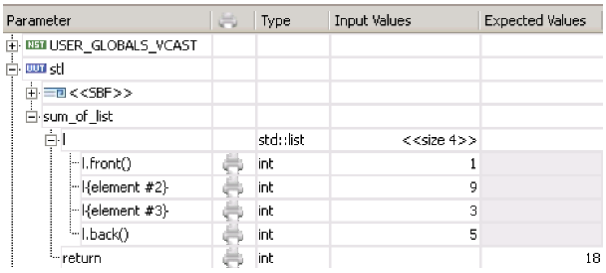
Notice that in the parameter tree, the first element name is automatically named *l.front()* and the last element is named *l.back()*. This is standard STL terminology for the first and last elements of a list. VectorCAST uses standard STL terminology in identifying elements of STL containers. This makes it

easy to identify the elements of the container when entering and verifying values in the Parameter Tree.

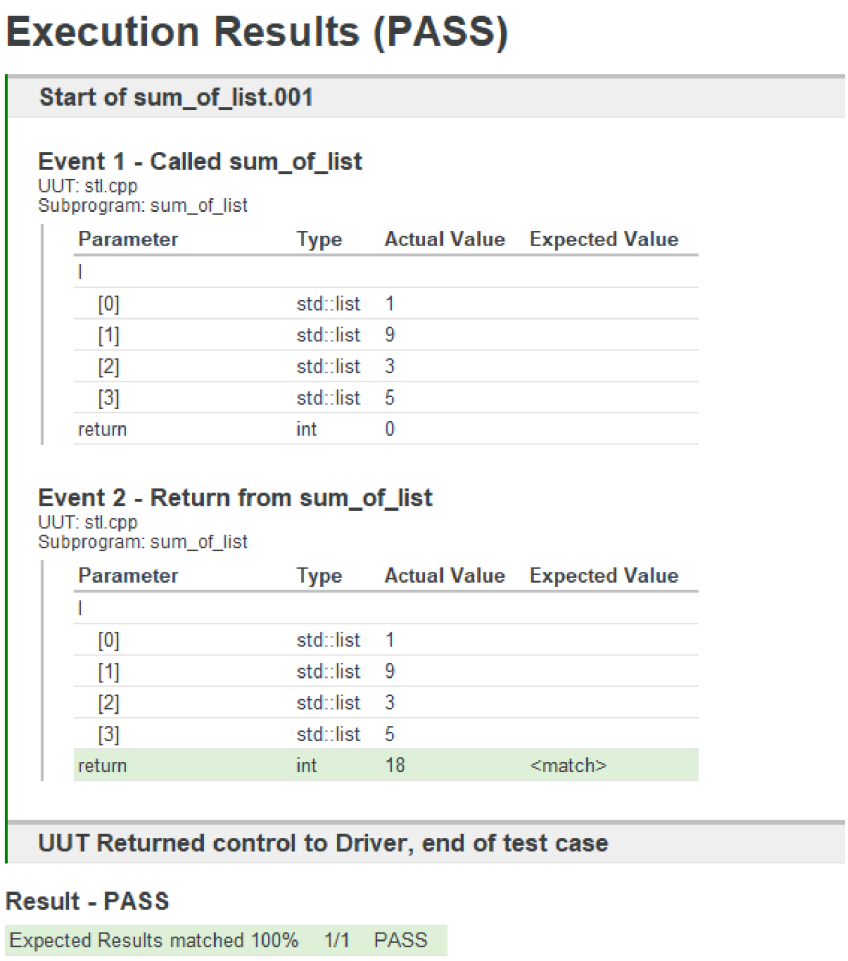
In the Parameter Tree, VectorCAST displays an element label for some STL containers. The following table lists the container types that display element labels, the element position within the container and the associated element label.

STL Container	Element Position	Element Label
List	First, Last	front, back
Pair	First, Second	first, second
Queue	First	front
Priority Queue	First	top
Stack	First	top
Note: Both the STL *map* and *multimap* contain STL pairs as elements. When these elements are expanded in the Parameter tree, the pair ids, first and second, are shown.		

To add values to elements of the list, click each input value element in the Parameter Tree and enter a desired value. The sum_of_list method returns the sum of the list elements, so the value for the expected sum is placed in the *return* expected value cell.



When the test is executed using the values entered, the test passes and the following Execution Report is displayed.



Execution Results (PASS)

Start of sum_of_list.001

Event 1 - Called sum_of_list
UUT: stl.cpp
Subprogram: sum_of_list

Parameter	Type	Actual Value	Expected Value
l			
[0]	std::list	1	
[1]	std::list	9	
[2]	std::list	3	
[3]	std::list	5	
return	int	0	

Event 2 - Return from sum_of_list
UUT: stl.cpp
Subprogram: sum_of_list

Parameter	Type	Actual Value	Expected Value
l			
[0]	std::list	1	
[1]	std::list	9	
[2]	std::list	3	
[3]	std::list	5	
return	int	18	<match>

UUT Returned control to Driver, end of test case

Result - PASS

Expected Results matched 100% 1/1 PASS

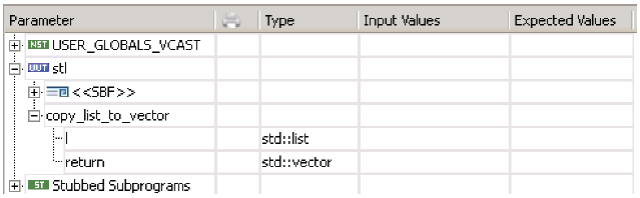
copy_list_to_vector

```
std::vector<int> copy_list_to_vector(const std::list<int>& l)
{
   std::vector<int> v;
   for (std::list<int>::const_iterator i = l.begin(); i != l.end(); ++i)
      v.push_back(*i);

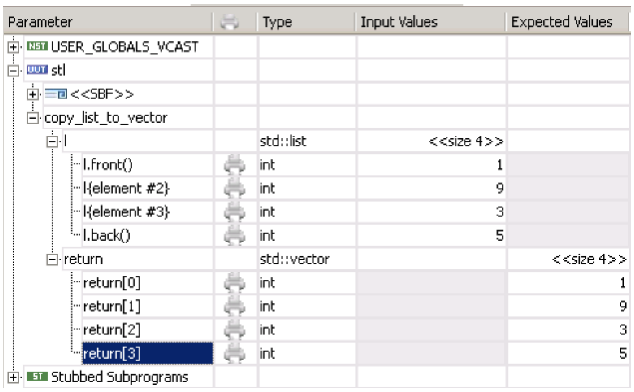
   return v;
}
```

This function accepts an STL *list* as its input parameter, iterates through the list and then copies each element to an STL *vector* element and returns the vector.

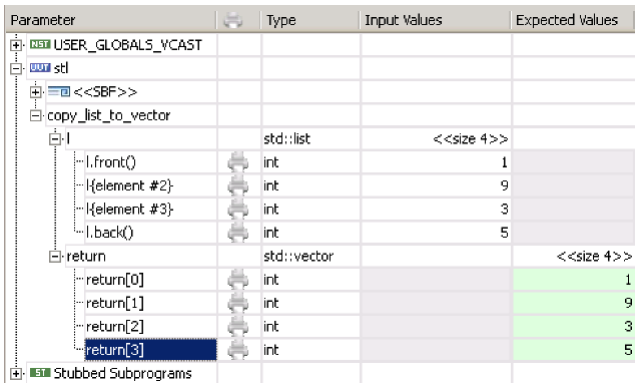
When a test case is added to the Test Case Tree for this subprogram, the parameter tree contains an empty *list* for the input parameter and an empty *vector* for the return value.



The size of the containers is set using the test case editor as was shown in the previous example. Both Input Values and Expected Values are then entered into the parameter tree.



When the test is executed, the values match and the test passes.

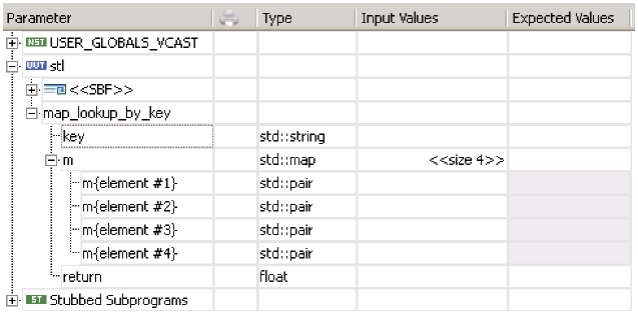


map_lookup_by_key

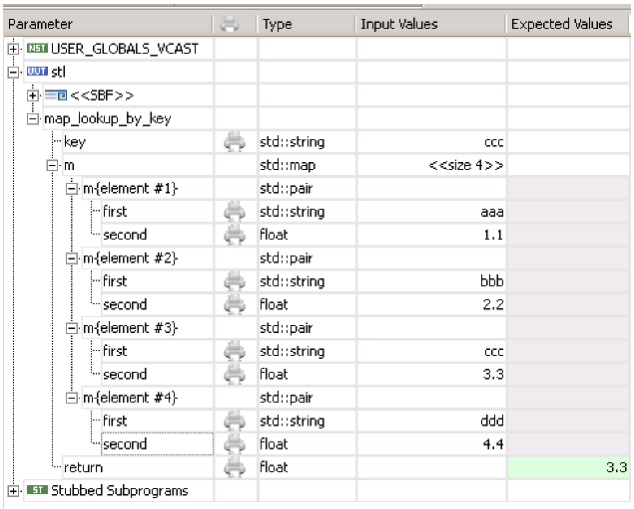
```
float map_lookup_by_key(const std::string& key, const std::map<std::string, float>&
m)
{
   std::map<std::string, float>::const_iterator i = m.find(key);
   if (i != m.end())
      return i->second;
   else
      return 0.0;
}
```

This method searches an STL *map* using the input parameter *key* as the search value.

When the size of the map is entered in the Parameter tree, each element appears as an STL *pair*.



Using the test case editor, each pair is expanded and data for the pairs is entered. After the expected result is entered the test case is ready to be executed. In this example, the search key is “ccc”, which is contained in the map. The expected value of 3.3 is returned and the test passes.



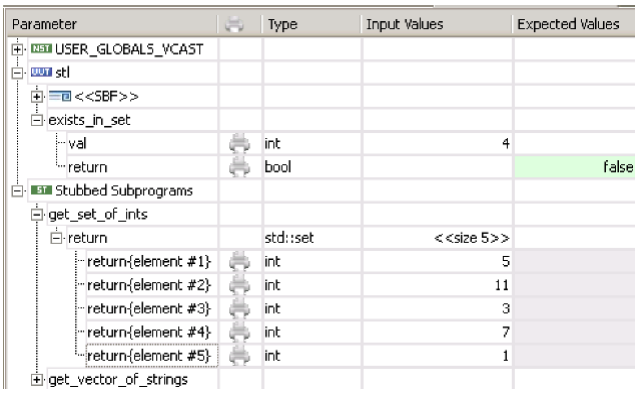
exists_in_set

```
std::set<int> get_set_of_ints();


bool exists_in_set(int val)
{
   std::set<int> s = get_set_of_ints();
   return s.find(val) != s.end();
}
```

This function searches for the input parameter value in an STL *set*. The function returns *true* if the value is found and *false* if not The function calls the forward referenced function *get_set_of_ints* that returns a set. Notice that there is no implementation provided for this function. VectorCAST

automatically creates a stub for get_set_of_ints in the Stubbed Subprograms section of the Parameter Tree. Using the test case editor, expand the stub and enter values for the elements of the set. To test the negative case, the search value *val* is set to 4. Since the value 4 is not contained in the set, the expected return value is set to false. When executed, the test passes.



Compare the order of elements of the set in the Parameter Tree with that in the Execution report.

Execution Results (PASS)

Start of exists_in_set.001

Event 1 - Called exists_in_set
UUT: stl.cpp
Subprogram: exists_in_set

Parameter	Type	Actual Value	Expected Value
val	int	4	
return	bool	false	

Event 2 - Stubbed get_set_of_ints
UUT: stl.cpp
Subprogram: get_set_of_ints

Parameter	Type	Actual Value	Expected Value
return			
[0]	std::set	1	
[1]	std::set	3	
[2]	std::set	5	
[3]	std::set	7	
[4]	std::set	11	

Event 3 - Return from exists_in_set
UUT: stl.cpp
Subprogram: exists_in_set

Parameter	Type	Actual Value	Expected Value
val	int	4	
return	bool	false	<match>

UUT Returned control to Driver, end of test case

Result - PASS

Expected Results matched 100% 1/1 PASS

The data shown in the Execution Report has been reordered. VectorCAST implements the stub using actual STL code. Because an STL set is defined to sort its elments from lower to higher values, this standard behavior for the container is displayed in the execution report.

STL Stack in the Parameter Tree

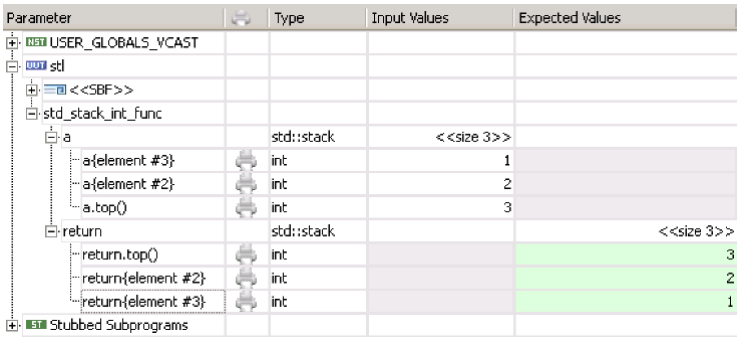
The following function accepts an STL *stack* of ints as its input and returns the very same stack.

```
std::stack<int> std_stack_int_func(std::stack<int> a)
{
    return a;
}
```

In the Parameter Tree shown below, the size of the stack is set to 3. Notice that the ordering of the elements of the stack is different for Input Values and Expected Values. VectorCAST treats and displays Input Value elements for the stack as if they were “pushed” onto the stack. The first data item added to the stack has the value of 1 and the last data item, also called “top” has the value of 3.

VectorCAST treats and displays Expected Values as if they were “popped”. The “top” item is the first to be retrieved.

The order of elements in the Parameter Tree reflects this “push-pop” relationship.



Working with Arrays

To Enter Values for Constrained Array Types

Array Types that contain a fixed number of components are called *constrained arrays*. Consider the following function prototype:

```
void test (int array_param [3]);
```

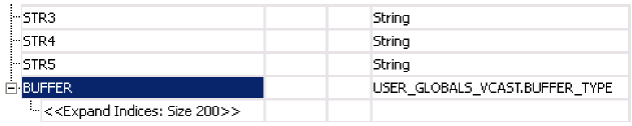
This array has three elements. You can enter an input and expected value for a specific array element, or set multiple array elements to the same value. The display of array parameters and global objects is “collapsed” by default, meaning that only a single entry appears in the Parameter Tree. You can “expand” this display and then enter data for each expanded elements, or you can specify a data value and apply it to specific elements. These methods are discussed below.

The purpose of expanding or collapsing array data is to enable you to control the size of the Parameter Tree.

To Expand All Elements of an Array

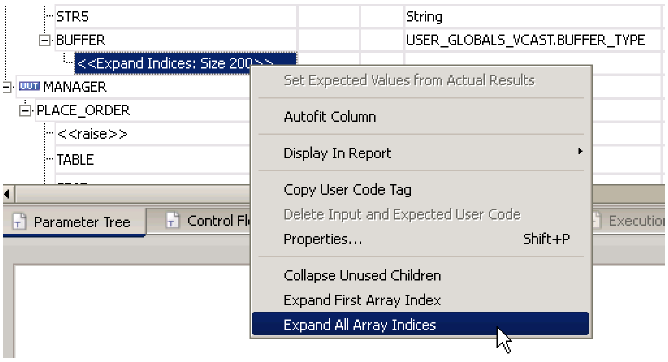
When a parameter is an array type, use the plus button to expand out the array. For example, expand <<USER_GLOBALS_VCAST>>, then expand <<GLOBAL>> and scroll down to see the user global array called BUFFER, an array of 200 elements, and click the plus button. The notation for unexpanded elements appears as:

<<Expand Indices: Size n>>, where n is the number of elements in the array.

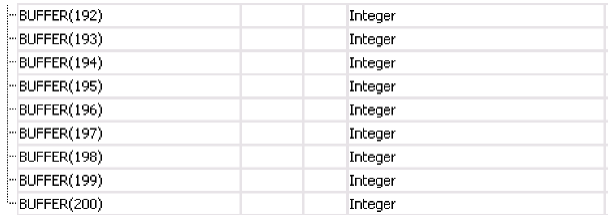


To expand all elements of the array, right-click on <<Expand Indices>> and choose **Expand All**

Array Indices.

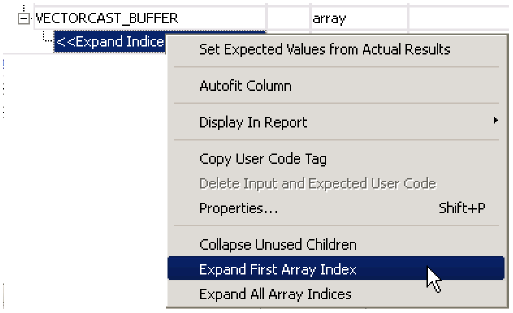


All array elements are expanded in the Parameter Tree.



To Expand the First Element of an Array

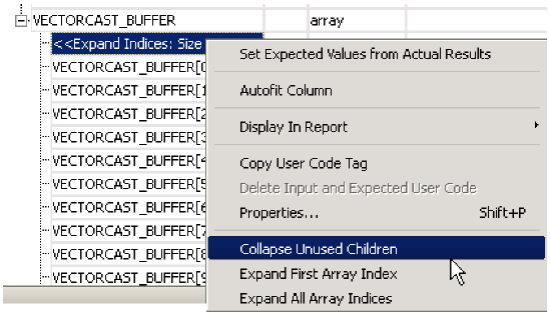
To expand only the first element of the array, right-click on <<Expand Indices>> and choose **Expand First Array Index**. The first array element is expanded in the Parameter Tree.



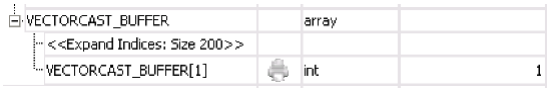
To Collapse Unused Elements of an Array

If you have expanded some elements of an array and you find that you don't need to enter Input or Expected data for them, you can collapse them in the Parameter Tree.

To collapse unused array elements, right-click on <<Expand Indices>> and choose **Collapse Unused Children.**



The array elements that do not have Input or Expected data are removed from the Parameter Tree. In the example below, all 200 elements of the VECTORCAST_BUFFER had been expanded, but data was entered for the element VECTORCAST_BUFFER [01]. When the unused array elements were collapsed, all array elements are closed except for index [01], which has data.



To Expand Certain Elements of an Array

There are several ways to expand (or display) certain elements of an array. In the Input Value column in the Parameter Tree tab, you can enter:

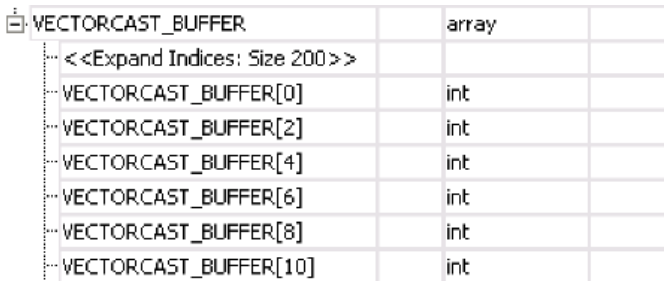
- a single index, such as 7
- a range, such as 0..199/2
- or a list, such as 1,5,45

To Expand the Odd Array Elements

In the example below, the odd array elements from 1 to 199 are specified by typing 1..199/2 in the Input Values column next to <<Expand Indices: Size n>>. Use the format <lower bound>..<upper bound>/delta.



When you press enter, the array expands.



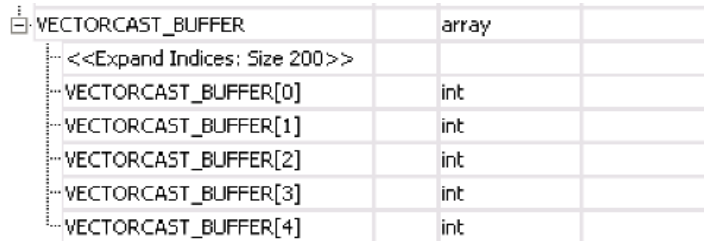
(and so on)

To Expand the First 5 Elements

The first five Elements are expanded by typing 0..4 in the Input Values column next to <<Expand Indices: Size n>>.



When you press enter, the array expands.

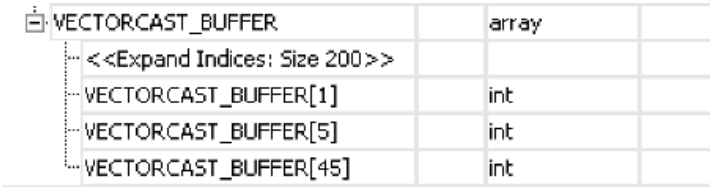


To Expand Array Indices 1, 5, and 45

Array elements 1, 5, and 45 are expanded by typing the list 1,5,45 in the Input Values column next to <<Expand Indices: Size n>>.



When you press enter, the array expands.



To Apply Values to an Array

If you bring up the Parameter dialog on an expanded array element, you can enter input values or expected values to one or more elements of an array. The Scalar Values, Range Values, and List Values tabs have an added section, called “Apply to Array Indices,” located near the bottom of the dialog.



Once you have enabled Input or Expected values and filled in the data, you can easily apply these values to multiple elements of the array. The choices are:

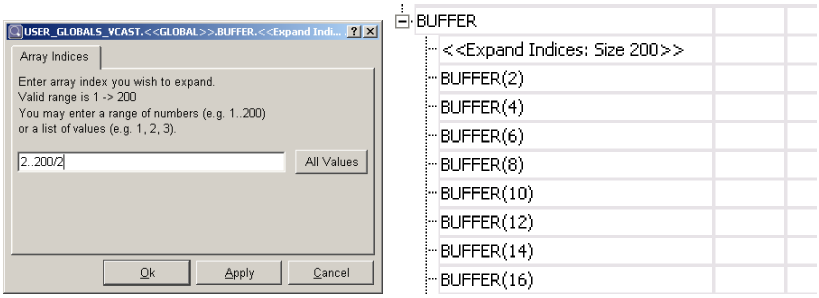
- **This** (default) to set the one element that you expanded and double-clicked
- **All** to set all elements of the array
- **Range** to set a range of elements, (x..y, or 1,3,5,7, or 1..200/2 etc.).

The **Apply to expanded array indices only** checkbox modifies the “All” or “Range” options by limiting their effect to the currently expanded elements in the Parameter Tree.

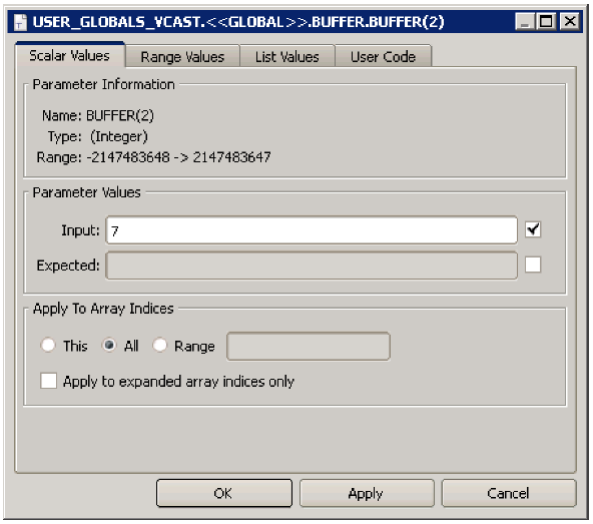
See "To Expand All Elements of an Array" on page 306 for more information on expanding array elements.

For example, if you expanded elements 0 to 199, even only, using the Expand Indices dialog, you could then assign a value using the “All” choice in the Parameter dialog. It would only apply to the even elements.

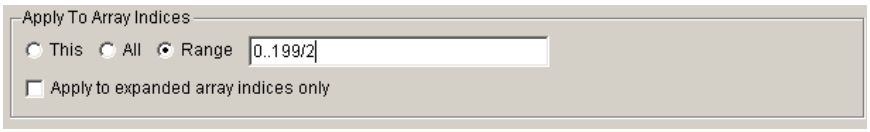
First, expand the even elements as shown below:



Then, double-click on any element to bring up the Parameter dialog. Set an input value, then click the **All** radio button and the checkbox next to **Apply to expanded array elements only.**



Alternatively, you could skip the step where you expand the even elements and instead expand the first element, double-click it to go directly to the Parameter dialog, and use the Range radio button with the expression 0..199/2.

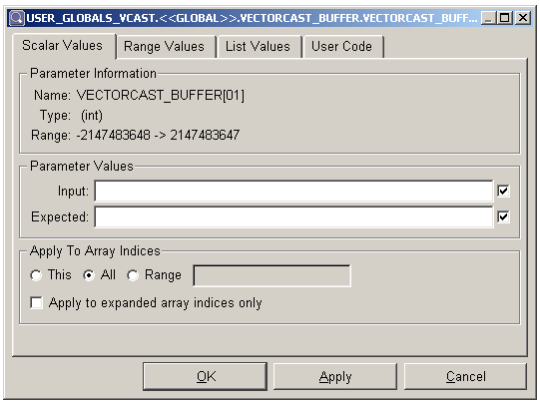


To Clear Values from an Array

To clear scalar, range, or list data from an array, do the following:

1. Check the Enable box(es).
2. Delete the text in the Input and/or Expected boxes.
3. Select the All or Range radio button.
4. Click **Apply**.

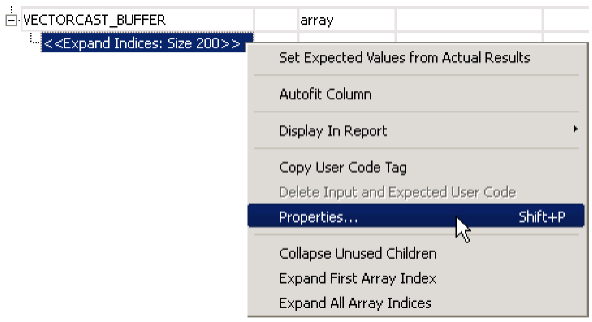
Doing so clears the data from all of the indicated range of array elements.



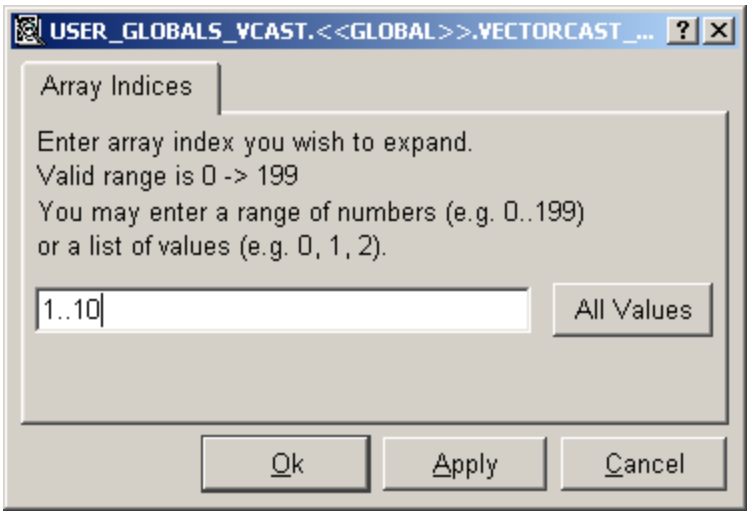
Using the Array Properties Dialog

There are two ways to access the Array Properties dialog:

- in the Parameter Tree, double-click on an array's <<Expand Indices: Size n>>
- in the Parameter Tree, right-click on an array's <<Expand Indices: Size n>> and choose **Properties...**



You can enter an expression to control which array elements are expanded or to expand the entire array, click the **All Values** button. The syntax of expressions that can be entered is described in the previous section.



Click **OK** to exit the Array Indices dialog and expand the array.

Using Enumerals to Size and Index an Array

In the case where an array is defined using an enumeral, VectorCAST allows test cases to be built where the size of the array is automatically defined by the number of elements within the enumeration.

For example, if you have the following definitions:

```
enum Beverages {NoBeverage, Wine, Beer, MixedDrink, Soda};
int DrinkPrices[MixedDrink];
```

All values of the Beverages enumeration can be used to index the DrinkPrices array. The benefit of this feature is that it allows test cases to be completely portable in cases where enumerals are added to the enumeration type after the VectorCAST test cases have been created.

The enumerals are displayed in the Test Case Editor, the Test Case Scripts, and the Execution Reports.

To disable this feature, use the Builder tab option: **Disable direct array indexing** or from clicast, use the following command:

```
clicast -lc option VCAST_DISABLE_DIRECT_ARRAY_INDEXING True

The default value is False.
```

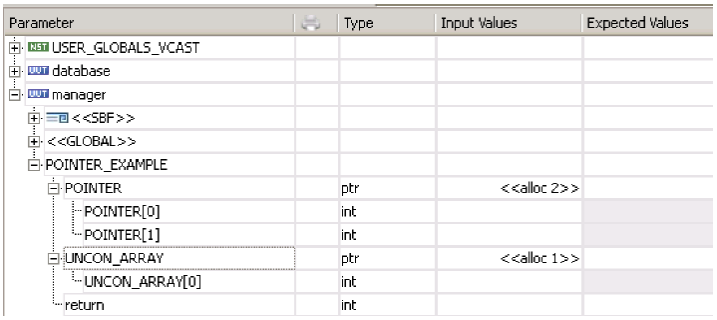
Unconstrained Arrays and Pointer Types

Array types that do not contain a defined number of components are called *unconstrained arrays*. Unconstrained arrays and pointer types are handled exactly the same by VectorCAST. In order to set values for these types, you must first allocate memory, and then set a value.

Consider the following C function prototype:

```
int POINTER_EXAMPLE (int* POINTER, int UNCON_ARRAY[]);
```

To set test data for either the POINTER parameter or the UNCON_ARRAY parameter, you must first choose the number of elements to allocate and then set values for each of the elements. A single element will be automatically added when a pointer type is double-clicked. In the following example, we have chosen to allocate 2 elements for the POINTER parameter, and 1 element for the UNCON_ARRAY parameter.



As you change the number of allocated elements, VectorCAST dynamically updates the Parameter Tree with additional rows that enable you to choose the values for each element.

Entering Data with the Parameter Dialog Box

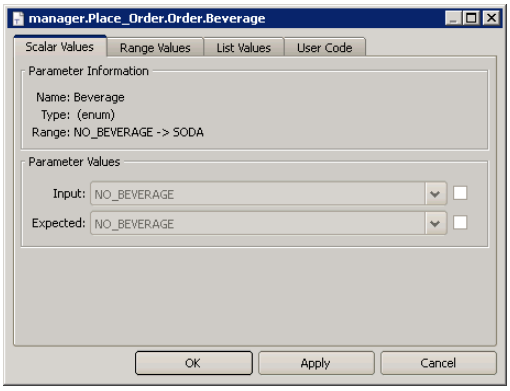
To Access the Parameter Dialog

The Parameter dialog gives you easy access to both Input Value and Expected Value of a parameter, as well as Min, Mid, and Max values, ranges of values, and lists of values.

There are two ways to access the Parameter dialog:

- in the Parameter Tree, double-click on any parameter or global object
- in the Parameter Tree, right-click on a parameter and choose **Properties...**

For example, double-clicking the parameter **`Beverage`**, in the data object Order, in the subprogram Place_Order, in the UUT manager brings up the Parameter dialog as shown below:



The Parameter dialog box contains the following four tabs:

- Scalar Values
- Range Values
- List Values
- User Code

All tabs contain a header which shows the base type of the data object as well as the valid range. Note that the valid range saves you from having to search through your source code files to find the range of a particular data item.

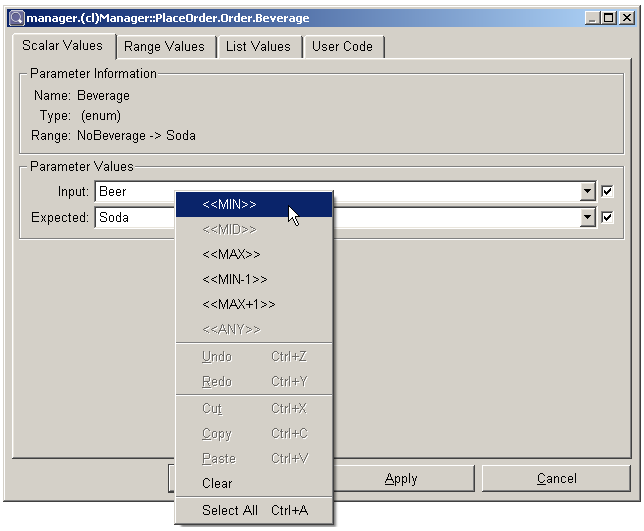
In all of the tabs, the small checkboxes next to the data values are used to "enable" the data value for the test case. Before entering an Input or Expected value, you must click on the checkbox to activate that value. Similarly, if you "unchecked" a value, it is deactivated and the value is removed from the test case.

The Scalar Values Tab

The Scalar Values tab enables you to define an individual value to be assigned to the data object. Right-click to set the **Min-1**, **Min**, **Mid**, **Max**, or **Max+1** values. (<<MID>> values are not allowed for enumerated parameters.)

When you enable the Expected checkbox, the **Any** menu item also becomes available. It is used only with expected values, and indicates that you want to allow any value for that parameter's expected

value. Since <<Any>> is a data value, it is recorded in the test execution results. Use <<Any>> if you want to see the value of a non-global parameter in the test execution results without making an actual comparison.



The items in the right-click menu depend on the data type of the parameter. Floats have <<ZERO>>, <<POS_INF>>, <<NEG_INF>>, and <<NAN>>. Strings only have <<ANY>> (for expected values). Enumerals do not have <<MID>> or <<ZERO>>.

To Apply Values to an Array

If you bring up the Parameter dialog on an expanded array element, you can enter input values or expected values to one or more elements of an array. The Scalar Values, Range Values, and List Values tabs have an added section, called “Apply to Array Indices,” located near the bottom of the dialog.



Once you have enabled Input or Expected values and filled in the data, you can easily apply these values to multiple elements of the array. The choices are:

- **This** (default) to set the one element that you expanded and double-clicked
- **All** to set all elements of the array
- **Range** to set a range of elements, (x..y, or 1,3,5,7, or 1..200/2 etc.).

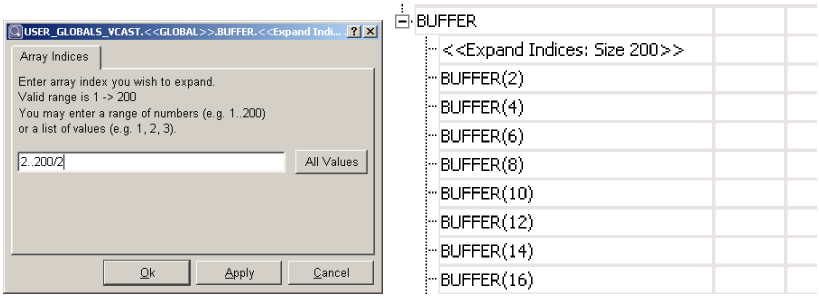
The **Apply to expanded array indices only** checkbox modifies the “All” or “Range” options by limiting their effect to the currently expanded elements in the Parameter Tree.

See "To Expand All Elements of an Array" on page 306 for more information on expanding array elements.

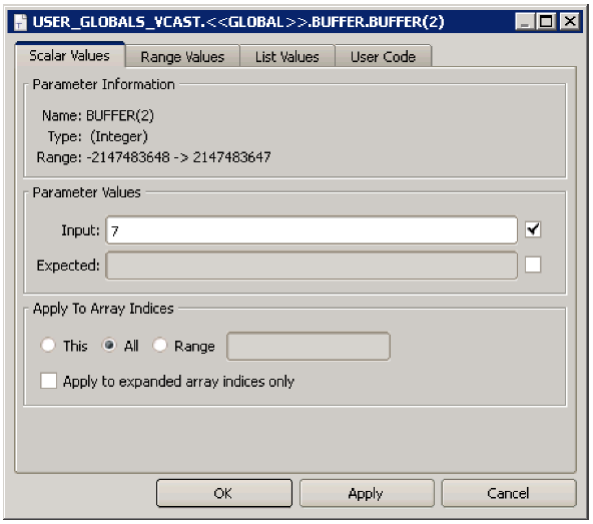
For example, if you expanded elements 0 to 199, even only, using the Expand Indices dialog, you

could then assign a value using the “All” choice in the Parameter dialog. It would only apply to the even elements.

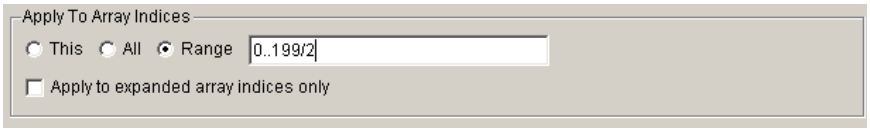
First, expand the even elements as shown below:



Then, double-click on any element to bring up the Parameter dialog. Set an input value, then click the **All** radio button and the checkbox next to **Apply to expanded array elements only.**



Alternatively, you could skip the step where you expand the even elements and instead expand the first element, double-click it to go directly to the Parameter dialog, and use the Range radio button with the expression 0..199/2.



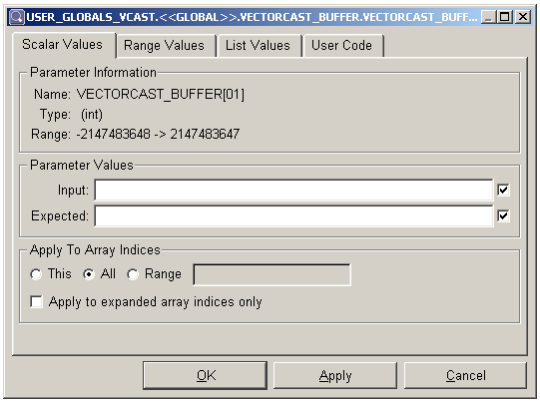
To Clear Values from an Array

To clear scalar, range, or list data from an array, do the following:

1. Check the Enable box(es).
2. Delete the text in the Input and/or Expected boxes.

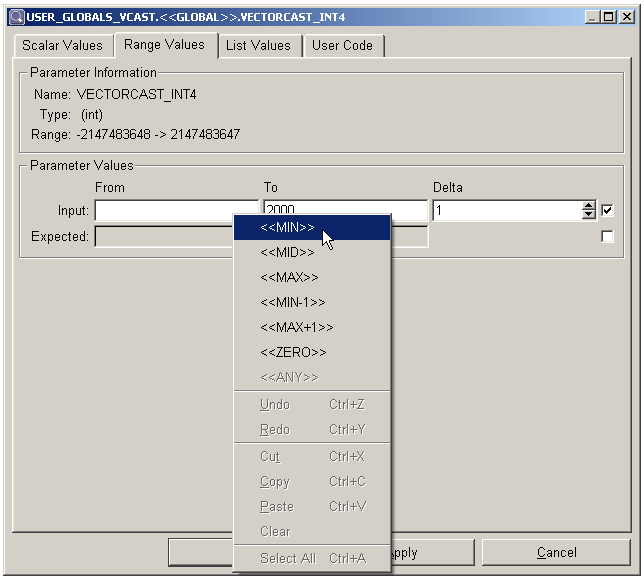
3. Select the All or Range radio button.
4. Click **Apply**.

Doing so clears the data from all of the indicated range of array elements.



The Range Values Tab

The Range Values tab enables you to build a test case that will automatically exercise a parameter over a specified range of values. As with scalar values, once you have enabled, or "activated" the input or expected value, then right-click to special values <<Min-1>>, <<Min>>, <<Mid>>, <<Max>>, or <<Max+1>>. If the data type is enum, then the actual minimum, mid, or maximum value is displayed once you click **Apply** or **OK**.



The **From** edit box is the first value used, and the **To** edit box contains the last value used each time the subprogram is called. These two numbers represent the range, and the **Delta** is used to get from the first number to the second. For example, to vary an input from 1 to 6 by 2, use:

From: 1

To: 6

Delta: 2

Input values used for parameter: 1, 3, 5

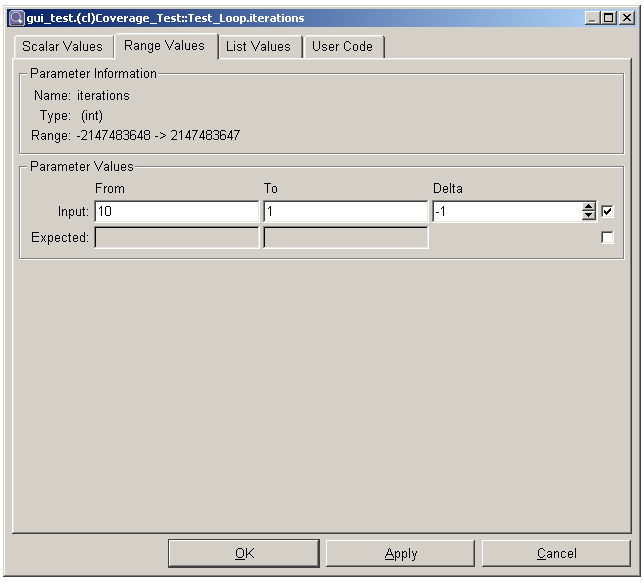
To enter a descending range, use a negative delta:

From: 10

To: 1

Delta: -1

Input values used for parameter: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1.



See also "To Apply Values to an Array" on page 315.

To Set Up an Input Range or List for More Than One Parameter

When setting up a test case for range testing, you may vary different parameters simultaneously, in parallel or in combination. This means that different range expressions may exist within a single test case. A range in the Input Values column causes the test case to iterate. This capability enables you to vary multiple values simultaneously, or vary one value while holding the others constant. The default for the maximum number of parameters to vary is 20. This value can be modified using the “Maximum varied parameters” option, located on the **Target** tab and the **Harness Size Options** sub-tab of the **Tools => Options** dialog.

To Use Ranges as Expected Values

The Range tab can also be used to set an expected value. When used this way, an actual value that is between the “From” and “To” values, inclusive, is considered a match when the test case runs.

The List Values Tab

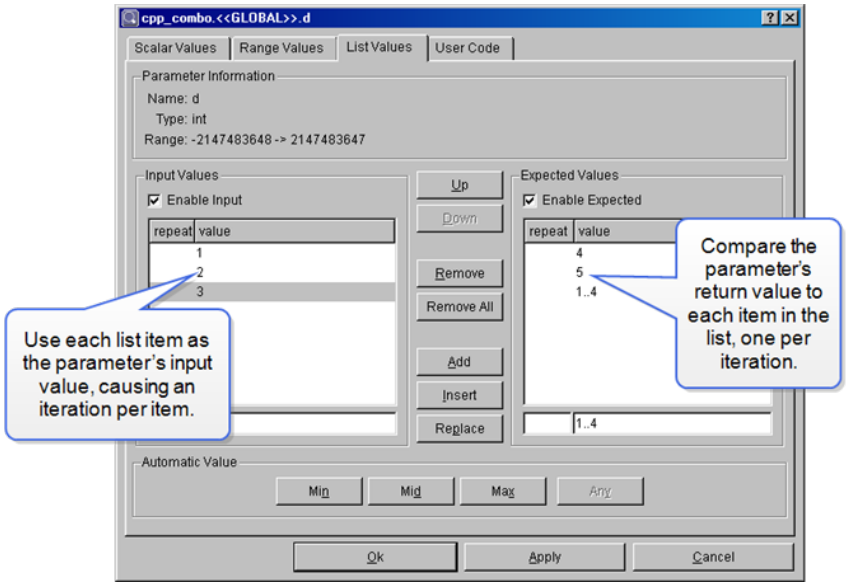
The List Values tab enables you to provide a series of values to be used when calling the unit under test or to provide a sequence of specific scalar values to be returned from stubs when a stub is called multiple times during a test case execution. A list is similar to a range, except the items in a list can have different deltas between them, so you specify each item in the list. Items in a range are always separated by the same delta, so you only need specify the beginning and end items, and the delta.

When you add items to the Input Values list for a parameter, you are assigning a series of input values to that parameter. Adding more than one item to the list causes an iteration, that is, the test case is executed once for each item. Repeating a value in the list causes additional iterations.

Using an Input Values list for a stub parameter means that the stubbed subprogram returns one item from the list each time it is called. If a stub is called more time than you have provided values for in your list, then the last value in the list is used repeatedly, once the list items are exhausted. Each time a test case iterates, the Input Values list for a stub is reset to the beginning.

See also "Return Data List Options" on page 463 for information on how to make the Input Values list span iterations in a test case or a slot in a compound.

When you add items to the Expected Values list for the same or a different parameter, you are assigning a series of expected values for that parameter. One expected value is compared to the parameter's value per test iteration. The expected value can be a range of acceptable values for that iteration, as described below.



Controlling Values in the List

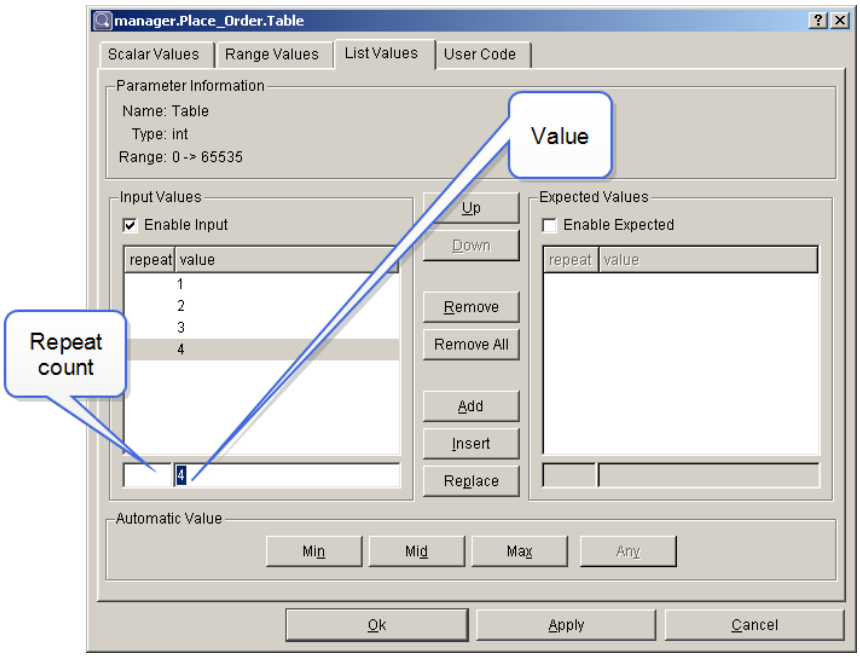
To enter values in a list, click the checkbox next to "Enable Input" or "Enable Expected." The edit boxes below become active, allowing you to type a value. To add a value, enter a value in the edit box below the list area, and click the **Add** button or press the **Enter** key. Repeat to add more entries to the bottom of the list.

To insert a new entry into an existing list, select an item in the list, enter a new value, and click the

Insert button. The new entry appears below the selected one. To change an entry already in the list, first select it. Make the change in the text area below (either in the repeat box, the value box, or both), and click the **Replace** button. The entry in the list is replaced with the new values.

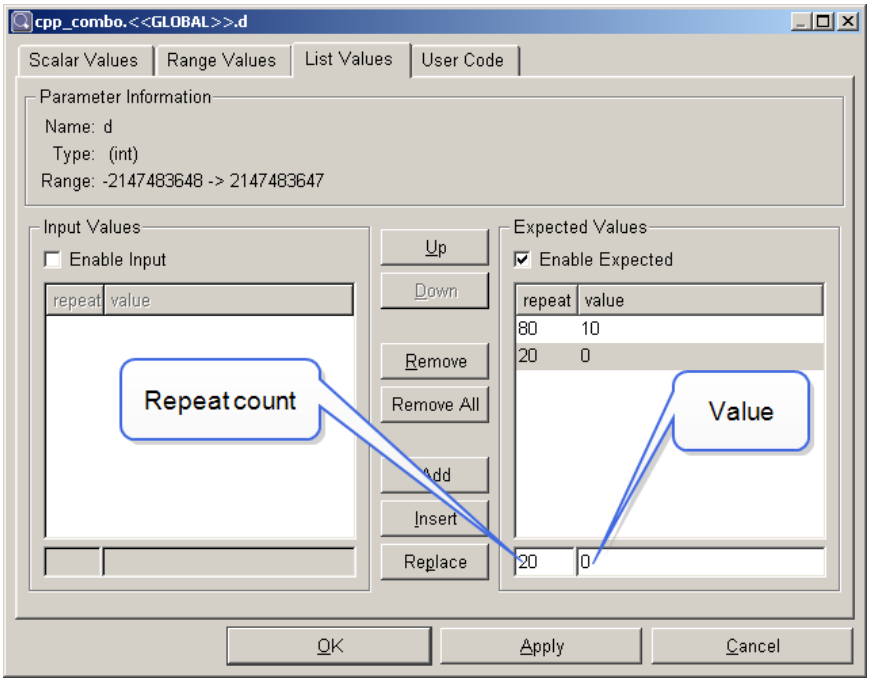
An entry can be moved up or down in the list by selecting the entry and clicking the **Up** or **Down** button.

An entry can be deleted by clicking the **Remove** button. When you click the **Remove All** button, all entries are deleted from the list.



To Repeat Values in the List

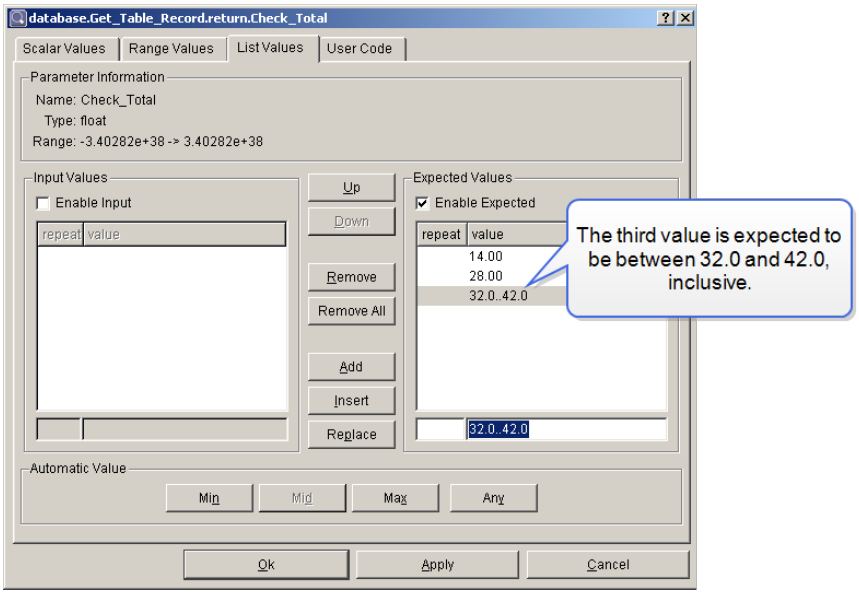
Each value in a list can have an optional repeat count. The repeat count enables you to easily manage lists of values when there are repeating values in the list. For example, if you want to have a list of 100 integers where the first 80 values are 10 and the next 20 values are 0, you enter these values using the repeat count as shown below.



To Use a Range Expression in an Expected Value List

When working with integer or float values, you may also provide a range expression in the list of expected values. Simply type in the data in low..high format and that range is added to the list. When an actual value is compared to the range, it matches if it is in the range.

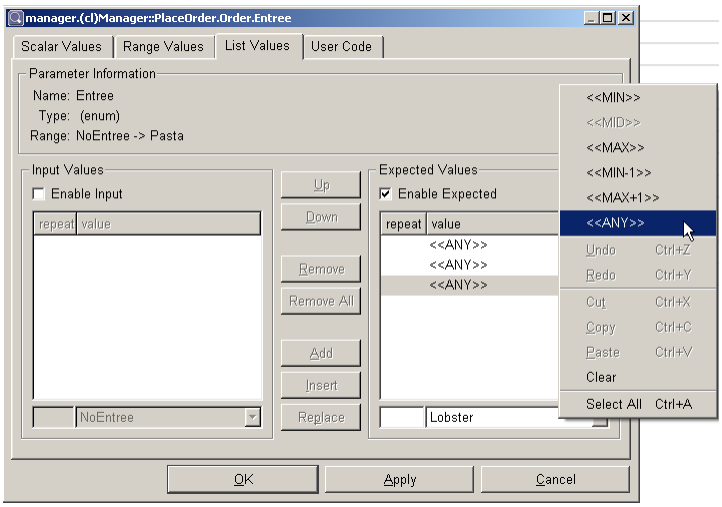
Ranges are not permitted in the Input Values list. If you need an input range, you can express it as an input list, or just go to the Range Values tab and add the input range there.



To Use the <<Any>> Tag in a List

When you provide a list of expected values, VectorCAST uses each value in the list in the order it is encountered. In some cases you may not care what the value of an object is until a specific point in the test execution. The **Any** tag enables you to indicate an "I don't care" state for the data value. For example, the following screen shot assumes that the Entree field in parameter Order will be encountered 4 times, and the tester only cares what the value is on the fourth iteration.

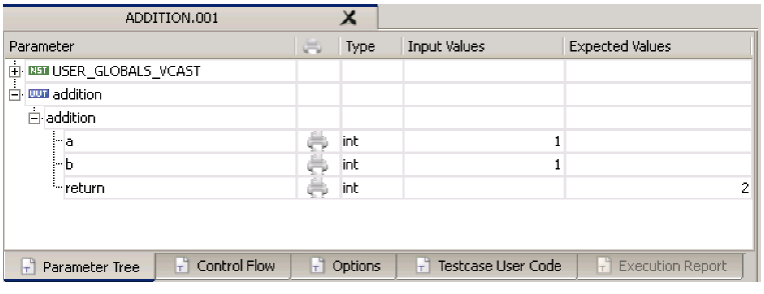
A value of <<ANY>> does not get compared; it is merely a placeholder, so that other values are compared on the desired iteration.



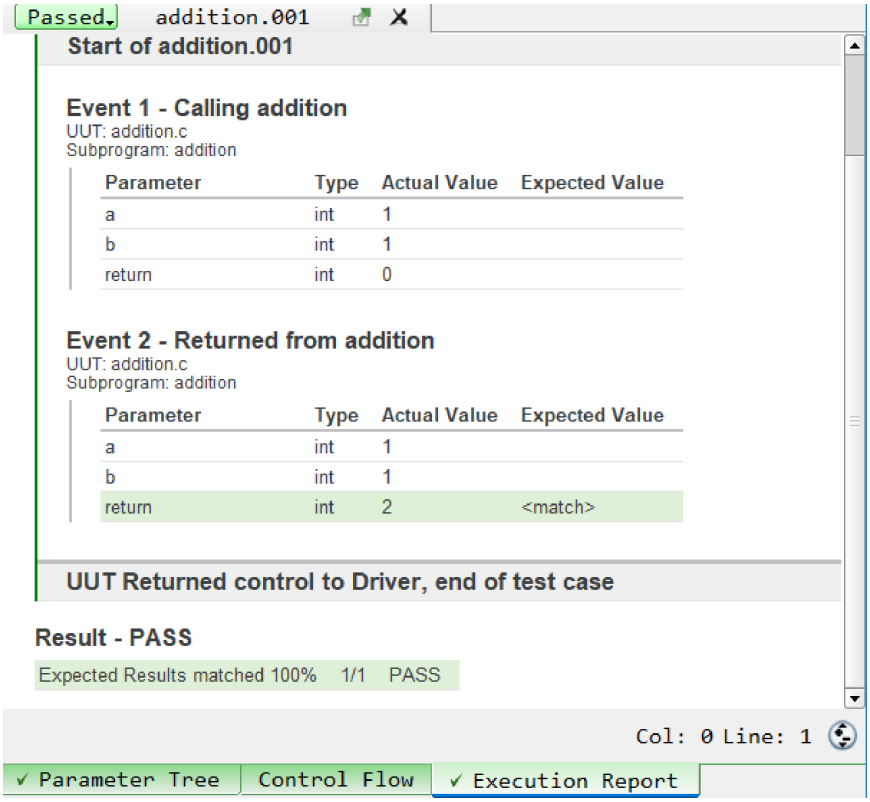
Range and List Example

To better understand the functionality of the list, we'll use a simple example. An environment is built using the following source code as the UUT. A test case for the subprogram addition is inserted.

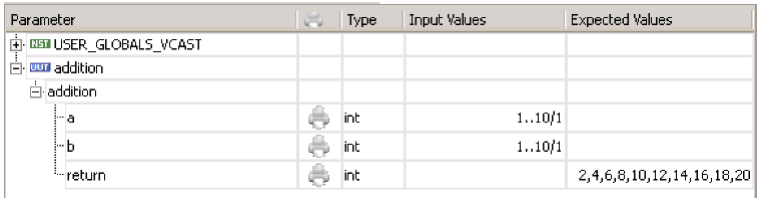
```
int addition( int a, int b)
{
return = a + b;
}
```



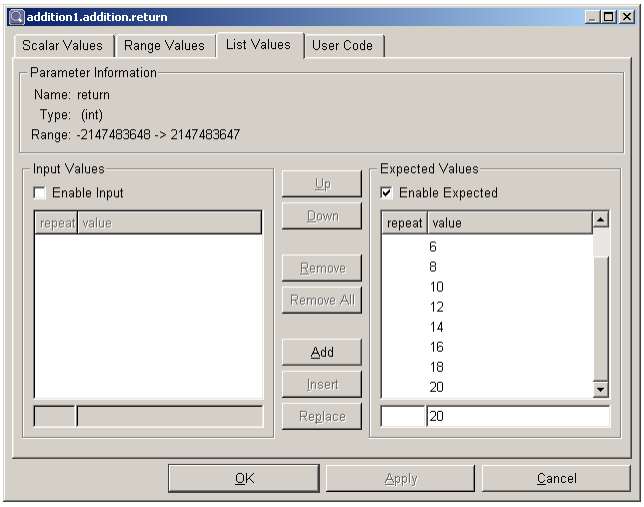
The values **a** and **b** are assigned an input value of 1. We can expect the return to have a value of 2 when the test case is executed.



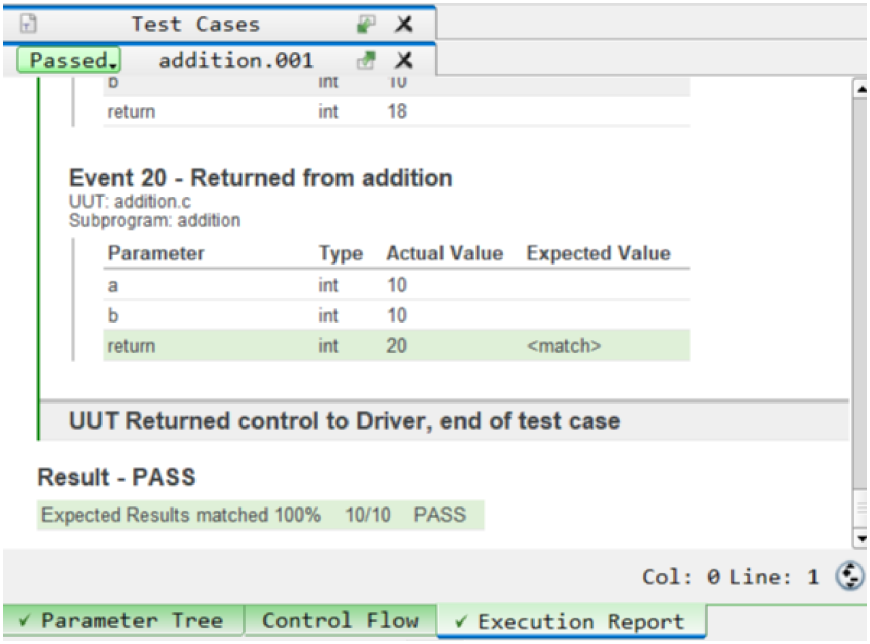
But we really want to test this code across a range of values to make sure that the code behaves properly. What we would like to do is to vary **a** from 1 to 10 and vary **b** from 1 to 10, and verify the performance of our function. To implement this test, we will use a range expression for parameters **a** and **b**, and a list of expected values for the return.



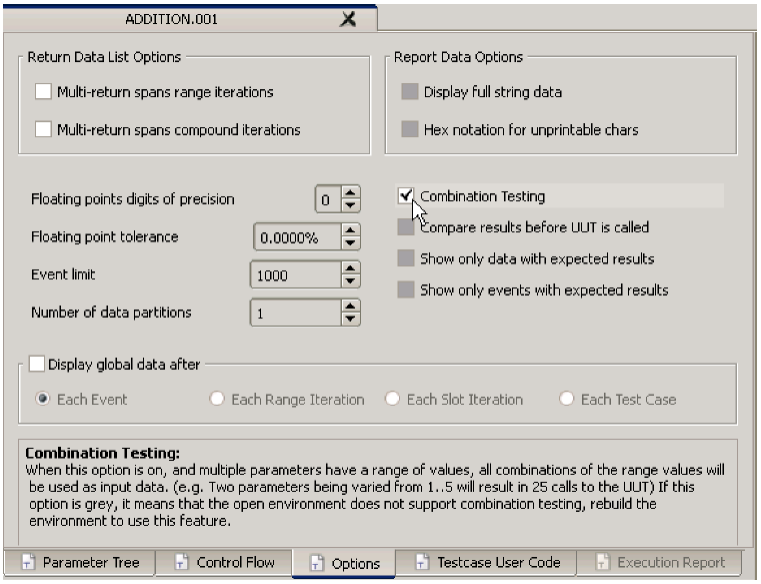
To enter the list values for the **return** parameter, you can either type the list **2,4,6,8,10,12,14,16,18,20** into the Expected Values column, or use the List tab, as shown below.



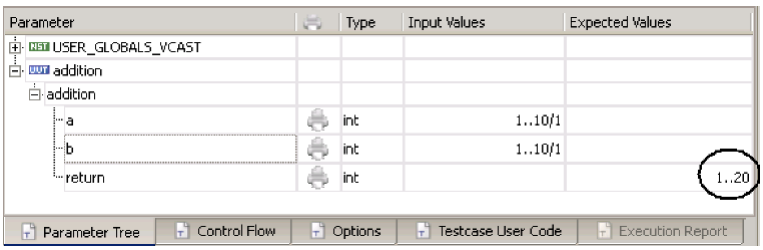
When this test is executed, all ten inputs match. There are 10 range iterations.



To verify that the function handles every combination of inputs correctly, we turn on the option **Combination testing.** This option is accessible on the **Tools => Options** dialog, **Execute** tab, and on the **Test Case Options** tab, as shown below for the test case ADDITION.001.



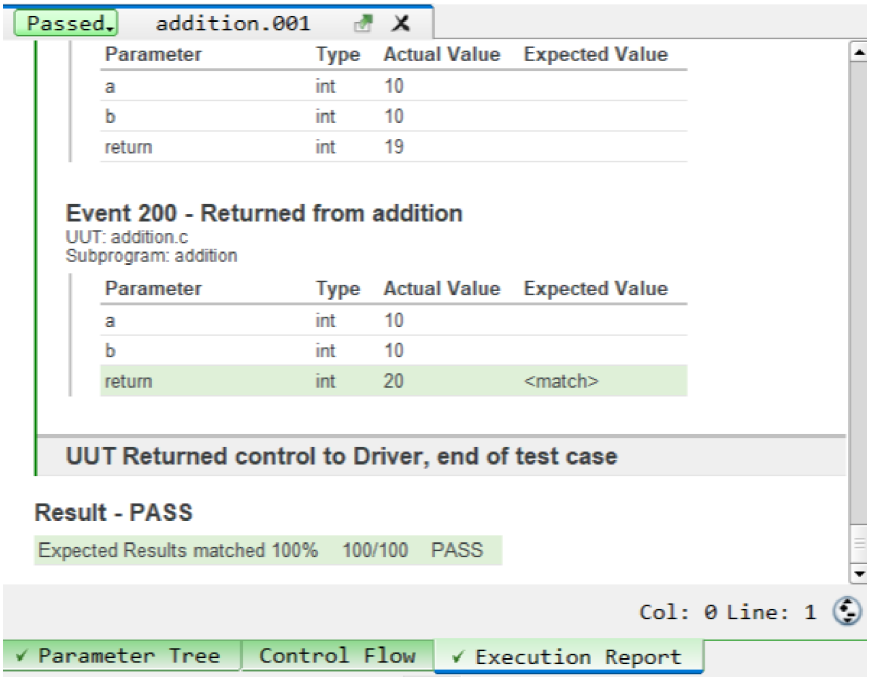
With this option on, each combination of the two input ranges is used as an input to the function **`addition`**. Each input for **`a`** is paired with each value of **`b`**, then the next value of **`a`** is paired with each value of **`b`**, and so on. As a result, we need 100 expected values in the list for the **`return`** parameter.



a	**b**	**return**
1	1	2
1	2	3
1	3	4
1	4	5
1	continue up to 10...	continue up to 11...
2	1	3
2	2	4
2	continue up	continue up to

	to 10...	12...
3	1	4
3	2	5
3	continue up to 10...	continue up to 13...
...	...	...
10	8	18
10	9	19
10	10	20

When the test case is executed with **`Combination testing`** on, all 100 expected values match, as shown below. There are 100 range iterations.



Test Case Scripting

In addition to being used to archive test cases for regression use, VectorCAST test scripts enable you to build test cases using ASCII text files. Scripting can be an alternative to entering test case data interactively, via the VectorCAST GUI.

Note: Test Case Scripting is an advanced workflow which can have unintended side effects and should be used with caution.

VectorCAST Scripting complements the normal test case creation process. With Scripting you can create a single test case using the VectorCAST GUI, export the script file, edit the file to contain multiple similar test cases, and import the script to quickly generate tens or hundreds of test cases.

Additionally, test case scripting enables you to import data from other tools, such as MATLAB™, that may contain high precision models or data sets.

To Export Test Cases to a Test Script

The **Test => Scripting => Export Script** command provides a means to output test case data to a test script.

The Export Script command is sensitive to what is currently selected in the Test Case Tree. You can click (or multi-select) nodes from the Test Case Tree, then choose **Test menu => Scripting => Export Script**. When you click **Save**, VectorCAST saves the file in the working directory (by default) with the extension .tst. The file contains test data for all selected test cases.

```
clicast -e <env> [-u <unit> [-s <sub> [-t <test>]]] TESt Script CReate
<script file>

Create (export) test script file from existing test case(s). If only the
environment parameter is specified, the script will contain data for all
test cases (including <<init>> and <<compound>>). If the unit parameter is
specified, the script will contain data for all testcases for that unit. If
the unit and subprogram parameters are specified, the script will contain
data for all testcases for the specified subprogram. If unit, subprogram,
and testcase parameters are specified, the script will contain data for the
specified test case.
```

See also "Strict Test Case Importing" on page 442.

See also “Test Script Language” on page 678.

Duplicate Names in Exported Scripts

If any of the test cases in Unapplied Test Script Data have the same name as existing test cases, then the exported Unapplied Test Script Data test cases are assigned new names. This name follows the default naming format described in "Test Case Naming" on page 248.

Any name changes of exported tests from Unapplied Test Script Data are propagated through the compound test slots in Unapplied Test Script Data.

Test Script Organization

VectorCAST generates test scripts in alphabetical order. If different versions of source code define their functions in differing order, the test scripts can still be easily compared.

<<INIT>> testcases are placed at the top of the file in order by test case name. Next, units are listed in alphabetical order; within each unit, the subprograms are listed in alphabetical order, and then testcases for each subprogram are listed alphabetically. At the end of the file <<COMPOUND>> tests are listed alphabetically by test name.

This feature works from both the command line (where scripts are generated from the environment,

unit, subprogram, or testcase level) as well as from the GUI (where scripts are generated by selecting any combination of unit, subprogram, and/or individual testcase).

To Clear Unapplied Test Script Data

To remove any unapplied test script data from the environment, right-click the Unapplied Test Data node in the Test Case Tree and select **Delete Unapplied Test Data Script** from the menu.



Note: This permanently removes the unapplied test script from the environment and cannot be undone.

```
clicast -e <env> [-u <unit> [-s <sub> [-t <test>]]] TESt UNapplied Clear

Clear unapplied test script data from environment.
```

To Import Test Cases from a Test Script

The **Test => Scripting => Import Script** command enables you to import a test case script. When you choose **Test => Scripting => Import Script**, the File Open dialog appears. Here, select one or more test scripts (.tst files) and click Open. The test scripts are loaded into the environment, creating new test cases.

When a test script is imported, whether by you or as part of another operation (such as Updating or rebuilding an environment), the Test Script Log appears in the MDI window. If any test cases had to be renamed or other errors occurred, they are documented in the Test Script Log.

The Test Script Log is displayed in the MDI window after a test script is imported, using **Test => Scripting => Import Script** or during another operation, such as rebuilding or updating the environment. To view the Test Script Log at any other time, choose **Test => Scripting => Import Log**.

Duplicate Names in Imported Scripts

If any of the test cases in the script have the same name as existing test cases, then the imported test cases are assigned new names. This name follows the default naming format described in *Test Case Naming*.

If an imported compound test case includes an individual test case that must be renamed, the new name is propagated to the compound test case, so that the imported test cases remain consistent.

```
clicast -e <env> TESt Script Run <scriptfile>

Load (import) test cases from a script file.
```

```
clicast -e <env> TESt Script Log

Display to standard output the Test Script Log that was generated by the
last test script import operation.
```

See also "Strict Test Case Importing" on page 442.

See also "Test Script Language" on page 674.

Logic Effects on Test Script Import

When importing a test script, the assigned logic of a test is implemented as follows:

- No logic assigned or assigned logic evaluates to Enabled.
  - Normal import process, meaning that any data that can be imported is imported and anything else is relegated to Unapplied Test Script Data.
- Assigned logic evaluates to Disabled.
  - On import error, regardless of error type, the entire test is sent to Unapplied Test Script Data.

To Create a Test Case Template

The **Test => Scripting => Template** command builds a test case script that contains all possible input and output data for a particular environment. The templates contain all global data, formal parameters of the unit(s) under test, and formal parameters of any stubbed dependent unit. The purpose of the test script template is to show you the correct syntax for every possible TEST.VALUE line for the environment. In addition to showing the command syntax, each line shows the range of allowable values for each parameter.

```
clicast -e <env> TESt Script Template <scriptfile>

Create a test script template containing entries for each parameter and
global object in the environment, along with range information, and save it
in <scriptfile>.
```

The following template excerpt is from the manager.c tutorial:

```
-- Test Case Script
--
-- Environment    : TEST
-- Unit Under Test: manager
--
-- Script Features
TEST.SCRIPT_FEATURE:CPP_CLASS_OBJECT_REVISION
TEST.SCRIPT_FEATURE:MULTIPLE_UUT_SUPPORT
TEST.SCRIPT_FEATURE:STANDARD_SPACING_R2
--
-- Values are mandatory for SUBPROGRAM and NAME.
-- Delete any TEST.VALUE or TEST.EXPECTED commands not used.
--
TEST.SUBPROGRAM:
TEST.NEW
TEST.NAME:
TEST.NOTES:
TEST.END_NOTES:
... USER_GLOBALS_VCAST here...
```

```
TEST.VALUE:manager.Add_Included_Dessert.Order:1
TEST.VALUE:manager.Add_Included_Dessert.Order[0].Soup: NO_SOUP..CHOWDER
TEST.VALUE:manager.Add_Included_Dessert.Order[0].Salad: NO_SALAD..GREEN
TEST.VALUE:manager.Add_Included_Dessert.Order[0].Entree: NO_ENTREE..PASTA
TEST.VALUE:manager.Add_Included_Dessert.Order[0].Dessert: NO_DESSERT..FRUIT
TEST.VALUE:manager.Add_Included_Dessert.Order[0].Beverage: NO_BEVERAGE..SODA
TEST.VALUE:manager.Place_Order.Table: 0..65535
TEST.VALUE:manager.Place_Order.Seat: 0..65535
TEST.VALUE:manager.Place_Order.Order.Soup: NO_SOUP..CHOWDER
TEST.VALUE:manager.Place_Order.Order.Salad: NO_SALAD..GREEN
TEST.VALUE:manager.Place_Order.Order.Entree: NO_ENTREE..PASTA
TEST.VALUE:manager.Place_Order.Order.Dessert: NO_DESSERT..FRUIT
TEST.VALUE:manager.Place_Order.Order.Beverage: NO_BEVERAGE..SODA
TEST.VALUE:manager.Place_Order.return: -2147483648..2147483647
...
```

Troubleshooting Test Script Template

Range Checking Set to None Before Creating Template

If you have Range Checking set to None in the **Tools => Options** dialog, **Execute tab** when you rebuild the environment , the template generated looks like this:

```
TEST.SUBPROGRAM:
TEST.NEW
TEST.NAME:
TEST.NOTES:
TEST.END_NOTES:
TEST.VALUE:MANAGER.PLACE_ORDER.TABLE: <<unknown>>..<<unknown>>
TEST.VALUE:MANAGER.PLACE_ORDER.SEAT: <<unknown>>..<<unknown>>
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.SOUP: <<unknown>>..<<unknown>>
TEST.VALUE:MANAGER.PLACE_ORDER.ORDER.SALAD: <<unknown>>..<<unknown>>
...
```

The reason there are <<unknown>> entries for the values is because VectorCAST does not determine the range for parameters if **`Range Checking`** is set to None. Change the setting to **All** or **Disable**, rebuild the range data, and generate the template again.

Maximum Number of Scalars Exceeded Before Creating Template

Another case in which the template information will not be generated, is if the total number of scalar data items in the test environment is greater than the setting for **`Maximum lines in EDG range file`** on the **Tools => Options** dialog, **Builder tab**. In this case the generated template file will contain the following text:

```
-- Test Case Script
```

```
--
-- Environment    : NAME
-- Unit Under Test: unit
TEST.UNIT:
TEST.SUBPROGRAM:
TEST.NEW
TEST.NAME:
TEST.NOTES:
TEST.END_NOTES:
TEST.END
--
```

The lack of TEST.VALUE and TEST.EXPECTED lines, or having only a few of them, means that the setting for the maximum number of lines in the range file was too low, so the range file could not be expanded. To overcome this situation, set the value of the option higher, rebuild the range data **(Environment => Rebuild Range Data)**, and then generate the template again.

Automating Test Script Maintenance

VectorCAST provides the VectorCAST Test Script Maintenance Utility which is automatically invoked when rebuilding an environment to maintain existing test scripts across source code changes. At the beginning of environment rebuild, the utility makes a copy of the original test script in the environment directory, naming it **`convertScript-original.tst.`** It then uses the changes in the source code to translate any modified subprogram names, parameters, and global objects to their new names, and writes a converted test script as **`convertScript-translated.tst.`** At the end of environment rebuild, this converted test script is imported.



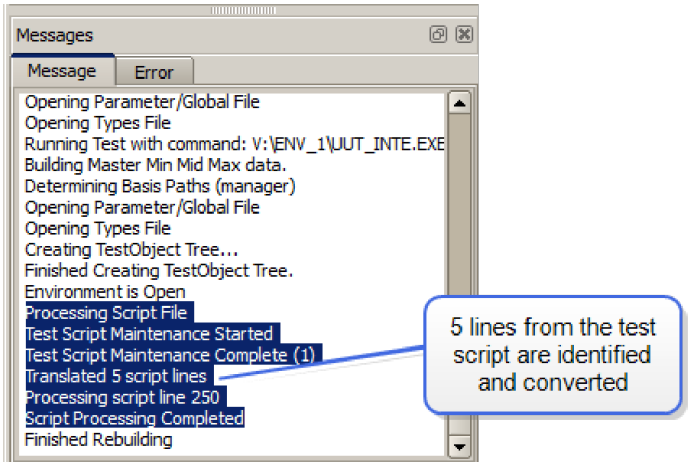
Note: If the option VCAST_DISABLE_TEST_MAINTENANCE is TRUE, then the VectorCAST Test Script Maintenance Utility is disabled.

Using the Test Script Maintenance Utility

The Test Script Maintenance Utility is invoked automatically during environment rebuild. The following example shows the Test Script Maintenance Utility doing an automatic conversion where only one field differs and there is no ambiguity concerning which conversion to choose.

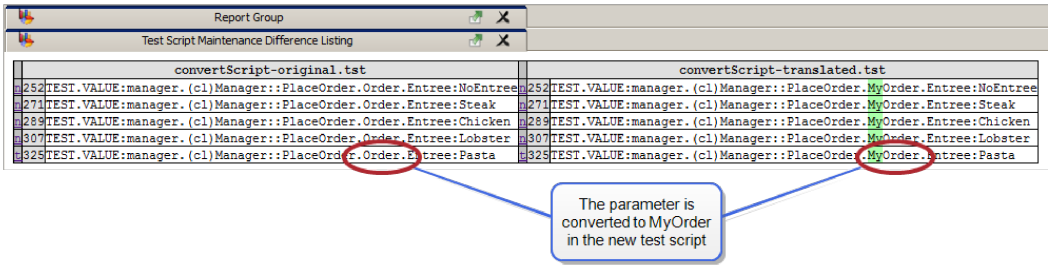
To automatically convert test scripts, you must first have an existing environment with test cases. Next, upload a new version of the source code and select **Environment => Rebuild Environment** from the Menu Bar.

As the environment rebuilds, the Message window shows the progress of the automated test script conversion.



Upon completion of the rebuild, both the Test Script Log and the Test Script Maintenance Difference Listing are displayed. The Test Script Log can be accessed at any time from the Menu Bar by selecting **Test => Scripting => Import Log**.

The Test Script Maintenance Difference Listing shows the changes made in the original script. Note in our example that the five translated script lines are listed and that the parameter "Order" has been converted to "MyOrder". The Test Script Maintenance Difference Listing can be accessed at any time from the Menu Bar by selecting **Test => Scripting => Test Script Maintenance Difference Listing**. This GUI-only report is located in the environment directory and is named **`convertScript-differences.html`**.



The original test script file is saved in the environment directory as **`convertScript-original.tst`**. The new converted test script file is also saved in the environment directory as **`convertScript-translated.tst`**.

Using the Test Script Maintenance Utility With Multiple Solutions

If the comparison of the old and new files reveals multiple solutions to a conversion, the utility prompts the user to choose the translation. For example, if the comparison reveals the following lines:

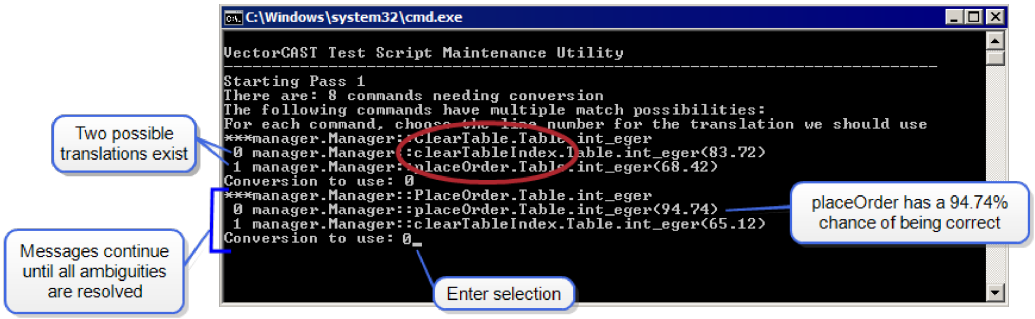
```
OLD: manager.Manage::ClearTable.Table.int_eger

NEW: manage.Manage::clearTableIndex.Table.int_eger

NEW: manager.Manage::placeOrder.Table.int_eger
```

The Test Script Maintenance Utility recognizes that there are two possible translations, **`manager.Manage::ClearTable.Table.int_eger`** or **`manager.Manage::placeOrder.Table.int_eger`**, but does not know which one to use.

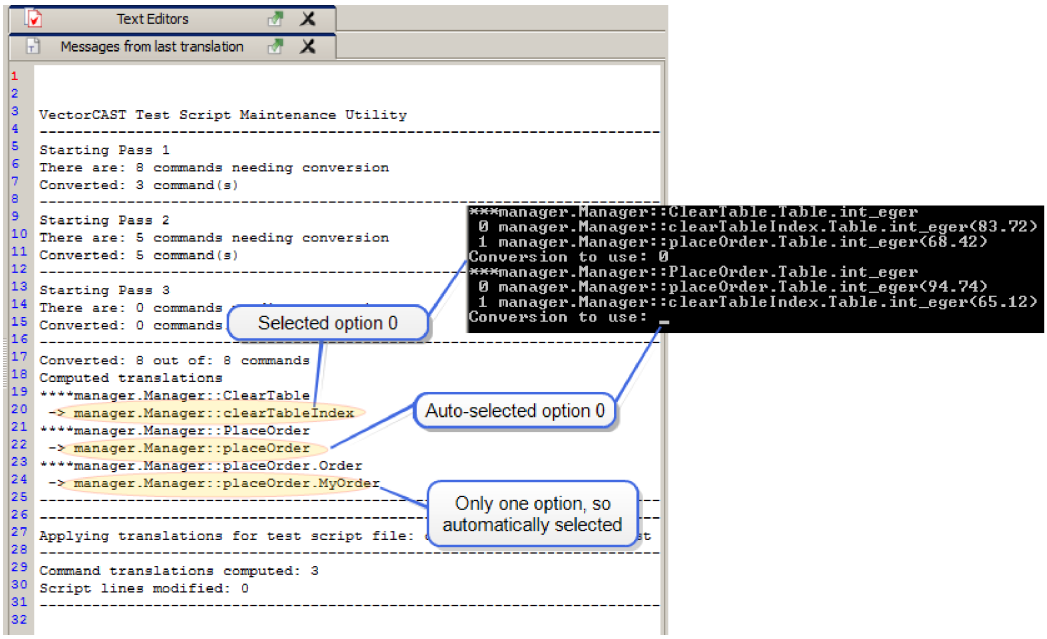
When using the Test Script Maintenance Utility in the GUI, a prompt is displayed allowing the user to resolve the ambiguity and select the translation. Each choice is displayed along with a score indicating the likelihood of the choice being correct. The list is sorted with the highest score first.



Note: The highest score is auto-selected when using the command line tools for regression testing.

The user enters the number of the desired choice, or, if no selection is made and the user hits **Return**, the highest score is automatically selected. The message window remains open and cycles through until all the ambiguities are resolved.

To see the full set of status messages after the message window closes, select **Test => View => Scripting => Messages from last translation** from the Menu Bar.



Importing and Converting Test Scripts on Rebuilt Environments With No Test Cases

The following CLICAST command can be invoked in instances where an environment having no test cases is rebuilt and the user wants to import and convert the test script:

```
clicast -e <env> test script convert <script>

This command imports the test script and converts it to accommodate source
code changes in the environment.
```

Delete Test Script Translation

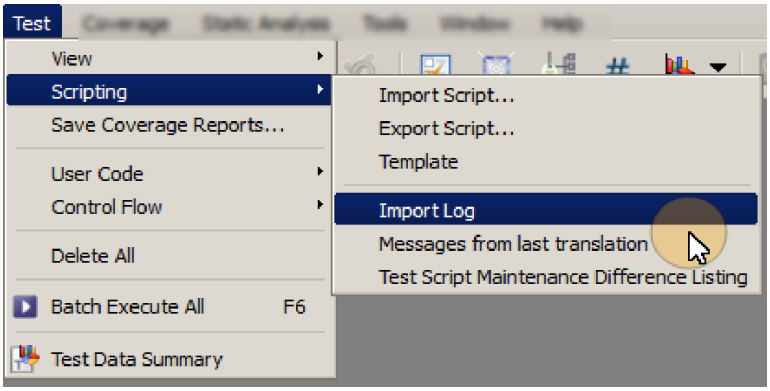
To delete a test script translation and revert to the original test script, perform the following steps:

1. Delete all of the test cases.
2. Select **Test => Scripting => Import Script** from the Menu Bar.
3. Navigate to the environment directory and select the file **`convertScript-original.tst`**.

Test Script Maintenance Reports

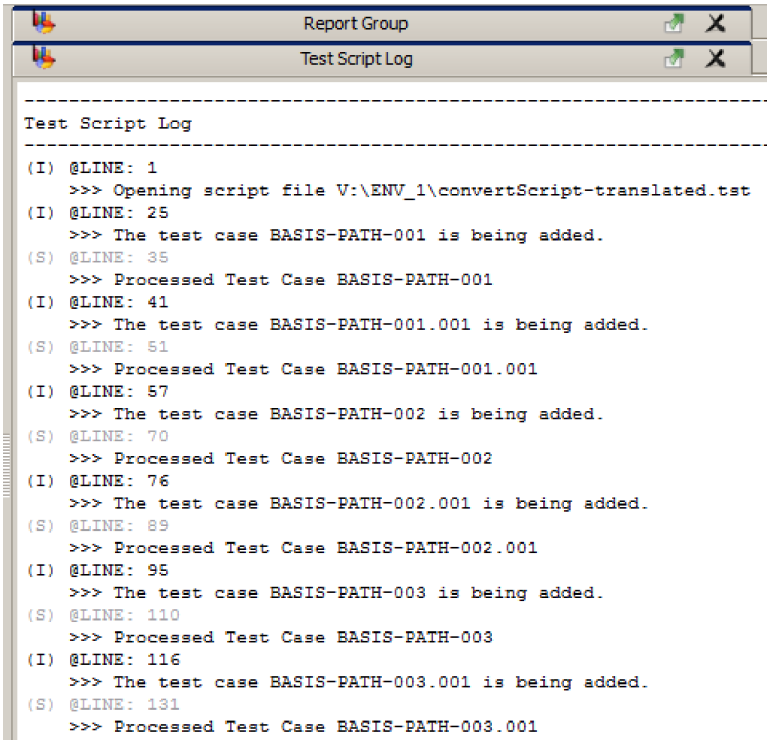
Data on the test script conversion is available from the Menu Bar by selecting **Test => Scripting** and selecting one of the report types:

- **Import Log**
- **Messages from last translation**
- **Test Script Maintenance Difference Listing**

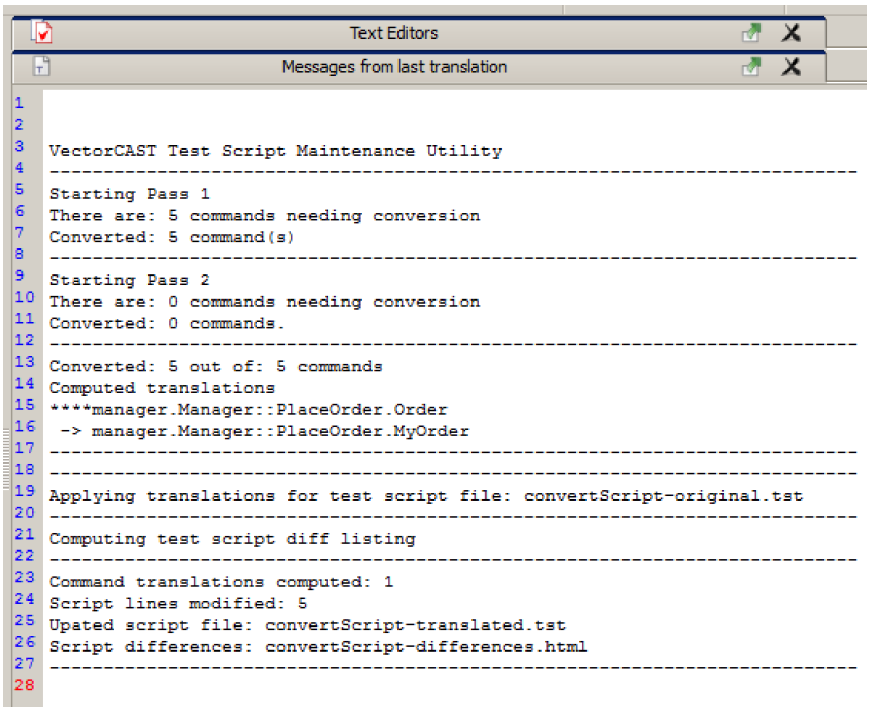


The reports are described below.

Selecting **Import Log** opens the Test Script Log. The Test Script Log reports on the processing of the test cases and any warnings or errors that occurred.

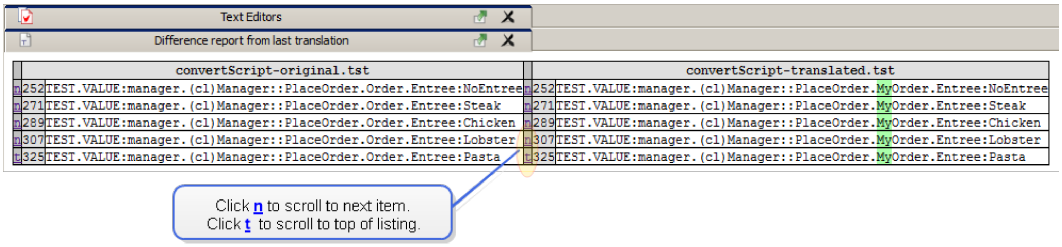


Selecting **Messages from last translation** opens the Messages From Last Translation report. This report provides status messages for the Test Script Maintenance Utility.



Selecting **Test Script Maintenance Difference Listing** opens the Test Script Maintenance

Difference Listing, and shows the changes between the original and translated scripts.



Test Script Conversion Processing

Test script conversion requires that a backup environment directory (.BAK) exists in order to begin processing. The Test Script Maintenance Utility first reads the parse data in the **`param.xml`** and **`types.xml`** files in the .BAK version of the environment, and compares it to the newly-rebuilt environment before the test script has been re-imported. It maps the functions, parameters and global objects from the old version to the new, translating the test script in the process. At the end of the environment rebuild, the translated test script is imported. Output from the Test Script Maintenance Utility can be found at **Test => Scripting => Messages** from last translation, or in the file **`convertScript-log.txt`**.

Generating Test Cases Using Automatic Test Case Generation (ATG)

VectorCAST supports creating test cases using the Automatic Test Case Generation (ATG) facility. ATG is a successor technology to the existing basis path technology in VectorCAST/C++. The goal of ATG is to automatically generate high-coverage test cases from the source code. ATG automatically creates test cases that cover a high percentage of the unit under test.

ATG is integrated as a "background" process into the VectorCAST GUI which allows tests to be generated while continuing to author tests manually. The test generation process can be stopped by choosing **Abort** from the ATG menu.

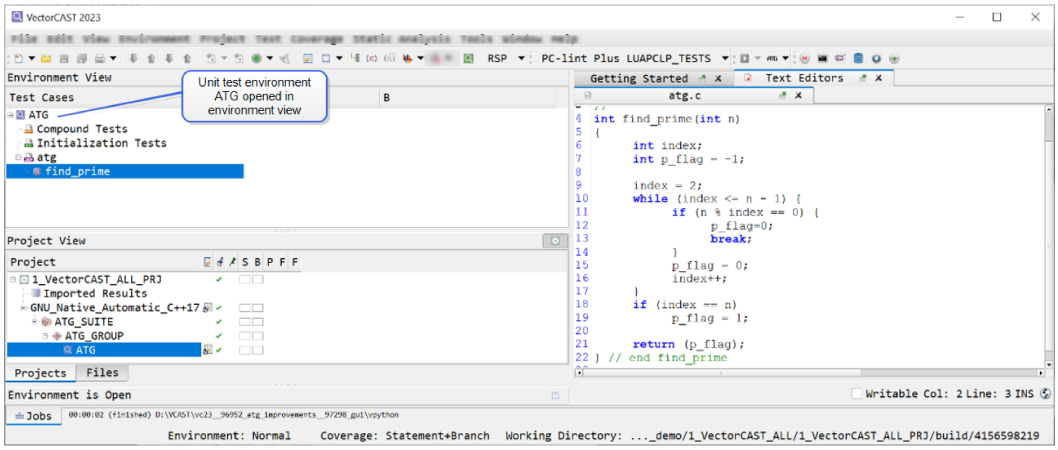


Note: The ATG tests are created with the prefix ‘ATG-TEST-‘ to distinguish them from other tests in the environment. Care should be taken to not use this prefix in the name of other tests in the environment.

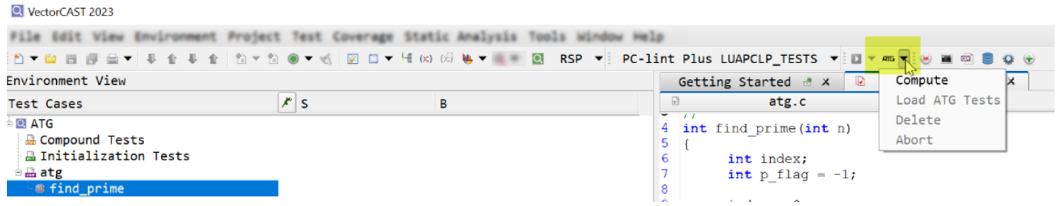
To Generate Test Cases Using ATG

To generate test cases using ATG, you start with an existing C/C++ Unit Test Environment.

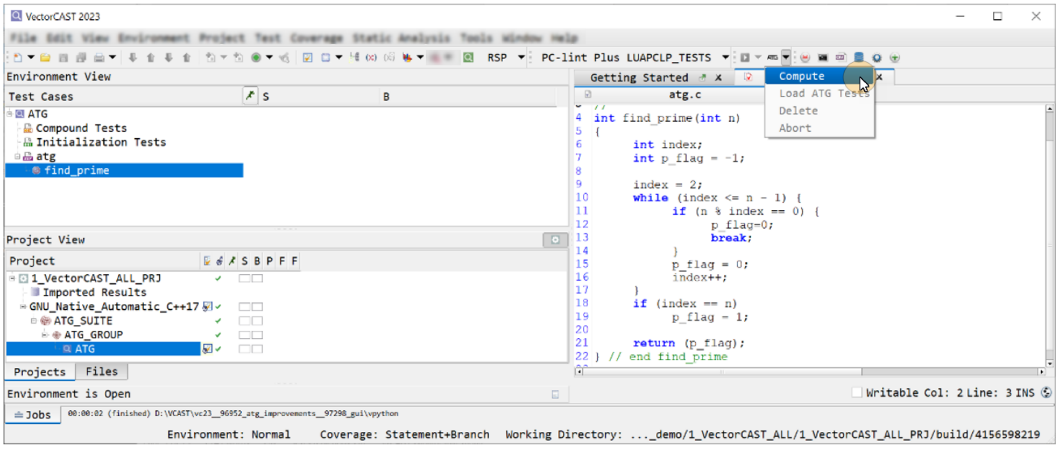
1. Open a C/C++ Unit Test Environment.



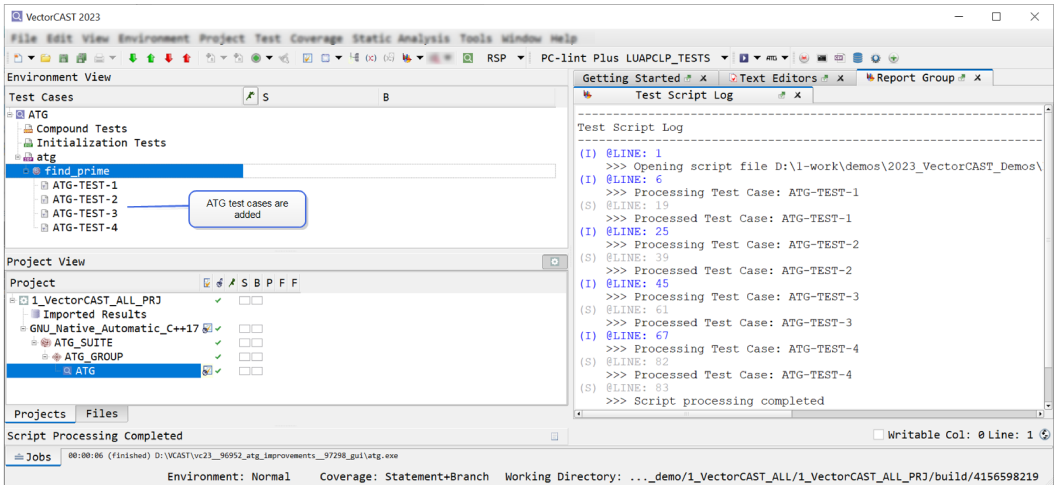
2. Select the **ATG** pulldown menu from the Tool Bar.



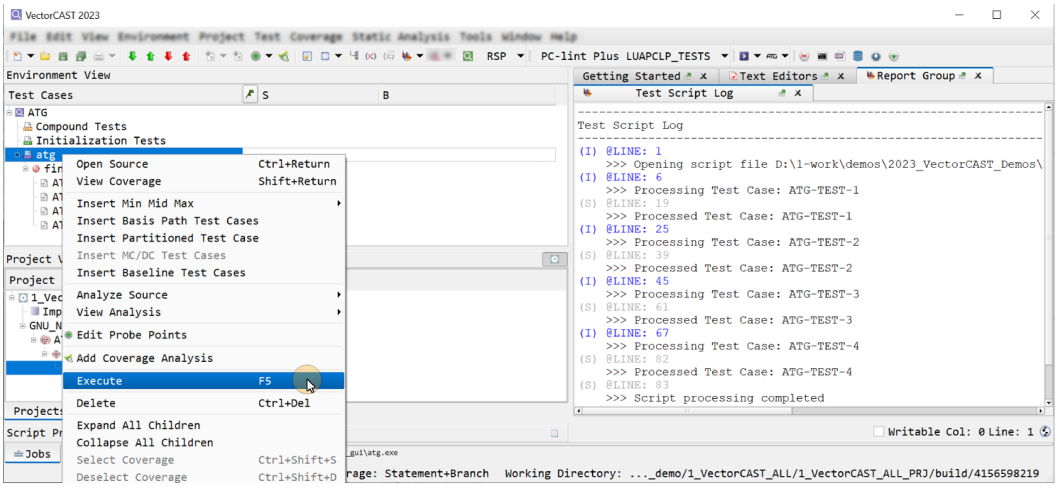
3. Select **Compute**.



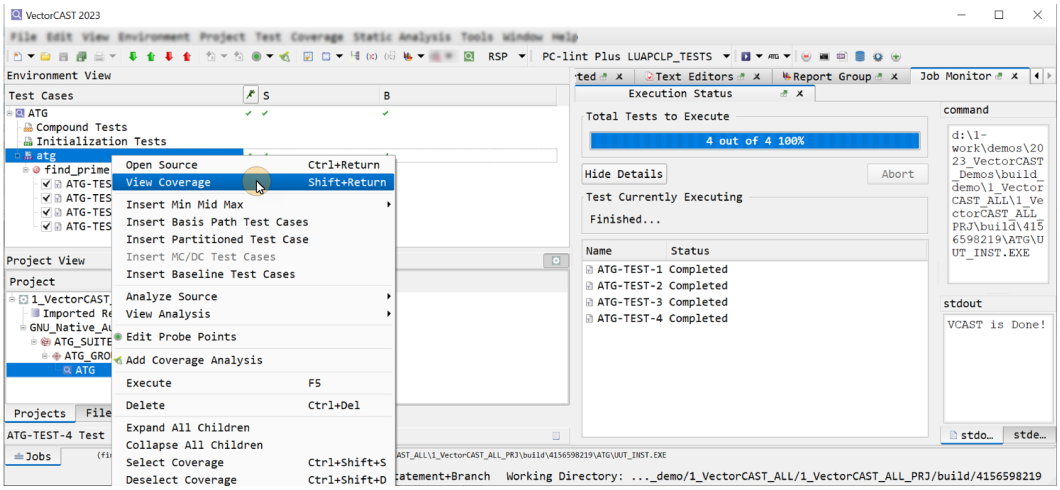
4. Test cases are added.



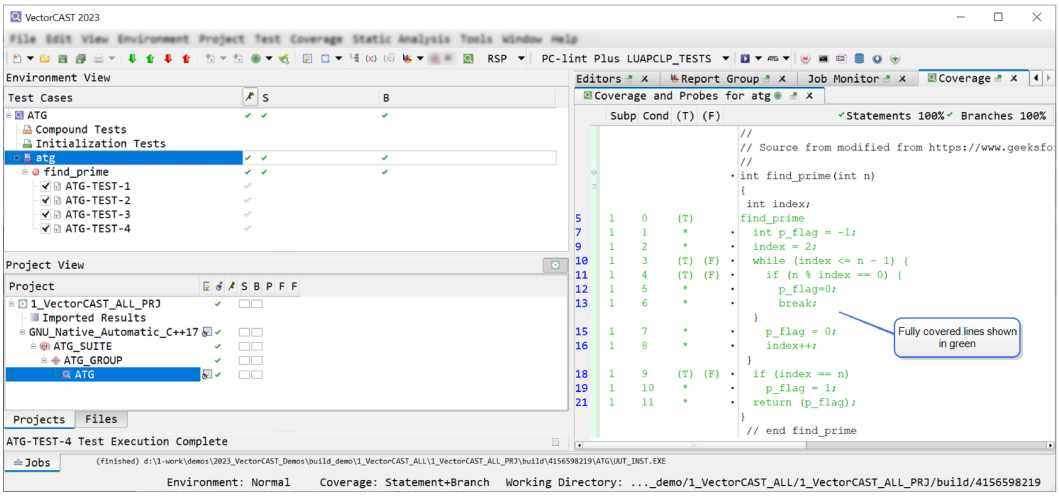
5. Run the test cases.



6. View the coverage results.

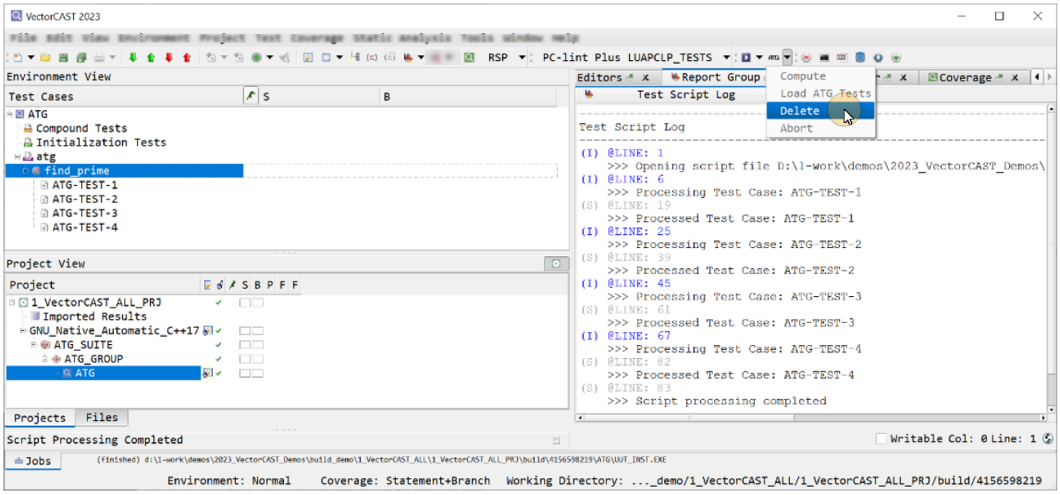


7. Coverage viewer shows all fully covered lines in green.

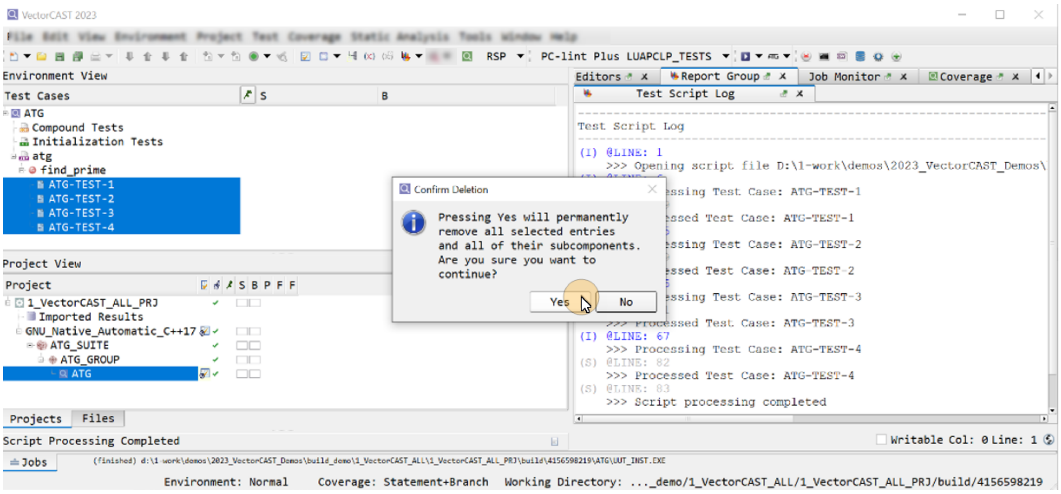


To Delete and Reload Test Cases Created by ATG

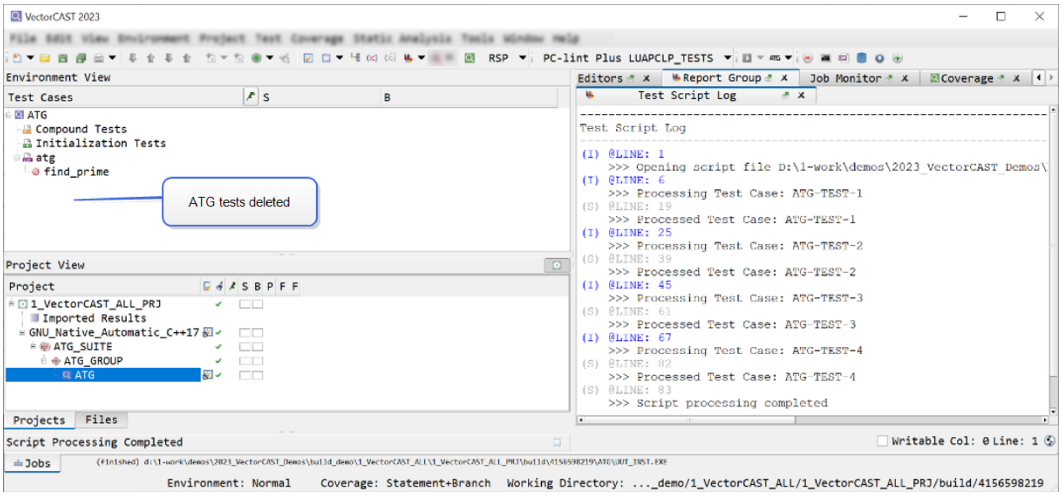
1. Select **Delete** from the **ATG** menu.



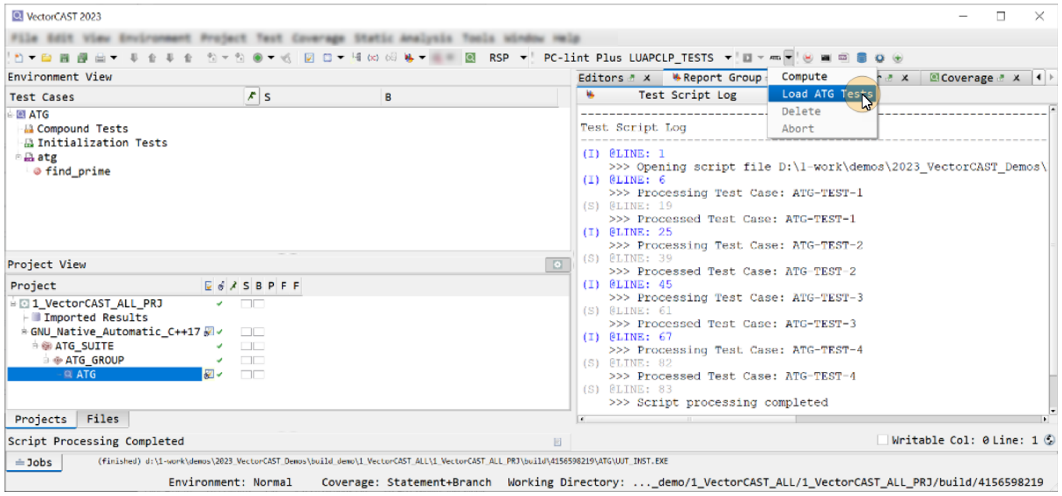
2. Confirm the deletion.



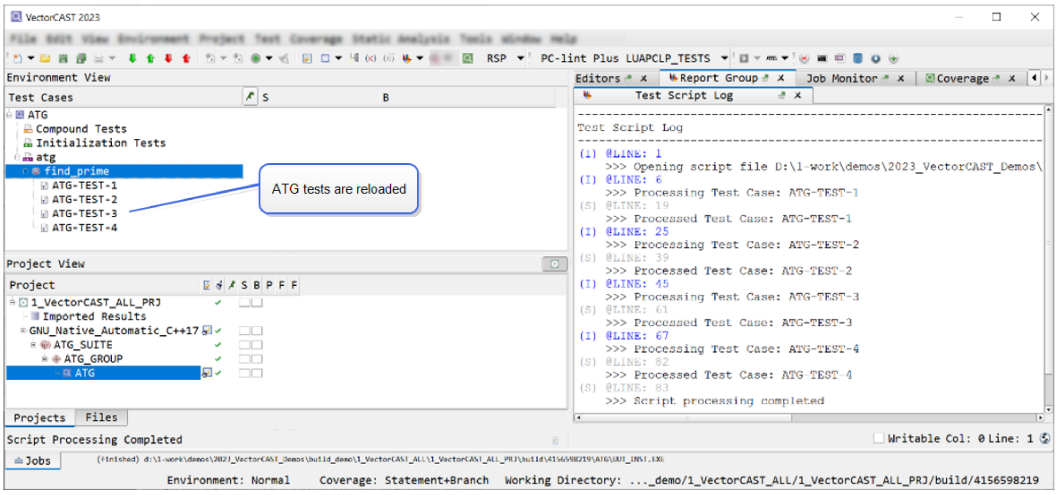
3.Test cases are deleted.



4. Reload the test cases by selecting **Load ATG Tests**.



5. Tests are reloaded.



ATG Baselining Functionality

VectorCAST ATG supports a baselining functionality, which allows ATG to precisely encapsulate the behavior of existing code by capturing expected values from test case execution. This is achieved by integrating ATG with the ability of VectorCAST to specify and record certain values during test runs (i.e., using TEST.ATTRIBUTE lines). These recorded values are then automatically transformed into expected values through test execution and recorded as TEST.EXPECTED lines within the generated test-cases.

Baselining Process

ATG's baselining process is as follows:

1. Backup and Remove Existing Tests: Before performing baselining on your environment, ATG backs-up and then removes any tests currently imported.

2. Generate and Import ATG Test Cases: New test-cases are then generated using ATG and imported into the environment.

3. Execute Test Cases: The newly generated test-cases are executed and have their values captured.

4. Convert Values into Expected Values: The captured values are transformed into expected values.

5. Re-execute Test Cases: To ensure consistency, the test-cases are executed again.

Repeat steps 3 to 5 if any values fail after re-execution (e.g., due to uninitialized test inputs, random values, "unstubbable" functions, etc.). ATG's baselining process will automatically remove any failing expected values during these iterations.

Example Use Case

To baseline an existing VectorCAST/C++ environment, use ATG from the command line:

```
atg --baselining <output_file>
```

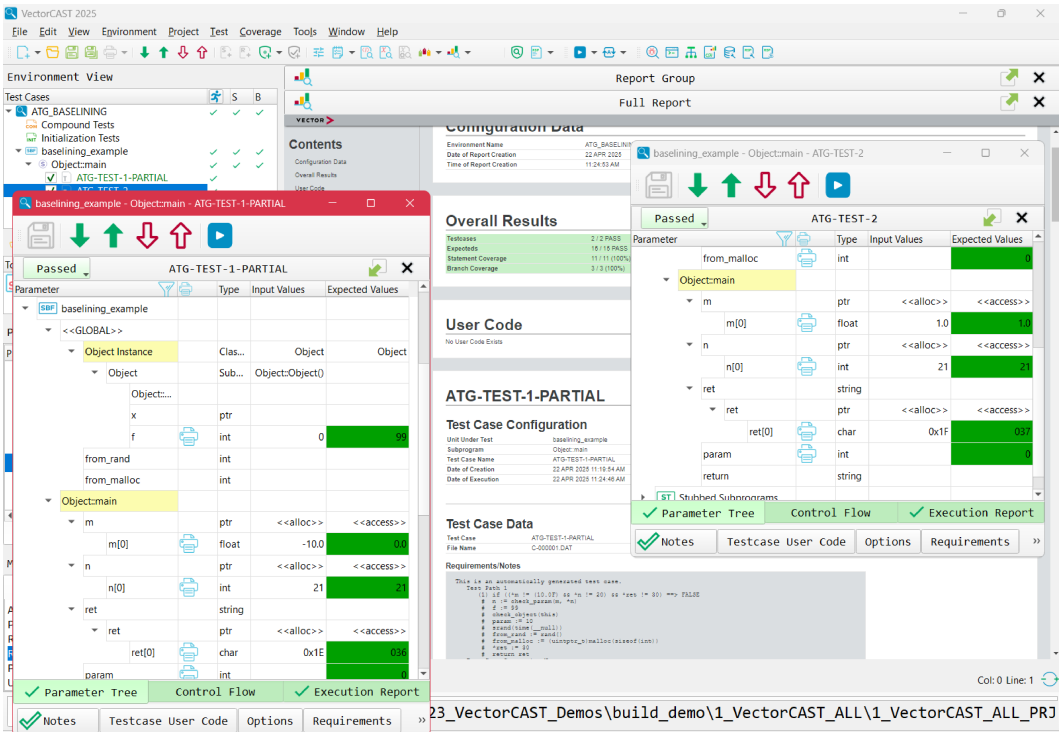
Replace **<output_file>** with the actual file name where the generated values and expected lines for the environment will be stored. If no file name is provided, the output defaults to **atg.tst**.

An example is shown below:

```
>atg --baselining
Archiving existing tests
Processing unit: baselining_example
Subprogram: Object::main(float*,int*,char*,int)char* 1.................
2..
Baselining iteration 1 (generating expected values)
       Deleting current tests
       Importing tests from the previous iteration
       Executing tests
       Setting actuals to expecteds
       Exporting tests for the next iteration
Baselining iteration 2 (verifying generated expected values)
       Deleting current tests
       Importing tests from the previous iteration
       Executing tests
       Exporting tests for the next iteration
       Total mismatched expecteds: 3
Baselining iteration 3 (test differences found; re-verifying)
       Deleting current tests
       Importing tests from the previous iteration
       Executing tests
       Exporting tests for the next iteration
       Tests are the same between iterations; baselining complete
Exporting baselining tests
Restoring archived tests
Baselining completed

>clicast -lc -e ATG_BASELINING TEST Script run atg.tst
```

The figure below shows the tests and test report for an example test environment with tests generated using the **atg --baselining** command.



Advanced Features

Several other command line options are available for enhancing the baseline functionality:

- **`--baseline-on-tst <path to test script>`**: Use this option to baseline using a pre-existing file, without generating additional value lines.
- **`--baselining-allow-atg-tests-in-env`**: This checks if any ATG tests are loaded in the current environment before proceeding with baselining.
- **`--no-baselining-delete-tests`**: Select this option to retain currently imported tests when generating expected values.

Generating Test Cases Using Coded Tests

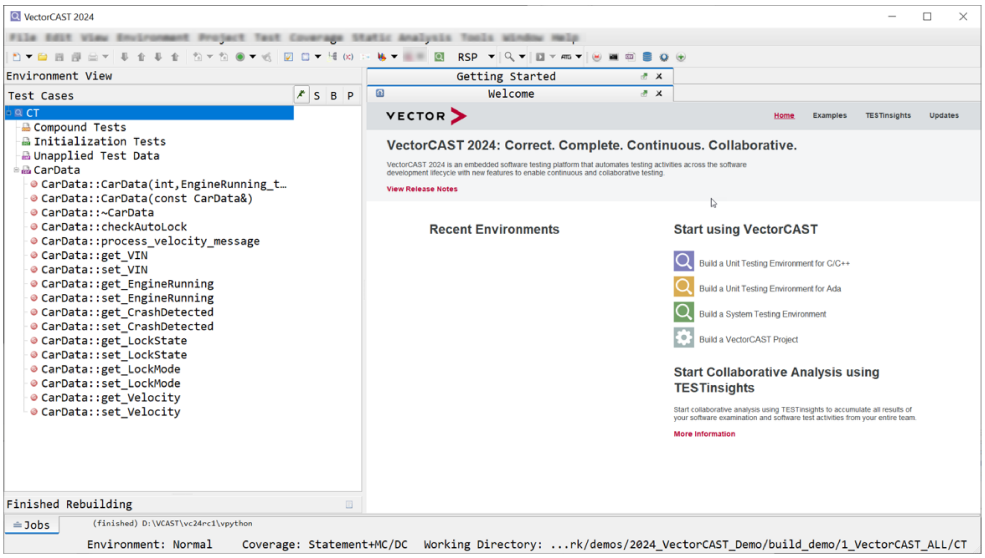
VectorCAST supports creating test cases using the Coded Test facility. Coded Tests allow for the use of C++ code to create test cases.

Note: To use Coded Tests, the compiler you use must support at least C++11.

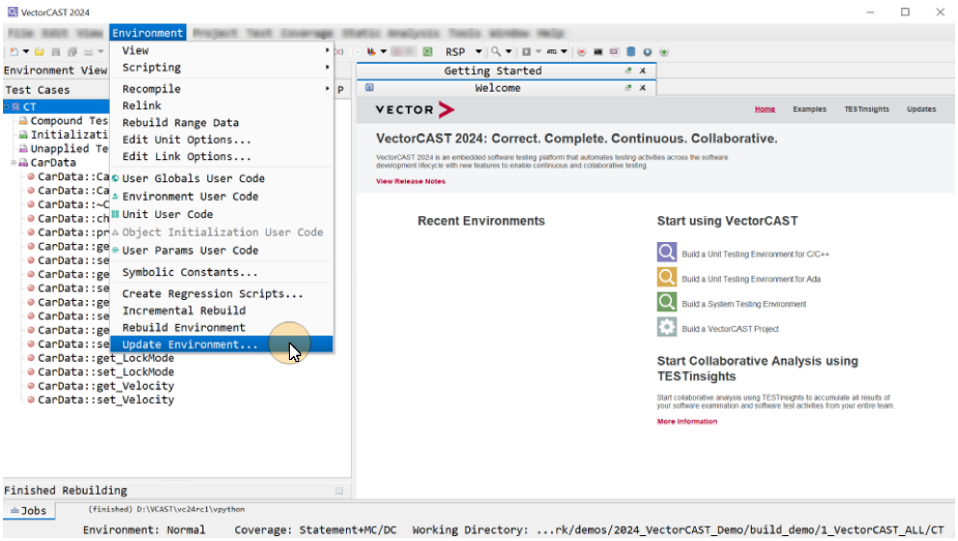
To Generate Test Cases Using Coded Tests

To generate test cases using Coded Tests, you must start with an existing C/C++ Unit Test Environment.

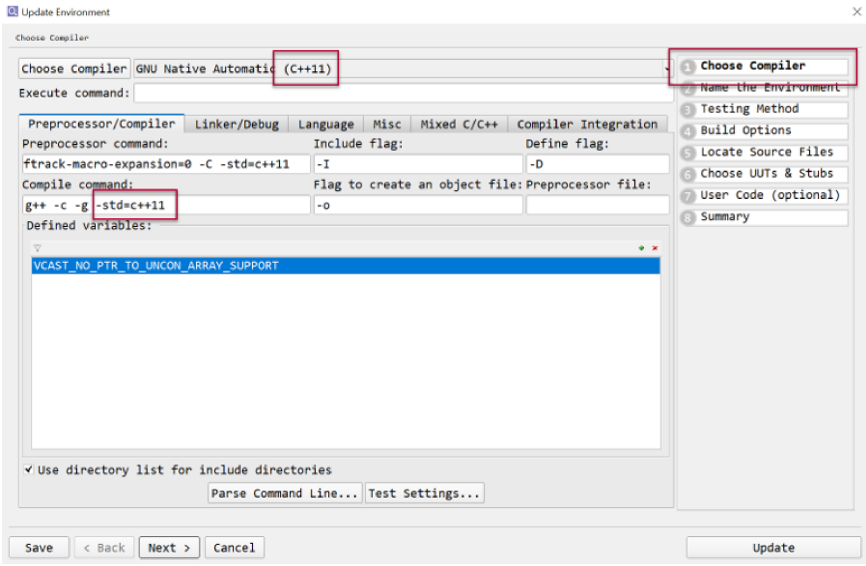
1. Open a C/C++ Unit Test Environment.



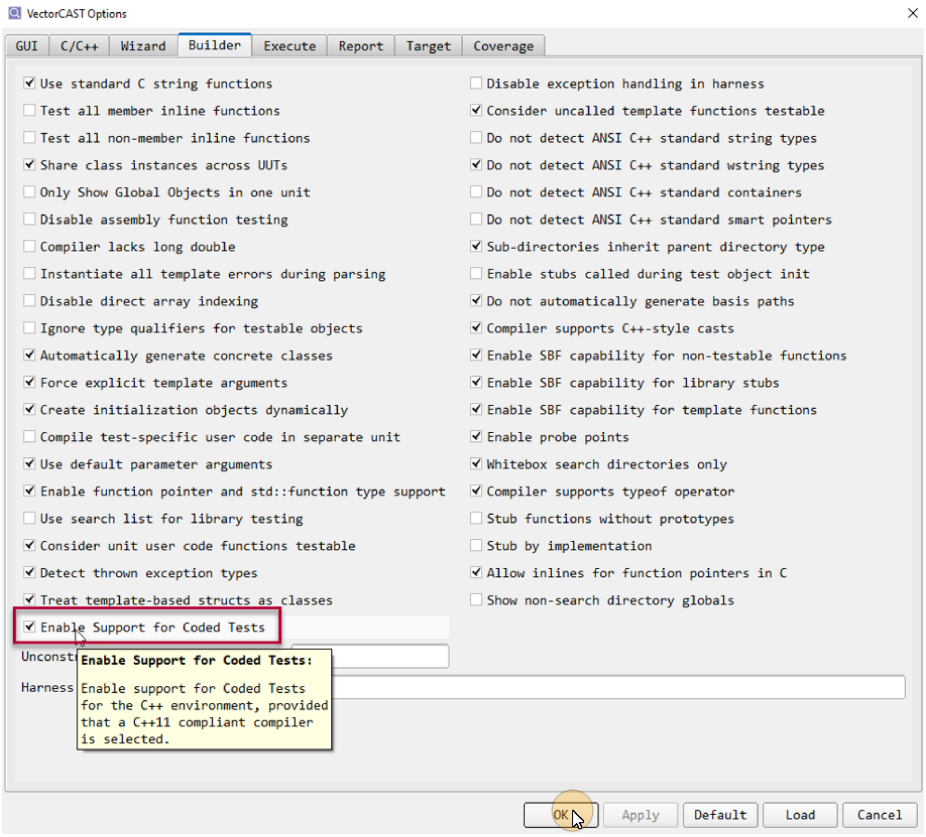
2. Open the Update Environment Wizard by selecting **Environment > Update Environment...** from the Menu Bar.



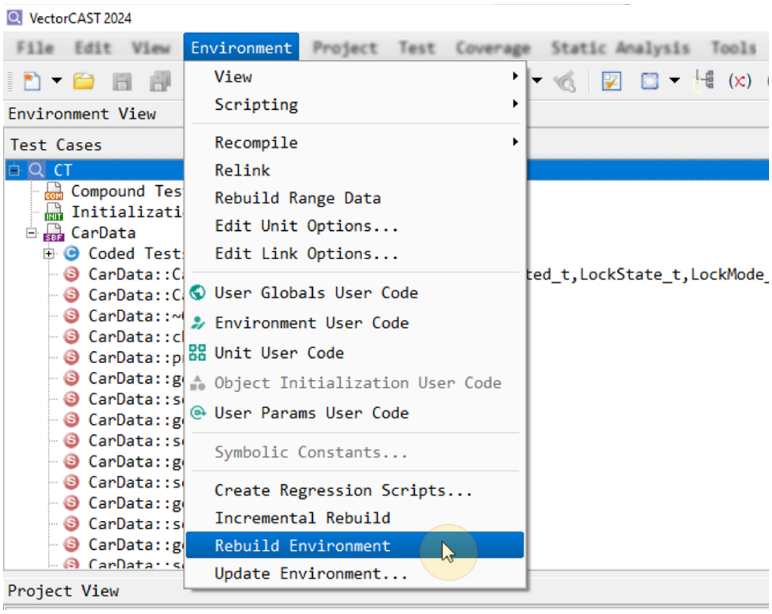
3. On the **Choose Compiler** tab, ensure that the compiler supports at least C++11.



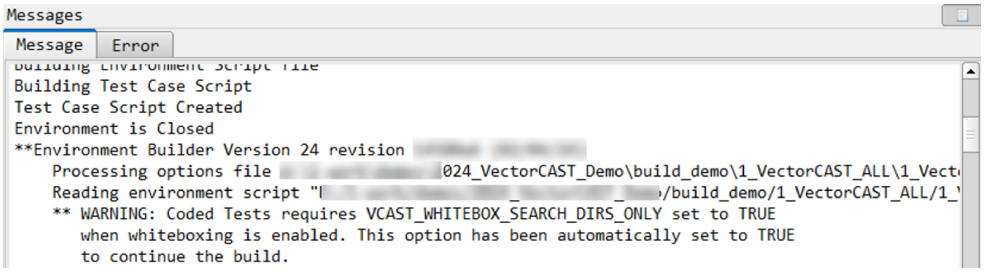
4. Close the Update Environment Wizard. Open the Options panel by selecting **Tools > Options...** from the Menu Bar. Select the **Builder** Tab and then select **Enable Support for Coded Tests**. Select **OK**.



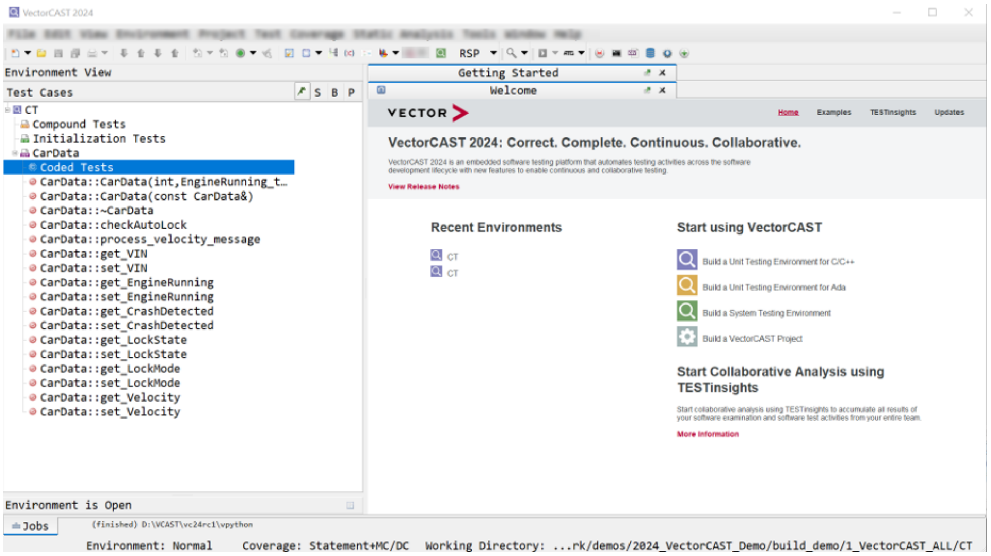
5. Rebuild the environment by selecting **Environment > Rebuild Environment**.



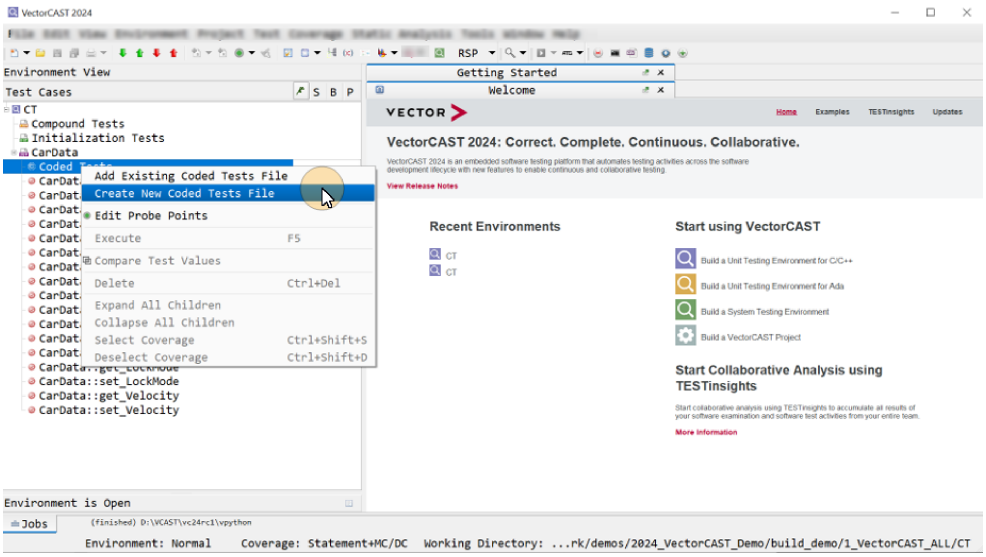
6. VectorCAST will rebuild the environment. You may see a message when rebuilding the environment if your environment is set to enable Whitebox but your configuration is not set for Whitebox Search Directories Only. The message indicates that Whitebox Search Directories Only has been set automatically.

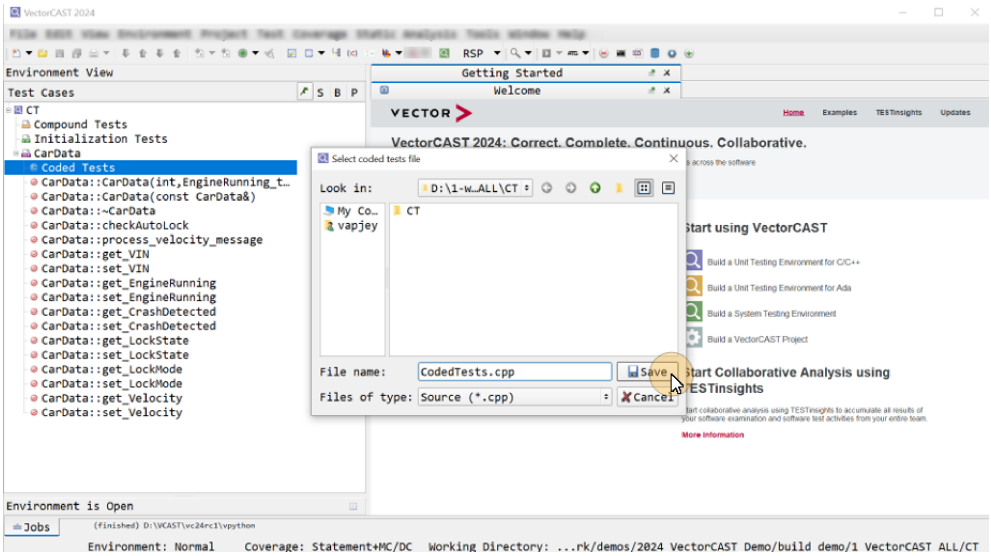


7. There will be a new node labeled **Coded Tests**.

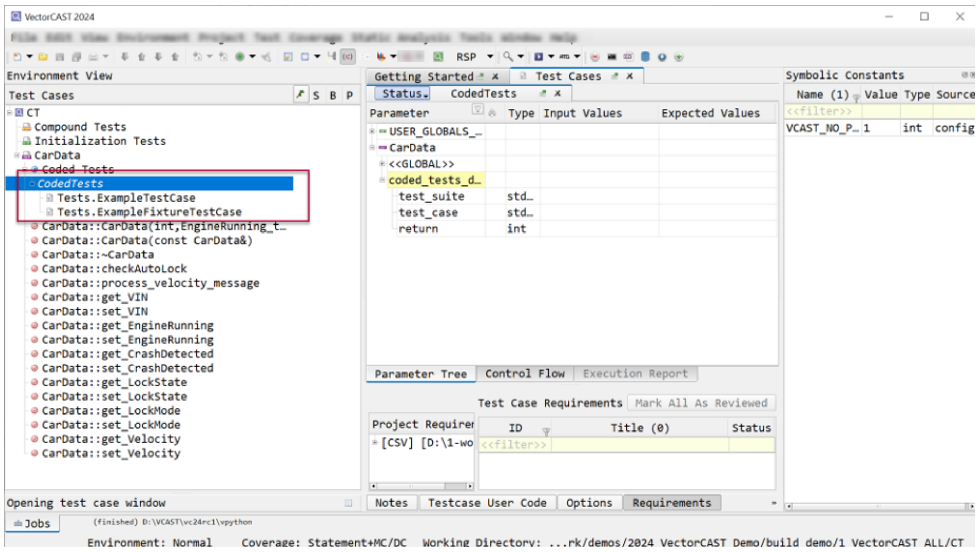


8. Right click on the **Coded Tests** node and select **Create New Coded Tests File** and then select the name and location for the file. Select **Save**.

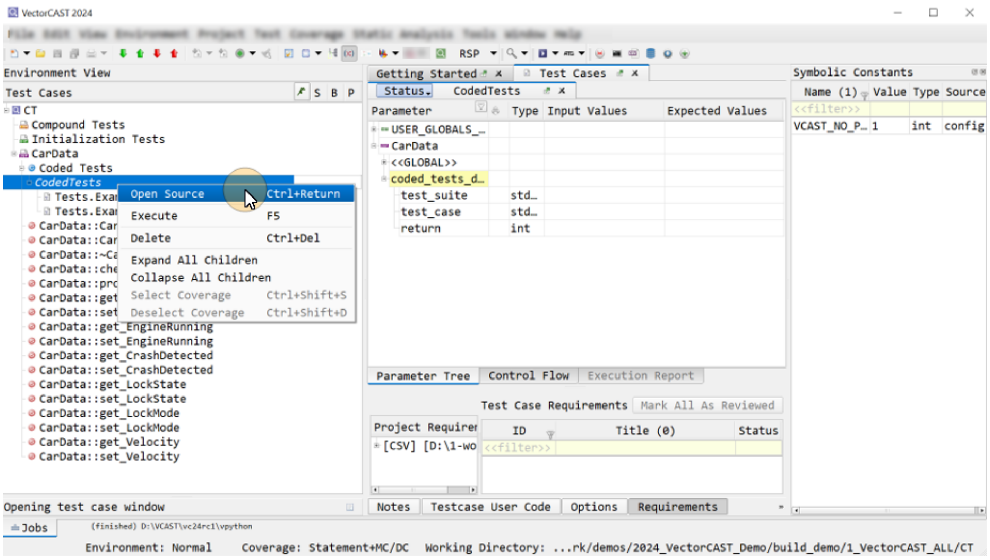




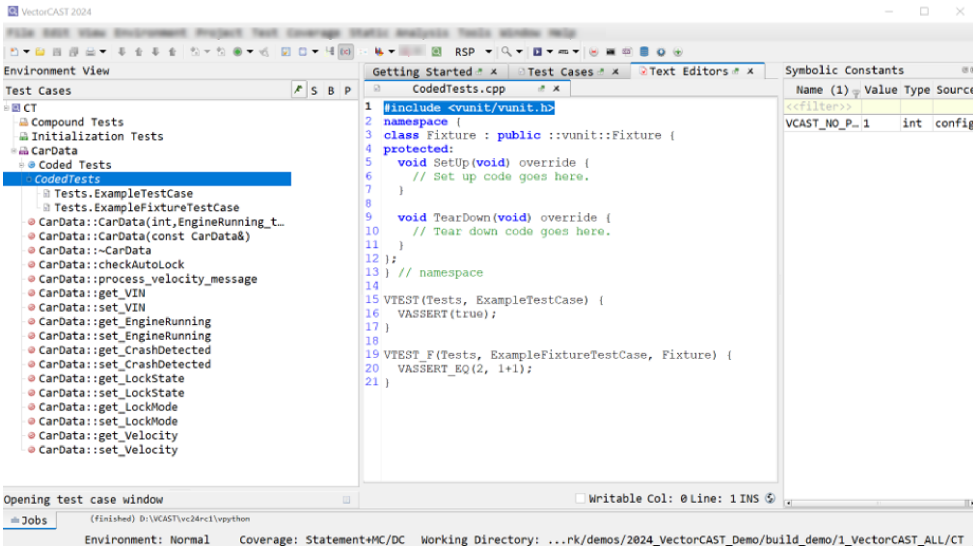
9. A new node will appear, named after the filename given in the previous step, along with two example coded tests.



10. Right click on the node and select **Open Source**.



11. The example coded tests file will be shown in the text editor.



Finding Help for the Coded Tests Syntax

The syntax for defining the coded tests can be found in the VectorCAST distribution under the directory '**`vunit`**'.

Generate Coded Tests with ATG

An advanced use case for ATG uses the command line version of ATG to generate Coded Tests. When ATG is used to generate Coded Tests, the unit test environment must already be configured for Coded Tests. See the section "Generating Test Cases Using Coded Tests" on page 344 for more information on Coded Tests.

The basic command to request ATG to generate Coded Tests is:

```
atg -e <env_name> --vc-coded-tests <output_file>

Use the ATG --vc-coded-tests command parameter to generate a test script.
If the <output_file> is specified it will be used as the name of the test
script file. If no filename is provided, the output defaults to <env_
name>.atg.tst.
```

Basic Use Case

To generate coded tests for an existing VectorCAST/C++ environment that supports Coded Tests, use the ATG command from the command line with the **`--vc-coded-tests`** option.

An example of this use case is shown below. In this environment, there are two units under test, **`CarDataA`** and **`CarDataB`**. A (**`.tst`**) file is generated for each.

Run the ATG command

```
>atg -e ATG_CTA --vc-coded-tests
 [...]
 Processing unit: CarDataA
 Subprogram: CarDataA::CarDataA(uint32_t) 1/1..
 Subprogram: CarDataA::~CarDataA() 1/1.
 Subprogram: CarDataA::functionA(uint32_t)uint32_t 1/1.....
 Processing unit: CarDataB
 Subprogram: CarDataB::CarDataB(uint32_t) 1/1..
 Subprogram: CarDataB::~CarDataB() 1/1.
 Subprogram: CarDataB::functionB(uint32_t)uint32_t 1/1..
 ATG tests (2 in total) have been saved to file:
 ATG_CTA.atg.tst
 To load these tests into the environment use the command:
 clicast -e ATG_CTA test script run ATG_CTA.atg.tst
```

Generated Output Files

ATG Coded Tests mode generates two files:

1. Test Script File (**`.tst`**): A minimal VectorCAST test script that references the coded tests file
2. Coded Tests File (**`.ct.cpp`**): The C++ source file containing the actual test implementations

For the example above, ATG generates:

> `ATG_CTA.atg.tst`

> `ATG_CTA_CarDataA.ct.cpp`

> `ATG_CTA_CarDataB.ct.cpp`

The naming convention for the coded tests file is: **`<env_name>_<unit_name>.ct.cpp`**

Example .tst file content:

```
-- Script Features
TEST.SCRIPT_FEATURE:C_DIRECT_ARRAY_INDEXING
TEST.SCRIPT_FEATURE:CPP_CLASS_OBJECT_REVISION
TEST.SCRIPT_FEATURE:MULTIPLE_UUT_SUPPORT
TEST.SCRIPT_FEATURE:REMOVED_CL_PREFIX
TEST.SCRIPT_FEATURE:MIXED_CASE_NAMES
TEST.SCRIPT_FEATURE:STATIC_HEADER_FUNCS_IN_UUTS
TEST.SCRIPT_FEATURE:VCAST_MAIN_NOT_RENAMED
TEST.SCRIPT_FEATURE:STANDARD_SPACING_R2
TEST.SCRIPT_FEATURE:OVERLOADED_CONST_SUPPORT
TEST.SCRIPT_FEATURE:UNDERSCORE_NULLPTR
TEST.SCRIPT_FEATURE:FULL_PARAMETER_TYPES
TEST.SCRIPT_FEATURE:STRUCT_DTOR_ADDS_POINTER
TEST.SCRIPT_FEATURE:STRUCT_FIELD_CTOR_ADDS_POINTER
TEST.SCRIPT_FEATURE:IGNORE_NAME_VALUE_ERRORS
--
-- ATG tests generated
-------------------------------
TEST.UNIT:CarDataA
TEST.SUBPROGRAM:coded_tests_driver
TEST.NEW
TEST.NAME:ATG_CTA.atg
TEST.CODED_TESTS_FILE:ATG_CTA_CarDataA.ct.cpp
TEST.END
-------------------------------
-------------------------------
TEST.UNIT:CarDataB
TEST.SUBPROGRAM:coded_tests_driver
TEST.NEW
TEST.NAME:ATG_CTA.atg
TEST.CODED_TESTS_FILE:ATG_CTA_CarDataB.ct.cpp
TEST.END
-------------------------------
```

Advanced Features

The following command line option is available for customizing the filenames for the coded tests output:

--coded-tests-filename <pattern>

Specify a custom filename pattern for the coded tests file. The pattern can include placeholders:

- **{env}**: Environment name
- **{unit}**: Unit name (required when running in an environment with multiple units)
- **{ext}**: File extension

Default pattern: **{env}_{unit}.{ext}**

Generated Code Structure

ATG Coded Tests mode generates a Coded Tests source file for each unit under test. This section gives the details on how the generated source file .ct.cpp is structured.

Test File Format

The generated .ct.cpp file includes:

1. Standard headers (#include <cstdlib>)
2. Mock functions section (in an anonymous namespace)
3. Test cases using the VTEST macro

Each test case includes detailed comments showing:

- Subprogram name and path number
- The expected execution path through the code with decision outcomes

Example test header:

```
1  /**************************************************************************
   ******************
2   *                                              TEST
   START                                           *
3   **************************************************************************
   ******************/
4  // --------------------------------------------------------------------------
   -------------------
5  //
6  //  SUBPROGRAM: CarDataA::CarDataA(uint32_t)
7  //  PATH NUMBER: 0001
8  //
9  // --------------------------------------------------------------------------
   -------------------
10 //
11 // PATH:
12 //
13 //        #  Velocity := Velocity
14 //
15 // --------------------------------------------------------------------------
   -------------------
16 // TEST BODY
17 VTEST(ATG, CarDataA__CarDataA_uint32_t__0001)
18 {
19 unsigned Velocity;
20 Velocity = 1;
21 CarDataA test_instance(Velocity);
22 }
23
```

For functions that need to be stubbed, ATG generates a mock function infrastructure in an anonymous namespace. In the generated code for a mock function, the label "ROUTINE" is referencing a VectorCAST subprogram.

Example mock function:

```
namespace {

// ROUTINE: DataBase::GetTableRecord(unsigned,TableDataType*)

int num_ret__ZN8DataBase14GetTableRecordEjP13TableDataType;
using FPTR__ZN8DataBase14GetTableRecordEjP13TableDataType_sbr = void (*)
(TableDataType*);
FPTR__ZN8DataBase14GetTableRecordEjP13TableDataType_sbr* sbr_fptr__
ZN8DataBase14GetTableRecordEjP13TableDataType;

void _ZN8DataBase14GetTableRecordEjP13TableDataType_atg_stub(
    ::vunit::CallCtx<DataBase> vunit_ctx,
    unsigned Table,
    struct TableDataType *Data)
{
    static int call_num = -1;
    if(call_num < num_ret__ZN8DataBase14GetTableRecordEjP13TableDataType - 1)
    call_num++;
    sbr_fptr__ZN8DataBase14GetTableRecordEjP13TableDataType[call_num](Data);
}

}  // namespace
```

For each stub, value assignment functions are generated if needed to set specific stub-by-reference output values:

```
void values_assign_Manager__PlaceOrder_unsigned_unsigned_OrderType__0001__
ZN8DataBase14GetTableRecordEjP13TableDataType_0(TableDataType* Data)
{
    if(Data)
    {
        Data[0].NumberInParty = -3;
        Data[0].CheckTotal = -15;
    }
}
```

Test Body Structure

Each test uses the VTEST macro with the ATG test group and has a descriptive name:

```
VTEST(ATG, Manager__AddIncludedDessert_OrderType___0002)
{
    OrderType* Order;
    Manager* _ATG_THIS_INSTANCE;
    Order = (OrderType*)malloc(sizeof(OrderType)*1);
    _ATG_THIS_INSTANCE = new Manager;
    Order[0].Salad = Green;
    Order[0].Entree = Lobster;
    Order[0].Beverage = Wine;
    _ATG_THIS_INSTANCE->AddIncludedDessert(Order);
    free(Order);
```

```
12 | }
```

For tests requiring mocked functions, a MockSession is created:

```
1  | VTEST(ATG, Manager__PlaceOrder_unsigned_unsigned_OrderType__0001)
2  | {
3  |     auto vmock_session = ::vunit::MockSession();
4  |
5  |     // Configure mock for DeleteTableRecord
6  |     num_ret__ZN8DataBase17DeleteTableRecordEP13TableDataType = 1;
7  |     sbr_fptr__ZN8DataBase17DeleteTableRecordEP13TableDataType = new FPTR__
   | ZN8DataBase17DeleteTableRecordEP13TableDataType_sbr[1];
8  |     sbr_fptr__ZN8DataBase17DeleteTableRecordEP13TableDataType[0] = dummy_sbr__
   | ZN8DataBase17DeleteTableRecordEP13TableDataType;
9  |
10 |     // ... mock registration using vmock_session.mock<>().assign() ...
11 |
12 |     // Test body
13 |     unsigned Table = 0;
14 |     unsigned Seat = 3;
15 |     OrderType* _ATG_CLASS_PTR_Order = (OrderType*)malloc(sizeof(OrderType)*1);
16 |     Manager* _ATG_THIS_INSTANCE = new Manager;
17 |     _ATG_CLASS_PTR_Order[0].Entree = Steak;
18 |     _ATG_THIS_INSTANCE->PlaceOrder(Table, Seat, *_ATG_CLASS_PTR_Order);
19 |
20 |     // Cleanup
21 |     free(_ATG_CLASS_PTR_Order);
22 |     delete[] sbr_fptr__ZN8DataBase17DeleteTableRecordEP13TableDataType;
23 |     // ... additional cleanup ...
24 | }
```

To generate coded tests for an existing VectorCAST/C++ environment that supports Coded Tests, use the ATG command from the command line with the --vc-coded-tests option. Then run the generated test script with the **`clicast`** command.

Enter:

```
atg -e <env> --vc-coded-tests
clicast -e <env> test script run <env>.atg.tst
```

Generating Test Cases Using CSV- or Tab-Delimited Files

VectorCAST supports importing test data from CSV or tab-delimited data files. Test cases are imported by first creating an association between the columns in the CSV file with scalar data item in the test case parameter tree. After creating this association, the CSV data file can be imported, and each row in the CSV file will become a separate test case in VectorCAST.

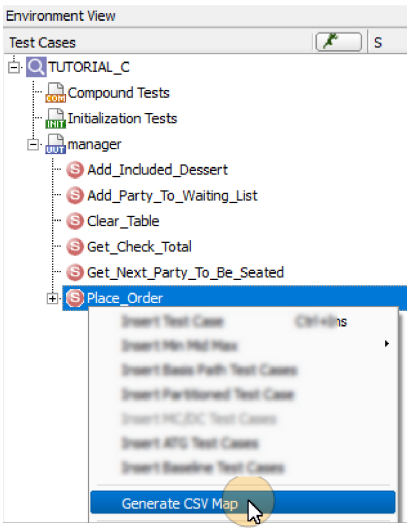
If you do not already have test data in this format, VectorCAST can be used to create the mapping, and then dump a "template". Once created, the template is filled with data, one row for each set of test case data using an external tool. The completed CSV file is then imported into VectorCAST to create the test cases.

The first step in either importing an existing file or in creating a template file is to create a *map test* case in VectorCAST. This special test case maps the columns of the file to any VectorCAST subprogram test case data item. Procedures for both creating a template file and for using a pre-existing file are discussed below. Note that once created, a map test case is imported and exported the same as any other test case.

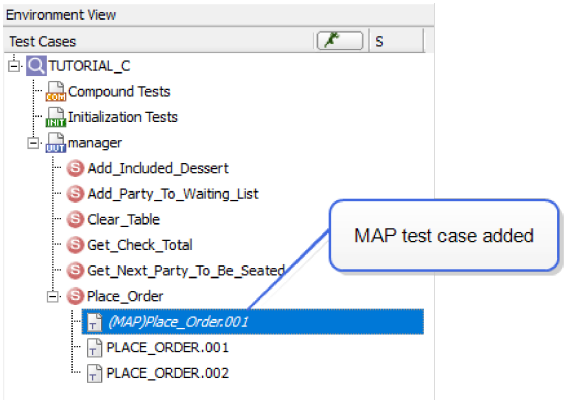
To Create a Template for a CSV- or Tab-Delimited File

VectorCAST can help you generate a CSV or tab-delimited template file that contains columns for test names and test case data items. After the template file is created, you use an external tool to enter test case data. To create a template file for a new `.csv` or `.tab` file, follow these steps:

1. Right-click a subprogram in the Test Case tree and select **Generate CSV Map**.

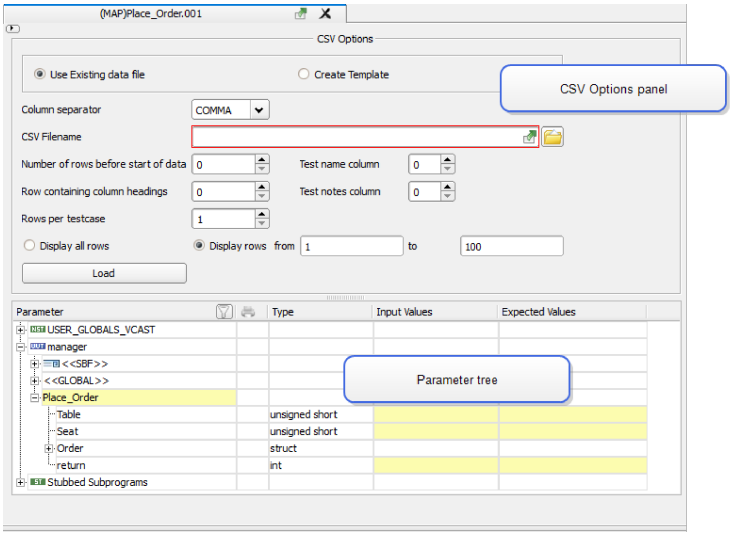


2. A (MAP) test case is placed in the Test Case tree.

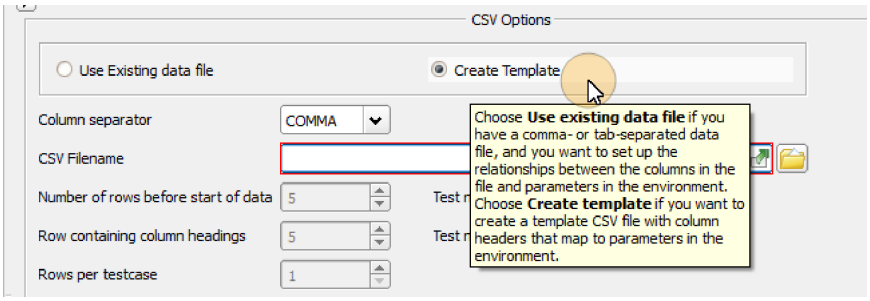


The CSV Options panel opens and below the CSV Options panel a Parameter tree opens. This tree looks similar to the Test Case Parameter tree. When creating a template file, the Parameter tree is used to select test case data items that will become columns in the template file. It is also

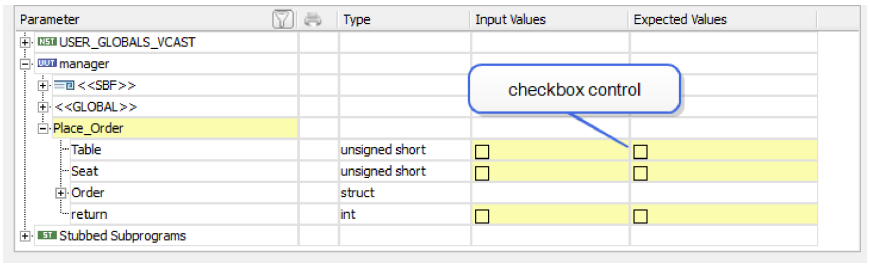
used to set the order of the columns within the template.



3. Select the **Create Template** radio button in the CSV Options panel.

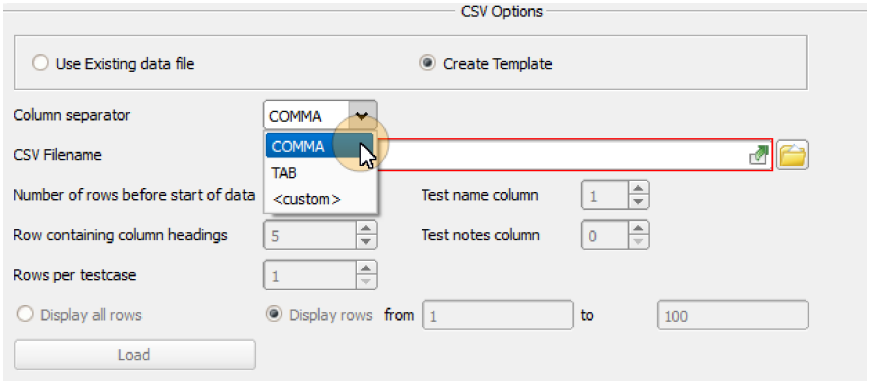


When this option is selected, each input and expected value data item in the Parameter Tree contains an associated *checkbox control*. The check boxes are used to select the data items that will become columns in the template file. Notice that check boxes appear not only for the parameters of the subprogram, but also for global data items and for SBF data items contained in the tree. In addition, if a data item is a pointer type, you can optionally use the Parameter Tree to create an instance of the pointer data item type. You can then select instance members of that type to be included as columns in the template file.

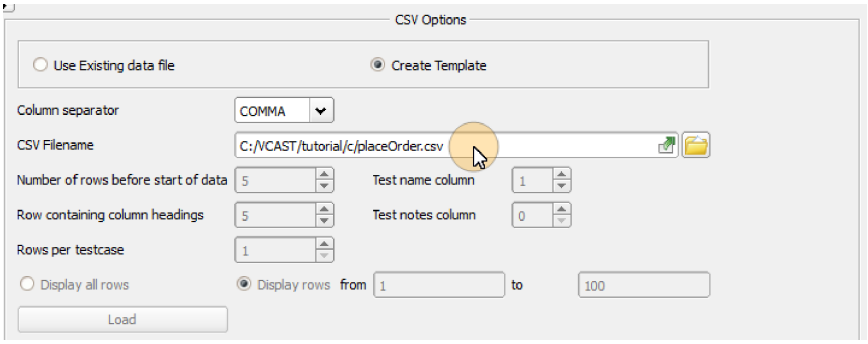


In the CSV Options panel, select a column separator by selecting either **COMMA**, **TAB** or **<custom>** from the drop-down menu for the delimiter you wish to use. Select **COMMA** if your

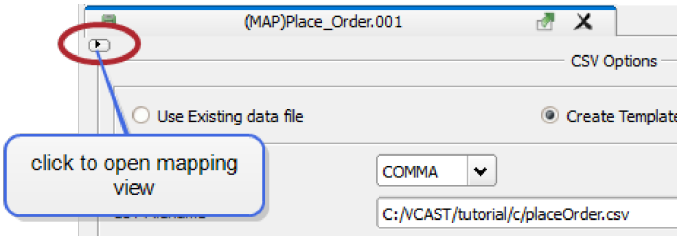
data file or template uses the comma character to separate the field in a row. Select **TAB** if your data file or templates uses the tab character to separate the fields in a row. Select **<custom>** and enter any other character that your data file uses to separate fields in a row.



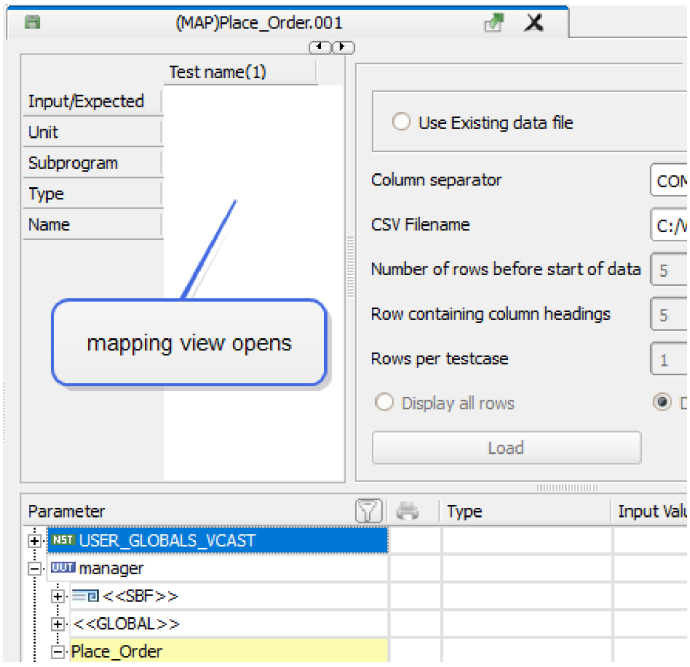
Enter a name for the template file. You may wish to enter an extension for the file name, i.e., **`.csv`**. By doing so, an external program, Microsoft Excel for example, will recognize the file type and open it appropriately so that its contents can easily be edited. You can optionally navigate to a specific directory by using the file browse button.



4. Next, click the arrow button, found in the upper left of the MDI window.

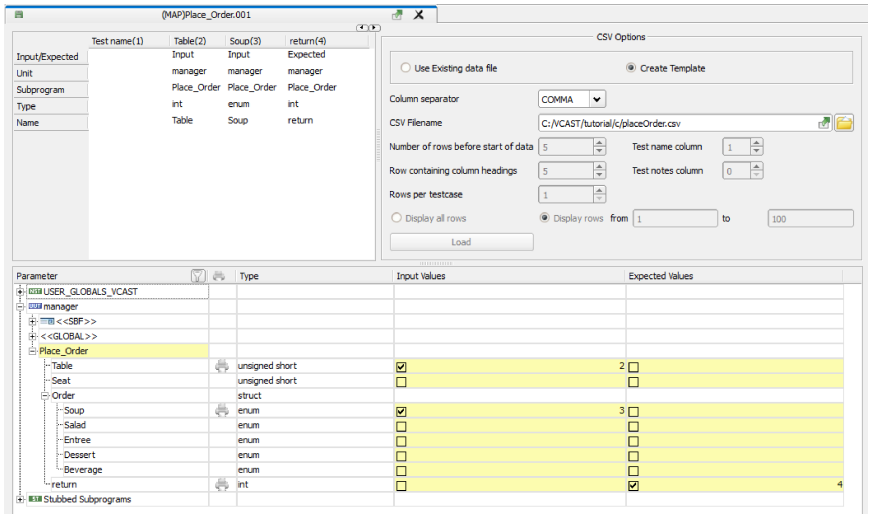


A blank Mapping View appears above the Parameter Tree.

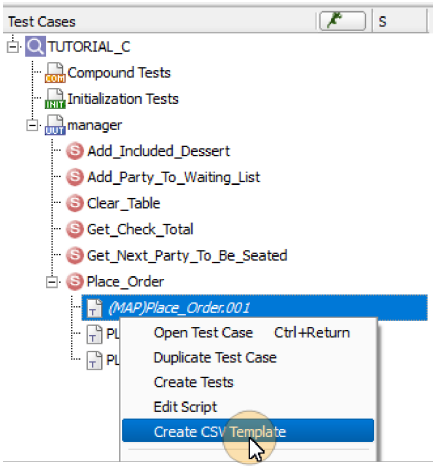


5. Next, select the test case data items to be included in the template file by clicking the check boxes in the Parameter tree. As each data item is selected, a header entry is added to the Mapping View. The header information consists of five rows of information for each data item and will be included in the template file when it is generated. For each column, the template header information consists of:
   - **Input** or **Expected**
   - **Unit** name
   - **Subprogram** name
   - data item **Type** (int, float, enum, etc.)
   - data item's **Name**
6. As input or expected values are selected for inclusion, a number appears in the associated Parameter tree cell.
7. The same number also appears next to the column header name in the Mapping View. These numbers indicate the column number that the test data item is mapped to. The column order can be easily changed using either of the following methods:
   - The first method is to change the value in the Test Tree Editor. Click the desired cell in the editor and change the column number to the desired number and press **Enter**. The new column order will be immediately reflected in both the Mapping View and the Parameter Tree. All columns are renumbered appropriately based on the change.
   - The second method is to drag and drop columns within the Mapping View column title row. Holding the **CTRL** key, select and drag a column title in the column title row to the desired column position. The column numbers are automatically updated in

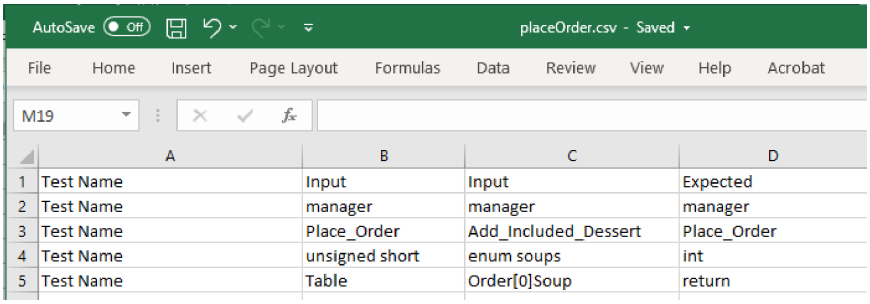
both the Mapping View and the Parameter Tree using this method as well.



8. When you are finished selecting columns and ordering them, save the map by clicking the **Save** button  on the Toolbar, or selecting **File => Save**. Note that this saves the mapping created above, but a template file has not yet been generated.
9. To create the template file from the map, right-click on the (MAP) test case in the test case tree and choose **Create CSV Template**.

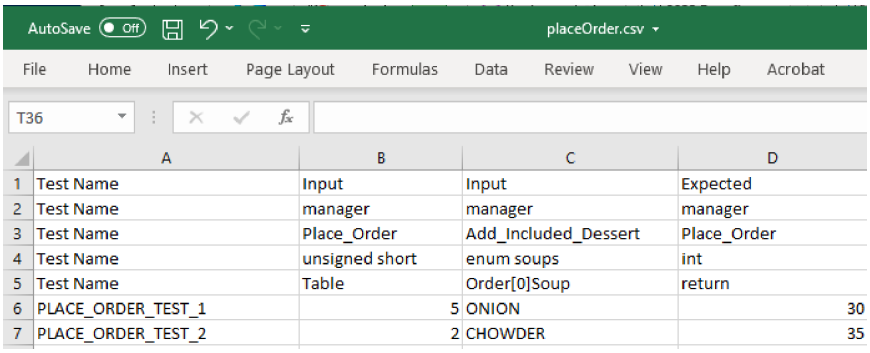


This generates the template file using the name specified in the CSV Options menu and will place the file in the directory you specified. If no path was given, the file is placed in the working directory. The file contains the five rows of header information that were shown in the Mapping View when the map was created.



	A	B	C	D
1	Test Name	Input	Input	Expected
2	Test Name	manager	manager	manager
3	Test Name	Place_Order	Add_Included_Dessert	Place_Order
4	Test Name	unsigned short	enum soups	int
5	Test Name	Table	Order[0]Soup	return

10. Open the generated file using an external tool. The first column of the file represents the test name, and the following columns represent the test data items that were selected when the map was created. Below, two test cases have been added to the template file. The first test case is named PLACE_ORDER_TEST_1, and the second is PLACE_ORDER_TEST_2. Input parameters for Table and Soup have been entered and an expected return value has also been provided. When you are done adding data, save the file, and the populated test file is now ready to be imported.



	A	B	C	D
1	Test Name	Input	Input	Expected
2	Test Name	manager	manager	manager
3	Test Name	Place_Order	Add_Included_Dessert	Place_Order
4	Test Name	unsigned short	enum soups	int
5	Test Name	Table	Order[0]Soup	return
6	PLACE_ORDER_TEST_1	5	ONION	30
7	PLACE_ORDER_TEST_2	2	CHOWDER	35

```
clicast -e <env> -u <unit> -s <sub> -t <MAP> TESt CSv2tst TEmplate
[<outputfile>]

Generate a CSV or (tab-separated) template file using template
information identified by the map test case -t <MAP>. The output
filename is specified in the MAP test case; if the file exists it will
not be overwritten.
```

To Create a Map from an Existing CSV or Tab-Delimited File

To demonstrate this feature, an example CSV file, “table.csv”, located in the Examples/c/csv_ example directory in the VectorCAST installation directory will be used. The first few lines of the file contain the following entries:

	A	B	C	D
1	Y	Slope	X	Y-Intercept
2	15	-1	-10	5
3	12.6	-0.9	-9	4.5
4	10.4	-0.8	-8	4
5	8.4	-0.7	-7	3.5
6	6.6	-0.6	-6	3
7	5	-0.5	-5	2.5
8	3.6	-0.4	-4	2
9	2.4	-0.3	-3	1.5
10	1.4	-0.2	-2	1
11	0.6	-0.1	-1	0.5

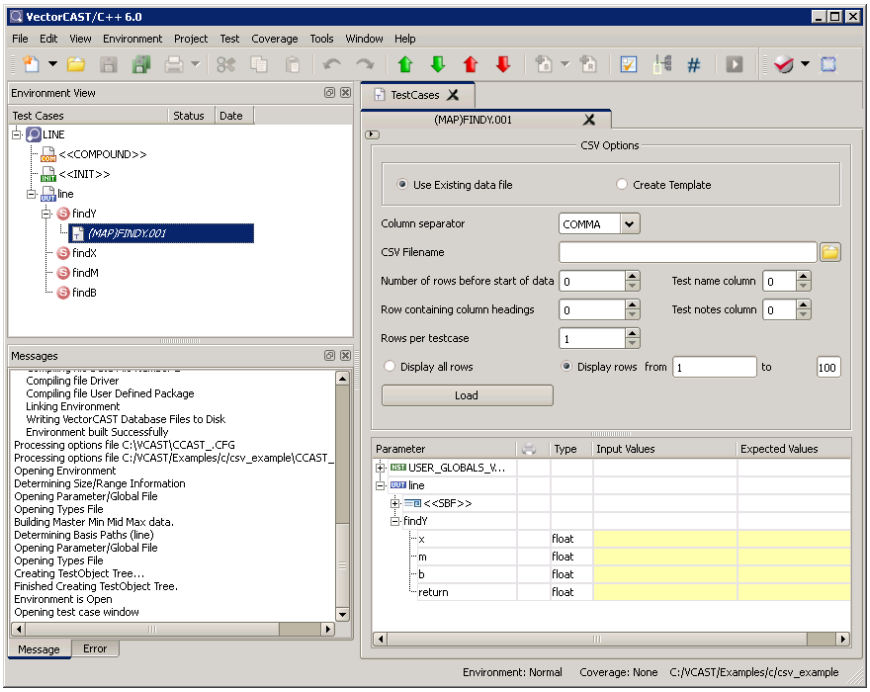
The first row of the file is a header row containing the names of the input and expected values for each test. Each of the following rows represents the test data for each test case to be created. The function *findY* will be tested using the values in “table.csv”. The source code for findY is as follows:

```
float findY ( float x, float m, float b ) {
   return m * x + b;
}
```

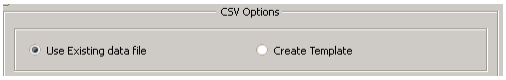
- Column A, labeled “Y,” is the expected value returned from findY.
- Column B, labeled “Slope,” is the input parameter “m”.
- Column C, labeled “X,” is the input parameter “x”.
- Column D, labeled “Y-Intercept,” is the input parameter “b”).

To create the mapping, follow these steps:

1. Create a test environment for line.c that is located in the Examples/c/csv_example directory in the VectorCAST installation directory. Right-click on the findY subprogram in the test tree and select Generate CSV Map. A (MAP) test case is created for findY and a CSV options panel appears.



2. In the CSV Options panel select the Use Existing data file radio button



3. Select the type of delimiter used to separate values. The example uses a CSV file, so select Comma-separated values.

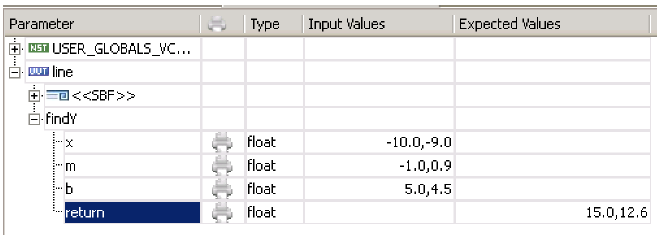


4. Select the file "CSV Filename" by browsing for the file "table.csv"
5. The controls below "CSV Filename" in the CSV Options panel provide the facility to load files that are custom tailored.

> **Number of Rows to skip** - enter the number of rows of header information at the top of the file. "Table.csv" has 1 header row, so enter 1. If there were no headings, this value would be set to zero. For template files created with VectorCAST, this value is 5.

> **Row containing column headings** - is the row that contains the data item name. This information is used by the Mapping View as the heading for each column. In the example this is row 1. If there were no heading rows, this value would be set to zero. When using a template file created with VectorCAST, this value is 5.

> **Rows per testcase** - for this example set the value to 1. Typically, each row represents a single testcase, but if you wish to combine multiple rows into one test case (the test parameters are combined as a list), set this value to the number of rows to be combined into one test case. If for example, we selected a value of 2, each parameter would have a list containing two values and two rows would be used per test case. This feature is only used with very large data files.

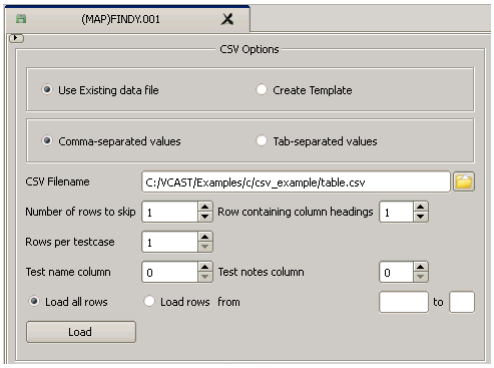


> **Test name column** – in the example, there is no column with test names, so the value is set to 0. The test names for this example will automatically be generated by VectorCAST. When using a template file created by VectorCAST, the name column is 1.

> **Test notes column** – in the example, there is no column containing test notes, so the value is set to 0. When test notes are available, they will be added to the “Notes” tab of the test. Place the column number containing the notes in this field. When using a template file created by VectorCAST, the name column is 0.

> **Load all rows** - selecting this option will load all of the rows from the file into the Mapping View.

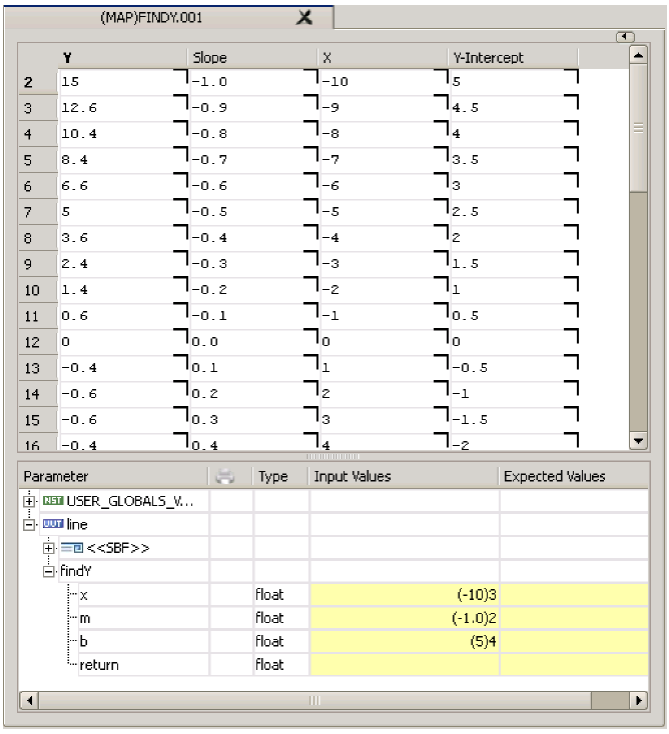
> **Load rows from/to** – this allows a subset of rows to be loaded into the Mapping View, specifying a starting and ending row. If you have a very large CSV or tab-delimited file and only want to a see a small subset in the Mapping View after loading, select this option and specify the range of rows to load.



6. Click **Load** to open the Mapping View using the current settings. After clicking **Load**, the CSV Mapping View appears. It contains the rows selected for loading and is used to map the file columns to test data items of the subprogram.
7. To map a column in the Mapping View to a test case data item, drag any value in a column in the Mapping View and then drop it to a cell in the Parameter Tree.

When you select a cell in the mapping view, all cells with a matching data type in the Parameter Tree are highlighted in yellow. This helps you select an appropriate cell for mapping. You can drag a value from any row. The number in the parentheses in the Parameter Tree is the value contained in the cell selected in the Mapping View that was dragged into the Parameter Tree. The number to the right of that is the column number in the Mapping View that is mapped to the test data item.

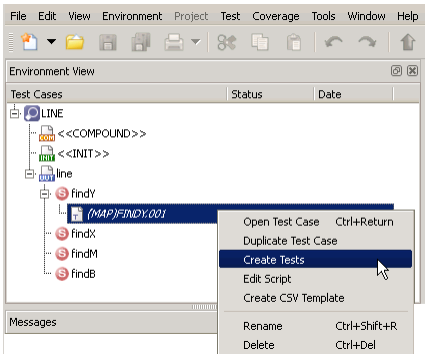
8. When you have completed mapping all of the columns to input or expected values, click the Save icon in the toolbar or choose **File => Save**.



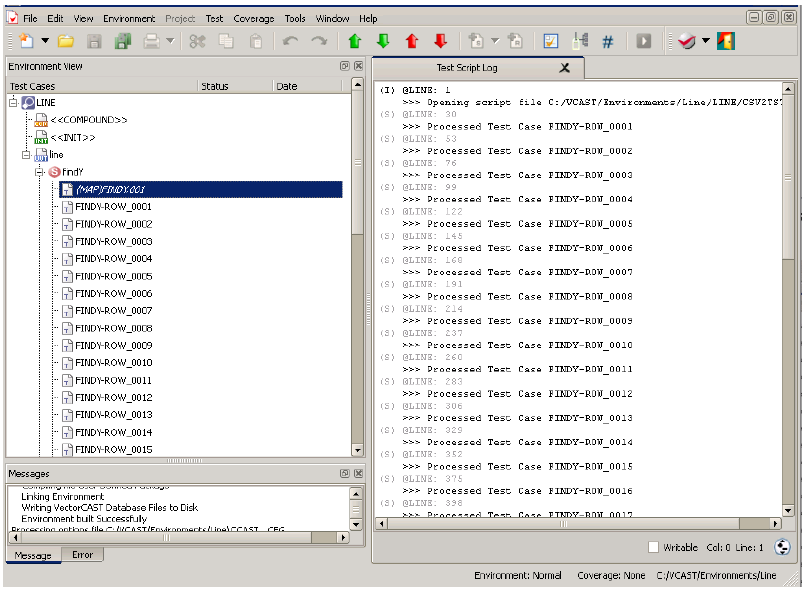
9. Mapping is now complete, and the map may be used to create tests.

To Create Test Cases from a MAP Test Case

Right-click on a (MAP) test case in the Test Case tree and select **Create Tests**. A dialog box indicating the number of tests to be generated is presented. Click **OK** to proceed.



All tests are created and appear in the Test Case tree, and a test script log is displayed. You may now select and execute the tests.

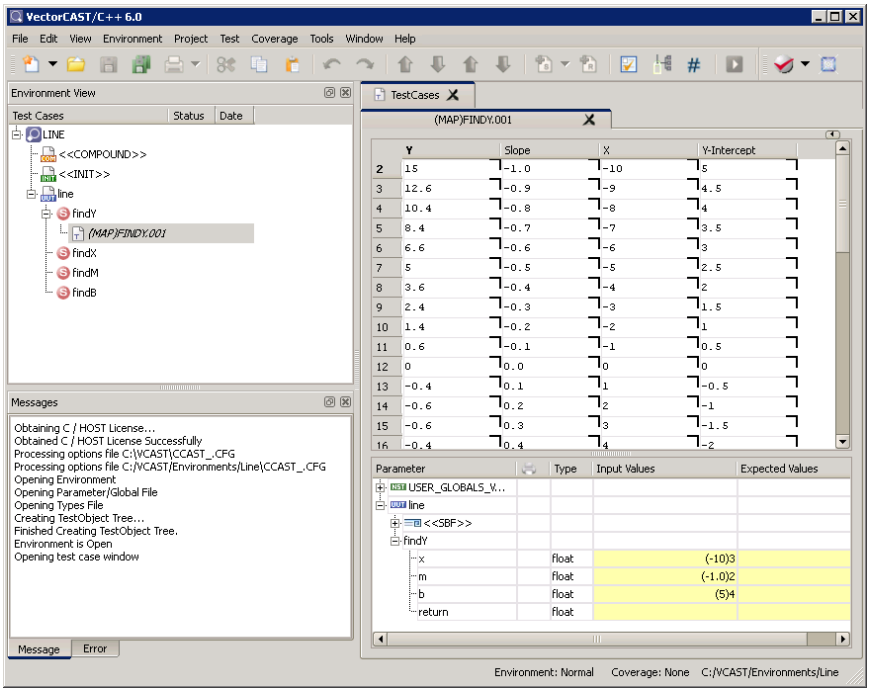



```
clicast -lc -e <env> [-u <unit> [-s <sub> [-t <test>]]] TESt CSv2tst Load

Generate a test case containing a test case for each row in the CSV file
identified by map test case(s) in the environment.
```

To Edit an Existing Map Test Case

An existing map test case may be edited by double clicking the map entry in the Test Case Tree. A Mapping View and Parameter Tree will appear.



The cells that are highlighted in yellow in the Parameter Tree are available for editing. The value in parenthesis in the Parameter Tree cell is the column value for the data item in the map. For example in the picture above, “x” in the Parameter Tree contains (10). This is the same value for “X” in the first row of the Mapping View. The number in the cell outside of the parenthesis is the column number of the Mapping View that the data item is mapped to. In this example, “X” is column 3 in the Mapping View, and cell in the Parameter Tree also contain a 3. This allows you to easily verify the current mapping and to make changes if necessary.

To remove a data item from the map, simply click in the Parameter Tree cell and delete the column number entry and press **Enter**.

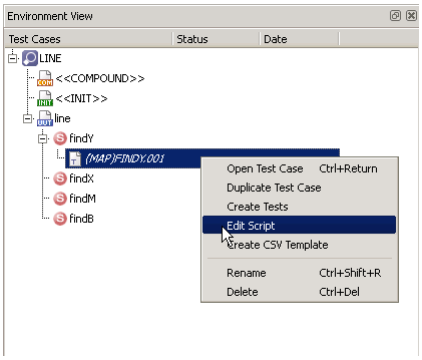
To map a column that is currently not mapped to the Parameter Tree or to change a currently mapped column, click a value in the column and drag and drop it into the desired cell of the Parameter Tree. You may also select a Parameter Tree cell and type a column number directly and pressing **Enter**. You will see the value and its column position immediately updated in the Parameter Tree upon releasing the mouse button or pressing Enter.

When you are finished editing the map file, click the **Save** button on the Toolbar  or select **File => Save**.

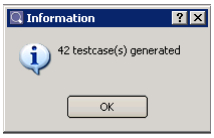
To Create and Edit a Test Script from a Map File

Right-click on a map file in the Test Case tree, and select **Edit Script**

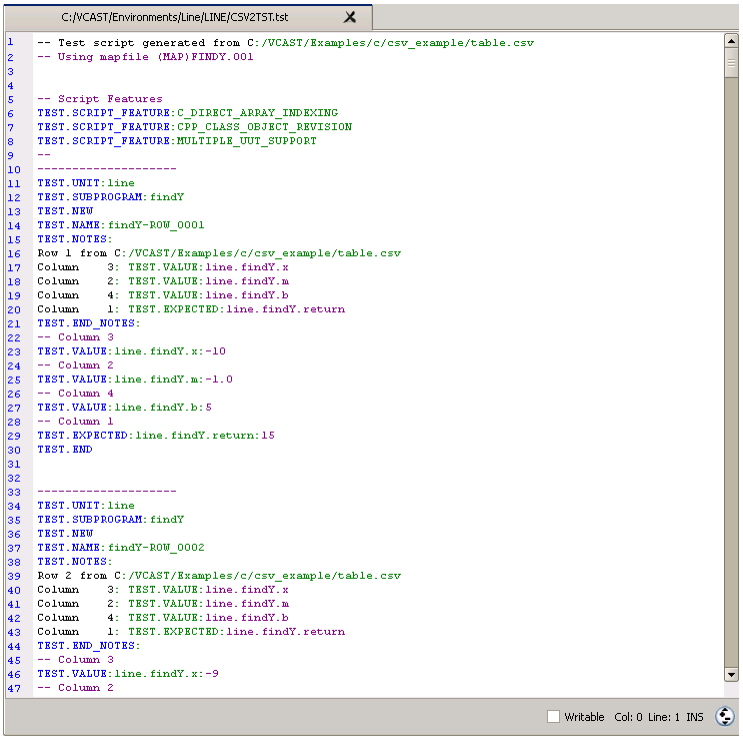
.



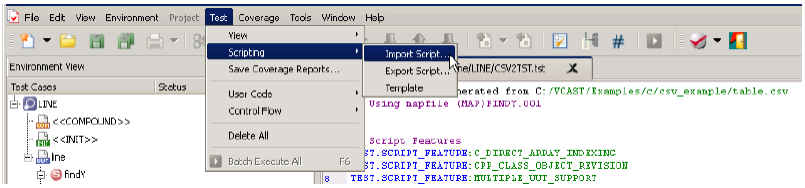
A dialog box appears indicating the number of tests generated from the associated CSV or tab-delimited file. Click **OK**.



A test script, containing all of the generated tests appears in a test script editor.



Review the tests and make any changes you wish to the script file. The test script can be imported by selecting **Test => Scripting => Import Script**




```
clicast -lc -e <env> [-u <unit> [s- <sub> [-t <test>]]] TESt CSv2tst CReate
<test script>

Generate a test script containing a test case for each row in the CSV file
identified by map test case(s) in the environment. Output file <test
script> must be specified.
```

Generating Test Cases Using the Vector Test Data Editor

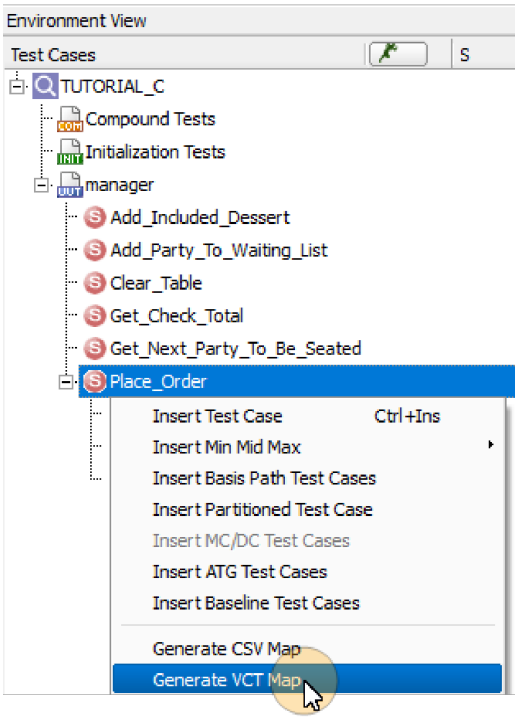
The Vector Test Data editor is used to generate automated test cases using a Classification Tree Method for C or C++.

Note: The Test Data editor is only available on Windows64 platforms and requires an Enterprise Edition License.

Generating test cases using the Test Data editor is a multi-step process, similar to using comma-separated value files (**`.csv`**) to specify the parameter values. The following steps describe the process.

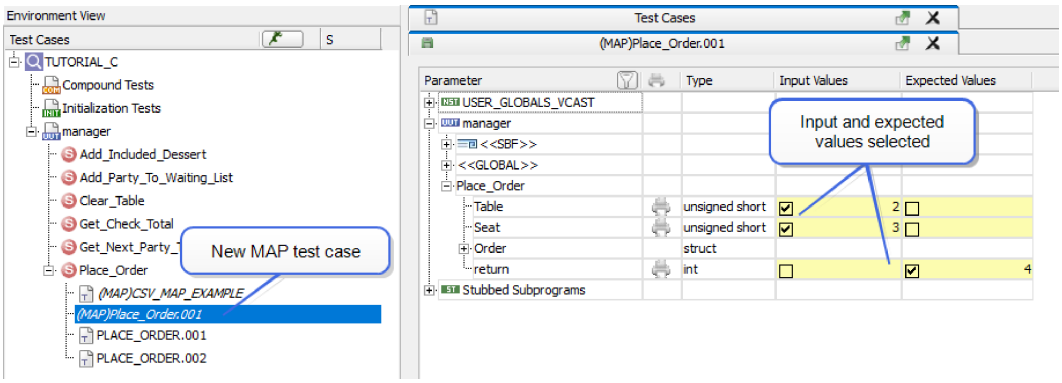
Make a New VCT MAP Test Case

In a C/C++ unit test environment on Windows64, insert a VCT MAP test case under a subprogram by right-clicking the subprogram and selecting **Generate VCT Map** from the menu.



In our example, the inserted MAP test case is named *(MAP)Place_Order.001*. This MAP test case cannot be executed; only the generated children are executed.

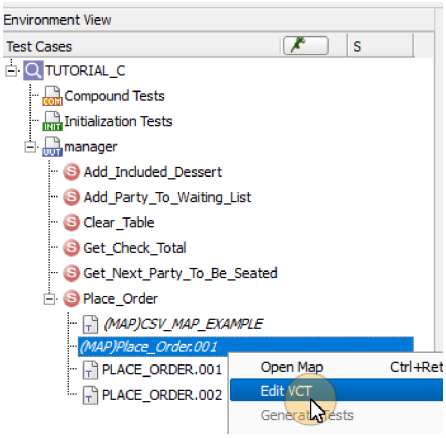
The MAP test case opens in the Parameter Tree, enabling you to place a checkmark next to any Input or Expected Values that you intend to exercise in the generated tests. You must save your selections when complete.



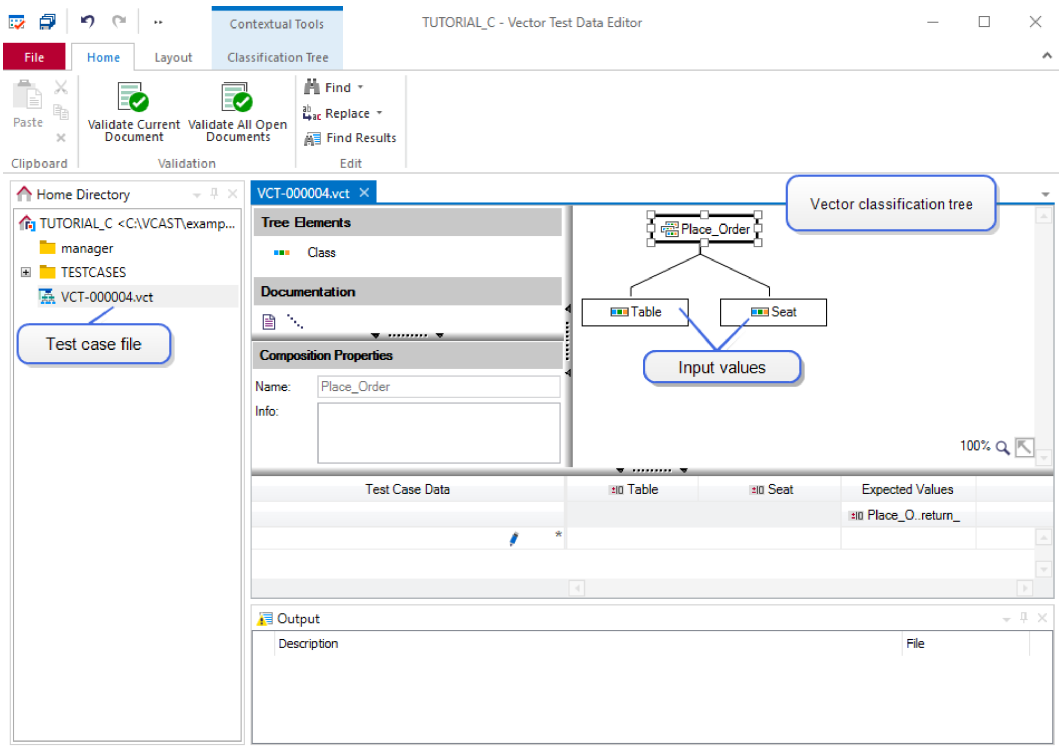
Note that no data is set in this window; only the parameters you plan to use in the Test Data editor are indicated.

Use the Test Data Editor to Set Input and Expected Values

Right-click on the new MAP test case and select **Edit VCT** from the menu.

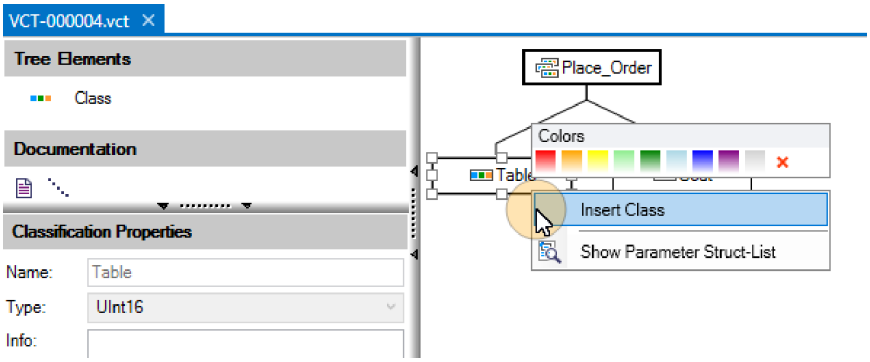


The Test Data editor opens. The Test Data editor creates one file per MAP test case and places it in the environment directory. The files use the naming convention **`VCT-nnnnnn.vct`**, where **`nnnnnn`** represents a six-digit unique identification number corresponding to the MAP test case. In our example, the file is named **`VCT-000004.vct`**.

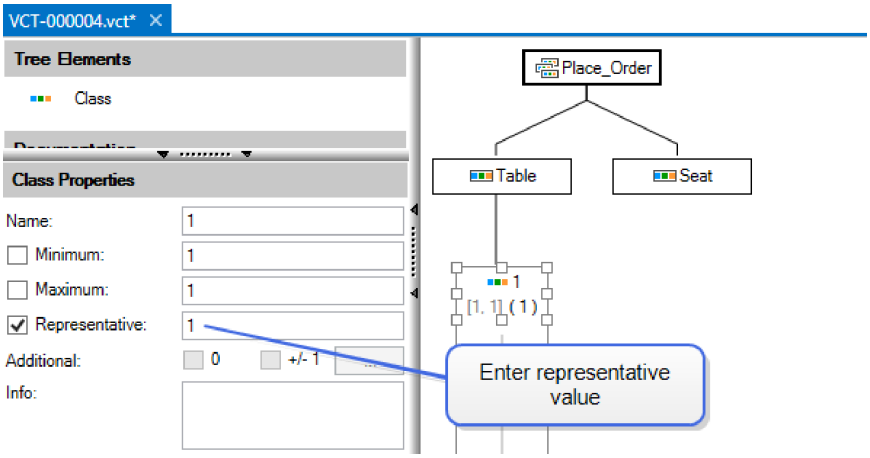


The Test Data editor displays the Vector Classification Tree, showing an item for each input value you checkmarked in the MAP test case. In our example we have selected Table and Seat as input values.

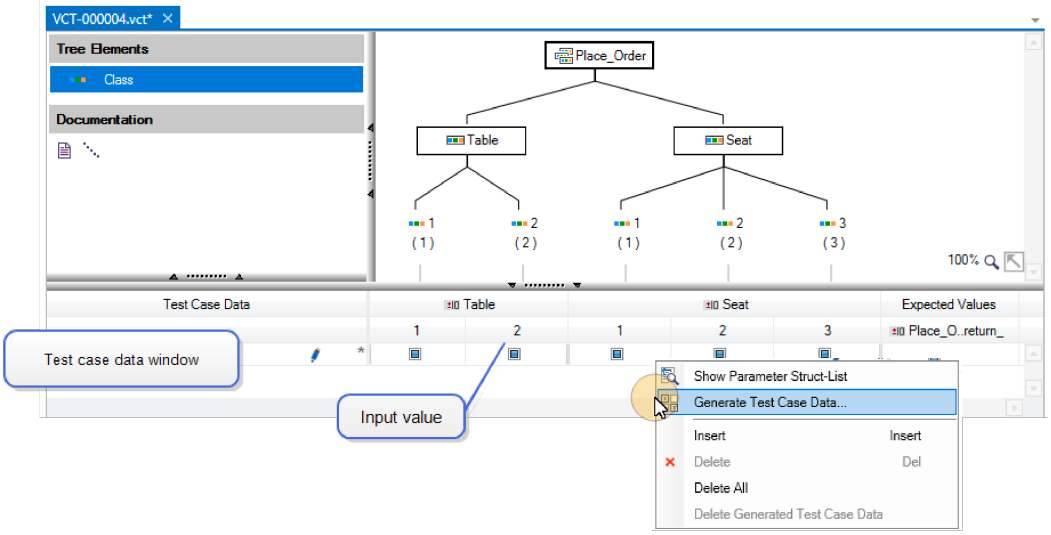
Define test values to be used as Input by right-clicking an item and selecting **Insert Class** from the menu.



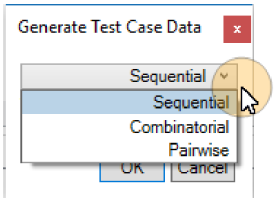
Click on a class in the Classification Tree and the Class Properties pane opens. Set a representative value to be used as the Input Values for the selected parameter. Additional information can also be entered. For our example, we will enter **1** as a representative value for our selected class, and also enter **1** as the class name.



Along the bottom of the Test Data Editor is the Test Case Data window. Right-click anywhere in the Test Case Data pane and select **Generate Test Case Data...** from the menu.

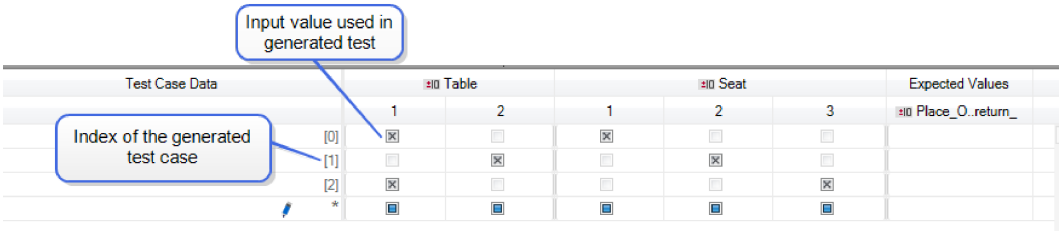


The Generate Test Case Data dialog opens. Use the drop-down menu to select the method of automatic test case generation. The options are:
> **Sequential** - Test case data sets are generated by using each equivalent class at least once.
> **Pairwise** - Test case data sets are generated so that all equivalent class pairs are used at least once.
> **Combinatorial** - Use for higher test coverage. This option ensures that all possible combinations of equivalent classes are generated.



VectorCAST generates test cases for each test defined in the VCT file. For our example, we will select the Sequential option.

The test cases are displayed in the Test Case Data window. Each Input Value heads a column, and can be used in a generated test by putting a checkmark in that column. Any Expected Values selected in the MAP test case are on the right-hand side, where you can enter a value manually.



For more detailed information on using the Test Data editor, see the Test Data Editor Help included in your installation directory and located at **`$VECTORCAST_DIR/test_data_`**

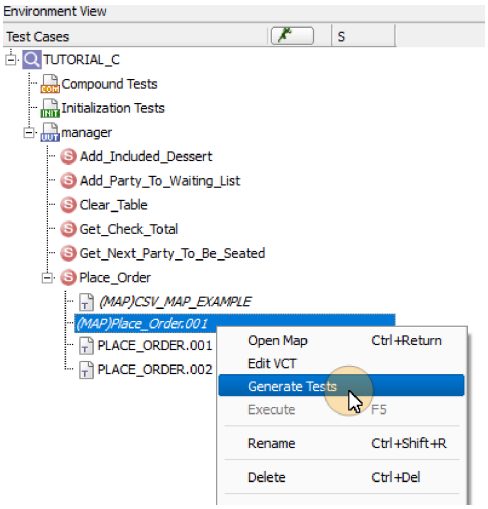
editor/TestDataEditor.chm.

When done, save and close the Test Data Editor, and return to the VectorCAST environment.

Generate Automated Tests

After closing the Test Data Editor and returning to VectorCAST, you will receive a prompt asking if you would like to generate the test cases that were entered in the Test Data Editor. Select **Yes** to generate the test cases or **No** to continue to the VectorCAST environment.

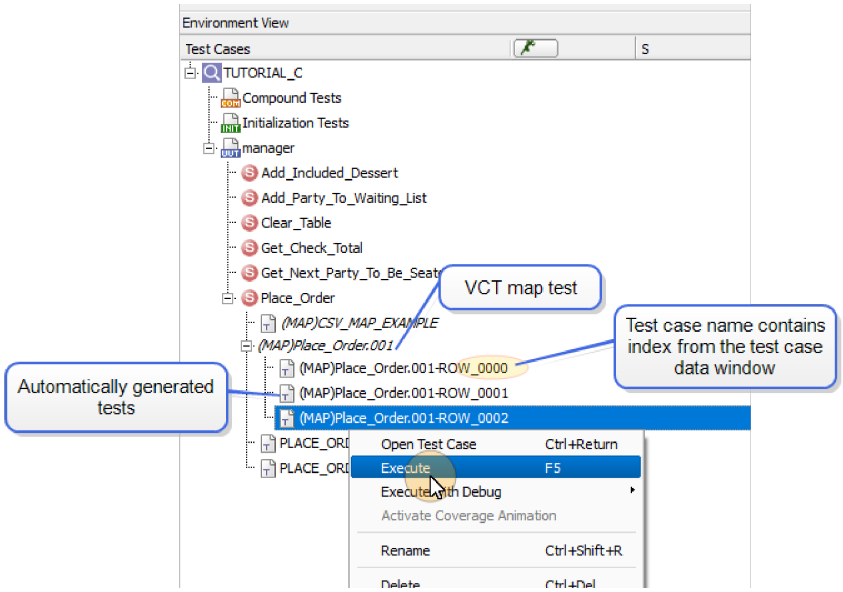
You can automatically generate a test case for each row in VectorCAST at any time once you have specified Input and Expected Values in the Test Data Editor. Right-click the MAP test case and select **Generate Tests** from the menu. One child test case is created under the MAP test case for each row in the Test Data Editor



Note that when selecting the Generate Tests menu option, any tests for the Map that already exist will be deleted and replaced with new tests.

Generated test cases are treated as read-only in the VectorCAST GUI. Additionally, the following features are unavailable for generated test cases: Notes, Test Case User Code, Test Case Options, Requirements, and Control Flow.

The generated tests can be executed individually or in bulk (by executing the VCT Map test) with a pass or fail result if Expected Value(s) were provided in the Test Data Editor.



Export/Import VCT MAP Test Case and .vct File

When exporting the MAP test case, only the MAP data is saved in the VectorCAST test script (**`.tst`**). The generated tests are not exported, but are re-generated upon import. In addition, and very importantly, the **`VCT-nnnnn.vct`** file is also exported. This file must be adjacent to the test script during test script import.

Although generated test cases cannot be exported with the GUI or the normal export command, generated tests can be exported via the following clicast command:

```
clicast -lc -e <env> [-u <unit> [-s <sub> [-t <test>]]] TESt VCt2tst
CReate_detached <test script>

Generate a test script containing a detached test case for each generated
test associated with the specified VCT Map testase(s). The Map testcase(s)
are not included in the test script.
```

See the KnowledgeBase article The Vector Test Data Editor for an online tutorial on using the Test Data editor.

VectorCAST Tools

To Set VectorCAST Options

The **Tools => Options** dialog is used to access the many options and settings available to configure VectorCAST. To access the dialog, choose **Tools => Options...** or click the **Options** button  on the toolbar. The options are grouped by category on the following tabs:

- GUI
- C/C++
- Wizard
- Builder
- Execute
- Report
- Target
- Coverage

When you click the **OK** or **Apply** button, VectorCAST writes the options to the CCAST_.CFG file, located in the current working directory. This directory is noted in the status bar of the main window. Any changes made to the configuration file in the working directory affect other environments in the directory.

If you run VectorCAST in command line mode (clicast), the same file is accessed for Options settings.

When the VectorCAST application is launched, it looks for a configuration file in the following locations:

- VectorCAST installation directory
- User's home directory
- Current working directory

Options are read in the order specified with each file taking precedence over previous files.

If you want some options to apply to all environments (e.g., the compiler you are using), you can set these options in the VectorCAST installation directory.

If you have VectorCAST installed on a network and each user has a client installation, you can create a configuration file that applies to all users of VectorCAST. To do this, set the working directory to network drive, and set any options you want all users to have, such as compiler settings. This configuration file is used unless settings from an environment-specific configuration file overrides them.

If you add a line or make a change to your configuration file external to VectorCAST (while it is running), you can read those values back into VectorCAST. To do this, click the **Load** button. The changes that you made to the file are now reflected in the Options dialog box.

The default values for the current active tab can be reset by clicking the **Default** button.

Clicking **Cancel** closes the dialog box, without saving any changes since the last time you clicked the Apply button.

See "To Set the Working Directory" on page 103 for information on changing the current working directory.

See "CLICAST - Command Line VectorCAST" on page 696 for information on setting options in CLICAST.

Tools Menu

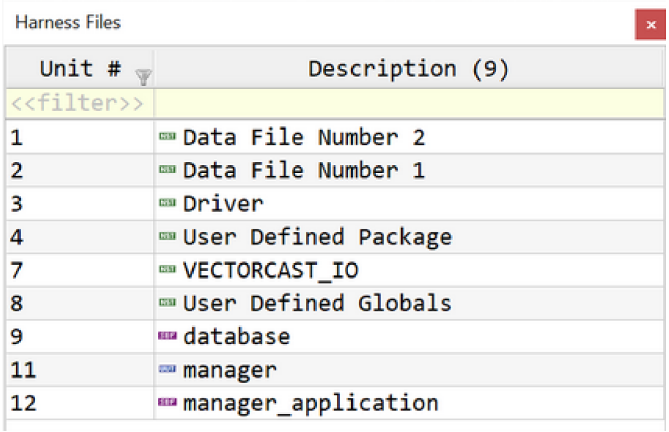
The following sections cover the Tools Menu selections. The selections on the menu that are discussed are:

- View Harness Source
- Basis Path
- MC/DC
- Coupling (see *VectorCAST/QA User's Guide* section "Using Coupling Verification")
- Industry Mode (see "To Set the Industry Mode for Coverage" on page 30)
- Custom Tools (see "Working With Custom Tools" on page 380)
- Requirements Gateway (see "Using the Requirements Gateway" on page 630)
- Options (see "To Set VectorCAST Options" on page 376)

View Harness Source

The View Harness Source command enables you to view the test harness source code for all of the units in a test environment. To view the harness files, select **Tools => View Harness Source**.

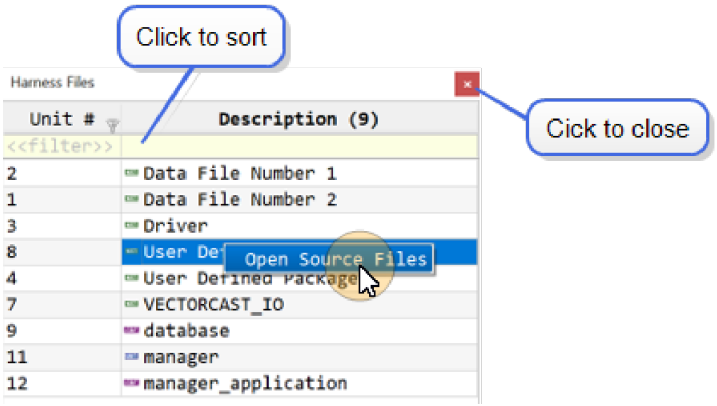
The pop-up window contains the UUT and any dependent units, along with all the harness source files. Selecting one of them opens up the source code file. This feature provides a convenient way to browse the source code as tests are being developed. The figure below shows the pop-up window for an environment having **`manager`** as the UUT and having **`database`** and **`manager_application`** as dependent units.



To aid in cross-referencing the source filenames with the VectorCAST unit numbers, the VectorCAST unit numbers are displayed in the Harness Files window.

To view one of the files, select the name, right-click, and click **Open Source File**. Clicking on the

header bars at the top sorts the files in reverse order by Unit #, alphabetically by description. Clicking again reverses the order. To dismiss the window, click on the button located in the upper right corner.



Once you have chosen a harness source file to view, it appears in the MDI window. You can view the source file or even edit it here. If you change the source code file, and want the changes incorporated into the test harness, choose **Environment => Recompile**. Keep in mind that any changes you make to the test harness source files are not reflected in your original source code files, as these files are separate and distinct from the test harness.

See also "To Set Up an External Text Editor" on page 63.

See also "To View the Source for a Subprogram" on page 254.

Basis Path

This menu item has four options:

- Perform Analysis
- View Analysis Report
- Generate Tests
- Generate Test Scripts

To Perform a Basis Path Analysis

The **Tools => Basis Path => Perform Analysis** command enables you to generate a Basis Path Analysis. The contents of this analysis reflect the items selected in the Test Case Tree.

To View a Basis Path Report

The **Tools => Basis Path => View Analysis 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.

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.

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

Create a Basis Path report and output to HTML file <env>_basis_path_
report.html, standard output as text, or to the specified output file. If
the Unit parameter is specified, the Basis Path report includes only the
specified unit; otherwise, the Basis Path report includes all units in the
environment.
```

See also "To Insert Basis Path Test Cases" on page 257 for information on having VectorCAST automatically generate a test case for each basis path.

To Generate Basis Path Tests

The **Tools => Basis Path => Generate Tests** command enables you to generate the Basis Path Tests for the selected units in the environment.

To Generate a Basis Path Test Script

When you choose **Tools => Basis Path => Generate Tests**, VectorCAST creates a test script containing all of the automatically generated tests, and then loads this script into the test environment.

To (Re)Generate the Basis Path Tests

After changing your source code, recompiling your units, and relinking the test harness, you may want to regenerate the basis path report. Choose **Tools => Basis Path => Generate Tests** to force VectorCAST to fetch the latest version of the UUTs, and regenerate the basis path report. If you have coverage on, this command will reset the coverage data. Use this command to compute the basis paths if the environment was built without basis paths (the option "**`Do not automatically generate basis paths`**" was set).

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

Update the statement basis paths for the environment.
```

See also "To Incorporate Source Code Changes into Environment " on page 232.

MC/DC

This menu item has two options:

- View Equivalence Matrices
- Test Case Analysis (this menu item has four sub-options):
  >> Perform Analysis
  >> View Analysis Report
  >> Generate Tests
  >> Generate Test Scripts

To View Equivalence Matrices

Selecting **Tools => MC/DC => View Equivalence Matrices** will display the MC/DC Condition Tables Report for the selection in the environment.

To Perform MC/DC Analysis

Selecting **Tools => MC/DC => Test Case Analysis => Perform Analysis** will perform a new MC/DC analysis for the selection in the environment.

To View the MC/DC Report

Selecting **Tools => MC/DC => Test Case Analysis => View Analysis Report** will display the MC/DC analysis for the selection in the environment.

To Generate MC/DC Tests

Selecting **Tools => MC/DC => Test Case Analysis => Generate Tests** will generate MC/DC tests for the selection in the environment.

To Generate MC/DC Test Scripts

Selecting **Tools => MC/DC => Test Case Analysis => Generate Test Scripts** will generate MC/DC test scripts for the selection in the environment.

Working With Custom Tools

You can add external utilities or applications to the VectorCAST Tools menu and toolbar. The associated tool is added to the Custom Tools dialog as well.

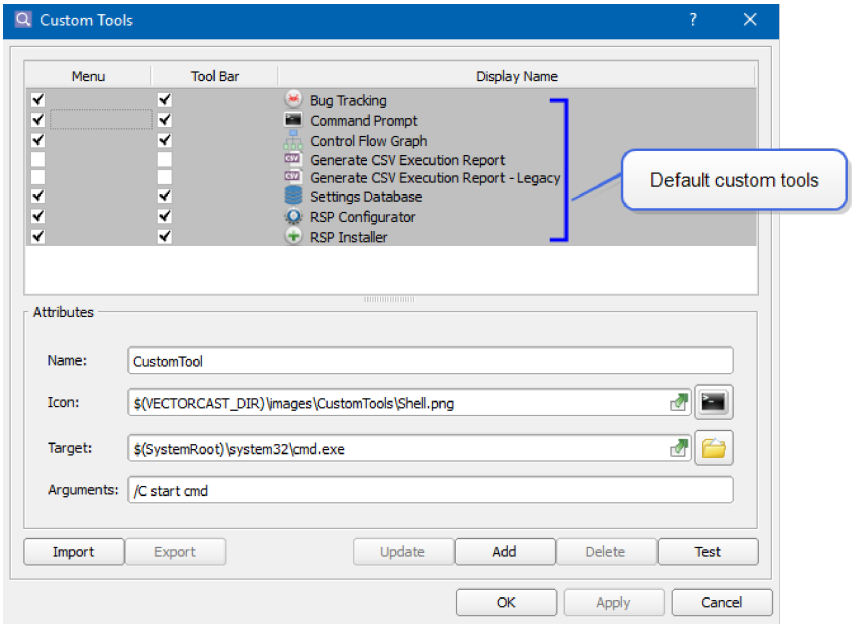
With the Custom Tools dialog, you can:

- Add a new tool.
- Delete a tool.
- Choose an icon for the tool.
- Specify that the icon appear in the VectorCAST Tool Bar and Custom Tools Menu.
- Update the settings for a tool.
- Test a tool.
- Import and Export custom tool configurations.

Default Custom Tools

VectorCAST provides default custom tools for users' convenience. The Custom Toolbar contains the default custom tools and is displayed in the main Toolbar by default. Custom tools are also displayed in the Custom Tools menu. The following custom tools are provided:

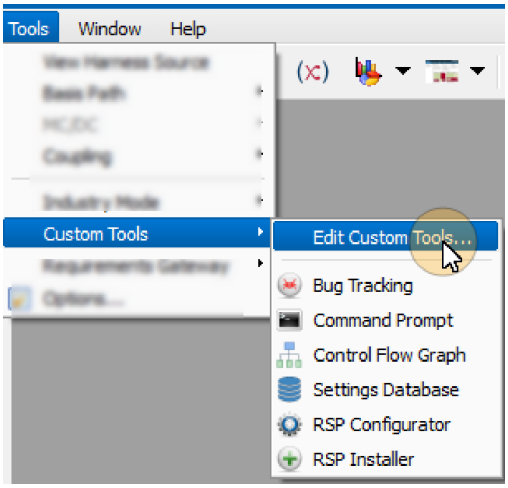
- Bug Tracking - Allows users to create bugs via an integrated bug tracking tool.
- Command Prompt - On Windows, a DOS command prompt is available. On Linux, an XTerm window is available.
- Generate CSV Execution Report - Dumps the Execution Report for the currently selected test case in CSV format, and on Windows, opens the resultant file in Microsoft Excel. By default, this custom tool is not shown in the Toolbar or the Custom Tools menu.
- Settings Database - Provides a visual editor for use with a vcshell database file.
- RSP Configurator - Allows users to select the compiler family and modify and save associated arguments.
- RSP Installer - Allows users to download and install customer-specific VectorCAST/RSPs.



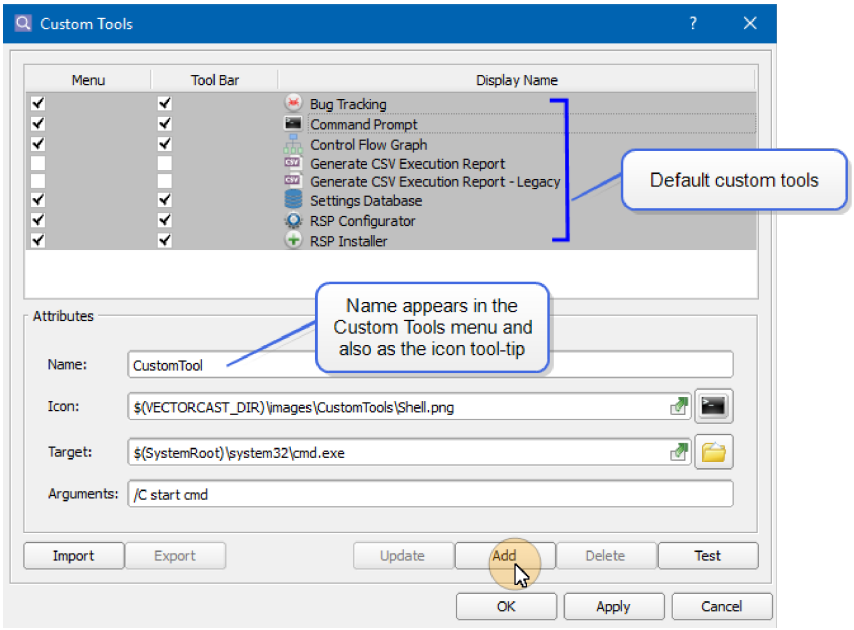
Default custom tools are displayed with a gray background color, and are not editable by the user.

Add a Custom Tool

Add a custom tool by selecting **Tools => Custom Tools => Edit Custom Tools...** from the Main Menu and open the Custom Tools dialog.



The Custom Tools dialog appears:



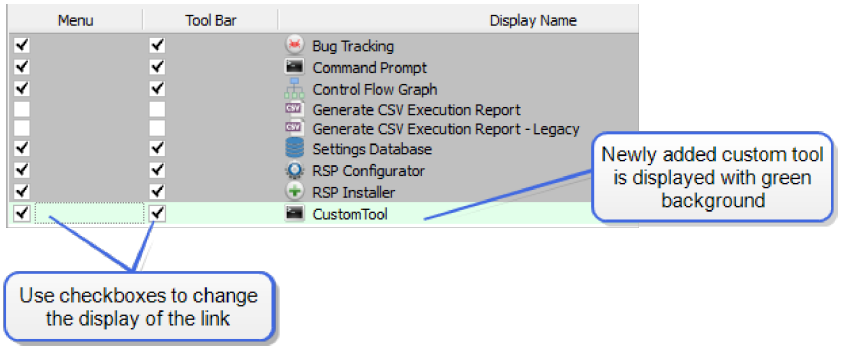
To add a tool, enter the label you want to appear in the Custom Tools menu into the **Name** field. The name is also used as the tool-tip for the icon.

Optionally, enter or browse for the path to the icon image file in the **Icon** field.

Enter or browse for the path to the relevant application file. The path to the executable file, shell script, or batch file is entered into the **Target** field.

Enter any arguments needed for the target in the **Arguments** field.

Select the **Add** button.



The new custom tool is added to the list of Custom Tools. The background for the tool is temporarily set to green, to aid the user in identifying and tracking newly added items. Clicking the **Apply** button adds the tool to the list and the background turns to white.

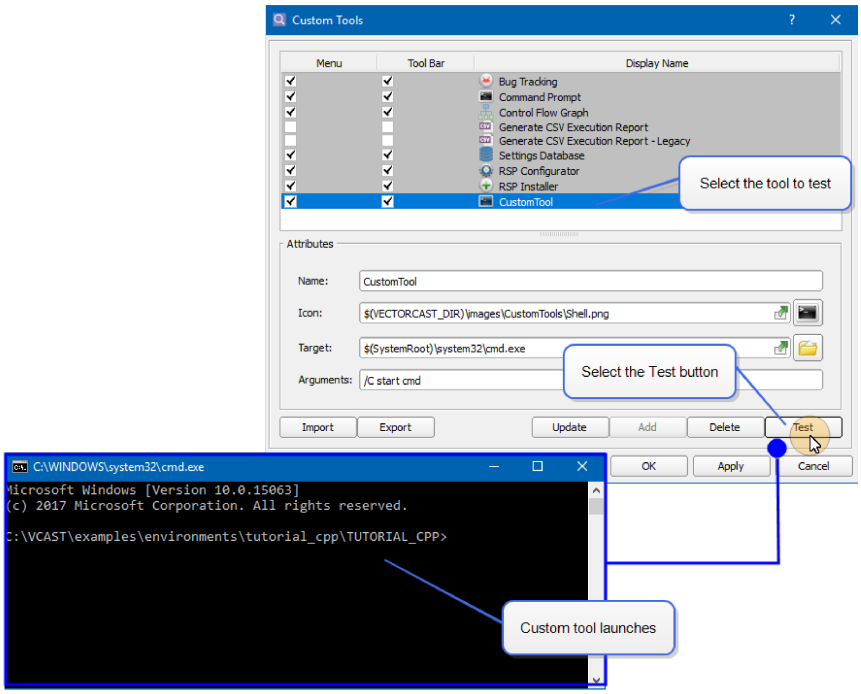
Use the associated check boxes to change the display of the link. By default, the tool's link is displayed in the both the Custom Toolbar and the Custom Tools menu.

Clicking **OK** closes the dialog.

Test the Tool

Test a custom tool by selecting **Tools => Custom Tools => Edit Custom Tools...** from the Main Menu and open the Custom Tools dialog.

To test the tool to make sure it works, press the **Test** button. The tool opens in a separate window.

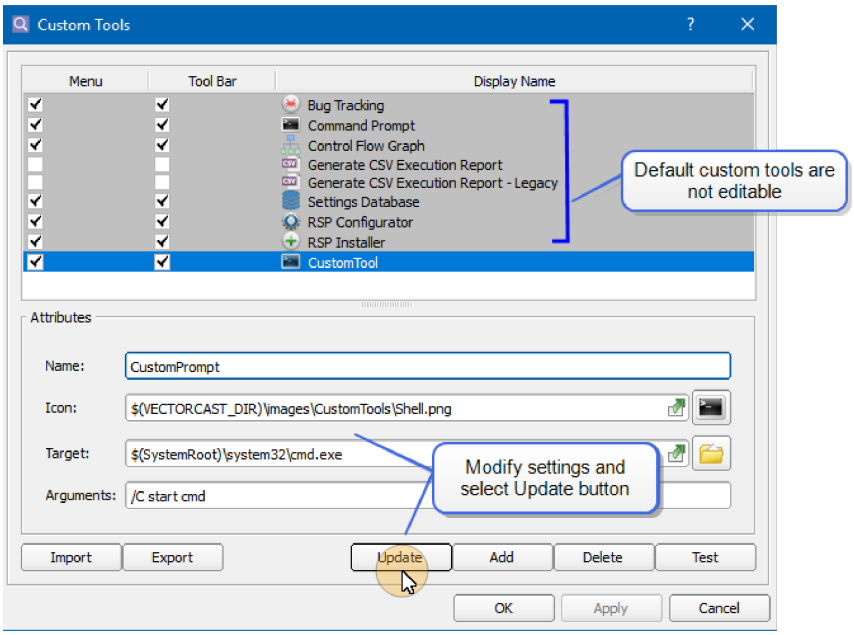


Update the Settings for a Tool

Update the settings for a custom tool by selecting **Tools => Custom Tools => Edit Custom Tools...** from the Main Menu and open the Custom Tools dialog.

Highlight the tool in the Display Name column. The settings for the tool are displayed in the Attributes pane.

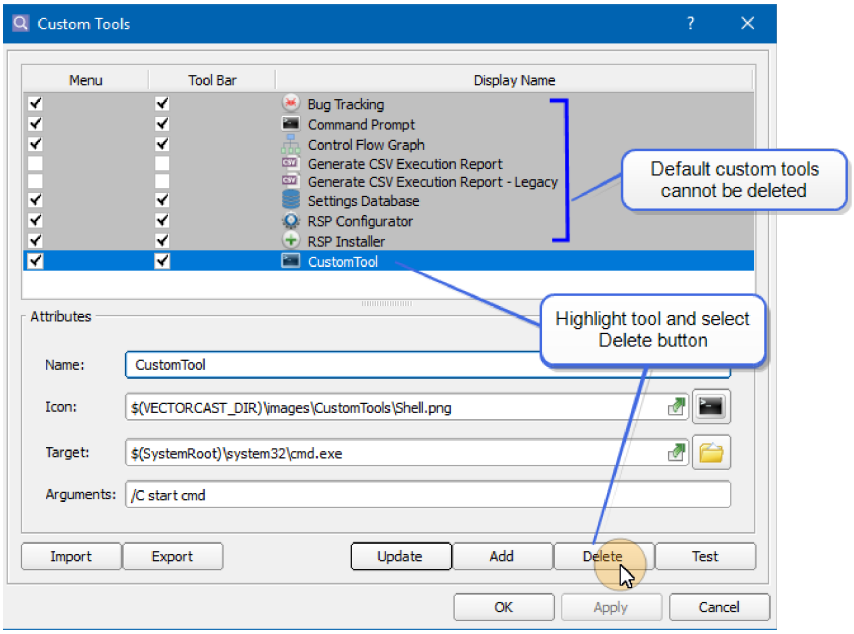
To update the settings for a tool, in the Attributes pane, make changes in the Name, Icon, Target, or Arguments fields. Click the **Update** button. Click **OK** to close the dialog.



Note: Default custom tools are not editable.

Remove a Custom Tool

Remove a custom tool by selecting **Tools => Custom Tools => Edit Custom Tools...** from the Main Menu and open the Custom Tools dialog.

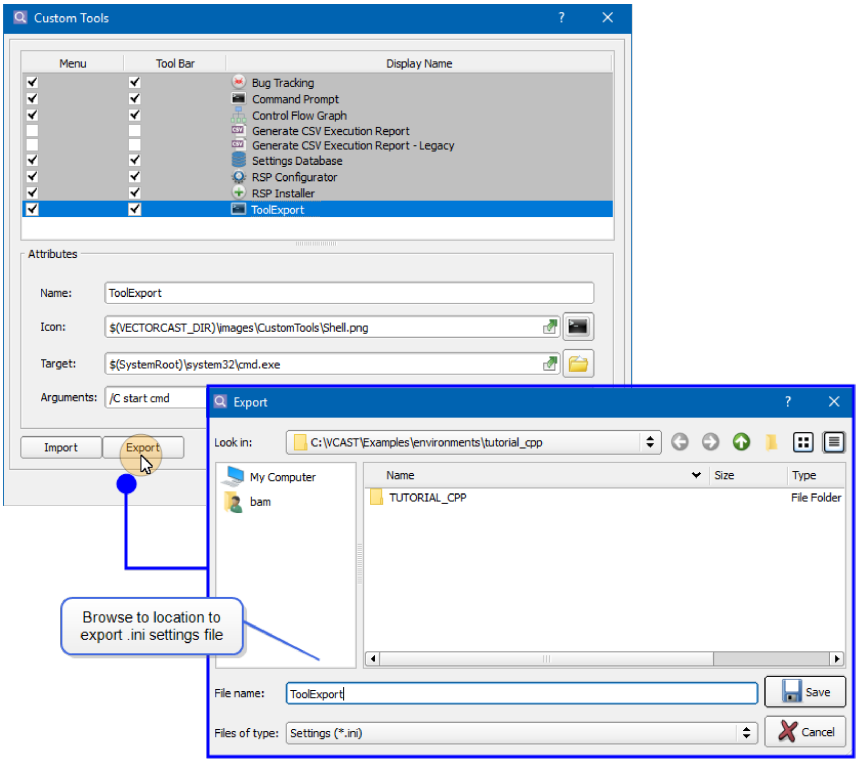


Select the tool to delete and then click the **Delete** button. Click **OK** to close the dialog.

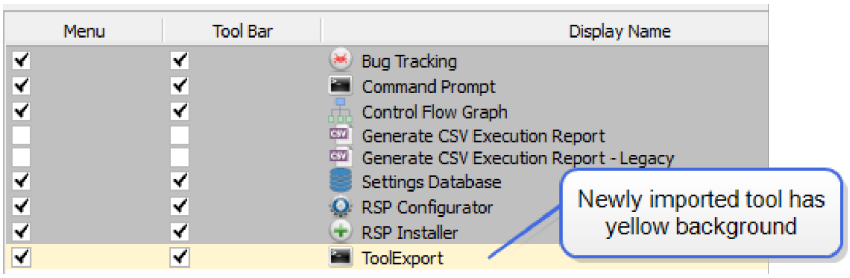
Import/Export Custom Tools

You can share Custom Tools with other VectorCAST users using the Import / Export functionality of the Custom Tools dialog.

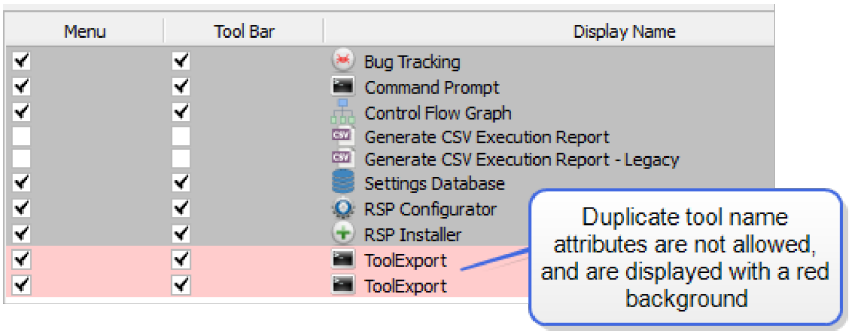
To export an existing custom tool, highlight the tool and select the **Export** button. Note that default custom tools cannot be exported. Browse to the location where you will store the `.ini` settings file and enter the file name. In our example, the file is named `ToolExport.ini`.



To import a custom tool, open the Custom Tools dialog and select the **Import** button. Browse to the location of the `.ini` settings file and select **Open**. The tool is added to the Custom Tools list, and the background for the tool is temporarily set to yellow, to aid the user in identifying and tracking newly imported items.

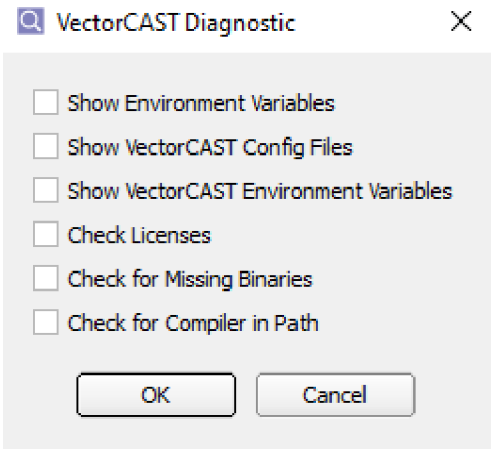


If VectorCAST detects a duplicate name attribute in the list, the background for the tool is set to red to alert the user of the conflict.



To Create a Diagnostic Report

The diagnostic report is useful to diagnose tool configuration problems. To run diagnostic checks, choose **Help => Diagnostics => Create Diagnostic Report**. A dialog box appears, allowing you to check which items to want to see in the report. When you click OK, the report opens in the Report Viewer. You can save the information to a file using the **File => Save As** command.




```
clicast -lc REports DIagnostic [<outputfile>]

Generates diagnostic report to standard output or the specified output
file.
```

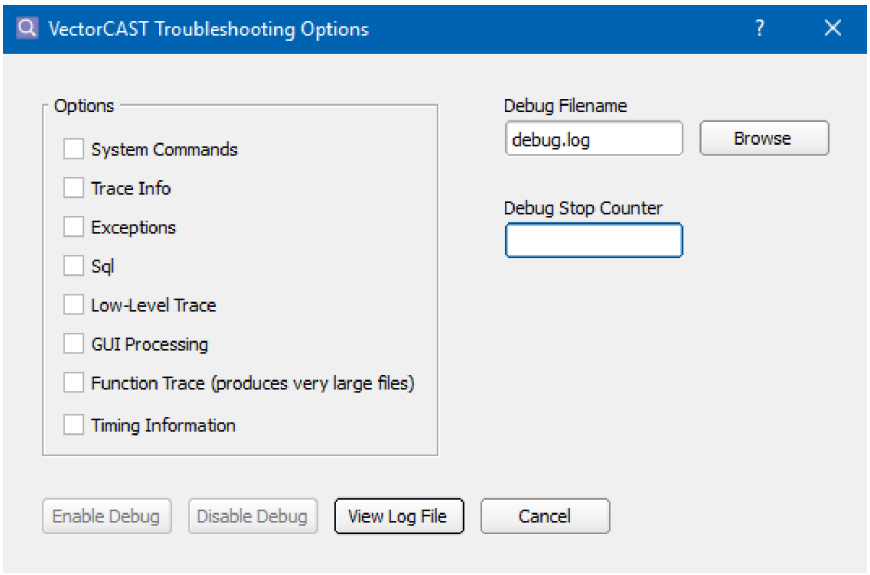
Troubleshooting VectorCAST

Note: This feature should be used only as directed by VectorCAST's Technical Support to resolve issues in the field.

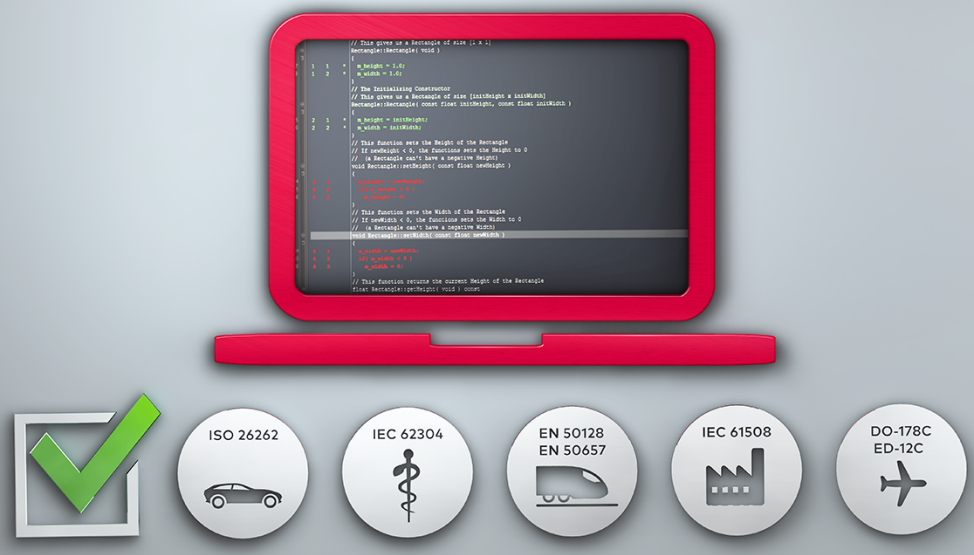
To troubleshoot VectorCAST and create a debug log, choose **Help => Diagnostics => Troubleshooting**. A dialog box appears, allowing you to check which items to want to see in the log. Click the Browse button to specify a location for the file. When you click **Enable Debug**, a debug log is written while you run VectorCAST.

When debug logging is enabled, the default log output location defaults to a file with the name prefix

debug.log. To override this prefix, the environment variable VCAST_DEBUG_FILENAME can be specified. To direct debug log output to **stdout** instead of a file, set VCAST_DEBUG_FILENAME to "-".



To view the log later, choose **Help => Diagnostics => Troubleshooting**. Click the **View Log File** button. In the **Open File** dialog, navigate to the **debug.log.* file**, and click **Open**. The debug log is displayed in the MDI window.



Executing Tests

Executing Test Cases

To Execute a Test Case and View Results

There are several ways to execute a test case:

- Right-click on a test case and select **Execute** from the context menu.
- Select a test case to run, then click this toolbar icon: 
- Select a test case to run, then press the **F5 key** on the keyboard.
- Choose **Test => Batch Execute All**

You can multi-select any combination of test cases, subprograms, and units, before choosing Execute.

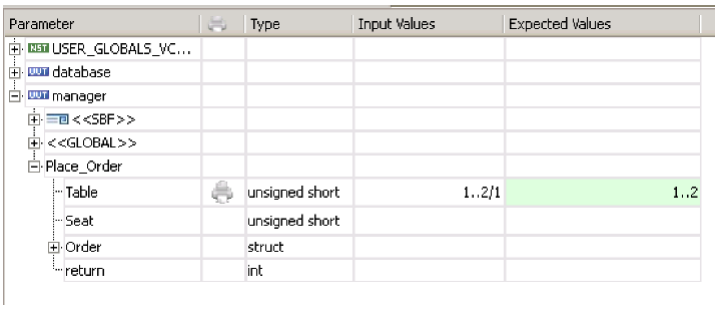
When a test case is executing, the Message window gives you feedback on the progress. It informs you as it processes each event and when test execution is complete.

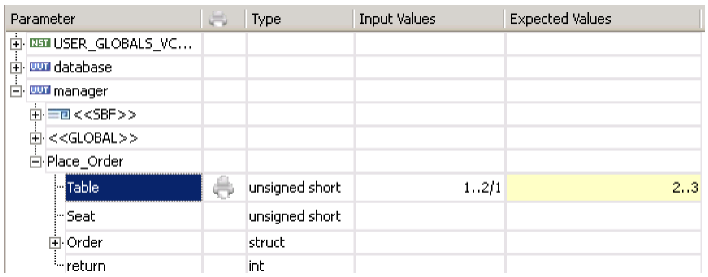
The Execution Status viewer automatically opens when a test case is executed. The Execution Status viewer provides detailed real time test case execution information. See the "Execution Status Viewer" on page 58 for more information.

If a test case contains expected values, those values are compared to the actual values when a test is executed and the Test Case Tree is updated with a PASS or FAIL indication.

```
clicast -e <env> -u <unit> -s <sub> -t <testcase> Execute Run

Execute the specified test case.
```





To Automatically Save Test Cases Before Execution

Choose **Tools => Options**, and click the **GUI** tab and the **Test Case Editor Options** sub-tab.

The **Automatically Save Test Cases Before Execution** option saves the current test case prior to execution. If this option is not selected, you are prompted to save a modified test case when **Execute** is chosen. By default, modified test cases are automatically saved before executing.

When you have made your selection, click **Apply** or **OK** to close the dialog.

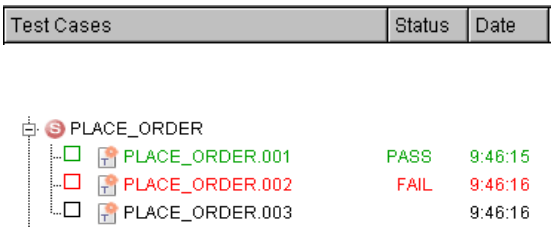
This information is saved to a file named `.vcast-qt` in the user's `HOME`directory.

To Select the Next Passing or Failing Test Case

The red and green arrows on the tool bar allow you to easily find passed and failed test cases on the Test Case Tree. When an arrow is clicked, the Test Case Tree is traversed in the direction of the arrow, and the node containing the next passed or failed test case is automatically opened.

Test Case Status in the Test Case Tree

The test case name appears in the first column. If all expected values match then the test case passes, and the test case name is displayed in green with the word "PASS" in the Status column. If any expected values do not match, then the test case fails, the test case name is displayed in red, with the word "FAIL" in the Status column.



The time of test execution appears in the Date column. If you open an environment with a test case that you executed a day or more ago, then the date, rather than the time, is displayed in the Date column.

Coverage Checkbox in the Test Case Tree

If the environment has code coverage enabled, then a checkbox appears next to the test case name after it is executed.



Clicking this checkbox opens the Coverage Viewer and displays the unit's coverage achieved by the selected test cases.

See also "Using Code Coverage" on page 477 for more details on code coverage.

Viewing Execution Results

By default, the test execution results are displayed in a tab on the test case editor window. To view the execution results report, you can either:

- Execute a test case (the Execution Report tab opens automatically)
- Open a test case editor and click the Execution Report tab, or
- Select a test in the Test Case Tree and choose **Test => View => Execution Results**.

The Execution Report tab is automatically opened after a single test case is executed. If you execute multiple test cases, the Test Case Management report is displayed.

Execution Results Report

Configuration Data

Environment Name	TUTORIAL_CPP
Date of Report Creation	23 JAN 2019
Time of Report Creation	3:04:20 PM

(CL)MANAGER::PLACEORDER.001

Test Case Configuration

Unit Under Test	manager
Subprogram	(cl)Manager::PlaceOrder
Test Case Name	(CL)MANAGER::PLACEORDER.001
Date of Creation	23 JAN 2019 1:55:18 PM
Date of Execution	23 JAN 2019 3:00:57 PM

Execution Results (PASS)

Start of (CL)MANAGER::PLACEORDER.001

Event 1 - Called (cl)Manager::PlaceOrder
UUT: manager.cpp
Subprogram: (cl)Manager::PlaceOrder

Parameter	Type	Actual Value	Expected Value
Table	int	2	
Seat	int	0	
Order			
Soup	enum	Onion	
Salad	enum	Caesar	
Entree	enum	Steak	
Beverage	enum	MixedDrink	

Event 2 - Stubbed DataBase::GetTableRecord
UUT: database.h
Subprogram: DataBase::GetTableRecord

Parameter	Type	Actual Value	Expected Value
Data			
[0]			
NumberInParty	int	0	
CheckTotal	int	0	

Event 3 - Stubbed DataBase::UpdateTableRecord
UUT: database.h
Subprogram: DataBase::UpdateTableRecord

Parameter	Type	Actual Value	Expected Value
Data			
[0]			
IsOccupied	bool	true	<match>
NumberInParty	int	1	<match>
Order			

If you turn on the “Use external browser” option, located on the GUI tab of the **Tools => Options dialog**, and specify an external browser (IE, Netscape, FireFox, etc.), then all reports, including the Execution Results, are displayed in a separate browser window. If you would like the Execution Results to be displayed in a report tab, you can turn on the “Use external browser” option but leave the browser unspecified.

```
clicast -e <env> [-u <unit> [-s <sub> [-t <testcase>]]]
Reports Custom ACtual [<outputfile>]

Extract multiple execution results to standard output or the specified
output file. If you have recently changed the VCAST_CUSTOM_REPORT_FORMAT
then you should execute the test cases before using this command.
```

Browsing the Execution Report

When you are viewing an Execution Report, four toolbar buttons are available to easily find the next match or unmatched data item in an Execution Report, Coverage Viewer, Parameter Tree, or Compound Tree.

 Previous matched item (green)

 Next matched item (green)

 Previous unmatched item (red)

 Next unmatched item (red)

Understanding the Execution Report

The Execution Results include a list of calls made by the VectorCAST driver into the UUT and calls made from the UUT into a stubbed dependent subprogram. In both cases, the parameters and data associated with the calls are displayed as well as any global data values that are being monitored. Each transfer of control is an *event*. Event 1 is always the test harness calling the subprogram under test. Events between the first and last events are calls to stubbed functions (if any). The last event is always the return to the test harness.

The maximum number of events is set at 1000 by default, but can be changed for an individual test case or for all test cases.

Expected Values

If a parameter has an input value or expected value, it is included in the Execution Report. If a parameter’s value at the time of the call matches the Expected Value or falls within an expected range, then the Expected Values column is colored green for that parameter. If it doesn’t match or fall within an expected range, then the column is colored red. If all parameters with Expected Values matched, then the test case status is set to PASS. If any expected values do not match, the test case status is set to FAIL.

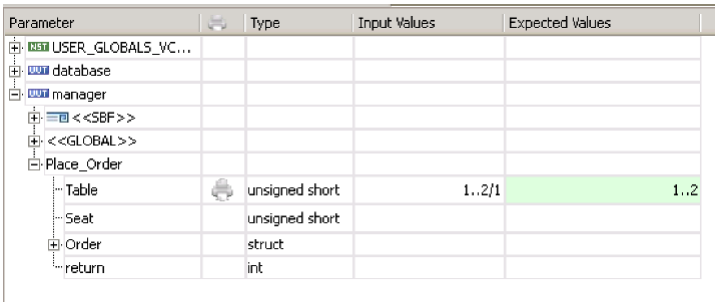
A section called “User Code Expected Values” is included at the end of the Execution report when User Code contains an expected value. The expected value for the item is compared to the actual value and if they match, a line is added to the section and colored green. The item is displayed in the left column of the report, and the status, <<match>> or [fail] is displayed in the right column. If the item being compared is lengthy, then the display is truncated to the page width. In HTML, this width is 80 characters. In TEXT format, this width is configurable, using the “Page width” option on the Reports tab, Format tab, Text sub-tab.

To cause a parameter to be included in the report without having to set an Input Value or a specific Expected Value, enter <<ANY>> as the Expected Value. Doing so causes the parameter to match any value, and also to be included in the report.

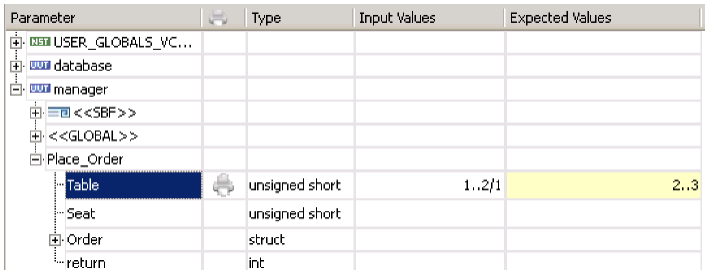
If a test case has no Expected Values, then there are no values to match, and so the status is neither PASS nor FAIL.

In the Parameter Tree, the expected values are also colored to indicate PASS or FAIL. If an expected values is provided and it matches at some events in the Execution Report but fails in others, then it is colored yellow in the Test Case Editor.

For example, suppose a parameter has a range of input values 1..2, which causes two iterations of the test case. If the expected value for this parameter is set to the range 1..2, then both Input Values match.



If we change the Expected Value for this parameter to the range 2..3, then the first Input Value (1) does not match, but the second (2) does. This situation results in the expected value cell being colored yellow. (if a test case partially passes and it is part of a compound, then the slot is considered failed. When there are multiple iterations of a slot and some pass in a compound then that slot partially passes.)



Unused Expected Results

If expected results are provided for parameters that are not encountered during test execution, the unused expected results are appended to the Execution Results report and the test case status is set to FAIL. For example, if a list of three expected values is provided for a parameter of a stubbed subprogram, but the subprogram is only called twice (thus providing only two input values to compare), then the third expected value is unused.

For compound test cases, the unused expected results are grouped by slot.

Controlling Report Information

VectorCAST enables you to control the amount of data included in the test result reports. The **Tools => Options dialog**, **Report tab**, **Content sub-tab** contains several options to control the amount of information in the Execution Report.

By default, all parameters, both input and expected, that have values specified in the Parameter Tree, are included on the Test Execution Report.

Optionally, Input Values populated within the Parameter Tree can be hidden on the Test Execution report. To hide the Input Value, right-click the  icon next to the parameter on the Parameter Tree and select **Hide from Report** on the context menu. This option is not available if the Expected Values column contains data.



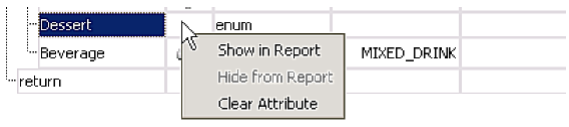
.

The  icon is then displayed indicating that the Input Value is now hidden on the report.

To re-enable the Input Value on the report, right-click the icon and select **Show in Report**. Selecting the **Clear Attribute** allows the value to observe the attribute state for its parent.

For Expected Results with a specified value, the icon appears on the Parameter Tree to indicate that the value will be printed on the Test Execution report, but it is not user controllable and is always included in the report.

For both Input Values and Expected Results parameters that have no values specified, right clicking in the report control column will display the following context menu.



Select **Show in Report** and the **force print icon**  appears indicating that the parameter is currently included in the report.

To remove the parameter from the report, right-click the **force print icon**. Choose **Hide from Report**, to suppress printing, and the  icon is displayed. Selecting **Clear Attribute** at any time will suppress printing and remove the icon from the Parameter Tree.

In addition, the attribute items are available on composite parameters as well. When the user selects a composite object (e.g. "struct" object) and sets the attribute, all members (recursively) will receive the selected attribute unless directly overridden (with the added limitation that a parameter with an expected value will not inherit the "Hide" attribute).

VectorCAST captures output for variables as follows:

- If you set an input or an expected value for a parameter, VectorCAST will capture the output. This includes the use of the expected value **<<ANY>>**.
- If you set an input value, but you then use the printer option  and select **Hide in Report**, VectorCAST will not capture the output. (Note that VectorCAST needs to capture the output in order to perform the test for the expected value. This scenario sets up a contradiction by hiding the variable and preventing capture.)
- If you do not set an input or an expected value, then you can use the printer option  **Show in Report** and VectorCAST will capture the output.

Output from the Unit Under Test

In some cases, the code under test may generate output to either stdout or stderr. VectorCAST can capture this output and include it in the Execution Report.

See "Redirect Standard Error".

Other Execution and Report Options

There are many other options that control test case execution, report content, and report format.

Cover Report CSV Metrics Delimiter

The character separator for the Cover report created by the **clicast cover report csv_metrics** command. Enter the character in quotes, as in “@”. To enter a tab, use “\t”. Valid delimiters are: ? , ' ; | { } [ ] @ ~ # $ _ \t \n

```
clicast -lc option VCAST_RPTS_DELIMITER <delimiter>

Character separator used in clicast cover report csv_metrics. The default
value is comma ",".
```

Wrap Text at Column Width

The Wrap Text at Column Width option specifies that when text is too long to fit in a table cell, it is wrapped to the next line.

```
clicast -lc option VCAST_WRAP_TEXT_REPORT TRUE | FALSE

Wrap the text in table cells if it is too wide. Its default value is True.
```

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 Reportsin either TEXT or HTML format.

```
clicast -lc Option VCAST_RPTS_WRAP_NOTES True | False

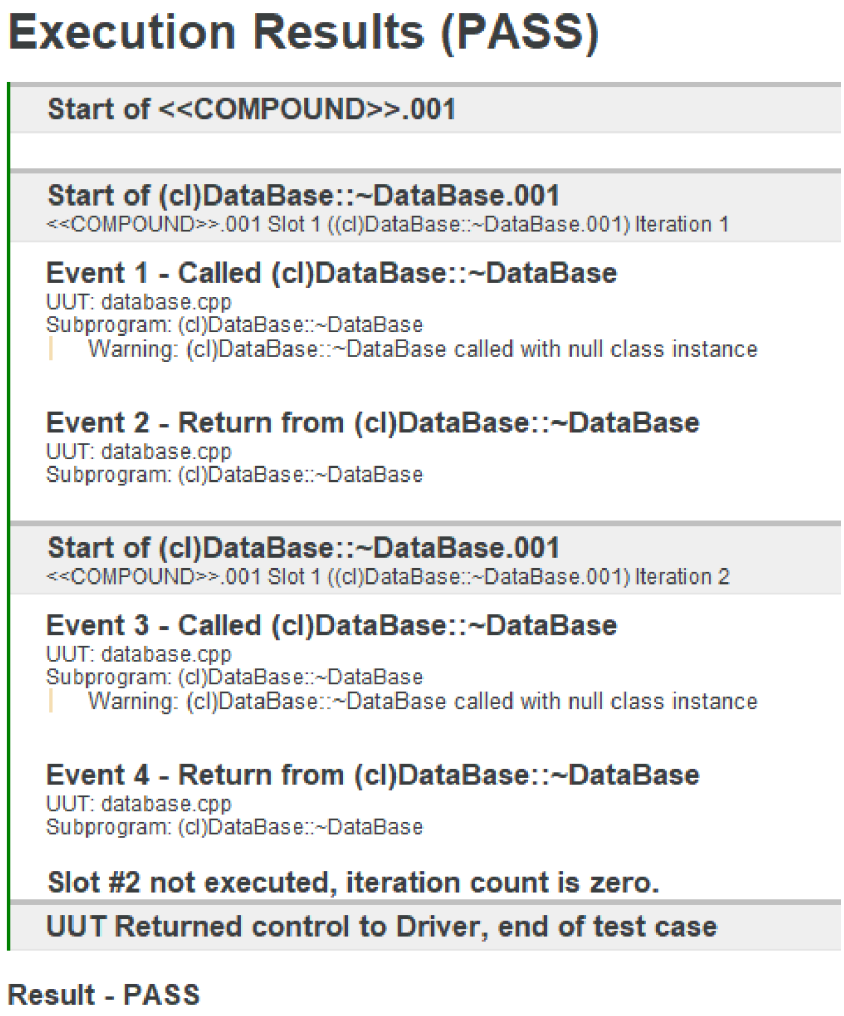
This option will cause the text in the notes section to wrap at the first
space before 80 characters. The default value is FALSE.
```

Compound Test Case Execution and Reports

The test cases in a compound test are executed in order, from top to bottom. Data is persistent between test cases in a compound test case, unlike data between simple test cases that are executed together. An <<INIT>> test case in the first slot can be used to initialize variables or instantiate class pointers that subsequent test cases access.

By default, each slot in a compound is executed once, unless the number of iterations is specified. If the cumulative number of events exceeds the environment-wide Event Limit (specified on the **Tools => Options dialog**, **Execute tab**), compound test case execution terminates, with a message in the Execution Report. For test cases that have their own Event Limit specified on their Options tab, then that Event Limit is used when comparing the cumulative number of events. Following that, the environment-wide Event Limit is used.

The Execution Report for a compound test case is very similar to that of a simple test case. The only difference is in the Event header for an Event that marks the start of a new slot. The first slot is executed starting with Event 1, and the second slot's start is indicated as "Event *n*, Slot 2, Iteration 1." If you have specified that a slot iterate, then the Execution Report reflects the slot being called multiple times.

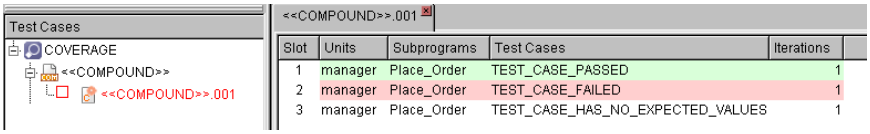


See " To Specify the Number of Iterations of a Test in a Compound" on page 262 for information on specifying slot iterations.

The Compound Tree

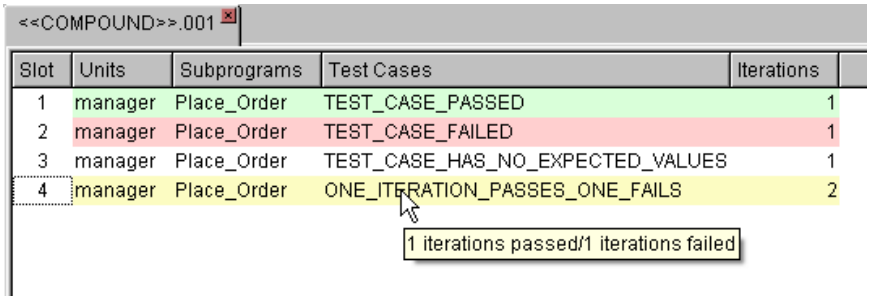
As with the Parameter Tree for simple test cases, the Compound Tree tab reflects the PASS or FAIL status of the compound test case. If all slots in the compound passed or had no expected values, then the compound as a whole passes. If any slot in the compound fails, then the compound as a whole fails.

For example, suppose we have a compound test case with three slots: a test case that passes, a test case that fails, and a test case that has no expected values, so it neither passes nor fails. When this compound is executed, it is given the status of FAIL as a whole, due to slot 2 failing.



The slots in the Compound Test Case are colored to reflect the pass or fail status of each individual slot. A slot that passes is colored green; a slot that fails is colored red, and a slot that has no expected values is left uncolored.

If a slot has more than one iteration specified, then all iterations of that test case must pass in order for that slot to be considered PASSed. In the example below, a test case was added to the compound. One of the parameters in the test case has an Expected List of 2 values, but only 1 input value. The first input will pass, but the second expected value won't match. In the compound, the number of iterations is set to 2. When the compound is executed, this 4th slot is colored yellow, because the first iteration passed, but the second iteration failed, as shown in the tool-tip.



To Execute Multiple Test Cases

There are several ways to execute multiple test cases. The **Test => Batch Execute All** command enables you to execute all test cases. All simple and compound test cases are executed in order from top to bottom of the Test Case Tree. There is no interaction of data between the test cases, and the Event Limit starts over with each test case. When you execute multiple test cases, the Test Case Management report appears.

Alternatively, you can right-click the top node of the Test Case Tree (the environment name) and choose **Execute**, which also executes all test cases. If you click a node other than the top node of the Test Case Tree, then only the test cases under that node are executed.

If a test case is marked "Compound Only" it is only executed when its compound test is executed.

The keyboard shortcut for Batch Execute All is the **F6 key**.

```
clicast -e <env> [-u <unit> [-s <sub>]] EXecute Batch

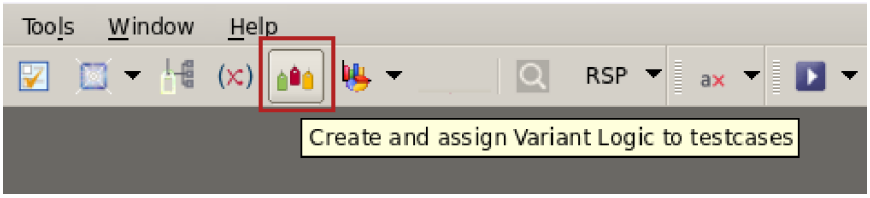
Execute multiple test cases. To run all test cases for a subprogram,
specify the subprogram name on the command. To run all tests for an
environment, do not specify the subprogram name.
```

Control Test Case Execution Using Variant Logic

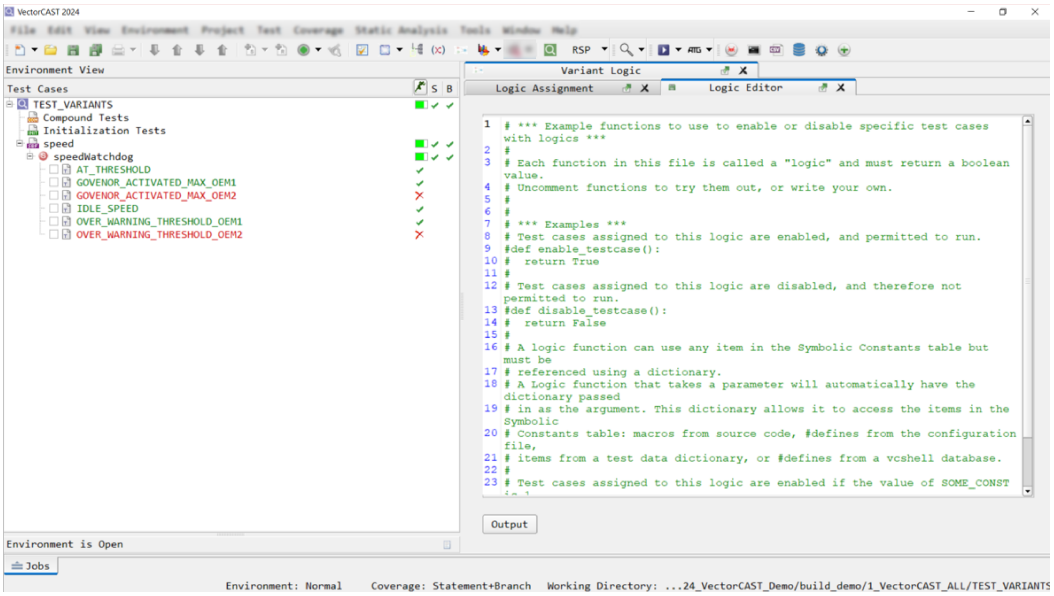
If a test environment includes tests for multiple code variants, a Test Variant Logic file can be used to

select which tests should be run for this test configuration. A primary use case would be to control test case execution for target environments.

A Logic file can be added to a test environment by selecting the Variant Logic button  located in the VectorCAST toolbar.



Once the button is selected, the Logic Editor will appear with a sample Logics file.



The Logics file is Python-based. Python is used to implement the logic that will determine which test case should be enabled or disabled in the environment. Examples are given in the default file.

Edit this file in the Logic Editor to define your Logics. A very simple example is shown below.

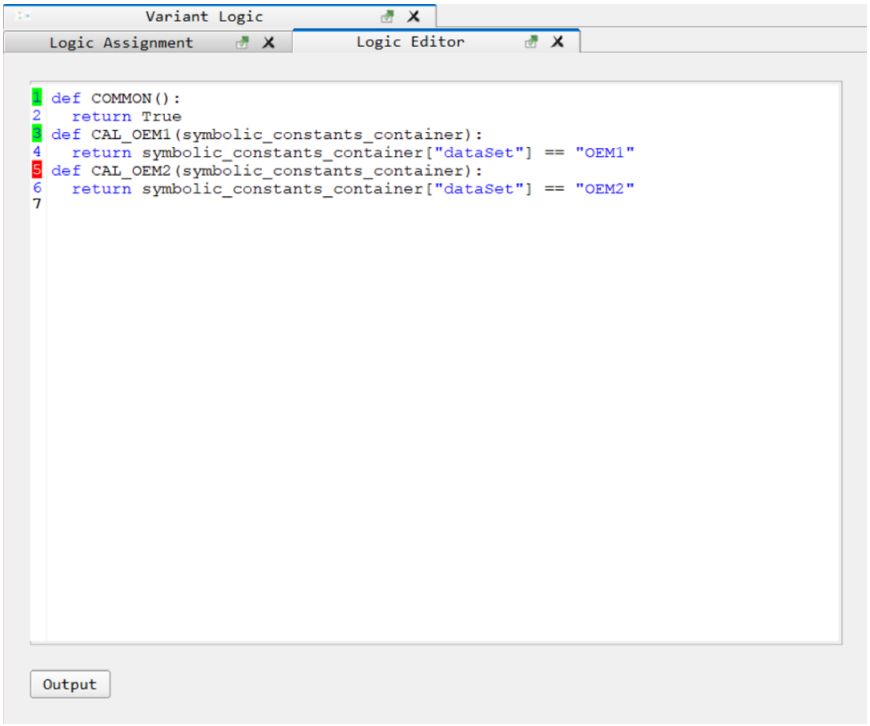
```
#
# Test cases assigned to this logic are enabled, and permitted to run.
def enable_testcase():
  return True

#Test cases assigned to this logic are disabled, and therefore not
#permitted to run.
def disable_testcase():
  return False
```

For this example, the following Logics file will be used.

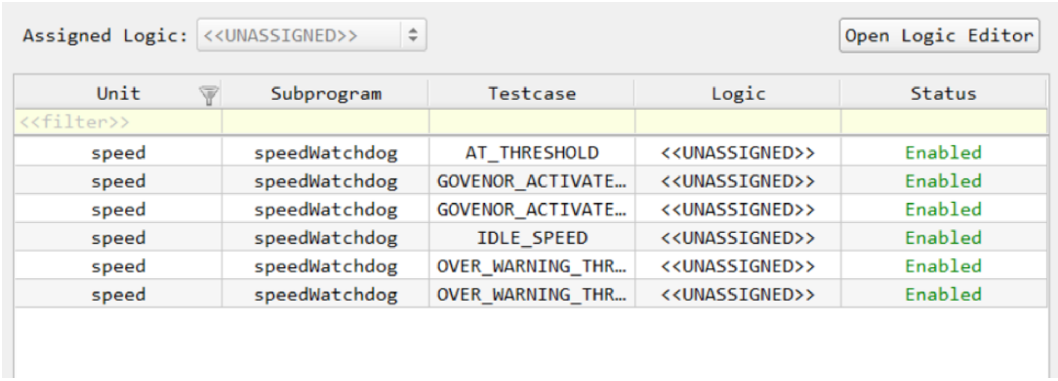
```
def COMMON():
  return True
def CAL_OEM1(symbolic_constants_container):
  return symbolic_constants_container["dataSet"] == "OEM1"
def CAL_OEM2(symbolic_constants_container):
  return symbolic_constants_container["dataSet"] == "OEM2"
```

After the edits are completed, save the file. The Logic Editor will look like this:

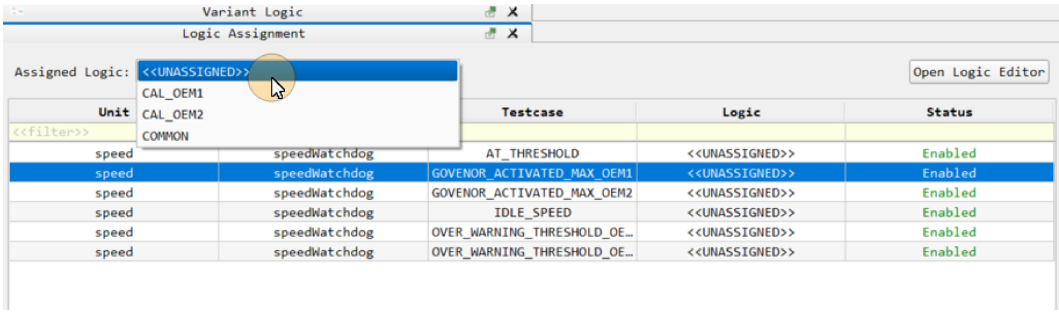


Save As functionality is incorporated into the editor tab. In order to move or rename the logics file, use the Save As option from the File menu.

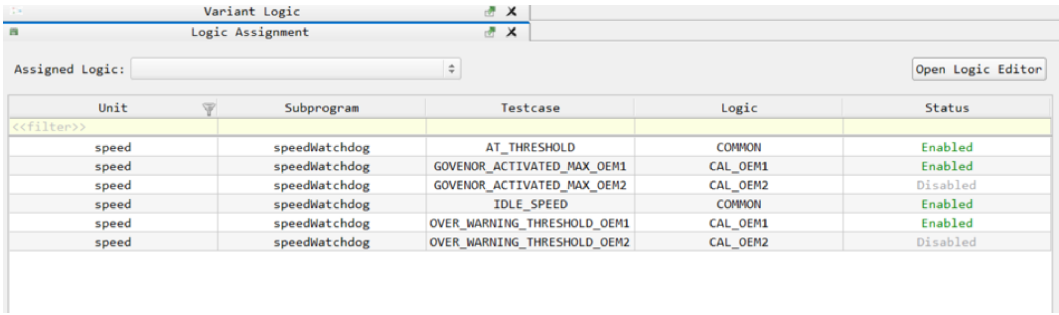
Close the tab and the Logic Assignment tab will appear. Initially none of the Logics will be assigned as shown. Any logics that evaluate to True return a status of Enabled and will have a green highlight on their line number. Any logics that evaluate to False will return a status of Disabled and have a red highlight on their line number. As soon as the Logic Editor is modified all evaluation markings are cleared from the line numbers until the logic file is evaluated again through either saving or reloading the file.



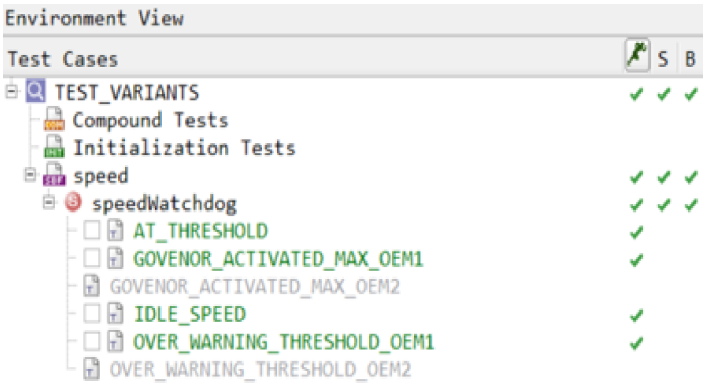
Select a test case and then select the drop-down menu next to the **Assigned Logic:** field to assign a Logic to a test case.



Once all the Logics are assigned, the display will look like this:

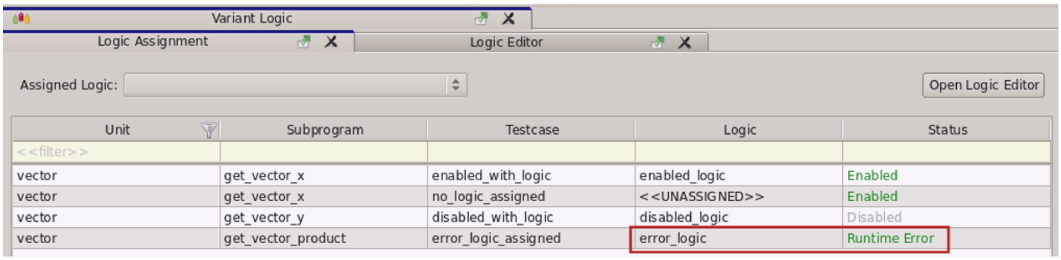


Save the Assignments (hint: close the Tab and it will ask if you want to save). The Test Cases will now show in grey if they are Disabled.

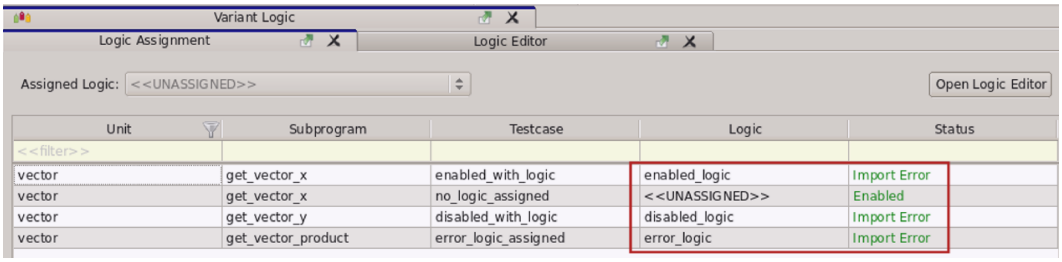


When there is an error during evaluation of a logics file, two types of errors can be present:

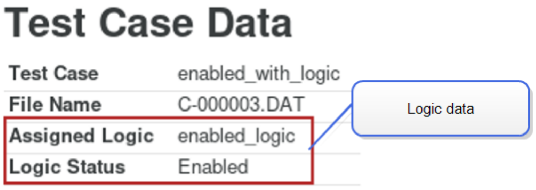
1. The error is contained to a single logic which results in that logic having a status of Runtime Error and any associated test cases are now Enabled.



2. The error affects the evaluation of the entire file and results in all logics having a status of Import Error and all test cases are now Enabled.



In reporting, logic is treated as another field of test case data and is displayed with each test case that has logic assigned, along with the status of the logic in the Test Case Data Report and Full Report. Below is an example of this information displayed in the context of the Test Case Data Report:

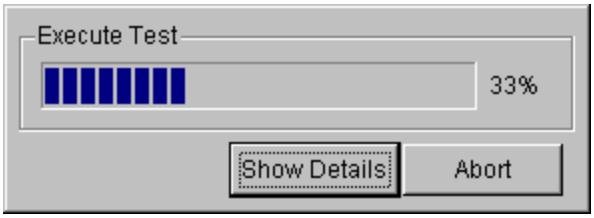


Logic File Basics

- Importing modules is not allowed in the logics file. The keywords "**`exec`**", "**`import`**", and "**`eval`**" are prohibited from use in the logics file.
- Adding a hidden helper function. In order to add a helper function that won't be detected as a logic, prefix the function name with an underscore "_" and it will not appear in the list of available logics for assignment.
- The module **`logic_utils`** allows access to the **`getenv()`** function which allows the use of environment variables in the logics as well.

To Abort Test Case Execution

When executing a test case that exceeds two or three seconds in duration or when executing multiple test cases, the following dialog appears. Click the **Abort** button to halt the test case execution and return control to VectorCAST. This is useful for test cases that are running indefinitely, not responding, or when target communication is suspect.



To Execute a Test Case in the Debugger

It is often desirable to execute test cases under control of the underlying compiler's debugger. To do this, select the test case name, right-click, and choose **Execute with Debug**. This results in the debugger being launched with the VectorCAST test harness code loaded. (The VectorCAST window is not useable at this time.) From the debug window you can perform all debug commands as you would normally. Upon quitting from the debugger, control is passed back to VectorCAST. The VectorCAST reports generated will reflect the execution history from the debug session.

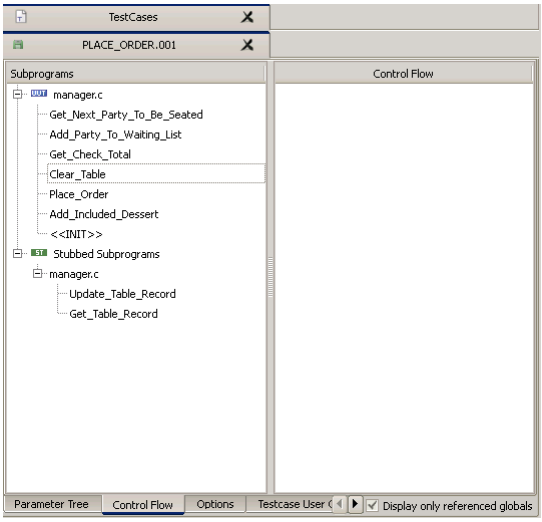
```
clicast -e <env> -u <unit> -s <sub> -t <testcase> EXecute Debug

Execute the specified test case under control of the debugger.
```

Working with a Control Flow

To view or edit a test case control flow, click the **Control Flow** tab in the Test Case Editor. The left hand side of the Control Flow screen, the *Subprograms* panel, contains a list of UUT subprograms, non-stubbed units as well as stubbed units. <<INIT>> is listed once, under the first unit.

To expand all nodes of the tree, right-click anywhere in the *Subprograms* panel and select **Expand All** from the context menu. To collapse the tree, select **Collapse All** from the context menu.

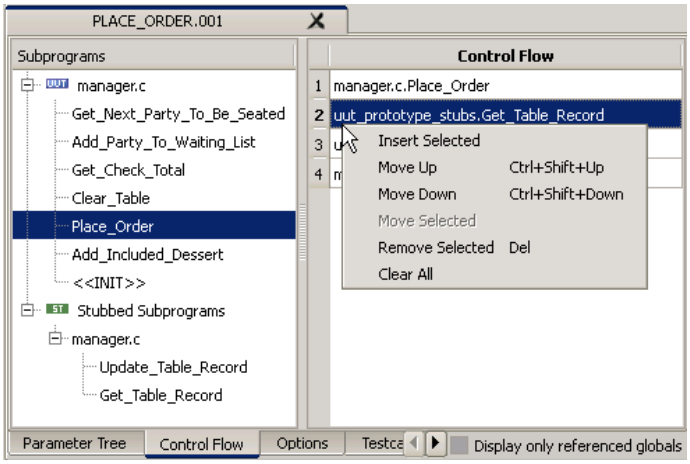


If the *Control Flow* panel is empty, a control flow was not created or saved for the test case. See "To Save the Control Flow" on page 409.

You can manually add a subprogram to the control flow by double-clicking any subprogram listed in the Subprogram panel. The item will be added to the bottom of the list in the Control Flow panel

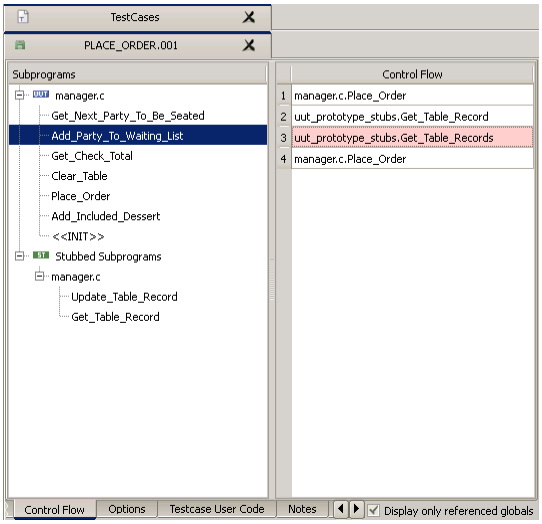
You can edit the control flow by right-clicking on any item in the Control Flow panel. Using the context menu you can:

- **Insert Selected** – this menu item will duplicate the selected subprogram and place a copy directly below the selected subprogram in the Control Flow panel.
- **Move Up** – the selected subprogram will move up one position in the control flow. This selection choice is inactive if the subprogram selected is at the top of the list. (**Shortcut: Ctrl+Shift+Up**)
- **Move Down** - the selected subprogram will move down one position in the control flow. This selection choice is inactive if the subprogram selected is at the bottom of the list. (**Shortcut: Ctrl+Shift+Down**)
- **Move Selected** – this menu item allows you to select and move one or more subprograms to a different position in the Control Flow. To activate this feature, first, select the subprograms you wish to move by CTRL-clicking each subprogram. Next, select the “move to” position by CTRL+right-clicking the new position. When the context menu appears, select **Move Selected.** The subprograms will be moved just above the CTRL+right-click position.
- **Remove Selected** – delete the selected subprogram or subprograms from the control flow. (**Shortcut: Del**)
- **Clear All** – delete all subprograms in the control flow. See also, “To Clear the Control Flow”.



Stubbed subprograms listed in the Control Flow panel are identified with the prefix identifier “uut_ prototype_stubs.” for easy recognition.

By double-clicking a subprogram in the control flow list, you can edit the name of the subprogram. If you accidentally enter a subprogram name that is either misspelled or does not exist, the entry will have a light red background to alert you to the error.



Control flow can change as a result of new source code being introduced into the test environment or as a result of manually editing the control flow.

If a control flow has changed, the Test Execution Report identifies the event where the control flow mismatch occurred as well as providing a list missing control flow events.

Result - FAIL

Start of PLACE_ORDER.001

Event 1 - Called Place_Order

UUT: manager.c
Subprogram: Place_Order

Parameter	Type	Actual Value	Expected Value
Table	unsigned short	2	
Seat	unsigned short	0	
Order			
Soup	enum	ONION	
Salad	enum	CAESAR	
Entree	enum	STEAK	
Beverage	enum	MIXED_DRINK	

Control Flows

<match>

Event 2 - Stubbed Get_Table_Record

UUT: manager.c
Subprogram: Get_Table_Record

Parameter	Type	Actual Value	Expected Value
return			
Number_In_Party	unsigned short	0	
Check_Total	float	0	

Control Flows

[uut_prototype_stubs.Update_Table_Record]

Event 3 - Stubbed Update_Table_Record

UUT: manager.c
Subprogram: Update_Table_Record

Parameter	Type	Actual Value	Expected Value
Data			
Is_Occupied	enum	v_true	<match>
Number_In_Party	unsigned short	1	<match>
Order			
[0]			
Dessert	enum	PIE	<match>
Check_Total	float	14	<match>

Control Flows

[uut_prototype_stubs.Get_Table_Record]

Event 4 - Return from Place_Order

UUT: manager.c
Subprogram: Place_Order

Parameter	Type	Actual Value	Expected Value
Table	unsigned short	2	
Seat	unsigned short	0	
Order			
Soup	enum	ONION	
Salad	enum	CAESAR	
Entree	enum	STEAK	
Beverage	enum	MIXED_DRINK	

Control Flows

[(extra event)]

To Save the Control Flow

The Save Control Flow command enables you to save the execution flow of a particular test run for comparison against future runs. To use this command you first create and execute a test case. If, after viewing the results, you wish to save the order of the calls as an expected result, select **Test => Control Flow => Save**. Future runs of the same test case result in a comparison of the call sequence to the saved call sequence. Using this command results in an additional comparison of expected and actual results for pass/fail metrics.

```
clicast -e <env> -u <unit> -s <sub> -t <testcase> EXecute Save Flow

Save the control flow for the specified test case.
```

To Clear the Control Flow

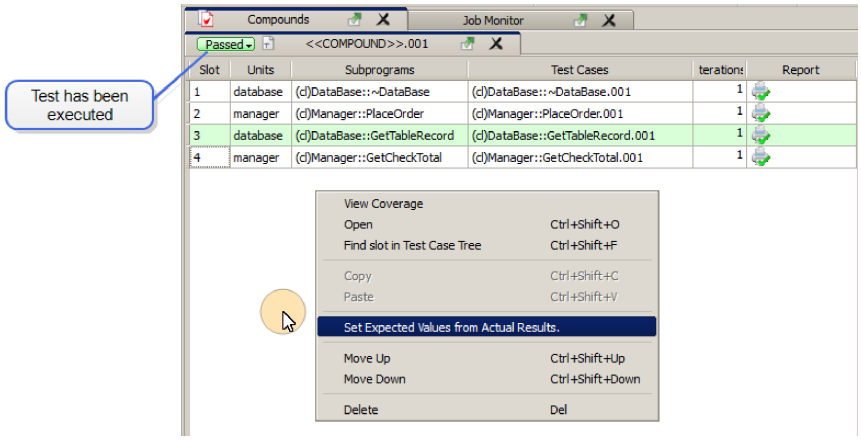
If you no longer wish to use the saved control flow for a particular test case, simply select the test case and select **Test => Control Flow => Clear**. Future runs of the same test case will no longer compare the call sequence with the saved call sequence.

```
clicast -e <env> -u <unit> -s <sub> -t <testcase> EXecute Clear Flow

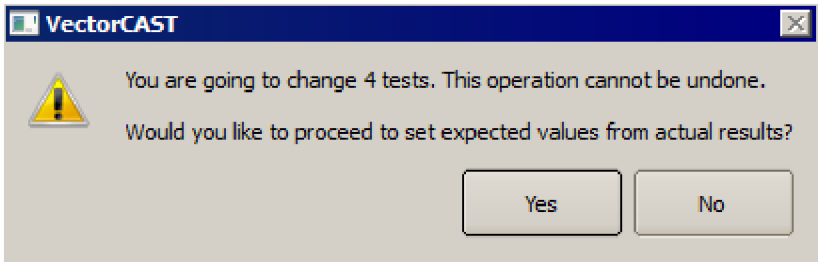
Clear the control flow for the specified test case.
```

To Set Expected Values to Actual Values

After executing a test case, Expected Values can be set to the actual execution result values. Right-click anywhere in the Compound Test Editor and select **Set Expected Values from Actual Results** from the context menu.



A warning dialog appears indicating the number of test cases that will be changed by the action. You have the opportunity to abort the action by selecting the **No** button. Selecting the **Yes** button will allow the operation to proceed and modify the test cases. Note that this action cannot be undone.



All Expected Values listed in the Execution Report are set to their actual values.

See Controlling Report Information for more information on capturing output for variables.

```
clicast -l<lang> -e <env> [-u <unit> [-s <sub> [-t <testcase>]]] TESt
Actuals_to_expected

This will set the actual values for a test case as the expected values.
Thus 100% of the expected results will match for that test case.
```

Viewing Test Reports

The **Test => View** commands reflect the item selected in the Test Case Tree. For example, if a subprogram is selected in the Test Case Tree, then the report generated using the **Test => View** menu reflects the data from the test cases for that subprogram.

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 **Test => 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 user's HOME directory, and thus applies to all unit test environments.



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

View Reports in an External Browser

If you prefer to view HTML reports in an external browser (IE or Mozilla, for example), then you can specify an external browser in the **Tools => Options** dialog, **GUI** tab, under the subtab, **External Browser Options**. There you can add the command to start the browser. To use the external browser, select "**Use external browser**".

If you switch your Report Format from the default HTML to TEXT, then you can view the test reports in an external text editor. If you specified an external editor (notepad or emacs, for example) and indicated that it should be used as the File Viewer in the **Tools => Options dialog, GUI tab**, then the test reports are displayed in that external editor.

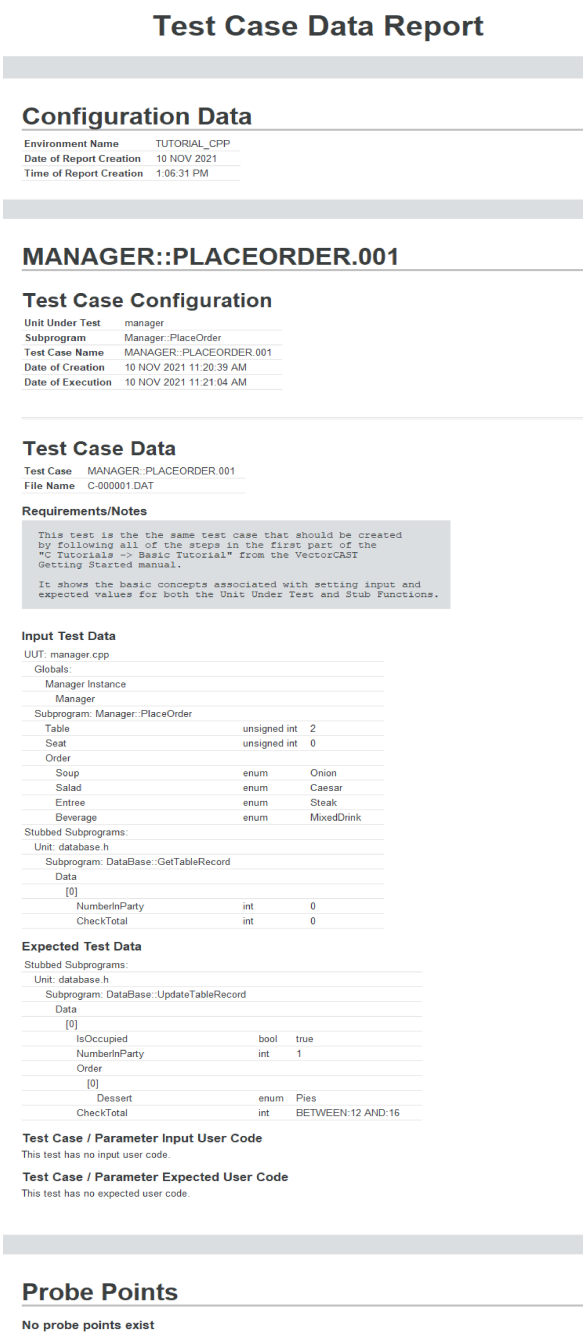
View a Test Case Data Report

The **Test => View => Test Case Data** command enables you to view the contents of a test case, once it has been saved. The contents of this report reflects the item(s) selected in the Test Case Tree.

```
clicast -e <env> [-u <unit> [-s <sub> [-t <test>]]]
Reports Custom Test <outputfile>

Create a Test Case Data report and output to HTML file <env>_test_case_
data_report.html, standard output as text, or to the specified output file.
If the Unit parameter is specified, the report includes the Test Case Data
for all test cases for the specified unit. If the Unit and Subprogram
parameters are specified, then the report includes the Test Case Data for
only those test cases for that subprogram of the specified unit. If the
Unit, Subprogram, and Test Case parameters are specified, then the report
includes the Test Case Data for only the specified test case. Otherwise,
the report includes the execution results for all test cases in the
environment, including <<COMPOUND>> and <<INIT>>.
```

The following report is the result of the **Test => View => Test Case Data** command for a simple test case.



Test Case Data Report

Configuration Data

Environment Name	TUTORIAL_CPP
Date of Report Creation	10 NOV 2021
Time of Report Creation	1:06:31 PM

MANAGER::PLACEORDER.001

Test Case Configuration

Unit Under Test	manager
Subprogram	Manager::PlaceOrder
Test Case Name	MANAGER::PLACEORDER.001
Date of Creation	10 NOV 2021 11:20:39 AM
Date of Execution	10 NOV 2021 11:21:04 AM

Test Case Data

Test Case	MANAGER::PLACEORDER.001
File Name	C-000001.DAT

Requirements/Notes

```
This test is the the same test case that should be created
by following all of the steps in the first part of the
"C Tutorials -> Basic Tutorial" from the VectorCAST
Getting Started manual.

It shows the basic concepts associated with setting input and
expected values for both the Unit Under Test and Stub Functions.
```

Input Test Data

UUT: manager.cpp		
Globals:		
Manager Instance		
Manager		
Subprogram: Manager::PlaceOrder		
Table	unsigned int	2
Seat	unsigned int	0
Order		
Soup	enum	Onion
Salad	enum	Caesar
Entree	enum	Steak
Beverage	enum	MixedDrink
Stubbed Subprograms:		
Unit: database.h		
Subprogram: DataBase::GetTableRecord		
Data		
[0]		
NumberInParty	int	0
CheckTotal	int	0

Expected Test Data

Stubbed Subprograms:		
Unit: database.h		
Subprogram: DataBase::UpdateTableRecord		
Data		
[0]		
IsOccupied	bool	true
NumberInParty	int	1
Order		
[0]		
Dessert	enum	Pies
CheckTotal	int	BETWEEN:12 AND:16

Test Case / Parameter Input User Code

This test has no input user code.

Test Case / Parameter Expected User Code

This test has no expected user code.

Probe Points

No probe points exist

View Test Execution Results

The **Test => View => Execution Results** command enables you to view the execution results for a test case, a subprogram, or environment, depending on where you clicked before choosing the menu item. The report is the same as the one that appears immediately after executing that test case. If the menu item is dimmed, this means the test case has not been executed.

The contents of this report reflects the items selected in the Test Case Tree. If you click on a subprogram name or a unit name, then the Execution Report contains results for all test cases for that

subprogram or unit. If you click on the name of the environment, then the report contains results for all compound test cases, simple test cases, and compound-only test cases.

Similarly, if you click on <<COMPOUND>> or <<INIT>>, then the report contains results for all Compound test cases or Init test cases, respectively.

The following report is the result of the **Test => View => Test Case Data** command for a simple test case.

Execution Results Report

Configuration Data

Environment Name	TUTORIAL_CPP
Date of Report Creation	10 NOV 2021
Time of Report Creation	1:34:48 PM

MANAGER::PLACEORDER.001

Test Case Configuration

Unit Under Test	manager
Subprogram	Manager::PlaceOrder
Test Case Name	MANAGER::PLACEORDER.001
Date of Creation	10 NOV 2021 11:20:39 AM
Date of Execution	10 NOV 2021 11:21:04 AM

Execution Results (PASS)

Start of MANAGER::PLACEORDER.001

Event 1 - Calling Manager::PlaceOrder
UUT: manager.cpp
Subprogram: Manager::PlaceOrder

Parameter	Type	Actual Value	Expected Value
Table	unsigned int	2	
Seat	unsigned int	0	
Order			
Soup	enum	Onion	
Salad	enum	Caesar	
Entree	enum	Steak	
Beverage	enum	MixedDrink	

Event 2 - Stubbed DataBase::GetTableRecord
UUT: database.h
Subprogram: DataBase::GetTableRecord

Parameter	Type	Actual Value	Expected Value
Data			
[0]			
NumberInParty	int	0	
CheckTotal	int	0	

Event 3 - Stubbed DataBase::UpdateTableRecord
UUT: database.h
Subprogram: DataBase::UpdateTableRecord

Parameter	Type	Actual Value	Expected Value
Data			
[0]			
IsOccupied	bool	true	<match>
NumberInParty	int	1	<match>
Order			
[0]			
Dessert	enum	Pies	<match>
CheckTotal	int	14	<match>

Event 4 - Returned from Manager::PlaceOrder
UUT: manager.cpp
Subprogram: Manager::PlaceOrder

Parameter	Type	Actual Value	Expected Value
Table	unsigned int	2	
Seat	unsigned int	0	
Order			
Soup	enum	Onion	
Salad	enum	Caesar	
Entree	enum	Steak	
Beverage	enum	MixedDrink	

Global Data	Type	Actual Value	Expected Value
UNIT: manager.cpp			
Globals:			
Manager Instance			
Manager			

UUT Returned control to Driver, end of test case

Result - PASS

Expected Results matched 100% 4/4 PASS

```
clicast -e <env> [-u <unit> [-s <sub> [-t <test>]]]
Reports Custom ACtual <outputfile>

Create an Execution Results report and output to HTML file <env>_execution_
results_report.html, standard output as text, or to the specified output
file. If the Unit parameter is specified, the report includes the execution
results for all test cases for the specified unit. If the Unit and
Subprogram parameters are specified, then the report includes the execution
results for only those test cases for that subprogram of the specified
unit. If the Unit, Subprogram, and Test Case parameters are specified, then
the report includes the execution results for only the specified test case.
Otherwise, the report includes the execution results for all test cases in
the environment, including <<COMPOUND>> and <<INIT>>.
```

See also "To Execute a Test Case and View Results" on page 390 for more information on Execution Reports.

The Full Report

The **Test => View => Full Report** command generates a single report that includes the following sections:

- Table of Contents
- Configuration Data
- Overall Results
- Configure Stubs User Code
- Environment User Code
- Test Case Configuration
- Test Case Data Report
- Execution Results Report
- Aggregate Coverage Report
- Metrics Report
- Probe Points

The contents of this report reflects the items selected in the Test Case Tree. Much of the full report is the same for all test cases. The test case data and execution results are specific to the test case(s) included in the report.

```
clicast -e <env> [-u <unit> [-s <sub> [-t <testcase>]]]
Reports Custom Full <outputfile>

Create a Full report and output to HTML file <env>_full_report.html,
standard output as text, or to the specified output file. The Full report
includes the Aggregate Coverage data for all units or the specified unit,
the Metrics table for all units or the specified unit, the Test Case Data
report for the test case(s) specified, and the Execution Results report for
the test case(s) specified. The Subprogram and Test Case parameters impact
only the Test Case Data report and the Execution Results sections.
```

See also "View a Test Case Data Report" on page 411

See also "To Execute a Test Case and View Results" on page 390.

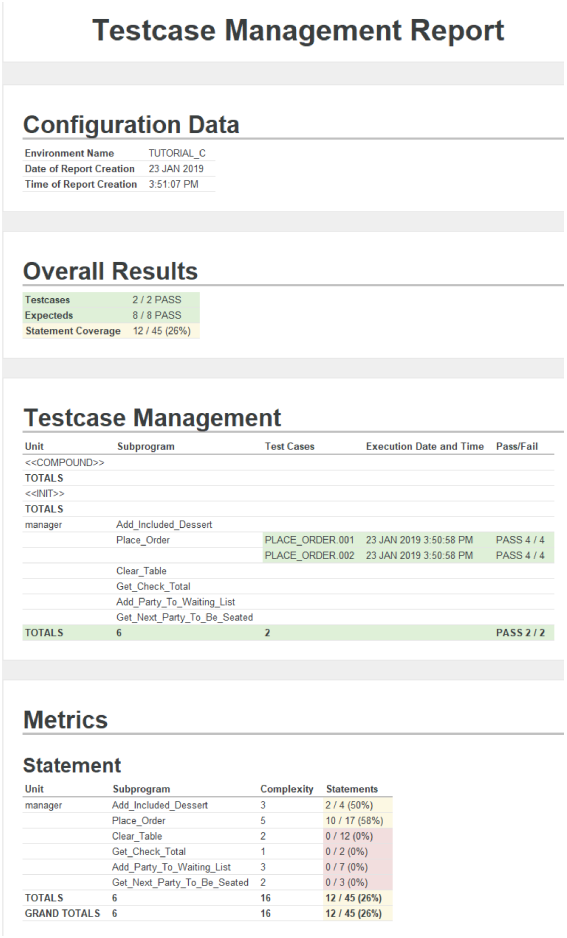
The Test Case Management Report

The Test Case Management Report is a summary of the testing status for a particular environment. The report contains information relating to pass/fail status for each executed test case, code coverage summary, code complexity metrics, overall pass/fail status for all test cases, and the number of Expected Value comparisons passing out of the total number of expected values. The contents of this report reflect the items selected in the Test Case Tree.

```
clicast -e <env> [-u <unit>] REports Custom MAnagement <outputfile>

Create a Test Case Management report and output to HTML file <env>_
management_report.html, standard output as text, or to the specified output
file. If the Unit parameter is specified, the report includes the Overall
Execution and Coverage summary for the specified unit, the Pass/Fail status
for all test cases in that unit, and the Metrics report for only the
specified unit. Otherwise, the report includes Overall Execution and
Coverage summary for the environment, the Pass/Fail status for all test
cases, including <<INIT>> and <<COMPOUND>>, and the Metrics report for all
units.
```

The following is an example Test Case Management report.



Testcase Management Report

Configuration Data

Environment Name	TUTORIAL_C
Date of Report Creation	23 JAN 2019
Time of Report Creation	3:51:07 PM

Overall Results

Testcases	2 / 2 PASS
Expecteds	8 / 8 PASS
Statement Coverage	12 / 45 (26%)

Testcase Management

Unit	Subprogram	Test Cases	Execution Date and Time	Pass/Fail
<<COMPOUND>>				
TOTALS				
<<INIT>>				
TOTALS				
manager	Add_Included_Dessert			
	Place_Order	PLACE_ORDER.001	23 JAN 2019 3:50:58 PM	PASS 4 / 4
		PLACE_ORDER.002	23 JAN 2019 3:50:58 PM	PASS 4 / 4
	Clear_Table			
	Get_Check_Total			
	Add_Party_To_Waiting_List			
	Get_Next_Party_To_Be_Seated			
TOTALS	6	2		PASS 2 / 2

Metrics

Statement

Unit	Subprogram	Complexity	Statements
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%)
GRAND TOTALS	6	16	12 / 45 (26%)

View Aggregate Coverage Report

An Aggregate Coverage Report includes for each unit selected in the Test Case Tree:

- an annotated source-listing, indicating the lines covered
- a metrics table giving the complexity and the level of coverage achieved for each subprogram

The report is sensitive to what level is selected in the Test Case Tree. If you select the top-level node for the environment, the report includes coverage achieved by test cases executed in all UUTs and non-stubbed units that have coverage instrumented using Custom Initialize. You can narrow the content down by selecting particular units in the Test Case Tree for which you want coverage achieved. Once you have made a unit selection, choose **Test => View => Aggregate Coverage Report**.

```
clicast -e <env> [-u <unit>] REports Custom Coverage <outputfile>

Create an Aggregate Coverage report and output to HTML file <env>_
aggregate_coverage_report.html, standard output as text, or to the
specified output file. If the Unit parameter is specified, the report
includes the annotated source code for the specified unit, and the Metrics
table includes only the specified unit; otherwise, the report includes the
```

```
annotated source code for all units and the Metrics table includes all
units in the environment.
```

For more information, see "To View the Aggregate Coverage Report" on page 499.

View Metrics Report

A Metrics Report includes a metrics table giving the complexity and the level of coverage achieved for each subprogram in the selected unit(s) for each unit selected in the Test Case Tree.

The report is sensitive to what level is selected in the Test Case Tree. If you select the top-level node for the environment, the Metrics table includes coverage achieved by test cases executed in all UUTs and non-stubbed units that have coverage instrumented using Custom Initialize. You can narrow the content down by selecting particular units in the Test Case Tree for which you want coverage achieved. Once you have made a unit selection, choose **Test => View => Metrics Report**.

```
clicast -e <env> [-u <unit>] Reports Custom Metrics <outputfile>

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

For more information, see "To View the Metrics Report" on page 500.

View 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.

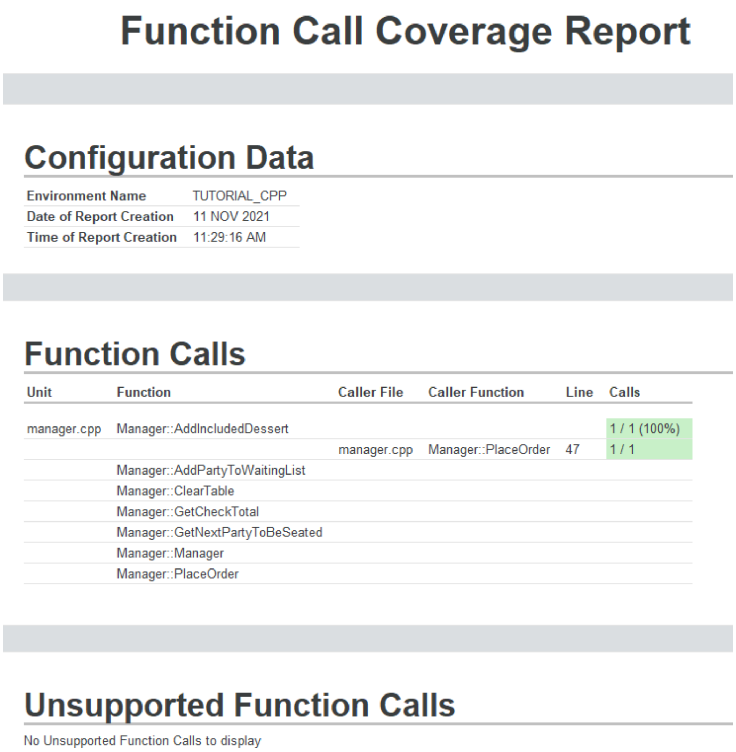
The second section of the report displays function calls in which the caller function cannot be determined. For example, C++ Virtual functions are shown in the Unsupported Function Calls section of the report for this reason.

To access the Function Call Report, select **Test => View => Function Call Report** from the Menu Bar. Note that this option is only available when function call coverage is present.

```
clicast -e <env> [-u <unit>] REports Custom 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	TUTORIAL_CPP
Date of Report Creation	11 NOV 2021
Time of Report Creation	11:29:16 AM

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

View 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 **Test => View => Covered By Analysis Report** from the Menu Bar.

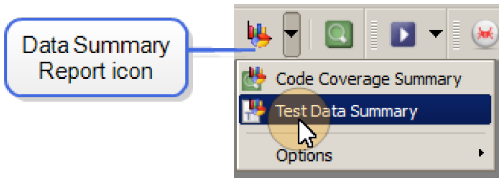
```
clicast -e <env> Reports Custom CBA <outputfile>

Create a CBA report and output to HTML file <env>_covered_by_analysis_
report.html, standard output as text, or to the specified output file.
```

For more information on the Covered By Analysis Report, see "Viewing Analysis Data in Reports" on page 587.

View Test Data Summary

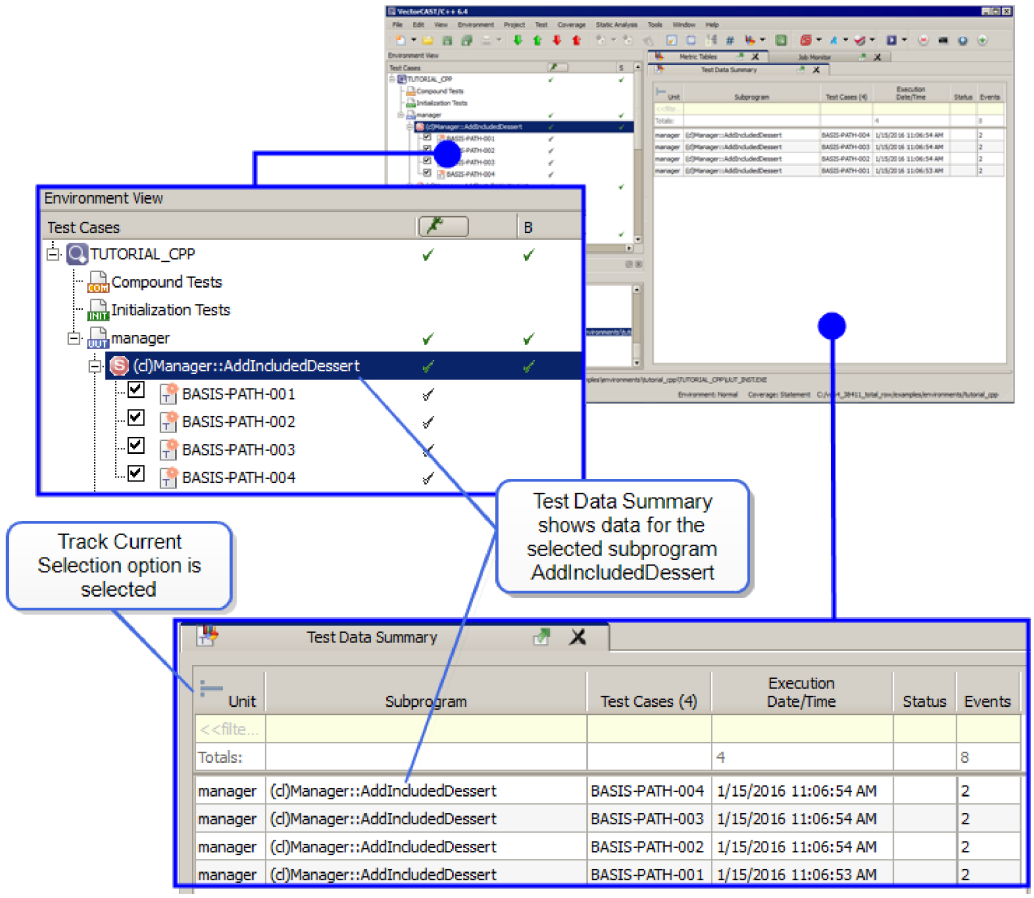
The Test Data Summary allows the user to easily view a summary of test data for UUTs, subprograms and test cases in an environment. To open the summary from the Toolbar, select **Test Data Summary** from the Data Summary Report icon drop-down menu.



Viewing Test Data

By default, the contents of the Test Data Summary reflect the items selected in the Test Case Tree, and show test case data for test cases, subprograms, UUTs or an environment, depending on where you clicked before selecting the menu item. A tracking icon is displayed at the top of the Unit column indicating that the Test Data Summary is currently tracking selected items from the Test Case Tree.

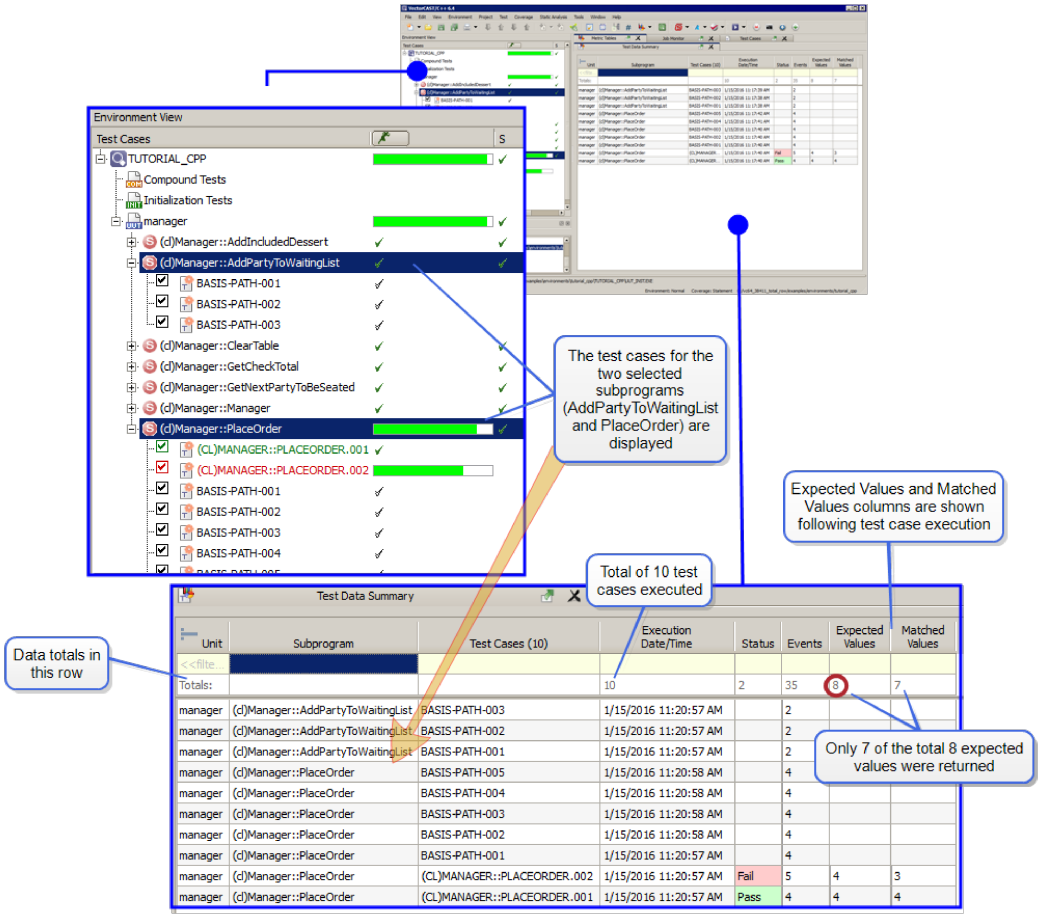
To override tracking of selected items in the Test Case Tree, open the drop-down menu for the Data Summary Report and select **Options => Track Current Selection**. Remove the check next to the option. The tracking icon on the Summary table will change to gray to indicate that the summary is now tracking all items in the Test Case Tree.



The Test Data Summary table is dynamic. When the **Track Current Selection** option is enabled, as

nodes are selected and deselected in the Test Case Tree, the Test Data Summary table updates in real time reflecting the selections.

The data displayed in the Test Data Summary includes the Unit name, Subprogram name, Test Case name, most recent Execution Date/Time, Status and Events for each tracked selection. Additional columns are added to the Test Data Summary when that data is available. These additional columns are: Expected Values, Matched Values, Control Flow, Signals, Exceptions and Abnormal Termination.



Unit	Subprogram	Test Cases (10)	Execution Date/Time	Status	Events	Expected Values	Matched Values
Totals:			10	2	35	8	7
manager	(d)Manager::AddPartyToWaitingList	BASIS-PATH-003	1/15/2016 11:20:57 AM		2		
manager	(d)Manager::AddPartyToWaitingList	BASIS-PATH-002	1/15/2016 11:20:57 AM		2		
manager	(d)Manager::AddPartyToWaitingList	BASIS-PATH-001	1/15/2016 11:20:57 AM		2		
manager	(d)Manager::PlaceOrder	BASIS-PATH-005	1/15/2016 11:20:58 AM		4		
manager	(d)Manager::PlaceOrder	BASIS-PATH-004	1/15/2016 11:20:58 AM		4		
manager	(d)Manager::PlaceOrder	BASIS-PATH-003	1/15/2016 11:20:58 AM		4		
manager	(d)Manager::PlaceOrder	BASIS-PATH-002	1/15/2016 11:20:58 AM		4		
manager	(d)Manager::PlaceOrder	BASIS-PATH-001	1/15/2016 11:20:57 AM		4		
manager	(d)Manager::PlaceOrder	(CL)MANAGER::PLACEORDER.002	1/15/2016 11:20:57 AM	Fail	5	4	3
manager	(d)Manager::PlaceOrder	(CL)MANAGER::PLACEORDER.001	1/15/2016 11:20:57 AM	Pass	4	4	4

Totals are provided for the data being tracked. The Totals row at the top of the table displays the totals for each data column. In the example above, note that we have a total of 10 test cases executed, and of the 8 total expected values, 7 were matched.

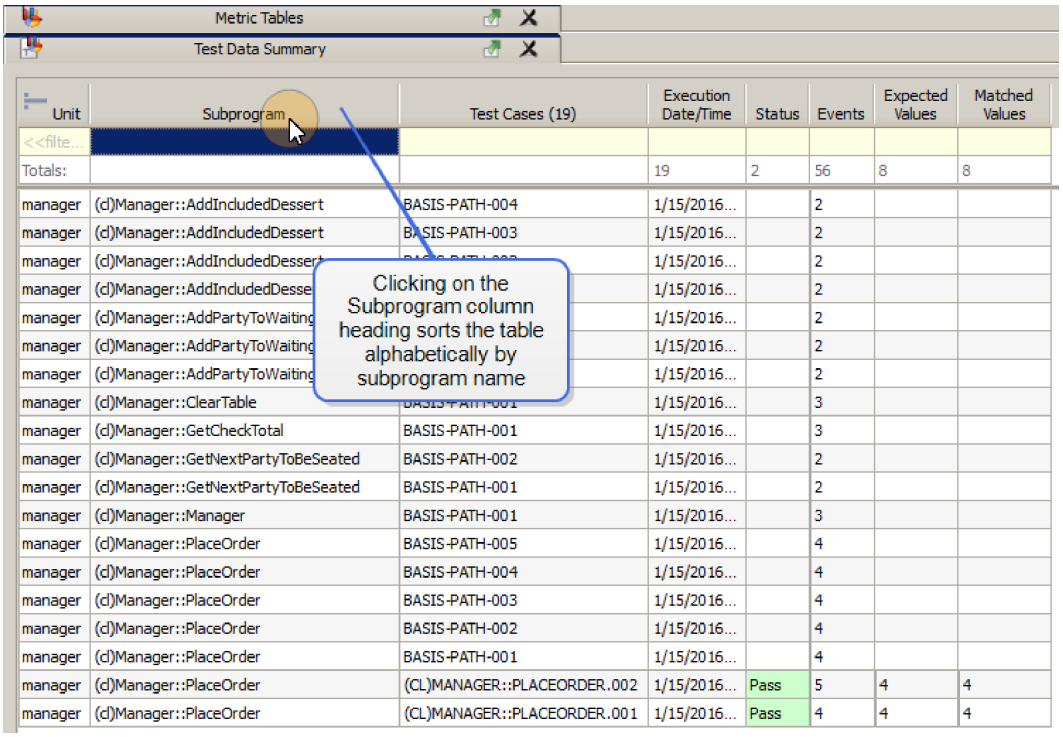
The Summary table updates whenever test data is updated. For example, the table refreshes when test cases are inserted, deleted, or duplicated, or following test execution.

Double-clicking on a line in the Test Data Summary opens the corresponding test case in the Test Case editor.

Sorting and Filtering Test Data

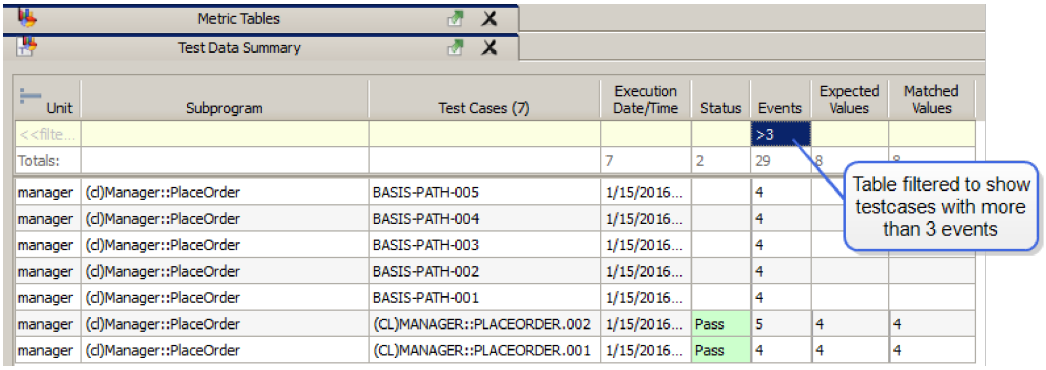
Sorting and filtering is available to locate data of interest. Sort by clicking on any column heading. The

data will sort in alphabetic or numeric order, as appropriate. Clicking the heading again reverses the order. By default, the table sorts using the Execution Date/Time column, showing the most recently executed test cases first.



Unit	Subprogram	Test Cases (19)	Execution Date/Time	Status	Events	Expected Values	Matched Values
<<filte...							
Totals:			19	2	56	8	8
manager	(d)Manager::AddIncludedDessert	BASIS-PATH-004	1/15/2016...		2		
manager	(d)Manager::AddIncludedDessert	BASIS-PATH-003	1/15/2016...		2		
manager	(d)Manager::AddIncludedDessert		1/15/2016...		2		
manager	(d)Manager::AddIncludedDesse		1/15/2016...		2		
manager	(d)Manager::AddPartyToWaiting		1/15/2016...		2		
manager	(d)Manager::AddPartyToWaiting		1/15/2016...		2		
manager	(d)Manager::AddPartyToWaiting		1/15/2016...		2		
manager	(d)Manager::ClearTable		1/15/2016...		3		
manager	(d)Manager::GetCheckTotal	BASIS-PATH-001	1/15/2016...		3		
manager	(d)Manager::GetNextPartyToBeSeated	BASIS-PATH-002	1/15/2016...		2		
manager	(d)Manager::GetNextPartyToBeSeated	BASIS-PATH-001	1/15/2016...		2		
manager	(d)Manager::Manager	BASIS-PATH-001	1/15/2016...		3		
manager	(d)Manager::PlaceOrder	BASIS-PATH-005	1/15/2016...		4		
manager	(d)Manager::PlaceOrder	BASIS-PATH-004	1/15/2016...		4		
manager	(d)Manager::PlaceOrder	BASIS-PATH-003	1/15/2016...		4		
manager	(d)Manager::PlaceOrder	BASIS-PATH-002	1/15/2016...		4		
manager	(d)Manager::PlaceOrder	BASIS-PATH-001	1/15/2016...		4		
manager	(d)Manager::PlaceOrder	(CL)MANAGER::PLACEORDER.002	1/15/2016...	Pass	5	4	4
manager	(d)Manager::PlaceOrder	(CL)MANAGER::PLACEORDER.001	1/15/2016...	Pass	4	4	4

Test data can be accessed all the way down to the individual test case 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 more than 3 events.



Unit	Subprogram	Test Cases (7)	Execution Date/Time	Status	Events	Expected Values	Matched Values
<<filte...					>3		
Totals:			7	2	29	8	8
manager	(d)Manager::PlaceOrder	BASIS-PATH-005	1/15/2016...		4		
manager	(d)Manager::PlaceOrder	BASIS-PATH-004	1/15/2016...		4		
manager	(d)Manager::PlaceOrder	BASIS-PATH-003	1/15/2016...		4		
manager	(d)Manager::PlaceOrder	BASIS-PATH-002	1/15/2016...		4		
manager	(d)Manager::PlaceOrder	BASIS-PATH-001	1/15/2016...		4		
manager	(d)Manager::PlaceOrder	(CL)MANAGER::PLACEORDER.002	1/15/2016...	Pass	5	4	4
manager	(d)Manager::PlaceOrder	(CL)MANAGER::PLACEORDER.001	1/15/2016...	Pass	4	4	4

Filtering supports the following symbols: <, >, =. Examples of filtering inputs are:

10 - lists test cases matching the specific value of "10" in the selected column

>50 - lists test cases greater than the value of "50" in the selected column

<90 - lists tests cases less than the value of "90" in the selected column

=100 - lists test cases matching the specific value of "100" in the selected column

< - lists test cases with empty values in the selected column

> - lists test cases with non-empty values in the selected column

= - lists test cases with non-empty values in the selected column

Saving and Printing Test Data Summary

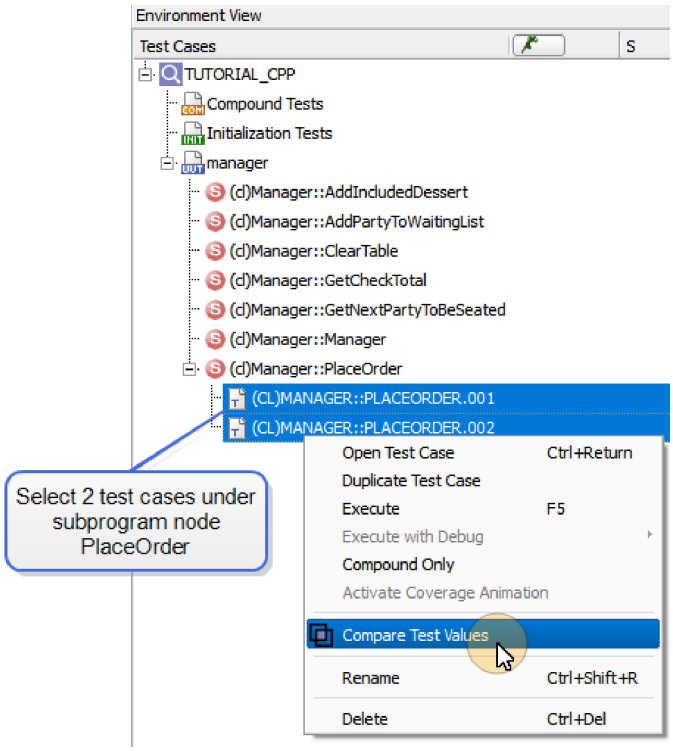
Saving the Test Data Summary in .html or .csv format is supported. See "To Save a Script or Text File " on page 86 for more information.

Printing the contents of the Test Data Summary is supported. See "To Print an Open Window" on page 87 for more information.

View the Test Comparison Report

The Test Comparison Report allows users to compare the test case data of selected test cases within a subprogram. The Test Comparison Report is presented as a filterable table, allowing the user to see similarities and differences in test cases as the data for multiple tests is compared. VectorCAST supports the generation of a CSV export of all test case data.

To open the Test Comparison Report, select two or more test cases under a single subprogram node, and right-click. Select **Compare Test Values** from the right-click context menu.

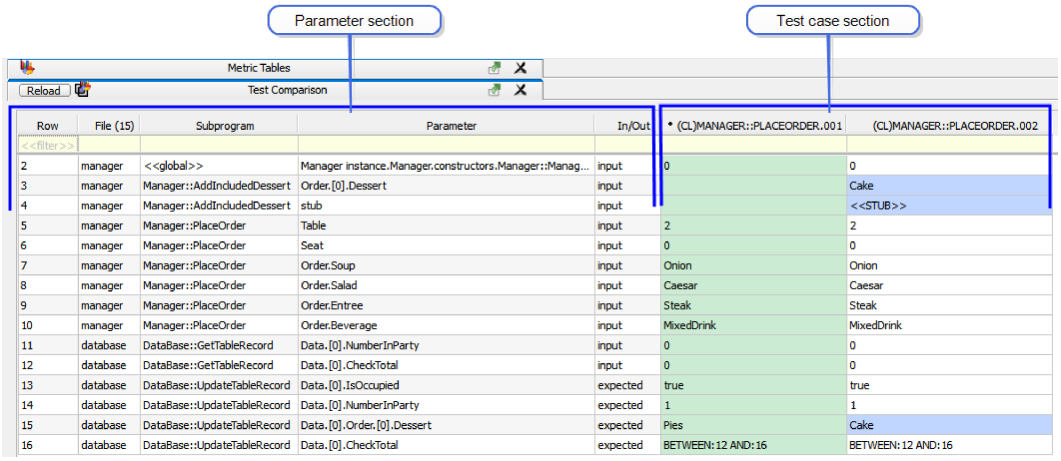


Note: Test cases from different subprograms cannot be compared. You must select at least two test cases from the same subprogram or <<INIT>> node (including specialized test cases and test cases in a TDD environment) to generate the Test Comparison Report.

Selection of (MAP)CSV test cases and <<MIN>>, <<MID>>, and <<MAX>> test cases is not supported.

Viewing the Test Comparison Data

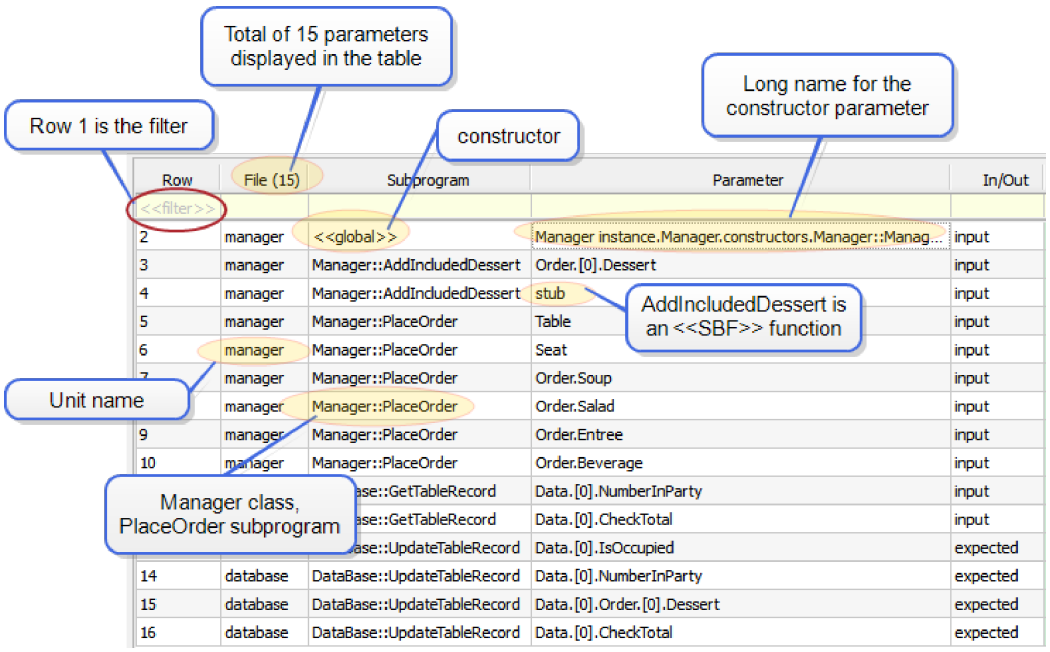
The Test Comparison Report is divided into two sections: the Parameter section and the Test Case section. The report is presented in table format and the user can control the display of the test data through the use of filters and using the Column Controller's show / hide capability.



Row	File (15)	Subprogram	Parameter	In/Out	• (CL)MANAGER::PLACEORDER.001	(CL)MANAGER::PLACEORDER.002
<<filter>>						
2	manager	<<global>>	Manager instance.Manager.constructors.Manager::Manag...	input	0	0
3	manager	Manager::AddIncludedDessert	Order.[0].Dessert	input		Cake
4	manager	Manager::AddIncludedDessert	stub	input		<<STUB>>
5	manager	Manager::PlaceOrder	Table	input	2	2
6	manager	Manager::PlaceOrder	Seat	input	0	0
7	manager	Manager::PlaceOrder	Order.Soup	input	Onion	Onion
8	manager	Manager::PlaceOrder	Order.Salad	input	Caesar	Caesar
9	manager	Manager::PlaceOrder	Order.Entree	input	Steak	Steak
10	manager	Manager::PlaceOrder	Order.Beverage	input	MixedDrink	MixedDrink
11	database	DataBase::GetTableRecord	Data.[0].NumberInParty	input	0	0
12	database	DataBase::GetTableRecord	Data.[0].CheckTotal	input	0	0
13	database	DataBase::UpdateTableRecord	Data.[0].IsOccupied	expected	true	true
14	database	DataBase::UpdateTableRecord	Data.[0].NumberInParty	expected	1	1
15	database	DataBase::UpdateTableRecord	Data.[0].Order.[0].Dessert	expected	Pies	Cake
16	database	DataBase::UpdateTableRecord	Data.[0].CheckTotal	expected	BETWEEN: 12 AND: 16	BETWEEN: 12 AND: 16

Parameter Section

The Parameter section displays one row for each parameter.



Row	File (15)	Subprogram	Parameter	In/Out
<<filter>>				
2	manager	<<global>>	Manager instance.Manager.constructors.Manager::Manag...	input
3	manager	Manager::AddIncludedDessert	Order.[0].Dessert	input
4	manager	Manager::AddIncludedDessert	stub	input
5	manager	Manager::PlaceOrder	Table	input
6	manager	Manager::PlaceOrder	Seat	input
7	manager	Manager::PlaceOrder	Order.Soup	input
8	manager	Manager::PlaceOrder	Order.Salad	input
9	manager	Manager::PlaceOrder	Order.Entree	input
10	manager	Manager::PlaceOrder	Order.Beverage	input
11		ase::GetTableRecord	Data.[0].NumberInParty	input
12		ase::GetTableRecord	Data.[0].CheckTotal	input
13		ase::UpdateTableRecord	Data.[0].IsOccupied	expected
14	database	DataBase::UpdateTableRecord	Data.[0].NumberInParty	expected
15	database	DataBase::UpdateTableRecord	Data.[0].Order.[0].Dessert	expected
16	database	DataBase::UpdateTableRecord	Data.[0].CheckTotal	expected

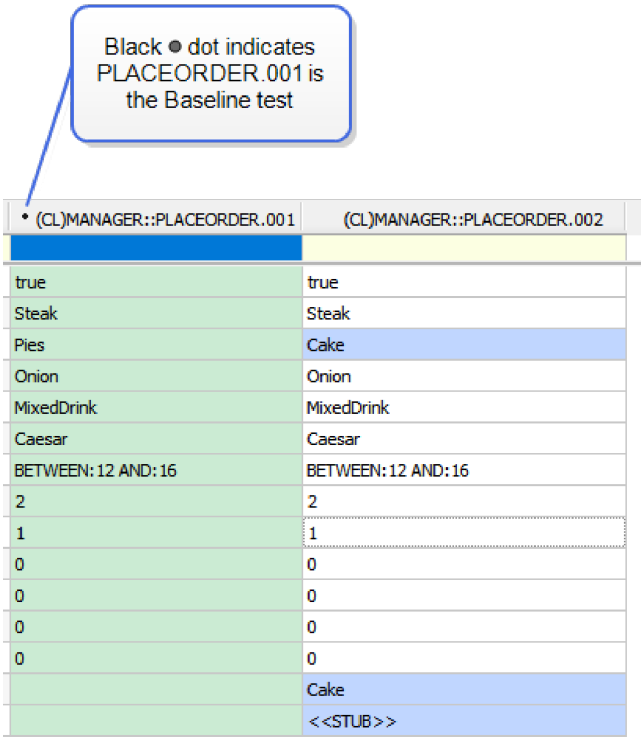
The following data is provided for each parameter:

- **Row** - Row number, starting with 2 (the filter occupies row 1 of the table)
- **File** - Unit name or stubbed unit name. For a <<GLOBAL>> parameter, USER_GLOBALS_VCAST is displayed. For <<INIT>> test cases, the unit is displayed as <<INIT>>. Note that in the header for the File column the total number of parameters in the table is displayed in parentheses. For example, File (15).
- **Subprogram** - For constructors and destructors, displayed as <<global>>. For <<SBF>> constructors and subprograms, displayed as class::subprogram. For example, Manager::PlaceOrder.
- **Parameter** - For constructors, the long name is displayed. For an <<SBF>> constructor or subprogram, the parameter name is "stub". For stub-by-implementation (ENVIRO>STUB:database) , the parameter name is "stub".
- **In/Out** - Values are "input" or "expected".

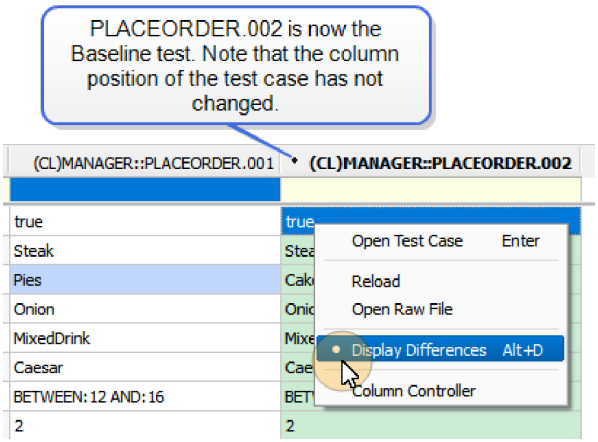
Test Case Section

Setting the Baseline Test

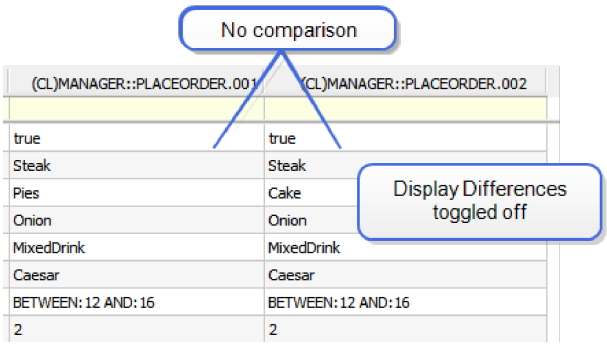
In the Test Case section, one test is designated as the Baseline Test, which the other tests are compared to. By default, the Baseline test is the first test created and is displayed in the first column following the In/Out column in the Parameter section. A black dot in the column header and green cell background indicate the location of the Baseline test in the report.



To change which test case is the Baseline test, right-click in any cell under a different test and select **Display Differences** from the context menu, or press **Alt+D**.

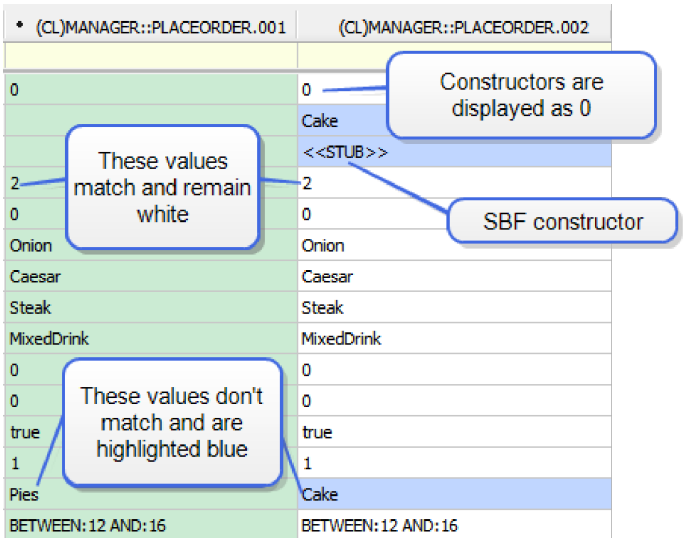


The **Display Differences** setting can be toggled on and off by using the right-click context menu or pressing the **Esc** key. When toggled off all cells have a white background and the table is not doing a comparison, but is just listing the values.



Test Case Comparison

When a test case is compared to the Baseline test, if there are differences in values, the cell background is blue. If there are no differences, the cell background remains white.

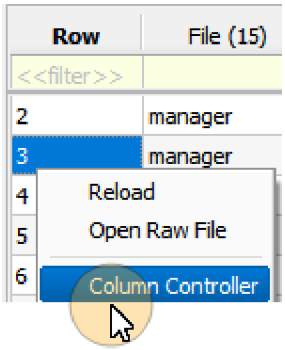


Constructors are displayed as 0. An <<SBF>> constructor or subprogram is displayed as <<STUB>>.

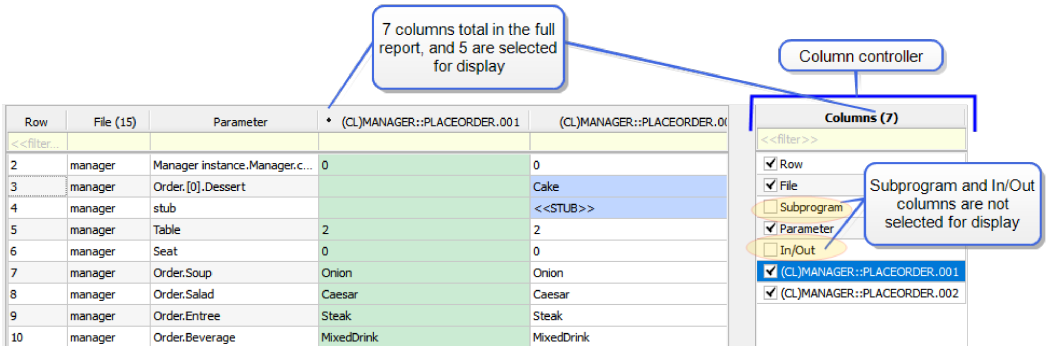
Parameter User Code is displayed as <user>. If both tests have any text in Parameter User Code, they are treated as a match. Note that Test Case User code is skipped.

The Column Controller

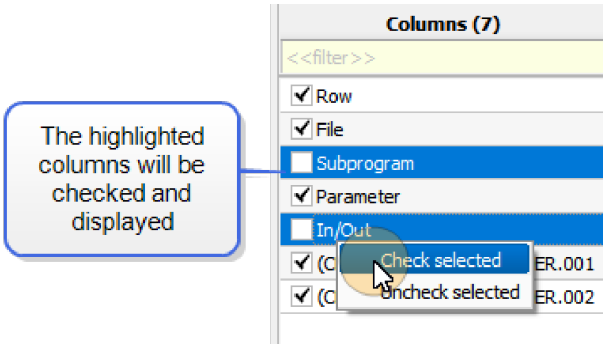
The Column Controller is used to show/hide the columns contained in the Test Comparison Report. To open the Column Controller, right-click on any cell in the Report and select **Column Controller** from the context menu.



The Column Controller panel opens to the right of the Report. Place a checkmark in the boxes of each column to be shown in the Report. Unselected boxes hide the column from view. Hiding columns does not impact the data contained in the Report.



You may also highlight the columns that you want to select or deselect, and right-click to use the context menu to **Check selected** or **Uncheck selected**.



Close the Column Controller panel by right-clicking in any cell in the Report and selecting **Column Controller** from the context menu to toggle the Controller off.

Open a Test Case From the Test Comparison Report

Test cases are accessible from any cell in a test case column. To open the Test Case Editor for a selected test case, perform one of the following:

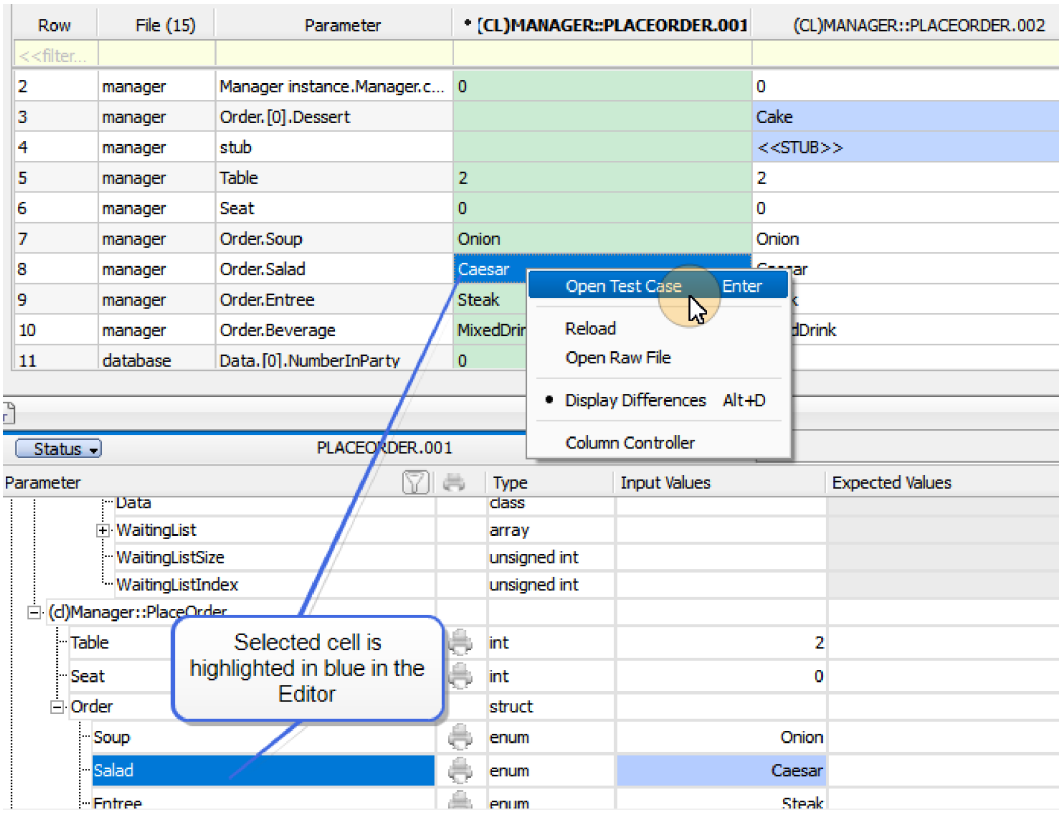
> Right-click on a cell in a test case column and select **Open Test Case** from the context menu. (This will leave keyboard focus in the cell in the Test Case Editor.)

> Highlight a cell in a test case column and select **Enter**. (This will leave keyboard focus in the cell in the Test Case Editor.)

> Highlight a cell in a test case column and select **Alt+Enter**. (This will leave keyboard focus in the cell in the Report, but using the up-arrow and down-arrow keys will move the selection to the associated parameter value in the Test Case Editor.)

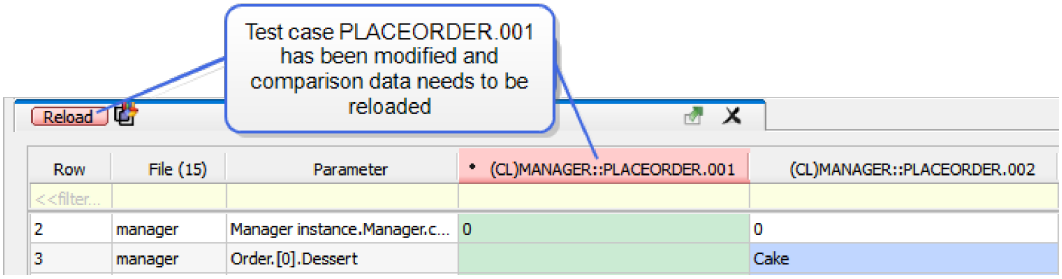
> Double-click on a cell in a test case column. (This will leave keyboard focus in the cell in the Test Case Editor.)

The Test Case Editor opens in a horizontal pane under the Report. The cell used to perform the Open action is highlighted in blue in the Editor, making it easy to locate the value to be edited.



Reloading Test Case Data

When data in the Test Case Comparison Report is current, the **Reload** button in the upper left corner of the Report has a gray background. Reload . When test case data is edited and saved, the comparison data becomes obsolete and must be reloaded. The **Reload** button changes to a red background, and the column header for the affected test case also changes to a red background.

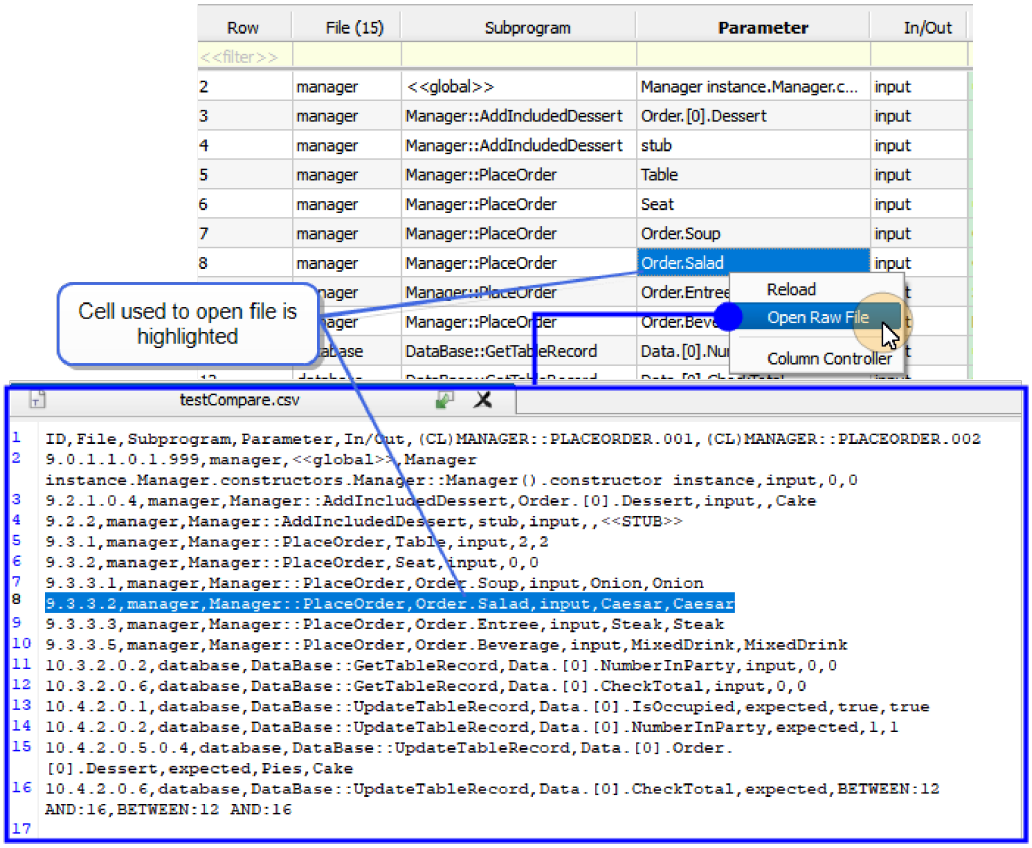


To reload the comparison data, click the **Reload** button. Alternatively, right-click on any cell in the Report and select **Reload** from the context menu. The values for the test case update. Note that reloading may result in a change in comparison, and cell colors (white = matching, blue = difference) may change.

Renaming a test case does not trigger a reload and renamed tests are left out of the reload. Reloading recreates the Report using the original arguments (test case names) and does not recognize the renamed test case.

Open Raw File

The job execution creates a raw file, named **`testCompare.csv`**, in the ENV directory. To view the raw file, right-click on any cell in the Report and select **Open Raw File** from the context menu. The cell used to perform the Open action is highlighted in blue in the CSV Viewer.



View a Test Case's Raw Data Set

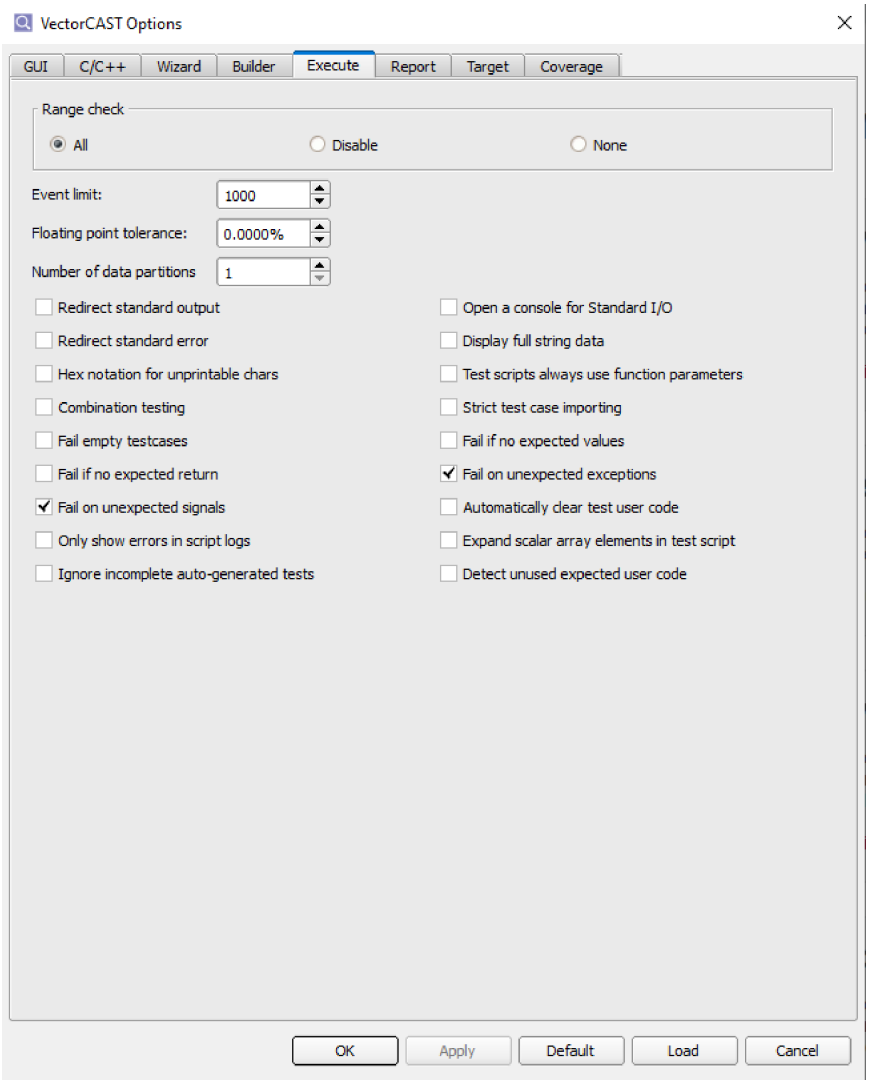
The **Test => View => Raw Data Set** command provides a view of the raw test case data and is used for Technical Support purposes only.

Setting Execution Options

Options: Execute Tab

Choose **Tools => Options** and click the **Execute tab**.

The Execution options tab is used to change the way VectorCAST executes test cases, on an environment-wide scale. To change execution options for an individual test case, use Test Case Options. Pass your cursor over any of the options to see an explanation of that option in a tool-tip.



See also "To Set VectorCAST Options" on page 376.

Range Check

The Range Check option provides the following choices:

- All (default) – determines valid ranges of data types, and enforces input and expected values within valid range
- Disable – determines ranges of data types, but does not enforce
- None – does not determine ranges of data types

VectorCAST provides for full range checking of test data during test cases building. This is reflected in the default range check setting of "All".

However, when building test cases it is often desirable to be able to specify out-of-range test values for parameters and global data. The Range Check option provides this capability with the “Disable” option. When range checking is disabled, the dialog boxes will still show you the data range that is

allowed, but will not enforce that range. This will allow you to enter values that are beyond the data range for the particular parameter or object (including enumerated values).

In order to determine data type ranges, VectorCAST must execute a small program. You can disable this processing by setting the **Range Check option** to **None.** The “None” position is similar to “Disable” with the added restriction that VectorCAST will not attempt to determine scalar ranges. The “None” option is normally used in target testing where execution speed is an issue.

```
clicast -lc option RANGE_CHECK Full | Disable | None

Indicates whether range checking should be performed when accepting input
and expected values. NONE indicates harness should not perform processing
used to verify ranges (useful on slow targets).  The default value is Full.
```

Event Limit

The Event Limit option enables you to control the maximum number of events per test harness execution. The event limit defaults to 1000. If this option is set to 0, no results data is captured, although coverage data is captured. If the event limit is exceeded, VectorCAST automatically stops test execution if your compiler supports signals, and the execution report shows:



The Execution Report shows <limit>+1 events.

The event limit applies to each test case. Each time a test executes, it resets the event counter to 1. If you execute several test cases at once using Ctrl+Shift click to select them, or execute them all by selecting the environment name and choosing **Test => Execute**, the counter starts over with each test case (because the test harness is being executed multiple times).

Note: An individual test case can have its own event limit, which overrides the environment event limit when that test case is executed. This option is accessed via the **Options tab** in the **Test Case Editor**.

When executing a compound test case, the event limit is not reset for each slot; the Event Limit setting applies to the Compound Test Case as a whole.

Note: The Event Limit feature is implemented through the raising of signals. If your target does not support signals, the event limit capability will not be available.

```
clicast -lc option EVENT_LIMIT <limit>

Maximum number of events for the harness to process. If this option is set
to 0, no results data is captured, although coverage data is captured. Its
default value is 1000.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.EVENT_LIMIT. Its default value is the environment-wide setting for Event Limit, accessible on the Execute tab of the **Tools => Options**

dialog.

The harness event limit data type can be changed by setting the macro **VCAST_EVENT_LIMIT_TYPE** located in **Tools => Options => C/C++ => Defined variables**. For example, specify **VCAST_EVENT_LIMIT_TYPE=long** to allow a larger event limit. The default type for **VCAST_EVENT_LIMIT_TYPE** is **unsigned int**.

```
clicast -lc option C_DEFINE_LIST VCAST_EVENT_LIMIT_TYPE=<type>

where <type> is one of: long, short, or char. The default type is unsigned
int, achieved by removing the VCAST_EVENT_LIMIT_TYPE define from the
configuration.
```

See also "Key Terminology" on page 17 for an explanation of an event.

Floating Point Tolerance (%)

The Floating Point Tolerance option enables you to set the tolerance of expected results for floating point numbers. This tolerance applies to all test case data in the environment.



Note: There is no tolerance around the value 0.

```
clicast -lc option FLOATING_POINT_TOLERANCE <tolerance>

Percentage that a floating point actual value can vary from its expected
value and still be considered a match. <percentage> is a floating point
number. Its default value is 0.000000.
```

In a test script, the testcase-wide floating point tolerance is specified using:

TEST.FLOATING_POINT_TOLERANCE:<tolerance>



Note: For very small tolerances, the floating point representation of the value may not be exact due to the resolution of floating point numbers; in this case, the floating point tolerance may show more digits than the user expects. For example, a tolerance of 0.000000001 may actually be displayed as 0.00000000099999.

Number of Data Partitions

This option specifies the number of iterations to span an entire range of a scalar parameter. It is used in conjunction with Range Values to allow a specification of iterations instead of using a range delta.

```
clicast -lc Option NUMBER_OF_DATA_PARTITIONS <positive number>

Number of iterations to span an entire range of a scalar parameter.  Used
with Range Values to allow specification of iterations instead of range
delta. The default value is 1 partition.
```

Redirect Standard Output

The redirect standard output option enables you to redirect test stdout execution output to a file. Captured text is appended to the end of the Execution Report.

```
clicast -lc option STANDARD_OUTPUT Normal | Redirect

Indicates whether standard output should go to the console or be captured
to a file for inclusion into the execution report. Its default value is
NORMAL (off).
```

Redirect Standard Error

The redirect standard error option enables you to redirect test execution stderr output to a file. Captured text is appended to the end of the Execution Report.

```
clicast -lc option STANDARD_ERROR Normal | Redirect

Indicates whether standard error should go to the console or be captured to
a file for inclusion into the execution report. Its default value is NORMAL
(off).
```

Hex Notation for Unprintable Chars

If a string contains a non-printable character (excluding NULL), those characters are displayed using octal numeric representation. If this option is set, the characters will be displayed using hexadecimal notation. To display the terminating NULL character and all following elements of a character array, use the Display Full String Data option.

```
clicast -lc option VCAST_HEX_NOTATION true | false

Display non-printable characters using hexadecimal notation. The default
value is False.
```

In a test case script, this option is:

```
TEST.VALUE:<<OPTIONS>>.HEX_NOTATION_FOR_UNPRINTABLE_CHARS

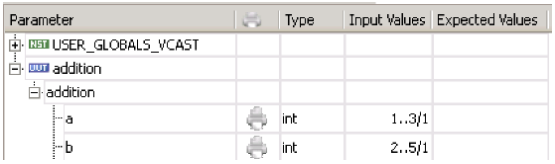
Testcase-specific value for "Hex Notation for Unprintable Chars." Its
default value is the environment-wide setting for "Hex notation for
unprintable chars,"accessible on the Execute tab of the Tools => Options
dialog.
```

Combination Testing

If a test case has a parameter with an input range or list, then the test harness executes the test case once for each iteration. When two or more parameters have a range iteration, then each parameter's iteration count is incremented at the same time, by default.

Combination testing can be turned on for an individual test case, using Test Case Options, or for the whole environment, using the **Tools => Options** dialog, **Execute tab**. If combination testing is enabled, VectorCAST will determine the combinations of input values and use those values as stimulus values.

For example, if parameter A has a range 1 to 3 with a delta of 1 and parameter B has a range 2 to 5 with a delta of 1, and Combination Testing is off, then the test case executes this way:



Range Iteration #1:

A is 1
B is 2

Range Iteration #2:

A is 2
B is 3

Range Iteration #3:

A is 3
B is 4

Range Iteration #4:

A is 3 again
B is 5

If the combination testing option is enabled, then the test case executes this way:

Range Iteration #1:

A is 1
B is 2

Range Iteration #2:

A is 1

B is 3

Range Iteration #3:

A is 1

B is 4

Range Iteration #4:

A is 1
B is 5

Range Iteration #5:

A is 2

B is 2

Range Iteration #6:

A is 2
B is 3
Range Iteration #7:
A is 2
B is 4
Range Iteration #8:
A is 2
B is 5
Range Iteration #9:
A is 3
B is 2
Range Iteration #10:
A is 3
B is 3
Range Iteration #11:
A is 3
B is 4
Range Iteration #12:
A is 3
B is 5

This feature is not available with an environment built with VectorCAST version 3.2 or earlier. Rebuilding the environment will enable this feature's functionality.

```
clicast -lc option VCAST_DO_COMBINATION true | false

Use all combinations of the values in the range expressions for test
stimulus values. The default value is False.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.DO_COMBINATION. Its default value is the environment-wide setting for Combination testing, accessible on the Execute tab of the **Tools => Options** dialog.

Fail Empty Testcases

If this option is set, then empty testcases will be marked as failed.

```
clicast -lc Option VCAST_EMPTY_TESTCASE_FAIL True | False

If this option is set, then empty testcases will be marked as failed.
```

Fail If No Expected Return

If this option is set, testcases that have no expected return value or expected user code for the

function's return value are marked as failed.

Note that while similar to the option VCAST_TESTCASE_FAIL_ON_NO_EXPECTED, which marks any test as failed if it has no Expected Value or Expected User code, this option is specific to the return value of the subprogram under test only.

```
clicast -lc Option VCAST_TESTCASE_FAIL_ON_NO_EXP_RETURN True | False

If this option is set, then then testcases that have no expected return
value or expected user code for the return value will be marked as failed.
```

Fail on Unexpected Signals

When this option is set and a signal is raised during the execution of a test case, then the test case status is set to FAIL.

```
clicast -lc option VCAST_UNEXPECTED_SIGNALS_FAIL true | false

Sets test case status to Fail if a signal is raised during test case
execution. Its default value is TRUE.
```

Only Show Errors In Script Logs

When checked, this option limits the contents of test script logs to error messages. When loading a large test script, setting this option will make it much easier to find errors that occurred during test script importing.

```
clicast -lc option VCAST_SCRIPT_LOG_SHOW_ONLY_ERRORS true | false

Limits script logs to show errors only. The default value is False.
```

Ignore Incomplete Auto-Generated Tests

When VectorCAST generates automatic test cases (such as Basis Path tests), it labels the test as "partial" if it cannot specify all of the data for the test, and "template" if it cannot specify any of the data for the test.

This option tells VectorCAST to discard those partial or template tests when importing the generated test script.

```
clicast -lc option VCAST_IGNORE_INCOMPLETE_TESTS true | false

When building Basis Path or MCDC test cases, there are three outcomes for
each test: Complete, Partial, and Template. When this option is on,
VectorCAST will discard the Partial and Template tests, and only load the
Complete tests. The default value is False.
```

Open a Console for Standard I/O

If your program under test reads from stdin during test execution, then the test will hang if this stdin is not provided. To overcome this problem on Windows, you can set the VectorCAST option “Open a console for Standard I/O.” If this option is set, a DOS Command Prompt window (console) is opened before test case execution and you can use this window to type in stdin data.

If the option “Redirect Standard Output” is on as well, then standard output continues to be directed to the execution reports, and standard input is ignored. Therefore, turn off “Redirect Standard Output” when using this option. Under Linux, this option is dimmed, as harness standard input and standard output are by default available in the console window from which VectorCAST was started.

```
clicast -lc option VCAST_SHOW_STDOUT_CONSOLE true | false

Opens a console during test harness execution to facilitate standard input
and output. On Windows, if either of the execute options “Redirect standard
output” or “Redirect standard error” is set, then standard output and error
continue to be redirected to the Execution reports, but standard input is
ignored. The default value is False.
```

Display Full String Data

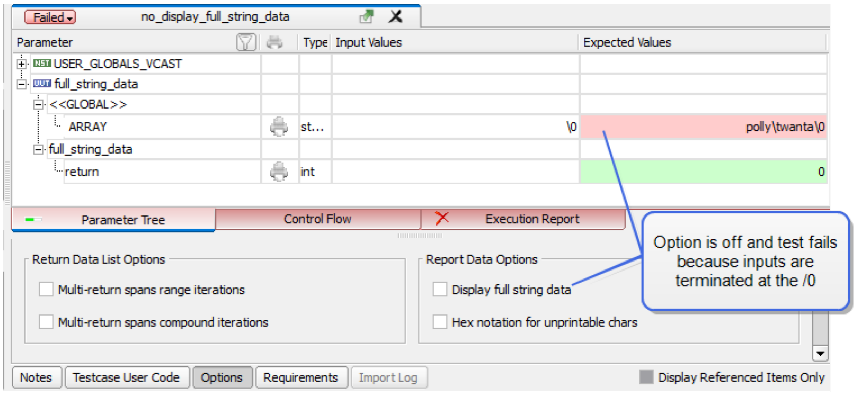
When this option is off (default), strings in the Input Values column are displayed up to the null terminator (\0) in the Execution report. During execution, the value of the parameter is the string up to the null terminator.

Note: An Expected Value is not affected by this option. An Expected Value is always displayed and compared in its entirety.

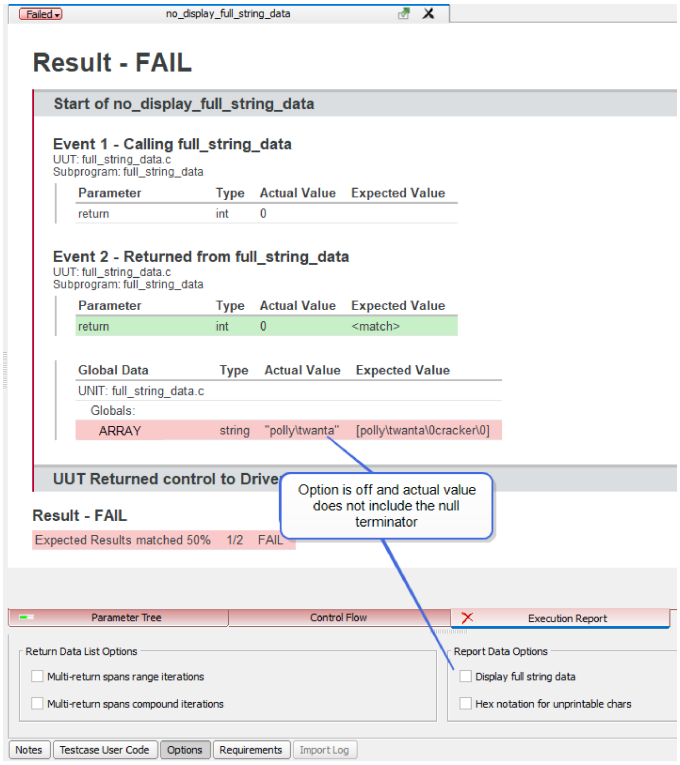
In the example shown below, a string global is defined:

```
char ARRAY[20];
```

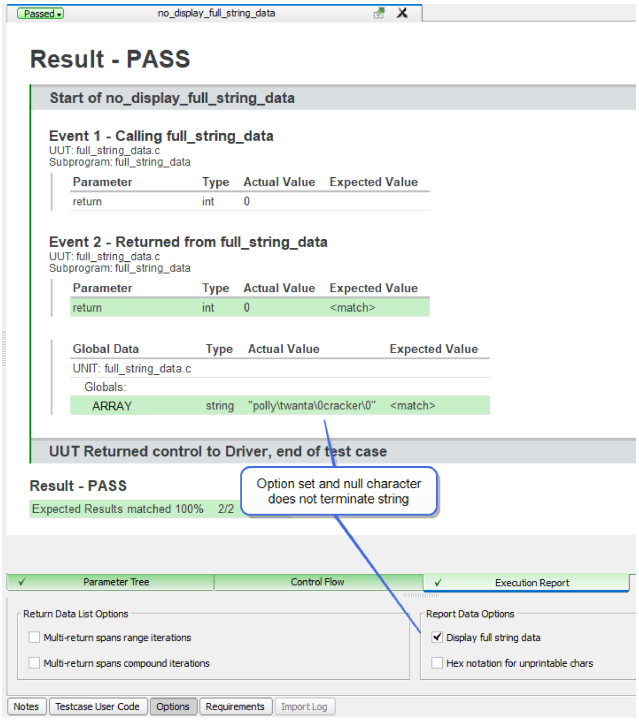
The code for this example sets the characters in the array and includes the unprintable characters \t and \0, the null terminator. When the option is off, these inputs are terminated at the \0. To pass, the Expected Value will only be compared until the null character.



The Execution Report, with the Display full string data option off, shows the Actual Value string up to the null terminator and shows that the Expected Value does not match the Actual Value.



When the Display full string data option is set, all elements of the array are displayed, including non-printable characters. As a result, the entire array is shown; the null character does not terminate the string.



Use the option Hex notation for unprintable chars to change the output to hexadecimal representation. For more information, see "Hex Notation for Unprintable Chars" on page 435.

```
clicast -lc option VCAST_FULL_STRINGS true | false

Display all elements of a character array, including non-printable
characters, using their numeric representation. Use the VCAST_HEX_NOTATION
option to choose octal or hexadecimal representation. This option also
affects how strings are compared during test case execution. The default
value is False.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.DISPLAY_FULL_STRING_DATA. The default value is False.

Test Scripts Always Use Function Parameters

When True, this option causes C++ function parameter and return types to always be included with the function name in test scripts. The default value is False, which causes parameterized names to only be used when a function is treated as overloaded.

The True value is useful mainly for tools that generate test scripts independently of an environment, when it isn't certain whether a function will be treated as overloaded. Note that toggling this option can lead to import failures with test scripts that were generated before the option was toggled.

```
clicast -lc Option VCAST_SCRIPT_ALWAYS_USES_PARAMS True | False

This option causes C++ function parameter and return types to always be
```

```
included with the function name in test scripts. The default value is
False.
```

Strict Test Case Importing

When you import a test script into a VectorCAST environment, it is possible that there will be error in some of the script values. By default, VectorCAST discards the erroneous values and still creates the test case. If you select Strict Test Case Importing, VectorCAST marks the test case with a FAIL status if any error occurred when importing the script.

The status of the test case will be FAIL until you edit and save the test case in the Test Case Editor. If you export the test case to a test script before correcting the problem, “TEST.IMPORT_FAILED” is written to the test script for that test case.

```
clicast -lc option VCAST_STRICT_TEST_CASE_IMPORT true | false

Any errors encountered during test script import cause the test case's
status to be set to Fail. Its default value is FALSE.
```

Fail If No Expected Values

This option identifies test cases that have no Expected Values. When set to True, a test is prohibited from executing if it does not contain at least one Expected Value, Expected Parameter User Code, or Test Case User Code. It is marked as Failed in the Test Case Tree, and appears as "Abnormal Termination - No expected values" in the Testcase Management Report.

```
clicast -lc option VCAST_TESTCASE_FAIL_ON_NO_EXPECTED true | false

If this option is set, then test cases that have no expected results or
expected user code will be marked as failed. Its default value is FALSE.
```

Fail on Unexpected Exceptions

When checked, this option causes a test case to be marked as failed if an exception is thrown or signal is raised that was not expected.

```
clicast -lc option VCAST_UNEXPECTED_EXCEPTIONS_FAIL true | false

Sets test case status to Fail if an exception is raised during test case
execution but no exception was expected. Its default value is TRUE.
```

Automatically Clear Test User Code

If selected, this option tells VectorCAST to clear test user code prior to executing tests. Parameter user code for the user globals unit is automatically cleared as well. This option is helpful when you need to keep the executable size small.

```
clicast -lc option VCAST_AUTO_CLEAR_TEST_USER_CODE true | false

When set to TRUE, user code is cleared prior to executing tests. The
default value is False.
```

Expand Scalar Array Elements in Test Script

By default, when VectorCAST writes scalar array elements to a test script, it combines script lines when the values are identical. This option turns off this "collapse" capability.

```
clicast -lc option VCAST_SCRIPT_EXPAND_ARRAYS true | false

By default, when an array of scalars is written to a test script, identical
values are condensed to one script line. By setting this option, each array
element will get its own script line. The default value is False.
```

Detect Unused Expected User Code

When enabled, this option detects test-specific expected user code that has not been executed during a test. Such code is most likely to be associated with a stub that has not been called. When True and some testcase or parameter user code has not been executed, the test fails, and the Execution Report displays the unused values, the unit, and whether that unit is stubbed or not-stubbed.

Disable this option if you have added parameter user code to a stub that you expect to never get called. (You might add code to an uncalled stub if you want the test to show a failure if the stub ever gets called.)

```
clicast -lc option VCAST_DETECT_UNUSED_EXPECTED_UC true | false

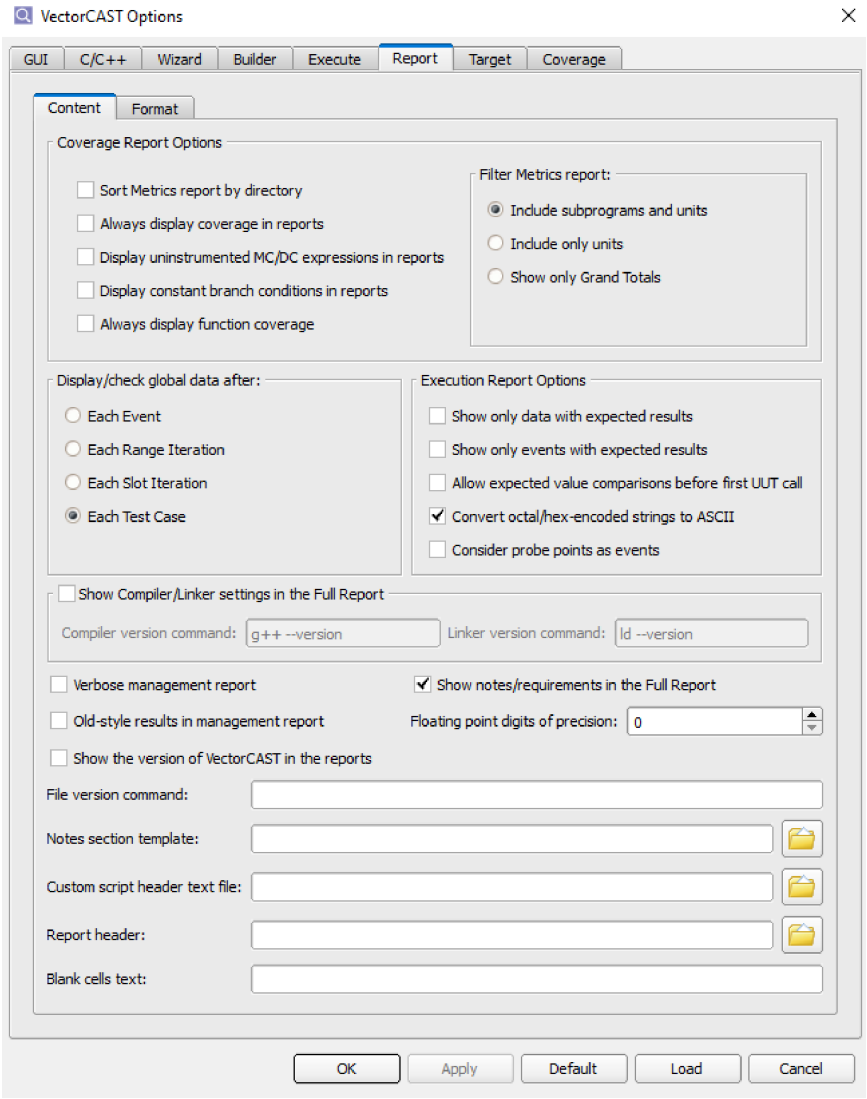
Set this option to True to detect when any test-specific user code that
checks expected values with the "{{...}}" syntax has not been executed. In
that case, the test will be flagged as having unused expected results. Such
code is most likely to be associated with a stub that has not been called.
Any test case or parameter user code, either Input or Expected, that uses
the conditional syntax "{{...}}" is considered when the option is True. Set
this option to False if you have added parameter user code to a stub that
you expect to never get called. (You might add code to an uncalled stub if
you want the test to show a failure if the stub ever gets called. The
default value is False.
```

Setting Report Options

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.



Coverage Report Options

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>

Display annotated source code in reports even if no run-time coverage data
exists. If this option is disabled, the message "No Coverage Data Exists"
will be displayed instead. The default value is False.
```

Display Uninstrumented MC/DC 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 MC/DC 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 MC/DC or Statement + MC/DC
- 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>

 When True, this option adds a 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 user may have turned off MC/DC instrumentation by setting options such
as "Instrument logical expressions in assignment statements" to 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.
```

Always Display Function Coverage

Extra column that reports if a function was entered.

A subprogram is considered covered in Function coverage if it is entered during test execution. Selecting this option will display a "Function Coverage" column in each report instance that includes the Metrics report (Full Report, Test Case Management Report and Aggregate Coverage Report).



Note: The configuration option VCAST_DISPLAY_FUNCTION_COVERAGE has no effect when used with the Function coverage type.

```
clicast -lc Option VCAST_DISPLAY_FUNCTION_COVERAGE <True | False>

Extra column that reports if a function was entered. This option is only
used if the coverage type is not Function. 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_METRICS
Date of Report Creation	01 FEB 2019
Time of Report Creation	1:10:34 PM

Metrics

Default Branch

Unit	Subprogram	Complexity	Branches
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 / 5 (40%)
	Place_Order	5	2 / 6 (33%)
	Clear_Table	2	0 / 3 (0%)
	Get_Check_Total	1	0 / 1 (0%)
	Add_Party_To_Waiting_List	3	0 / 5 (0%)
	Get_Next_Party_To_Be_Seated	2	0 / 3 (0%)
TOTALS	**6**	**16**	**4 / 23 (17%)**
manager_driver	main	8	0 / 9 (0%)
TOTALS	**1**	**8**	**0 / 9 (0%)**
whitebox	InitDay	1	0 / 1 (0%)
	InitColor	1	0 / 1 (0%)
	Initialize	1	0 / 1 (0%)
TOTALS	**3**	**3**	**0 / 3 (0%)**
GRAND TOTALS	**12**	**29**	**6 / 37 (16%)**

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_METRICS
Date of Report Creation	01 FEB 2019
Time of Report Creation	1:16:55 PM

Metrics

Default Branch

Unit	Subprogram	Complexity	Branches
database	2	2	2 / 2 (100%)
manager	6	16	4 / 23 (17%)
manager_driver	1	8	0 / 9 (0%)
whitebox	3	3	0 / 3 (0%)
GRAND TOTALS	12	29	6 / 37 (16%)

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_METRICS
Date of Report Creation	01 FEB 2019
Time of Report Creation	1:19:03 PM

Metrics

Default Branch

Unit	Subprogram	Complexity	Branches
GRAND TOTALS	12	29	6 / 37 (16%)

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 result to filter out of the Metrics Report. The default value
is No_Filtering.
```

Display/Check Global Data Report Options

Display/Check Global Data After

The Display Global Data After feature gives you control over when global data objects are displayed in the Test Execution report. If an expected value for the global object is set, then that expected value is compared to its actual value at the same time.

Ranging from displaying more frequently to less frequently, you can choose to display global data after:

- Each event
- Each range iteration
- Each slot iteration
- Each test case (default)

Each Event displays the current value(s) of global data objects at every transfer of control flow.

Each range iteration displays the current value(s) of global data objects after every iteration caused by a parameter's range of input values. If no parameter has an iteration, then the globals are displayed at the end of the test execution.

Each slot iteration displays the current value(s) at the end of each iteration of a test case in a compound test case. A compound can have one or more test cases in it, each with one or more iterations. This setting causes the globals to be displayed (and compared) when each test case finishes an iteration. If you apply this setting to a test case and then execute that test case alone (i.e. not in a compound), then the globals are displayed at the end of test execution, because it is the only "slot." In the example below, the global values would be displayed twice, at the end of each slot iteration.

Slot	Units	Subprograms	Test Cases	Iterations
1	manager	Get_Check_Total	Get_Check_Total.001	2

The default **Each test case** displays the current value(s) of global data at the end of a slot in a compound test case. A compound can have one or more test cases in it; this setting causes the globals to be displayed and compared when each test case in a compound finishes executing all of its iterations. If you apply this setting to a test case and then execute that test case alone (i.e. not in a compound), then the globals are displayed at the end of test execution, because it is the only "slot."

```
clicast -lc option GLOBALS_DISPLAY <Each_event|Range_Iteration|Slot_
Iteration|Testcase>

When to display and check global values in execution results: at each
event; at the end of each range iteration; at the end of each slot
iteration; at the end of each slot. Its default value is Testcase.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.GLOBAL_DATA_DISPLAY. Its default setting is the environment-wide setting for the Display global data option, accessible on the **Report** tab, **Content** sub-tab of the **Tools => Options** dialog.

Example of Display Global Data

Suppose you insert a test case for Get_Check_Total, from the tutorial environment. It is named GET_CHECK_TOTAL.001. Get_Check_Total only calls the stubbed function Get_Table_Record. To set up the example, enter the following Input and Expected values:

	Input	Expected
VCAST_USER_GLOBALS		
<<GLOBALS>>		
VECTORCAST_INT1	44	55
Get_Check_Total		
Table	**2..3/1**	

Execute the test case GET_CHECK_TOTAL.001. In the execution results, you see 6 events, three for the iteration in which Table is 2, and three for the iteration in which Table is 3. Because Each Test Case is the default setting, the global data value is displayed following Event 6, at the end of the test case.

The following table shows when the global data is displayed based on the setting of the “Display Global Data After” option. Global data is never compared to its expected value at the start of an iteration.

	Global Data Displayed After:			
	Each Event	**Each Range Iteration**	**Each Slot Iteration**	**Each Test Case**
Event 1: call UUT Range Iteration 1 (Table is 2)	Yes	No	No	No
Event 2: call stub	Yes	No	No	No
Event 3: return from UUT	Yes	Yes	No	No
Event 4: call UUT	Yes	No	No	No

Range Iteration 2 (Table is 3)				
Event 5: call stub	Yes	No	No	No
Event 6: return from UUT	Yes	Yes	Yes	Yes

Execution Report Options

Show Only Data with Expected Results

In some cases, a test may contain a huge number of global data input, but you may not want to capture all of this data in the Execution Reports. In the test case execution report, global and parameter data will only be displayed in the test execution report if expected values are provided; globals and parameters with input values only are not displayed.

```
clicast -lc option VCAST_SHOW_ONLY_DATA_WITH_EXPECTED_VALUES True | False

When this option is set, global and parameter data will only be displayed
in the test execution report if expected values are provided. This option
will result in smaller execution reports and faster test execution in cases
where there is a lot of global and / or parameter data input. The default
value is False.
```

```
TEST.VALUE:<<OPTIONS>>.SHOW_ONLY_DATA_WITH_EXPECTED_RESULTS:TRUE | FALSE

If specified in a test case script, this option causes only events that
have Expected values to be included in the Execution Report. It overrides
the environment-wide setting for "Show only events with expected results."
If not specified in a test script, the default value is the environment-
wide setting, accessible on the Report tab, Content sub-tab of the Tools =>
Options dialog.
```

Show Only Events with Expected Results

Setting this option causes VectorCAST to leave out any events in the Execution Report that do not have expected data for any parameter or global value. This option will result in smaller execution reports in cases where there is a lot of global and /or parameter data input.

```
clicast -lc option VCAST_SHOW_ONLY_EVENTS_WITH_EXPECTED_VALUES True | False

When this option is set, events will only be displayed in the test
execution report if expected values exist for that event. This option will
result in smaller execution reports in cases where there is a lot of global
and / or parameter data input. The default value is False.
```

```
TEST.VALUE:<<OPTIONS>>.SHOW_ONLY_EVENTS_WITH_EXPECTED_RESULTS:TRUE | FALSE

If specified in a test case script, this option causes only events that
```

```
have Expected values to be included in the Execution Report. It overrides
the environment-wide setting for "Show only events with expected results."
If not specified in a test script, the default value is the environment-
wide setting, accessible on the Report tab, Content sub-tab of the Tools =>
Options dialog.
```

Allow Expected Value Comparisons Before First UUT Call

This option is used to cause VectorCAST to make comparisons of expected values to actual values before the UUT is even called. It is useful if a stub is called before you get to the UUT call (from user code in an <<INIT>> test case, for example), and you want to verify that a value was passed in correctly. By default, this option is false. Set it to true if you want to compare expected values before the UUT call.

```
clicast -lc option VCAST_EXPECTED_BEFORE_UUT_CALL True | False

Setting this option enables comparisons between expected values and actual
values to occur before the call to the UUT. The default value is False.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.COMPARE_BEFORE_UUT. Its default value is the environment-wide setting for Compare Expected Values Before UUT Call, accessible on the **Report** tab, **Content** sub-tab of the **Tools => Options** dialog.

Convert Octal / Hex-Encoded Strings to ASCII

By default, strings with unprintable characters are displayed in octal in the Execution report. To display these strings in hex, select **Tools => Options = Execute tab** and set the option "**Hex notation for unprintable chars**."

Setting the option, "**Convert octal/hex-encoded strings to ASCII**" under Execution Report Options causes the printable portions of these strings to be displayed in ASCII in the Execution report, while the unprintable characters remain in octal or hex.

```
clicast -lc option VCAST_CONVERT_OCTAL_HEX_STRINGS_TO_ASCII True | False.

This option is used to determine whether or not to convert "unprintable"
strings encoded in octal or hexidecimal notation to ASCII encoding in the
report output. It doesn not affect the actual string comparisons in the
test execution. Its default value is TRUE.
```

Consider Probe Points as Events

This option directs VectorCAST to include a probe point call in the Execution Report as a separate event. For more information, see "Probe Point Events" on page 567.

```
clicast -lc option VCAST_PROBE_POINTS_AS_EVENTS True | False

Set this option to have an event generated at each probe point. The default
```

```
value is False.
```

Show Compiler/Linker Settings in the Full Report

Setting this option causes VectorCAST to include the compiler and linker settings that were used to build the test harness in the Full Report. This information is taken from the **compiler settings** on the **C/C++ tab** in the **Options** dialog.

The settings included are:

- Compiler name
- Compiler template (the compiler tag)
- Execute command
- Preprocess command
- Compile command
- Include flag
- Syntax only flag
- Define flag
- Linker command
- Linker options
- Linker output file flag

```
clicast -lc option VCAST_COMPILER_TEMPLATE_SECTION <True | False>

Include the compiler and linker settings used to compile the test harness
in the Full report. The default value is false.
```

Report Options

Verbose Management Report

When True, this option copies the contents of the 'Test Notes' box in the Test Case Editor to the Test Case Management Report. One possible use for this is for test requirement numbers to be correlated to test cases in the Test Case 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
```

Old-style Results in Management Report

When this option is set to True, the Test Case Management Report uses the "old-style" pass/fail results in the main table of the report. The Overall Results section does not change.

The "old-style" Test Case Management Report combined the Expected Values matched with the

Control Flow results, signals, exceptions, and terminations in the Pass/Fail column. Enable this option to once again generate the Test Case Management Report with the data displayed in this way.

```
clicast -lc option VCAST_OLD_STYLE_MANAGEMENT_REPORT <True | False>

When set, this option reverts the Test Case Management Report to the
behavior found in VectorCAST version 6.0 and prior, in which the main body
of the report combined the Expecteds and Control Flow in one fraction in
the "Pass/Fail" column. The 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
```

Show Notes / Requirements in the Full Report

By default, the Notes / Requirements section is included in the Full Report. When cleared, the option "Show notes / requirements in the Full Report" excludes the Notes / Requirements in the Full Report. Note that the section is always included in the Test Case Data Report.

```
clicast -lc  Option SHOW_NOTES_IN_FULL_REPORT True | False

This option will add a section to the Full Report listing test notes and
requirements. The default value is True.
```

Floating Point Digits of Precision

This value controls exactly how the decimal part of a floating point number will be displayed. It modifies the amount of precision that VectorCAST will use to print a floating point number.

The floating point format is determined as follows: If the value of the floating point precision is n, then the **`sprintf`** format string that VectorCAST uses is %.nlg, where 0 < n <= 17. If n is 0 then VectorCAST simply uses %lg.

```
clicast -lc option VCAST_FLOAT_PRECISION <integer number>

Controls the number of digits used when the test harness prints floating
point Actual Values for the Execution Report or Range Data. This option's
setting is significant in the comparison of Expected Values for floating
point numbers as well as determining the min and max values.

In a C/C++ environment, a non-zero precision value causes floating point
numbers to print in the report with no more than the given number of
significant digits, and a zero precision value usually results in a maximum
of 6 digits.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.FLOAT_POINT_DIGITS_OF_PRECISION. Its default value is the environment-wide setting for Floating Point Digits of Precision, accessible on the **Reports** tab of the **Tools => Options** dialog.

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 -lc 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, Parameter User Code, and Test Case User Code, 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 import the template at the cursor location in a User Code editor, first click the checkbox next to Enable. Right-click in the blank text area and choose **Import template**. The text from the template file appears in the text area.

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 -lc 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.
```

Custom Script Header Text File

This option specifies a text file containing text you want VectorCAST to place at the top of exported test scripts. Use the full path to the file, or a path relative to the environment directory (not working directory). In the exported test script, the text is preceded with a comment marker (--) automatically, in order to retain the custom text when the environment is rebuilt or updated.

```
clicast -lc option VCAST_DEFAULT_SCRIPT_HEADER <Path to Template text file>

This option allows you to enter a path to a text file that contains a
custom header for all generated test scripts.
```

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>
```

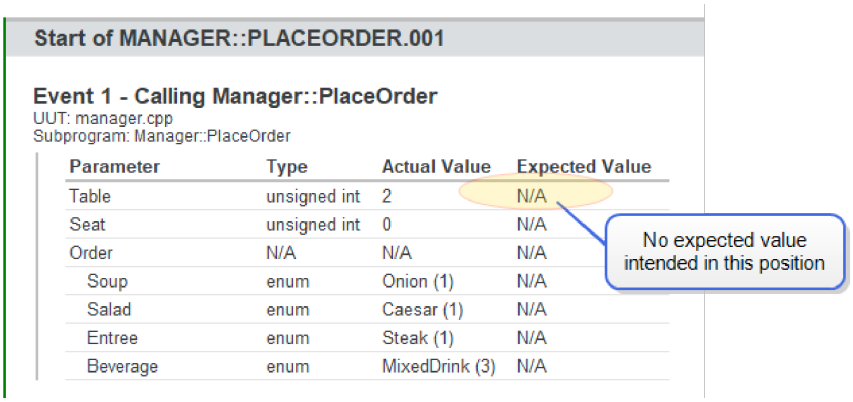
```
clicast -lc option VCAST_RPTS_HEADER <text to prepend to title> | <header
text> | <html file>

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.
```

Blank Cells Text

This option directs VectorCAST to include a text string in the Execution Report and Test Case Data Report, as well as those sections in the Full Report, in place of empty cells in the Input and Expected Value tables.

For example, use the text "N/A" for blank cells in reports to indicate that no Input Value or Expected Value is intended in this position.



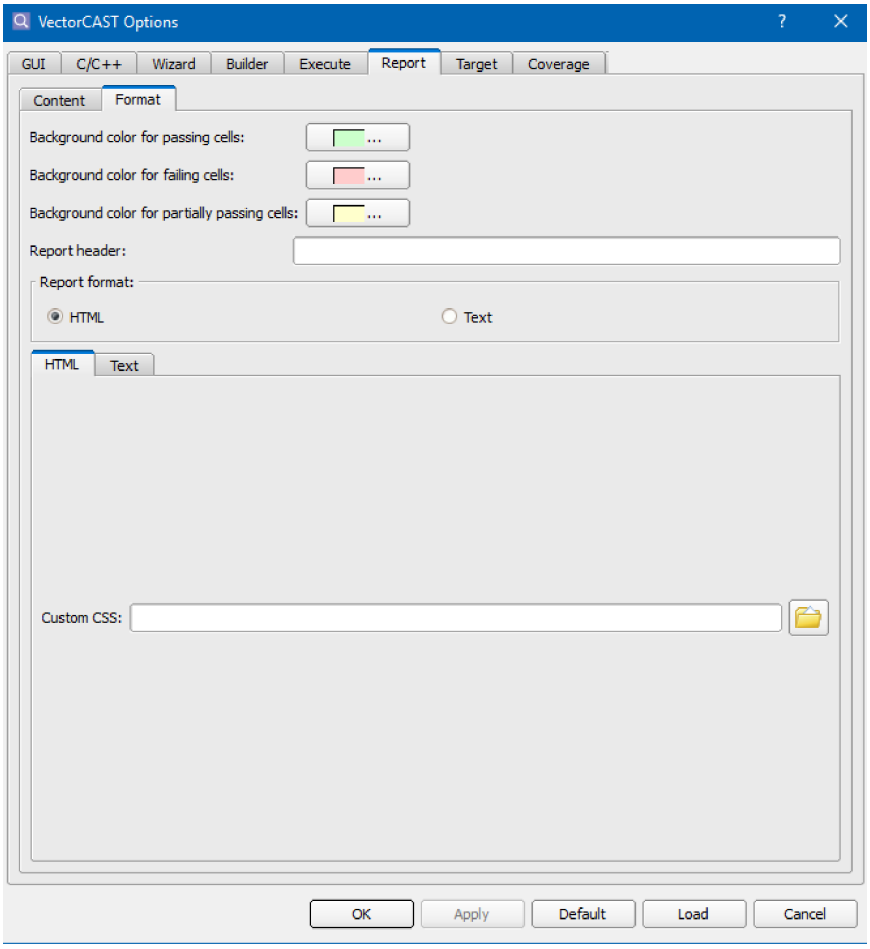

```
clicast -lc option VCAST_RPTS_EMPTY_DATA_STRING <string>

Text to show in blank cells in the Execution and Test Case Data report
sections. The default value is unset.
```

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 -lc 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 -lc 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 -lc 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**.

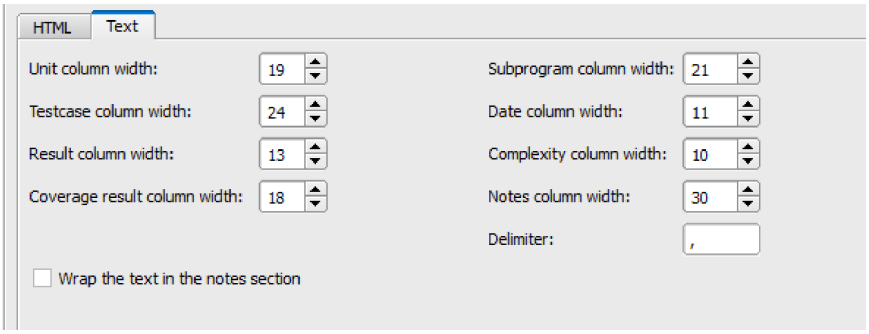
```
clicast -lc option VCAST_RPTS_CUSTOM_CSS <path to custom.css file

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 -lc 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 -lc 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 -lc 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 -lc 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 -lc 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 -lc 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 -lc 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 -lc 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 r**`eports alternate`**. Enter the character in quotes, as in “@”. To enter a tab, use “\t”. Valid delimiters are: ? , ' ; | { } [ ] @ ~ # $ _ \t \n

```
clicast -lc 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 -lc 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.
```

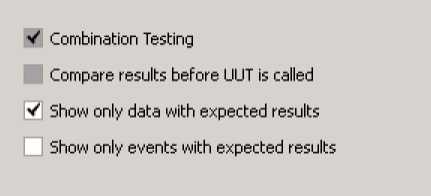
Setting Test Case Options

The Testcase Options Tab

Most of the options that are available in the Testcase Options tab are also available on the **Tools => Options dialog, Execute tab**. The environment-level settings are “inherited” by each test case, but can be over-ridden by setting a testcase-specific option here.

The states of a test case option are indicated below:

- Gray checkbox with a check – option is enabled and it is inherited.
- Gray checkbox with no check – option is disabled and it is inherited.
- White checkbox with a check – option is enabled and overrides inherited setting.
- White checkbox with no check – option is disabled and overrides inherited setting.



Return Data List Options

By default, the test harness resets the Input Values list returned from a stub immediately prior to calling the UUT. For example, if you are testing a UUT by varying a parameter between 1 and 100 by one, and the UUT calls a stubbed subprogram once per execution, the same data value will be returned from the stubbed subprogram each time, even if a list of values is provided. The multi-return list is reset to the first item at the start of each invocation of the unit under test. If the stub is called multiple times by the UUT, then multiple values will be returned.

Multi-returns Span Range Iterations

By choosing this option, the Input Values list or range provided for a stub will not be reset at the start of each range iteration. Instead, the list or range spans the range iterations in a UUT’s parameter.

In a test case script, this option is specified as TEST.VALUE:<<OPTIONS>>.MULTI_RETURN_SPANS_RANGE. Its default value is FALSE.

Multi-return Spans Compound Iterations

By choosing this option, the Input Values list or range provided for a stub will not be reset at the start of each iteration of a slot in a compound test case. Instead, the list or range spans iterations of a test

case in a slot of a compound test. If you turn on this option, you must also turn on “Multi-returns span range iterations.”

In a test case script, this option is specified as TEST.VALUE:<<OPTIONS>>.MULTI_RETURN_SPANS_TESTCASES. Its default value is FALSE.

Report Data Options

Display Full String Data

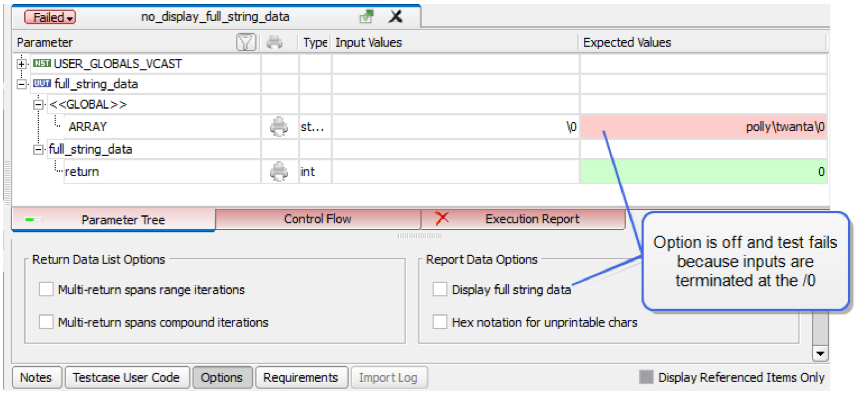
When this option is off (default), strings in the Input Values column are displayed up to the null terminator (\0) in the Execution report. During execution, the value of the parameter is the string up to the null terminator.

Note: An Expected Value is not affected by this option. An Expected Value is always displayed and compared in its entirety.

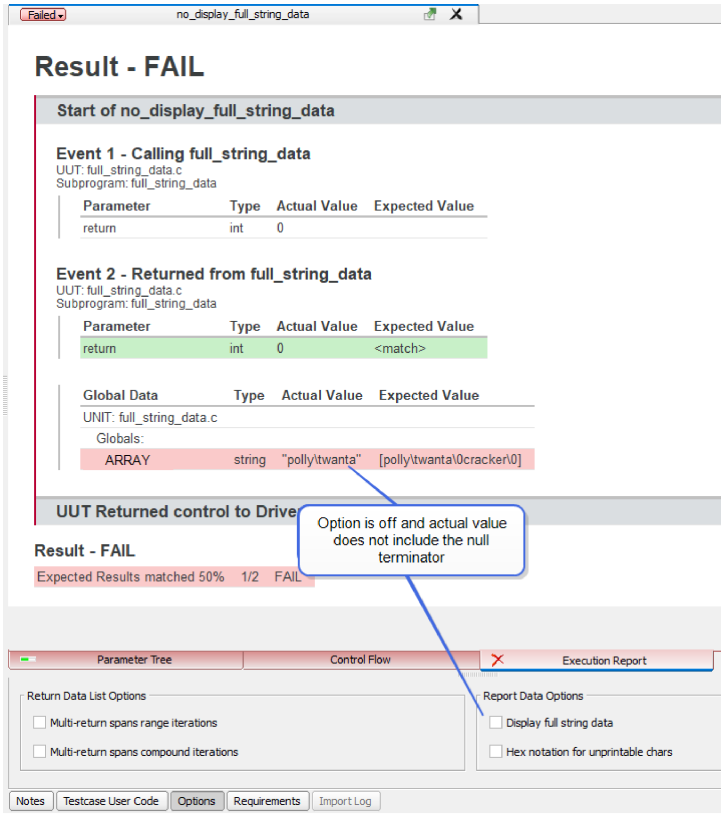
In the example shown below, a string global is defined:

```
char ARRAY[20];
```

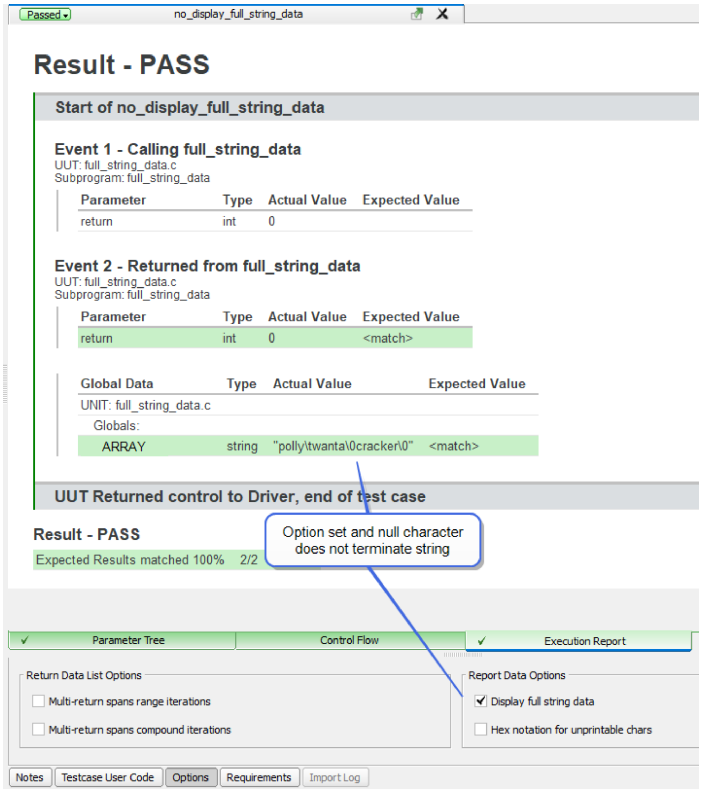
The code for this example sets the characters in the array and includes the unprintable characters \t and \0, the null terminator. When the option is off, these inputs are terminated at the \0. To pass, the Expected Value will only be compared until the null character.



The Execution Report, with the Display full string data option off, shows the Actual Value string up to the null terminator and shows that the Expected Value does not match the Actual Value.



When the Display full string data option is set, all elements of the array are displayed, including non-printable characters. As a result, the entire array is shown; the null character does not terminate the string.



Use the option Hex notation for unprintable chars to change the output to hexadecimal representation.

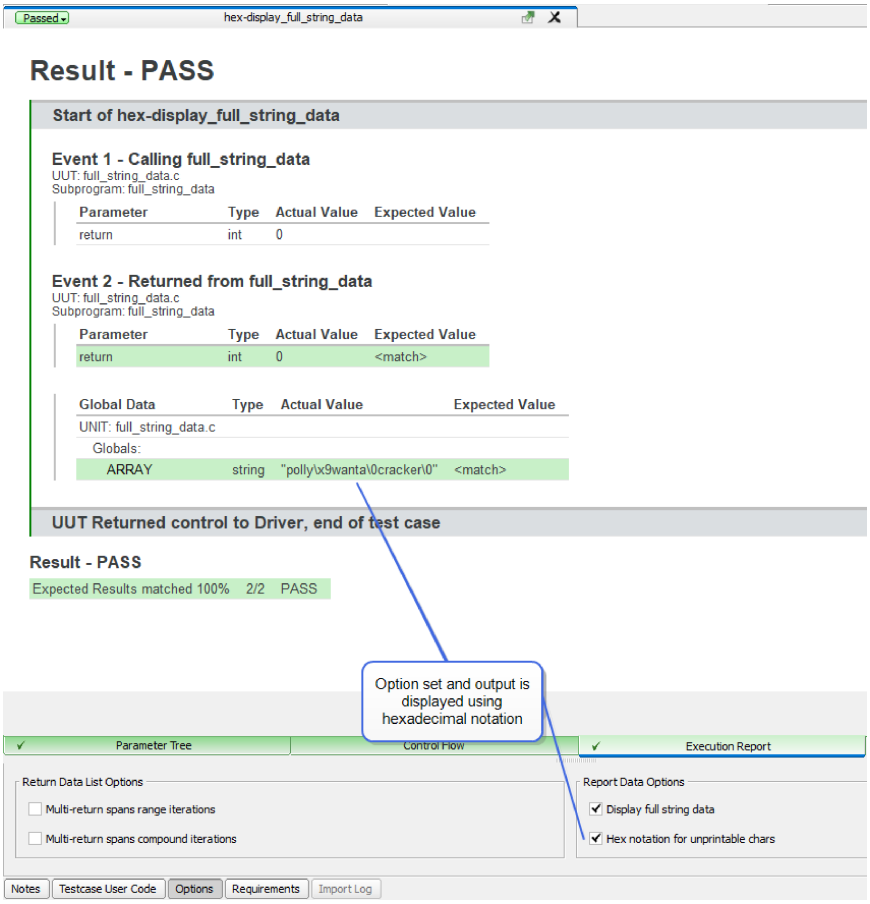
```
clicast -lc option VCAST_FULL_STRINGS True | False

When this option is set, all elements of a character array (including non-
printable characters) are displayed using their numeric representation. As
a result, the entire array will be shown (the NULL character does not
terminate the string). Use the VCAST_HEX_NOTATION option to choose between
octal and hexadecimal representation. Note: This option also affects how
strings are compared during test case execution. Its default value is
FALSE.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.DISPLAY_FULL_STRING_DATA. Its default value is FALSE.

Hex Notation for Unprintable Chars

If a string contains a non-printable character (excluding NULL), those characters are displayed using octal numeric representation. If this option is set, the characters will be displayed using hexadecimal notation. To display the terminating NULL character and all following elements of a character array, use the Display Full String Data option.




```
clicast -lc option VCAST_HEX_NOTATION true | false

Display non-printable characters using hexadecimal notation. Its default
value is FALSE.
```

In a test case script, this option is:

```
TEST.VALUE:<<OPTIONS>>.HEX_NOTATION_FOR_UNPRINTABLE_CHARS

Testcase-specific value for "Hex Notation for Unprintable Chars." Its
default value is the environment-wide setting for "Hex notation for
unprintable chars," accessible on the Execute tab of the Tools => Options
dialog.
```

Other Test Case Options

Floating Point Digits of Precision

This value controls exactly how the decimal part of a floating point number will be displayed. It modifies the amount of precision that VectorCAST will use to print a floating point number.

The floating point format is determined as follows: If the value of the floating point precision is n, then

the **sprintf** format string that VectorCAST uses is %.nlg, where 0 < n <= 17. If n is 0 then VectorCAST simply uses %lg.

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.FLOAT_POINT_DIGITS_OF_PRECISION. Its default value is the environment-wide setting for Floating Point Digits of Precision, accessible on the Reports tab of the **Tools => Options** dialog.

Floating Point Tolerance (%)

The Floating Point Tolerance option enables you to set the tolerance of expected results for floating point numbers. This tolerance applies to all test case data in the environment.



Note: There is no tolerance around the value 0.

```
clicast -lc option FLOATING_POINT_TOLERANCE <tolerance>

Percentage that a floating point actual value can vary from its expected
value and still be considered a match. <percentage> is a floating point
number. Its default value is 0.000000.
```

In a test script, the testcase-wide floating point tolerance is specified using:

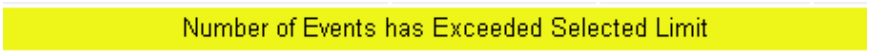
TEST.FLOATING_POINT_TOLERANCE:<tolerance>



Note: For very small tolerances, the floating point representation of the value may not be exact due to the resolution of floating point numbers; in this case, the floating point tolerance may show more digits than the user expects. For example, a tolerance of 0.000000001 may actually be displayed as 0.00000000099999.

Event Limit

The Event Limit option enables you to control the maximum number of events per test harness execution. The event limit defaults to 1000. If this option is set to 0, no results data is captured, although coverage data is captured. If the event limit is exceeded, VectorCAST automatically stops test execution if your compiler supports signals, and the execution report shows:



The Execution Report shows <limit>+1 events.

The event limit applies to each test case. Each time a test executes, it resets the event counter to 1. If you execute several test cases at once using Ctrl+Shift click to select them, or execute them all by selecting the environment name and choosing **Test => Execute**, the counter starts over with each test case (because the test harness is being executed multiple times).

Note: An individual test case can have its own event limit, which overrides the environment event limit when that test case is executed. This option is accessed via the **Options tab** in the **Test Case Editor**.

When executing a compound test case, the event limit is not reset for each slot; the Event Limit setting applies to the Compound Test Case as a whole.

Note: The Event Limit feature is implemented through the raising of signals. If your target does not support signals, the event limit capability will not be available.

```
clicast -lc option EVENT_LIMIT <limit>

Maximum number of events for the harness to process. If this option is set
to 0, no results data is captured, although coverage data is captured. Its
default value is 1000.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.EVENT_LIMIT. Its default value is the environment-wide setting for Event Limit, accessible on the Execute tab of the **Tools => Options** dialog.

The harness event limit data type can be changed by setting the macro **VCAST_EVENT_LIMIT_TYPE** located in **Tools => Options => C/C++ => Defined variables**. For example, specify **VCAST_EVENT_LIMIT_TYPE=long** to allow a larger event limit. The default type for **VCAST_EVENT_LIMIT_TYPE** is **unsigned int**.

```
clicast -lc option C_DEFINE_LIST VCAST_EVENT_LIMIT_TYPE=<type>

where <type> is one of: long, short, or char. The default type is unsigned
int, achieved by removing the VCAST_EVENT_LIMIT_TYPE define from the
configuration.
```

See also "Key Terminology" on page 17 for an explanation of an event.

Number of Data Partitions

This option specifies the number of iterations to span an entire range of a scalar parameter. It is used in conjunction with Range Values to allow a specification of iterations instead of using a range delta.

```
clicast -lc Option NUMBER_OF_DATA_PARTITIONS <positive number>

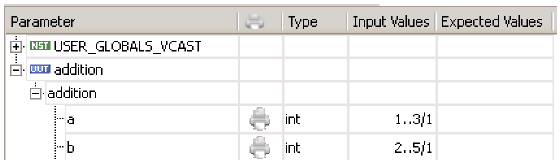
Number of iterations to span an entire range of a scalar parameter.  Used
with Range Values to allow specification of iterations instead of range
delta. The default value is 1 partition.
```

Combination Testing

If a test case has a parameter with an input range or list, then the test harness executes the test case once for each iteration. When two or more parameters have a range iteration, then each parameter's iteration count is incremented at the same time, by default.

Combination testing can be turned on for an individual test case, using Test Case Options, or for the whole environment, using the **Tools => Options** dialog, Execute tab. If combination testing is enabled, VectorCAST will determine the combinations of input values and use those values as stimulus values.

For example, if parameter A has a range 1 to 3 with a delta of 1 and parameter B has a range 2 to 5 with a delta of 1, and Combination Testing is off, then the test case executes this way:



Range Iteration #1:

A is 1

B is 2

Range Iteration #2:

A is 2

B is 3

Range Iteration #3:

A is 3

B is 4

Range Iteration #4:

A is 3 again

B is 5

If the combination testing option is enabled, then the test case executes this way:

Range Iteration #1:

A is 1

B is 2

Range Iteration #2:

A is 1

B is 3

Range Iteration #3:

A is 1

B is 4

Range Iteration #4:

A is 1

B is 5

Range Iteration #5:

A is 2

B is 2

Range Iteration #6:

A is 2

B is 3

Range Iteration #7:

A is 2

B is 4

Range Iteration #8:

A is 2

B is 5

Range Iteration #9:

A is 3

B is 2

Range Iteration #10:

A is 3

B is 3

Range Iteration #11:

A is 3

B is 4

Range Iteration #12:

A is 3

B is 5

This feature is not available with an environment built with VectorCAST version 3.2 or earlier. Rebuilding the environment will enable this feature's functionality.

```
clicast -lc option VCAST_DO_COMBINATION true | false

Use all combinations of the values in the range expressions for test
stimulus values. Its default value is FALSE.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.DO_COMBINATION. Its default

value is the environment-wide setting for Combination testing, accessible on the Execute tab of the Tools => Options dialog.

Compare Results Before UUT is Called

This option is used to cause VectorCAST to make comparisons of expected values to actual values before the UUT is even called. It is useful if a stub is called before you get to the UUT call (from user code in an <<INIT>> test case, for example), and you want to verify that a value was passed in correctly. By default, this option is false. Set it to true if you want to compare expected values before the UUT call.

```
clicast -lc option VCAST_EXPECTED_BEFORE_UUT_CALL <True | False>

Setting this option enabled comparisons between expected values and actual
values to occur before the call to the UUT. Its default value is FALSE.
```

In a test case script, this option is TEST.VALUE:<<OPTIONS>>.COMPARE_BEFORE_UUT. Its default value is the environment-wide setting for Compare Expected Values Before UUT Call, accessible on the Reports tab of the **Tools => Options** dialog.

Show Only Data with Expected Results

In some cases, a test may contain a huge number of global data input, but you may not want to capture all of this data in the Execution Reports. In the test case execution report, global and parameter data will only be displayed in the test execution report if expected values are provided; globals and parameters with input values only are not displayed.

```
clicast -lc option VCAST_SHOW_ONLY_DATA_WITH_EXPECTED_VALUES True | False

When this option is set, global and parameter data will only be displayed
in the test execution report if expected values are provided. This option
will result in smaller execution reports and faster test execution in cases
where there is a lot of global and / or parameter data input. The default
value is False.
```

```
TEST.VALUE:<<OPTIONS>>.SHOW_ONLY_DATA_WITH_EXPECTED_RESULTS:TRUE | FALSE

If specified in a test case script, this option causes only events that
have Expected values to be included in the Execution Report. It overrides
the environment-wide setting for "Show only events with expected results."
If not specified in a test script, the default value is the environment-
wide setting, accessible on the Report tab, Content sub-tab of the Tools =>
Options dialog.
```

Show Only Events with Expected Results

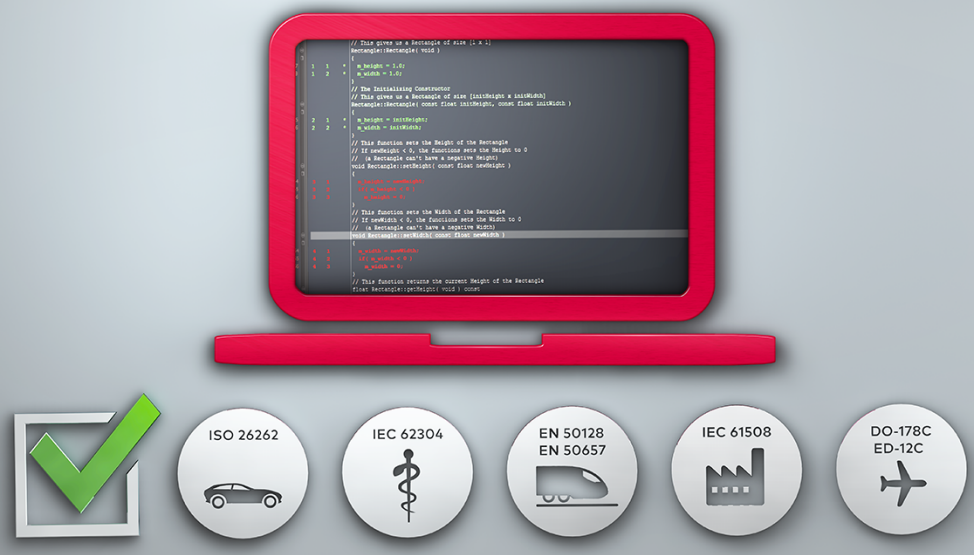
Setting this option causes VectorCAST to leave out any events in the Execution Report that do not have expected data for any parameter or global value. This option will result in smaller execution reports in cases where there is a lot of global and /or parameter data input.

clicast -lc option VCAST_SHOW_ONLY_EVENTS_WITH_EXPECTED_VALUES True | False

When this option is set, events will only be displayed in the test execution report if expected values exist for that event. This option will result in smaller execution reports in cases where there is a lot of global and / or parameter data input. The default value is False.

TEST.VALUE:<<OPTIONS>>.SHOW_ONLY_EVENTS_WITH_EXPECTED_RESULTS:TRUE | FALSE

If specified in a test case script, this option causes only events that have Expected values to be included in the Execution Report. It overrides the environment-wide setting for "Show only events with expected results." If not specified in a test script, the default value is the environment-wide setting, accessible on the **Report** tab, **Content** sub-tab of the **Tools => Options** dialog.



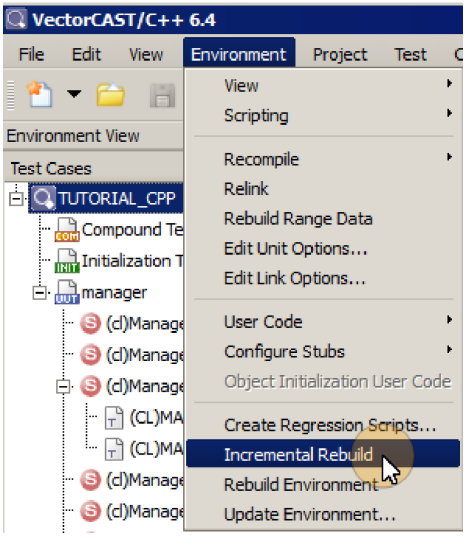
Incremental Rebuild

Performing an Incremental Rebuild

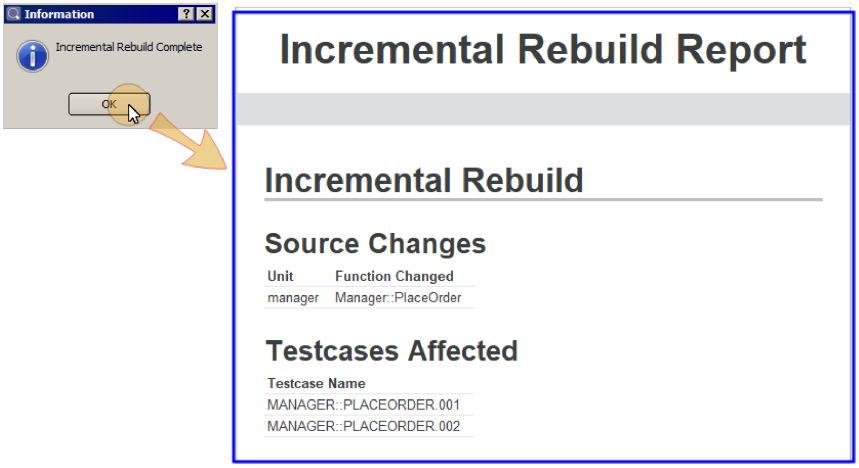
The Incremental Rebuild feature allows you to incorporate source code changes into the test harness without having to fully rebuild the environment. The environment must have coverage instrumented and tests executed in order for VectorCAST to be able to detect which subprograms have been modified and which test cases have been affected.

If the environment does not have coverage or is blackbox, a full rebuild is required because the changes are considered global in scope.

To perform an incremental rebuild, first make any modifications to a UUT or non-stubbed dependent. Next, save the changes. Then select **Environment => Incremental Rebuild** from the Menu Bar.



Upon selection of the Incremental Rebuild option, if VectorCAST detects a change to UUTs or non-stubbed dependents, it determines which units or header files have changed, updates the associated portions of the test harness, and recompiles and then re-instruments those portions. A pop-up message is displayed indicating the incremental rebuild is complete. Select the **OK** button and the Incremental Rebuild Report is displayed in the MDI window.

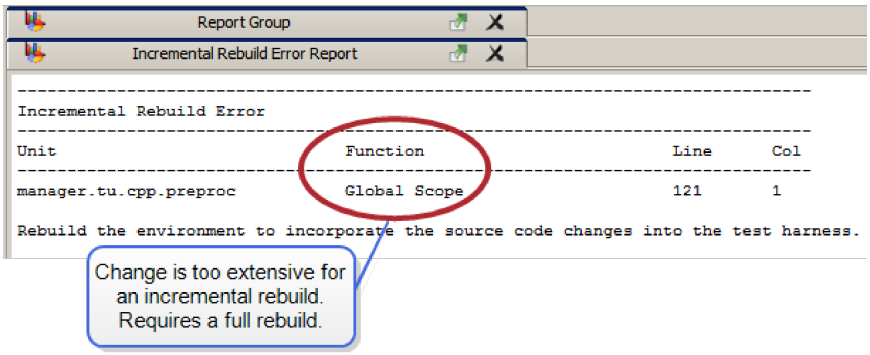


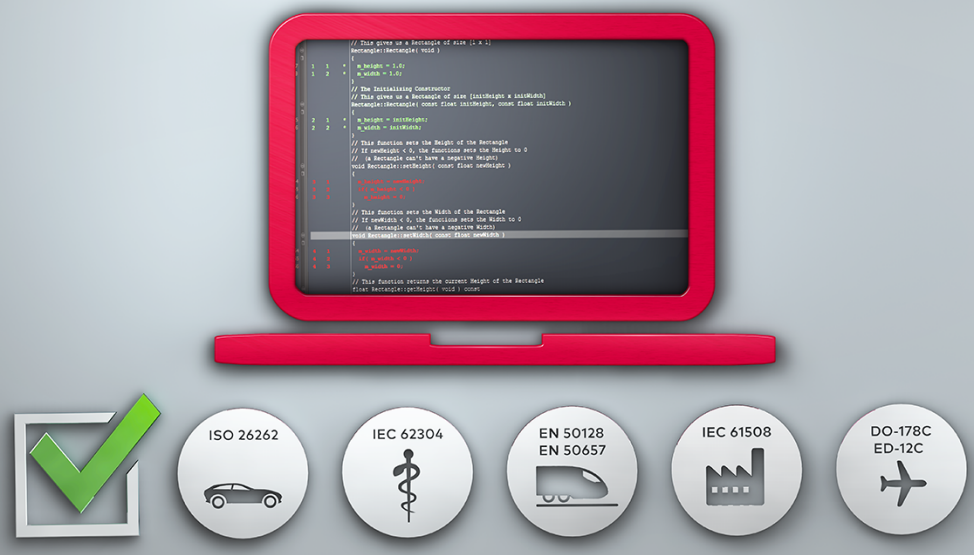
The Incremental Rebuild Report indicates which units were modified and which subprograms were affected by the change. If execution results exist, the report also indicates which test cases were affected.

Execution and coverage status are selectively removed following an incremental rebuild. Assuming the environment has coverage instrumented, the test cases that call the modified function (and those whose non-stubbed subprograms call the modified function) are considered "affected," and their coverage data and execution data are removed from the environment. Unaffected test cases retain their execution and coverage data. If the environment does not have coverage instrumented, then all execution data is removed and the environment is fully rebuilt.

Note: The Parameter Tree does not reflect changes to the source code after an Incremental Rebuild; only the test harness is affected. To see changes such as renaming an enumerated value, changing a parameter name, etc. reflected in the Parameter Tree, you must fully rebuild the environment.

Global source code changes require a full rebuild. If a source code modification adds a new dependency, such as a new function reference, extern reference, or adding a static class member, when incremental rebuild is initiated the user is notified that the change is too large to be rebuilt incrementally. The Incremental Rebuild Error Report prompts the user to do a full rebuild of the environment to incorporate the source code changes into the test harness.





Using Code Coverage

Code Coverage

The Code Coverage tool determines which lines of source code (statement), which branches of source code (branch), or which equivalence pairs (MC/DC) have been executed by one or more sets of test case data. The reports show you the completeness of your test suite. By analyzing the untested code, you can easily work backward designing test cases to test these portions of your unit.

VectorCAST initializes the following coverage types:

- **Statement** - reports on which executable lines of source code are being executed as the program runs.
- **Branch** - displays which outcomes have occurred for each branch point in the program.
- **MC/DC** - 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.
- **Statement + MC/DC** - reports on which executable lines of source code are being executed as the program runs and on the outcomes of all Boolean expressions, and reports on the values of all Boolean components of a complex conditional.
- **Function Coverage** - provides a boolean indicator of whether or not the entry point of a function was invoked.
- **Function + Function Call Coverage** - indicates whether or not a subprogram in the unit under test has been entered during test execution.
- **Statement + Branch** - reports on which executable lines of source code are being executed as the program runs and displays which outcomes have occurred for each branch point in the program.
- **Basis Paths** - identifies all possible paths of execution and consists of two parts: the list of basis paths and an associated annotated source listing.

The current Industry Mode (see “To Set The Industry Mode” in Section 1) determines the set of coverage choices. If no Industry Mode is set, Default is used. The following is a list of coverage types listed by Industry Mode:

- Avionics (DO-178 B/C)
  - Level A (Statement, Branch, MC/DC),
  - Level B (Statement, Branch)
  - Level C (Statement)
- Automotive (ISO-26262)
  - Architectural Level ASIL A/B (Function)
  - Architectural Level ASIL C/D (Function, Function Call)
  - Unit Level ASIL A (Statement)
  - Unit Level ASIL B/C (Statement, Branch)
  - Unit Level ASIL D (Statement, Branch, MC/DC)
- Industrial (IEC-61508)
  - SIL4 (Statement, Branch, MC/DC)
  - SIL3 (Statement, Branch)
  - SIL 1/2 (Statement)

- Railway (EN-50128)
  - SIL4 (Statement, Branch, MC/DC),
  - SIL3 (Statement, Branch)
  - SIL 1/2 (Statement)
- Medical (IEC-62304)
  - Class C (Statement, Branch, MC/DC)
  - Class B (Statement, Branch)
  - Class A (Statement)
- Default
  - Statement
  - Branch
  - Basis Paths
  - MC/DC
  - Function
  - Function + Function Call
  - Statement + Branch
  - Statement + MC/DC

Initializing a type of coverage results in the instrumentation, compilation, and linking of an instrumented version of the test harness which enables VectorCAST to monitor coverage during the execution of test cases.



Note: Because code coverage runs in parallel with normal VectorCAST functionality, test cases run with coverage ON can always be re-run on the non-instrumented test harness to verify real-time performance.

To Initialize 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 initialize Statement coverage, select a source file and choose **Coverage => Initialize =>**

Statement.

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 initialize Statement coverage when using Industry Modes, choose **Coverage => Initialize => *<appropriate type>*** based on the current Industry Mode.

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> TOols Cover Statement [ <unitname> | ALL]

Initialize coverage instrumentation of all UUT(s) in  the environment for
Statement coverage. Similar to Initialize Custom, the optional argument
<unitname> is the name of a non-stubbed unit to be instrumented in addition
to the UUTs, or "ALL" to instrument all non-stubbed units in the
environment as well.
```

To Initialize 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:

```
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 initialize Branch coverage, choose **Coverage => Initialize => Branch**. Once VectorCAST has completed instrumenting the test harness, the following message is displayed:

```
Completed coverage instrumentation processing
```

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



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

clicast -e <env> TOols Cover Branch [<unitname> | ALL]

Initialize instrumentation of all UUT(s) in the environment for Branch coverage. Similar to Initialize Custom, the optional argument <unitname> is the name of a non-stubbed unit to be instrumented in addition to the UUTs, or “ALL” to instrument all non-stubbed units in the environment as well.

To Initialize 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.

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

Pairs Status

When MC/DC coverage is initialized, an additional “Pairs” metric is available in the coverage Metrics Report, **Test => View => Metrics Report** and the Test Case Management Report, **Test => View => Test Case 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.

To initialize MC/DC coverage, select a source file, and choose **Coverage => Initialize => MC/DC**.

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 Initialization on Compile Error

In some rare cases, VectorCAST cannot successfully instrument MC/DC coverage on an expression in your source code. As a workaround (be sure to contact Technical Support), 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. 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. There can be no spaces after the comment characters.

For example:

```
//VCAST_DONT_DO_MCDC
or
/*VCAST_DONT_DO_MCDC*/
Specific statement to which you do not want MC/DC coverage applied

//VCAST_DO_MCDC
or
/*VCAST_DO_MCDC*/
Specific statement to which you DO want MC/DC coverage applied
```

To Initialize 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 initialize Statement+MC/DC coverage when using Industry Modes, choose **Coverage => Initialize => *<appropriate type>*** based on the current Industry Mode.

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> Tools COVER STATEMENT+MC/DC [<unitname> | ALL]

Initialize instrumentation of all UUT(s) in the environment for
STATEMENT+MC/DC coverage. The optional argument <unitname> is the name of a
non-stubbed unit to be instrumented in addition to the UUTs, or "ALL" to
instrument non-stubbed units in the environment as well.
```

To Initialize Function Coverage

Function coverage indicates whether or not a subprogram in the unit under test has been entered during test execution. The Function coverage type is especially suitable for the ISO-26262 Industry mode.

Function coverage is a boolean indicator of whether or not the entry point of a function was invoked. A subprogram is considered covered 100% in Function coverage if it is entered during test execution, and 0% if it is not.

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 initialize Function coverage, choose **Coverage => Initialize => Function**. Once VectorCAST has completed instrumenting the test harness, the following message is displayed:

```
Completed coverage instrumentation processing
```

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

```
clicast -e <unit test env> TOols COVer FUNCTION [<unitname> | ALL]

Initialize instrumentation of all UUT(s) in the environment for Function
coverage. Similar to Initialize Custom, the optional argument <unitname> is
the name of a non-stubbed unit to be instrumented in addition to the UUTs,
or "ALL" to instrument all non-stubbed units in the environment as well.
```

When using Function coverage with a vcshell database, use one of these commands after creating the database and setting the compiler template:

- For Stub None (Object Files) environments:
  `vcutil parse --instrument function`
- For parallel instrumentation of a Cover environment:
  `vcutil instrument --coverage_type function`



Note: The configuration option VCAST_DISPLAY_FUNCTION_COVERAGE has no effect when used with the Function coverage type.

To Initialize Function + Function Call Coverage

Function coverage indicates whether or not a subprogram in the unit under test has been entered during test execution. Function Call coverage identifies all of the calls that each function makes to other functions and reports on whether those calls have been tested. The Function +Function Call coverage type is especially suitable for the ISO-26262 Industry mode.

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 initialize Function + Function Call coverage, choose **Coverage => Initialize => Function + Function Call**. Once VectorCAST has completed instrumenting the test harness, the following

message is displayed:

```
Completed coverage instrumentation processing
```

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

```
clicast -e <unit test env> TOols COVer FUNCTION+FUNCTION_CALL [<unitname> |
ALL]

Initialize instrumentation of all UUT(s) in the environment for Function +
Function Call coverage. Similar to Initialize Custom, the optional argument
<unitname> is the name of a non-stubbed unit to be instrumented in addition
to the UUTs, or "ALL" to instrument all non-stubbed units in the
environment as well.
```

When using Function + Function Call coverage with a vcshell database, use one of these commands after creating the database and setting the compiler template:

- For Stub None (Object Files) environments:
  `vcutil parse --instrument function+function_call`
- For parallel instrumentation of a Cover environment:
  `vcutil instrument --coverage_type function+function_call`

To Initialize Statement+Branch

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 initialize Statement + Branch coverage when using Industry Modes, choose **Coverage => Initialize => <*appropriate type*>** based on the current Industry Mode.

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> TOols Cover STATEMENT+BRANCH [<unitname> | ALL]

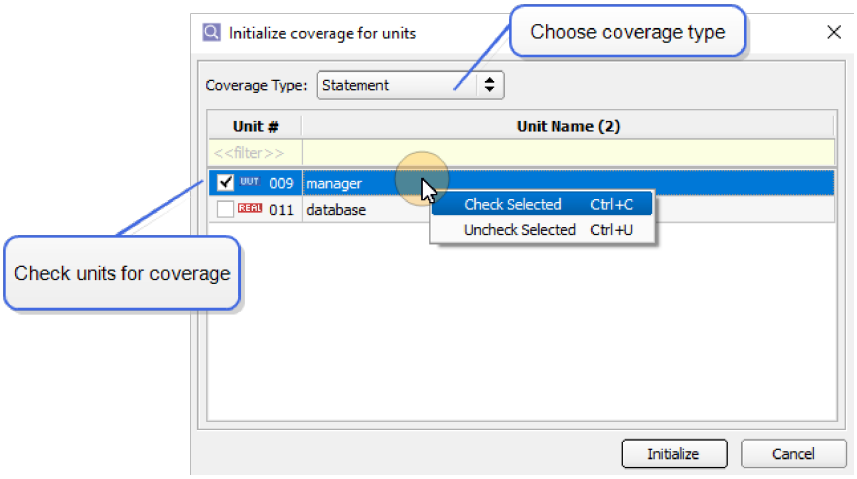
Initialize instrumentation of all UUT(s) in  the environment for
STATEMENT+BRANCH coverage. The optional argument <unitname> is the name of
a non-stubbed unit to be instrumented in addition to the UUTs, or "ALL" to
instrument all non-stubbed units in the environment as well.
```

To Initialize Coverage for Units Other than UUT

Coverage = > Initialize Custom... enables you to instrument multiple units (the UUTs and non-stubbed dependents, for example). When you choose **Coverage => Initialize Custom..**, a dialog

appears listing all units that can be instrumented. The unit(s) under test (UUTs) are always listed first in the list. Each UUT displayed is capable of being selected for coverage.

In the figure below, the **manager** unit is a UUT (UUT), and **database** is a non-stubbed unit (REAL). Any unit that is not-stubbed or ignored is given the icon REAL .



You may select individual units for coverage by checking the box next to the unit name or right-clicking and choosing **Check Selected**. After selecting the units, choose a coverage type from the drop-down list, and click the **Initialize** button to initialize coverage for the selected units.

Once you initialize a coverage type for real units (that is, non-stubbed or ignored units), any subsequent use of **Coverage => Initialize** affects those units as well.

Note: It is not possible to specify different types of coverage for different units.

To exit the dialog without initializing coverage, click the **Cancel** button.

```
clicast -e <env> Tools <coverage-type> [<unit_name1> <unit_name2…] | [ALL]

where <coverage-type> is one of < STATEMENT | STATEMENT+MC/DC |
STATEMENT+BRANCH | MC/DC | BASIS_PATHS | NONE >

Initialize instrumentation of all UUT(s) in the environment for <coverage-
type>. Similar to Initialize Custom, the optional arguments <unit_Xname> is
the name of a non-stubbed units to be instrumented in addition to the UUTs,
or "ALL" to instrument all non-stubbed units in the environment as well.
```

To disable coverage for a particular UUT, use the following CLICAST command:

```
clicast -e <env> -u <unit> TOols Cover UUT_Disable <coverage-type>

where <coverage-type> is one of Statement | Branch | Basis_paths | MCDC |
STATEMENT+MC/DC | STATEMENT+BRANCH | None
```

```
Disable instrumentation of <unit>  the next time coverage is instrumented.
The <coverage-types> argument specifies the type of coverage to initialize
for the other UUTs which still have coverage enabled.
```

To 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*/

Code that you do not want to be instrumented for code coverage...

/*VCAST_DONT_INSTRUMENT_END*/

or

//VCAST_DONT_INSTRUMENT_START

Code that you do not want

to be instrumented for code coverage...

//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.

When a conditional statement or macro expands to longer than 5,000 characters, the environment variable VCAST_MAX_LINE_LENGTHneeds to be set to allow VectorCAST's coverage instrumenter to expand its memory allocation processing.

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.

To Uninstrument an Environment

To completely remove coverage instrumentation from unit test environments, select **Coverage => Uninstrument** from the Menu Bar. Selecting the Uninstrument command removes all the coverage data and the environment is no longer instrumented.



Note: Execution results are not removed when the environment is uninstrumented.

```
clicast -e <env> TOols Cover None

Remove coverage from all UUTs and non-stubbed units with custom coverage in
a unit test environment.
```

To Enable Coverage

The **Coverage => Enable** command turns on code coverage for the current open environment. This command is only valid for environments that previously had coverage initialized and disabled. Subsequent to enabling coverage, all test cases will be run against the instrumented test harness (UUT_INST.EXE).



Note: The status bar indicates that coverage is enabled by displaying "Coverage type: <type>."

```
clicast -e <env> TOols Cover Enable

Run subsequent test cases with coverage monitoring enabled, using the
instrumented test harness.
```

To Disable Coverage

The **Coverage = > Disable** command turns off coverage for the current environment. This command

is only available for environments that have coverage initialized and enabled. All subsequent test case execution will use the non-instrumented harness. The purpose of disabling coverage is to enable you to run test cases using the original, non-instrumented test harness (UUT_INTE.EXE), so that you can ensure that the coverage instrumentation does not affect the test results.

If this menu item is dimmed, coverage is already disabled or has not been initialized.

Note: The status bar indicates that coverage is disabled by displaying "Coverage type: <type>" (disabled)."

```
clicast -e <env> TOols Cover Disable

Run subsequent test cases with coverage monitoring disabled, using the un-
instrumented test harness.
```

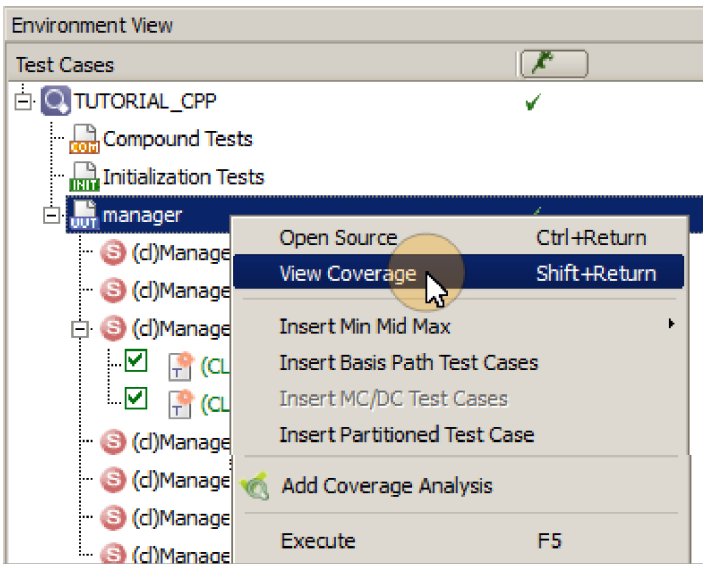
Viewing Coverage Results

To Turn on Coverage Results

Once coverage has been initialized and test case(s) run, there are several ways to view the achieved coverage. Each unit has its own Coverage Viewer which contains an annotated version of the source file, colorized to indicate the coverage level achieved.

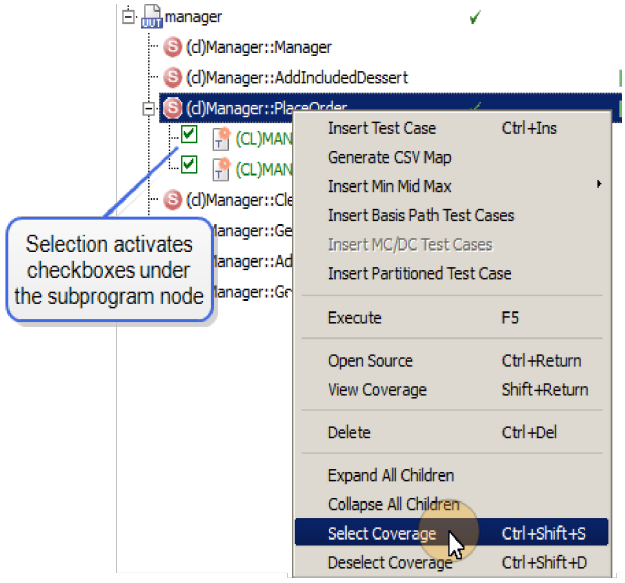
The Coverage Viewer is opened by highlighting unit, subprogram or test case nodes in the Test Case Tree and clicking the View Coverage icon in the Toolbar.

You can also open the Coverage Viewer for any number of units by selecting those units in the Test Case Tree, right-clicking, and selecting **View Coverage** from the context menu.



To view coverage achieved by multiple test cases, first select multiple test case, subprogram, or unit nodes in the Test Case Tree. Then right-click and choose **Select Coverage**, which activates the

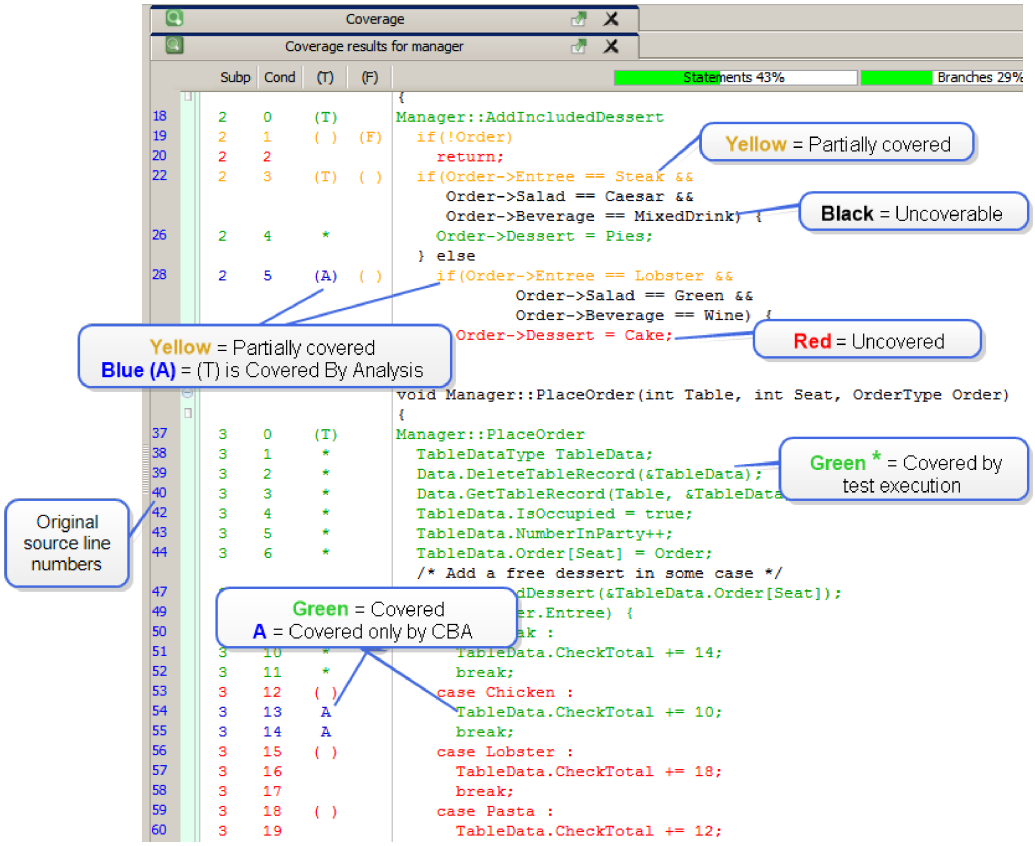
checkboxes next to all selected test cases. The Coverage Viewer reflects the coverage achieved by the selected test cases.



The Coverage Viewer

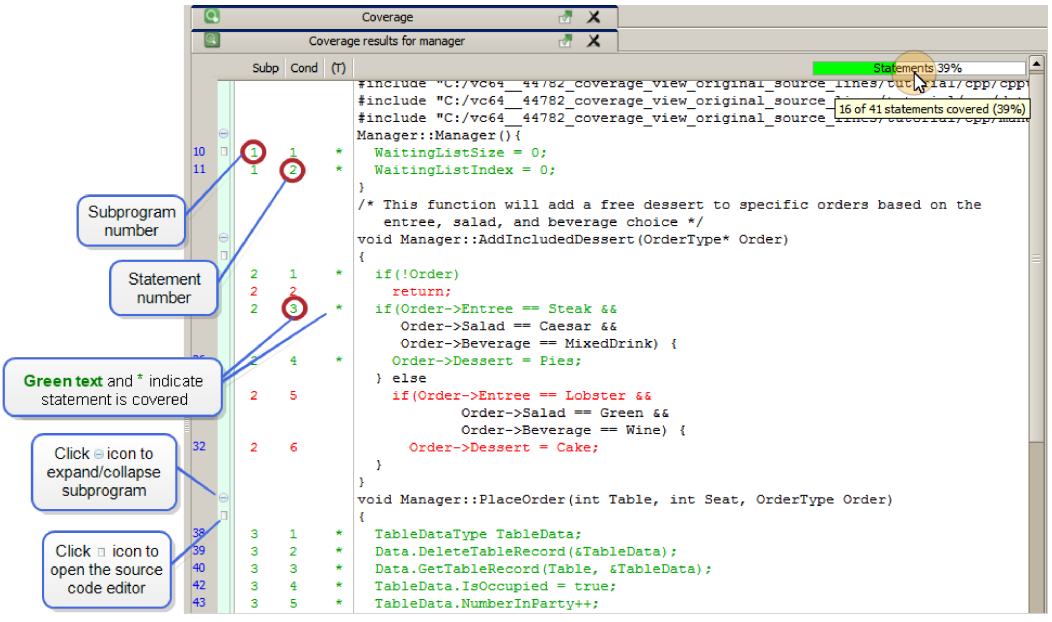
The Coverage Viewer contains an annotated version of the source file, colorized to indicate the coverage level achieved:

- 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

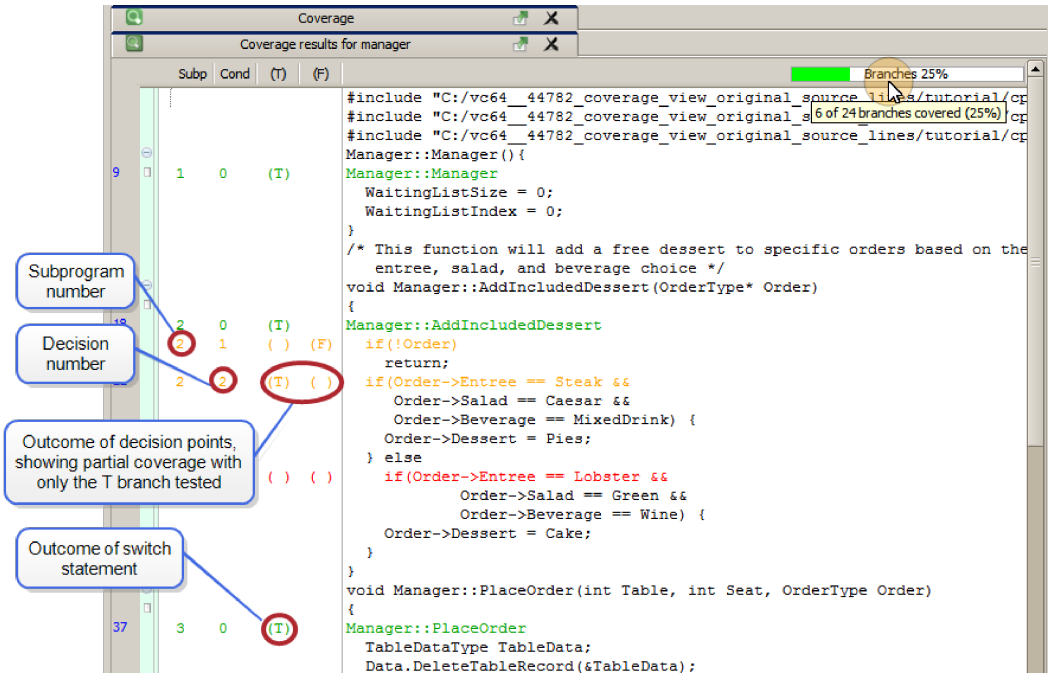
When Statement coverage is active, the Coverage Viewer looks like the following example:



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

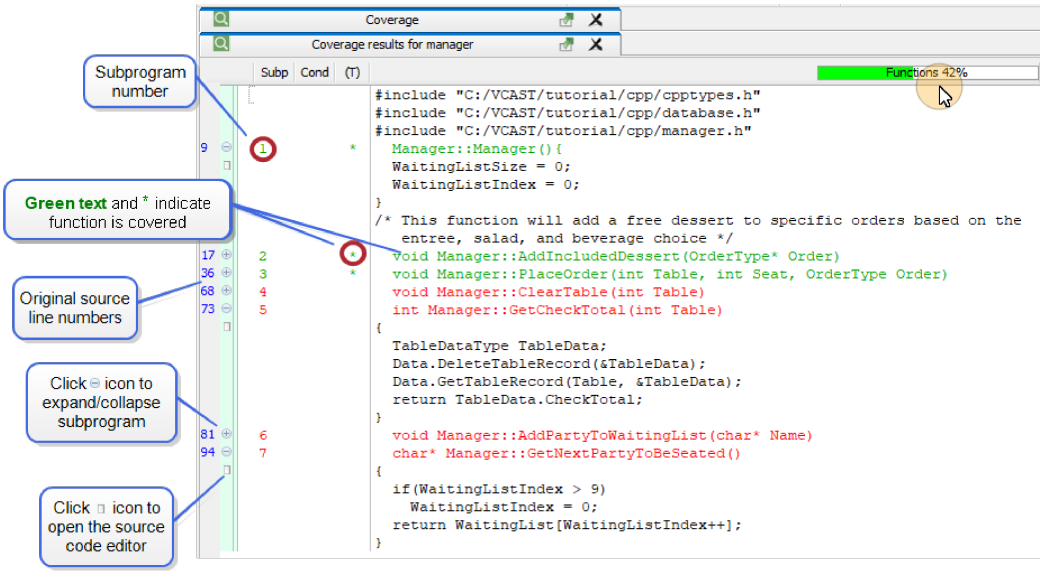
When Branch coverage is active, the Coverage Viewer looks like the following example:



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.

Function Coverage

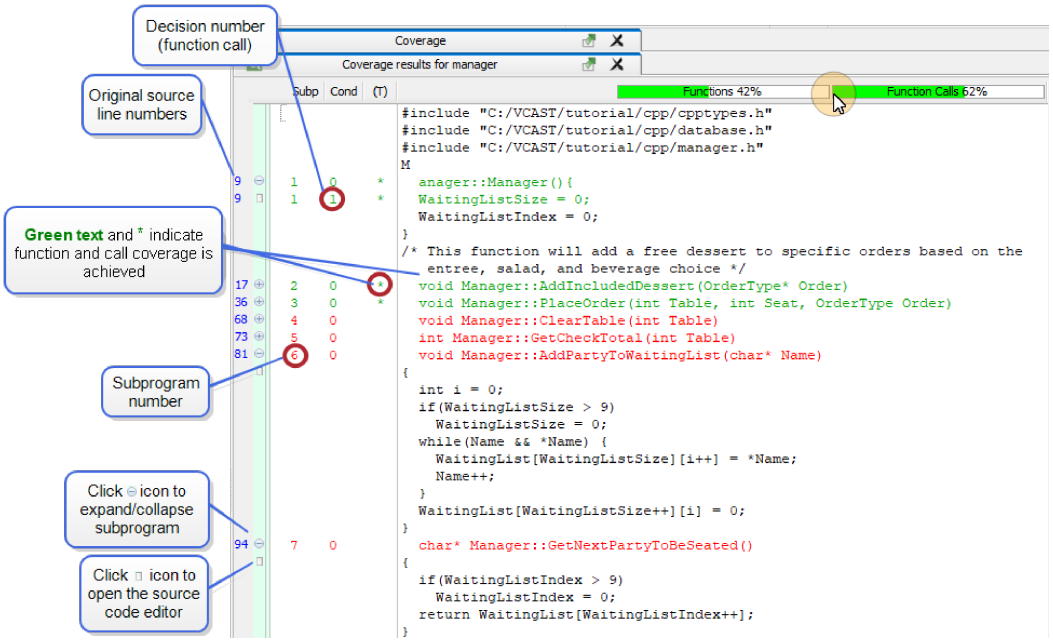
When Function coverage is active, the Coverage Viewer looks like the following example:



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 + Function Call Coverage

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

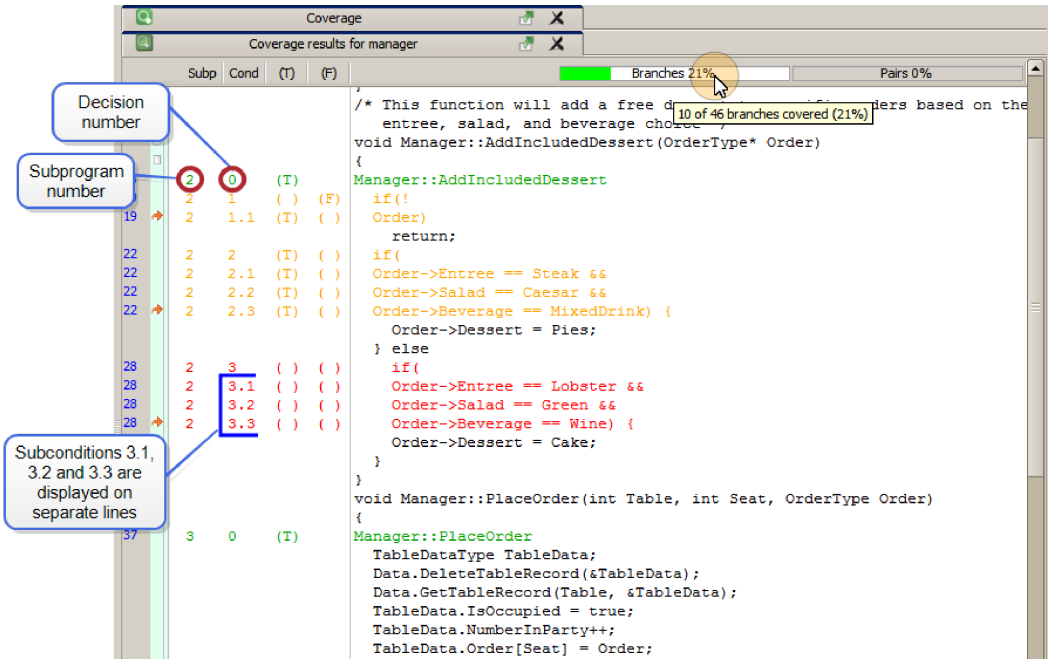


The numbers along the left-hand margin indicate the subprogram number. An asterisk (*) indicates that the function was covered (along with the color-coding). A subprogram is considered covered

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.

MC/DC Coverage

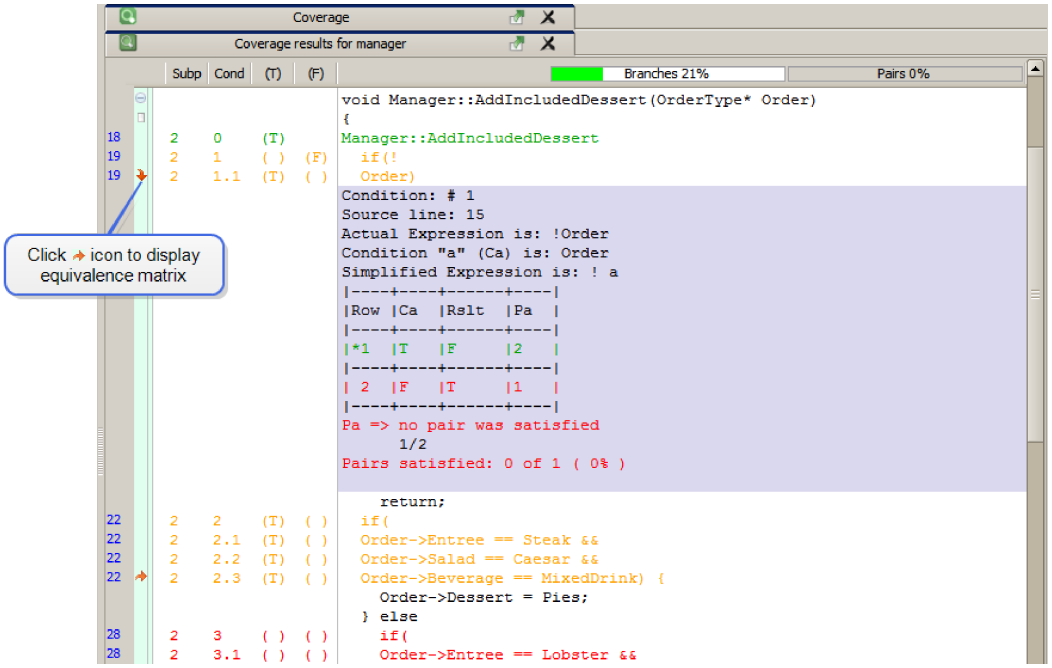
When MC/DC coverage is active, the Coverage Viewer looks like the following example:



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.

Note: When you re-initialize coverage, rebuild or update the environment, or regenerate basis paths, the coverage results are removed.

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.



See also the Coverage option "Provide Code Coverage in Header Files" on page 530.

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.

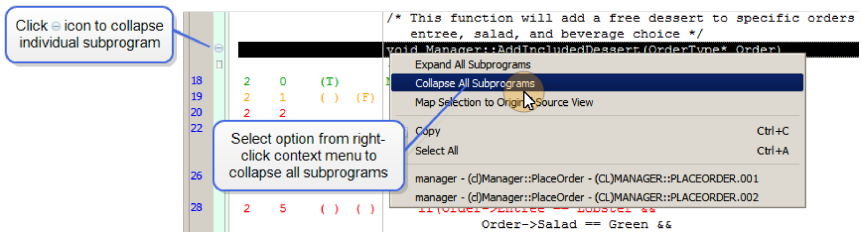
If there are no lines found, the message “Search reached bottom” or “Search reached top” appears, depending on which direction you were searching.

In addition, pressing the **Home key** or **End key** (or alternatively, pressing **Ctrl+Home** or **Ctrl+End**) brings the cursor to the beginning or end of the file, respectively.

To Customize the Coverage Viewer

By default, each unit's tab in the Coverage Viewer shows all subprograms in that unit as fully expanded. You can customize the display by collapsing and expanding subprograms.

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.



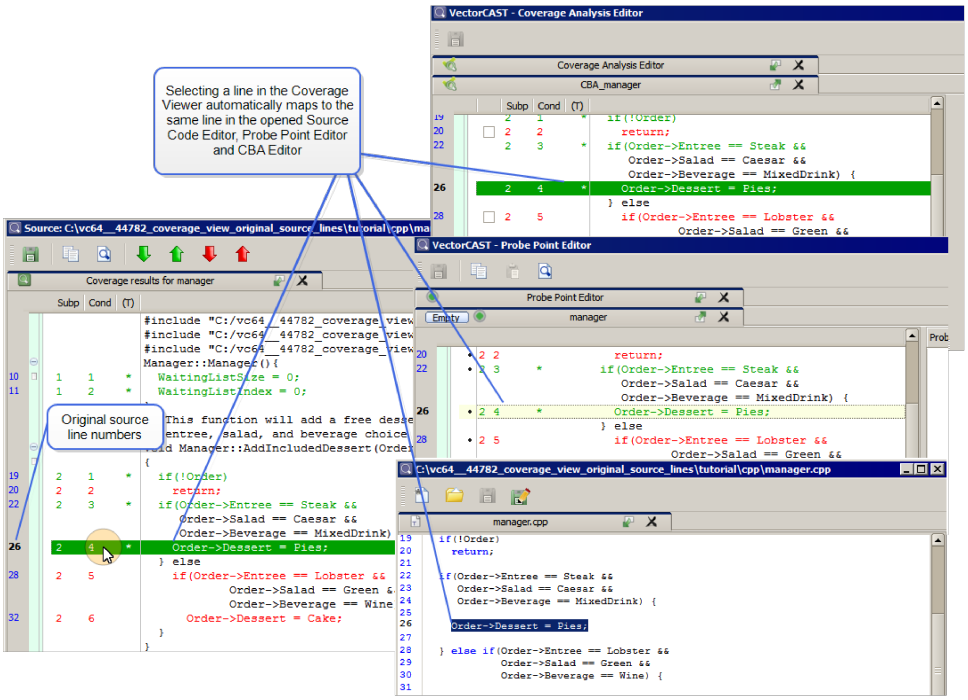
Use the Expand and Collapse menu items as "filters" or "layers" to get the exact display you want in the Coverage Viewer.

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

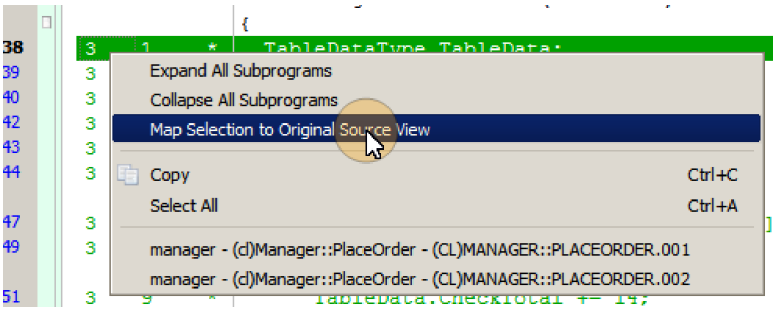
Collapsing and expanding the subprograms in the Coverage Viewer does not affect any of the Coverage reports.

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. If the Probe Point Editor or the CBA Editor are also open concurrently, the source line mapping is also enabled in those editors.



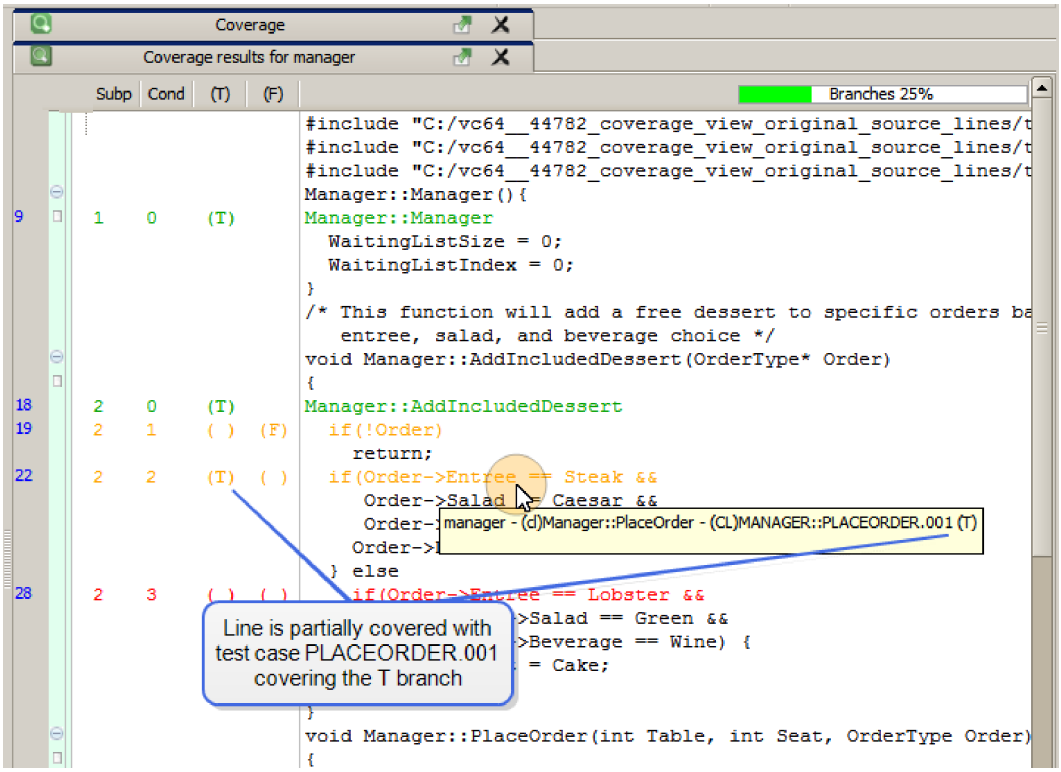
By default, selecting a line in the Coverage Viewer, Original Source text editor, Probe Point Editor or CBA Editor will automatically scroll to the corresponding line in any other open editor. To change the behavior, right-click within the Coverage Viewer, Probe Point Editor or CBA Editor and uncheck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



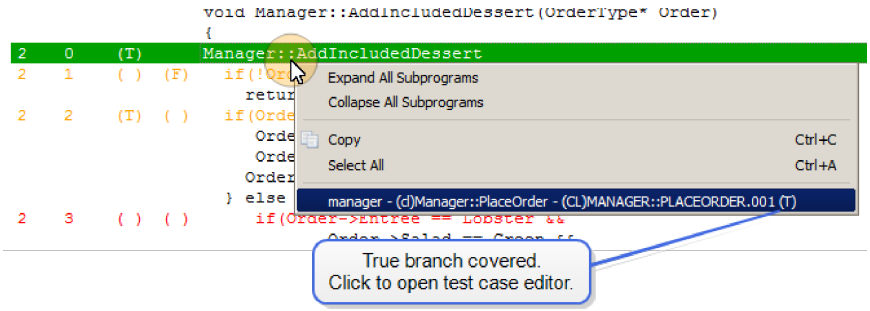
Alternatively, in the Original Source text editor, right-click and uncheck the **Map Current Line to Coverage View** option on the context menu.

To Open a Test Case That Covers a Line

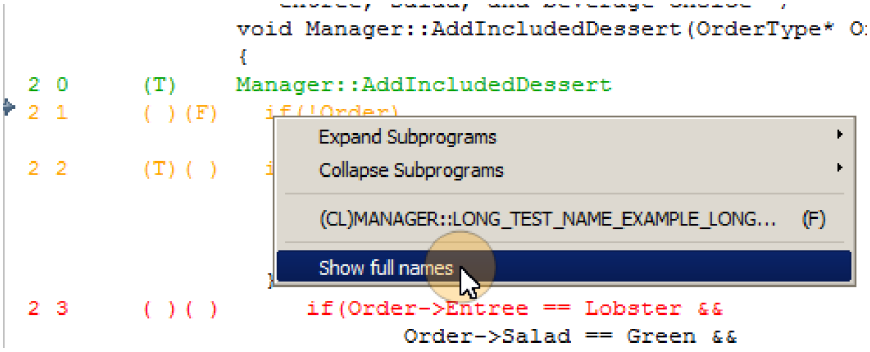
When viewing code coverage, if you hover the cursor over a covered (green) or partially-covered (yellow) line, a pop-up list is displayed that shows the names of all test cases that caused that line to be executed. Hovering over a branch will display (T) if the test case executes the True branch, (F)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



Right-click a covered line in the Coverage Viewer to see a pop-up menu listing the test case(s) that cover it. True (T), False (F) or both True and False (TF) branch coverage is also indicated. To open that test case in the Test Case Editor, right-click, and select the test case name from the pop-up list.



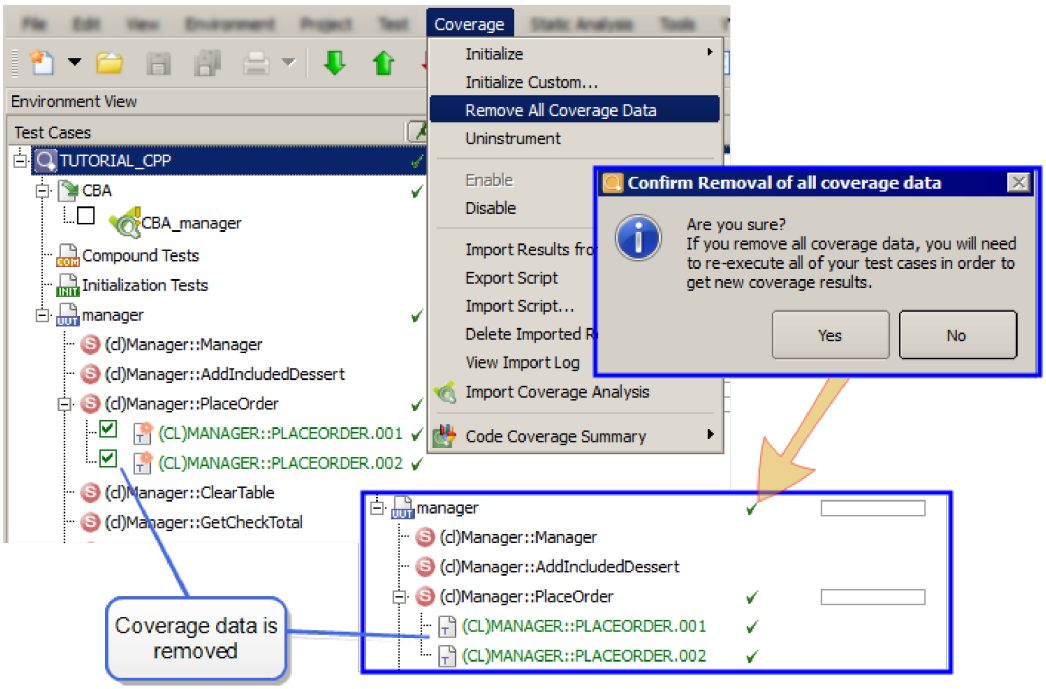
Note: Long test case names are truncated in the pop-up menu. Select the **Show full names** option to display the full test case name. This option is only available when test case names are truncated.



To Remove All Coverage Results

If you wish to discard all existing coverage data for an environment, choose **Coverage => Remove All Coverage Data**. A dialog appears asking for confirmation.

Clicking Yes in this dialog removes coverage checkboxes from the Test Case Tree, as shown below. Coverage remains initialized.



To View the Aggregate Coverage Report

An Aggregate Coverage Report includes for each unit selected in the Test Case Tree:

- an annotated source-listing, indicating the lines covered
- a metrics table giving the complexity and the level of coverage achieved for each subprogram

The report is sensitive to what level is selected in the Test Case Tree. If you select the top-level node

for the environment, the report includes coverage achieved by test cases executed in all UUTs and non-stubbed units that have coverage instrumented using Custom Initialize. You can narrow the content down by selecting particular units in the Test Case Tree for which you want coverage achieved. Once you have made a unit selection, choose **Test => View => Aggregate Coverage Report**.

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

The annotated source code is color-coded to mark the exercised lines (green), the un-exercised lines (red), lines covered by analysis (blue) and partially exercised lines (yellow), similar to what is seen in the Coverage Viewer.

```
clicast -e <env> [-u <unit>] REports Custom Coverage <outputfile>

Create an Aggregate Coverage report and output to HTML file <env>_
aggregate_coverage_report.html, standard output as text, or to the
specified output file. If the Unit parameter is specified, the report
includes the annotated source code for the specified unit, and the Metrics
table includes only the specified unit; otherwise, the report includes the
annotated source code for all units and the Metrics table includes all
units in the environment.
```

XML Aggregate Coverage Report

An Aggregate Coverage Report in XML format is accessible from CLICAST only. The report indicates whether each line of the specified unit's original source file is covered or not.

The XML Aggregate report provides valid data for C and C++ units only.

```
clicast -lc -e <env> [-u <unit>] Reports Custom Xml_aggregate
[<outputfile>]

Create an Aggregate Coverage report in XML format, indicating whether each
line of <unit>'s original source file is covered or not. The report is
output to the specified <output file> or to <env>_coverage.xml if none is
specified. If the <unit> parameter is specified, the XML Aggregate Coverage
Report includes only the specified unit; otherwise, the report includes all
units in the environment.
```

To View the Metrics Report

A Metrics Report includes a metrics table giving the complexity and the level of coverage achieved for each subprogram in the selected unit(s) for each unit selected in the Test Case Tree.

The report is sensitive to what level is selected in the Test Case Tree. If you select the top-level node for the environment, the Metrics table includes coverage achieved by test cases executed in all UUTs and non-stubbed units that have coverage instrumented using Custom Initialize. You can narrow the content down by selecting particular units in the Test Case Tree for which you want coverage achieved. Once you have made a unit selection, choose **Test => View => Metrics Report**.

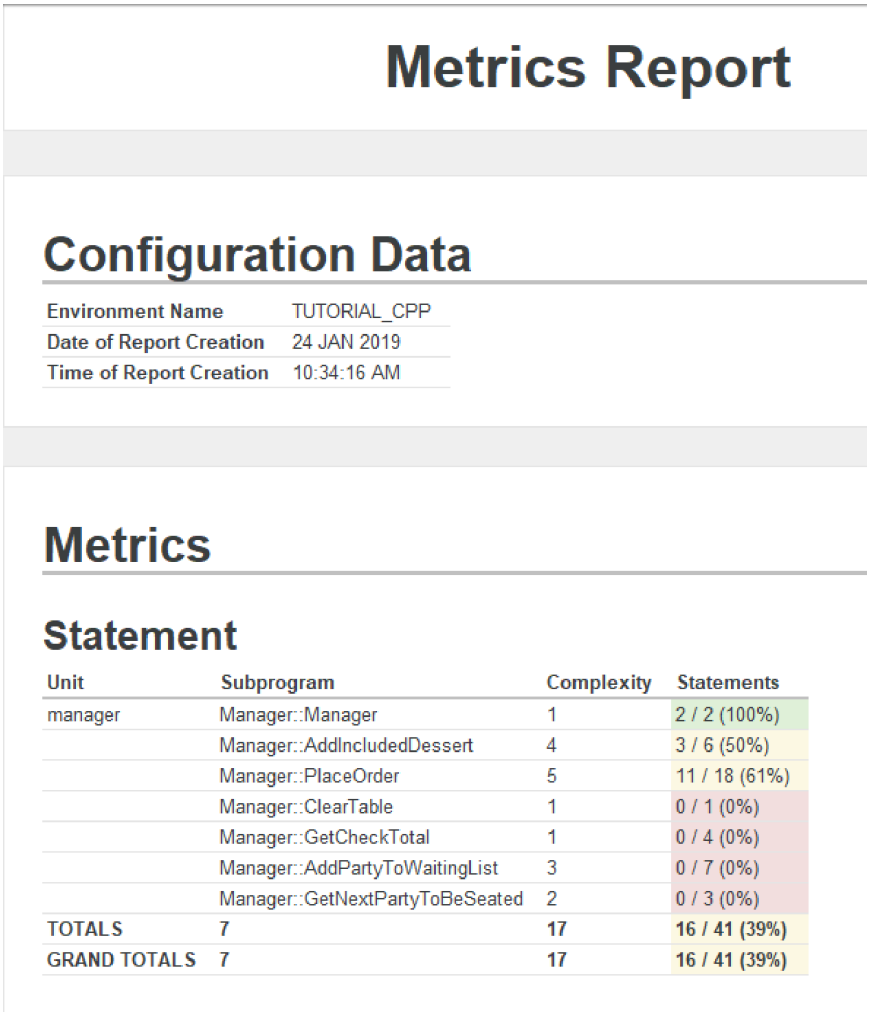
The report contains coverage data from all test cases within the selection, regardless of whether they are toggled on to show coverage or not.

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>] Reports Custom Metrics <outputfile>

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

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



Metrics Report

Configuration Data

Environment Name	TUTORIAL_CPP
Date of Report Creation	24 JAN 2019
Time of Report Creation	10:34:16 AM

Metrics

Statement

Unit	Subprogram	Complexity	Statements
manager	Manager::Manager	1	2 / 2 (100%)
	Manager::AddIncludedDessert	4	3 / 6 (50%)
	Manager::PlaceOrder	5	11 / 18 (61%)
	Manager::ClearTable	1	0 / 1 (0%)
	Manager::GetCheckTotal	1	0 / 4 (0%)
	Manager::AddPartyToWaitingList	3	0 / 7 (0%)
	Manager::GetNextPartyToBeSeated	2	0 / 3 (0%)
TOTALS	**7**	**17**	**16 / 41 (39%)**
GRAND TOTALS	**7**	**17**	**16 / 41 (39%)**

Pairs Status

When MC/DC or STATEMENT+MC/DC coverage is initialized, an additional “Pairs” column is added to the Metrics Report. 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	TUTORIAL_CPP
Date of Report Creation	24 JAN 2019
Time of Report Creation	11:00:26 AM

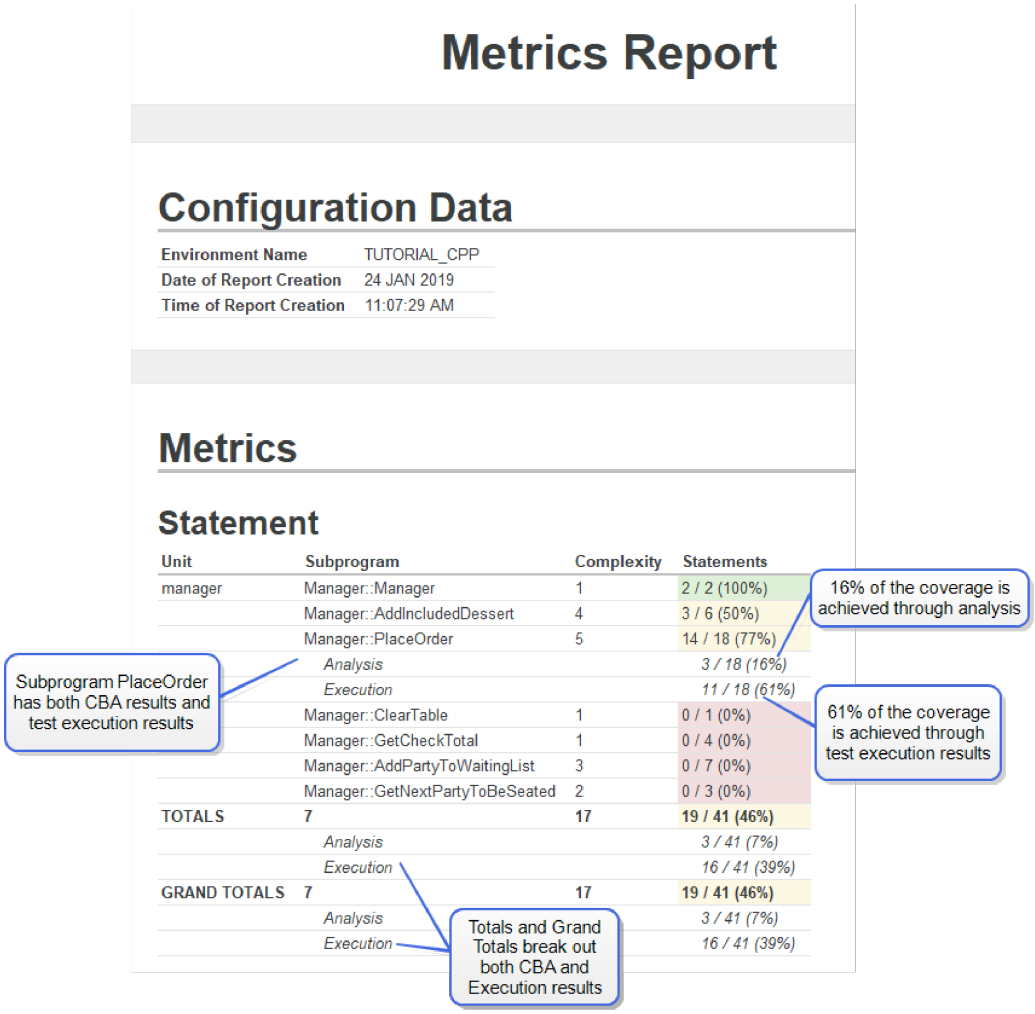
Metrics

Statement+MC/DC

Unit	Subprogram	Complexity	Statements	Branches	Pairs
manager	Manager::Manager	1	2 / 2 (100%)	1 / 1 (100%)	
	Manager::AddIncludedDessert	4	3 / 6 (50%)	7 / 21 (33%)	0 / 7 (0%)
	Manager::PlaceOrder	5	12 / 23 (52%)	2 / 6 (33%)	
	Manager::ClearTable	1	0 / 1 (0%)	0 / 1 (0%)	
	Manager::GetCheckTotal	1	0 / 4 (0%)	0 / 1 (0%)	
	Manager::AddPartyToWaitingList	3	0 / 7 (0%)	0 / 11 (0%)	0 / 3 (0%)
	Manager::GetNextPartyToBeSeated	2	0 / 3 (0%)	0 / 5 (0%)	0 / 1 (0%)
TOTALS	**7**	**17**	**17 / 46 (36%)**	**10 / 46 (21%)**	**0 / 11 (0%)**
GRAND TOTALS	**7**	**17**	**17 / 46 (36%)**	**10 / 46 (21%)**	**0 / 11 (0%)**

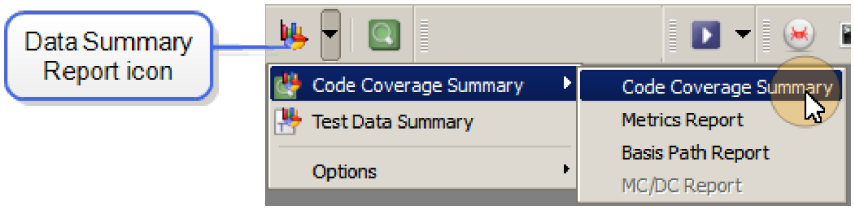
Covered By Analysis Results

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 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.

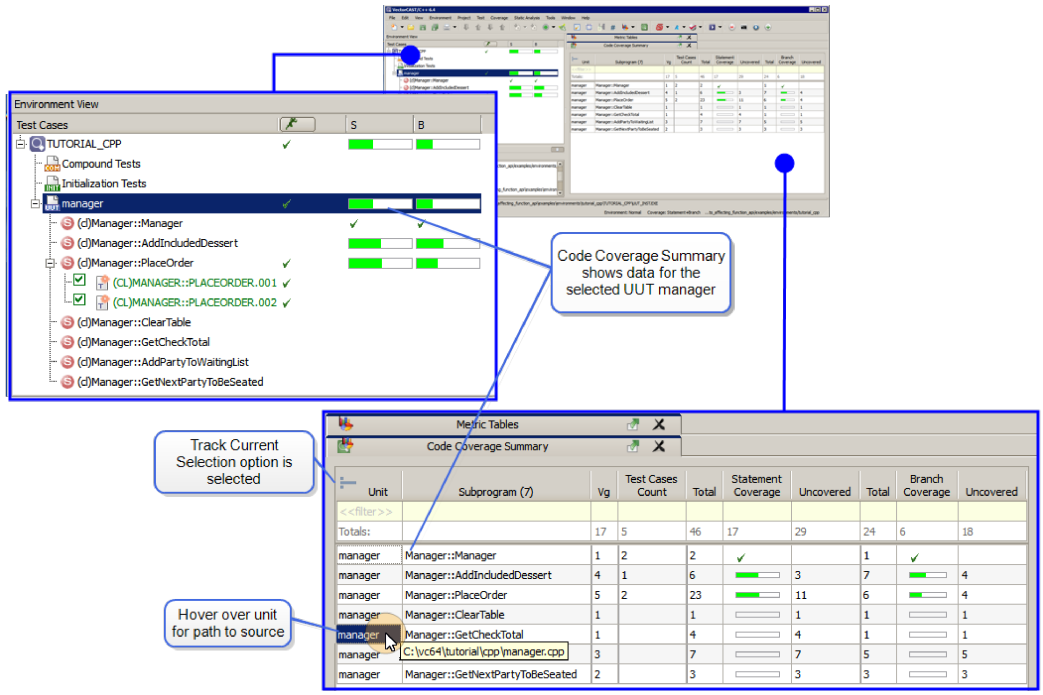


Alternatively, from the Menu Bar, select **Coverage => Code Coverage Summary => Code Coverage Summary**.

Viewing Code Coverage

By default, the contents of the Code Coverage Summary reflect the UUTs selected in the Test Case Tree. A tracking icon is displayed at the top of the Unit column indicating that the Code Coverage Summary is currently tracking selected UUTs from the Test Case Tree.

To override tracking of selected UUTs in the Test Case Tree, 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 Test Case Tree.

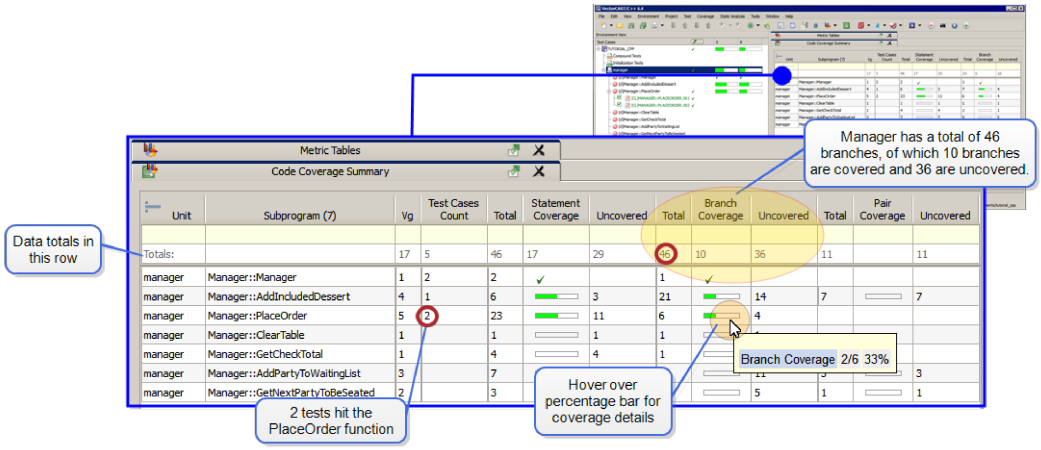


Unit	Subprogram (7)	Vg	Test Cases Count	Total	Statement Coverage	Uncovered	Total	Branch Coverage	Uncovered
<<filter>>									
Totals:		17	5	46	17	29	24	6	18
manager	Manager::Manager	1	2	2	✓		1	✓	
manager	Manager::AddIncludedDessert	4	1	6		3	7		4
manager	Manager::PlaceOrder	5	2	23		11	6		4
manager	Manager::ClearTable	1		1		1	1		1
manager	Manager::GetCheckTotal	1		4		4	1		1
manager		3		7		7	5		5
manager	Manager::GetNextPartyToBeSeated	2		3		3	3		3

The Code Coverage Summary table is dynamic. When the **Track Current Selection** option is enabled, as units are selected and deselected in the Test Case Tree, 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 Test Cases Count (showing the number of unique test cases that have hit a particular subprogram during execution, with compound tests being counted as one test regardless of the number of slots in the compound), 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.



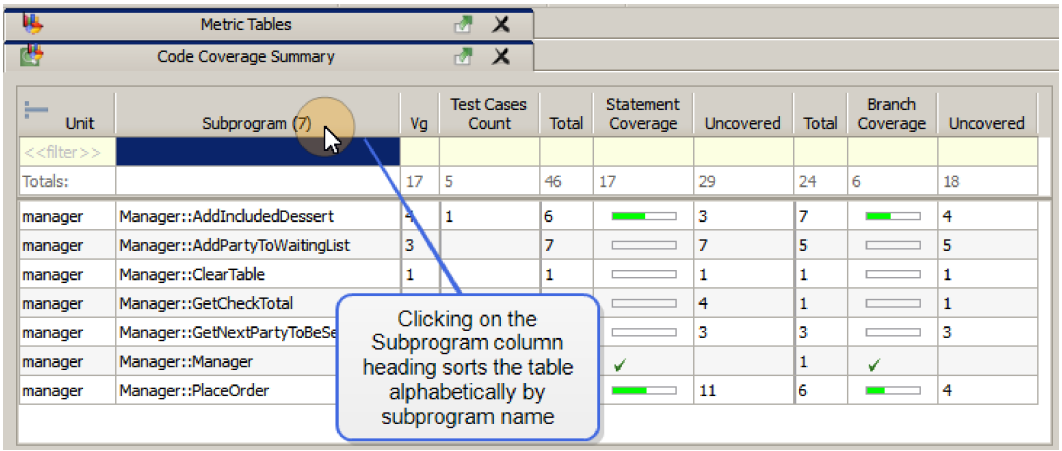
The Totals row at the top of the table displays the totals for each data column. In the example above, note that in the unit Manager we have a total of 46 branches, of which 10 branches are covered and 36 are uncovered.

The Summary table updates whenever coverage data is updated. For example, the table refreshes when coverage is initialized, or following test execution.

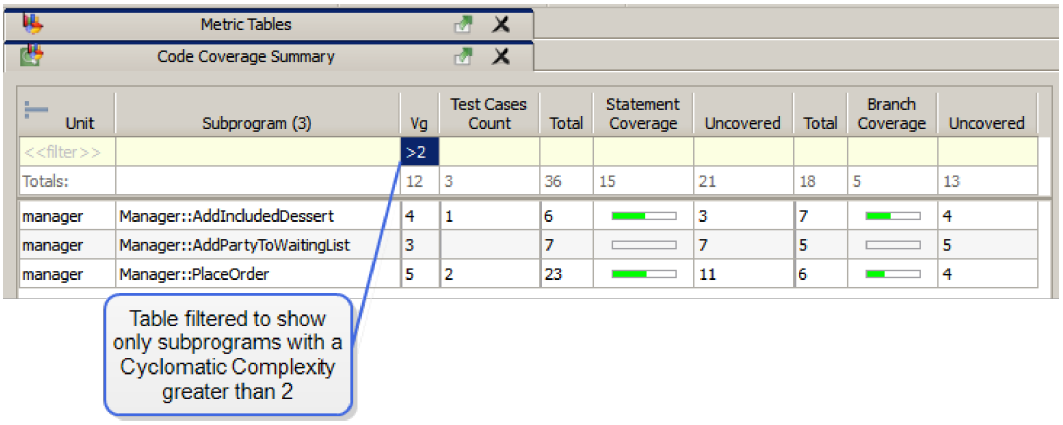
Double-clicking on a line in the Code Coverage Summary opens the corresponding UUT in the Coverage Viewer.

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. See "To Save a Script or Text File " on page 86 for more information.

Printing the contents of the Code Coverage Summary is supported. See "To Print an Open Window" on page 87 for more information.

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
```

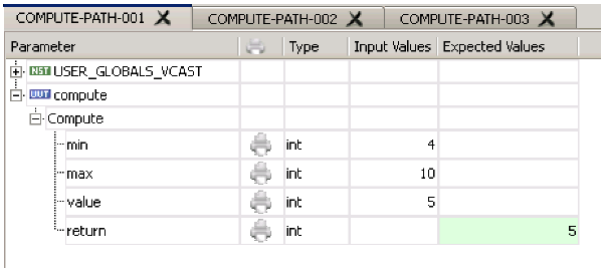
```
  Source Code Listing

                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;
                 }
```

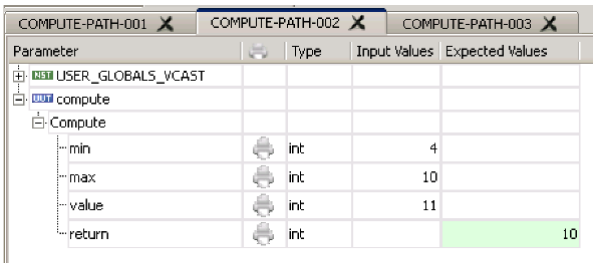
To Build Test Cases from Basis Path Analysis

When using the basis path analysis to build test cases, you should build at least one test case for each basis path ("Test Path"). The test case should contain the collection of data that will set the conditions specified in the basis path listing.

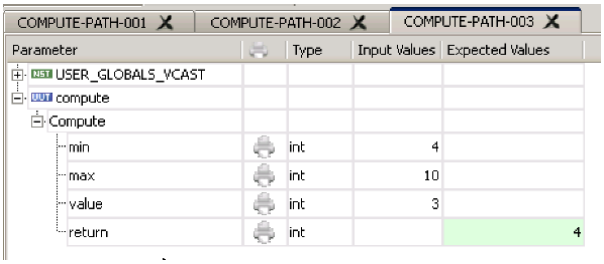
Using the above example, Test Path 1 specifies that **`value < min`** should be False, and **`value > max`** should also be False. If we use 4 for **`min`** and 10 for **`max`**, setting **`value`** to 5 satisfies this Test Path.



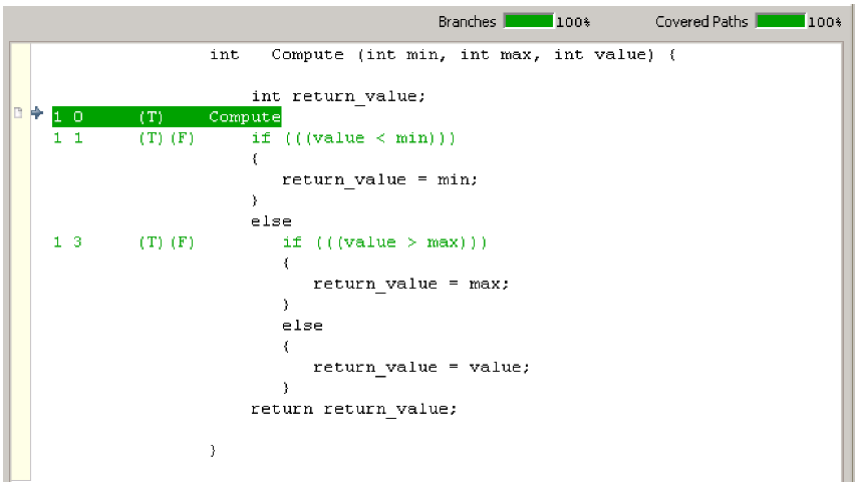
Test Path 2 specifies that **`value < min`** should be False, and **`value > max`** should be True. If we use 4 for **`min`** and 10 for **`max`**, setting **`value`** to 11 satisfies this Test Path.



Test Path 3 specifies that **`value < min`** should be True. We can use 4 for **`min`** and 3 for **`value`**. There is no need to set **`max`**; it is irrelevant.



Executing these three test cases gives us 100% branch coverage and 100% basis paths coverage.



VectorCAST can automatically build the basis path test cases. For details, see "To Insert Basis Path Test Cases" on page 257

To View a Basis Paths Report

The **Tools => Basis Path => View Analysis 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 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.

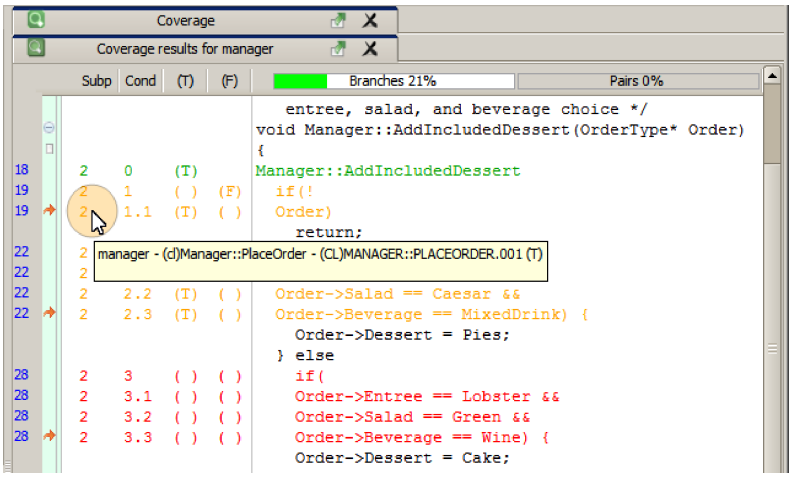
```
clicast -e <env> [-u <unit>] TOols Basis_path [<outputfile>]

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
```

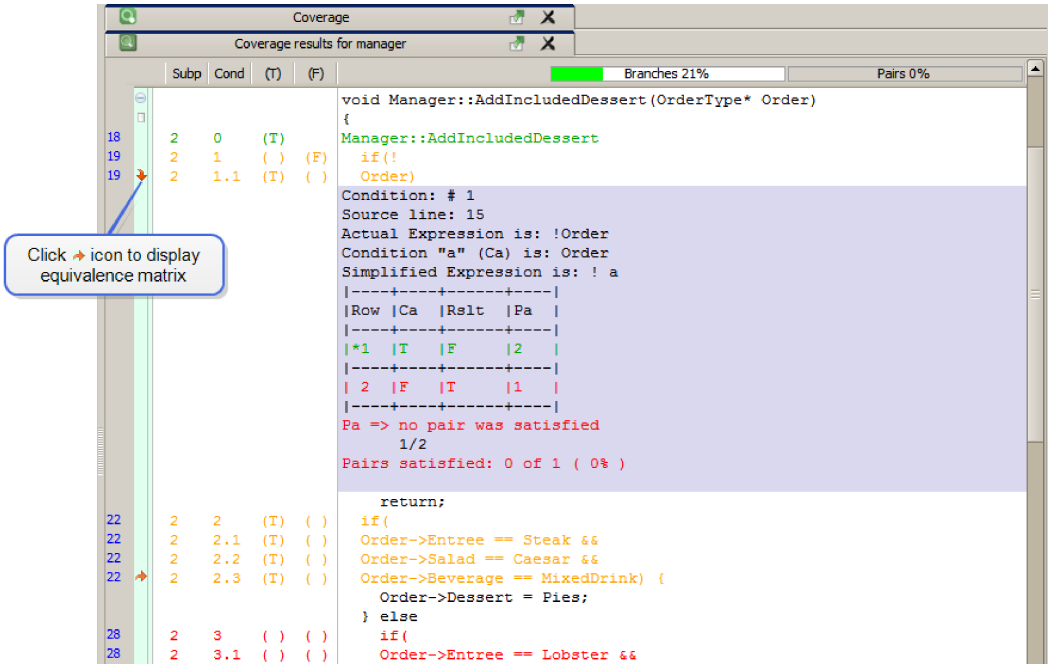
```
HTML and <outputfile> is not specified, the file is saved to <env>_basis_
path_report.html.
```

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 tooltip 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 Equivalence Matrix. The matrices can also be viewed by selecting

Tools => MC/DC => View Equivalence Matrices and viewing the MC/DC Condition Tables Report. See "To View the Equivalence Matrices Report" on page 511 for more information.

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 the row has been covered. When an equivalence pair has been covered, the Pa through P*n* summary lines reflect that information.

The Equivalence Pair Matrix shows only rows having equivalence pair information. Rows that do not have any pair infomation are eliminated from the display because they add no useful 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


      Note: 5 of 6 branch outcomes satisfied

4 1     (T)(F)     if ((
4 1.1   (T)(F)        a &&
4 1.2   (T)( )        b )   ) {
                                 local = 1;
                    }                   }

===== 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.

To View the Equivalence Matrices Report

Choose **Tools => MC/DC => View Equivalence Matrices** to display the MC/DC Condition Tables report. This report includes the equivalence matrices for all units instrumented for MC/DC coverage in the environment as well as the MC/DC equivalence pair status.

The MC/DC Condition Tables report shows only rows having equivalence pair information. When VectorCAST instruments for MC/DC or Statement+ MC/DC coverage, it calculates the Equivalence Pair matrix. Each row represents a combination of possible outcomes for each condition in the expression, and the resulting output. A pair is a set of two rows which demonstrate that the result of the conditional (Rslt column) changes from True to False as one component (Ca or Cb) changes, while all other components are held constant.

For example, in the matrix below, rows 1 and 3 form a pair, and rows 1 and 2. However, row 4 does not form a pair with any other row because changing one of the components, such as Ca, from F to T does not change the result of the conditional (F) to True.

```
Source line: 179
Actual Expression is: ( p && q )
Condition "a" (Ca) is: p
Condition "b" (Cb) is: q
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    |     |
```

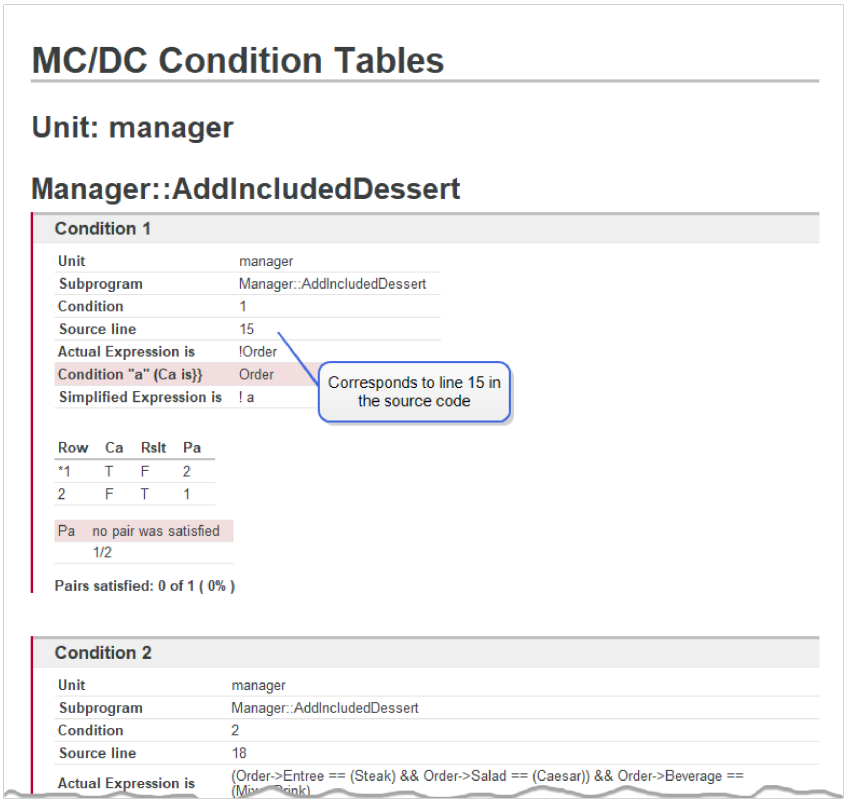
```
|-----+-----+-----+-----+-----+-----|
|4    | F   | F   | F   |     |     |
|-----+-----+-----+-----+-----+-----|
```

VectorCAST displays the rows that do not have any pair information as having blank boxes under the Pair column (Pa or Pb, in the example). These rows are eliminated from the display of the Equivalence Pair matrix because they add no useful information.

```
clicast -e <env> [-u <unit>] Reports Custom Equivalence_matrices
[<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 <env>_mcdc_
tables_report.html.
```

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

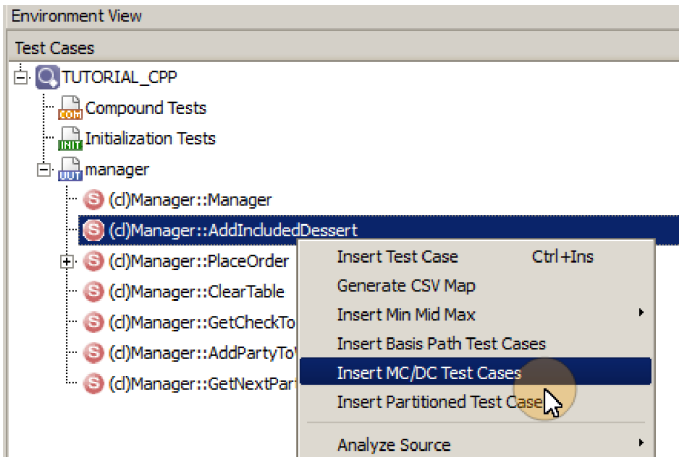


The **Source line #** refers to the source code line in the Coverage Viewer that corresponds to the expression being evaluated in the matrix. After noting the line number, right-click the unit in the Test Case Tree, and choose **View Coverage**. Then, in the Coverage Viewer, type **Ctrl+F** (or choose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

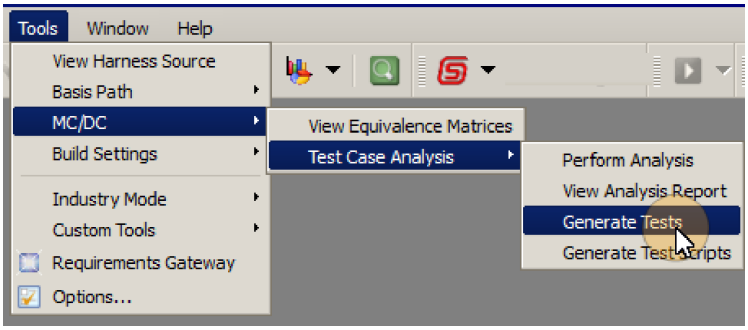
To Insert MC/DC Test Cases

VectorCAST can automatically generate MC/DC test cases to help achieve 100% pair coverage.

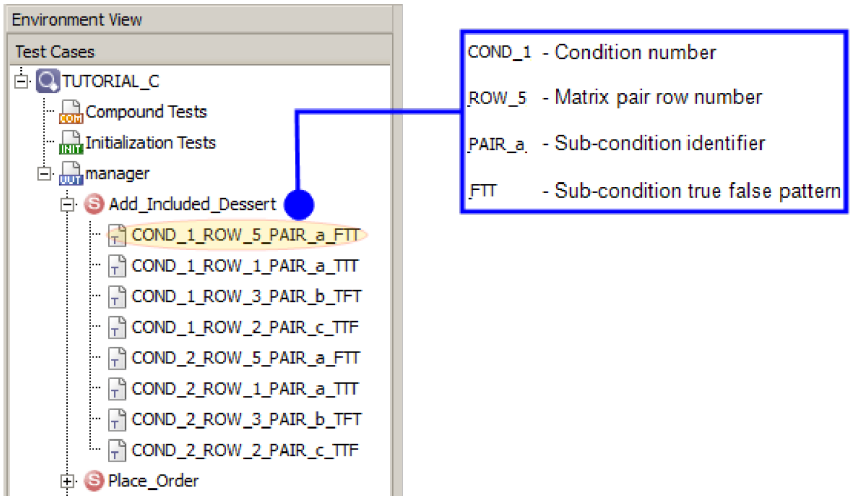
To generate the MC/DC test cases for the desired subprograms within an Environment View, select the subprograms by right-clicking the items in the test case tree and then selecting **Insert MC/DC Test Cases** from the context menu. Tests are created for each row in the equivalence matrix (see "Understanding MC/DC Analysis" on page 510). In addition to creating the test cases, VectorCAST automatically generates input values and sets required test preconditions for the test cases.



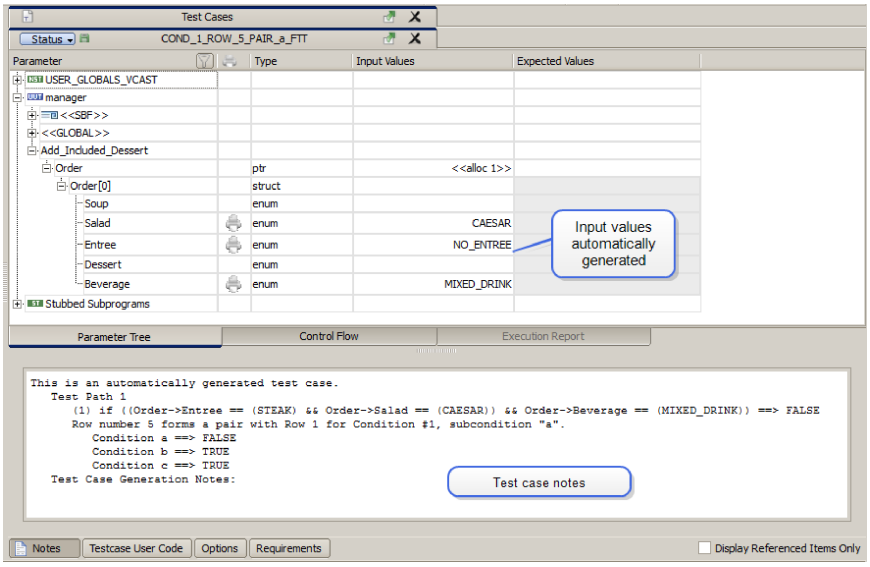
You may also generate tests by selecting **Tools => MC/DC => Test Case Analysis => Generate Tests**.



Test cases are automatically named to identify the condition number, the equivalence matrix pair row number, sub-condition identifier and the sub-condition true false pattern for each test. For example, using the first test case generated for the Add_Included_Dessert subprogram shown below, the condition number is COND_1, the matrix pair row number is ROW_5, the sub-condition identifier is PAIR_a, and the sub-condition true false pattern is FTT.

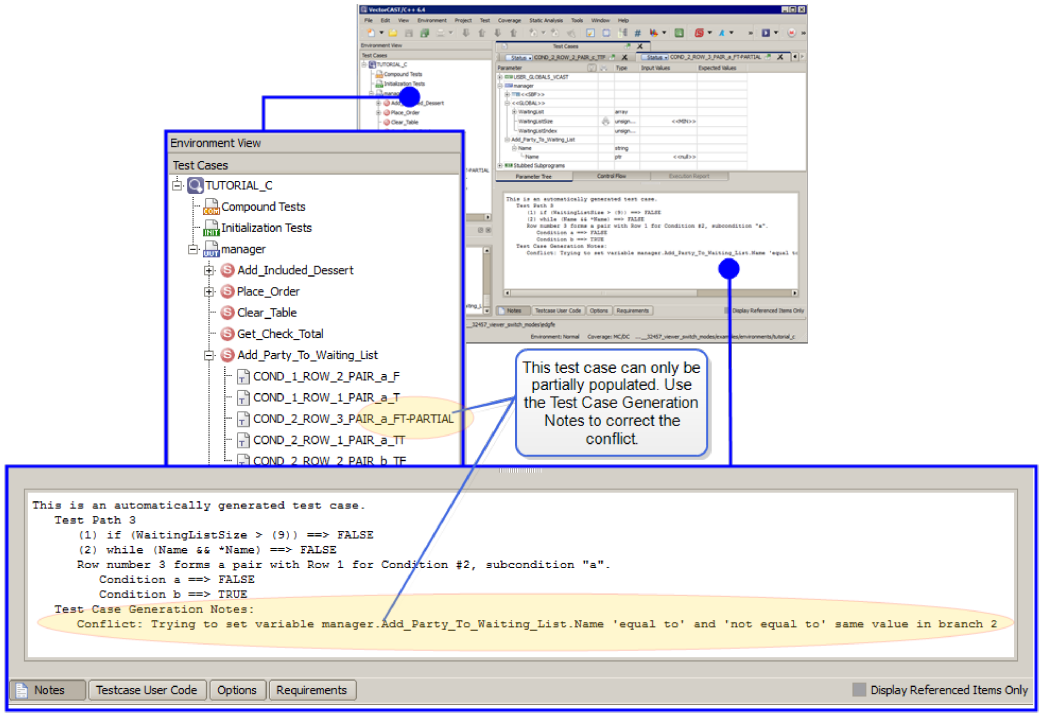


Opening the test case editor for COND_1_ROW_5_PAIR_a_FTT shows the automatically populated input values for the test case, and the notes section describes the test case.



If the generated test is not able to fully populate the test, “-PARTIAL” is appended to the test name. If the test case generated could not be populated at all, “-TEMPLATE” is appended to the test case name.

The “Test Case Generation Notes” section of the Notes tab for the test case describes any problems encountered in creating the test case. Use this section to aid in manually completing the test case.



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 Test Case Tree 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.



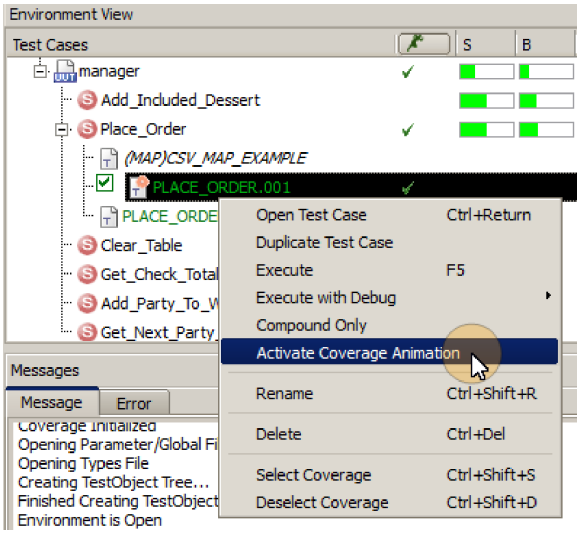
Note: If the Animation toolbar is dim, activate a test case for animation.

To Activate Coverage Animation

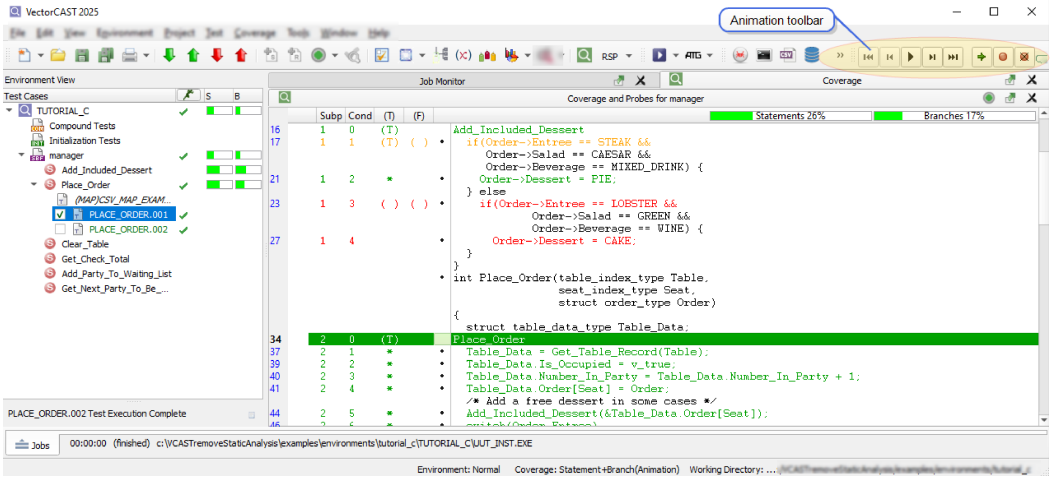
To use the Animation feature, select **Tools => Options** dialog, Coverage tab, and then select Animation for the **Coverage I/O Type**.

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 Test Case Tree, 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 

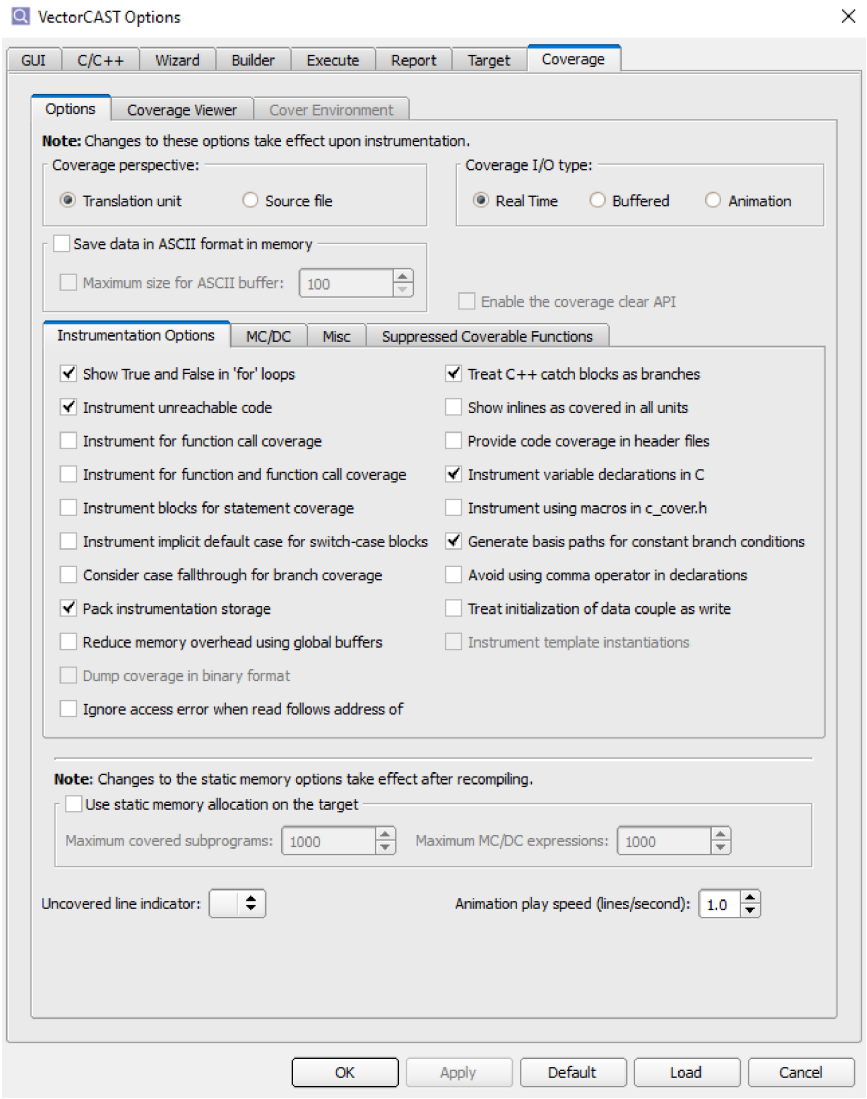
There is no limit to the number of breakpoints you can set.

Setting Coverage Options

General Coverage Options

Choose **Tools => Options** and click the **Coverage** tab, then click the **Options** sub-tab.

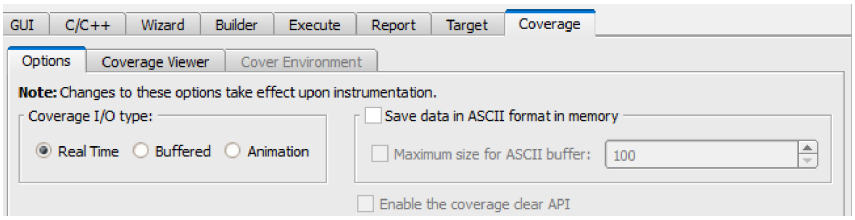
The **Coverage** options tab, **Options** sub-tab is used to set coverage options for unit test environments and VectorCAST Cover environments. When used for Cover environments, these options are frequently stored in the CCAST_.CFG file. Some options require re-instrumentation to take effect; others require recompiling the test harness.



See also "To Set VectorCAST Options" on page 376.

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.
- **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 and read out at the end of the test execution.

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 (used in this example).

After changing this option, you must reinitialize coverage.

clicast -lc option VCAST_MAX_CAPTURED_ASCII_DATA <integer number>

The <integer number> tells VectorCAST how much memory should be set aside for buffered ASCII data. When this option is unchecked, VectorCAST automatically calculates the buffer size for the ASCII data. This option should be increased only when the ASCII buffer overflows when using MCDC coverage. See the file INVALID_COVERAGE_LINES.LOG for more information.

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 at any time during application execution, then clear the coverage data, and repeat the process until you've collected all the necessary coverage data.

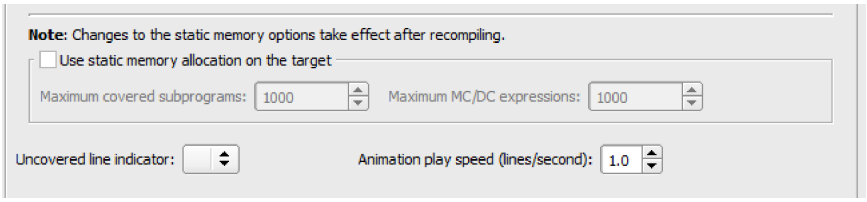
After changing this option, you must reinitialize coverage.

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 tab** if it is not selected.



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** tab if it is not selected.

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.

Note: After changing this option, you must recompile.

```
clicast -lc 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 tab** if it is not selected.

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.

For example, suppose the option is set to 499. During execution, VectorCAST encounters 499 unique MC/DC expressions in one function call. Because VectorCAST includes the function call entrance in computing the total, we have a total of 500 expressions (499 unique MC/DC expressions plus one function call entrance). This exceeds our limit of 499, and a text error message is added to the coverage data indicating that the limit was reached. This text error message is used to generate pop up messages and CLICAST output messages.



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.

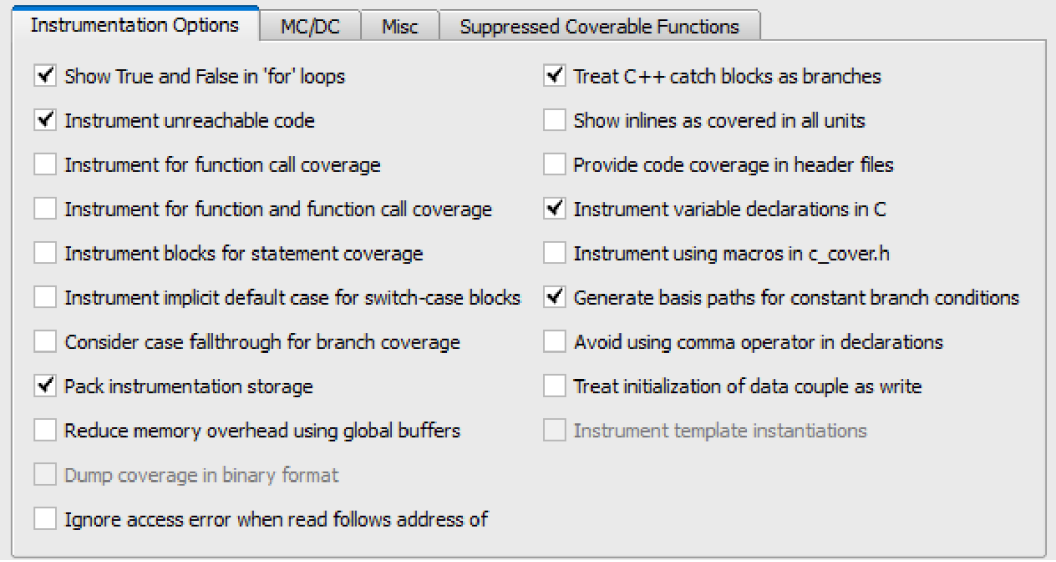
The “Animation play speed” option enables you to change the speed of coverage animation progress. By default, coverage animation proceeds at the rate of one line per second. To go faster, increase the value. To go slower, decrease the value. The ratio 1.0 represents 1 line / 1 second.

```
clicast -lc 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.
```

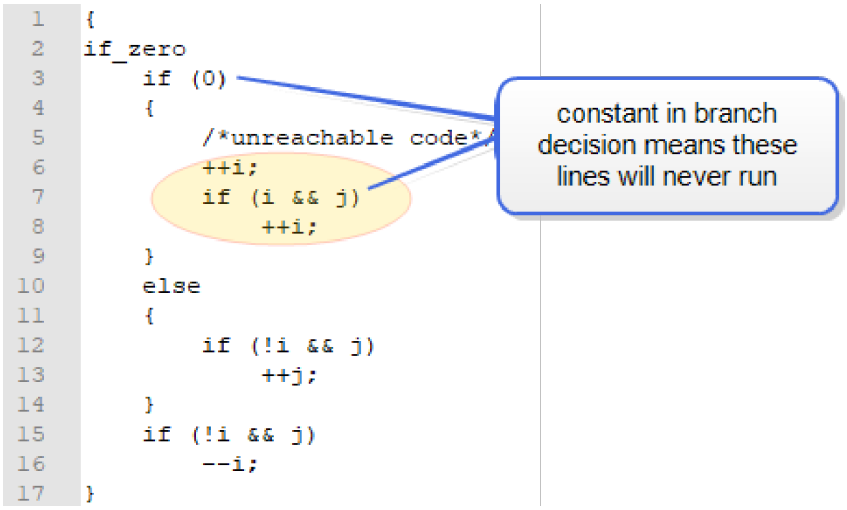
Instrument Unreachable Code

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.




```
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.
```

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.
```

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.
```

Provide Code Coverage in Header Files

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 provides code coverage on the header files for each unit. When a unit is displayed in the Coverage Viewer and this option is off (default), any **`#include`** lines are not expanded or coverable. If this option is on, any **`#include`** lines are expanded and therefore are coverable. 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_HEADERS [True | False]

When True, this option provides code coverage for C/C++ header files found
in search directories. When False, the unit's coverage data shows the
headers as #include... . 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:

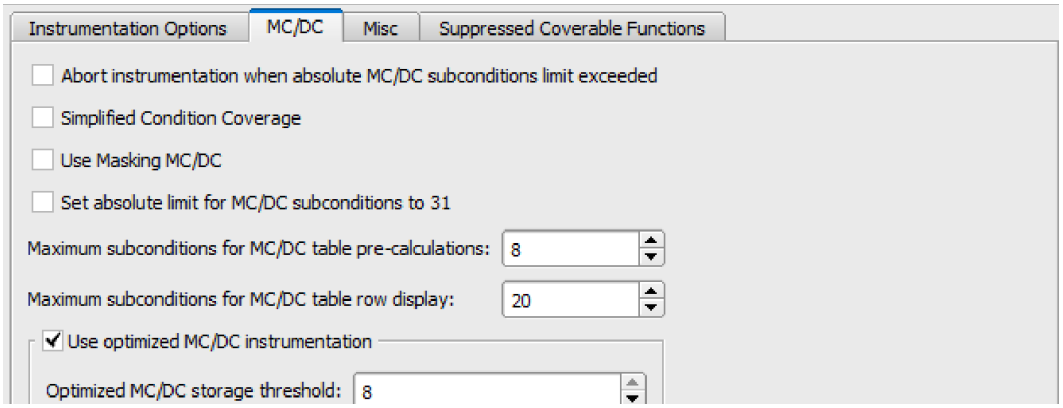
- Global initialization expressions contribute towards establishing a couple.
- Global initialization expressions contribute towards coverage.
- A read following an initialization does not cause the initialization expression to be marked as needing analysis.

```
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.
```

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.

After changing this option, you must reinitialize coverage.

```
clicast -lc option VCAST_MAX_MCDC_CONDITIONS <integer 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 536 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.

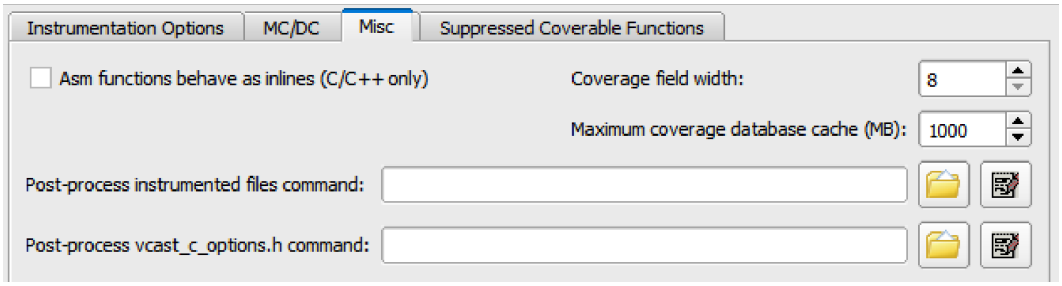
```
clicast -lc option VCAST_OPTIMIZED_MCDC_STORAGE_THRESHOLD 0..8

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.



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.
```

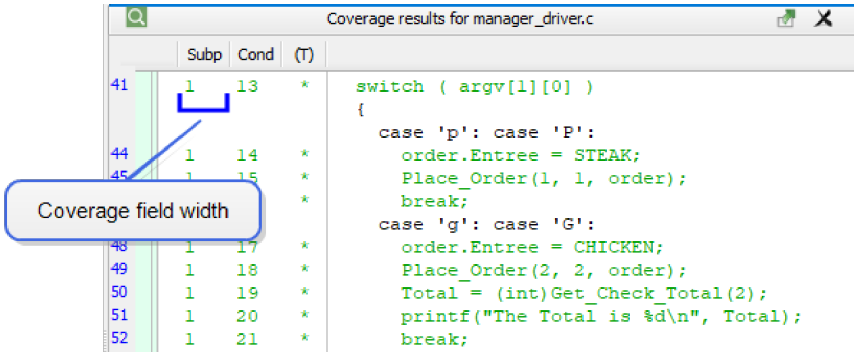
Coverage Field Width

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 C/C++ files instrumented prior to VectorCAST 2020.



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 -lc 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.

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.

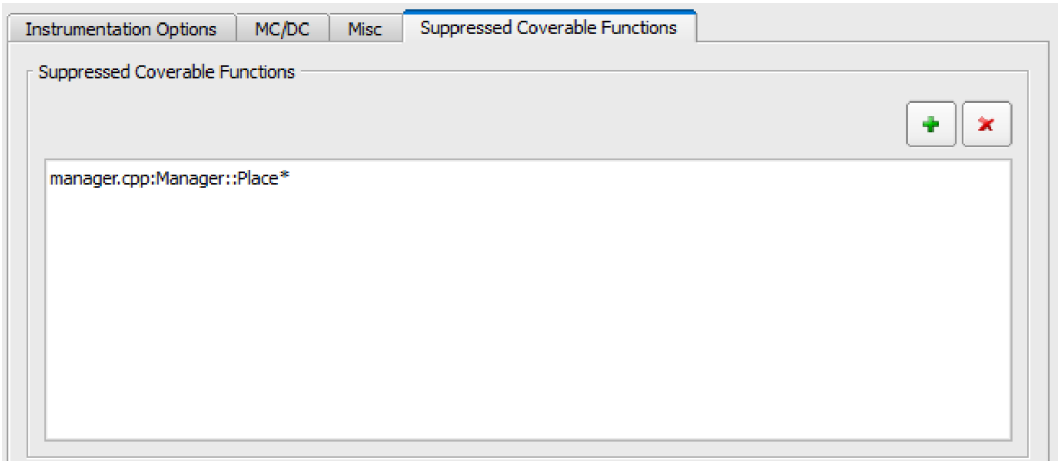
If you have an external text editor defined on the GUI tab, clicking the Edit File button ( ) opens the script in the specified external text edit.

```
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.



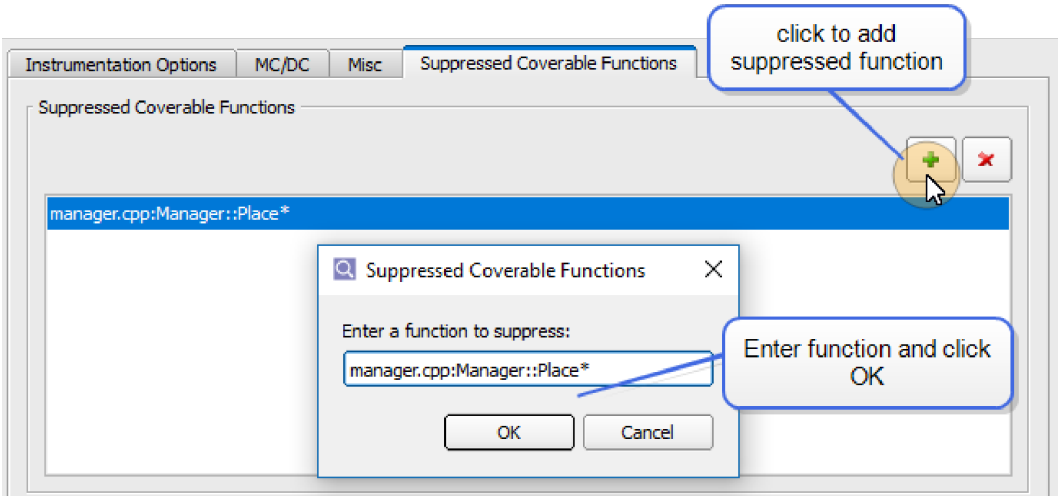
Suppressed Coverable Functions

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.

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`**

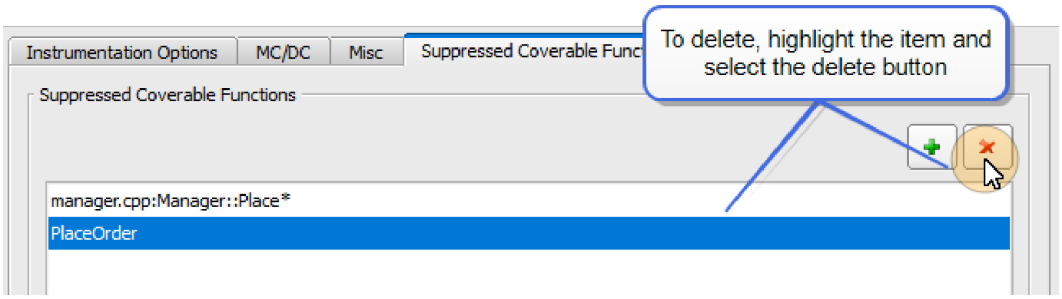
`C:/src/manager.c:*` – all functions in **`manager.c`**

`*/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>]

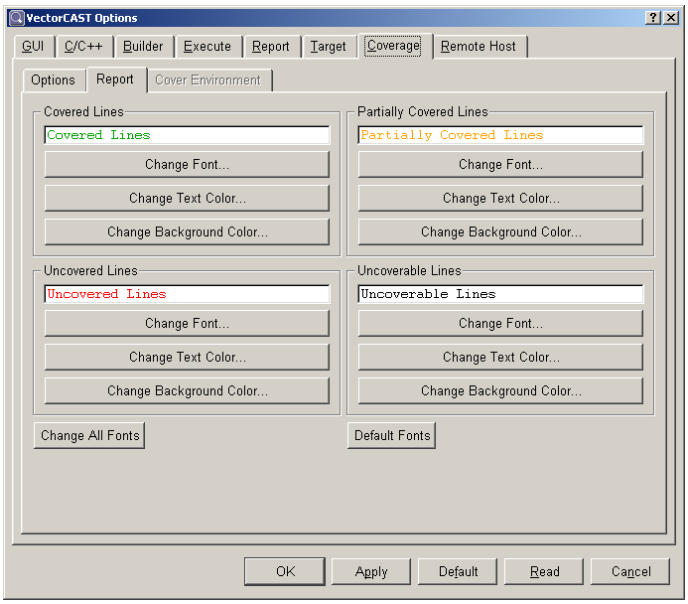
To suppress directories, files or functions from being instrumented, enter
lines in the form of [path:]function-name.
```

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 and the various coverage reports. 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.

Note: These options do not have a CLICAST counterpart; they are strictly for the Coverage Viewer.



To Format the Text in the Coverage Viewer

Choose **Tools => Options** and click the Coverage tab. Then click the Report sub-tab. Choose one of the four 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.

Uncovered Lines – lines or branches that have not yet been executed.

Uncoverable Lines – non-executable statements (e.g., comments).

To Reset the Fonts to Default

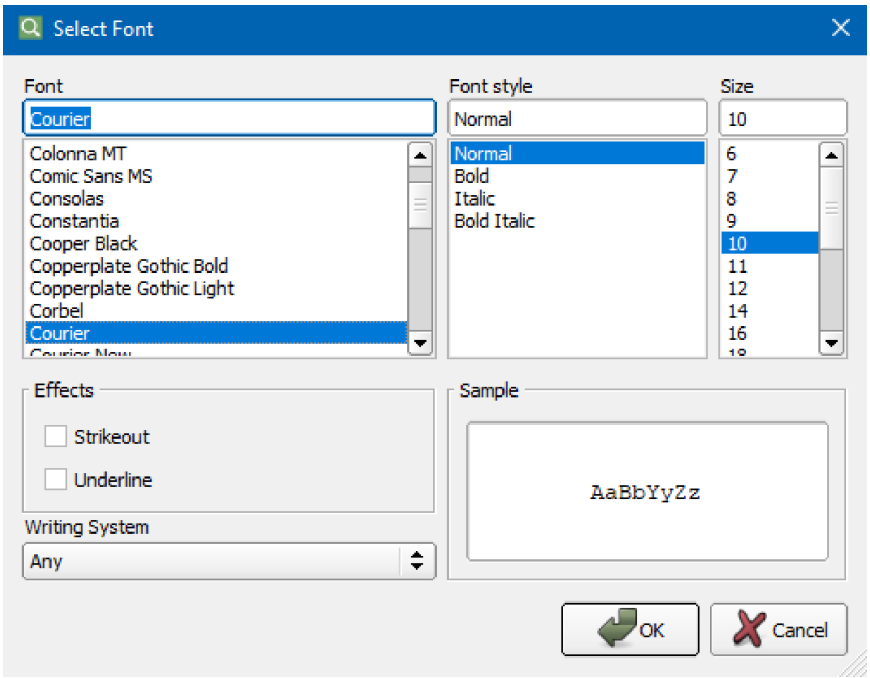
Choose **Tools => Options** and click the **Coverage** tab, then click the **Coverage Viewer** sub-tab.

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 the Fonts

Choose **Tools => Options** and click the **Coverage** tab, then click the **Coverage Viewer** sub-tab.

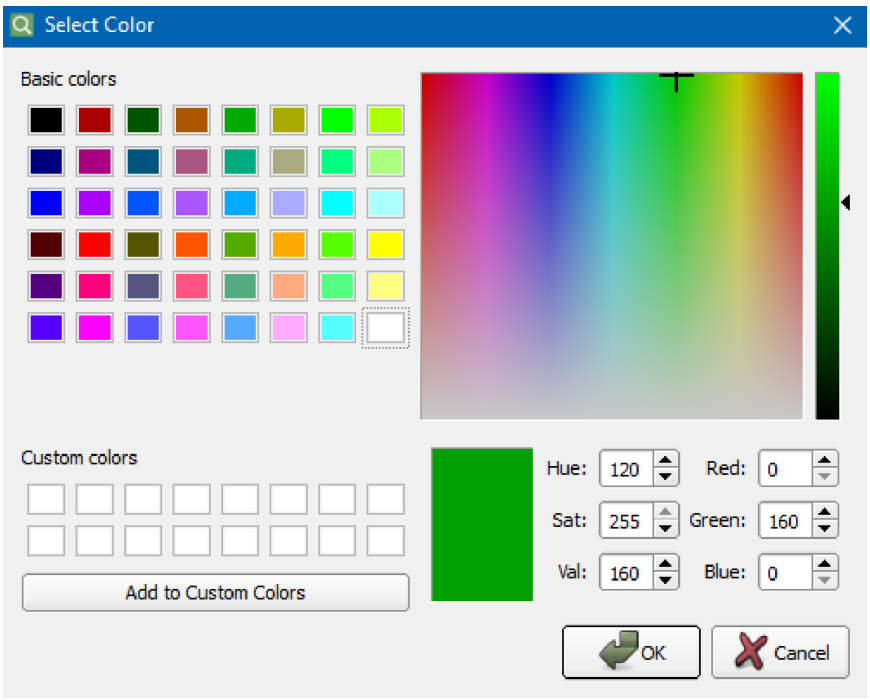
The **Font...** button for each line type and the **Change All Fonts...** button enable you to select the font, font style and size for one or all line types in the Coverage Viewer. The **Default Fonts** button enables you to revert to the default font settings. Default font settings are: Courier font, Normal Font style, 10 pt Size.



To Change Text Color

Choose **Tools => Options** and click the **Coverage** tab, then click the **Coverage Viewer** sub-tab.

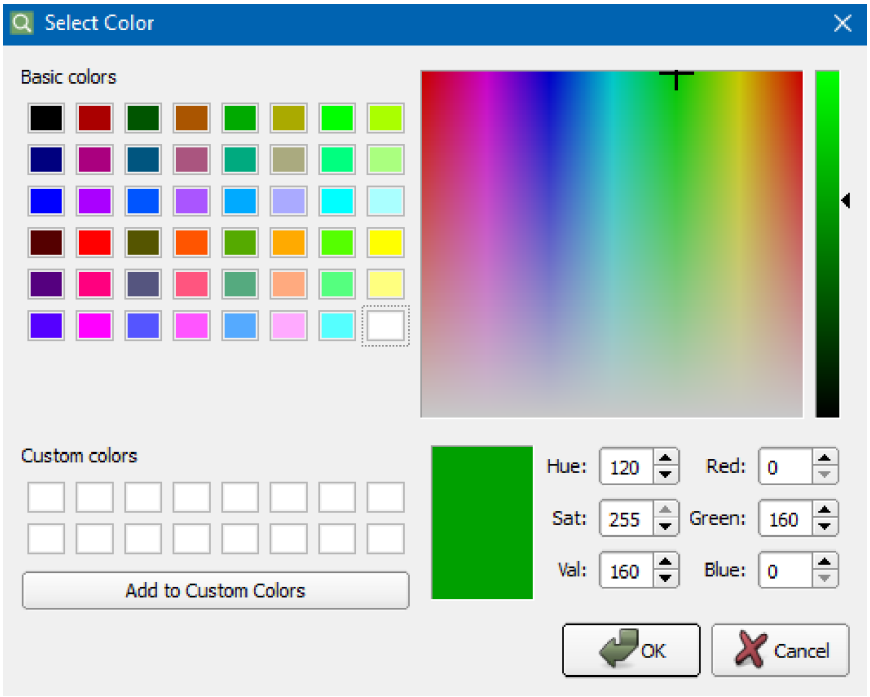
The **Text Color...** buttons enable you to change the text color for each of the five line types.



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.



Importing Coverage Results

The **Coverage => Import Results from Environment** feature is useful if you have used another VectorCAST environment to generate some coverage data, and want to import that data into the current test environment to increase the code coverage. Importing coverage data enables you to achieve complete coverage using any combination of unit, integration, and functional tests.

To import coverage results, choose **Coverage => Import Results from Environment**. If the menu item is dimmed, first initialize coverage by choosing **Coverage => Initialize => *coverage-type***. If coverage has been initialized but disabled, you must enable it for the menu item to be available.

Preparing to Import Results

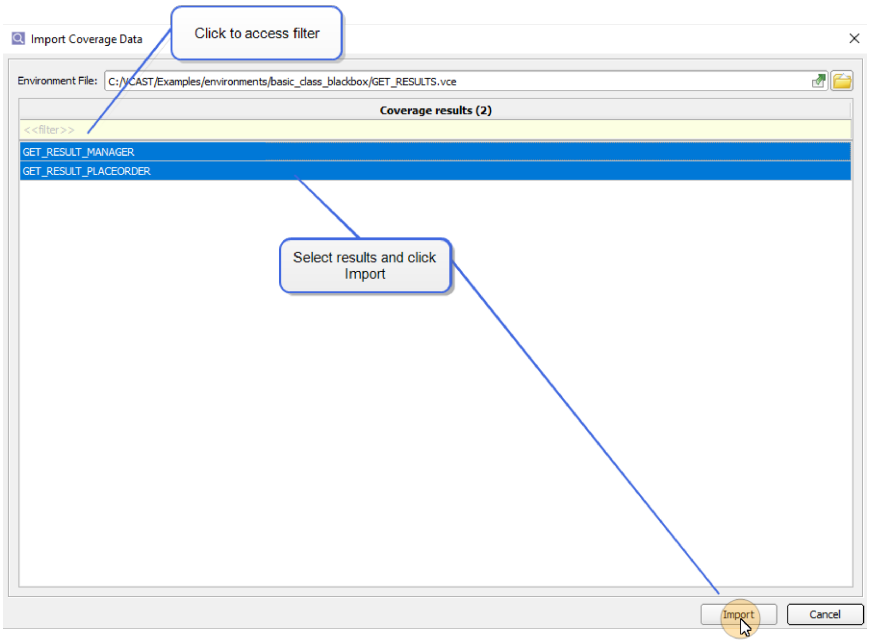
In order to import coverage, several conditions must be met.

- 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 or imported results or both.



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. Clear the filter by right-clicking in the top row and selecting **Clear Filter** from the context menu.

When you are ready to import the coverage results, click the **Import** button. To exit the dialog box without importing, click **Cancel**.

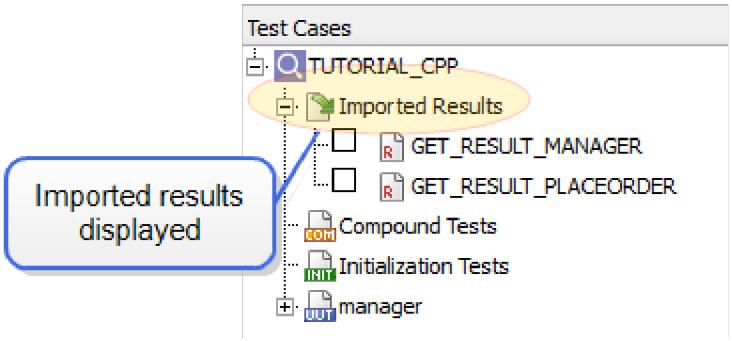
After the import is complete, a Coverage Import/Export log is displayed showing a series of status (S) or error (E) messages. Some of the messages you may see in the log file:

- **`(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 Coverage Import/Export Log file is named **IMPORT.LOG** and resides in the environment directory. You can view it at any time by selecting **Coverage => View Import Log**.

The imported results appear in the Test Case Tree, near the top. They are listed in a folder named "Imported Results", as shown below.



To Export a Script of Coverage

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>/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 unit test environments, that refers to test cases that have been executed. VectorCAST looks in the **<env>/COVERAGE_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 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.

An example of a coverage script (.cvr) file, below, exported from a Cover environment, shows the test result "Pasta" and the result "__COMPOUND__.001 which was imported from a unit test environment. The coverage script was created by choosing **Coverage => Export Script => All Results**.

```
-- VectorCAST 5.1b (06/22/10)
-- Imported Coverage Results Script
IMPORT.BEGIN
IMPORT.SOURCE.BEGIN
IMPORT.SOURCE.UNIT:1
IMPORT.SOURCE.ORIG_FILENAME:C:\vcast_tutorial\CPP\manager.cpp
IMPORT.SOURCE.TIMESTAMP:1277932110
IMPORT.SOURCE.COVERAGE_STATUS:TRUE
IMPORT.SOURCE.COVERAGE_TYPE:BRANCH
IMPORT.SOURCE.FILE_CHECKSUM:450167824
IMPORT.SOURCE.END
IMPORT.SOURCE.BEGIN
IMPORT.SOURCE.UNIT:2
IMPORT.SOURCE.ORIG_FILENAME:C:\vcast_tutorial\CPP\database.cpp
IMPORT.SOURCE.TIMESTAMP:1277932110
IMPORT.SOURCE.COVERAGE_STATUS:TRUE
IMPORT.SOURCE.COVERAGE_TYPE:BRANCH
IMPORT.SOURCE.FILE_CHECKSUM:3328281410
IMPORT.SOURCE.END
RESULT.NEW
RESULT.NAME:IMPORTED_RESULTS\__COMPOUND__.001
RESULT.DATA:2 1 0 T
RESULT.DATA:2 1 0 T
RESULT.DATA:1 1 0 T
RESULT.DATA:1 3 0 T
RESULT.DATA:2 3 0 T
RESULT.DATA:1 2 0 T
RESULT.DATA:1 2 1 F
RESULT.DATA:1 2 4 T
RESULT.DATA:1 3 2 T
RESULT.DATA:2 4 0 T
RESULT.DATA:1 3 0 T
RESULT.DATA:2 3 0 T
RESULT.DATA:1 2 0 T
RESULT.DATA:1 2 1 F
RESULT.DATA:1 2 4 F
```

```
RESULT.DATA:1 2 6 F
RESULT.DATA:1 3 4 T
RESULT.DATA:2 4 0 T
RESULT.DATA:1 3 0 T
RESULT.DATA:2 3 0 T
RESULT.DATA:1 2 0 T
RESULT.DATA:1 2 1 F
RESULT.DATA:1 2 4 F
RESULT.DATA:1 2 6 F
RESULT.DATA:1 3 6 T
RESULT.DATA:2 4 0 T
RESULT.DATA:1 3 0 T
RESULT.DATA:2 3 0 T
RESULT.DATA:1 2 0 T
RESULT.DATA:1 2 1 F
RESULT.DATA:1 2 4 F
RESULT.DATA:1 2 6 F
RESULT.DATA:1 3 8 T
RESULT.DATA:2 4 0 T
RESULT.DATA:2 3 0 T
RESULT.DATA:1 5 0 T
RESULT.DATA:2 3 0 T
RESULT.END
RESULT.NEW
RESULT.NAME:IMPORTED_RESULTS\Pasta
RESULT.NOTES
Order Pasta for dinner.
RESULT.END_NOTES
RESULT.REQUIREMENTS
000000a1/13
RESULT.END_REQUIREMENTS
RESULT.DATA:2 1 0 T
RESULT.DATA:1 1 0 T
RESULT.DATA:1 3 0 T
RESULT.DATA:2 3 0 T
RESULT.DATA:1 2 0 T
RESULT.DATA:1 2 1 F
RESULT.DATA:1 2 4 F
RESULT.DATA:1 2 6 F
RESULT.DATA:1 3 8 T
RESULT.DATA:2 4 0 T
RESULT.DATA:2 2 0 T
RESULT.END
IMPORT.END
```

clicast -e <env> TOols 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> TOols EXPORT_Coverage <scriptfile>

Create a coverage script containing only imported coverage results that are
present in <env>.
```

```
clicast -e <env> TOols EXPORT_Testcase <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.

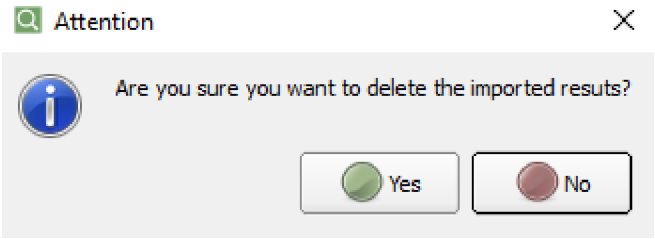
```
clicast -e <env> TOols Import_coverage <scriptfile> | <env>

Import a coverage script named <scriptfile> or import all coverage results
from <env>.
```

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**.

To Delete Imported Results

To delete all imported coverage results, choose **Coverage => Delete Imported Results**. A dialog appears asking you to verify that you want to continue with this operation, as shown below.

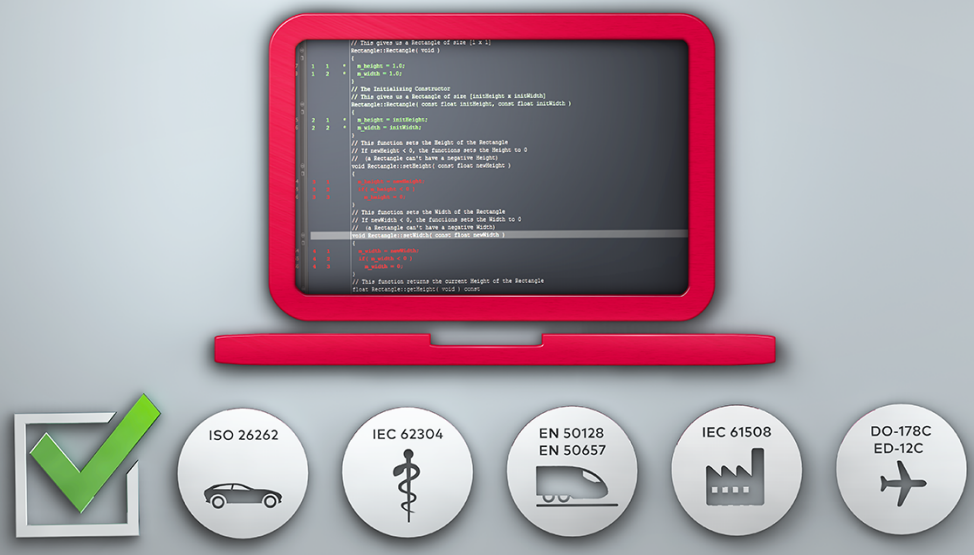


Clicking **No** cancels the operation without removing the imported coverage results. Clicking **Yes** removes the imported coverage results from the environment and the **IMPORTED_RESULTS** directory from the environment directory. There is no clicast command to delete all imported results.

There is no clicast command to delete all imported results.

To View the Coverage Import Log

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**.



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.

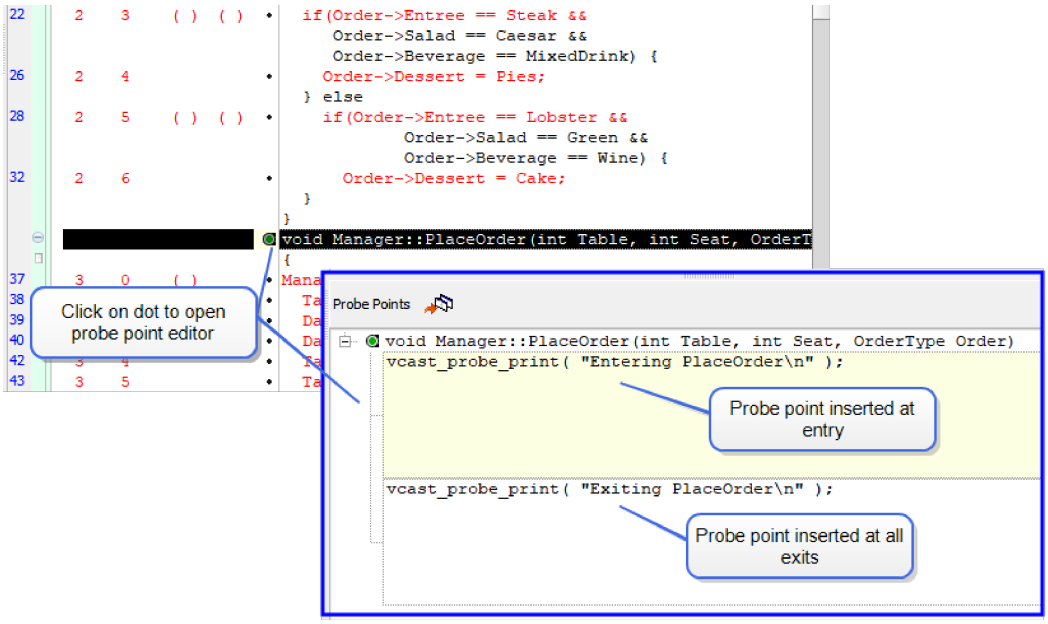
Probe points are frequently used in the following testing scenarios:

- 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).

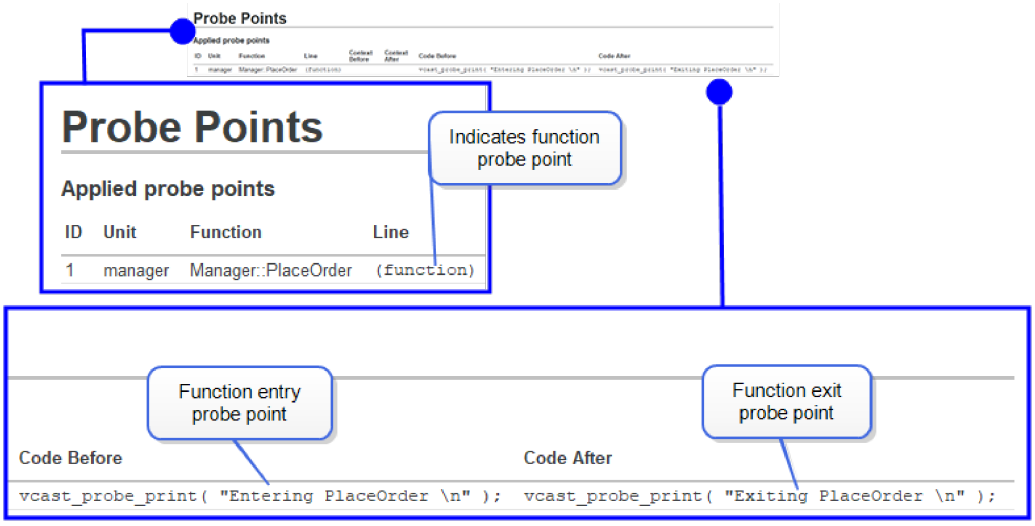
Entry and Exit function probe points can be entered via the Probe Point Editor. Small black dots ● in the left-hand column of the Coverage Viewer indicate a possible probe point location. To insert a probe point, click on the black dot next to the executable line where you want to insert the probe point. 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

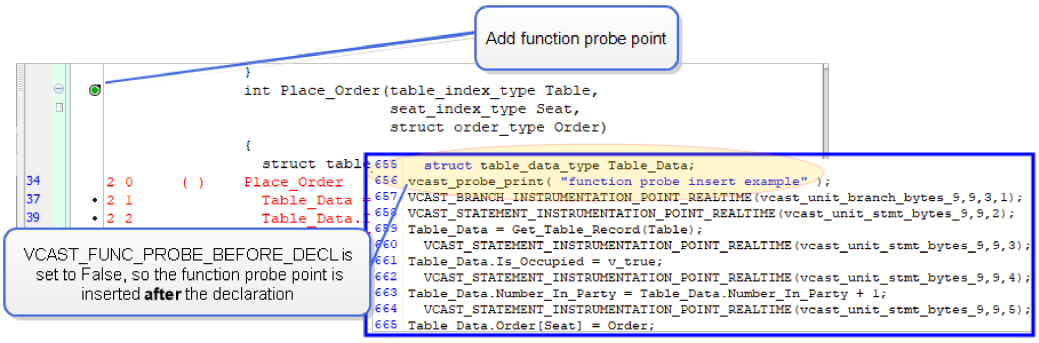
Function entry probe points are inserted just inside the opening curly brace in C++, or after the declarative section in C. This clicast-only option is provided to instruct VectorCAST to insert the function probe point before the declarative section for C sources.

```
clicast -lc option VCAST_FUNC_PROBE_BEFORE_DECL True | False

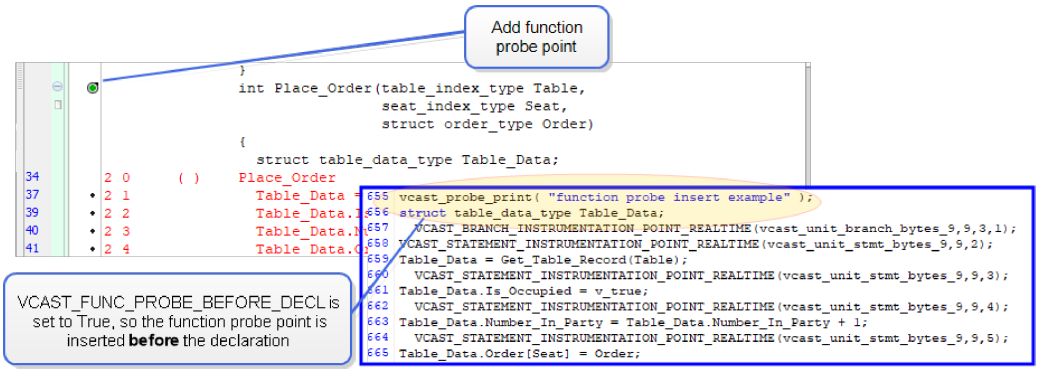
Set this option to True to force the insertion of function based probe
points before the declarative region of the function in C. The default
value is False.
```

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

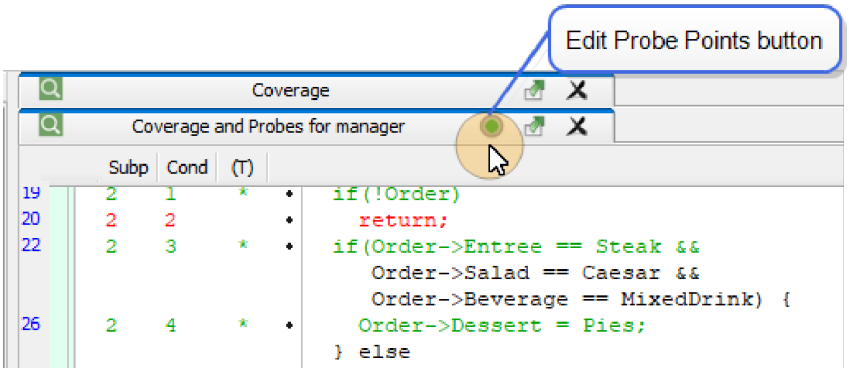
Open the Probe Point Editor

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:

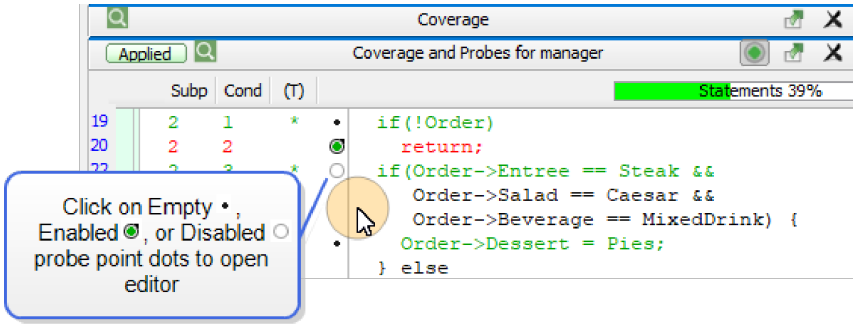
> Select a unit in the Test Case Tree and click the **Edit Probe Points** button on the Toolbar.

> Right-click the unit or subprogram in the Test Case Tree and select **Edit Probe Points** from the context menu.

> In the Coverage Viewer, click the **Edit Probe Points** button located on the tab.

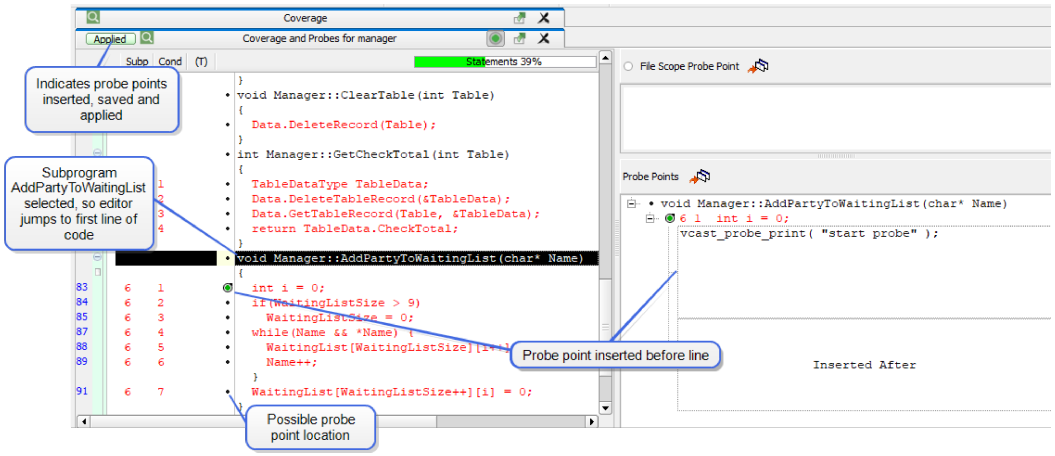


> In the Coverage Viewer, click on any Probe Point Dot. Any of the states (Empty, Enabled, or Disabled) will open the Editor.



The Coverage Viewer opens in the MDI area in the left pane and displays the coverage and probe points for the selected unit. The Probe Point Editor opens in the right pane.

If a subprogram has been selected in the Test Case Tree, the coverage viewer pane jumps to the first line of code for the selected subprogram.




```
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.
```

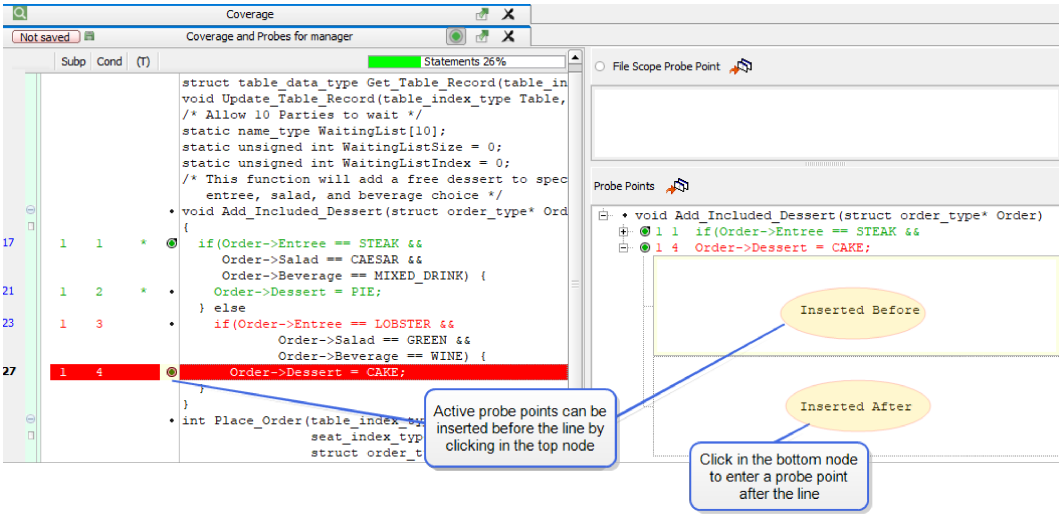
Insert a Probe Point

Probe points are associated with one unit. To open the Probe Point Editor and add a new probe point, select a unit in the Test Case Tree 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

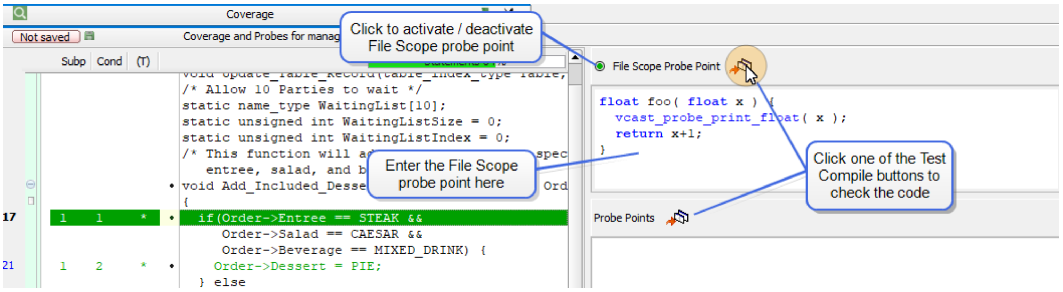
VectorCAST Probe Point supports adding a probe point at file scope. Only one File Scope probe point can be entered per unit. Any valid code is accepted, such as new function definitions or global variables.

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 a unit in the Test Case Tree 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 559 for more information.

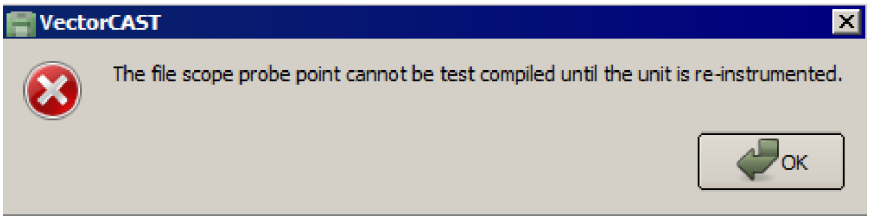
When you save and apply the probe points for the unit, the File Scope probe point is inserted in the instrumented source file just after the last #include line.

File Scope probe points are written to the regression scripts along with the regular function scope probe points.

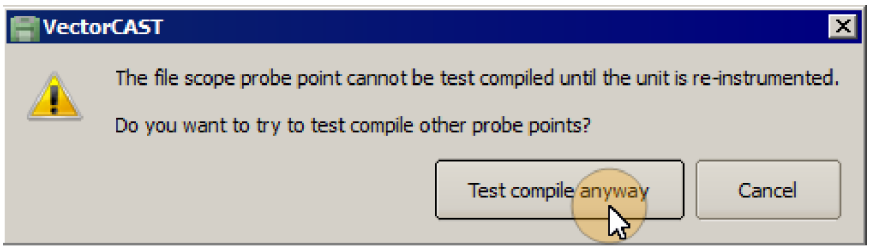
Using File Scope Probe Points with Older Environments

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

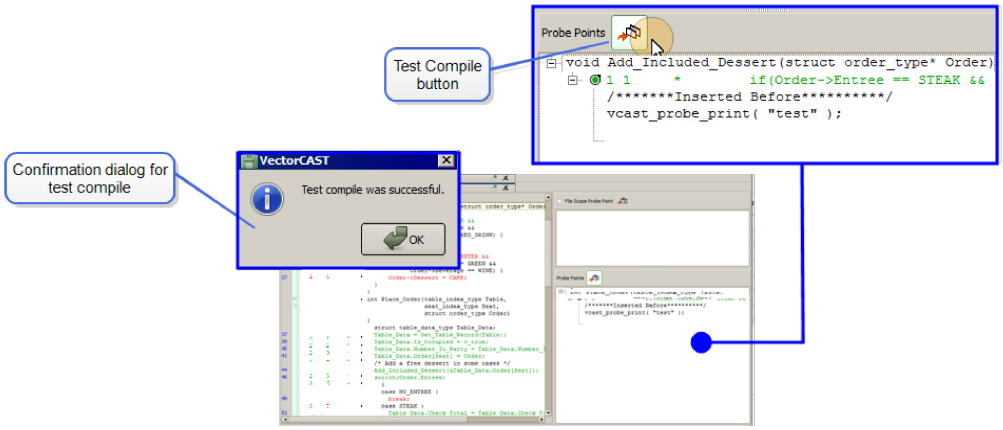
A **Test Compile** button  is located in the upper right of the Probe Point Editor and the File Scope Probe Point Editor. This button performs a test compile of all active probe points in the unit. Note that deactivated probe points are not compiled.

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 559.

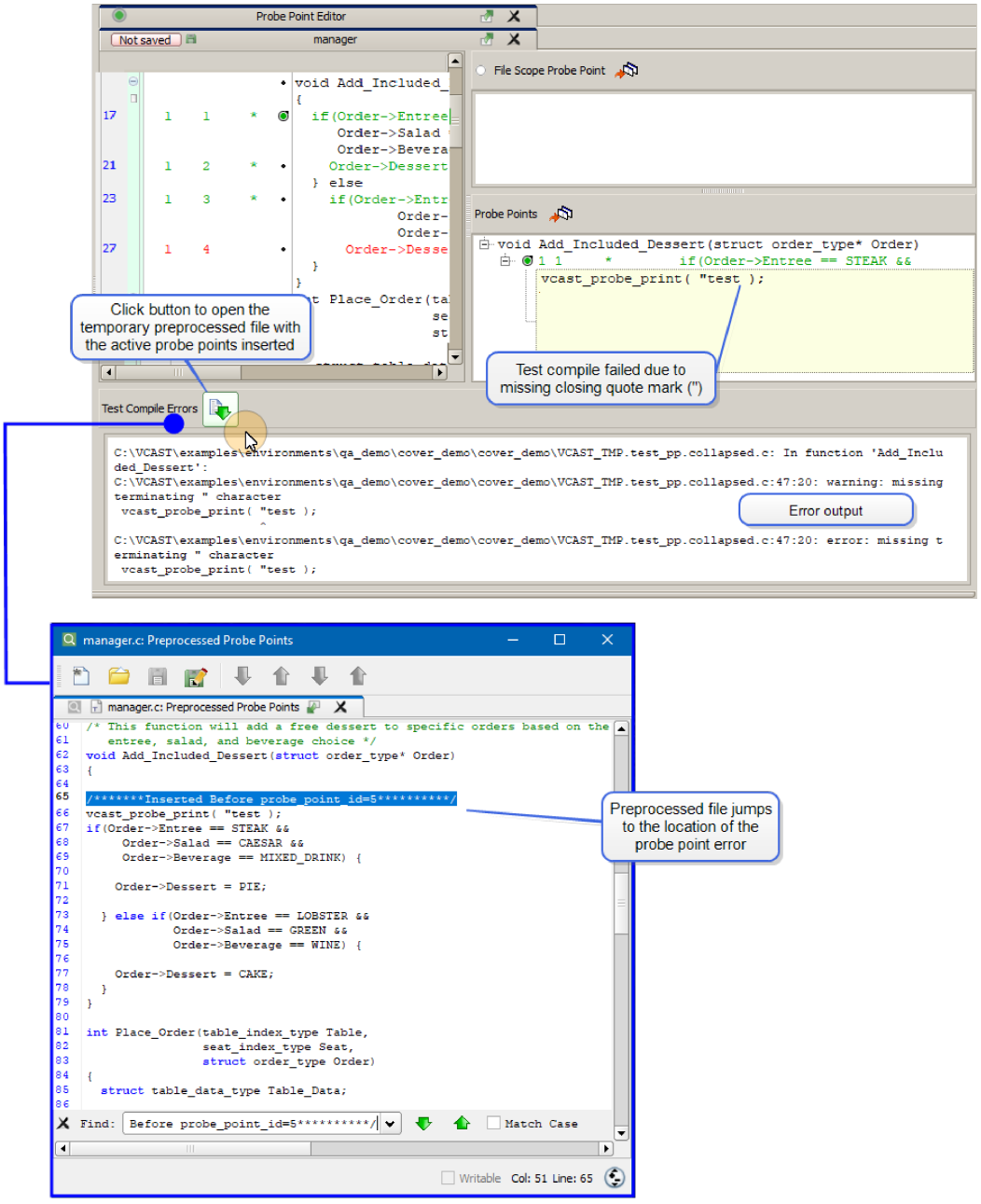
A test compile can be performed on a probe point in any status. It can be Not Saved, Not Applied, or Applied. See "Probe Point Status Buttons" on page 563 for more information.

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.

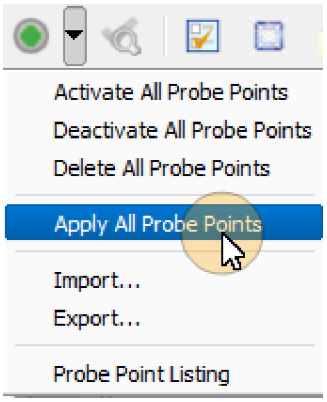
Applying causes the unit to be parsed, instrumented, and compiled in both the test harness and instrumented test harness, and then both harnesses are relinked. Changes can also be applied by performing either a full rebuild or an incremental rebuild of the environment. Once applied, the probe point text appears in the instrumented version of the UUT, in the environment directory.

```
clicast -lc -e <env> TOols Probe_point Apply

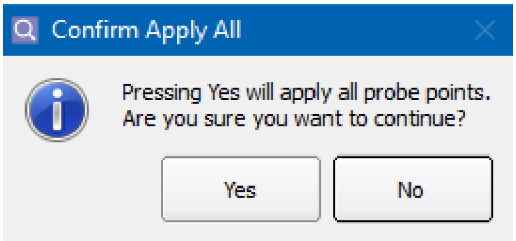
Applies the probe points. Causes the unit(s) with the probe point to be
parsed, instrumented, compiled in both the test harness and the
instrumented test harness, and both test harnesses to be relinked.
```

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.

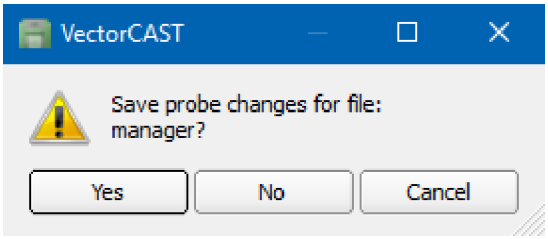


A confirmation dialog appears to confirm the action.



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 an Incremental Rebuild is performed.

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 points by performing an incremental rebuild and link of the environment.
Applied	= Probe points applied.

Note: Clicking the status buttons saves changes and performs reinstrumenting only on the individual unit currently in focus in the Probe Point Editor. For scenarios where there are changes to multiple units in multiple Probe Point Editors, use the **Save All** button in the Toolbar and select **Environment => Incremental Rebuild** to more efficiently apply multiple probe points.

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.

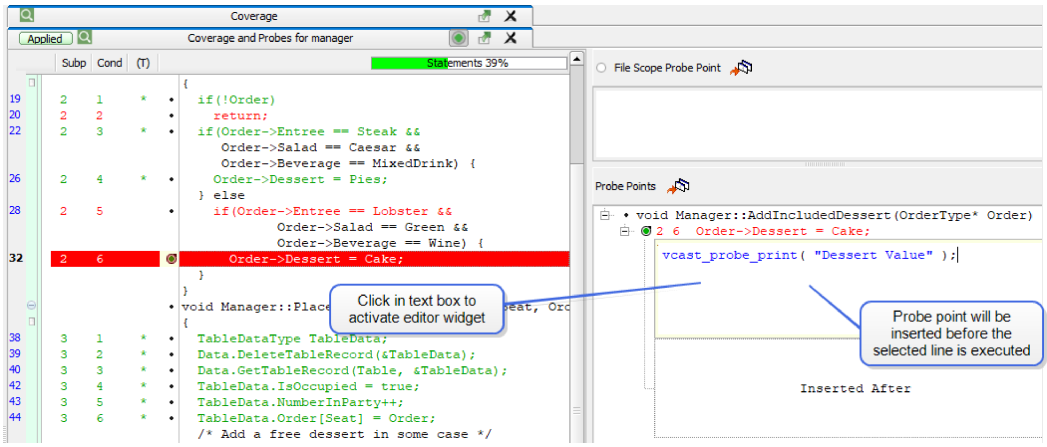
The keyboard shortcuts Select All (**Ctrl+A**) and Copy (**Ctrl+C**) are also available from the context menu in the Coverage View pane.

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 a unit in the Test Case Tree and select **Edit Probe Points** from the context menu. Alternatively, select a unit in the Test Case Tree 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

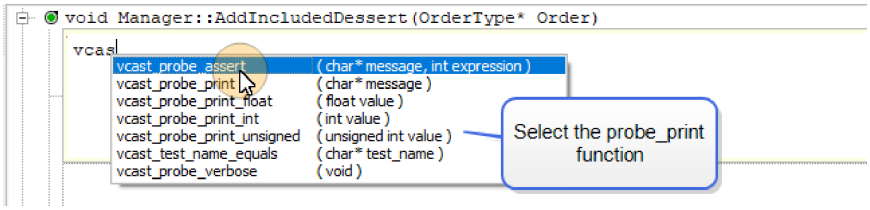
The Probe Point Editor provides built-in functions and macros for ease of use in creating probe points.

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 stdout for Unit Test 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_assert();`

This function allows the user to make an assertion in a probe point, which is then evaluated during test execution. Use the syntax **`vcast_probe_assert`**

("description", expression); where **"description"** is a string and **"expression"** is an int (C/C++) or boolean (C++).

When the probe point is executed, an event is added to the Execution Results Report. The event states the assertion's description and whether the assertion was true (a match) or false (a failure). This type of probe point increases the number of expected values that are counted as part of the overall status of the test case. Probe point assertions do not affect Control Flow.

```
vcast_probe_print();
```

This function causes its standard text argument to be captured and written to the Execution Results Report. Only character strings with double quotes are accepted.

```
vcast_probe_print_float();
```

This function causes its floating point argument to be captured during test execution and written to the Execution Results Report. Only floating point values are accepted.

```
vcast_probe_print_int();
```

This function causes its integer argument to be captured during test execution and written to the Execution Results report. Only integer values are accepted.

```
vcast_probe_print_unsigned();
```

This function causes its unsigned argument to be captured during test execution and written to the Execution Results Report. Only unsigned int values are accepted.

```
vcast_test_name_equals (char*test_name)
```

This function supports probe point code that is dependent on a test name. The function returns a Boolean True if the string matches the current test name, and a Boolean False is returned if the string does not match the current test name. This function works with simple and <<INIT>> test cases, but not with compound test cases.

```
vcast_probe_verbose (void)
```

This function enables printing of probe code in probe point events when "Consider probe points as events" is True.

Deactivate/Activate Probe Points

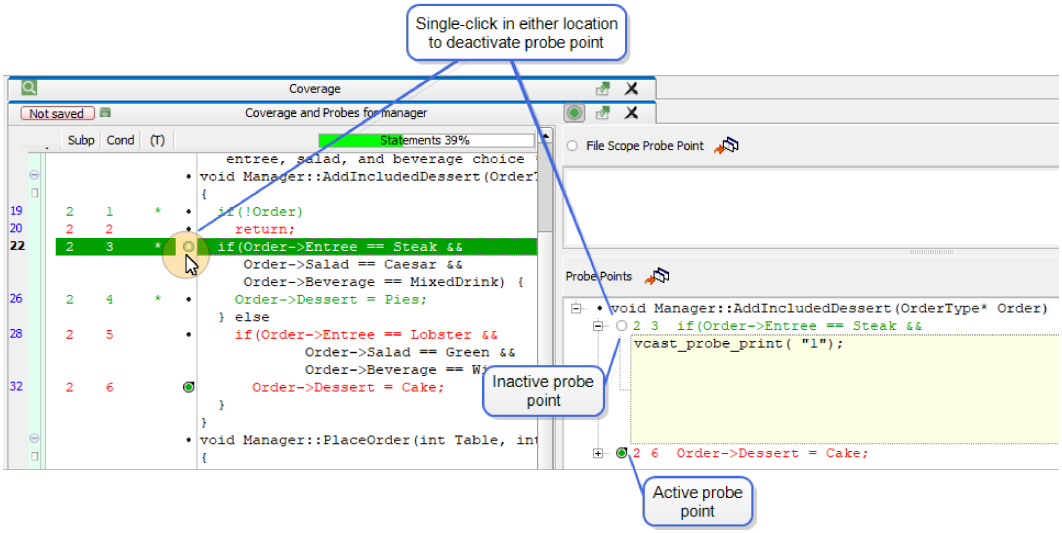
Clicking on the probe point icon in either the Coverage Viewer or the Probe Point Editor toggles the probe point between the active and inactive states. To deactivate an individual probe point, single-click on the active probe point icon . The inactive probe point icon is displayed in both the Coverage 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.

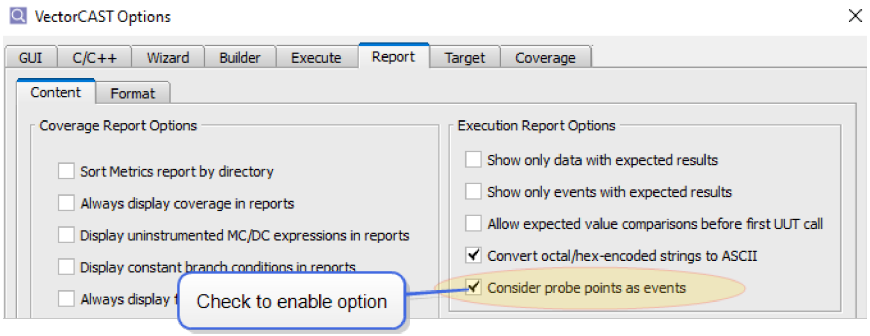
Note: Selecting the **Delete All Probe Points** option permanently removes *all* probe points in *all* units in the environment, and cannot be undone.

```
clicast -lc -e <env> TOols Probe_point REMove_all

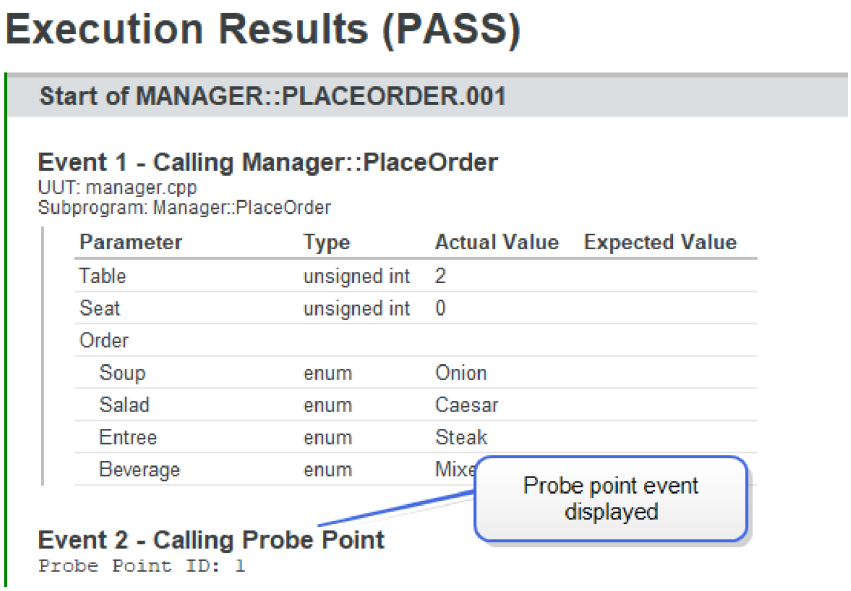
Remove all active and inactive probe points from the environment and
incrementally rebuild.
```

Probe Point Events

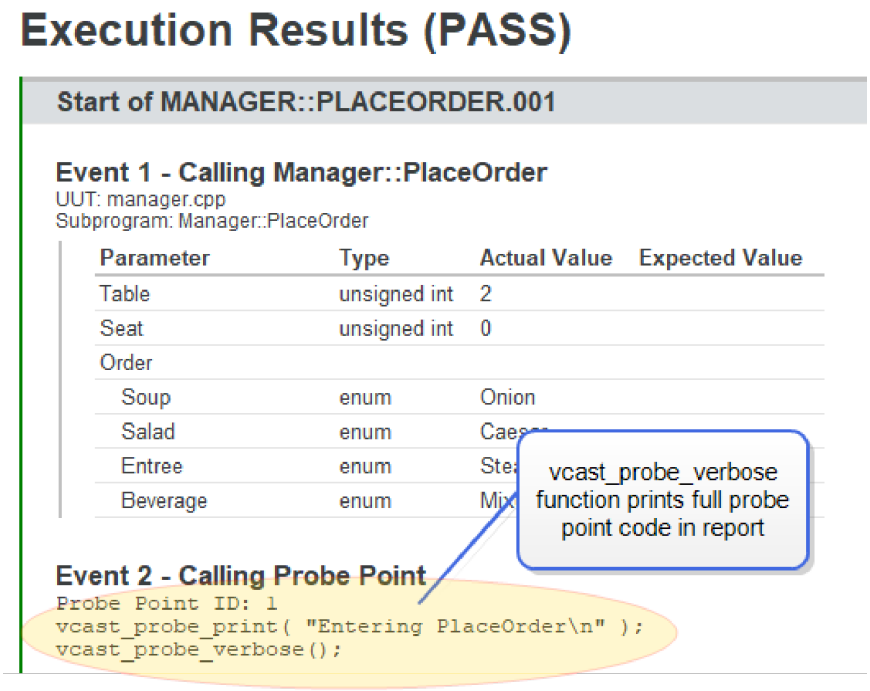
VectorCAST provides a Report option called "Consider probe points as events". This option is disabled by default. When enabled, this option directs VectorCAST to include a probe point call in the Execution Report as a separate event for each executed probe point. To enable this option, choose **Tools => Options**, and click the **Report** tab. Then click the **Content** sub-tab and the **Execution Report Options** pane.



In the example below, the calling probe point is listed as an event and is identified by probe point ID in the Execution Results report.



If a call to the support function **`vcast_probe_verbose()`** is added in the probe point code, VectorCAST prints the full text of the probe point inline in the event.



Note: A Probe Assert (**`vcast_probe_assert()`**), continues to display as its own event with a <match> or failure, distinct from the call to the probe point as an event.

If an environment contains many probe point calls when the option is True, the Event Limit could be

reached earlier than anticipated, and may need to be increased for complete test execution.

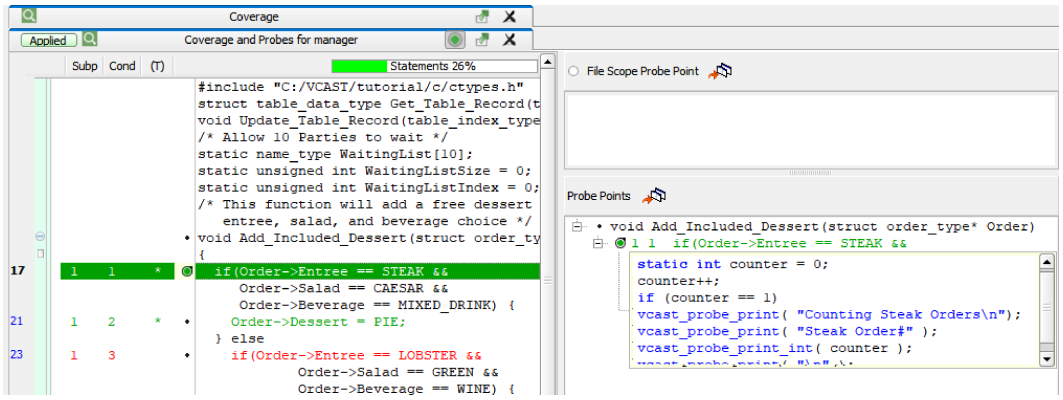
```
clicast -lc option VCAST_PROBE_POINTS_AS_EVENTS True | False

Set this option to have an event generated at each probe point. The default
value is False.
```

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 Test Execution Report and can be used as a debugging aid.

Note: You must have the Redirect Standard Output option, STANDARD_OUTPUT, set to **`Redirect`** in order to redirect the execution output to a file. The captured probe point text is appended to the end of the Execution Report.

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

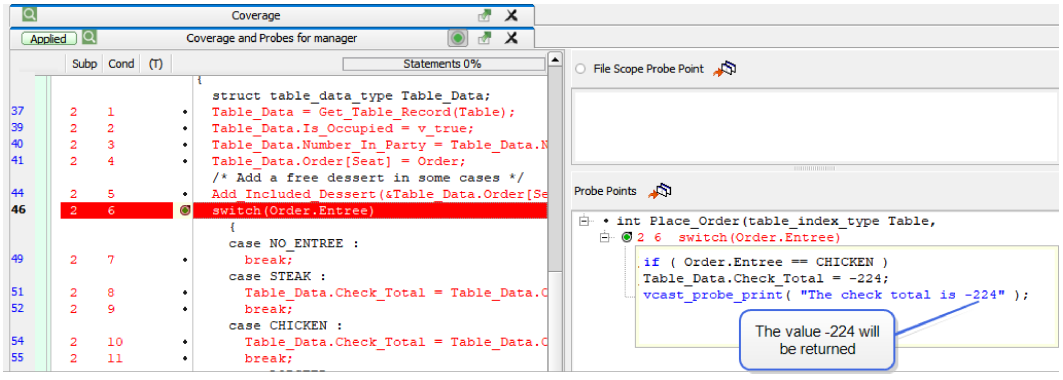
Result - PASS

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.

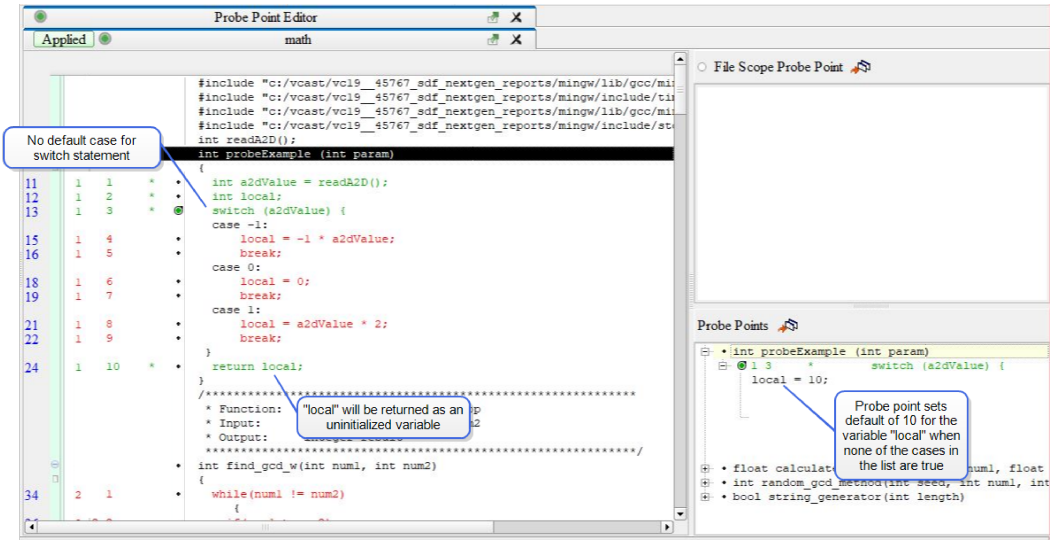


The probe point is saved and applied. Note that when the function is executed it returns a value of -224.

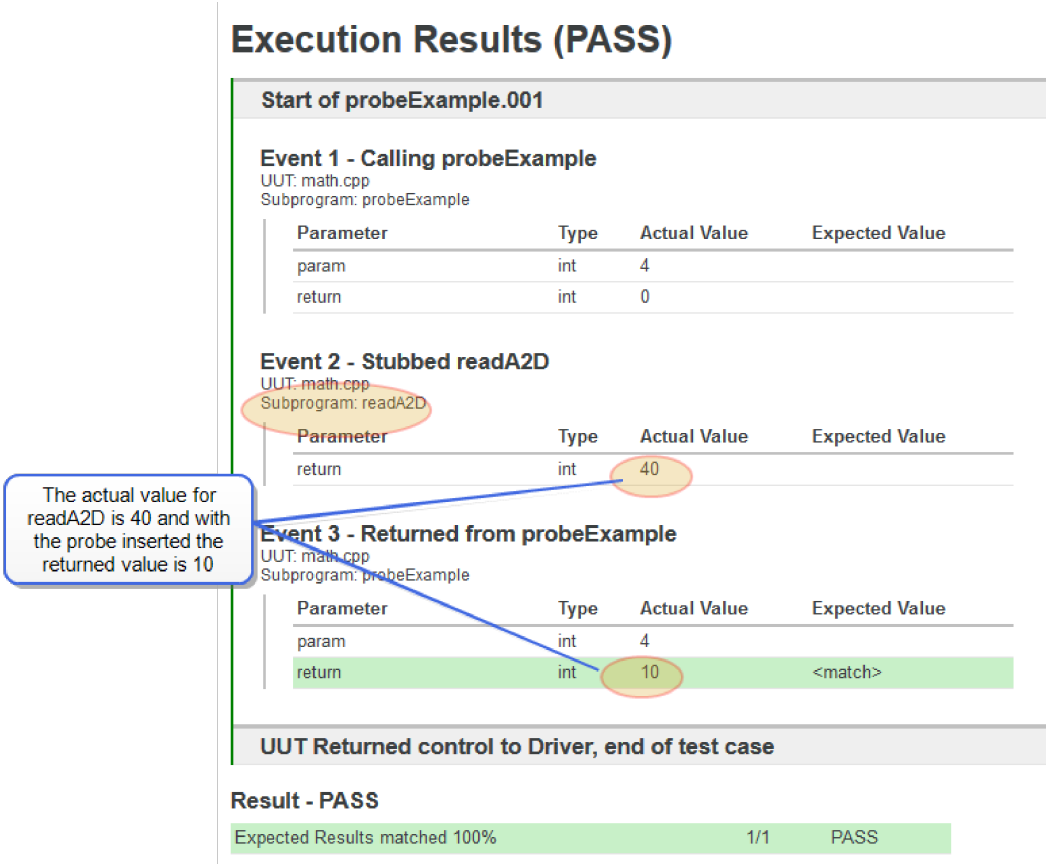
Patch Faulty Code

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.



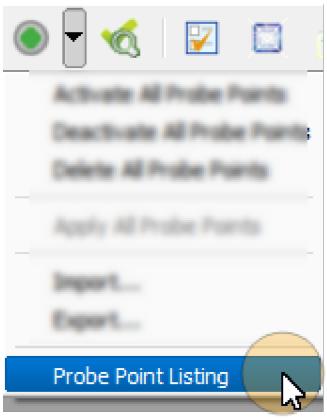
To fix this bug, we insert a probe point that initializes **`local`** to a known value of 10 and then verify that the value 10 is returned when the param is out of bounds. When the test is executed, the results show that the actual value for **`readA2D()`** is 40, which does not match the case list, so the default value of 10 is returned.



Now that the patch is verified, the changed code can be committed.

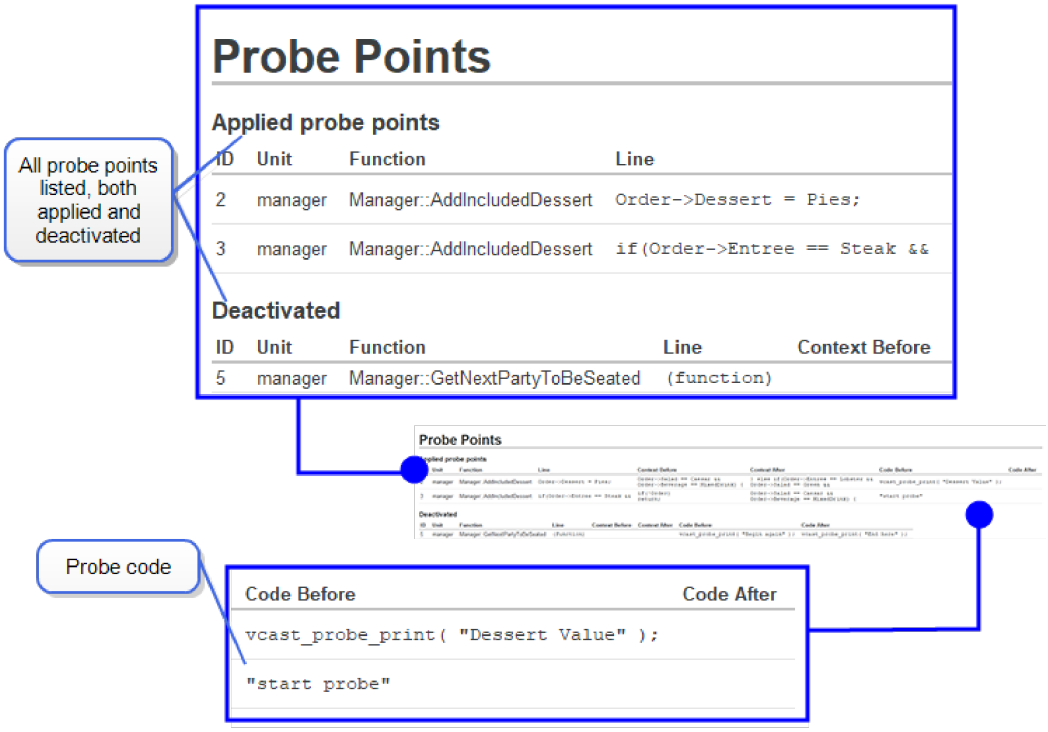
Probe Point Listing

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.




```
clicast -lc -e <env> ENvironment Extract Probe_point_log <outputfile>

Extract the probe point listing to standard output or to a file if
specified.
```

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

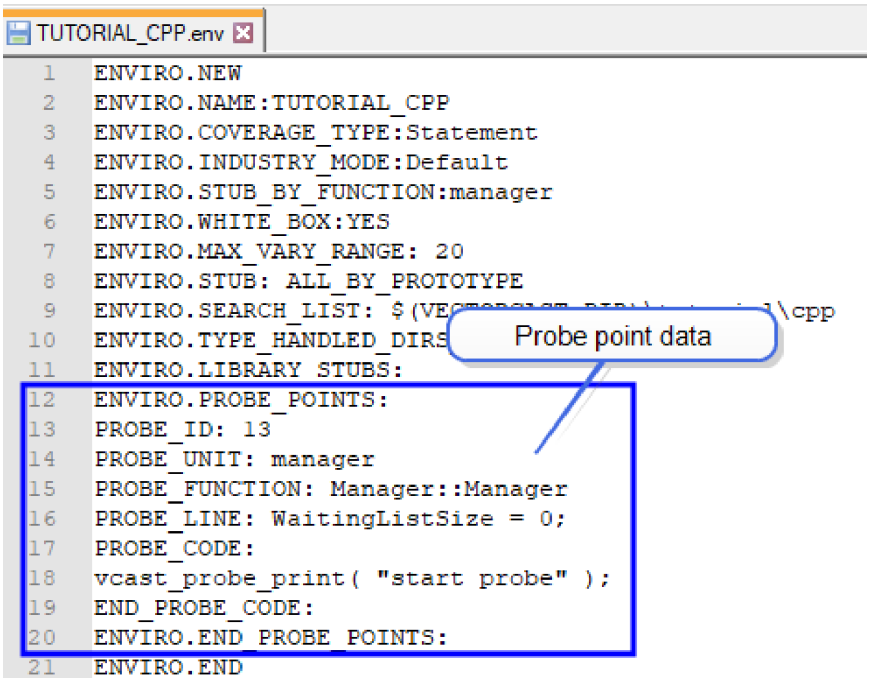
The probe point file is a result of building the regression script. A probe point is first created (either via the GUI or via CLICAST), and then saved and applied. Next, a regression script is generated by selecting **Environment => Create Regression Scripts...** from the Menu Bar.

The probe point file can reside anywhere. In our example, it resides in our environment directory: **`C:\VCAST\Examples\environments\tutorial_cpp`**.

Once the regression script is generated, you will see three files in the selected directory:

- **<env-name>.bat**
- **<env-name>.env**
- **<env-name>.tst**

The probe point data is contained within the **<env-name>.env** file. In our example, we have opened the **TUTORIAL_CPP.env** file. The definition of our probe point begins at line 12.



The user can modify the probe point data contained in the file.

Probe Matching Line Index

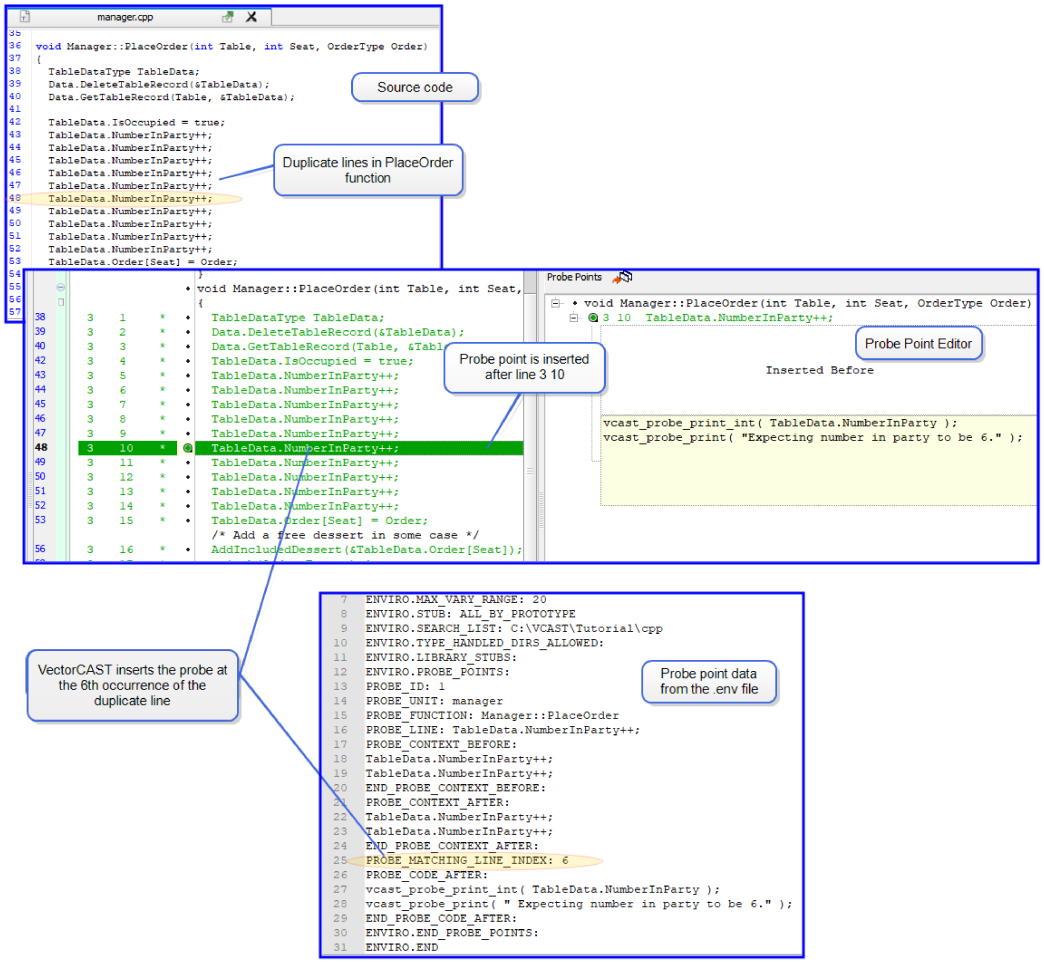
By default, when a probe point is applied using the Probe Point Editor or imported from a file, VectorCAST uses two lines of context around the line containing the probe point to differentiate between duplicate lines of code in the function.

In cases where there are duplicate lines of code in the function, it is best practice to utilize the PROBE_MATCHING_LINE_INDEX entry in the Probe Point file to indicate whether a probe point is on the first, second, or Nth occurrence of that particular duplicate line.

The duplicate line specification is employed during any action that performs a Probe Point Apply, such as Apply in the Probe Point Editor, import, and environment rebuild.

Note: When inserting probe points to lines that are duplicated throughout the function and have similar context, it is important to be careful when modifying the source code. If another duplicate line with the same context is added before the intended line, VectorCAST cannot discern the correct location. It is recommended to use comments in the source code in these situations to provide VectorCAST with clear context lines so that probe points are not inadvertently applied in the wrong location.

In the example below, the function PlaceOrder contains 10 duplicate lines. A probe point is placed at line 3 10. VectorCAST detects the duplicate lines of code, and identifies that the probe point is located at the 6th occurrence of the duplicate line: PROBE_MATCHING_LINE_INDEX: 6.



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>`

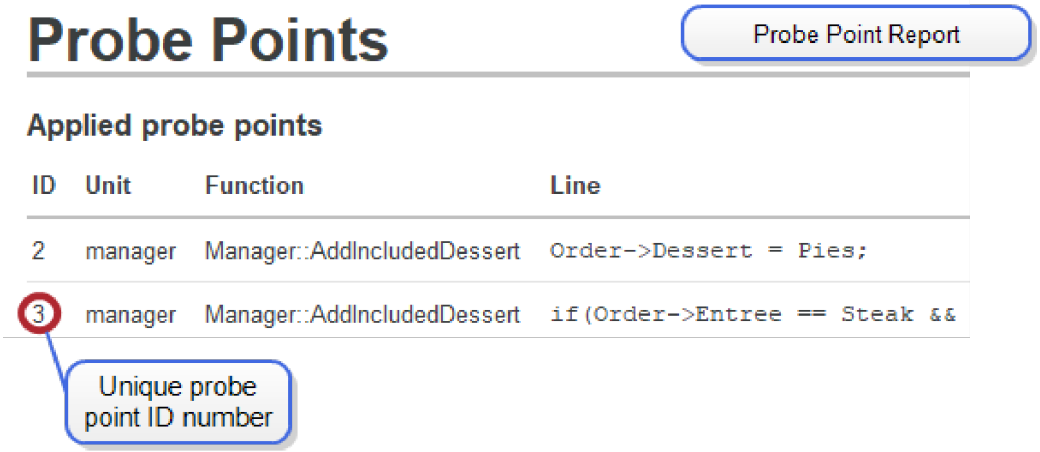
positioned after the definition of the probe point in the .env script or probe point file.

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.

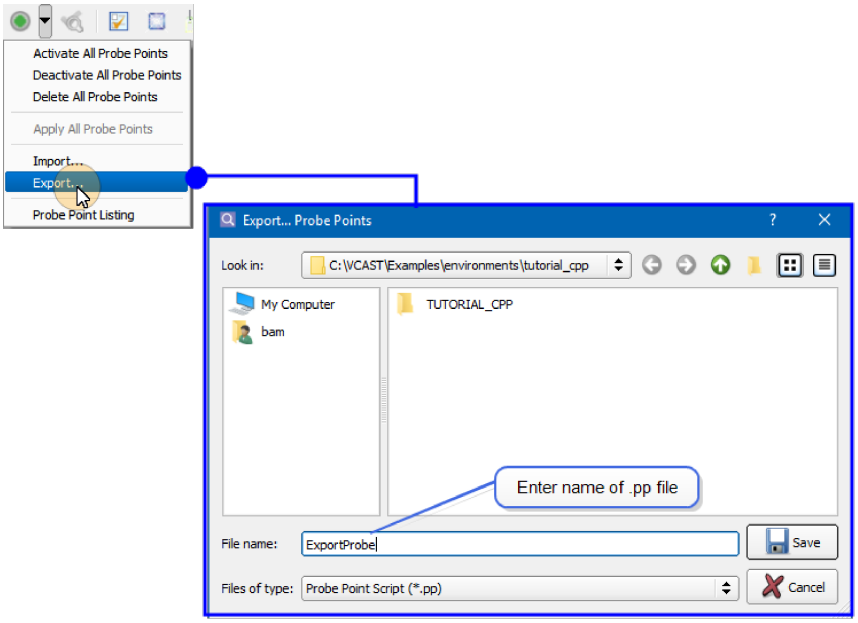
When a probe point is deleted or dropped due to changes in the source code, its ID is also deleted and can be re-used for another probe point.

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.

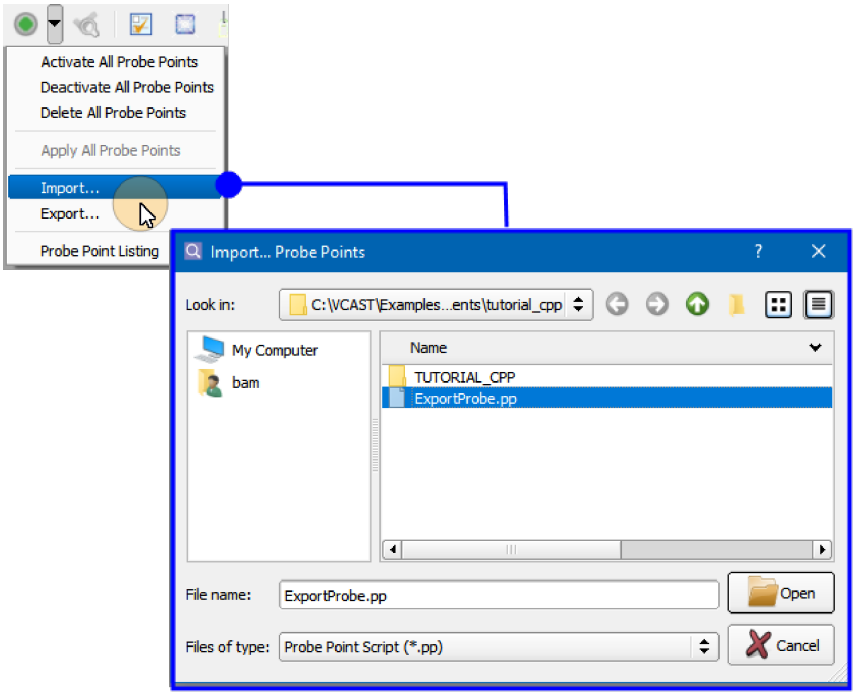



```
clicast -lc -e <env> TOols PRobe_point EXport_file <probe_point_file>

Exports probe points specified in a .pp file. Requires 'clicast -e <env>
TOols 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, the environment is rebuilt incrementally and the Probe Point Import Log is displayed.

```
clicast -lc -e <env> TOols PRobe_point IMPort_file <probe_point_file>

Adds probe points specified in a .pp file. Requires 'clicast -e <env> TOols
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> TOols PRobe_point ENAble_id <probe_point_id>

Enable (activate) the probe point specified by <probe_point_id> from the
environment. Requires 'clicast -e <env> TOols PRobe_point APPly' to be run
in order to take effect.
```

Disable Probe Point

```
clicast -lc -e <env> TOols PRobe_point DISable_id <probe_point_id>

Disable (deactivate) the probe point specified by <probe_point_id> from the
environment. Requires 'clicast -e <env> TOols PRobe_point APPly' to be run
in order to take effect.
```

Remove Probe Point

```
clicast -lc -e <env> TOols PRobe_point REMOVE_Id <probe_point_id>

Remove the probe point specified by <probe_point_id> from the environment.
Requires 'clicast -e <env> TOols 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:

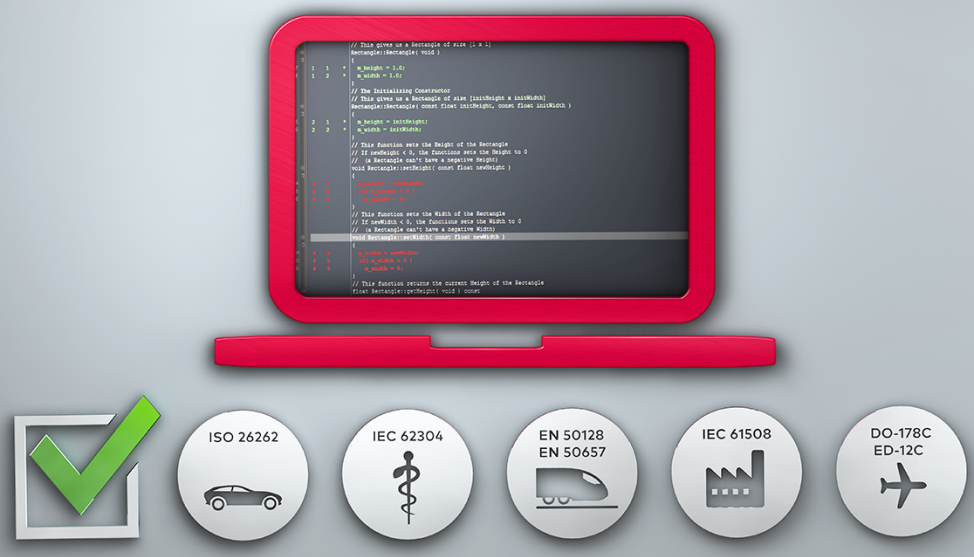
```
clicast -lc -e <env> TOols PRobe_point Xml [<outputfile>]

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:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <probe_points version="1.0">
3    <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"/>
4    <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"/>
5    <file_scope unit="9" placement="0" tu_line="48"/>
6  </probe_points>
```



Using VectorCAST Covered By Analysis

Covered By Analysis (CBA)

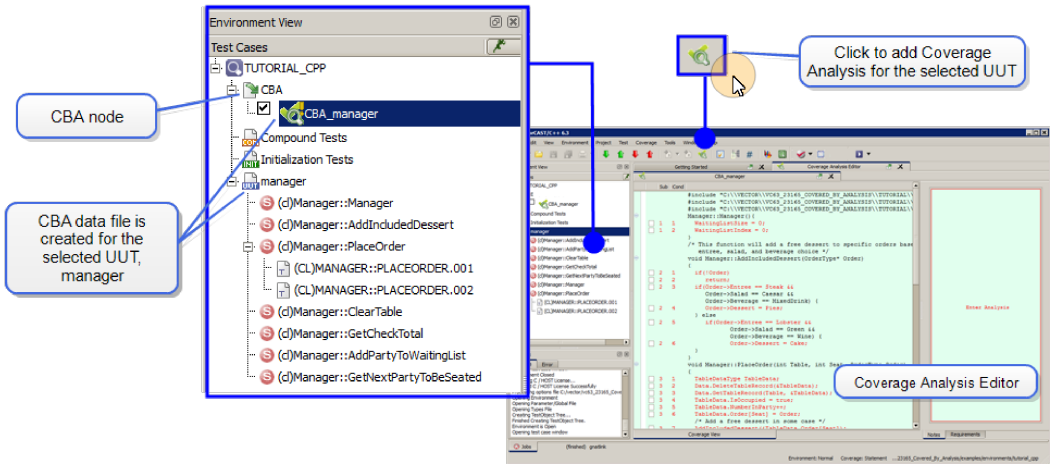
Covered By Analysis (CBA) allows users to augment test coverage metrics with coverage analysis data sets. Small portions of embedded applications are commonly impossible to test, but regulated industries require documented analysis of uncovered code to meet the requirement of 100% structural coverage.

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.

To Add Coverage Analysis

The environment must first be instrumented for coverage. A CBA Analysis result is associated with one UUT or non-stubbed unit. To add Coverage Analysis for a selected unit, select the **CBA** button  on the Toolbar. Clicking the **CBA** button requires a unit to be selected. Alternatively, right-click a UUT in the Test Case Tree and select **Add Coverage Analysis** from the context menu.

The Coverage Analysis Editor opens in the MDI Window and a CBA node displays in the Test Case Tree. A CBA data file is created for the selected UUT and this Analysis result displays beneath the CBA 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 UUT, but you can create multiple Analysis results per UUT.



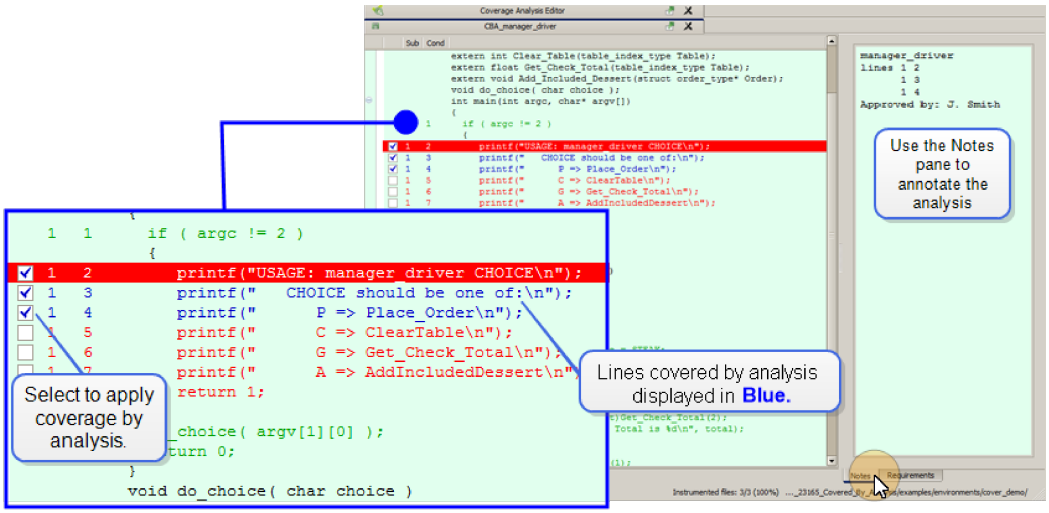
Using the Coverage Analysis Editor

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.



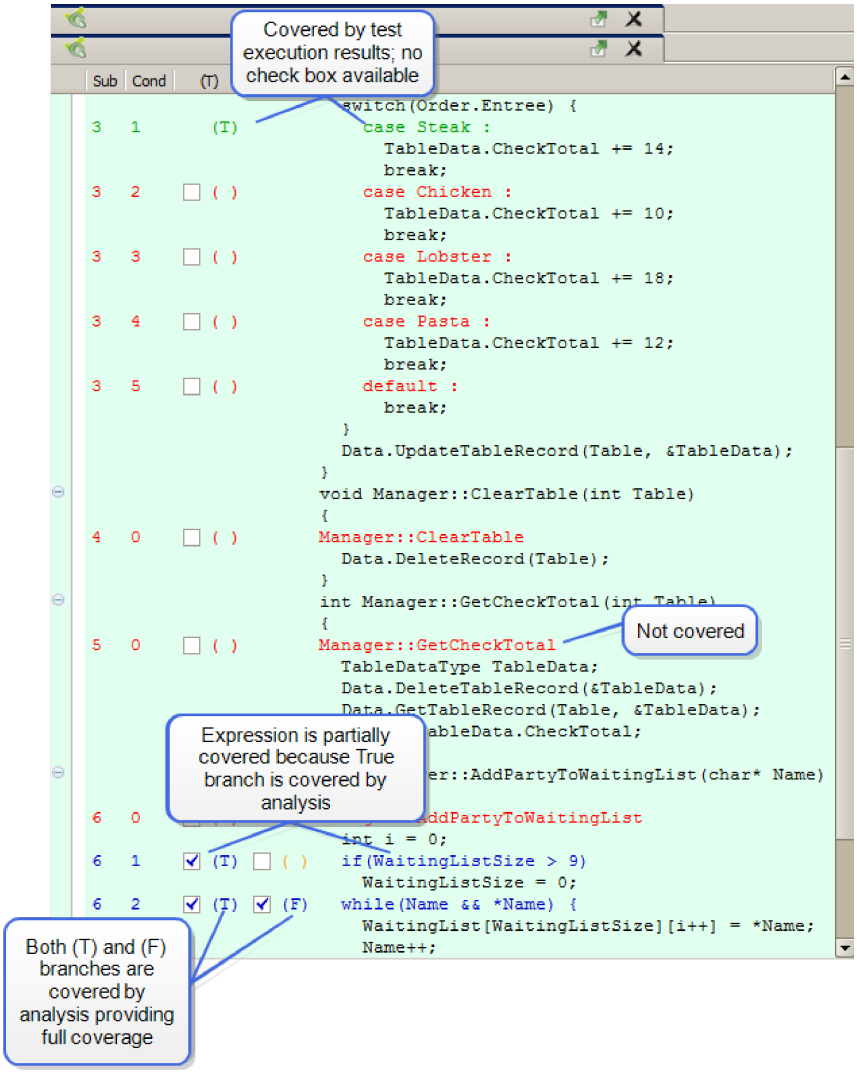
In the Coverage Analysis Editor, a check box does not appear next to a line if it is already covered by a test case execution result. Regular test case results, if they are present in the environment, take precedence over CBA results.

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.

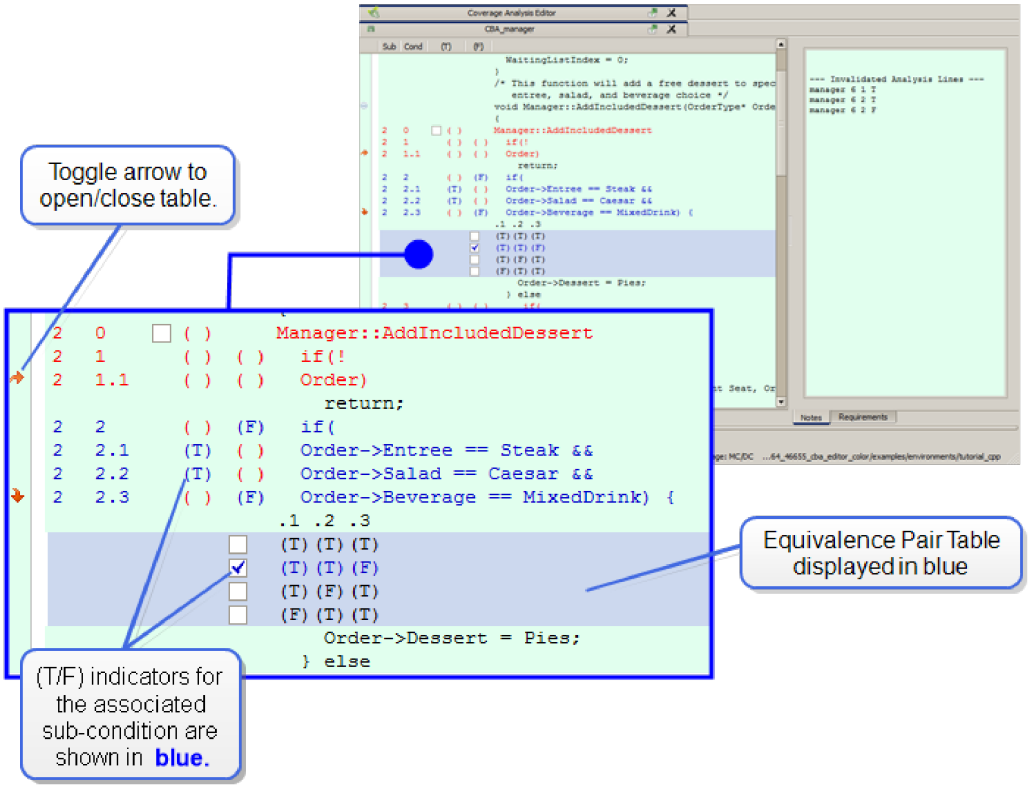


As with Statement coverage, if either or both of the True and False branches is already covered by regular test case execution results, then the check box is not available in the Coverage Analysis Editor, and the expression shows yellow if partially covered, or green if already fully covered.

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.

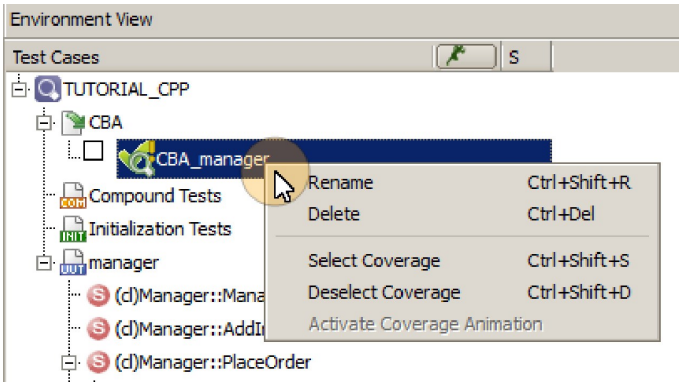
To access the equivalence pair table, click the arrow to the left of the condition. The truth table opens and a check box is displayed for each row. When a checkbox is selected, the associated sub-condition is considered "covered" and displayed in blue.



To Edit an Existing Analysis Result

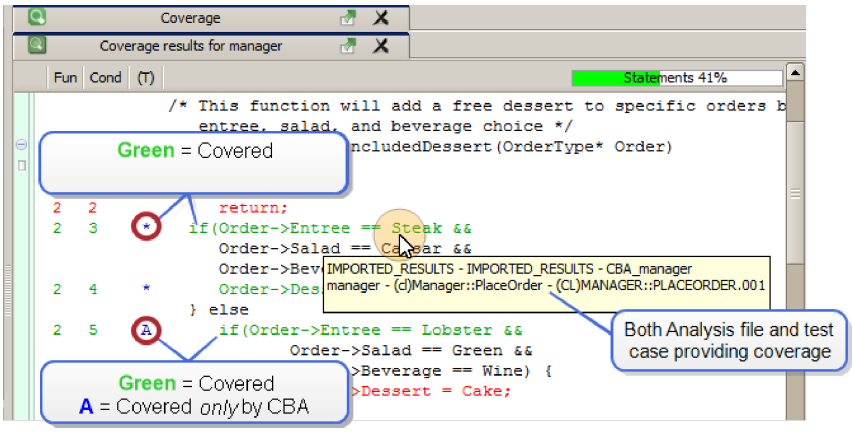
Any existing Analysis results can be edited, regardless of how they were created. This includes Analysis results that are imported via **`.cvr`**, **`.cba`**, and after rebuilding an environment.

Double-click on an existing Analysis result in the Test Case Tree to open the Coverage Analysis Editor. Right-clicking on the Analysis result provides a contextual menu allowing you to Rename, Delete, Select Coverage and Deselect Coverage. Select and Deselect turns On/Off the Analysis result in the Coverage Viewer. Deleting an Analysis result removes its data, notes, and requirements from the environment.



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 Analysis result providing coverage.

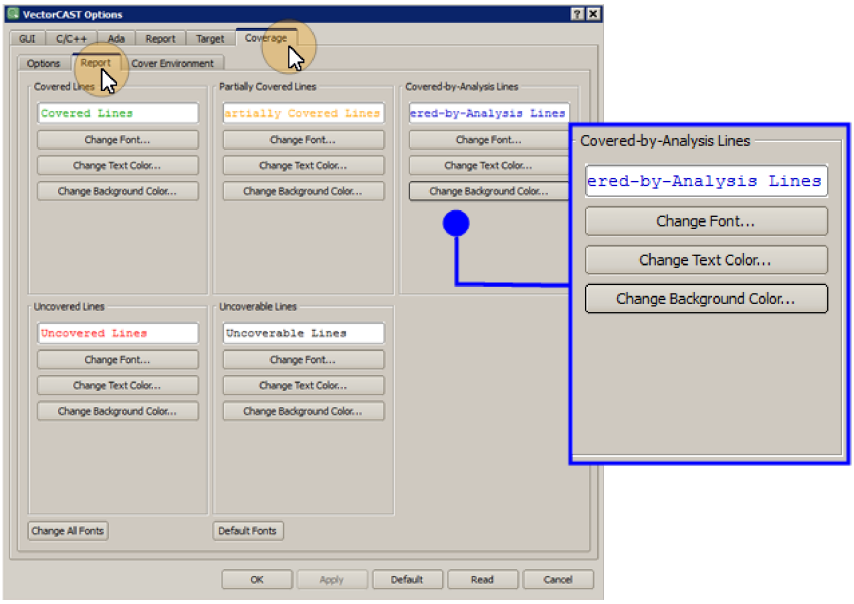


Lines covered only by CBA results are displayed in green (indicating the line is covered) with a blue "A" (indicating the line is covered *only* by CBA).

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.

For Statement coverage, the .cba file syntax is:

```
Unit Name :                Line Expression
manager.cpp :              1-3, 12, 14-34
```

The .cba file above covers lines 1-3, 12 and 14-34 of the manager.cpp source file.

For Branch coverage, the .cba file syntax is:

```
Unit Name :                Line Expression
manager.cpp :              74
manager.cpp :              84-87
```

Note that both T and F are covered when a line is specified. You cannot specify that only T or F be covered by a .cba file.



Note: You cannot use the same .cba file for both Statement and Branch coverage. While the same syntax is used in both, in Statement coverage the line number might be different due to different instrumentation.

MC/DC coverage is not supported in a .cba file.

Note that CBA data can be exported to a .cvr script and can therefore be imported.

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> TOols EXPORT_CBA <scriptfile>

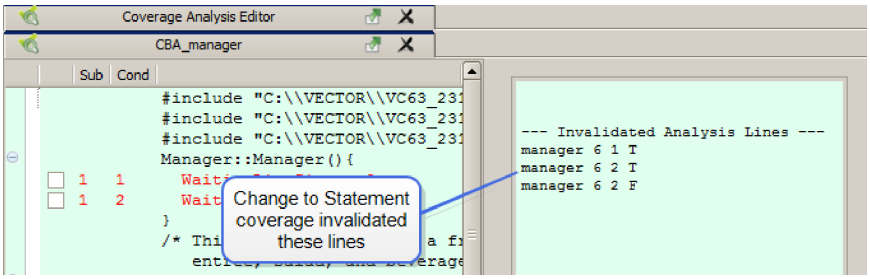
Generate a script containing Coverage By Analysis results.
```

To Remove Analysis Files

To remove CBA data files, right-click on the Analysis result and select **Delete** from the context menu. The CBA coverage data, Notes, and Requirements are removed.

Re-Instrumenting With CBA Data

When you change the coverage type in the environment, the Notes and Requirements for CBA are automatically retained and include information on the data that was invalidated. The data that is invalidated is recorded in the Notes section. 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.

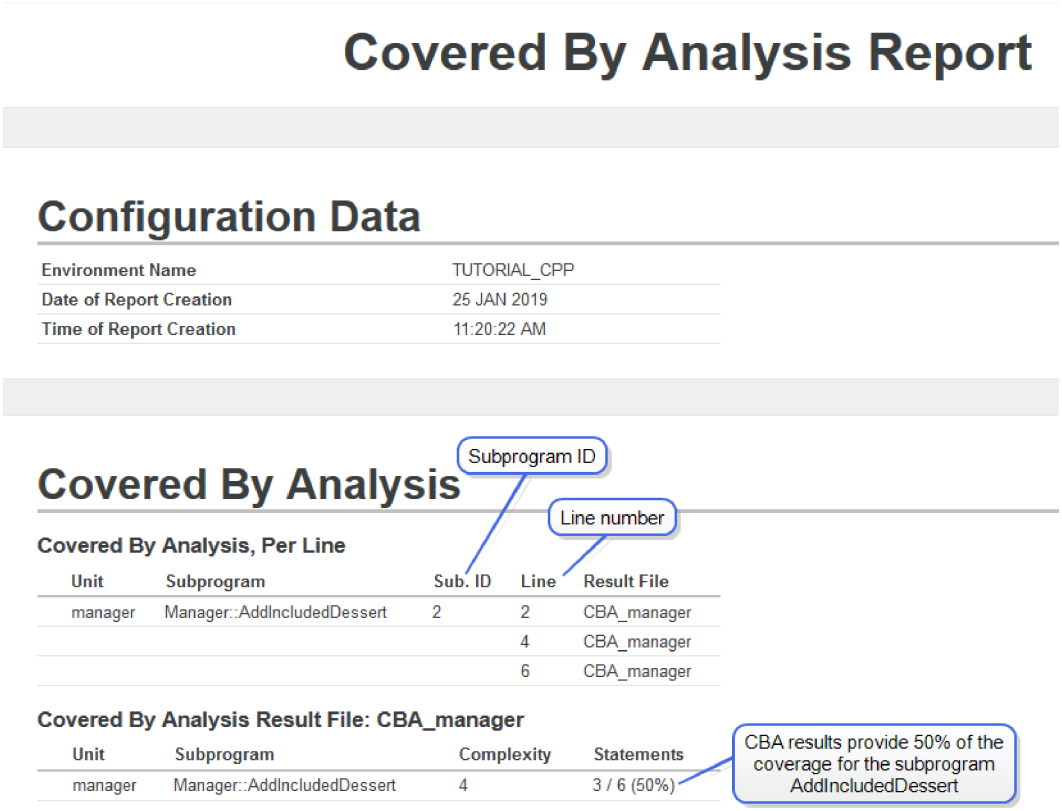


Viewing Analysis Data in Reports

Covered By Analysis Report

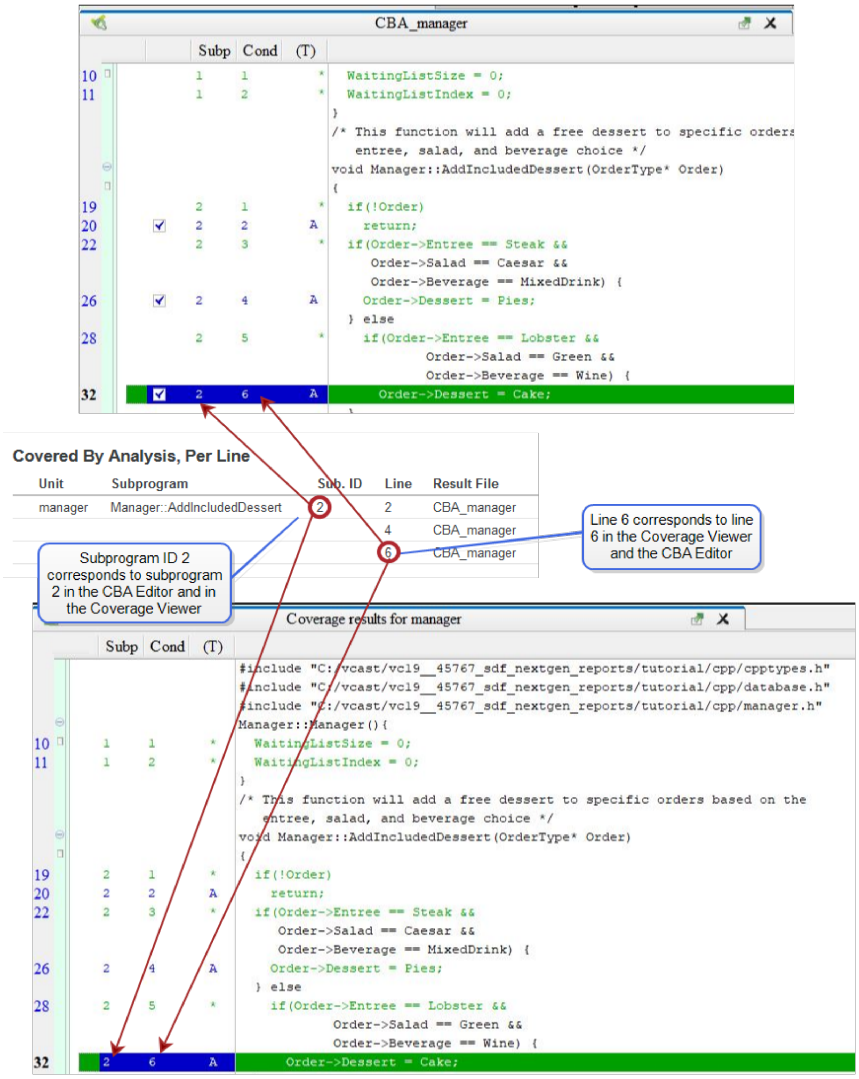
The Covered By Analysis (CBA) Report provides an overview of the Analysis data, including the total number of lines or conditions covered and any associated notes and requirements. The CBA Report includes only CBA results and disregards any regular test case results which normally take precedence over CBA results.

To access the Covered By Analysis Report, select **Test => View => Covered By Analysis Report** from the Menu Bar.



The CBA Report has two sections: The **Covered By Analysis, Per Line** section and the **Covered By Analysis Result File** section(s).

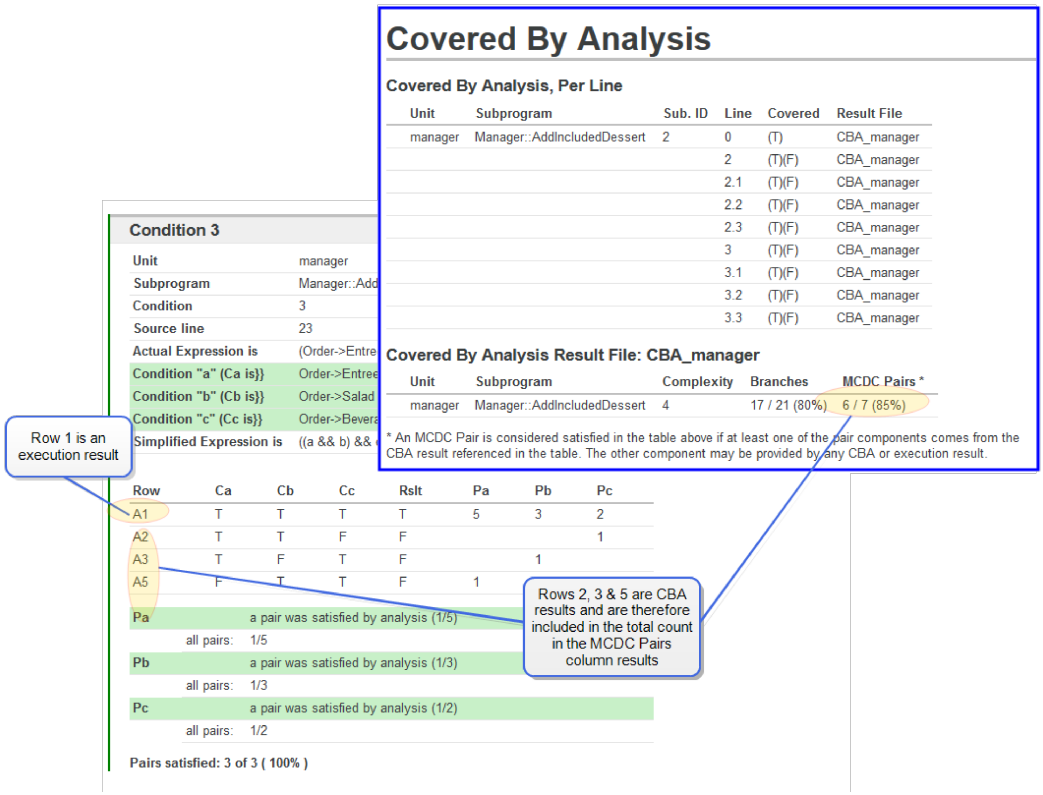
The **Covered By Analysis, Per Line** section lists each covered line in the unit, and identifies which CBA result file covers that line. The Subprogram ID corresponds to the left-hand number in the Coverage Viewer and CBA Editor. The Line number corresponds to the right-hand number in the Coverage Viewer 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,three pairs are covered. Each pair has one component covered by the CBA result file CBA_manager (rows 2, 3 and 5). The other component (row 1) is covered by an execution result.

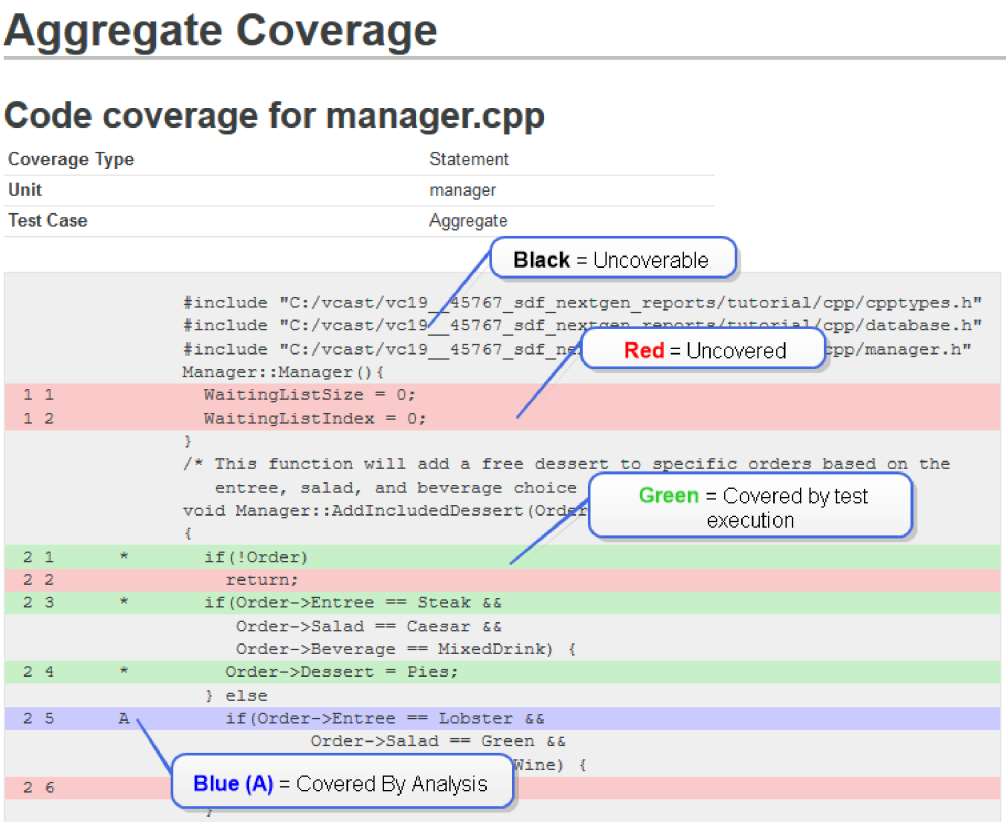



```
clicast -e <env> Reports Custom CBA <outputfile>

Create a CBA report and output to HTML file <env>_covered_by_analysis_
report.html, standard output as text, or to the specified output file.
```

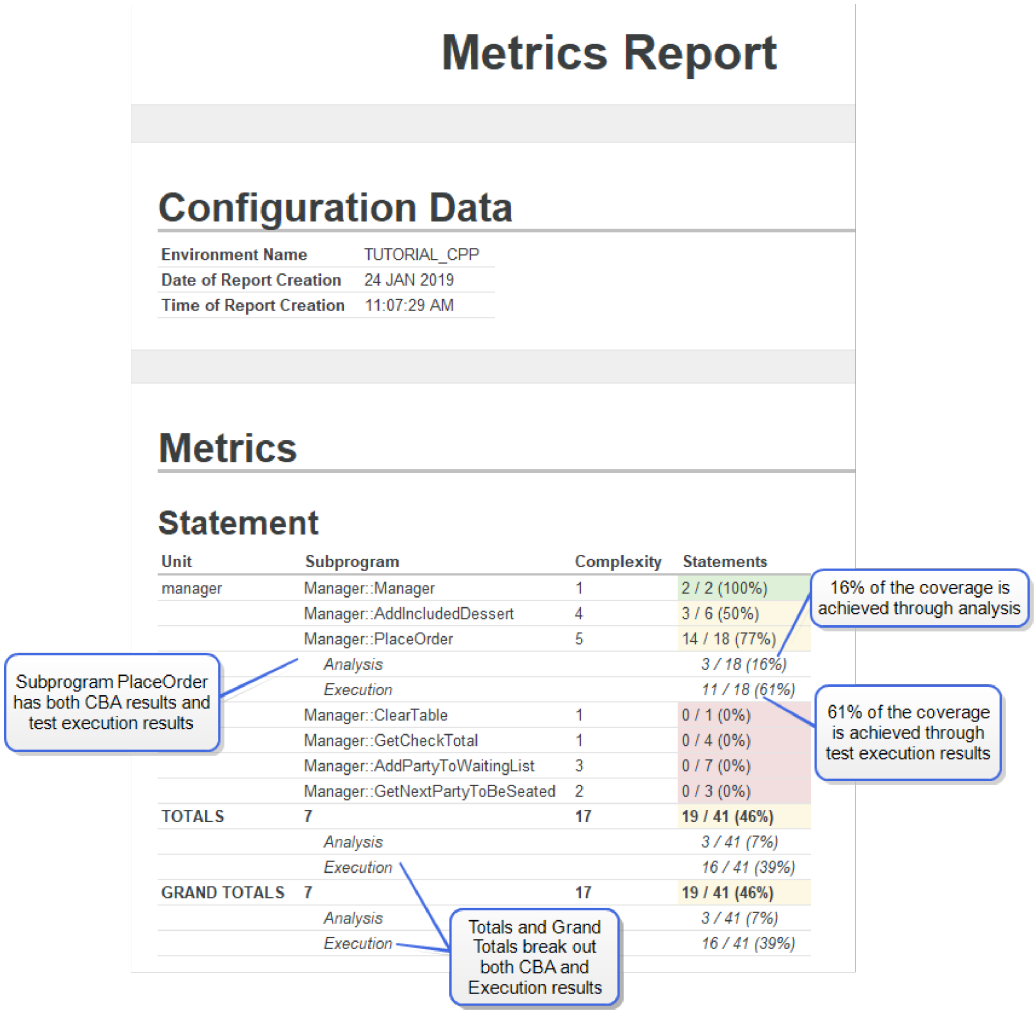
Aggregate Coverage Report

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.



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.

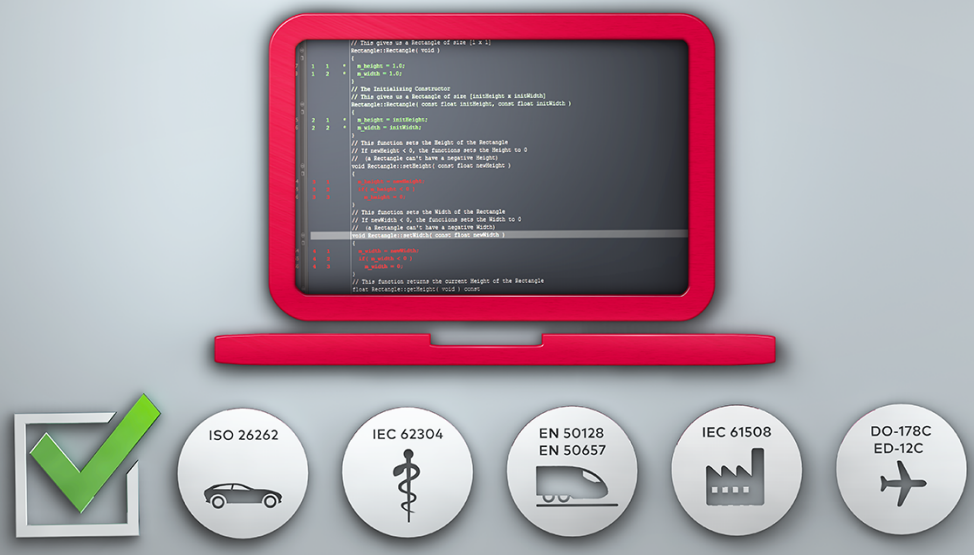
Changes for the Covered By Analysis Report

> 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.



User Code

Understanding User Code

Types of User Code

VectorCAST's User Code capability enables you to write C/C++ language expressions to set or check the value of data objects, or to do some other dynamic test case setup. With user code, you can write code to read data from a file, call initialization routines, or assign and verify data objects based on dynamic criteria.

There are several types of user code: environment user code, test case user code, individual parameter user code, and stub user code.

- Environment User Code is best suited for operations that relate to all test cases; loading data from a file or initializing a database are two examples.
- Stub User Code is added to functions that have been stubbed, allowing you to specify extra logic to be executed when the stub is called.
- Test Case User Code is used to include input value or expected value user code which is not associated with a specific parameter. It applies to simple test cases, not compound test cases. It is accessed by clicking the Testcase User Code tab in the Test Case Editor window.
- Parameter User Code is used to set a parameter value based on a dynamic expression, or to verify a parameter value against some expression.

See "ENVIRO.ADDITONAL_UNIT_BODIES" on page 670, "ENVIRO.UNIT_APPENDIX_USER_CODE" on page 665 for information on these types of Environment User Code that are parsed but not executed.

See also "ENVIRO.DRIVER_PREFIX_USER_CODE" on page665 for information on a type of Environment User Code that is not parsed or executed.

See also "ENVIRO.STUB_ENTRY_USER_CODE" on page 669.

Order of Execution

User code is executed in the following order during test case execution:

- Environment User Code – Harness Init
- Environment User Code – Test Case Init
- Test Case Input User Code
- Constructor User Code
- Parameter Input User Code
- Timer Start
- [Invocation of a Unit Under Test (Test Harness calls UUT)
    - >> [Invocation of a stubbed subprogram]
    - >> Timer Stop
    - >> Configure Stubs User Code (Beginning of Stub)
    - >> Environment User Code (tags only) – Stub Entry User Code

- >> Environment User Code – Stub Processing
- >> Parameter Input User Code for Stub
- >> Environment User Code (tags only) – Stub Exit User Code
- >> Configure Stubs User Code (End of Stub)
- >> Timer Start
- >> [Return from stubbed subprogram]

- > [Return from UUT to Test Harness]
- > Timer Stop
- > Environment User Code – Test Case Term
- > Parameter Expected User Code
- > Test Case Expected User Code
- > Environment User Code – Harness Term

Editing User Code

Each type of user code is entered by accessing its user code dialog:

- > Environment User Code is accessed by choosing **Environment => User Code => Edit**.
- > Stub User Code is accessed by choosing **Environment => Configure Stubs => Edit**.
- > Test Case User Code is accessed in the Testcase User Code tab of the Test Case Editor.
- > Parameter User Code is accessed by right-clicking on a parameter in the Parameter Tree and selecting “Properties…” or double-clicking the parameter and selecting the User Code tab.

User Code Tags

User code can consist of any legal C/C++ source code. Beyond that, there are some special tags that can be used to refer to VectorCAST-generated variable names. When the test harness is generated, VectorCAST creates a global variable for each parameter of each subprogram in each unit that is either under test or stubbed. These variable names may change each time a test harness is rebuilt. To accommodate this, you refer to these variables using a special tag syntax.

Some examples of the tag syntax follow (see the next section for information on obtaining the correct tag syntax automatically):

- > Parameter objects are specified using the notation:

```
<<unitname.subprogram.parameter>>
```

- > Structures and arrays are specified using the notation:

```
<<unitname.subprogram.array_name>>.element
```

- > Pointer to a structure is specified using the notation:

```
<<unitname.subprogram.parameter>>[index].element
```

- Array objects are specified using the notation:

```
<<unitname.subprogram.parameter>>.Array[index].element
```

- If **element** is an array itself, keep going, as in:

```
<<unitname.subprogram.parameter>>.Array[index].element
```

- When the unit is a stubbed-by-prototype unit, the **unitname** is

```
uut_prototype_stubs.
```

- Global objects are specified using the notation:

```
<<unitname.<<GLOBAL>>.parameter>> ...
```

- Class objects are specified using the notation:

```
<<namespace::class instance>>
```

In addition to assigning values to parameters and global objects, you can also compare values against an expected expression. To have the result of a user code comparison show up in the test results, you must enclose the comparison in double curly braces, **{{ }}**. For example, **{{ <<unitname.subprogram.parameter>> == value }}** would result in a comparison being performed between the value of the parameter and **value**. Of course, a dynamic expression of any sort can be substituted for **value**.

When specifying Expected User Code, the code between the double braces (**{{...}}**) must evaluate to a boolean expression. The boolean expression will be evaluated as the test runs, and the value will be reported in the test results listing. The result of the evaluation of the boolean expression will be included in the pass/fail analysis of the test case. A boolean expression can be combined with other C-code to accomplish a more complex comparison.

Features Common to All User Code

In addition to a common syntax, all user code edit areas support an external editor, importing from a file, and automatic insertion of User Code tags.

Insert User Code Tag

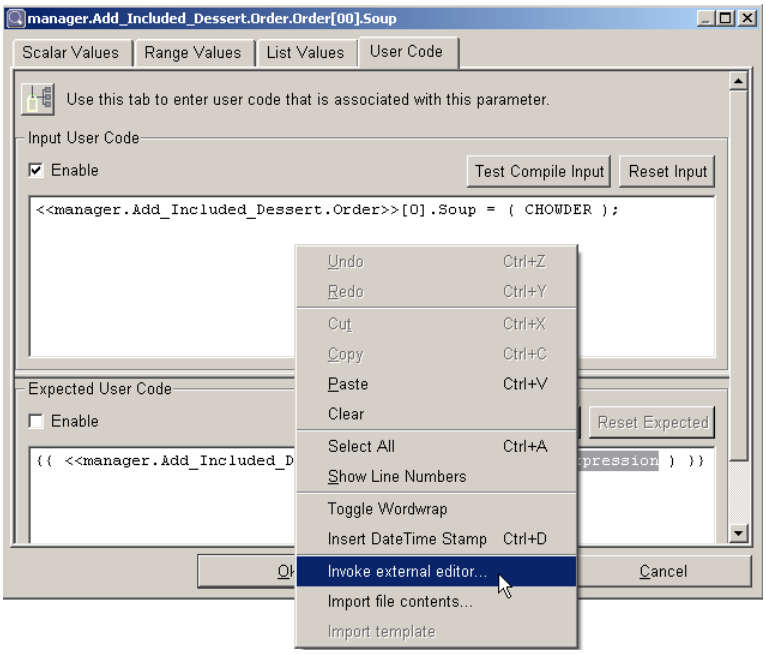
To make it easier to enter user code tags for objects, you can click the User Code Tags button  in any of the user code dialogs or on the toolbar. Alternatively, select **View => Dock Control => User Code Tags** from the Menu Bar. A new window opens in non-docked mode, which displays a hierarchal tree of objects in the environment. Using this tree, navigate to the object whose tag you want to insert.

A Find Banner is available to assist in searching and filtering the text. The Find Banner appears at the

bottom of the window and is off by default. Access the Find Banner using the **Ctrl + F** keyboard shortut. For more information on using this feature, see "To Search for Text Using the Find Banner" on page 89.

Using an External Editor

If you chose an external text editor (notepad or emacs, for example) in the **Tools => Options** dialog, GUI tab, then you can invoke that editor by right-clicking in any of the user code text areas, and selecting **Invoke external editor** from the pop-up menu. When you are finished editing, save and exit the editor and the text is placed in the user code section.

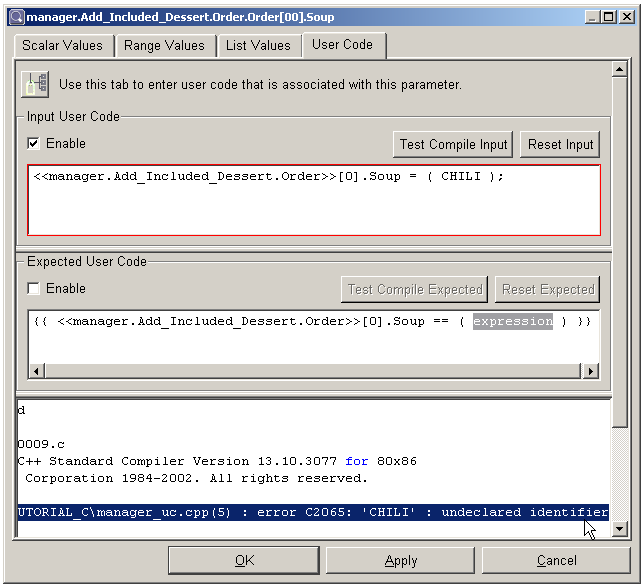


Importing Contents of a File

In addition to typing in user code, you can add the contents of a file. To import the contents of a file to any of the user code dialogs, right-click in the text area, and select **Import file contents** from the pop-up menu. An Open File dialog appears, allowing you to choose a file. Once you've chosen a file, click **Open**, and the text from the file is added to the user code section.

Test Compile Button

Before committing your user code to the test harness, a **Test Compile** button enables you to determine if there are any compilation errors in the code. The various user code dialogs have the Test Compile button in different locations, and some have a separate one for input user code and expected user code, but they all behave in essentially the same way. When you click a Test Compile user code button, the user code is compiled, and VectorCAST informs you if it was successful or there were errors, and displays the errors. It may be necessary to scroll to the bottom of the dialog to see the errors displayed.



Environment User Code

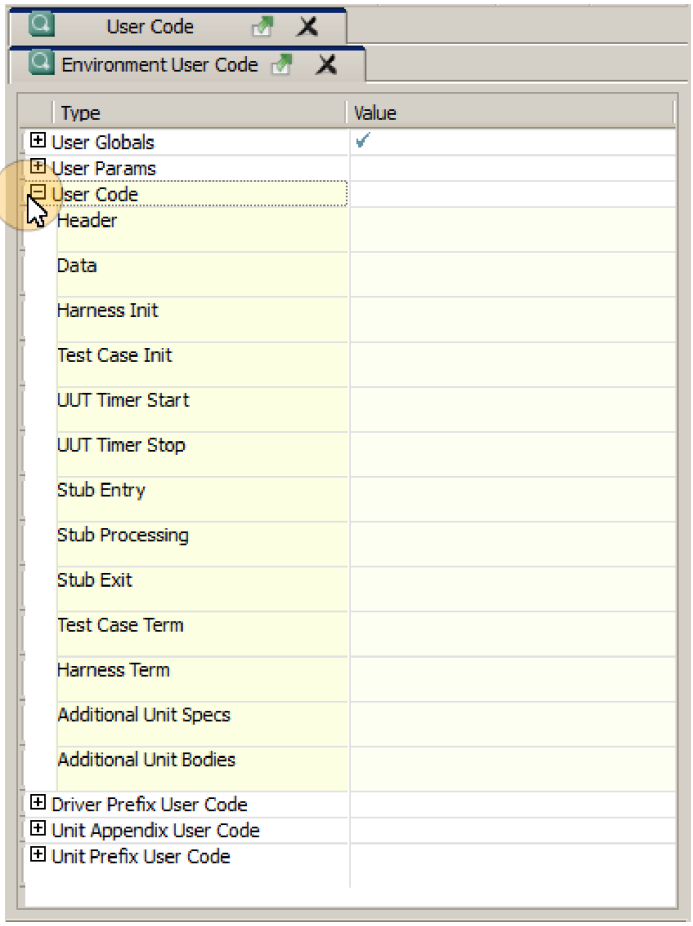
Types of Environment User Code

VectorCAST's User Code capability enables you to write source code to handle some of the harness tasks that are not easily accomplished with static data. With User Code, you can write code to read data from a file, call initialization routines, or assign and verify data objects based on dynamic criteria. *Environment User Code* is best suited for operations that relate to the harness as a whole; loading data from a file or initializing a database unit are two examples.

Choose **Environment => User Code => Edit** to enter or modify the environment user code.

The Environment User Code dialog enables access to User Globals, User Parameters, Environment User Code, Driver Prefix User Code, Unit Appendix User Code, and Unit Prefix User Code.

Expand the node for "User Code" to see the input boxes for each type of Environment User Code.



Header – Source code that normally appears at the top of a source file. Usually consists of #include or #define statements.

Data – Source code that normally occurs near the top of a source file. This could consist of type and data declarations, as well as any support subprograms that might be needed.

Harness Init – Source code in this area gets executed once per harness execution, immediately after elaboration has completed. This could include calling initialization routines.

Test Case Init – Source code in this area gets executed immediately after the test case's data is loaded, but before the UUT is called by the harness. This usually includes initializing data for a specific test case. If your test case sets global values, the Test Case Init code has the opportunity to write over those values.

UUT Timer Start – Source code in this area gets executed just prior to entering the call to the UUT. Used to start timer.

UUT Timer Stop – Source code in this area gets executed just after returning from the call to the UUT. Used to stop timer.

Stub Entry – Source code in this area gets executed upon entry to a stubbed subprogram, after Configure Stubs Beginning User Code.

Stub Processing – Source code in this area is executed every time any stub is called.

Stub Exit – Source code in this area gets executed upon exit from a stubbed subprogram, before Configure Stubs Ending User Code

Test Case Term – Source code in this area gets executed immediately after the UUT returns control to the harness. This usually includes examining data affected by specific test cases.

Harness Term – Source code in this area gets executed once per harness execution, immediately before exiting. This could include saving data to a file or other cleanup processing.

Additional Unit Specs – Specification portion of an additional code unit.

Additional Unit Bodies – Implementation portion of additional code unit, to be added to test harness.

To enter user code for a *particular* stubbed subprogram, see "To Enter Configure Stub User Code" on page 623.

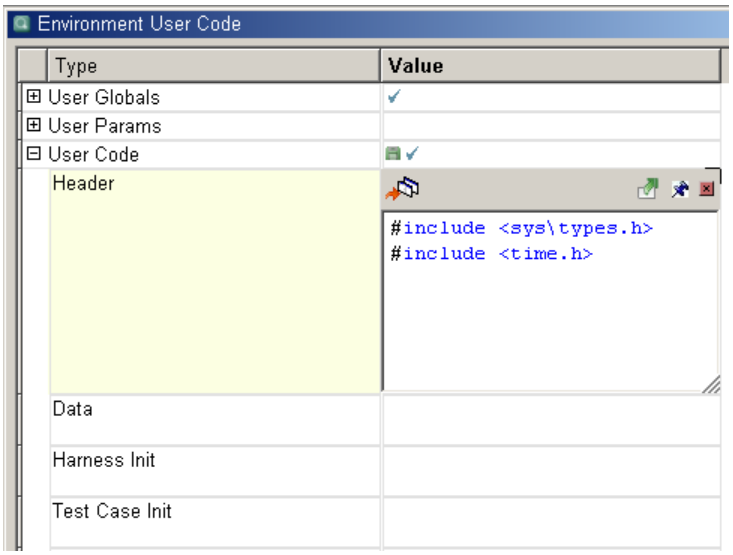
See "Environment Script Language" on page 661 for more information on the user code tags in the environment scripting language.

To Edit Environment User Code

To enter code for environment user code, follow these steps:

1. **Environment => User Code => Edit**.
2. Expand the node **"User Code"**
3. Choose which type of environment user code you need
4. Double-click in the **"Value"** column for that type

An edit box opens in which you can type, copy/paste, and test compile  source code.



To test compile this code, click the Test Compile button  . Test compiling in one cell of Environment User Code incorporates the source code in all other Environment User Code cells.

Therefore if you have an error in any cell, the Test Compile Errors report is displayed.

The cell sports several icons and buttons:

 – contains code (not empty)

 – needs saving

 – needs to be compiled and linked

 – currently has an error

 – test compile the unit with the appendix user code

 – Save, or Save, Compile and Link (from the drop-down menu) the unit, but does not affect the Parameter Tree of a test case until environment is rebuilt

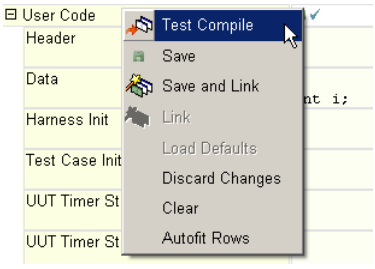
 – pop cell out into its own window for easier editing

 – pin, or preserve size

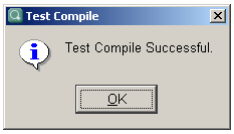
 – close the edit box for the cell

To Test Compile Environment User Code

There are several ways to test compile environment user code. In each individual cell, click the **Test Compile** button  to compile the code before it is saved to the test harness. Or, right-click a node and choose Test Compile. Either way, *all* environment user code is test compiled, not just the individual cell.



If successful, a message appears informing you, as shown in the following figure:

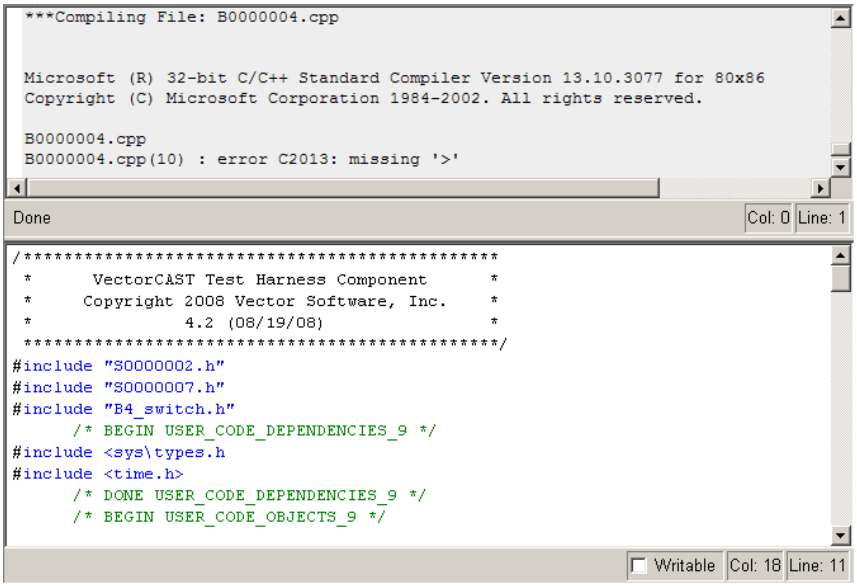


and any error state icons  are removed.

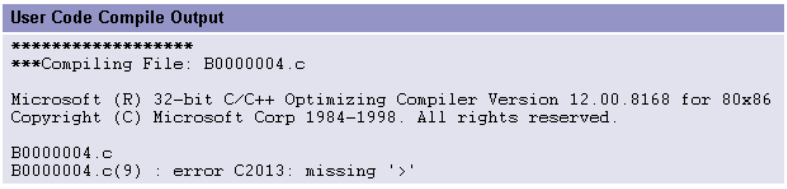
If there are compile errors, a dialog informs you and a window explains the errors. For example, suppose the following user code is entered in the Header section. Note the missing angle bracket at the end of the first line.

```
#include <sys\types.h
#include <time.h>
```

When the test compile button is clicked, the MDI window shows the following:



The top pane shows the compiler error and the line in which the error occurs.



The lower pane shows the user code file with the temporary user code inserted. Use Ctrl+G (Goto Line) to go to the line with the error.

```
#include "S0000002.h"
```

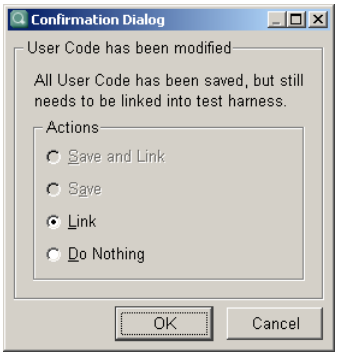
```
#include "S0000007.h"
#include "B4_switch.h"
      /* BEGIN USER_CODE_DEPENDENCIES_9 */
#include <sys\types.h
#include <time.h>
      /* DONE USER_CODE_DEPENDENCIES_9 */
      /* BEGIN USER_CODE_OBJECTS_9 */

      /* DONE USER_CODE_OBJECTS_9 */
```

To close the Test Compile Errors window, use the ✕ in the upper right corner or **File => Close**.

To Save Environment User Code

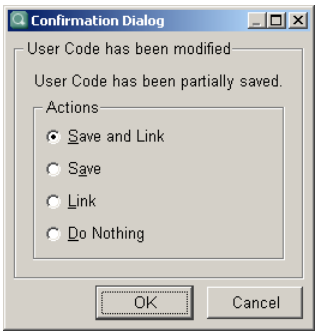
To save all sections of Environment User Code, click the **Save** button on the Toolbar  , right-click a node and choose **Save**, or choose **File => Save**. The following dialog appears.



If you are ready to compile the user code and link it into the test harness, leave the radio button on **Link** and click **OK**.

If you plan to rebuild the environment anyway, put the radio button on **Do Nothing**, and click **OK** or **Cancel**.

If you attempt to close the Environment User Code window (using the ✕ window control) and there are any unsaved sections, the same dialog appears, but this time indicating that a Save needs to be performed.



Choosing the **Save and Link** radio button saves, compiles and links the user code into the test harness before closing the window. Choosing the **Save** radio button saves your changes before closing the window. Choose **Environment => User Code => Compile and Link later**. Choosing the **Link** radio button when some cells have been modified discards the changes and compiles and links the user code. Clicking the **Do Nothing** radio button discards your changes and closes the Environment User Code window.

User Globals

User Globals is a collection of global objects for use by the User Code functionality and for manipulation of objects of void* type. You can also use this area to define temporary data objects. These objects can also be accessed in the Create New Environment wizard, User Code page, and are visible in the Parameter Tree of a test case.

Choose **Environment => User Code => Edit** to enter or modify the user globals. If you want to see the change in the Parameter Tree of a test case, you will need to rebuild the environment. Otherwise, compiling and linking the change into the test harness may suffice.

The Environment User Code dialog enables access to User Globals, User Parameters, Environment User Code, Driver Prefix User Code, Unit Appendix User Code, and Unit Prefix User Code.

Expand the node for “User Globals” to see the edit box of default user globals. The User Globals section provides a mechanism for user-defined types and objects to be included into the test harness. By default, the file has five integer objects, five floating point objects, five string objects, and an array of 200 integer elements. All data objects that are defined in the User Globals file when the environment is built can be manipulated as test data when building a test case.

You can add to these objects at the time you create the environment using the User Code page in the Create New Environment wizard.

The User Globals file is accessible in the Create New Environment dialog, the Environment User Code dialog, and the USER_GLOBALS_VCAST section of a test case’s Parameter Tree. The following list explains how to move between the three:

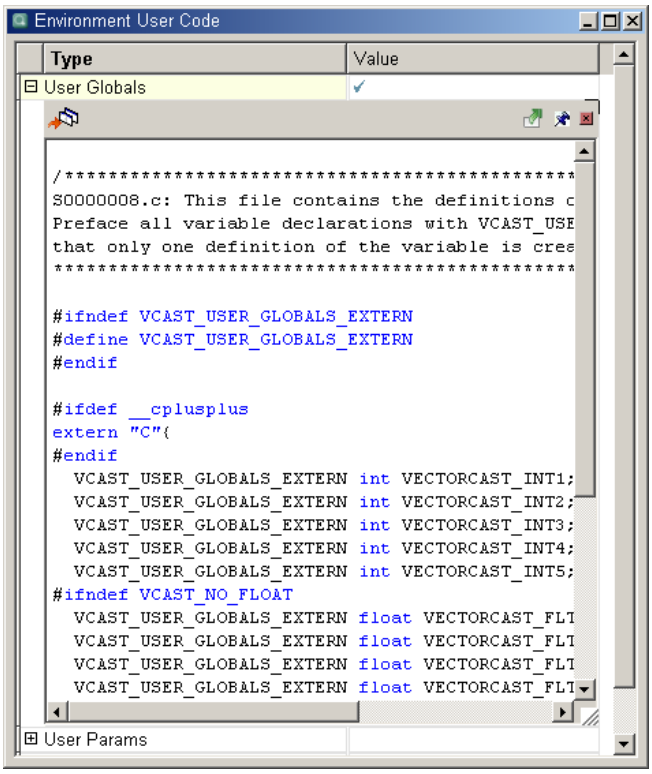
- Any User Globals added in the Create New Environment wizard, User Code page, are seen in the Environment User Code dialog, User Globals section and in the Parameter Tree of a test case.
- Adding any User Globals in the Environment User Code dialog, User Globals section, requires you to rebuild the environment before they are visible in the Parameter Tree of a test case. To see them in the Update Environment wizard, choose **Environment => Update Environment**.
- The default version of the User Globals is delivered in source code format in the VectorCAST Installation Directory, subdirectory DATA, filename GLOBALS.C. If you modify the default file, the changes are reflected in the User Code page in the Create New Environment wizard for each new environment.

To Edit User Globals

To enter code for environment user code, follow these steps:

1. **Environment => User Code => Edit**.
2. Expand the node “User Globals”
3. Double-click in the yellow cell to edit

An edit box opens in which you can type, copy/paste, and test compile source code.



Be sure to preface all variable declarations with VCAST_USER_GLOBALS_EXTERN to ensure that only one definition of the variable is created in the test harness.

To test compile this code, click the Test Compile button . Test compiling in one cell of Environment User Code incorporates the source code in all other Environment User Code cells. Therefore if you have an error in any cell, the Test Compile Errors report is displayed.

The cell sports several icons and buttons:

– contains code (not empty)

– needs saving

– needs to be compiled and linked

– currently has an error

– test compile the unit with the appendix user code

 – Save, or Save, Compile and Link (from the drop-down menu) the unit, but does not affect the Parameter Tree of a test case until environment is rebuilt

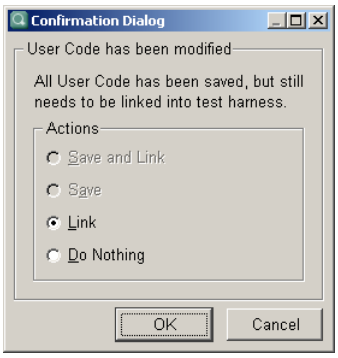
 – pop cell out into its own window for easier editing

 – pin, or preserve size

 – close the edit box for the cell

To Save User Globals

To save the User Globals, click the **Save** button on the toolbar  , right-click the User Globals node and choose **Save**, or choose **File => Save**. The following dialog appears.

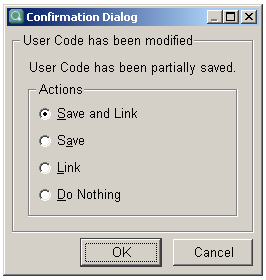


To see the changes to the User Globals in the Parameter Tree of a test case, you'll need to rebuild the environment. Put the radio button on **Do Nothing**, and click **OK**. Choose **Environment => Rebuild later**. If you have other Environment User Code changes, you probably want to use Link.

If you only need access to the new User Globals (without seeing them in the Parameter Tree), leave the radio button on **Link** and click **OK**.

To cancel the save operation without saving, click the **Cancel** button.

If you attempt to close the Environment User Code window (using the window control) and there are any unsaved sections, the same dialog appears, but this time indicating that a Save needs to be performed.



Choosing the **Save and Link** radio button saves, compiles, and links the all environment user code into the test harness before closing the window. Choosing the **Save** radio button only saves your changes before closing the window; it does not compile and link. Choose **Environment => User Code => Compile and Link later**. Choosing the **Link** radio button when some cells have been modified discards their changes and compiles and links the user code. Clicking the **Do Nothing** radio button discards your changes and closes the Environment User Code window.

To Recompile Environment User Code

The **Environment => User Code => Compile** and Link command causes a compilation of the user code file to be performed. This command enables you to compile and re-link the modified USER_CODE_VCAST unit with the VectorCAST-created executable test harness. Use this command when you are ready to compile the environment user code after you have edited the environment user code, but you chose “Save Only” rather than “Save and Link” at the time you edited the user code.

```
clicast -e <env> ENvironment User COmpile

Compile and link Environment User Code into test harness.
```

Unit Appendix User Code

Unit Appendix User Code enables you to add any source code you want to the end of a UUT unit. The user code is parsed, preprocessed during header expansion, and compiled into the unit. It is literally #included to the source code for the specified UUT. This type of user code is particularly useful if *unit* contains abstract classes. For example, you can **`#include`** a concrete subclass of an abstract class defined in *unit*.

The syntax is any valid C or C++ source code.

```
clicast -e <env> -u <unit> ENvironment Unit Appendix <pathed filename>

Add the contents of the file given by <pathed filename> to the Unit
Appendix User Code for <unit>.
```

To enter code for Unit Appendix User Code, follow these steps:

1. **Environment => User Code => Edit**.
2. Expand the node “Unit Appendix User Code”
3. Expand the node for the unit to which you want to add the user code
4. Double-click in the yellow cell to edit

An edit box opens in which you can type, copy/paste, test compile, and save and link the unit.

The cell sports several icons and buttons:

 – contains code (not empty)

 – needs saving

 – needs to be compiled and linked

 – currently has an error

 – test compile the unit with the appendix user code

 – Save, or Save, Compile and Link (from the drop-down menu) the unit, but does not affect the Parameter Tree of a test case until environment is rebuilt

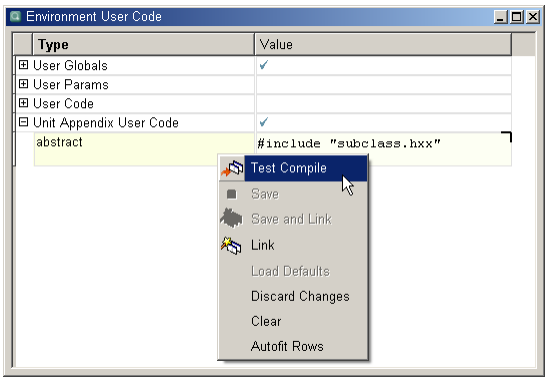
 – pop cell out into its own window for easier editing

 – pin, or preserve size

 – close the edit box for the cell

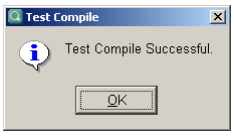
To Test Compile Unit Appendix User Code

There are several ways to test compile unit appendix user code. In each individual cell, click the **Test Compile** button  to compile the code in that unit before it is saved to the test harness. Or, right-click a node and choose Test Compile.



You can also right-click a node higher in the hierarchy, such as “Unit Appendix User Code” level. In this case, choosing Test Compile performs a test compilation on all of the cells in the scope of the right-click. The light-yellow shading of the cells indicates the current scope of a right-click.

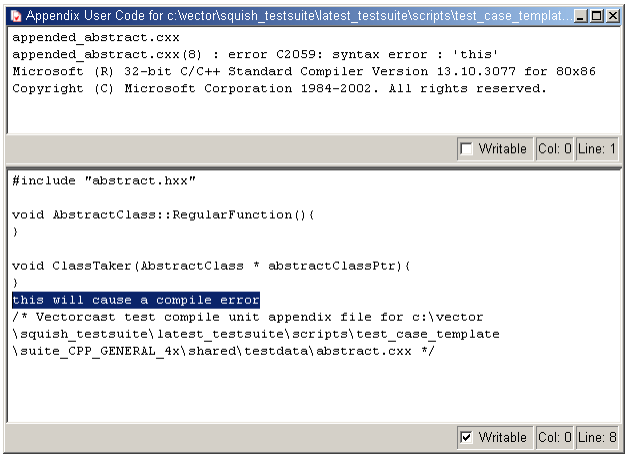
If successful, a message appears informing you, as shown in the following figure:



and any error state icons

 are removed.

If there are errors in your user code, clicking the Test Compile button opens a window titled “Appendix User Code for <unit>.” The top pane shows the compile error; the lower pane shows the stub file. In the example below, the code “this will cause a compile error” is present. By using Ctrl+G in the lower pane you can easily go to the line number where the error occurred. To correct the error, close the Appendix User Code window by clicking the X in the upper right corner, edit the user code, and click the Test Compile button again until it compiles successfully.



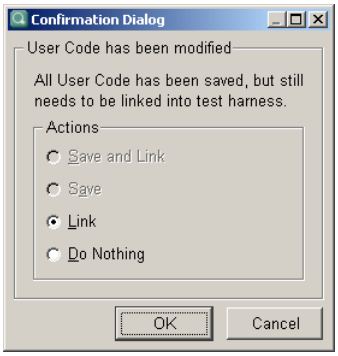
To Save Unit Appendix User Code

To save Unit Appendix User Code, click the **Save button on the Toolbar**  , right-click the Unit Appendix User Code node and choose **Save**, click the small Save icon on the cell’s toolbar  , or choose **File => Save**.

Most likely, you’ll want to rebuild the environment after adding or modifying Unit Appendix User Code, so that you can see the changes in the Parameter Tree for a test case. However, if you just want to compile and link the user code into the test harness, it is functional. You just won’t be able to see the changes in the Parameter Tree.

Depending on whether you plan to rebuild the environment now or later, you have two courses of action:

- If you plan to rebuild now, choose Environment => Rebuild Environment even though you have not linked the Appendix User Code. You see the following dialog.



Click the radio button next to **Do Nothing** and click OK. The environment begins rebuilding which will take case of the link that was needed before.

- If you plan to rebuild later, then click the Link button on the cell's toolbar , close the User Code window, or the Unit Appendix User Code node and choose **Link**. The Unit Appendix User Code is parsed, preprocessed, and compiled and linked into the test harness. If you see a compile error, you should rebuild the environment.
  Although you do not see the changes to the Unit Appendix User Code in the Parameter Tree of a test case, it is part of the test harness and therefore is functional.

To Recompile Unit Appendix User Code

```
clicast -e <env> -u <unit> ENvironment Compile Unit_appendix

Compile and link the Unit Appendix User Code in <unit>. Saved to
ENVIRO.AUX_UC, in the environment directory.
```

To Remove Unit Appendix User Code

xxx clear in the Editor, or Environment => User Code => Clear

Example of Unit Appendix User Code

The following example demonstrates how an abstract class can be made testable by #including a concrete subclass in Unit Appendix User Code. Suppose your source code looked like the following:

abstract.cxx:

```
#include "abstract.hxx"
void AbstractClass::RegularFunction(){ }
void ClassTaker(AbstractClass * abstractClassPtr){ }
```

abstract.hxx:

```
#ifndef _ABSTRACT
#define _ABSTRACT
```

```
class AbstractClass{
 public:
  AbstractClass();
  virtual int VirtualFunction() = 0;
  void RegularFunction();
};

#endif
```

subclass.hxx:

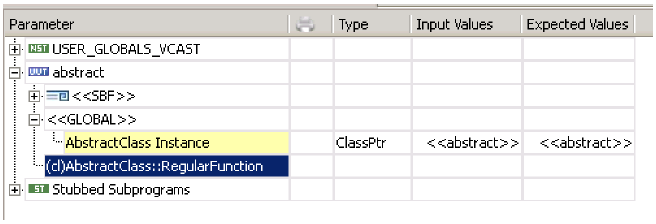
```
#ifndef _SUBCLASS
#define _SUBCLASS

#include "abstract.hxx"

class Subclass: public AbstractClass{
 public:
  Subclass();
  virtual int VirtualFunction(){return 3;}
};

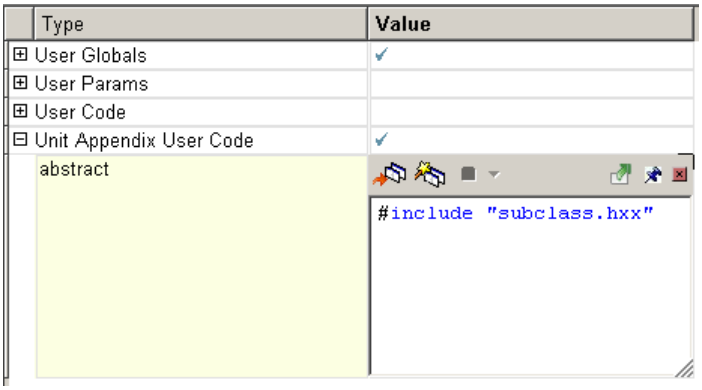
#endif
```

Before you **`#include`** the concrete subclass (in subclass.hxx), the Parameter Tree shows:

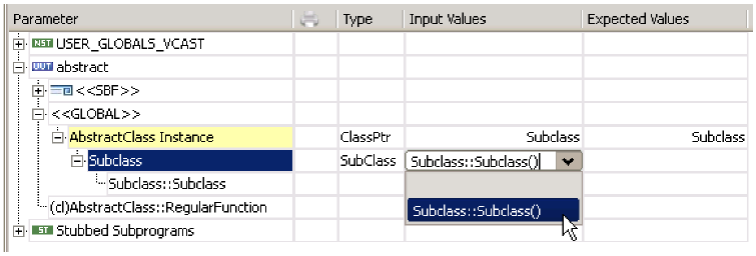


Enter the following code in the Unit Appendix User Code for the unit **`abstract`**, and then rebuild the environment:

```
#include "subclass.hxx"
```

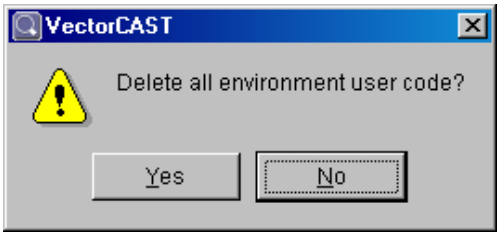


Once the environment is rebuilt, the Parameter Tree shows:



To Remove Environment User Code

The **Environment => User Code => Clear** command will clear the environment user code from the test harness and the Environment User Code dialog. Once selected, you will see the following dialog box confirming that you wish to delete the environment user code.




```
clicast -e <env> ENvironment User CLear yes

Remove Environment User Code from test harness.
```

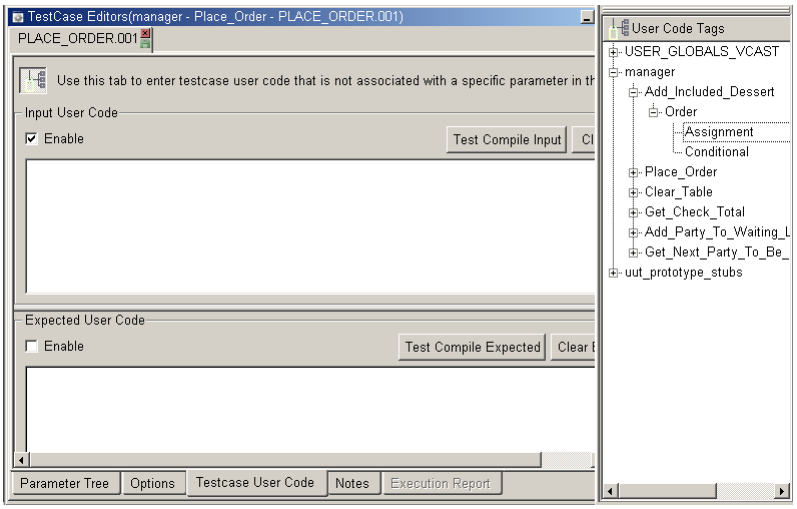
Test Case User Code

Test Case User Code is used to include input value or expected value user code which is not associated with a specific parameter. It applies to simple test cases, not compound test cases.

To Enter Test Case User Code

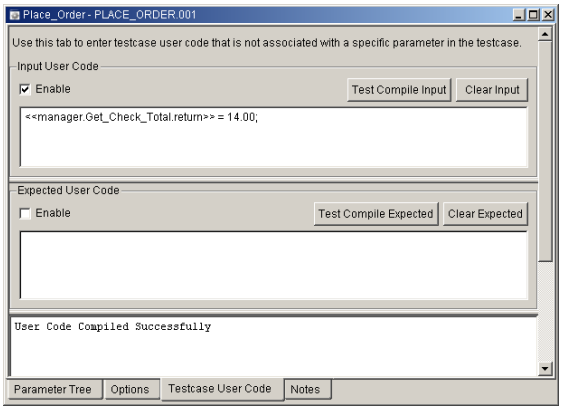
To enter input or expected test case user code, first insert or open a test case. Click the **Testcase User Code** tab along the bottom of the MDI window. Click the **Enable checkbox**, then you can enter code into the edit boxes.

Use the User Code Tags window to easily insert the symbol for a parameter, using the correct notation. To use this feature in either the Input or Expected edit box, click the User Code Tags button  to open the window.



To Test Compile Testcase User Code

After entering user code in this way, clicking the **Test Compile Input** button or the **Test Compile Expected** button compiles the entered code, and displays any errors. Clicking either Test Compile button does not commit any changes to the testcase user code.



Once the Test Compile has completed, a message window is displayed at the bottom of the dialog. You may need to re-size the dialog or scroll the window to see this message window. You can move the splitter between sections of the window by dragging it up or down. If the test compile was

successful, the message “User Code Compiled Successfully” is displayed. If there are errors, the compiler diagnostic messages are displayed.

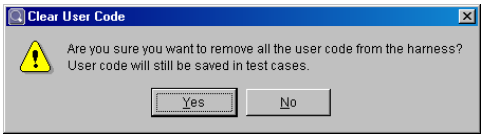
To delete all the text in the Input User Code section and start over, click the **Clear Input** button. To delete all the text in the Expected User Code section and start over, click the **Clear Expected** button.

To Clear Test Case User Code from Test Harness

The **Test => User Code => Clear All** command removes all test case user code from the test harness. The user code is not removed from the test cases (once it has been saved). This means that the next time a test case that contains user code is executed, its user code is added back into the user code unit.

The purpose of this command is to allow you to reset the user code unit in the test harness to its default (empty) state, and then selectively add the user code back in as you run tests. This process may be useful in cases where you experience unexpected results because of conflicts in your user code.

When you choose **Test => User Code => Clear All**, the following dialog box appears:



If you wish to remove all the user code from the harness, click **Yes**.

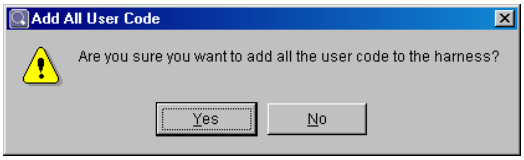
```
clicast -e <env> TESt User Clear

Remove all Test Case User Code from harness.
```

Add Test Case User Code Back Test Harness

This command adds the user code for all test cases that have user code into the user code unit of the test harness.

The following dialog box appears:



If you wish to add all the user code to the harness, click **Yes**.

```
clicast -e <env> TESt User Add

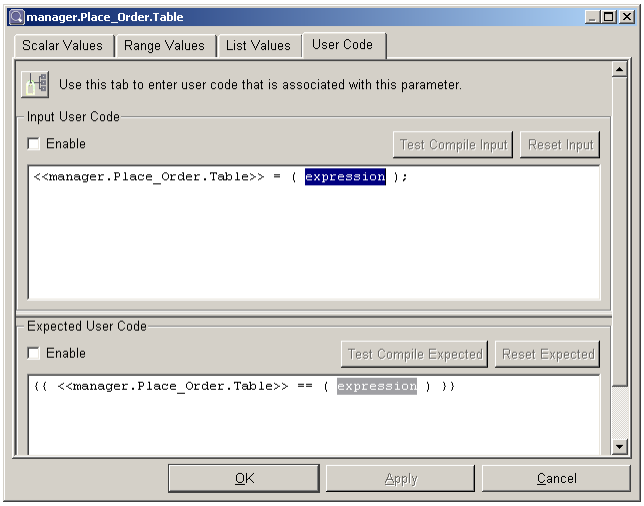
 Add all Test Case User Code into harness.
```

Parameter User Code

The Parameter User Code tab enables you to set a parameter value based on a function call, or to set or verify a parameter value against some dynamic expression.

To Enter Parameter User Code

Double-click a parameter in the Parameter Tree to access the dialog for that parameter and then click the User Code tab. The Parameter User Code tab is separated into two sections. The Input User Code section is used to set objects to the value of the parameter. The Expected User Code section is used to verify the value of a parameter. Parameter object names are specified using the VectorCAST notation of <<unitname.subprogram.parameter>>.



See also "Setting Input and Expected Values" for details on the notation for parameter object names.

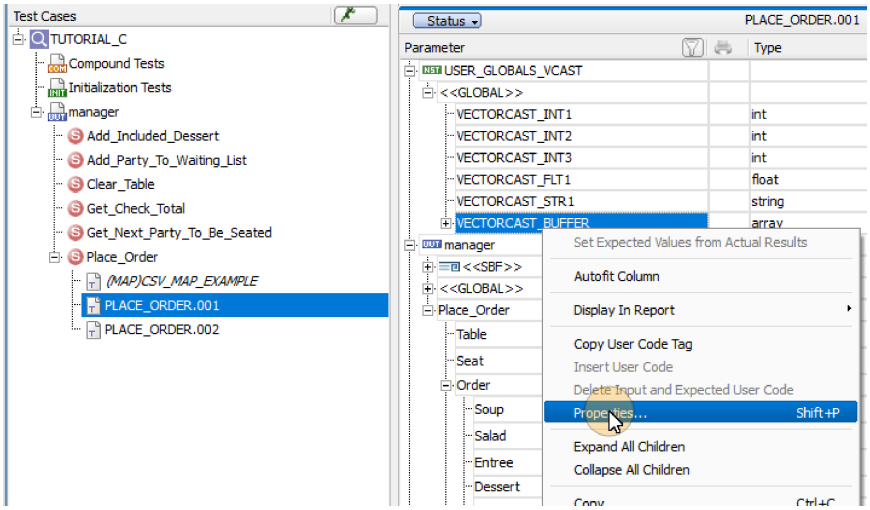
To Test Compile Parameter User Code

The code that you enter is automatically compiled into the test harness when the test case is run. The **Test Compile** buttons enable you to make sure your user code compiles before saving it. Any compiler errors will be displayed in the bottom pane of the window. The **Reset** buttons will delete any entered user code for the parameter and reset the dialog section to its initial configuration.

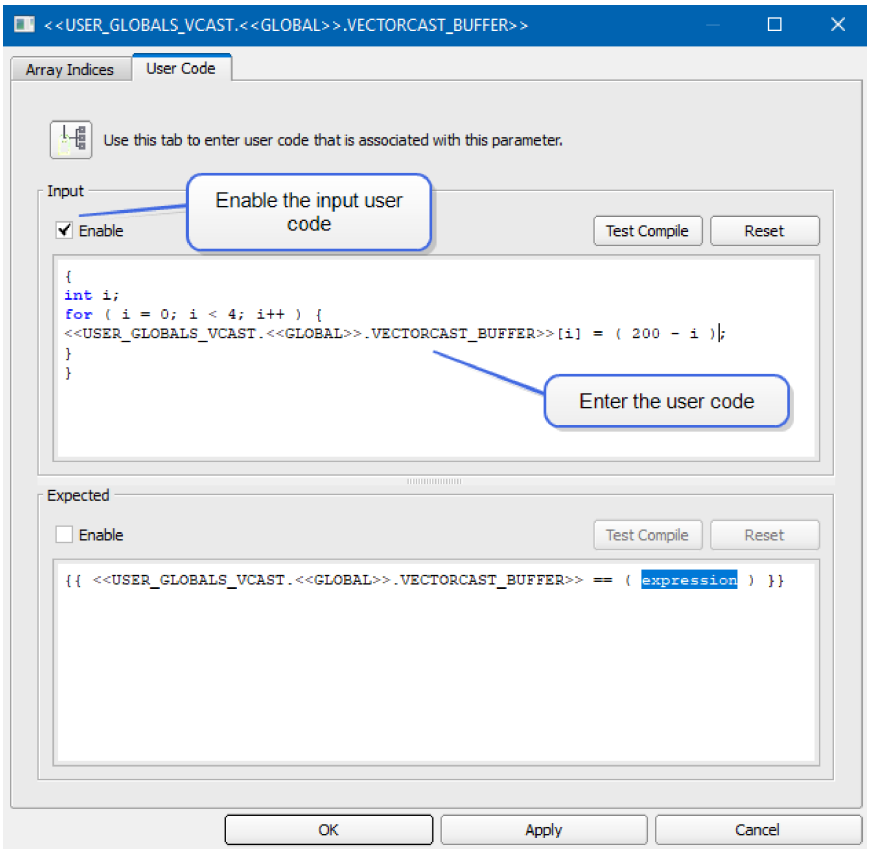
Input User Code Example

Input user code is used to set up data values before a test is run.

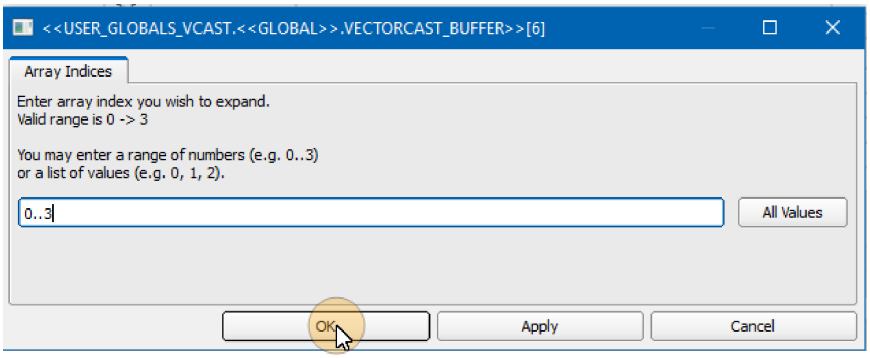
User code may consist of any number of lines of source code. For example, in the Parameter Tree, right-click on VECTORCAST_BUFFER in the <<GLOBALS>> node of a test case. Choose **Properties**, and then go to the **User Code** tab on the dialog.



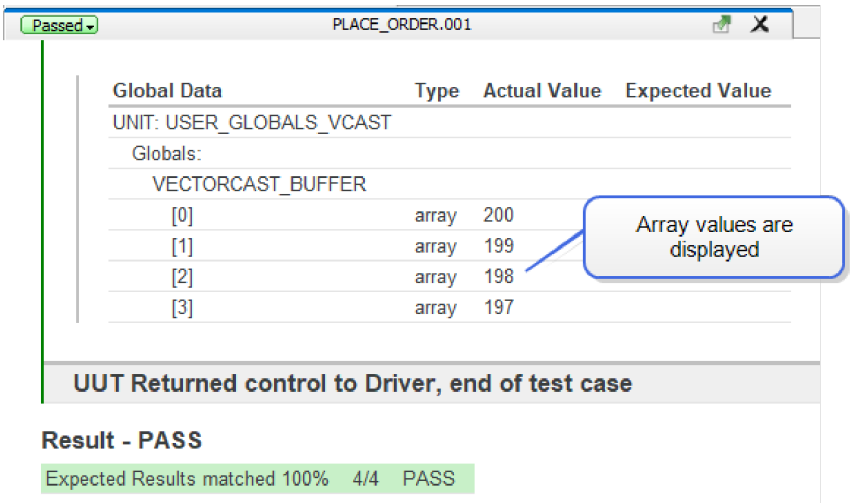
Enable the Input User Code by checking the **Enable** box, and set the VECTORCAST_BUFFER array parameter value using the following user code:



This code initializes an array "VECTORCAST_BUFFER" with the input values 200 to 197. Click on the **Array Indices** tab. Enter the values **0..3** and click **OK**.



To see the values in the Test Execution Report when the test case is executed, expand the VECTORCAST_BUFFER array. Then right-click on the VECTORCAST_BUFFER node and select **Display In Report => Show In Report** from the context menu.

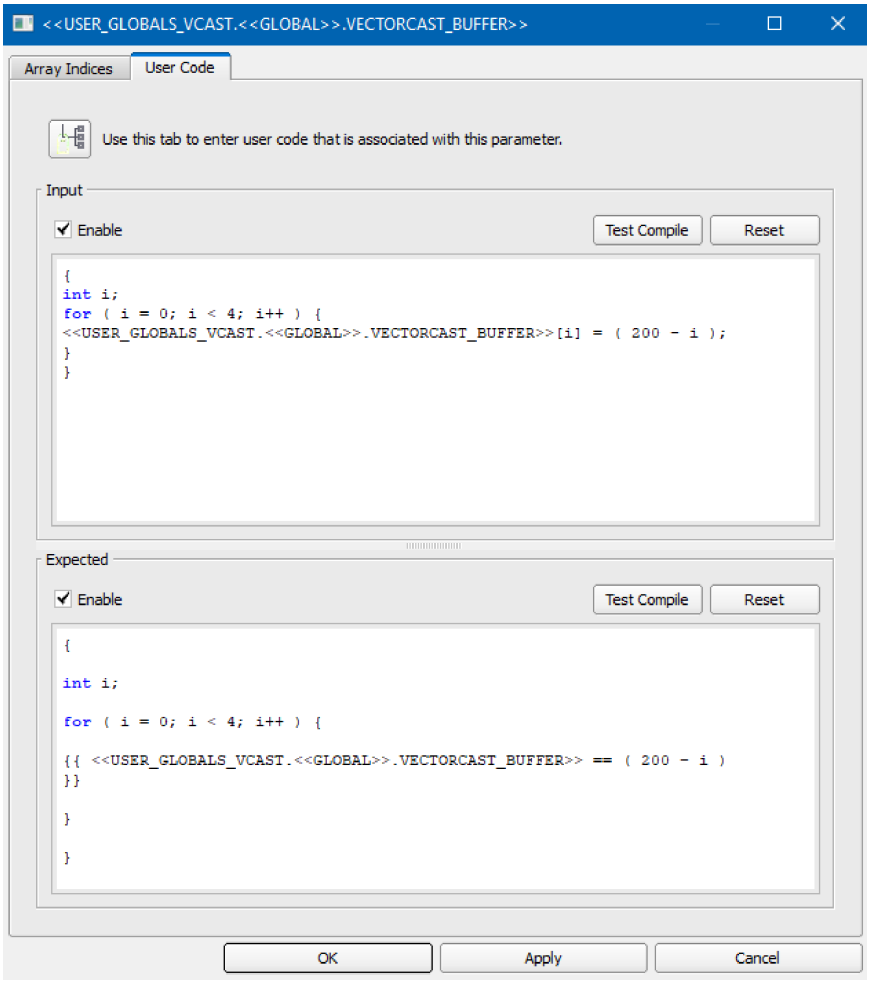


Expected User Code Example

Expected user code is used to provide an expression to test the value of the parameter after a test is run.

A boolean expression that is inside the {{ and }} markers is translated into a test expression that will output its result to the test results report. Variables are expanded as in value user code, and any user code not inside {{ and }} markers is executed as (non-conditional) normal code.

For example, you could check the value of the VECTORCAST_BUFFER array parameter using the following user code:



This code will test all 4 values of the array "VECTORCAST_BUFFER".

Display Actual Value When Entering Expected User Code

VectorCAST contains a feature which allows expected user code to be displayed in the Execution Report with an arbitrary label and with the actual value displayed.

To use this feature, your user code will need to make an explicit call to a harness function called vCAST_UC_WRITE_EXPECTED. This harness function records the result of evaluating an arbitrary test condition, along with the label and actual value you specify.

For example, you might replace user code that looks like this:

```
{{ <<uut.func.return>> == 1 }}
```

with this instead:

```
char actual[20];
double return_value = <<uut.func.return>>;
snprintf( actual, 20, "%lg", return_value );
vCAST_UC_WRITE_EXPECTED( "uut.func.return",
```

```
"Verify <<uut.func.return>> is 1.2+/-.1",
((1.1 <= return_value) && (return_value <= 1.3)), actual );
```

The prototype for vCAST_UC_WRITE_EXPECTED is:

```
void vCAST_UC_WRITE_EXPECTED (const char *param, const char *name, int
match, const char *actual);
```

where the parameters are:

- **`const char *param`** - This corresponds to the user code tag (sans angle brackets) for the parameter or global being checked. This is used by VectorCAST to determine what cell to color in the parameter tree. An empty string can be used if you do not care about the coloring of the parameter tree. If a non-empty string is used, it *must* correspond to a valid harness variable.
- **`const char *name`** - This is the text which will be used as a label for the user code when it is shown in the execution report.
- **`int match`** - If this is non-zero, the user code condition is considered matched. If it is 0, the user code is considered not matched (failed).
- **`const char *actual`** - If this is non-NULL, it should point to a printable string representing the actual value of the variable being tested.

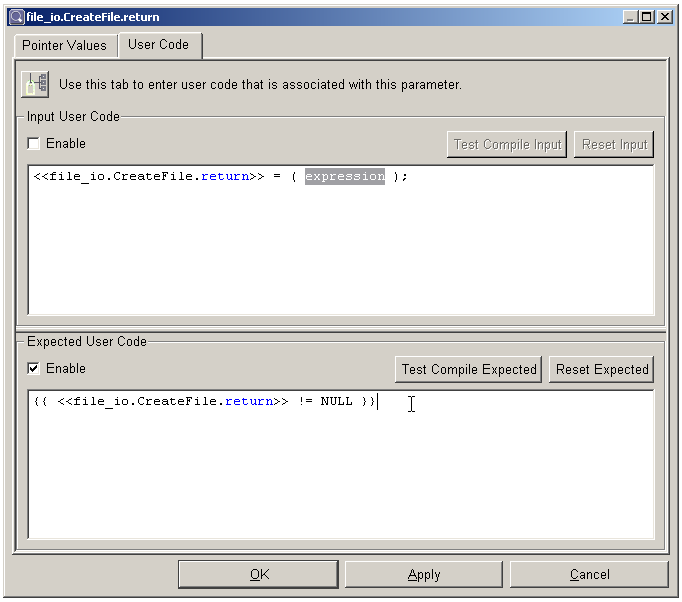
Another User Code Example

In some cases, you may want to use a parameter that is returned from one function as an input to another function. Parameter user code allows you to do this. Assume a unit writes data to a file. To test the unit, you would want the file identifier returned from the creation routine to be passed as input to the write routine.

The prototypes for the unit under test in this example might look like:

```
FILE *CreateFile ( char filename[] );
void WriteLine ( FILE *fp, char outputLine[] );
void CloseFile ( FILE *fp );
```

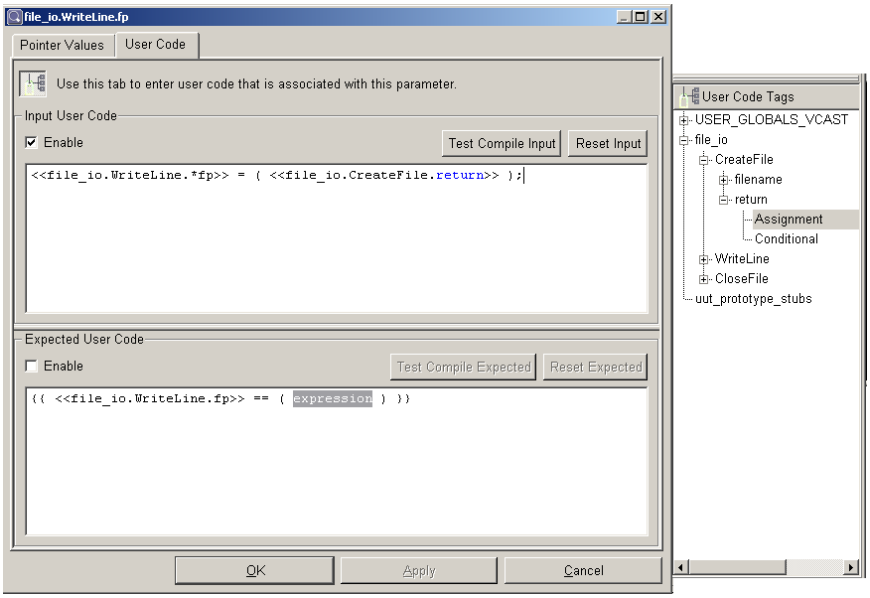
To implement this test, you would create separate tests for CreateFile, WriteLine, and CloseFile, and then link them together using a compound test. First, create a test case for CreateFile. To verify that the file pointer returned is not null, double-click on the “return” parameter, and use the Expected User Code feature as shown below.



When the test case is executed, the Test Results will have an extra section for User Code expected values, listing the parameters for which expectations were set and a flag indicating if they passed or not:

User Code Expected Values			
<<file_io.CreateFile.return>> != NULL			<match>

Next, create a test case for WriteLine using the file pointer returned from CreateFile as the input value for the fp parameter. Double-click the “fp” parameter and replace (expression) with the parameter tag for the CreateFile return, as shown here:



You can easily insert the correct parameter tag by first opening the User Code Tags window ( ),

navigating to the return parameter for file_io. Right-click Assignment and choose Copy. The tag goes into your copy buffer.

```
<<file_io.CreateFile.return>>=(expression);
```

Paste and edit so the line reads:

```
<<file_io.WriteLine.*fp>> = ( <<file_io.CreateFile.return>> );
```

This user code sets the value of the fp parameter being passed into WriteLine to the file pointer returned from CreateFile.

Stub User Code

There are several different kinds of user code for stubs: you can enter user code for a parameter, a test case, or the whole environment. Each type of stub user code has distinct characteristics, as described below.

- **Parameter User Code** – To enter expected data or input values for a stubbed subprogram, use the Parameter Tree as you would for a UUT. The user code entered here us implemented via two procedure calls (one at the start of the stub, which is *expected user code,* and one at the end of the stub, which is *input user code*). Parameter User Code is entered in the User Code tab of the Parameter Dialog. User Code that is entered in this way will only be executed when the stubbed function is called. An application of this type of user code would be to set the value of a global variable when a particular stubbed subprogram is called, or to set the value returned from a stub to a dynamic expression (e.g. **`foo()`**).
- **Environment User Code** – To enter user code for all stubbed subprograms, use Environment User Code, Stub Processing. The Stub Processing section of the **Environment => User Code => Edit dialog** is implemented in the same way as Parameter and Test Case User Code, but it is executed whenever any stubbed subprogram in the environment is called. For example, if you added stub processing user code that incremented a counter, at the end of each test execution you would have the total number of stubbed subprograms that were called during the test.
- To add user code for all stubbed subprograms to be processed upon entry to or exit from the stub, use the Stub Entry and Stub Exit section of the **Environment => User Code => Edit dialog**.
- **Configure Stubs** – Configure Stubs user code enables you to insert code directly into the definition of the stubbed subprogram. As a result, you have direct access to the named parameters of the function. An application of this type of user code is to cause a function to return a value based on the value of the input parameters.

To Enter Configure Stub User Code

Choose **Environment => Configure Stubs => Edit** to access the Configure Stubs dialog.

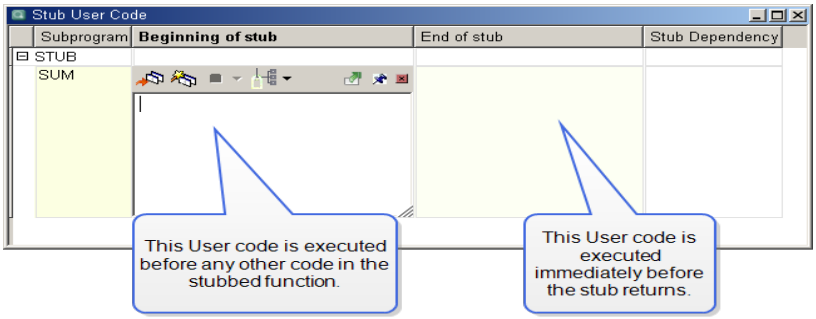
There are two editable fields on the Configure Stubs dialog. The first is marked “Beginning of Stub”, the second is marked “End of Stub.” Any code that is put into “Beginning of Stub” is placed before any other code in the stubbed function, and is the first code executed in the stub.

Any code that is put into “End of Stub” will be placed after all other code in the stubbed function (immediately preceding the **`return()`** statement).

To enter stub user code, follow these steps:

1. **Environment => Configure Stubs => Edit**.
2. Expand the node for the unit that contains the stub you are interested in.
3. Double-click in the yellow cell under “Beginning of Stub” or “End of Stub” to edit

An edit box opens in which you can type, copy/paste, and test compile  source code.



The cell sports several icons and buttons:

 – contains code (not empty)

 – needs saving

 – needs to be compiled and linked

 – currently has an error

 – test compile the unit with the appendix user code

 – Save, or Save, Compile and Link (from the drop-down menu) the unit, but does not affect the Parameter Tree of a test case until environment is rebuilt

 – pop cell out into its own window for easier editing

 – pin, or preserve size

 – close the edit box for the cell

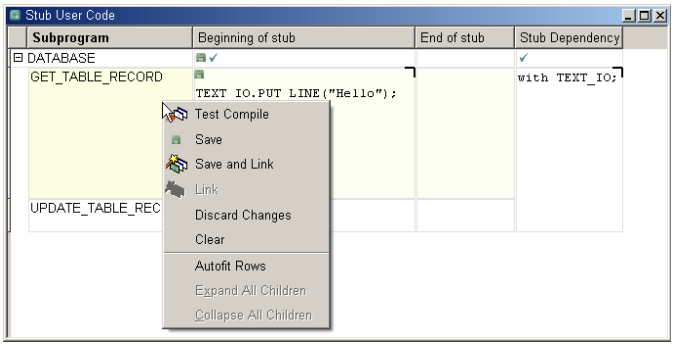
The Stub Dependency Column

The Stub Dependency column appears when dependents of the UUT are stubbed by implementation, rather than by prototype. In the Configure Stubs dialog, the Stub Dependency column appears to the

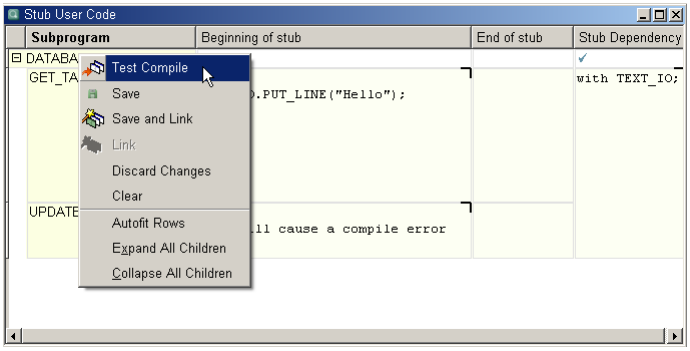
right side of the dialog, and is used to add `with` statements needed by the configure stubs user code. Code entered via the Stub Dependency column is inserted into the global (file) scope of the stubbed unit.

To Test Compile Stub User Code

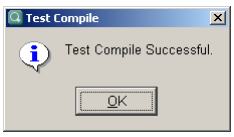
There are several ways to test compile configure stubs user code. In each individual cell, click the **Test Compile** button  to compile the code before it is saved to the test harness. Or, right-click a node and choose **Test Compile**. Either way, *both* the Beginning of Stub and End of Stub user code is test compiled, not just the individual cell.



You can also right-click a node higher in the hierarchy, such as the unit or the top level, Stubbed Subprograms. In this case, choosing Test Compile performs a test compilation on all of the cells in the scope of the right-click. The light-yellow shading of the cells indicates the current scope of a right-click. For example, in the following screen shot, the user has right-clicked on the unit manager.c. Therefore, the test compile is performed over both subprograms in the unit, and both the Beginning of Stub and End of Stub columns. As a result, the bad code in the second subprogram will cause a compile error.

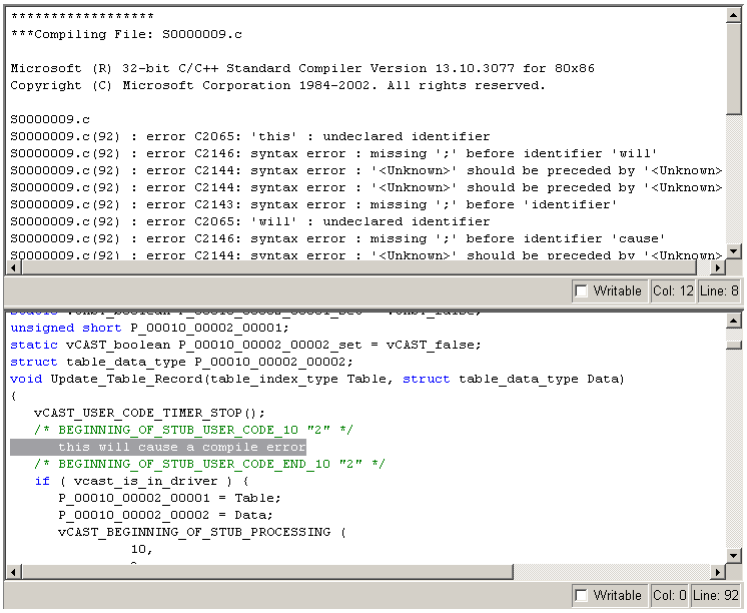


If successful, a message appears informing you, as shown in the following figure:



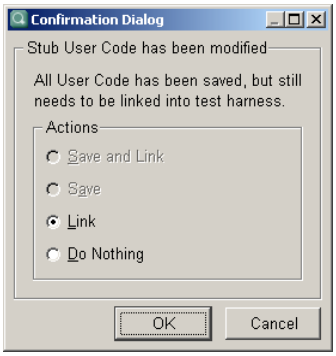
and any error state icons are removed.

If there are errors in your user code, clicking the Test Compile button opens a Stub Test Compile Errors window. The top pane shows the compile error; the lower pane shows the stub file. In the example below, the code “this will cause a compile error” is present. By using Ctrl+G in the lower pane you can easily go to the line number where the error occurred. To correct the error, close the Stub Test Compile Errors window by clicking the X in the upper right corner, edit the user code, and click the Test Compile button again until it compiles successfully.



To Save Stub User Code

To save all sections of Configure Stubs User Code, click the **Save** button on the Toolbar , right-click a node and choose **Save**, or choose **File => Save**. The following dialog appears.

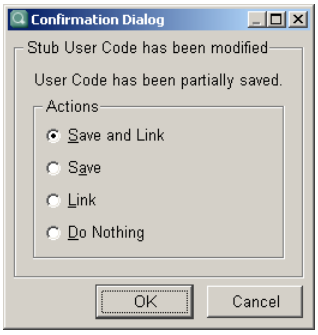


If you are ready to compile the user code and link it into the test harness, leave the radio button on **Link** and click **OK**.

If you plan to rebuild the environment anyway, put the radio button on **Do Nothing**, and click **OK**.

To cancel the save operation without saving, click the **Cancel** button.

If you attempt to close the Stub User Code window (using the window control) and there are any unsaved sections, the same dialog appears, but this time indicating that a Save needs to be performed.



Choosing the **Save and Link** radio button saves, compiles and links the user code into the test harness before closing the window. Choosing the **Save** radio button saves your changes before closing the window. Choose **Environment => Configure Stubs => Compile and Link later**. Choosing the **Link** radio button when some cells have been modified discards the changes and compiles and links the user code. Clicking the **Do Nothing** radio button discards your changes and closes the Stub User Code window.

To Recompile Stub User Code

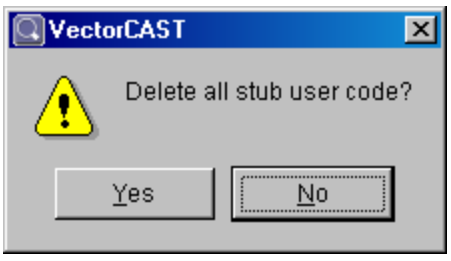
The **Environment => Configure Stubs => Compile and Link** command causes a compilation of stub user code file to be performed. Use this command when you are ready to compile the stub user code after you have edited it, but you chose **Save** rather than **Save and Link**.

```
clicast -e <env> ENvironment STub_user COmpile

Compile and link Configure Stub User Code into test harness.
```

To Remove Stub User Code

The **Environment => Configure Stubs => Clear command** will clear the stub user code from the test harness and the Configure Stubs dialog. Once selected, you see the following dialog box confirming that you wish to delete the environment user code.



VectorCAST will automatically re-compile the user code file after the user code has been removed.

```
clicast -e <env> ENvironment STub_user CLear YES

Remove Configure Stub User Code from test harness.
```

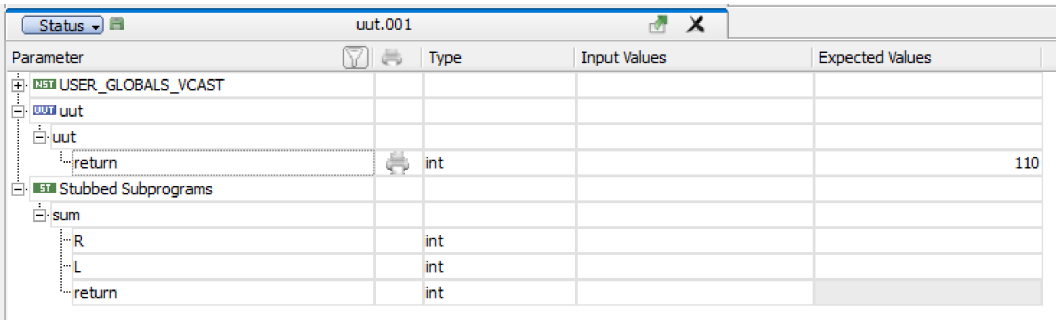
Example of Stub User Code

The following example demonstrates how to use the Configure Stubs feature to force the stub for a subprogram SUM to return the sum of its parameters. Assume that you have a subprogram **`uut()`** that calls **`sum()`** as in the following example.

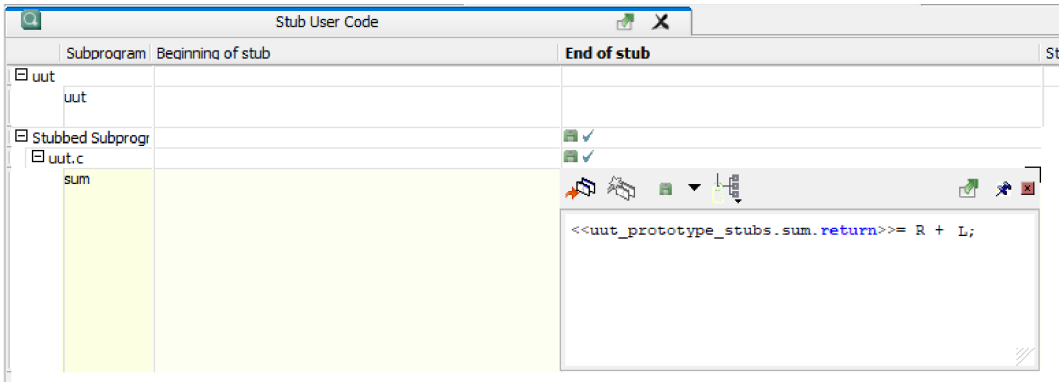
```
int sum (int R,int L);
int uut () {
int i, ret = 0;
for (i=1;i<=10;i++)
ret = ret + sum (i, i);
return ret;
}
```

The following steps can be used to build an environment and insert code to cause the sum function to return R + L.

1. Create an environment using the source code listing above.
2. Insert a test case for the function uut. Enter the value 110 for the expected value for uut.return. We expect the value 110 to be returned from the stub because (1+1) + (2+2) + (3+3) + (4+4) + (5+5) + (6+6) + (7+7) + (8+8) + (9+9) + (10 + 10) = 110.



3. Choose **Environment => Configure Stubs => Edit**. Expand the unit **`uut.c`** so you can see the stubbed subprogram **`sum`**. Double-click in the **End of Stub** cell at the end of the row for **`sum`** to open it for editing.
4. Add this code to the cell:



5. Click the **Test Compile** button  . If you have any compile errors, fix them.
6. Once the user code compiles successfully, click the **Save** button on the Toolbar  .
7. In the message that appears, leave the radio button on **Link** and click **OK**.
8. Save and execute the test case uut.001. At the end of the execution results file, you see that the stub user code summed the inputs R and L each time it was called, and the returned value was 110, which is what was expected.

Event 10 - Stubbed sum
UUT: uut.c
Subprogram: sum

Event 11 - Stubbed sum
UUT: uut.c
Subprogram: sum

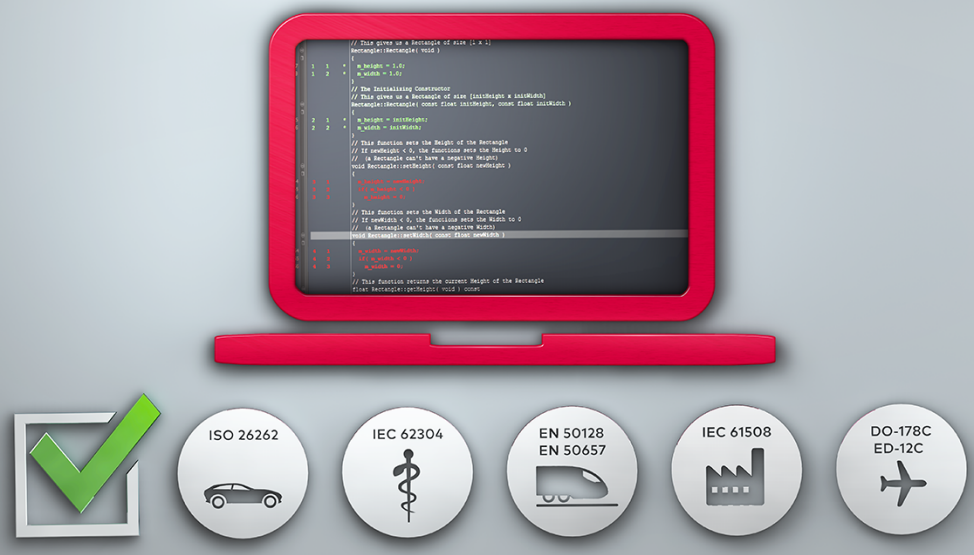
Event 12 - Returned from uut
UUT: uut.c
Subprogram: uut

Parameter	Type	Actual Value	Expected Value
return	int	110	<match>

UUT Returned control to Driver, end of test case

Result - PASS

Expected Results matched 100% 1/1 PASS



Using the Requirements Gateway

Requirements Gateway (RGW)

For assistance in migrating any legacy RGW databases to the new RGW data model (using `.json` files), contact Vector Tech Support at support@vector.com or visit https://support.vector.com/.

For VectorCAST/C++, the VectorCAST Requirements Gateway (RGW) provides traceability between software requirements and test cases and allows the import and mapping of requirements to test cases. Requirements Gateway provides a way to:

- Import testing requirements from your Requirements Management Tool (RMT) to a VectorCAST RGW repository
- Assign specific requirements to testcases in a unit test environment
- Export the resulting pass or fail status back to the RMT tool

In RGW, the requirements database is implemented using JSON, allowing the user to commit the repository to Source Control Management (SCM).

In addition to CSV files, VectorCAST RGW supports the following Requirements Management Systems:

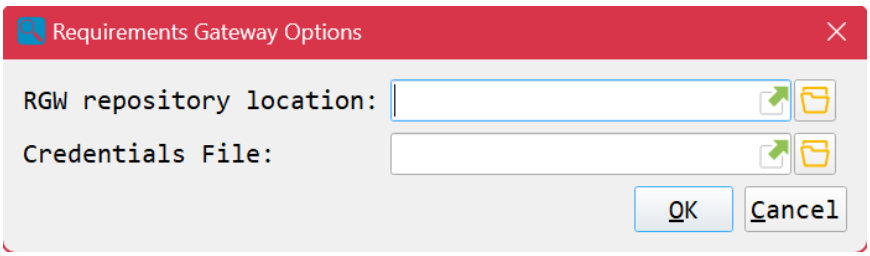
- Codebeamer
- IBM® Rational® DOORS®
- IBM ELM
- Jama
- Polarion®
- Visure

Using Requirements Gateway

The RGW repository location includes a directory named **`requirements_gateway`** which contains:

- **`export data`** - a directory which contains a separate file for each environment.
  - **`$ENV_NAME.json`** - this contains the test case data.
- **`settings.json`** - this contains the settings for the third party tool integration.
- **`requirements.json`** - this contains imported requirements data.

The location of the repository can be assigned to a directory by selecting **Tools => Requirements Gateway => Options...** from the Menu Bar.



The RGW credentials file can store the username and password for a specific user, to save time

when logging in to the RMT. The location of the credentials file can be assigned to a `.json` file by selecting **Tools => Requirements Gateway => Options...** from the Menu Bar.

Use the **Browse** button to navigate to either the location of the RGW repository or the location of the credentials file and then select the **OK** button. The paths to these values are typically specified as a full path, but can be a relative path.

Create a Requirements Repository

To use the Requirements Gateway, a location must be specified for the Repository. If no location is specified, a prompt opens allowing the user to select the location. A standard Choose a Directory dialog opens, prompting the user to enter the path to the directory.

A repository location can also be specified by selecting **Tools => Requirements Gateway => Options...** from the Menu Bar. In the **RGW Repository Location** field, enter the path to the repository and select the **OK** button.

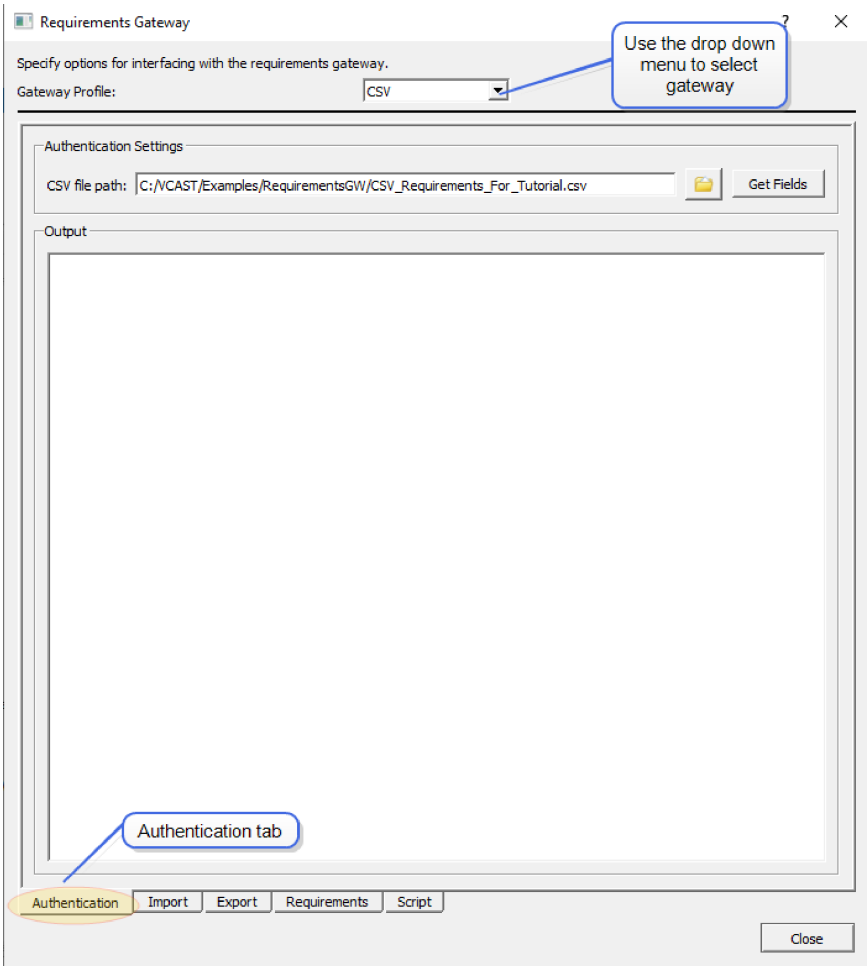
To open Requirements Gateway, select one of the following methods:

- Find the **Open Requirements Gateway**  button on the Toolbar. Select the drop-down menu next to the **Open Requirements Gateway** on the Toolbar and select the name of the gateway.
- Select **Tools => Requirements Gateway => <gateway name>** from the Menu Bar.

The Requirements Gateway dialog opens.

The Authentication Tab

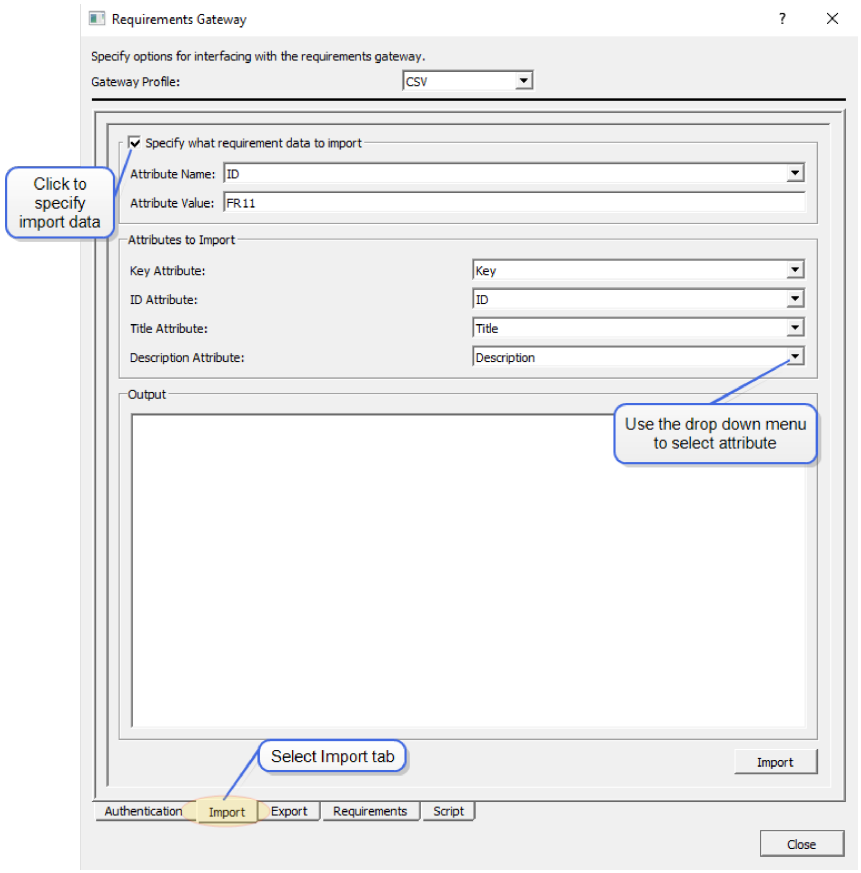
The **Authentication** tab allows the user to specify configuration options. Configuration options vary according to the gateway selected. See "Working With Supported Gateways" on page 642 to identify the configuration options for your subsystem. Use the **Gateway Profile** drop-down menu to select the gateway.



The Import Tab

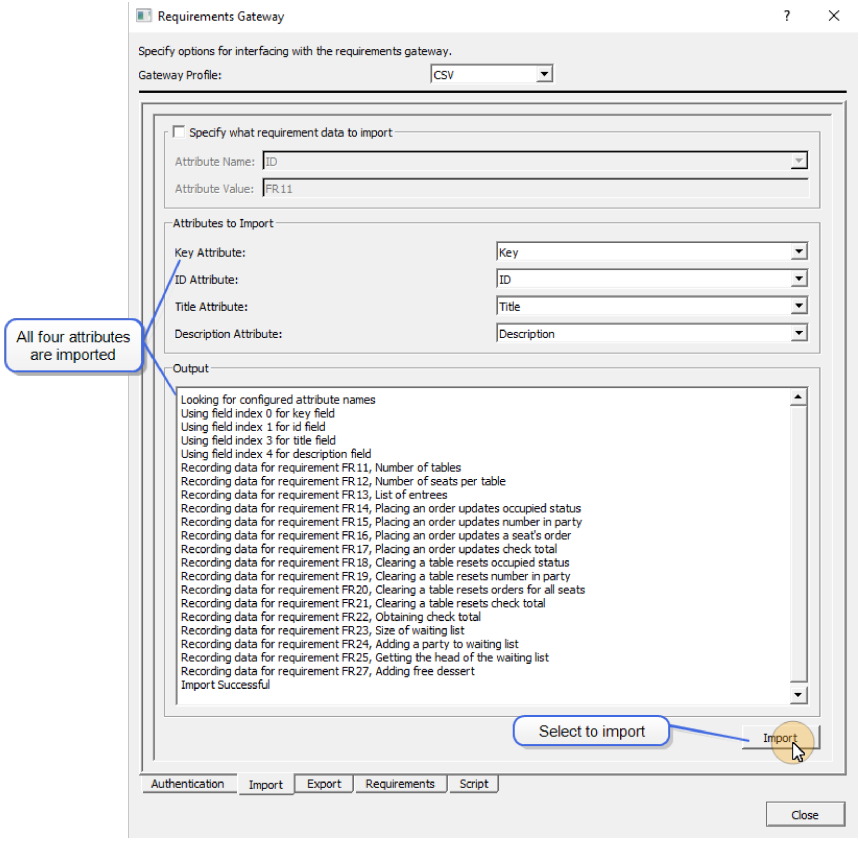
To import the requirements from the Requirements subsystem into the Repository, select the **Import** tab at the bottom of the Requirements Gateway dialog.

Configure the system to identify the data attributes to be imported. Available attributes vary according to the gateway selected. See "Working With Supported Gateways" on page 642 to identify the attributes required for your gateway. Our example below is based on a CSV gateway.



Note that the imported requirement data may be filtered by selecting the **Specify what requirement data to import** checkbox. When this option is selected, only objects which have an attribute with a specified **Attribute name** and **Attribute value** are imported.

Select the **Import** button to import the requirement data from the requirements subsystem. In our example, the option to specify what data to import was not selected, and all four attributes were imported.



The Export Tab

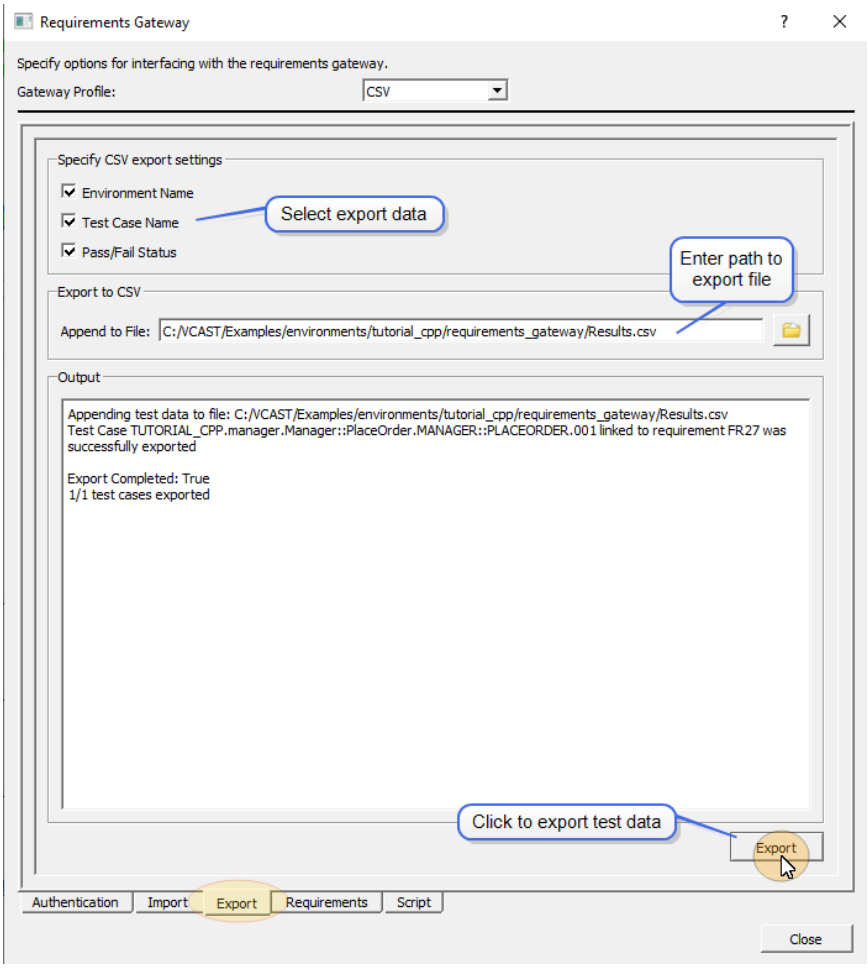
Once test case results have been obtained, the results can be exported to a new file. To export, select the **Export** tab on the Requirements Gateway dialog.

Available export settings vary according to the gateway selected. See "Working With Supported Gateways" on page 642 to identify the settings available for your gateway.

Our example shows export settings for a CSV gateway. The following test data can be exported:

- Environment name
- Test case name
- Pass/fail status

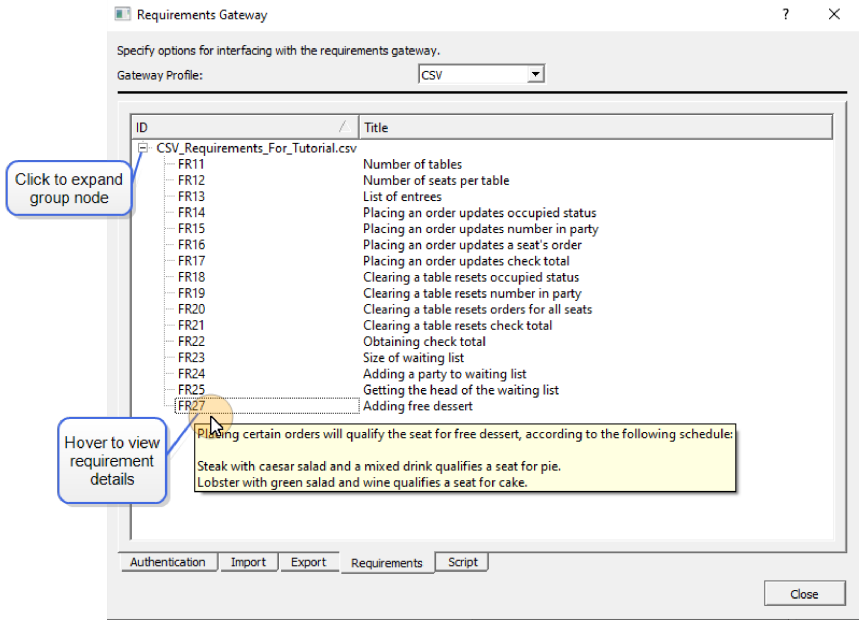
Use the browser dialog to enter the path to the export file.



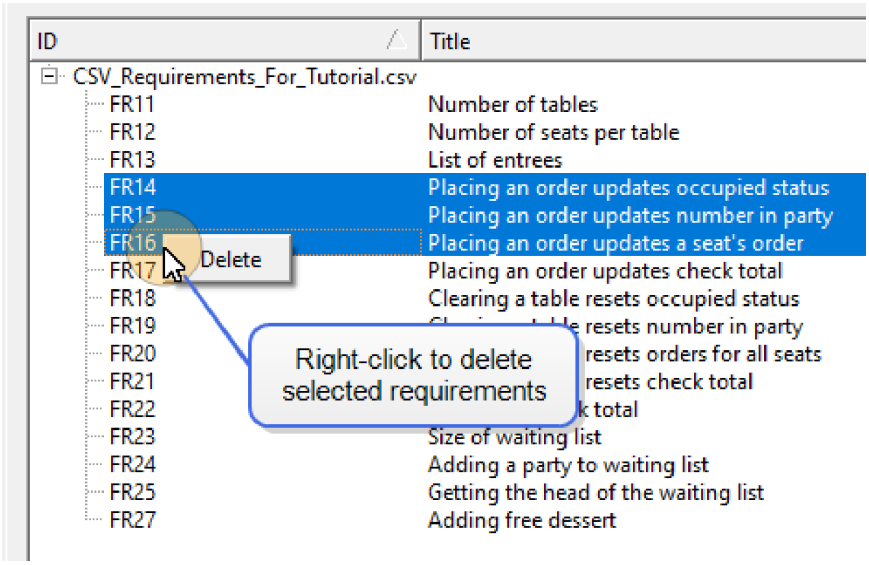
Select the **Export** button. The test data is appended to the export file.

The Requirements Tab

Select the **Requirements** tab at the bottom of the Requirements Gateway dialog to view a listing of all imported requirements. Hovering over a requirement displays a tool tip containing a detailed description of the requirement.

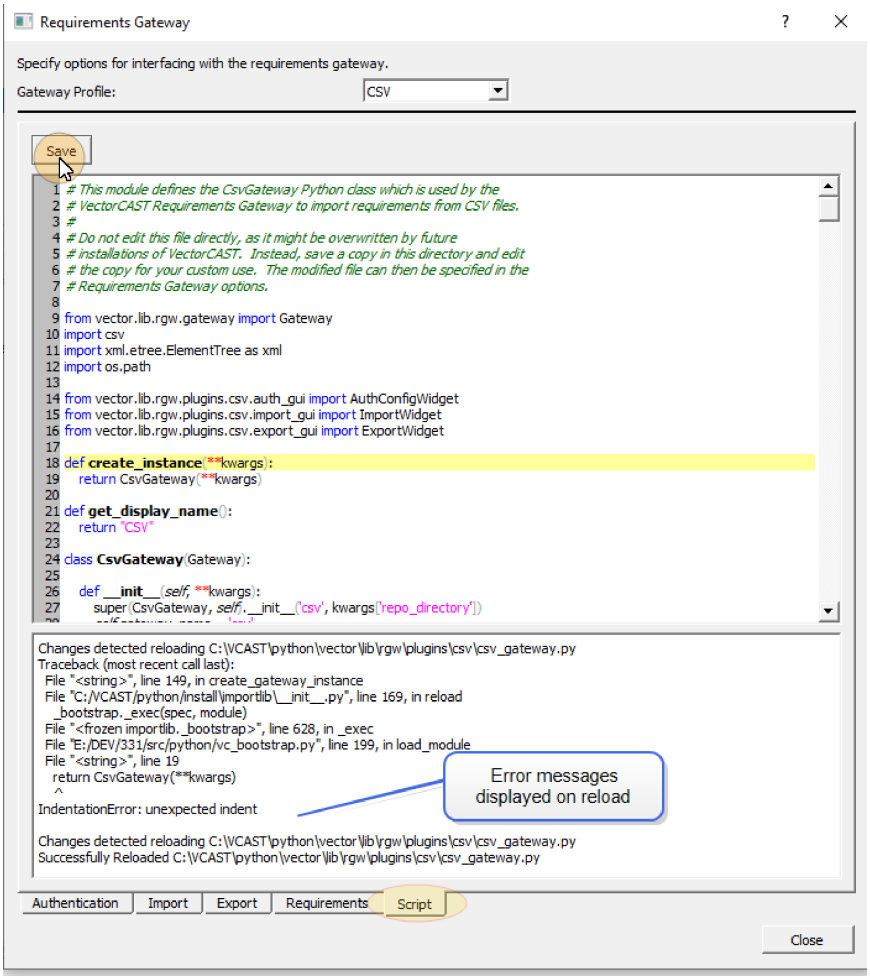


Single requirements, multiple requirements and entire requirement groups can be deleted from the repository. To delete a single requirement, select the requirement and select **Delete** from the right-click menu. To delete multiple requirements from the repository, use **ctrl-click** or **shift-click** to select the requirements and select **Delete** from the right-click menu. An entire group of requirements can be deleted by selecting the group node and selecting **Delete** from the right-click menu.



The Script Tab

To view and edit the **`csv_gateway`** Python script used to import requirements from the CSV file, select the **Script** tab located at the bottom of the Requirements Gateway dialog.



Once edits are complete, select the **Save** button to save changes and reload the Python file. If VectorCAST is unable to reload the file after editing, error messages are displayed in the message box below the script.

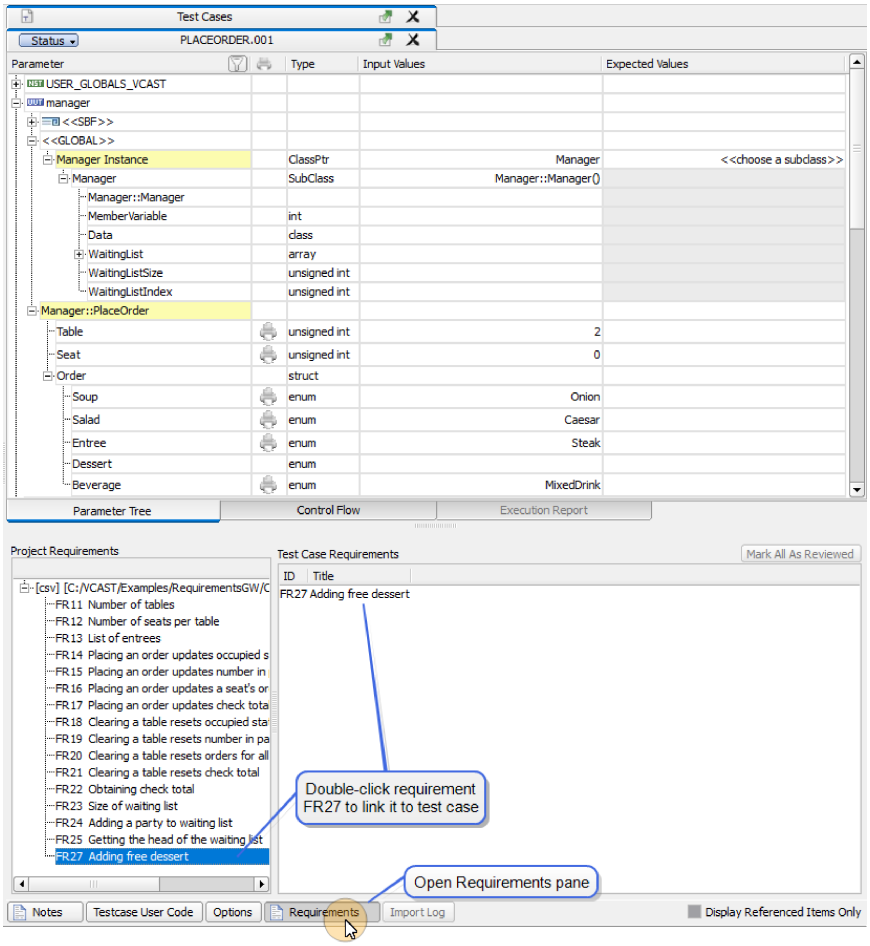
Note: Best practice is to not edit this file directly. The file may be overwritten by future VectorCAST installations. Instead, make a copy of the file and save it in the **`$VECTORCAST_DIR/python/vector/lib/rgw/plugins`** directory. Modify the file as required. Your custom file can then be specified in the Requirements Gateway options.

Link Requirements to Test Cases

To view or edit the requirements associated with a test case, open the test case and click on the **Requirements** tab in the Test Case Editor. The **Project Requirements** pane in the lower left of the MDI window contains a listing of all requirements imported into the repository. The **Test Case Requirements** pane in the lower right of the MDI window shows the requirements associated with the selected test case.

To link a requirement to a test case, double-click on the requirement in the **Project Requirements** pane. The requirement will be added to the list in the **Test Case Requirements** pane. In our example, we have linked the requirement **FR27 Adding free dessert** to test case

PLACEORDER.001 .

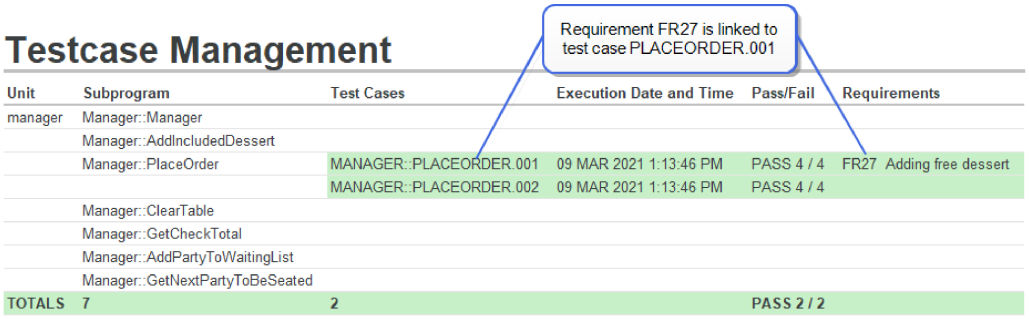


To remove a requirement from a test case, double-click on the requirement in the **Test Case Requirements** pane.

Save any changes by clicking on the **Save** icon in the Toolbar.

Once requirements have been associated with a test case, run the test case to generate a result. The result is then linked to that requirement in the repository.

Requirements that are associated with a test case are printed in the **Test Case Management Report**, which can be accessed by selecting **Test => View => Test Case Management Report** from the Menu Bar.



Tracking Requirement Change Impacts on Test Cases

VectorCAST provides users with the ability to track how VectorCAST test cases are impacted when requirements change.

The work flow for tracking the impact of requirement changes on test cases consists of the following:

1. Configure the Requirements Gateway and import your requirements for the first time.

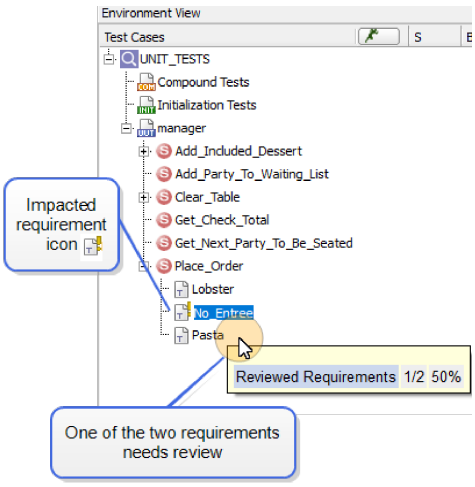
2. Link these requirements to VectorCAST test cases.

3. Make a change to some requirements in your chosen Requirement Management Tool (RMT).

4. In Requirements Gateway, re-import the requirements.

If VectorCAST detects that a requirement has been updated and it is linked to a test case in a VectorCAST unit test environment, then the requirement is considered "impacted". Any test cases linked to impacted requirements are then marked as needing review.

In the GUI, test cases linked to impacted requirements that need review are displayed in the Test Case Tree with a yellow "!" icon  . The test's tooltip also indicates how many requirements need review.

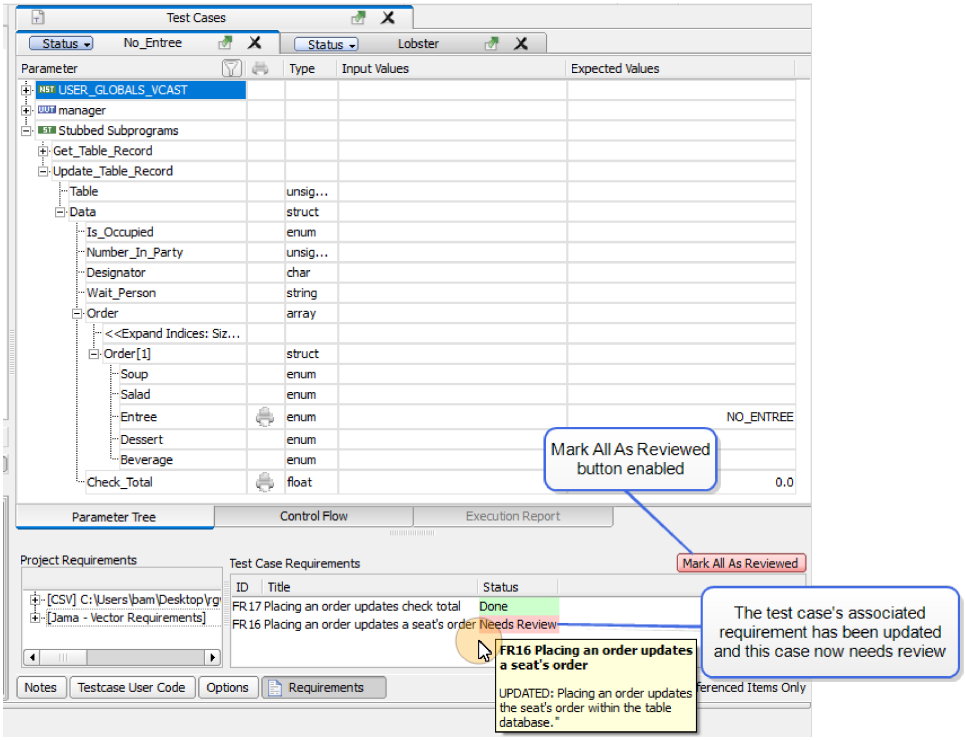


Note that the user is then prevented from changing the list of linked requirements and from exporting

test case pass / fail data back to the RMT.

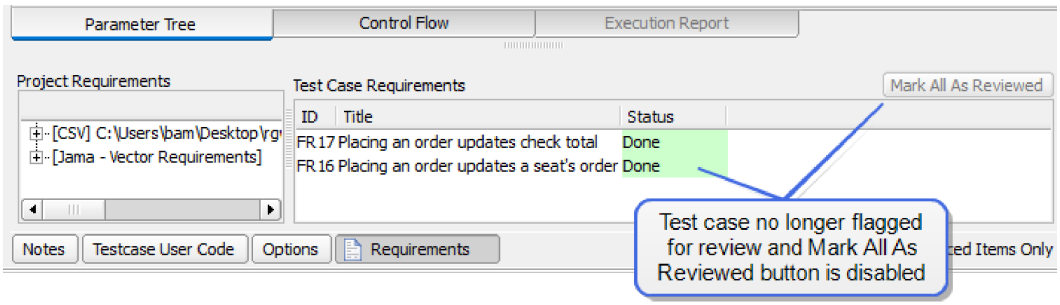
5. Verify that the test case still satisfies the test's associated requirement(s), perhaps modifying Input or Expected Values and executing as needed.

When such a test case is opened in the Test Case Editor, the impacted requirements are displayed as having a "Needs Review" status, and a **Mark All As Reviewed** button is enabled.



6. When done, mark the test case as having been reviewed.

Clicking the **Mark All As Reviewed** button clears the "Needs Review" status of the test case, and marks all impacted requirements as "Done" in that test case.



This action then frees up the restrictions on the test case, allowing the user to once again change the list of linked requirements and export the pass / fail status back to the RMT.

In the GUI, test cases linked to impacted requirements that have been reviewed are displayed in the

Test Case Tree with a green check mark icon icon  .

The Mark As Reviewed CLICAST command marks a specified test case as having been reviewed and approved, and clears the "Marked for Review" flag.

```
clicast -lc -e <env> -u <unit> -s <sub> -t <test> RGW Testcase MArk_as_
reviewed

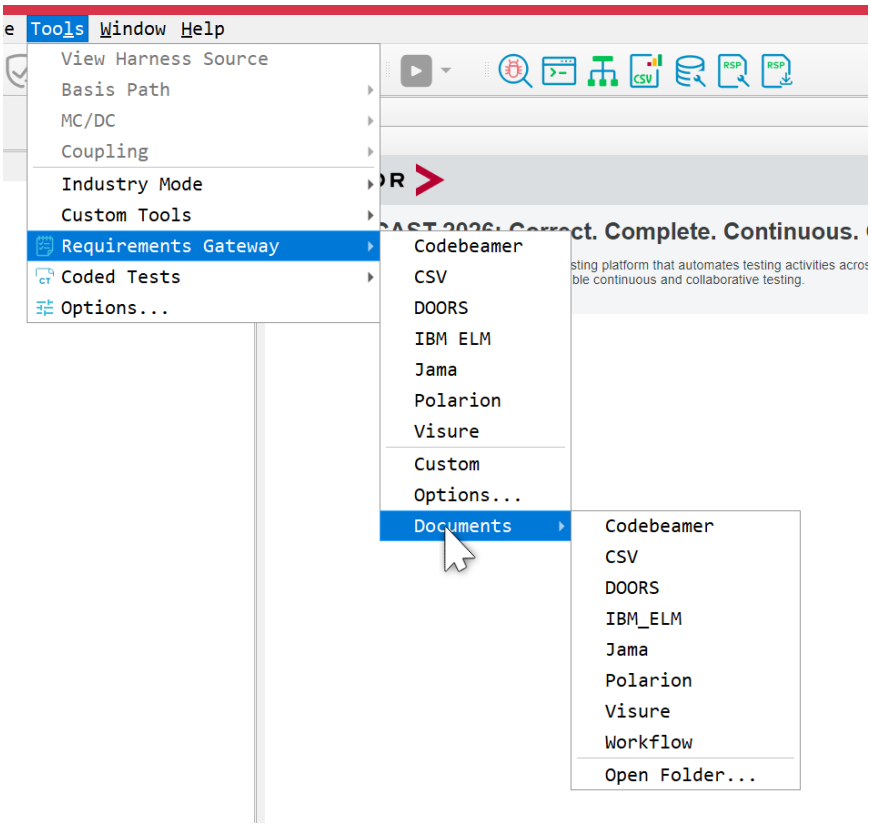
Where a Test Case is flagged as having been impacted by a Requirements
change, this marks the selected Test Case as reviewed.
```

Working With Supported Gateways

VectorCAST RGW supports several Requirements Management Systems, including:

- CSV
- Codebeamer
- IBM® Rational® DOORS®
- IBM ELM
- Jama
- Polarion®
- Visure

Instructions for these integrations are available from the Menu bar by selecting **Tools => Requirements Gateway => Documents** and then selecting the appropriate system from the menu.



To view the document in the system's Markdown (MD) viewer, select the **Open Folder...** option and then select the desired guide. For an enhanced Markdown rendering experience, the "Markdown Preview Enhanced" extension in Visual Studio Code provides excellent formatting and display capabilities.

The following sections discuss in detail how to set up and use the Requirements Gateway with the CSV Gateway.

Example Work Flow for RGW Command Line

The following example illustrates the process of setting up a CSV Requirements subsystem and importing and exporting requirements using the RGW command line.

This example covers:

- Creating and configuring the Repository
- Initializing the Gateway
- Importing requirements
- Linking requirements to test cases
- Executing test cases
- Exporting test data

This work flow uses the VectorCAST example environment Tutorial for C. Before starting the work flow example, you can automatically build the environment by selecting **Help => Example Environments => C => Tutorial for C** from the Menu Bar. Be sure that the **`TUTORIAL_C`**

environment is closed before beginning the work flow. Close the environment by selecting **File => Close Environment** from the Menu Bar.

The .csv file used in our example is located in the VectorCAST installation directory: **$VECTORCAST_DIR/Examples/RequirementsGW/CSV_Requirements_For_Tutorial.csv**.

Step 1 - Set the Repository

From a Windows command prompt, navigate to your VectorCAST Working Directory (which in our example is located at **C:\VCAST\examples\environments\tutorial_c**). Enter the following commands to first create a directory for the repository. For our example, the repository directory is named **REPO3**. Then, set the path to the repository directory:

```
$ %VECTORCAST_DIR%\mkdir REPO3

$ %VECTORCAST_DIR%\clicast -lc Options VCAST_REPOSITORY
C:\VCAST\examples\environments\tutorial_c\REPO3

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
```

Note that for the purposes of our example, the path to the new repository is located in the working directory. In actual practice, the repository is normally located in a different directory.

Step 2 - Initialize the Gateway

Enter the following commands to initialize the gateway:

```
$ %VECTORCAST_DIR%\clicast -lc RGw set gateway CSV

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
Warning: you will not be able to rebuild your environment without search
directories.

$ %VECTORCAST_DIR%\clicast -lc RGw INitialize
C:\VCAST\examples\environments\tutorial_c\REPO3

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
Attempting to create RGW Repository...
Creating RGW Repository in 'C:\VCAST\examples\environments\tutorial_
c\REPO3'
```

Step 3 - Set Configuration Options

Next, we configure the newly created Repository. The recommended workflow is to enter these

options from the GUI, using the option selection found on the Requirements Gateway dialog's Authentication and Import tabs.

From the command line, enter the following configuration options:

```
$ %VECTORCAST_DIR%\clicast -lc RGw set gateway CSV

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
Current Gateway was set to 'CSV'.

$ %VECTORCAST_DIR%\clicast -lc RGw Configure Set csv_selective_fetch 1

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
Key 'csv_selective_fetch' set to '1' for Subsystem Profile 'csv'.

$ %VECTORCAST_DIR%\clicast -lc RGw Configure Set csv_path
C:\VCAST\Examples\RequirementsGW\CSV_Requirements_For_Tutorial.csv

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
Key 'csv_path' set to 'C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG' for Subsystem Profile 'csv'.

$ %VECTORCAST_DIR%\clicast -lc RGw Configure Set id_attribute ID

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
Key 'id_attribute' set to 'ID' for Subsystem Profile 'csv'.

$ %VECTORCAST_DIR%\clicast -lc RGw Configure Set key_attribute Key

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
Key 'key_attribute' set to 'Key' for Subsystem Profile 'csv'.

$ %VECTORCAST_DIR%\clicast -lc RGw Configure Set title_attribute Title

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
Key 'title_attribute' set to 'Title' for Subsystem Profile 'csv'.


$ %VECTORCAST_DIR%\clicast -lc RGw Configure Set description_attribute
```

```
Description

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
Key 'description_attribute' set to 'Description' for Subsystem Profile
'csv'.
```

Step 4 - Import Requirements

Now that all the required system settings have been configured, import the requirements from the CSV Requirements system into the Repository by entering:

```
$ %VECTORCAST_DIR%\clicast -lc RGw set gateway CSV

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
Current Gateway was set to 'CSV'.

$ %VECTORCAST_DIR%\clicast -lc RGw IMport

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
********* starting import *********
********* import finished *********
```

Step 5 - Link Requirements to a Test Case

Next, we link specific requirements to a particular test case. In our example we will link the requirements FR11, FR12 and FR13 to the test case PLACE_ORDER.001 using the following commands:

```
$ %VECTORCAST_DIR%\clicast -lc -e TUTORIAL_C -u manager -s Place_Order -t
PLACE_ORDER.001 RGw Testcase Link FR11

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
Opening Environment
Opening Parameter/Global File
Opening Types File
Environment is Open
Successfully linked Test Case 'PLACE_ORDER.001' to Requirement with Key
'FR11'.

$ %VECTORCAST_DIR%\clicast -lc -e TUTORIAL_C -u manager -s Place_Order -t
PLACE_ORDER.001 RGw Testcase Link FR12
.
.
```

```
.
Successfully linked Test Case 'PLACE_ORDER.001' to Requirement with Key
'FR12'.

$ %VECTORCAST_DIR%\clicast -lc -e TUTORIAL_C -u manager -s Place_Order -t
PLACE_ORDER.001 RGw Testcase Link FR13
.
.
.
Successfully linked Test Case 'PLACE_ORDER.001' to Requirement with Key
'FR13'.
```

Step 6 - Run the Test Case

Now we will run the test case and gather the execution and coverage data. To run a test case from the command line, enter:

```
$ %VECTORCAST_DIR%\clicast -lc -e TUTORIAL_C -u manager -s Place_Order -t
PLACE_ORDER.001 Execute Run

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
Opening Environment
Opening Parameter/Global File
Opening Types File
Environment is Open
Preparing Test Data
Running Test Case
Running Test with command: C:\VCAST\examples\environments\tutorial_
c\TUTORIAL_C\UUT_INST.EXE
   Running Command: C:/Users/****\.vcastProcessMonitorLogs\1_Command
   Command Returned Exit Code: 0
Processing Execution Data
Updating Coverage Data
PLACE_ORDER.001 Test Execution Complete
TEST RESULT: pass
```

Step 7 - Configure the Export Settings

From the command line, enter the following export configuration options:

```
$ %VECTORCAST_DIR%\clicast -lc RGw set gateway CSV

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
Current Gateway was set to 'CSV'.

$ %VECTORCAST_DIR%\clicast -lc RGw Configure Set csv_export_path
C:\VCAST\examples\environments\tutorial_c\Results.csv

VectorCAST Copyright (C) 1993 - 2021
```

```
**Version 21 (%H%)
Processing options file C:\VCAST\Environments\CCAST_.CFG
Key 'csv_export_path' set to 'C:\VCAST\examples\environments\tutorial_
c\Results.csv' for Subsystem Profile'csv'.

$ %VECTORCAST_DIR%\clicast -lc RGw Configure Set test_name_attribute true

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\Environments\CCAST_.CFG
Key 'test_name_attribute' set to 'true' for Subsystem Profile'csv'.

$ %VECTORCAST_DIR%\clicast -lc RGw Configure Set test_status_attribute true

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\Environments\CCAST_.CFG
Key 'test_status_attribute' set to 'true' for Subsystem Profile'csv'.

$ %VECTORCAST_DIR%\clicast -lc RGw Configure Set environment_name_attribute
true

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\Environments\CCAST_.CFG
Key 'environment_name_attribute' set to 'true' for Subsystem Profile'csv'.
```

Step 8 - Export the Test Data

Now we export the test case data into the new file, **`Results.csv`**, by entering:

```
$ %VECTORCAST_DIR%\clicast -lc RGw set gateway CSV

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
Current Gateway was set to 'CSV'.

$ %VECTORCAST_DIR%\clicast -lc RGw Export CSV

VectorCAST Copyright (C) 1993 - 2021
**Version 21 (%H%)
Processing options file C:\VCAST\examples\environments\tutorial_
c\CCAST_.CFG
**************starting export*************
**************export finished*************
```

Using AI to Generate Test Cases from Requirements

The AI tool, **`reqs2tests`**, generates VectorCAST test cases from requirements. Given a set of requirements (with associated function/unit information), it uses LLM analysis to generate test cases that verify each requirement. The tool produces standard VectorCAST `.tst` files that can be

imported directly into your environment.

This command is part of a larger AI toolset that also includes a command to generate requirements from code. The entire toolset is documented as part of the VectorCAST Visual Studio Code integration: AI toolchain 'reqs2x' Documentation

CLI Example: Requirements to Tests

This AI toolset was primarily designed for use with Visual Studio Code; however, it is possible to use these AI commands in a command interpreter. In the example below, you will see the commands used to generate test cases from a VectorCAST Requirements Repository. Additional setup is required for these commands to work. Please refer to the Visual Studio Code integration: AI toolchain 'reqs2x' Documentation for more information.

The `panreq` command is used internally by the Visual Studio Code implementation but is necessary to create the traceability between the requirements and the code. The `panreq` tool converts requirements between different formats (CSV, Excel or Requirements Gateway) and (optionally) generates traceability data that maps existing requirements to their corresponding code implementations. This tool is mainly used internally in the Visual Studio Code extension, but can be used explicitly as a CLI, mainly to infer traceability on the main user-facing format: RGW. Since this command is not documented for the Visual Studio Code integration, a command reference is included here: "Using AI to Generate Test Cases from Requirements" on page 648 Using this command keeps you from manually linking each requirement to a test or tests.

Three commands are used to generate test cases from requirements. Each is shown below.

Commands used to generate test cases from requirements

```
echo === Create the traceability ^(traceability.json^) ===
panreq %REQ_REPO3_DIR% %REQ_REPO3_DIR% --target-format rgw --target-env %SUBPROJECT_NAME% --infer-traceability

echo === Generate tests with requirement keys ===
reqs2tests %REQ_REPO3_DIR% -e %SUBPROJECT_NAME% --export-tst generated_tests.tst --requirement-keys

echo === Import tests into VectorCAST ===
clicast -e %SUBPROJECT_NAME% test script run generated_tests.tst
```

The command output from running these commands is shown below.

Command output

```
=== Create the traceability (traceability.json) ===
>panreq REPO3 REPO3 --target-format rgw --target-env DOOR_LOCK --infer-traceability
Indexing codebase:
100%|████████████████████████████████████████████████████████████████████████████████| 1/1
[00:06<00:00,  6.06s/it]

=== Generate tests with requirement keys ===
>reqs2tests REPO3 -e DOOR_LOCK --export-tst generated_tests.tst --requirement-keys
Generating tests for requirements:
100%|████████████████████████████████████████████████████████████████████████████████| 7/7 [01:02<00:00,  8.93s/it]
```

```
 8
 9  === Import tests into VectorCAST ===
10  >clicast -e DOOR_LOCK test script run generated_tests.tst
11  VectorCAST Copyright (C) 1993 - 2026
12  **Version 26 revision 98e9c15 (02/12/26)
13      Processing options file \CCAST_.CFG
14      Opening Environment
15      Opening Parameter/Global File
16      Opening Types File
17      Environment is Open
18      Processing Script File
19      Processing script line 50
20      Processing script line 100
21      Script Creation Completed
22
23  --------------------------------------------------------------------------------
24  Test Script Log
25  --------------------------------------------------------------------------------
26  (I) @LINE: 1
27      >>> Opening script file \generated_tests.tst
28  (I) @LINE: 16
29      >>> Processing Test Case: Test_Engine_Not_Running_Unlock_Doors-REVIEW-NEEDED
30  (S) @LINE: 27
31      >>> Processed Test Case: Test_Engine_Not_Running_Unlock_Doors-REVIEW-NEEDED
32  (I) @LINE: 32
33      >>> Processing Test Case: Test_Lock_Doors_When_Engine_Running_No_Crash_Velocity_
    Above_50-REVIEW-NEEDED
34  (S) @LINE: 43
35      >>> Processed Test Case: Test_Lock_Doors_When_Engine_Running_No_Crash_Velocity_
    Above_50-REVIEW-NEEDED
36  (I) @LINE: 48
37      >>> Processing Test Case: Test_Lock_Doors_When_Engine_Running_No_Crash_Velocity_
    Exactly_50-REVIEW-NEEDED
38  (S) @LINE: 59
39      >>> Processed Test Case: Test_Lock_Doors_When_Engine_Running_No_Crash_Velocity_
    Exactly_50-REVIEW-NEEDED
40  (I) @LINE: 64
41      >>> Processing Test Case: Test_Return_Status_Code_Zero-REVIEW-NEEDED
42  (S) @LINE: 75
43      >>> Processed Test Case: Test_Return_Status_Code_Zero-REVIEW-NEEDED
44  (I) @LINE: 80
45      >>> Processing Test Case: Test_Undefined_Crash_State_Engine_Running-REVIEW-NEEDED
46  (S) @LINE: 91
47      >>> Processed Test Case: Test_Undefined_Crash_State_Engine_Running-REVIEW-NEEDED
48  (I) @LINE: 96
49      >>> Processing Test Case: Test_Unlock_Doors_When_Engine_Running_Crash_Detected-
    REVIEW-NEEDED
50  (S) @LINE: 107
51      >>> Processed Test Case: Test_Unlock_Doors_When_Engine_Running_Crash_Detected-
    REVIEW-NEEDED
52  (I) @LINE: 112
53      >>> Processing Test Case: Test_Unlock_Doors_When_Engine_Running_No_Crash_Velocity_
    Below_50-REVIEW-NEEDED
54  (S) @LINE: 123
55      >>> Processed Test Case: Test_Unlock_Doors_When_Engine_Running_No_Crash_Velocity_
    Below_50-REVIEW-NEEDED
56  (S) @LINE: 123
57      >>> Script processing completed
```

Once the commands are run, the unit test environment is updated with the new requirements linked to the generated test cases.

Command Reference for `panreq`

```
panreq <source_path> <target_path> --target-format <format> [--target-env
<env_path>] [--infer-traceability]
```

Command-Line Options

Option	Description
`<source_path>`	Path to the source requirements file (CSV, Excel, or RGW)
`<target_path>`	Path to save the converted/traced requirements
`--target-format`	Output format: csv, excel, or rgw (required)
`--target-env`	Path to the VectorCAST environment for traceability mapping
`--infer-traceability`	Use LLM to infer traceability from requirements to code

If --infer-traceability is given, --target-env must also be provided.

Note that if the target is RGW and --target-env is provided, the requirement gateway of the environment is automatically set to the newly created one (once a rebuild has taken place).

APPENDIX A: VECTORCAST/C++ MESSAGES

Start-up Messages

VECTORCAST_DIR Environment Variable is not set

This message indicates that an error was made in the installation of VectorCAST. Check all environment variables, symbols and path links that are required.

License Messages

FLEXlm: Licensed number of users already reached

This message indicates that an attempt was made to use more VectorCAST licenses than are authorized.

FLEXlm: Cannot find license file

This message indicates that license processing could not be completed. Check all environment variables, symbols and path links that are required.

FLEXlm: Feature has expired

This message indicates that the VectorCAST license has expired.

LICENSE ERROR: License Failure

This message indicates that some license anomaly was detected subsequent to the initial invocation of VectorCAST and will not allow VectorCAST to continue. All data will be saved and VectorCAST will be automatically terminated. In general, a FLEXlm error number is displayed. The meaning of the error code is available in the FLEXlm End User's Guide: **VectorCAST installation directory => DOCS => fnp_LicAdmin.pdf**.

Environment Building Informational Messages

Calling QuickParse Utility for: *directory name*

This message is displayed when the quickparse utility has been invoked for a specific directory. The directory_name will be one of the directories provided to VectorCAST as a search directory when building a test environment.

QuickParse Utility Completed

This message is displayed when the quickparse utility has completed executing for a specific directory.

Parsing UUT body for whitebox conversion
Processing whitebox data

These message are displayed during the environment construction when VectorCAST ios performing the parsing and conversion of the unit under test that is required for whitebox testing.

Processing UUT *unit name*

This message indicates that the parsing of the Unit Under Test is being accomplished by the VectorCAST environment builder.

Processing *unit name*

This message indicates that the indicated unit is being parsed and added to the current environment.

Building Driver

This message indicates that the Ada language driver program is being built by the VectorCAST environment builder and generator.

Building Harness for *unit_name*

This message is displayed when VectorCAST is building the data handling functions for a particular unit.

Compiling *unit name*

This message indicates that elements of the environment have been created and they are now being compiled into the Test Ada Library by VectorCAST.

Linking Environment

This message indicates that all elements of the environment have been created and they are now being linked into the Test Ada Library by VectorCAST.

Environment Built Successfully

This message indicates that all elements of the VectorCAST environment have been successfully constructed.

Environment Building Error Messages

ERROR: Compile Failed, unit: *unit name*

This message is displayed to indicate that the compilation of a particular harness source file has failed. The compiler diagnostic message will be displayed to help resolve the issue.

ERROR: Compile Failed, stubbed unit: *unit name*

This message indicates that during the compilation of the VectorCAST Test Harness components, a compile error occurred. The compiled error listing will be displayed in the VectorCAST display and will also be written to the file ACOMPILE.LIS.

ERROR: Abnormal Termination of Compile and Link

This message is displayed when VectorCAST encounters an unexpected error during the compilation or link of the test harness. For example this message may reflect the inability to create a file because of disk space or permission problems.

ERROR: Environment Directory Creation Failed

This message indicates that some sort of operating system or protection error occurred when VectorCAST attempted to create the disk directory that will contain the environment files

ERROR: Cannot Overwrite Environment

This message indicates that you have attempted to create a VectorCAST environment in a situation where a directory already exists with the same name. The directory may or may not be a VectorCAST environment.

ERROR: Driver Program did not link

ERROR: Instrumented Harness did not link

ERROR: Data Program did not link

These messages indicate that an executable program created by VectorCAST did not link properly. The linker diagnostic listing will be displayed and will be written to the file AALINKER.LIS.

ERROR: Driver Could not be Invoked

ERROR: Data Interface Could not be Invoked

These messages indicates that VectorCAST cannot execute one of the executables that it builds. Probable Cause: a previous error during environment creation, an operating system protection error on program execution, or "." is not on the user's path.

ERROR: Environment Creation Failed

This message indicates that the VectorCAST environment builder was not able to complete its processing normally. Nothing is usable in the partially created environment, and it will be deleted.

ERROR: *environment name* Could not be Created

This message indicates that a serious error occurred during environment creation. File protection or limit conditions may exists.

ERROR: *environment name* Could not be Deleted

This message indicates that there are unexpected files in the environment subdirectory or for some other reason the files in the environment subdirectory could not be deleted.

ERROR: *environment name* Could not be Opened

This message indicates that the environment selected either does not exist or has been corrupted.

INFO: Environment Built but not Compiled

This message indicates that the environment creation is complete, however there was a compile failure during creation that must be resolved before continuing.

INFO: Environment built but not linked

This message indicates that the environment creation is complete, however, there was a link failure during creation that must be resolved before continuing.

Deleting Files From Environment Build

This message is displayed if the environment creation aborts for any reason. VectorCAST will automatically delete all files that were created during the aborted environment build.

Test Case Messages

ERROR: Test Execution Failed. Could not execute UUT interface.

This message indicated that the execution of the test harness did not produce the expected output files for VectorCAST to produce the test reports. Common causes are a fatal error in the initialization of the test harness, or problems with the configuration of the target if you are using VectorCAST/RSP. To determine the cause, run the test case using the VectorCAST option: “Execute with debug” and step through the code under test.

ERROR: No Change Made to Test Data

This message is displayed during the creation of a test case, this error message is displayed to notify the operator that the entered change was not affected.

ERROR: Illegal Numeric Entry

This message indicates that an operator entered parameter value entered during test case generation is not consistent with the numeric value required by the context.

ERROR: *type mark* Could not be Found

This message indicates that the displayed type mark is of a type that could not be found in the environment.

ERROR: Value Out of Range - no Change Made to Data

This message indicates that a particular test data item is outside the allowable range for that type, and that no further processing will be performed.

INFO: Invalid Test Case Name

This message indicates that the selected test case name contains illegal characters or space characters.

INFO: No Results Exist

This message indicates that you have selected an operation dependent on the actual results file. However, no expected results exist for this test case.

INFO: Type Not Supported

This message is displayed when a type is encountered that is not supported by VectorCAST. See "Limitations and Restrictions" on page 690 for more information.

INFO: Index Expression Error

This message indicates that an operator entered array index value is invalid for the array specified.

Building Test Case Data

This message indicates that the ASCII data is being created which will allow viewing and printing of test cases.

Test Case Variants Messages

Information Messages

Logic file changes detected, reloading logics

This message is displayed when you modify the Variant Logics file held in the environment using an external editor and then perform a test case related action, such as execution, test script import, etc.

Variant Logics path saved to environment

Variant Logics path saved to CCAST_.CFG file

These messages are displayed when you save Logic Editor contents to a new Variant Logic file path, i.e., the messages only appear when “Save As” is performed.

Error Messages

Error evaluating Variant Logics file, refer to Logic Editor for details

This message is displayed after saving the Logic Editor and not fixing errors raised in the Output Pane prior to proceeding onto performing test case actions such as execution.

Invalid logic name entered, reset to previous value

This message appears when typing a non-existent logic name into a Logic Assignment Table row’s logic cell edit.



Note: The following error messages are not due to user error and instead are raised by programmatic errors. Please contact support if you see any of these error messages.

Error: Could not update test case name in Logic Assignment due to missing name

This message appears when the Logic Assignment Widget fails an attempted refresh upon a test case being renamed in the Test Case Tree.

Error: Could not update status for testcase logic assignment

Error: Could not update test case status in Logic Assignment due to missing name: <logic name>

These messages are displayed when test case logic assignment fails in the Logic Assignment Widget.

Logics reload failed

This message is displayed when you modify the Variant Logics file held in the environment using an external editor and then perform a test case related action, such as execution, test script import, etc.

Error retrieving Variant Logics Path from environment

Could not read ENVIRO.AUX, ‘<path>’ is not an environment directory

Not in a VectorCAST environment directory, thus Logics file not found

ENVIRO.AUX path not set or does not exist

These messages are displayed when performing actions, e.g. test case execution or test script import, which require use of the Variants Logics file, whose path cannot be fetch from the environment.

Could not save Variant Logics path to environment

This message is displayed when an attempt to save Logic Editor contents to a new Variant Logic file path fails.

Logic Editor failed to save

This message is displayed when an attempt to save the Logic Editor when changes were made fails.

CLICAST Messages

Variant Logic '<logic name>' assigned to test case '<test case>'

This message appears upon successful completion of logic assign command being run.

Error: Must specify a variant logic. The variant logics currently available are: <logic list>

This message appears upon unsuccessful completion of logic assign command being run. The command failed because no logic was passed by argument.

Test Variant Logic Name '<logic name>' not found

This message appears upon unsuccessful completion of logic assign command being run. The command failed because the logic name passed does not exist in the environment's Variant Logic file.

Variant Logic unassigned from test case '<test case>'

This message appears upon completion of logic unassign command being run.

Error evaluating Variant Logics file, evaluate file for details

This message is displayed when doing test case actions while the environment's Variant Logic file contains issues.

General Messages

Building Environment Script file

Building Test Case Script file

Building the CLI Script

These message are displayed when VectorCAST is building the three script files required to for regression testing.

Creating environment script

This message is displayed while VectorCAST is creating an environment script file

Script Building Completed

This message is displayed when VectorCAST is done building a test case script or an environment script.

Building Test Case Script Template

This message is displayed while VectorCAST is creating an test case script template

Linking Instrumented Harness

This message is displayed when you initialize coverage, or re-link the test environment and the instrumented test harness is being linked.

Getting Size and Range Data

This message is displayed when VectorCAST is building its database of range data for all of the various parameters, global objects and data types that are part of the test environment.

Down Loading Target Processor

Running transfer to execute UUT on: *board_name*

Running tornserv to execute target program

(Target Only) These message are displayed while VectorCAST is connecting to the target and downloading the test harness executable.

INFO: Invalid Environment Name

This message indicates that the entered environment name contains a space or some other character that is an illegal directory name on the host platform.

***environment name* Deleted Successfully**

This message indicates that the environment chosen for deletion has been completed.

Could not open environment *environment_name*

This message is displayed when there is an error opening an environment. Possible reasons are disk space and file protection problems.

Determining Basis Paths

This message is displayed when VectorCAST is performing the basis path computation for a particular unit.

Error Computing Basis Paths

This message is displayed when VectorCAST cannot compute the basis path listings for a module. Errors of this type should be reported to VectorCAST Technical Support (support@vector.com)

Error Initializing Coverage

This message is displayed when VectorCAST cannot perform the coverage instrumentation

processing for a module. Errors of this type should be reported to VectorCAST Technical Support (support@vector.com)

Rebuilding instrumented harness

This message is displayed when the user selects the “re-link” command and coverage is enabled. VectorCAST will automatically retrieve the latest version of the code under test, and re-instrument the code.

Error compiling instrumented file

This message is displayed when the compilation of an instrumented source module fails. Errors of this type should be reported to VectorCAST Technical Support (support@vector.com).

Building Min Mid Max Test Cases

This message is displayed while VectorCAST is generating the Min, Mid or Max test cases.

Environment has not been Compiled/Compile Environment Before Proceeding!

Environment has Unresolved Compile Errors/Resolve Errors Before Proceeding!

This message is displayed when an environment has been opened in VectorCAST and there are unresolved compile errors. This message indicates that no testing can be performed until the compile errors are resolved.

Environment has Unresolved Link Errors/Resolve Errors Before Proceeding!

This message is displayed when an environment has been opened in VectorCAST and there are unresolved link errors. This message indicates that no testing can be performed until the link errors are resolved.

Harness has Run-Time Errors / Resolve Errors Before Proceeding!

This message is displayed when an environment has been opened in VectorCAST and there are unresolved run-time errors. This message indicates that no testing can be performed until the run-time errors are resolved.

Updating Coverage Data

Updating Aggregate Coverage Report

This message is displayed after a test case execution when VectorCAST is updating the coverage data to reflect the test that was just run.

Environment Data not Expanded

Cannot Create Min Mid or Max Test Cases

This message indicates that there was some kind of unexpected error during the generation of the parameter and global data range information. As a result, testing may be performed, but the generation of Min, Mid, Max test cases is unavailable.

Reading Instrumentation Data Line: *line_number*

Sorting Instrumentation Data Line: *line_number*

These message are displayed to provide an update to the user when large coverage data files are being processed . The line numbers will update in 1000's as the file is processed.

ERROR: Could Not Build Test History

This message indicates a serious error occurred during the test execution and the expected output data did not get produced by the test harness. With Target testing this can indicate that the target communication failed. With host testing this can indicate a file lock, disk space, or permission problem.

CLICAST Messages

Environment must be specified on command line

Environment *environment_name* is invalid

These messages are displayed when you issue a clicast command that requires the name of an environment, and you do not provide an environment name or provide an invalid name as a command option.

Subprogram must be specified on command line

Subprogram *subprogram_name* is invalid

This message is displayed when you issue a clicast command that requires the name of a subprogram, and you do not provide a subprogram name or you provide an invalid name as a command option.

Test case must be specified on command line

Test Case *testcase_name* is invalid

This message is displayed when you issue a clicast command that requires the name of a test case, and you do not provide a test case name or you provide an invalid name as a command option.

No files specified

This message is displayed when a clicast command requires an output filename and no filename is provided.

Compiler *compiler_name* not supported

This message is displayed when an unknown compiler name is provided for a clicast command

CLICAST failed - parameter inconsistency

This message is displayed when you attempt to invoke clicast with a combination of parameters and options that VectorCAST cannot understand.

CLICAST failed with unknown exception

This message is displayed when the execution of a clicast command fails and there is no additional information available.

APPENDIX B: ENVIRONMENT SCRIPT LANGUAGE

The VectorCAST environment scripting enables you to build environments using text files. Scripting is provided as an alternative to entering commands interactively via the VectorCAST GUI.

The Environment Scripting language corresponds to the data required to construct an Environment using the GUI. Each command is entered on a single line. The script commands are not case sensitive, though the data may be.

Note the following conventions:

- All commands start with "ENVIRO." to indicate that an Environment command follows.
- All commands that require data must use a colon (:) to indicate that start of the associated data.
- Comments may be inserted, and are indicated by inserting the VectorCAST comment delimiter "--" as the first item on the line.

The following is a list of the valid script commands with a description of each command.

Enviro Command List

ENVIRO.NEW

Required. This command indicates that a new environment is to be created. ENVIRO.NEW must be the first non-comment line in the environment script.

ENVIRO.NAME:

Required. This command is used to provide a name for the Environment. Unlike test case scripts, the "name" command is mandatory. Since the Environment name will ultimately be the name of a sub-directory, it must be unique within its parent directory and not contain spaces.

ENVIRO.UUT:

This command indicates a Unit Under Test filename. If you have more than one UUT, list each unit name separately, one per line, each with its own ENVIRO.UUT command. If a unit is listed twice, in two different commands (i.e. ENVIRO.UUT and ENVIRO.STUB), then the first time the unit's name is encountered takes precedence over other appearances.

The environment script must include at least one ENVIRO.UUT line or ENVIRO.STUB_BY_ FUNCTION line.

ENVIRO.STUB_BY_FUNCTION: *unit*
ENVIRO.SBF: *unit*

This command indicates a Unit Under Test filename, whose individual functions are to be made stubbable at run time. If you have more than one UUT to be stubbed by function, list each unit name separately, one per line, each with its own ENVIRO.STUB_BY_FUNCTION command.

The environment script must include at least one ENVIRO.UUT line or ENVIRO.STUB_BY_FUNCTION line.

ENVIRO.CUSTOM_COVERAGE: *<unit>:<ON | OFF>*

Unit-specific coverage enable flag. Supports the specification of coverage on non-stubbed dependent units in test environments. Custom coverage on these units can be applied during environment build and retained during environment rebuild.

ENVIRO.SEARCH_LIST:

Required. This command specifies the path to a directory where source files exist. On platforms where the operating system is case sensitive (i.e. Linux) the exact case matched path must be entered. You may use multiple ENVIRO.SEARCH_LIST commands in a script to provide more than one directory containing source files.

The search list can be absolute, or relative to the working directory. The list may include environment variables using the syntax: $(ENV_VAR). For example: ENVIRO.SEARCH_LIST: $(VECTORCAST_DIR)/Tutorial/cpp

ENVIRO.LIBRARY_INCLUDE_DIR:

This command specifies the path to a directory where system headers or third-party libraries reside. On platforms where the operating system is case sensitive (i.e. Linux) the exact case matched path must be entered. You may use multiple ENVIRO.LIBRARY_INCLUDE_DIR commands in a script.

The library include list can be absolute, or relative to the working directory.

ENVIRO.TEST_VALUES_DICTIONARY:

This command specifies the path to a Test Values Dictionary file.

ENVIRO.VARIANT_LOGICS_PATH:

This command specifies the path to the Variant Logics file. On platforms where the operating system is case sensitive (i.e. Linux) the exact case matched path must be entered.

The path to the Variant Logics file can be absolute, or relative to the working directory and can contain environment variables, using the syntax $(ENV_VAR) or %ENV_VAR%.

ENVIRO.TYPE_HANDLED_SOURCE_DIR:

This command specifies the path to a directory where files providing data type handling. On platforms where the operating system is case sensitive (i.e. Linux) the exact case matched path must be entered. You may use multiple ENVIRO.TYPE_HANDLED_SOURCE_DIR commands in a script.

The type-handled source list can be absolute, or relative to the working directory.

ENVIRO.TYPE_HANDLED_DIRS_ALLOWED:

This command is written by the environment builder version 4.2 and greater. It signifies that the 3-line system (ENVIRO.SEARCH_LIST, ENVIRO.LIBRARY_INCLUDE_DIR, and ENVIRO.TYPE_ HANDLED_SOURCE_DIR) are being used, instead of reading library include directories from the CCAST_.CFG file. This command does not take an argument. Its presence alone is what is needed in the environment script.

ENVIRO.WHITE_BOX:

This command is optional and if not given, defaults to blackbox. If the Whitebox option is desired then the command "YES" is used.

ENVIRO.COVERAGE_TYPE: *type*

This command is optional and if not given, defaults to no coverage (None). To specify a coverage type, *type* can be None, STATEMENT+MC/DC, STATEMENT+BRANCH, FUNCTION+FUNCTION_ CALL, FUNCTION,MC/DC, BASIS_PATHS, BRANCH, STATEMENT .

ENVIRO.STUB:ALL_BY_PROTOTYPE

This command is used to tell VectorCAST to stub all dependent units *by prototype* during the environment creation. It can be combined with ENVIRO.STUB:<unit> to indicate that all units be stubbed by prototype except <unit>, which should be stubbed by implementation. Similarly, it can be combined with ENVIRO.DONT_STUB:<unit> to indicate that all units be stubbed by prototype except <unit>, which should be non-stubbed.

ENVIRO.STUB:
ENVIRO.STUB:ALL
ENVIRO.STUB:NONE
ENVIRO.STUB: *unit name to stub*

This command is used to tell VectorCAST which dependent units it should stub *by implementation* during the environment creation. The syntax for this command is one value per line. If all of the dependent units are to be stubbed, then use ENVIRO.STUB:ALL. If none of the dependent units are to be stubbed, then use ENVIRO.STUB:NONE.

To specify individual units to stub, each unit is named, one per line. By default, units that are not specified by name are not stubbed.

See also the environment variable VCAST_NEVER_STUB_LIST.

ENVIRO.DONT_STUB: *unit name to not stub*

This command is used to tell VectorCAST which units to not stub, that is, of which units to use the “real code.” This command can be combined with ENVIRO.STUB:ALL_BY_PROTOTYPE and ENVIRO.STUB:ALL.

This command was formerly known as ENVIRO.GLOBALS_ONLY, which is deprecated.

ENVIRO.CLASS_OF_INTEREST: *class name*

This command is for use with C++ environments only. Use it to specify the name of a class (inlined or not) in your source code files or header files that you want to appear as testable in the Parameter Tree. The class name is case sensitive. You can have as many lines as needed to list all the classes of interest, with one class name per line. By default, any class that has a member function defined in the UUT source file or an inlined function defined in a header that is included by the UUT source file, is already a class of interest.

ENVIRO.SUPPRESS_STUB: *function-name in unit*

This command suppresses Stub-by-Function instrumentation in the test harness for a particular testable function in an SBF unit under test.

ENVIRO.MAX_VARY_BY_RANGE: *num*

This command is used to tell VectorCAST how many scalars can be varied at one time. The default is 20. The setting is indicated in the **Tools => Options Dialog, Builder Tab**.

ENVIRO.UNIT_COMPILATION_ARGUMENTS: *unit* : *compilation arguments*

This command specifies arguments to be used for an individual unit when compiling it into the environment. Arguments to be used for *all* units are specified in the Tools => Options dialog, C/C++ tab, and are therefore stored in the CCAST_.CFG file.

ENVIRO.ADDITIONAL_TESTABLE_FUNCTION:<*function-name*>

This command is used to specify additional testable functions.

This command is useful in the situation where the configuration options **VCAST_TEST_ALL_INLINES** and /or **VCAST_TEST_ALL_NON_MEMBER_INLINES** are set to false. Setting **VCAST_TEST_ALL_INLINES** to false prevents testing of inline member functions in header files. Setting **VCAST_TEST_ALL_NON_MEMBER_INLINES** to false prevents testing of non-member inline functions in header files.

The **ENVIRO.ADDITIONAL_TESTABLE_FUNCTION** command takes precedence over the **ENVIRO.SUPPRESS_TESTABLE_FUNCTION** command. If **ENVIRO.ADDITIONAL_TESTABLE_FUNCTION** is specified for a function, then it is testable even if **ENVIRO.SUPPRESS_TESTABLE_FUNCTION** is also specified for that function.

ENVIRO.UNIT_PREFIX_USER_CODE
ENVIRO.UNIT_PREFIX_USER_CODE_FILE: *unit*
//user code to be parsed at the beginning of *unit*
ENVIRO.END_UNIT_PREFIX_USER_CODE_FILE:
ENVIRO.END_UNIT_PREFIX_USER_CODE:

These two pairs of tags, one inside the other, delimit user code that is prepended to the beginning of the specified unit. VectorCAST parses the unit prefix user code as if it were at the beginning of *unit*.

ENVIRO.UNIT_APPENDIX_USER_CODE:
ENVIRO.UNIT_APPENDIX_USER_CODE_FILE: *unit*
//user code to be parsed at the end of *unit*
ENVIRO.END_UNIT_APPENDIX_USER_CODE_FILE:
ENVIRO.END_UNIT_APPENDIX_USER_CODE:

These two pairs of tags, one inside the other, delimit user code that is appended to the end of the specified unit. VectorCAST parses the unit appendix user code as if it were at the end of *unit*, before compiling driver prefix user code.

This type of user code is particularly useful if *unit* contains abstract classes. For example, you can #include a concrete subclass of an abstract class defined in *unit*.

ENVIRO.DRIVER_PREFIX_USER_CODE:
ENVIRO.DRIVER_PREFIX_USER_CODE_FILE: *unit*
//user code to be placed at beginning of test harness driver for *unit*
ENVIRO.END_DRIVER_PREFIX_USER_CODE_FILE:
ENVIRO.END_DRIVER_PREFIX_USER_CODE:

These two pairs of tags, one inside the other, delimit user code that is prepended to the beginning of the test harness driver. VectorCAST does not parse this user code, or expand it. It is compiled after unit appendix user code.

This type of user code is particularly useful if *unit* contains a **`#define`** or **`#undef`** that should not be present in the test harness driver. You can change the **`#define`** or **`#undef`** in driver prefix user code so that the driver does not encounter it. For example, you can have **`#define int float`** in *unit*, and put **`#undef int`** in the driver prefix user code, to avoid a compile error in the driver.

ENVIRO.USER_GLOBALS:
ENVIRO.END_USER_GLOBALS:

The ENVIRO.USER_GLOBALS and ENVIRO.END_USER_GLOBALS delimit the lines of the script file to be put into the User Globals File. This command is optional (VectorCAST 4.2+) and if not given, defaults the following standard lines.

This file can be permanently altered by editing the GLOBALS.C file in the DATA directory in the VectorCAST installation directory. The User Globals are also accessible by choosing **Environment => User Code => Edit**, and then clicking the **User Globals** node.

```
/*************************************************************

S0000008.c: This file contains the definitions of variables used in
user code. Preface all variable declarations with VCAST_USER_GLOBALS_
EXTERN to ensure that only one definition of the variable is created
in the test harness.

*************************************************************/

#ifndef VCAST_USER_GLOBALS_EXTERN

#define VCAST_USER_GLOBALS_EXTERN

#endif

#ifdef __cplusplus

extern "C"{

#endif

  VCAST_USER_GLOBALS_EXTERN int VECTORCAST_INT1;

  VCAST_USER_GLOBALS_EXTERN int VECTORCAST_INT2;

  VCAST_USER_GLOBALS_EXTERN int VECTORCAST_INT3;

#ifndef VCAST_NO_FLOAT

  VCAST_USER_GLOBALS_EXTERN float VECTORCAST_FLT1;

#endif

  VCAST_USER_GLOBALS_EXTERN char VECTORCAST_STR1[8];

  VCAST_USER_GLOBALS_EXTERN int VECTORCAST_BUFFER[4];

#ifdef __cplusplus

}

#endif
```

ENVIRO.USER_PARAMETERS:
ENVIRO.END_USER_PARAMETERS:

These commands delimit specifications for custom variables to be used in place of the harness variables which VectorCAST generates to set or capture function parameter or return values. This feature is useful if there is an issue with how VectorCAST defines such harness variables.

The syntax consists of a pair on each line, with the user code tag representing the harness variable followed by the name of your custom variable. For example:

```
ENVIRO.USER_PARAMETERS:
<<manager.Place_Order.Table>> MY_TABLE
ENVIRO.END_USER_PARAMETERS:
```

This specifies that the variable **`MY_TABLE`** be used by the harness for setting or capturing the **`Table`**

parameter of the function **`Place_Order`**.

Note that the specified custom variable must be declared in either the unit or the user code, otherwise a compile error will occur. Also note that the specified custom variable must also be defined somewhere in the harness, otherwise a link error will occur. Defining such variables in unit prefix/appendix user code works in many cases.

User Parameters can be added by selecting **Environment => User Code => Edit** from the Menu Bar and expanding the User Params node. Double-click to open the edit box. Alternatively, User Parameters can be added in the Create New/Update Environment wizard by selecting Step 7 User Code and opening the User Params node.

ENVIRO.USER_CODE_DEPENDENCIES: ENVIRO.END_USER_CODE_DEPENDENCIES:

These commands delimit the dependencies (**`#include`** statements) for environment user code. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Header** section.

```
ENVIRO.USER_CODE_DEPENDENCIES:
          Header section
ENVIRO.END_USER_CODE_DEPENDENCIES:
```

ENVIRO.USER_CODE_OBJECTS: ENVIRO.END_USER_CODE_OBJECTS:

These commands delimit the object definitions (type definitions, globals object, and subprograms) for environment user code. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Data** section.

```
ENVIRO.USER_CODE_OBJECTS:
          Data section
ENVIRO.END_USER_CODE_OBJECTS:
```

ENVIRO.USER_CODE_ONE_SHOT_INIT: ENVIRO.END_USER_CODE_ONE_SHOT_INIT:

These commands delimit the harness initialization environment user code. This user code is processed upon test harness initialization. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Harness Init** section.

```
ENVIRO.USER_CODE_ONE_SHOT_INIT:
          Harness initialization section
ENVIRO.END_USER_CODE_ONE_SHOT_INIT:
```

ENVIRO.USER_CODE_INITIALIZE: ENVIRO.END_USER_CODE_INITIALIZE:

These commands delimit the test case initialization environment user code. This user code is processed immediately before the harness calls the UUT. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Test Case Init** section.

```
ENVIRO.USER_CODE_INITIALIZE:
          Test Case initialization section
ENVIRO.END_USER_CODE_INITIALIZE:
```

ENVIRO.USER_CODE_CAPTURE: ENVIRO.END_USER_CODE_CAPTURE:

These commands delimit the test case termination environment user code. This user code is processed immediately before the harness itself terminates. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Test Case Term** section.

```
ENVIRO.USER_CODE_CAPTURE:
          Test case termination section
ENVIRO.END_USER_CODE_CAPTURE:
```

ENVIRO.USER_CODE_ONE_SHOT_TERM: ENVIRO.END_USER_CODE_ONE_SHOT_TERM:

These commands delimit the test harness termination environment user code. This user code is processed upon harness termination. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Harness Term** section.

```
ENVIRO.USER_CODE_ONE_SHOT_TERM:
          Harness termination section
ENVIRO.END_USER_CODE_ONE_SHOT_TERM:
```

ENVIRO.USER_CODE_STUB_PROCESSING: ENVIRO.END_USER_CODE_STUB_PROCESSING:

These commands delimit the test stub processing environment user code. This user code is common to all stubs, and is processed between Parameter Expected User Code and Parameter Input User Code. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Stub Processing** section.

```
ENVIRO.USER_CODE_STUB_PROCESSING:
          Stub Processing section
ENVIRO.END_USER_CODE_STUB_PROCESSING:
```

ENVIRO.STUB_ENTRY_USER_CODE: ENVIRO.END_STUB_ENTRY_USER_CODE:

These commands delimit the stub subprogram entry configuration user code. It is processed for every stub upon entry, right after any Configure Stubs Beginning User Code and before any Expected Parameter User Code of the stub. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Stub Entry** section.

```
ENVIRO.STUB_ENTRY_USER_CODE:
          Stub subprogram entry configuration code
ENVIRO.END_STUB_ENTRY_USER_CODE:
```

ENVIRO.STUB_EXIT_USER_CODE: ENVIRO.END_STUB_EXIT_USER_CODE:

These commands delimit the stub subprogram exit configuration user code. It is processed for every stub upon exit, right before any Configure Stubs Ending User Code and after any Input Parameter User Code of the stub. They are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Stub Exit** section.

```
ENVIRO.STUB_EXIT_USER_CODE:
          Stub subprogram exit configuration code
ENVIRO.END_STUB_EXIT_USER_CODE:
```

ENVIRO.STUB_USER_CODE_FILE: ENVIRO.END_STUB_ USER_CODE_FILE:

These commands delimit the beginning of stub and end of stub user code for Configure Stubs. There are two sub-commands: one for the beginning of the stub

Repeat for each unit or subprogram that has stub user code to process at the beginning of the stub

and one for the end of stub. You can repeat these any number of times for different units or different subprograms within the same unit.

You can add or modify this user code by choosing **Environment => Configure Stubs => Edit**.

```
ENVIRO.STUB_USER_CODE_FILE:

 BEGINNING_OF_STUB.unit.subprogram
Configure Stubs dialog, Beginning of Stub
END_BEGINNING_OF_STUB.unit.subprogram
```



Note: Repeat these sections for each unit or subprogram that has stub user code to process at the beginning of the stub.

```
END_OF_STUB.unit.subprogram
Configure Stubs dialog, End of Stub
END_END_OF_STUB.unit.subprogram
```



Note: Repeat these sections for other units and/or subprograms at the end of stub.

```
ENVIRO.END_STUB_USER_CODE_FILE:
```

ENVIRO.STUB_DEPEND_USER_CODE_FILE: ENVIRO.END_STUB_DEPEND_USER_CODE_FILE:

These commands delimit the stub dependencies user code for Configure Stubs. You can add or modify this user code by choosing **Environment => Configure Stubs => Edit**, and opening the **Stub Dependencies** cell.

For each stubbed unit for which you want to write stub dependency user code, you enclose that unit's name with `BEGIN_UC` and `END_UC`.



Note: Repeat the following for each unit that has a dependency

```
ENVIRO.STUB_DEPEND_USER_CODE_FILE:
BEGIN_UC:
unit
          Configure Stubs dialog, Stub Dependencies column
END_UC:
ENVIRO.END_STUB_DEPEND_USER_CODE_FILE:
```

ENVIRO.ADDITIONAL_UNIT_BODIES ENVIRO.END_ADDITIONAL_UNIT_BODIES

These two commands delimit the implementation portion of additional units that you want added to the test harness, as though it were another source file (unit). For example, you may have a print utility, a timing routine, or a data setup routine that is not part of your deliverable, but necessary for unit testing. The code here is not parsed by VectorCAST during environment building. These tags are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **Additional Unit Bodies** section.

```
ENVIRO.ADDITIONAL_UNIT_BODIES:
          Implementation for additional units
```

```
ENVIRO.END_ADDITIONAL_UNIT_BODIES:
```

ENVIRO.USER_CODE_TIMER_START:
ENVIRO.END_USER_CODE_TIMER_START:

Code to start timer after the test harness performs data setup, but just prior to calling the UUT. These tags are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **UUT Timer Start** section.

```
ENVIRO.USER_CODE_TIMER_START:
          Code to start timer
ENVIRO.END_USER_CODE_TIMER_START:
```

ENVIRO.USER_CODE_TIMER_STOP:
ENVIRO.END_USER_CODE_TIMER_STOP:

Code to stop timer immediately after returning from the call to the UUT. These tags are accessible in the Create New/Update Environment wizard, Step 5 User Code, User Code node, or by choosing **Environment => User Code => Edit**, and opening the **UUT Timer Stop** section.

```
ENVIRO.USER_CODE_TIMER_STOP:
          Code to stop timer
ENVIRO.END_USER_CODE_TIMER_STOP:
```

ENVIRO.INDUSTRY_MODE

Code to set the Industry Mode. Industry Modes are set as follows for each Industry Mode type:

```
ENVIRO.INDUSTRY_MODE:DO-178 B/C (Avionics)
ENVIRO.INDUSTRY_MODE:ISO-26262 (Automotive)
ENVIRO.INDUSTRY_MODE:IEC-61508 (Industrial)
ENVIRO.INDUSTRY_MODE:EN-50128 (Railway)
ENVIRO.INDUSTRY_MODE:IEC-62304 (Medical)
ENVIRO.INDUSTRY_MODE:Default
```

ENVIRO.END

Required. This command marks the end of the script data for a particular environment. It must be the last line in the environment script. When this command is encountered, VectorCAST attempts to build an environment using the values read since ENVIRO.NEW was read.

Sample Environment Script

```
-- VectorCAST Environment Script
```

```
-- Copyright 2017 Vector Informatik, GmbH
--
-- This environment script refers to three units:
-- a, which is the UUT,
-- b, a stubbed (by implementation) dependent of a
-- and having subprogram b(),
-- and c, a stubbed dependent of a and having subprogram c().
ENVIRO.NEW
ENVIRO.NAME: TEST
ENVIRO.UUT: a
ENVIRO.WHITE_BOX: NO
ENVIRO.STUB: b
ENVIRO.STUB: c
ENVIRO.STUB_DEPEND_USER_CODE_FILE:
BEGIN_Uc:
b
// Configure Stubs | Stub Dependency, Unit b
END_Uc:
BEGIN_Uc:
c
// Configure Stubs | Stub Dependency, Unit c
END_Uc:
ENVIRO.END_STUB_DEPEND_USER_CODE_FILE:
ENVIRO.STUB_USER_CODE_FILE:
BEGINNING_OF_STUB.b.b
printf( " Configure Stubs | Beginning of Stub for Unit b, Subprogram b {\n" );
END_BEGINNING_OF_STUB.b.b
END_OF_STUB.b.b
printf( " } Configure Stubs | End of Stub for Unit b, Subprogram b\n\n" );
END_END_OF_STUB.b.b
BEGINNING_OF_STUB.c.c
printf( " Configure Stubs | Beginning of Stub for Unit c, Subprogram c {\n" );
END_BEGINNING_OF_STUB.c.c
END_OF_STUB.c.c
printf( " } Configure Stubs | End of Stub for Unit c, Subprogram c\n\n" );
END_END_OF_STUB.c.c
ENVIRO.END_STUB_USER_CODE_FILE:
ENVIRO.USER_CODE_CAPTURE:
printf( " } Environment User Code | Test Case Term\n\n" );
ENVIRO.END_USER_CODE_CAPTURE:
ENVIRO.STUB_ENTRY_USER_CODE:
printf(" stub entry user code\n");
ENVIRO.END_STUB_ENTRY_USER_CODE:
ENVIRO.STUB_EXIT_USER_CODE:
printf(" stub exit user code\n");
ENVIRO.END_STUB_EXIT_USER_CODE:
ENVIRO.USER_CODE_DEPENDENCIES:
// Environment User Code | Header
```

```
#include <stdio.h>
ENVIRO.END_USER_CODE_DEPENDENCIES:
ENVIRO.USER_CODE_INITIALIZE:
printf( " Environment User Code | Test Case Init {\n\n" );
ENVIRO.END_USER_CODE_INITIALIZE:
ENVIRO.USER_CODE_OBJECTS:
// Environment User Code | Data
ENVIRO.END_USER_CODE_OBJECTS:
ENVIRO.USER_CODE_ONE_SHOT_INIT:
printf( "Environment User Code | Harness Init {\n\n" );
ENVIRO.END_USER_CODE_ONE_SHOT_INIT:
ENVIRO.USER_CODE_ONE_SHOT_TERM:
printf( "} Environment User Code | Harness Term\n" );
ENVIRO.END_USER_CODE_ONE_SHOT_TERM:
ENVIRO.USER_CODE_STUB_PROCESSING:
printf( " Environment User Code | Stub Processing\n" );
ENVIRO.END_USER_CODE_STUB_PROCESSING:
ENVIRO.USER_CODE_TIMER_START:
printf(" timer start...\n");
ENVIRO.END_USER_CODE_TIMER_START:
ENVIRO.USER_CODE_TIMER_STOP:
printf(" timer end...\n");
ENVIRO.END_USER_CODE_TIMER_STOP:
ENVIRO.SEARCH_LIST: .
ENVIRO.END
```

APPENDIX C: TEST SCRIPT LANGUAGE

The scripting language corresponds to the data required to build a test case using the GUI. Each command is entered on a single line. The Script reader is case sensitive, conforming to the normal case sensitivity rules of the language.

Note the following conventions:

- All commands start with "TEST." to indicate that a test command follows.
- All commands that require data must use a colon (:) to indicate the start of the associated data.
- Comments may be inserted, and are indicated by inserting the comment delimiter "--" as the first item on the line.

Script Commands

The following is a list of the valid script commands with a description of each command.

TEST.NEW | TEST.ADD | TEST.REPLACE | TEST.REMOVE:

Required. Tell VectorCAST that the test case definition that follows defines a new test case, adds values to an existing test case, replaces an existing test case, or removes an existing test case. You can only use one (NEW, ADD, REPLACE or REMOVE) for each test case definition.

TEST.NEW is used to tell VectorCAST that the following commands are to start the construction of a new test case.

TEST.ADD is used to add additional input or expected data values to an existing test case.

TEST.REPLACE is used to delete an existing test case with the same name and replace it, using the data that follows in the test script. The script log informs you that the test case is being replaced.

TEST.REMOVE is used to remove an existing test case.

TEST.ATTRIBUTES: <data name>:<attribute expression>

Attributes for scalar data. These are specific to attributes contained in the test case that are unrelated to input or expected values. For example, you may right-click on the input or expected value field of a particular parameter and set the base for the value to be hex or octal. This attribute gets stored in the test script so that the same attribute is used when it is re-imported into the same or a different environment.

TEST.AUTOMATIC_FINALIZATION

Specify a <<COMPOUND>> or <<INIT>> test case as an Automatic Finalization test case. It sets a flag indicating a <<COMPOUND>> or <<INIT>> test case should be run automatically after other test cases. A single <<COMPOUND>> or <<INIT>> test case can be both Automatic Initialization and Automatic Finalization. An Automatic Finalization test case can be executed on its own, but it cannot be run before or after itself. There can only be one Automatic Finalization test per test environment. When exporting a test script, the Automatic Finalization test cases are also exported.

TEST.AUTOMATIC_INITIALIZATION

Specify a <<COMPOUND>> or <<INIT>>test case as an Automatic Initialization test case. It sets a flag indicating a <<COMPOUND>> or <<INIT>> test case should be run automatically before other test cases. A single <<COMPOUND>> or <<INIT>> test case can be both Automatic Initialization and Automatic Finalization. An Automatic Initialization test case can be executed on its own, but it cannot be run before or after itself. There can only be one Automatic Initialization test per test environment. When exporting a test script, the Automatic Initialization test cases are also exported.

TEST.BASIS_PATH: *x* of *y*

When basis path test cases are generated automatically (using **Tools => Basis Path => Generate Tests**), and a test script is exported for these test cases, the TEST.BASIS_PATH command is used to identify them as basis path test cases. The text "*x* of *y*" indicates that particular test case out of the total number of basis path test cases generated for that subprogram.

TEST.COMPOUND_ONLY

This command is used to tell VectorCAST that the test case is to be executed only from within a compound test case. During Batch Execute All, the test case is not executed unless it is part of a compound.

TEST.CSV_COLUMN_INFO: *col*, TEST.VALUE... | TEST.EXPECTED...

This command is used in a CSV MAP test case. It maps a column, *col*, in the .csv file to a particular parameter's Input or Expected Value. Use either TEST.VALUE:*unit.subprogram.param* or TEST.EXPECTED:*unit.subprogram.param* or a suitable identifier for the parameter.

```
-- Unit: line
-- Subprogram: findY
-- Test Case: (MAP)FINDY.001
TEST.UNIT:line
TEST.SUBPROGRAM:findY
TEST.NEW
TEST.NAME:(MAP)FINDY.001
TEST.CSV_FILENAME:C:/VCAST/Examples/c/csv_example/table.csv
TEST.CSV_DELIMITER:COMMA
TEST.CSV_HEADER_ROWS: 1
TEST.CSV_ROWS_PER_TESTCASE: 1
TEST.CSV_COLUMN_INFO: 3, TEST.VALUE:line.findY.x
TEST.CSV_COLUMN_INFO: 2, TEST.VALUE:line.findY.m
TEST.CSV_COLUMN_INFO: 4, TEST.VALUE:line.findY.b
TEST.CSV_COLUMN_INFO: 1, TEST.EXPECTED:line.findY.return
TEST.END
```

TEST.CSV_DELIMITER: TAB | COMMA

This command is used in a CSV MAP test case. It specifies whether a comma character or tab character is used as the delimiter in the .csv file. Use the literal text “TAB” or “COMMA” to specify.

TEST.CSV_FILENAME: *path to .csv or .tab file*

This command is used in a CSV MAP test case. It is used to tell VectorCAST the path to the .csv or .tab file that will be used to create a CSV MAP test case in the environment.

TEST.CSV_HEADER_ROWS: *num*

This command is used in a CSV MAP test case. It specifies how many rows at the beginning of the .csv or .tab file are considered headers. Any row following *num* row is considered data.

TEST.CSV_NAME_COL:

This command is used in a CSV MAP test case. It specifies the number of the column containing the test case name. Enter a value of "0" if you do not specify the test case name in the .csv file.

TEST.CSV_NOTES_COL:

This command is used in a CSV MAP test case. It specifies the number of the column containing test case notes. Enter a value of "0" if you do not specify Notes in the .csv file.

TEST.CSV_ROWS_PER_TESTCASE: <num>

This command is used in a CSV MAP test case. It specifies the number of rows of data that should be combined into one test case.

TEST.END

This command is used to indicate the end of a test case and causes the previously entered test case data to be stored into the VectorCAST environment.

TEST.EXPECTED: *identifier* : *value*

This command is used to provide expected values for parameters and global objects.

See the section "Setting Input and Expected Values" on page 683 for a full description of the <value> syntax.

TEST.EXPECTED_GLOBALS_USER_CODE: *global_value*
<enter user code here>
TEST.END_EXPECTED_GLOBALS_USER_CODE:

Together, these commands delimit the expected user code for global values. This user code is evaluated when other global values are evaluated, so it is possible for this user code to be evaluated when a test ends due to the event limit being reached.

```
TEST.EXPECTED_GLOBALS_USER_CODE:USER_GLOBALS_VCAST.<<GLOBAL>>.VECTORCAST_
INT2
{{ <<USER_GLOBALS_VCAST.<<GLOBAL>>.VECTORCAST_INT2>> == ( 2 ) }}
TEST.END_EXPECTED_GLOBALS_USER_CODE:
```

TEST.EXPECTED_USER_CODE: <<testcase>>
<enter user code here>
TEST.END_EXPECTED_USER_CODE:

Together, these commands delimit the test case's expected user code when "<<testcase>>" is used (without the quotes), and is not associated with any single parameter.

TEST.EXPECTED_USER_CODE: *unit.subprogram.parameter*
<enter user code here>
TEST_END_EXPECTED_USER_CODE:

Together, these commands delimit a parameter's expected user code in the UUT. Specify the parameter to which the user code applies using the notation *unit*.*subprogram*.*parameter*.

TEST.FLOATING_POINT_TOLERANCE: *number*%
(or)
TEST.FLOATING_POINT_TOLERANCE: *decimal number*

Using one of these commands enables you to set a floating-point tolerance at the test case level. You can set the floating-point tolerance at the environment level, the test case level, and the parameter level. Each level overrides the previous one. Therefore, you can set an environment-level tolerance (**Tools => Options dialog, Execute tab**), specify a test case tolerance for a particular test case, and within that test case, specify a tolerance for one parameter.

In the test script, you can specify the floating-point tolerance as a percent, or as the decimal equivalent. For example, to set the test case floating-point tolerance to 1.25%, use either of the following notations:

```
TEST.FLOATING_POINT_TOLERANCE: 1.25%
TEST.FLOATING_POINT_TOLERANCE: 0.0125
```

Upon test script import, the text "Test case floating point tolerance is: *number*%" is displayed in the log.

Note: For very small tolerances, the floating point representation of the value may not be exact due to the resolution of floating point numbers; in this case, the floating point tolerance may show more digits than the user expects. For example, a tolerance of 0.000000001 may actually be displayed as 0.00000000099999.

TEST.FLOW:
<unit> . <subprogram>
TEST.END_FLOW:

This is the expected sequence of subprograms called in the Unit Under Test and stubbed units. For each <unit>.<subprogram> that is called, there is a line in this section. For example, if the UUT is unit A, the test harness calls unit A's subprogram A() first, then unit B's subprogram B(), then unit C's subprogram C(), then the UUT returns control to the test harness:

```
TEST.FLOW
  A.A
  B.B
  C.C
  A.A
TEST.END_FLOW
```

TEST.IMPORT_FAILED

VectorCAST adds this line to an exported test script file if the option "Strict test case importing" is on, and there was an error when the script file was originally imported.

See also "Strict Test Case Importing" on page 442 for more information.

TEST.MCDC_BASIS_PATH: *x* of *y*

When MC/DC Basis path test cases are generated automatically (using **Tools => MC/DC=> Test Case Analysis => Generate Test Scripts**) and a test script is exported for these test cases, the TEST.MCDC_BASIS_PATH command is used to identify them as MC/DC test cases. Each test covers one row of one condition's MC/DC Equivalence Table. The text "**x of y**" indicates that particular test case out of the total number of test cases generated for that subprogram.

TEST.NAME: *name*

This command is used to provide a name for the test case. The *name* is case-sensitive, if you are using it with TEST.ADD. This command is not required. If a TEST.NAME command is not provided, VectorCAST will generate an automatic name as it does when creating test cases in the Test Case Editor. If this command is used, the *name* provided must be unique. Otherwise, it will be ignored and an automatic name will be assigned when the test script is processed.

TEST.NOTES:
<enter text here>
TEST.END_NOTES:

Together, these commands delimit a list of requirements or notes for the test case. Everything that exists between these delimiters will be included in the requirements section of the test case. Note that each of these commands must exist, by itself, on separate lines. Anything that follows these commands on the same line will be ignored. If you have the Report option “Verbose management report” turned on, then the first 89 characters in the Notes section are displayed in the Test Case Management Report.

TEST.REQUIREMENT_KEY: *requirement_key*

When an environment has imported requirements through the Requirements Gateway, a requirement can be associated with a testcase. Upon test script export, this command is used to identify the requirement key associated with each test case. The requirement key is imported, and should not be changed in the test script. If a test case has more than one requirement, then multiple commands are used, with one key per line. (VectorCAST with Requirements Gateway license)

TEST.SCRIPT_FEATURE: *special feature*

The TEST.SCRIPT_FEATURE line is automatically added by VectorCAST on test script export and should not need to be modified by the user. These commands tell VectorCAST if any special features are present in the environment which the test script importer should support.

TEST.SLOT: *slot num, unit, subprogram name, number of iterations, testcase name, print*

The command TEST.SLOT provides information on a slot in a compound test. It should be present when the TEST.SUBPROGRAM line specifies <<COMPOUND>>.

- *slot* can be any number. The test cases in the compound are added in the order they are listed in the TEST.SLOT command. *slot* is a quoted string. Example: “1”
- *unit* is a string identifying the unit under test. The name is case sensitive. *unit* is a quoted string. Example: “file_io”. If the slot is a nested compound slot, *unit* is specified as <<COMPOUND>> in quotes.
- *subprogram name* is a string identifying the subprogram in which this test case appears. *subprogram name* is a quoted string. Example: “CreateFile”. If the slot is a nested compound slot, *subprogram name* is specified as <<COMPOUND>> in quotes.
- *testcase name* is the name of the test case in the slot. *testcase name* is a quoted string. Example: “CREATEFILE.001”
- *number of iterations* is an integer specifying how many times the test case should repeat. To “turn off” a test case while keeping it in the compound, set the iterations to 0. Doing so causes the test case to be skipped during execution of the compound, while remaining in the compound test case. Note that setting iterations to 0 causes the test case to be skipped and not executed, and is not the same as setting PRINT=FALSE, which still executes the test case but does not gather the execution data. *number of iterations* is a quoted string. Example: “1”

- *print* is either a TRUE or FALSE value and is used to prevent the reporting of data for a slot. The default value is TRUE and if no value is specified, the value TRUE is assumed. The value FALSE prevents the harness from reporting on executing data. The test is executed, but the test harness does not gather any data for the execution report. Note that PRINT=FALSE is not the same as setting *iterations* to 0 for the slot, where the test case will be skipped and not executed. If the value is FALSE, the test case in the slot is required to have no expected values. Example: "PRINT=FALSE".

For example, the following test script specifies a compound test case which has three slots with the test cases CREATEFILE.001, WRITELINE.001, and CLOSEFILE.001 included in it. Note that the script for the test cases must be defined above the definition for the compound.



Note: Compound test case below doesn't get a TEST.UNIT line itself.

```
TEST.SUBPROGRAM:<<COMPOUND>>
TEST.NEW
TEST.NAME:<<COMPOUND>>.001
TEST.REQUIREMENT_KEY:000000a1/11
TEST.SLOT: "1", "file_io", "CreateFile", "1", "CREATEFILE.001"
TEST.SLOT: "2", "file_io", "WriteLine", "1", "WRITELINE.001"
TEST.SLOT: "3", "file_io", "CloseFile", "1", "CLOSEFILE.001"
TEST.END
```

TEST.STUB: *unit.subprogram*

This command is used to indicate that the ***subprogram*** in ***unit*** is stubbed when the test case is executed. The environment must be built with ***unit*** as a "stub by function" unit, not a regular UUT. (Use ENVIRO.STUB_BY_FUNCTION:<unit> in place of ENVIRO.UUT:<unit> in the environment script.) Subsequent TEST.VALUE and TEST.EXPECTED commands provide input and expected values for the parameters in the stubbed subprogram, as usual. (VectorCAST version 5.0+)

TEST.STUB_EXP_USER_CODE: *unit.subprogram.parameter*
<enter user code here>
TEST.END_STUB_EXP_USER_CODE:

Together, these commands delimit a parameter's expected user code in a stub.

TEST.STUB_VAL_USER_CODE: *unit.subprogram.parameter*
<enter user code here>
TEST.END_STUB_VAL_USER_CODE:

Together, these commands delimit a parameter's input user code in a stub.

TEST.SUBPROGRAM: *subprogram name*

Required. This command is used to indicate a subprogram for which test cases will be built. The

subprogram name corresponds to any subprogram of the Unit Under Test specified in the TEST.UNIT line. If several test cases are to be built for the same subprogram this command does not have to be repeated until a different subprogram is desired. To create an <<INIT>> test case, use the subprogram name <<INIT>>.

TEST.UNIT: *unit name*

Required. This command identifies the unit that the test case is for. The name is not case sensitive. This command can be used once to indicate that all of the following test case definitions are for a single unit, until another TEST.UNIT line appears. This line is not required for <<INIT>> or <<COMPOUND>> test cases.

TEST.VALUE: *identifier* : *value*

This command is the most important of the test case scripting commands. It is used to provide the individual input values for parameters and global objects.

See the section "Setting Input and Expected Values" on page 683 for a full description of the <value> syntax.

TEST.VALUE_USER_CODE: <<testcase>>
<enter user code here>
TEST.END_VALUE_USER_CODE:

Together, these commands delimit the test case's input user code, when "<<testcase>>" is used (without the quotes), and is not associated with any single parameter.

TEST.VALUE_USER_CODE: *unit.subprogram.parameter*
<enter user code here>
TEST.END_VALUE_USER_CODE:

Together, these commands delimit a parameter's input user code in the UUT. Specify the parameter to which the user code applies using the notation *unit*.*subprogram*.*parameter*.

In this example, a unit under test has three subprograms:

- CreateFile – returns a file pointer to the specified file name
- WriteLine – write a string to the file pointed by the file pointer passed in
- CloseFile – close the file pointed to by the file pointer passed in

To test these subprograms in VectorCAST, you would create a test case for each subprogram and then call the three test cases from a compound test case. The problem is there is no static way for VectorCAST to pass the file pointer created by CreateFile to the other two subprograms. With parameter user code, you have the ability to set data dynamically. In this example, you want to pass the file pointer returned from CreateFile as an input into WriteLine and CloseFile. With user code, you use the name of the file pointer returned by CreateFile as a parameter for the other two files. The following test case script shows how to do that:

```
-- Unit(s) Under Test: file_io
--
-- Script Features
TEST.SCRIPT_FEATURE:CPP_CLASS_OBJECT_REVISION
TEST.SCRIPT_FEATURE:MULTIPLE_UUT_SUPPORT
--
-- Unit: file_io
-- Subprogram: CreateFile
-- Test Case: CREATEFILE.001
TEST.UNIT:file_io
TEST.SUBPROGRAM:CreateFile
TEST.NEW
TEST.NAME:CREATEFILE.001
TEST.NOTES:
TEST.END_NOTES:
TEST.VALUE:file_io.CreateFile.filename:<<malloc 9>>
TEST.VALUE:file_io.CreateFile.filename:"TEMP.TXT"
TEST.VALUE:file_io.CreateFile.return:<<malloc 1>>
TEST.EXPECTED_USER_CODE:file_io.CreateFile.return
{{ <<file_io.CreateFile.return>> != NULL }}
TEST.END_EXPECTED_USER_CODE:
TEST.END

-- Subprogram: WriteLine
-- Test Case: WRITELINE.001
TEST.UNIT:file_io
TEST.SUBPROGRAM:WriteLine
TEST.NEW
TEST.NAME:WRITELINE.001
TEST.NOTES:
TEST.END_NOTES:
TEST.VALUE:file_io.WriteLine.outputLine:<<malloc 13>>
TEST.VALUE:file_io.WriteLine.outputLine:"Hello, World"
TEST.VALUE_USER_CODE:file_io.WriteLine.fp
<<file_io.WriteLine.*fp>> = ( <<file_io.CreateFile.return>> );
TEST.END_VALUE_USER_CODE:
TEST.END
-- Subprogram: CloseFile
-- Test Case: CLOSEFILE.001
TEST.UNIT:file_io
TEST.SUBPROGRAM:CloseFile
TEST.NEW
TEST.NAME:CLOSEFILE.001
TEST.NOTES:
TEST.END_NOTES:
TEST.VALUE_USER_CODE:file_io.CloseFile.fp
<<file_io.CloseFile.fp>> = ( <<file_io.CreateFile.return>>);
TEST.END_VALUE_USER_CODE:
```

```
TEST.END
-- COMPOUND TESTS
TEST.SUBPROGRAM:<<COMPOUND>>
TEST.NEW
TEST.NAME:<<COMPOUND>>.001
TEST.NOTES:
TEST.END_NOTES:
TEST.SLOT: "1", "file_io", "CreateFile", "1", "CREATEFILE.001"
TEST.SLOT: "2", "file_io", "WriteLine", "1", "WRITELINE.001"
TEST.SLOT: "3", "file_io", "CloseFile", "1", "CLOSEFILE.001"
TEST.END
--
```

The bold lines indicate the scripting language used to set the parameter "*fp" to be the return value from the call to CreateFile. You can also put any other executable code between the TEST.VALUE_USER_CODE and TEST.END_VALUE_USER_CODE delimiters.

TEST.VARIANT_LOGIC_NAME: logic name

This command is used to indicate which logic name in the environment's Variant Logics File is to be assigned to the test case. The status of the logic name determines whether the test is runnable.

If the logic name specified evaluates to False in the environment, the test is not runnable. If the logic name specified evaluates to True in the environment, the test is runnable. If there is no VARIANT_LOGIC_NAME, the test will always be runnable.

For information on how logic is evaluated, see "Control Test Case Execution Using Variant Logic" on page 400.

The environment's Variant Logics File can be found either by selecting **Environment => Update Environment..** from the Menu Bar and then selecting **Build Options** in the Update Environment wizard, or by performing Environment Script Create in the CLI.

Setting Input and Expected Values

Most of the commands in a test script will be TEST.VALUE or TEST.EXPECTED commands. This section explains the syntax of these commands, and provides examples intended to cover as many combinations as practical.

All test value commands have three fields. The first field is the test command and is always set to "TEST.VALUE:" or "TEST.EXPECTED". The second field is called the *identifier* field and the third field is called the *value* field. Each of the three fields is separated by colons.

The *identifier* field consists of the unit name, a dot, the subprogram name, a dot, and the parameter name. For example: manager.Place_Order.Seat.



Note: VectorCAST is not sensitive to spaces around the Value field. The colon marks the start of the value and the end-of-line marker marks the end of the value.

Numeric Types

With numeric types, the value field is simply the number to be assigned. For example, the test value commands to set values for the two numeric parameters of subprogram Place_Order would be as follows:

```
TEST.VALUE:manager.Place_Order.Table: 3
TEST.VALUE:manager.Place_Order.Seat: 1
```



Note: When working with numeric parameters you may use based notation or exponential notation as you would when building test cases in the GUI.

Structure Types

For structure types, the Identifier field is extended using the standard C syntax of referring to the fields by their names with a '.' separating the field name from the rest of the identifier.

```
TEST.VALUE:manager.Place_Order.Order.Salad: CAESAR
TEST.VALUE:manager.Place_Order.Order.Entree: STEAK
TEST.VALUE:manager.Place_Order.Order.Dessert: CAKE
```

Enumeration Types

With enumeration types, the value field will be the enumeral that you are selecting. In the previous example you may have noticed that the fields were all enumeration types. In this case the value field of the commands are the enumerals.



Note: Out of Range enumeral values are specified with test case scripting by entering a number in place of the enumeral.

Character and String Types

When setting the value field for a character or string, the standard C character and string delimiters (' and ") with the following exceptions:

- You must use the character delimiter if the character to be entered is itself the delimiter character (apostrophe) or a blank character.
- You must use the string delimiter if the string contains blank characters or contains string delimiters.

The following examples illustrate some of the variations:

```
TEST.VALUE:database.Get_Table_Record.Data.Designator: d
TEST.VALUE:database.Get_Table_Record.Data.Designator: ' '
TEST.VALUE:database.Get_Table_Record.Data.Designator: '''
TEST.VALUE:database.Get_Table_Record.Data.Wait_Person: Tim
TEST.VALUE:database.Get_Table_Record.Data.Wait_Person: ""Me""
TEST.VALUE:database.Get_Table_Record.Data.Wait_Person: "J Smith"
```

Note: As you would expect, the case of any character or string value is maintained.

Additionally you may enter non-printable characters by using the standard "C" syntax of \#. For example: \0 for null or \10 for LF.

Pointer and Array Types

The following are example script commands for:

Constrained Array Types:

```
TEST.VALUE:file.test.array_param[2] : 2
```

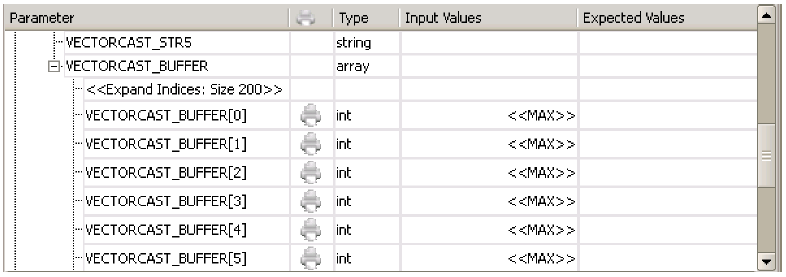
Unconstrained Arrays:

```
TEST.VALUE:file.test.array_param2[]: <<malloc 2>>
TEST.VALUE:file.test.array_param2[1]: 3
```

Pointers

```
TEST.VALUE:file.test.*ptr_param[0]: 12
```

When multiple conescutive array elements have the same value, a short hand notation in exported scripts may be used, making them easier to read. In the figure below, the VECTORCAST_BUFFER of global values has been expanded to display all 200 of its elements. Then they were filled with the <<MAX>> value as an input value. When this test case is exported, the repeated data in the array is shown in shorthand notation VECTORCAST_BUFFER[0..199]:<<MAX>> to make it easier to read.



Parameter	Type	Input Values	Expected Values
VECTORCAST_STR5	string		
VECTORCAST_BUFFER	array		
<<Expand Indices: Size 200>>			
VECTORCAST_BUFFER[0]	int	<<MAX>>	
VECTORCAST_BUFFER[1]	int	<<MAX>>	
VECTORCAST_BUFFER[2]	int	<<MAX>>	
VECTORCAST_BUFFER[3]	int	<<MAX>>	
VECTORCAST_BUFFER[4]	int	<<MAX>>	
VECTORCAST_BUFFER[5]	int	<<MAX>>	

```
-- Test Case Script
TEST.SUBPROGRAM:Update_Table_Record
TEST.NEW
TEST.NAME:UPDATE_TABLE_RECORD.001
TEST.NOTES:
TEST.END_NOTES:
TEST.VALUE:USER_GLOBALS_VCAST.<<GLOBAL>>.VECTORCAST_BUFFER[0..199]:<<MAX>>
```

```
TEST.END
--
```

If we change one element of the array to have a different value, say, VECTORCAST_BUFFER[4] is <<MIN>> now, instead of <<MAX>>, then a range of elements is shown in the shorthand notation, as shown below.

```
TEST.VALUE:USER_GLOBALS_VCAST.<<GLOBAL>>.VECTORCAST_BUFFER[0..3]:<<MAX>>
TEST.VALUE:USER_GLOBALS_VCAST.<<GLOBAL>>.VECTORCAST_BUFFER[4]:<<MIN>>
TEST.VALUE:USER_GLOBALS_VCAST.<<GLOBAL>>.VECTORCAST_BUFFER[5..199]:<<MAX>>
```

Structure Types

For structure types, the Identifier field is extended using the standard C syntax of referring to the fields by their names with a '.' separating the field name from the rest of the identifier, as in the following examples.

```
    TEST.VALUE:manager.Place_Order.Order.Salad: CAESAR
    TEST.VALUE:manager.Place_Order.Order.Entree: STEAK
    TEST.VALUE:manager.Place_Order.Order.Dessert: CAKE
```

Function Return Parameters

Up to this point we have been dealing with examples where we are setting the values of subprogram parameters. In the case of function return parameters, all of the same syntax applies, you simply use the keyword "RETURN" in place of the parameter name as in the following:

```
TEST.VALUE:manager.Get_Check_Total.RETURN: 23.0
```

Global Objects

To set the value of global objects rather than subprogram parameters, the basic syntax is UNIT.<<GLOBAL>>.OBJECT. All of the syntax related to the specific type of the object is the same as it is for subprogram parameters. The following example uses the same types from the Basic Tutorial, but we assume that there is a global object in unit manager called ORDER_OBJECT.

```
TEST.VALUE:manager.<<GLOBAL>>.ORDER_OBJECT.ENTREE: STEAK
```

Test Case Options

These options can be set in the Options tab for a test case. By default, these options are not specified (set to false).

Min, Mid, or Max Values

If you wish to generate a Min, Mid, or Max test case from a script, you only need to provide the special name **<<ALL_MIN>>**, **<<ALL_MID>>** or **<<ALL_MAX>>** after TEST.VALUE or TEST.EXPECTED, as in the following example:

```
TEST.NEW
TEST.NAME: PLACE_ORDER_MIN.001
TEST.VALUE:<<ALL_MIN>>
TEST.END
```

Range of Values for Input Data

During normal test case generation, individual scalar values are specified as input to the UUT or as output from stubs of dependent units. VectorCAST also provides a mechanism that enables you to define a single test case, which will test all values within a specified range of data. This range testing is useful in cases where it is desirable to exhaustively exercise a unit to verify that there is never an abnormal termination or that the routine always returns a legal value regardless of the input data.

To accommodate this type of testing, VectorCAST enables range expressions to be used in place of individual scalar values for any parameter. This is accomplished with the following script syntax:

```
vary from: <value> to: <value> by: <iteration value>
```

A simple example of using range expressions would be in the testing of trig functions.

Consider the following function prototypes that might be in math.c:

```
  float sine (float angle);
  float atan (float angle);
```

Using the range testing capability you could build a single test case that would exercise the sine function in the first quadrant every one tenth of a degree. The test script to accomplish this is shown here:

```
TEST.SUBPROGRAM:sine
TEST.NEW
TEST.VALUE: math.sine.angle:vary from: 0.0 to: 90.0 by: 0.1
TEST.EXPECTED: math.sine.return:0.0..1.0
```



Note: The range values defined allow you to loop forward or backward through the specified range as in the following examples:

Forward looping

```
TEST.VALUE:math.sine.angle:vary from: 0.0 to: 90.0 by: 0.1
```

Backward looping

```
TEST.VALUE:math.sine.angle:vary from: 90.0 to: 0.0 by: - 0.1
```

When setting up a test case for range testing, you may vary different parameters simultaneously. This means that different range expressions may exist within a single test case. This capability enables

you to vary multiple values simultaneously, or vary one value while holding the others constant. The default for the maximum number of parameters to vary is 20. This value can be modified using the Maximum Varied Parameters option, located on the **Target** tab and the **Harness Size Options** sub-tab of the **Tools =>Options** dialog.

Range of Values for Expected Results

The expected results capability enables the entry of a range of values for any numeric scalar value. When entering a value in a test case script for an expected value, simply use the syntax <min> .. < max> as in the following examples:

```
TEST.EXPECTED:database.UPDATE.VALUE: 257.1 .. 259.9
```

List of Values

When entering a list of values in a test case script, simply use the syntax item1, item2, itemN as in the following examples:

```
TEST.VALUE:manager.Place_Order.Seat:1,2,4
```

If a certain value should be repeated, rather than list it multiple times, just include the repeat count in parentheses before the item.

```
TEST.VALUE:manager.Place_Order.Seat:1,(4)2,4
```

In a list of expected values, if any value is acceptable, use the special notation <<ANY>>. Using <<ANY>> indicates that any value that is compared to that spot in the list is acceptable. <<ANY>> doesn't make a test case pass or fail; it is used as a placeholder in a list of expected values.

```
TEST.VALUE:manager.Place_Order.Seat:1,(4)2,<<ANY>>
```



Note: For a list of values, if the number of values entered is less than the number of calls to the particular subprogram, VectorCAST will use the last parameter value for the remaining calls.

Test Values Spanning Multiple Lines

The continuation character '\' enables TEST.VALUE commands to span many lines. This is useful when a script value line becomes large. The continuation character may be placed either after a value or used to break a value between lines. When VectorCAST creates a test case script it will format every line to fit in eighty columns. However, it is not necessary to format the script file in this way in order for VectorCAST to read it.

Example:

```
TEST.VALUE:manager.Place_Order.Order.BEVERAGE:Wine,NoBeverage,\
Beer,Wine,Soda,MixedDrink,\
MixedDrink,NoBeverage
```



Note: The Script Reader will ignore all white space so you may indent and space the values for maximum readability.

Working with Overloaded Subprograms

When two or more subprograms with the same name exist in the same unit, each instance of the overloaded subprogram will have the parameter type list appended to the subprogram name. The following example has commands to set the value of the parameter VAL for the four instances of the overloaded subprogram P that have parameter types of: integer, character, string and boolean.

```
TEST.VALUE:TEST1.P(int).VAL: 3
TEST.VALUE:TEST1.P(char).VAL:'d'
TEST.VALUE:TEST1.P(char*).VAL:"this is a test"
TEST.VALUE:TEST1.P(bool).VAL:TRUE
```

APPENDIX D: LIMITATIONS AND RESTRICTIONS

All required external information must be provided for the unit under test to be run. That is, if the unit solicits keyboard input during processing, this input must be provided by the tester. Similarly any external input retrieved from any other hardware device should be simulated or actually provided to ensure normal test function.

Maximum number of units (files) allowed in a VectorCAST environment is eight hundred (**800**). This is not a limit on how many files can exist in your application. It is a limit on how many dependent units a particular unit under test has.

Maximum number of subprograms in a unit is **999**.

The maximum number of parameters supported per subprogram is **4095**.

APPENDIX E: ENVIRONMENT VARIABLES

The following environment variables affect the behavior of VectorCAST. To set the value of an environment variable, use the following syntax:

Linux (csh) setenv VCAST_MAX_LINE_LENGTH 7000

Linux sh: export VCAST_MAX_LINE_LENGTH=7000

Windows (DOS): set VCAST_MAX_LINE_LENGTH 7000

Windows: Use **Control Panel => System icon => Advanced system settings => Environment Variables...**

You must set the environment variables *before* starting VectorCAST.

Environment Variables

VCAST_ALTERNATE_REPORT

If this environment variable is defined, then a tab-separated report is generated after test execution, in addition to the regular execution results. Use **`clicast reports alternate`** to extract the report to a file or standard output.

VCAST_DEBUG_FILENAME

When debug logging is enabled, the debug log output location defaults to a file with the name prefix **`debug.log`**. This environment variable overrides the default prefix. To direct debug log output to **`stdout`** instead of a file, set VCAST_DEBUG_FILENAME to "-".

VCAST_DEBUG_LEVELS

Enables all levels of debug logging (normal, trace, exception, syscmd, timing, vresult_trace, sql. Adding "**`-d all`**" to the command line will enable full debug logging. For other applications, set the environment variable VCAST_DEBUG_LEVELS to "**`all`**".

VCAST_DEBUG_SCRIPT

The debug script to use with the Keil compiler.

VCAST_DISABLE_FILE_DELETE

Do not delete temporary files before running the driver.

VCAST_END_WAIT

Specify a time to wait in tenths of seconds for the process to finish after end of execution marker is seen. Default value is 1, or 1/10th second.

VCAST_FORCE_OBJECT_EXTENSION

Force compiler to generate object file with extension specified in **Tools => Options, C/C++ tab, Linker sub-tab** (or C_OBJECT_EXT in the CCAST_.CFG file).

VCAST_GLOBAL_DEFINE_LIST

This environment variable enables you to create a set of defined variables that are global to your project. This list is concatenated with the test environment-specific list that is set using the existing “Defined Variables” option (on the **Tools => Options dialog, C/C++ tab**). The value of this variable should be a space-separated list of defined variables. For example:

```
setenv VCAST_GLOBAL_DEFINE_LIST “is_debug is_ppc”
```

VCAST_IGNORE_COMPILE_ERRORS

Do not stop compiling when an error is encountered. For use if your compiler returns an exit code not equal to zero even when successful.

VCAST_IGNORE_FILE_EXISTS_CHECK_FOR_COVERABLE

Allow coverage even if VectorCAST can’t find the source file.

VCAST_IGNORE_TYPEMARKS

VCAST_IGNORE_TYPEMARKS can be used to stop VectorCAST from building type processing functions for particular types. In the Parameter Tree, the types will require user code to be set. This feature is useful for types from third party libraries that will never be manipulated by the user. Multiple type marks should be separated by commas. For type marks that contain spaces, the entire typemark must be enclosed in quotes. For example:

```
setenv VCAST_IGNORE_TYPEMARKS
```

VCAST_LEGACY_CFG_FILE_INHERITANCE

Read *.CFG files from installation directory and user's home directory before reading from the local directory. If the environment variable VCAST_LEGACY_CFG_FILE_INHERITANCE is set, any options inherited from the installation directory or user's home directory are saved to the local *.CFG file, creating a complete set of options from all sources in one place (the local *.CFG file). Once this is done for each environment, the environment variable no longer needs to be set when running the regression script.

VCAST_MAX_LINE_LENGTH

By default, VectorCAST assumes a maximum source line length of 5000 characters. If any lines in the

translation unit (post-processed file) are greater than 5000 characters, then you may get compile errors when building a whitebox test environment, and when initializing coverage. To overcome this problem, VectorCAST provides an environment variable called VCAST_MAX_LINE_LENGTH to allow a user-defined maximum line length. If this variable is set, VectorCAST will use the value of the variable for the maximum line length. If this value is not set, the default value (5000) will be used.

VCAST_NEVER_STUB_LIST

A comma-separated list of unit names (without the file extension) that should not be stubbed even when the STUB ALL option is selected. This environment variable can also be set to the full path to a file, with one unit per line. This feature is useful when the never-stub list is long, or when sharing the list with others.

VCAST_NO_PARSE

Rebuild environment without reparsing source code.

VCAST_NUM_JOBS

This environment variable sets the number of jobs to run simultaneously when building an environment. On machines with multiple CPU cores, setting this variable to a value greater than 1 should improve environment build performance.

VCAST_QIK_ASSUME_SRC_HAS_NOT_CHANGED

Assume source files have not changed since last QuickParse. This environment variable overrides the option in the Create New Environment wizard, called “Source files have not changed.”

VCAST_RECURSION_DEPTH

Maximum number of levels to recurse when trying to generate basis path data.

VCAST_REGRESSION_COMMAND

An additional command that should be added to regression test shell script or batch file.

See also "To Post-Process the Regression Scripts " on page 228.

VCAST_RPTS_PRETTY_PRINT_HTML

This environment variable has turned into a Report option.

VCAST_RSP_USER_TEMPLATE

User-specified template for running with Visual DSP targets.

VCAST_TORNADO_OLD_SCRIPT

When running vxWorks, do not complain about missing symbols. In VectorCAST version 5.0b, this environment variable was renamed **VCAST_DONT_CHECK_FOR_VXWORKS_UNRESOLVED_SYMBOLS**.

VCAST_WHITEBOX

Force whitebox environment building. This environment variable overrides the setting of the Whitebox option in the **Tools => Options dialog, Builder tab**.

VCV_EDIT_LINE_NUMBER

This environment variable is set when opening files in an external editor. This variable is set to the line number of the selected function. The external editor can then be set to a script command that uses the environment variable VCV_EDIT_LINE_NUMBER to go to a specific line number in the external editor. When VCV_EDIT_LINE_NUMBER is set to 0, the file opens and sets the cursor to the top.

VCV_ENVIRONMENT_DIR

This environment variable is set to the full path to the current environment directory whenever the environment is opened.

This environment variable is available to all commands invoked by VectorCAST from the open environment, including:

- C_PREPROCESS_CMD,
- C_COMPILE_CMD,
- C_LINK_CMD,
- VCAST_CMD_DURING_ENV_OPEN,
- VCAST_CMD_DURING_ENV_CLOSE,
- VCAST_PRE_EXECUTE_CMD, and
- C_EXECUTE_CMD.

VCV_ENVIRONMENT_NAME

This environment variable is set to the name of the environment while the environment is open.

This environment variable is available to all commands invoked by VectorCAST from the open environment, including:

- C_PREPROCESS_CMD,
- C_COMPILE_CMD,
- C_LINK_CMD,
- VCAST_CMD_DURING_ENV_OPEN,
- VCAST_CMD_DURING_ENV_CLOSE,

- VCAST_PRE_EXECUTE_CMD, and
- C_EXECUTE_CMD.

VECTORCAST_DIR

The location of VectorCAST binary executables (installation directory).

VECTOR_LICENSE_FILE

The full path to VectorCAST's vendor-specific FLEXlm license file.

APPENDIX F: CLICAST - COMMAND LINE VECTORCAST

Quick Start

The following is a list of useful commands to get online CLICAST help. Each command is preceded by **$VECTORCAST_DIR/** (Linux) or **%VECTORCAST_DIR%** (Windows).

To see a list of all commands, type:

```
clicast help all
```

To see a list of categories, type:

```
clicast help
```

To see a list of commands in a single category, type:

```
clicast help <category>
```

where **<category>** is one of the following:

- ENvironment (or just “EN”)
- EXecute
- Report
- TEst
- TOol
- Option
- Get_option
- Language
- Cover

To see a list of all targets, type:

```
clicast -x tgt
```

To see a list of all options, type:

```
clicast help options all
```

To see a list of option categories, type:

```
clicast help options
```

To see a list of options in a single category, type:

```
clicast help options <category>
```

where **<category>** is one of the following:

- Language
- Builder
- Execute
- Report
- Target
- Coverage

To find out the value of an option, type:

```
clicast get_option <option>
```

Command Format

CLICAST is invoked using the following syntax:

Linux systems:

```
$VECTORCAST_DIR/clicast -eenv -uunit -ssub -ttestcase command arguments
```

Windows systems:

```
%VECTORCAST_DIR%\clicast /e:env /u:unit /s:sub /t:testcase command arguments
```

where ***env*** is the name of the environment, ***unit*** is the name of a unit under test, ***sub*** is the name of a subprogram, and ***testcase*** is the name of a testcase. If the environment has only one UUT, then **-u unit** is optional.

Throughout this User Guide, the corresponding CLICAST command is presented after each feature or option available in the VectorCAST application. The CLICAST command uses the following format:

```
clicast -lc -e <env> command | option  arguments

Description of the command, option, and arguments.
```

The following conventions are used:

- The vertical bar | stands for “or”, indicating that one of several choices is to be included in the command.
- Square brackets around a parameter indicate that a parameter is optional. For example,

  ```
  -e <env> [-u <unit>]
  ```

  indicates that the environment name must be specified, but the unit name can optionally be

specified on the command line.

> Angle brackets indicate that a source file name or script file name must be substituted. For example,

```
<output file>
```

indicates that an output file name must be provided.

> Square brackets and angle brackets can be combined to indicate that a substitution is optional. For example,

```
[<output file>]
```

indicates that if the optional output filename is not present, standard output is used.

> Capital letters in a command name indicate the minimum that needs to be typed to uniquely identify the command. For example,

```
clicast ENvironment Build <scriptfile>
```

indicates that the abbreviated command **`clicast EN B <scriptfile>`** can be used. To specify <<COMPOUND>> or <<INIT>> as the subprogram or when used in a testcase, enclose the name in quotes, as in :

```
clicast -e TUTORIAL_C -u manager -s "<<COMPOUND>>" -t "<<COMPOUND>>.001" exec
run
```

Console Report Format

The Console Report provides command line output with passes and failures at the iteration level and Expected/Actual value mismatch at the iteration level. After all tests have run, a test summary is shown along with a list of all failing tests.

The clicast-only option VCAST_CONSOLE_REPORT_FORMAT has four levels of output:

> DEFAULT
> LEGACY
> VERBOSE
> QUIET

`clicast -lc VCAST_CONSOLE_REPORT_FORMAT CONSOLE_REPORT_DEFAULT | CONSOLE_REPORT_LEGACY | CONSOLE_REPORT_VERBOSE | CONSOLE_REPORT_QUIET`

By default (CONSOLE_REPORT_DEFAULT), VectorCAST shows command line output with passes and failures at the iteration level, and Expected/Actual value mismatch at the iteration level. After all tests have run, a test summary is shown along with a list of all failing tests. With Legacy output (CONSOLE_REPORT_LEGACY), the previous style of console output is shown. Verbose output(CONSOLE_REPORT_VERBOSE) prints both Default and Legacy Output. When set to quiet (CONSOLE_REPORT_QUIET), minimal test/run information is shown in the console output.

The following sections illustrate the output for the four levels.

Default Output

By default (CONSOLE_REPORT_DEFAULT), VectorCAST shows command line output with passes and failures at the iteration level, and Expected/Actual value mismatch at the iteration level. After all tests have run, a test summary is shown along with a list of all failing tests. Output for testcases failing due to a signal failure (such as segmentation violation, for example) are not currently supported.

This example output shows that PLACE_ORDER.002 failed due to one of the range iterations expecting the CheckTotal to be 17, but it was 18.

```
Opening Environment
Opening Parameter/Global File
Opening Types File
Environment is Open
[  PASS  ] manager.Manager::PlaceOrder - MANAGER::PLACEORDER.001
[  FAIL  ] manager.Manager::PlaceOrder - MANAGER::PLACEORDER.002
[  PASS  ] manager.Manager::PlaceOrder - MANAGER::PLACEORDER.002.rangeIter1
[  FAIL  ] manager.Manager::PlaceOrder - MANAGER::PLACEORDER.002.rangeIter2
[        ]   Expected: CheckTotal = [17]
[        ]   Actual:   CheckTotal = [18]
[========] 2 tests ran.
[ PASSED ] 1 tests.
[        ] the following 1 tests failed:
[ FAILED ]   manager.Manager::PlaceOrder - MANAGER::PLACEORDER.002
[        ]   manager.Manager::PlaceOrder - MANAGER::PLACEORDER.002.rangeIter2
```

Legacy Output

With Legacy output (CONSOLE_REPORT_LEGACY), the previous style of console output is shown and is displayed by default for VectorCAST Project output.

```
Environment is Open
Running all manager.Manager::PlaceOrder test cases
Running: MANAGER::PLACEORDER.001
Preparing Test Data
Running Test Case
Running Test with command: ...\UUT_INTE.EXE
Processing Execution Data
Test Execution Complete
Running: MANAGER::PLACEORDER.002
Preparing Test Data
Running Test Case
Running Test with command: ...\UUT_INTE.EXE
Processing Execution Data
Test Execution Complete
Completed Batch Execution processing
```

Verbose Output

Verbose output (CONSOLE_REPORT_VERBOSE) prints both Default and Legacy output.

```
Environment is Open
Running all manager.Manager::PlaceOrder test cases
Running: MANAGER::PLACEORDER.001
```

```
Preparing Test Data
Running Test Case
Running Test with command: ...\UUT_INTE.EXE
Processing Execution Data
[  PASS  ] manager.Manager::PlaceOrder - MANAGER::PLACEORDER.001
Test Execution Complete
Running: MANAGER::PLACEORDER.002
Preparing Test Data
Running Test Case
Running Test with command: ...\UUT_INTE.EXE
Processing Execution Data
[  FAIL  ] manager.Manager::PlaceOrder - MANAGER::PLACEORDER.002
[  PASS  ] manager.Manager::PlaceOrder - MANAGER::PLACEORDER.002.rangeIter1
[  FAIL  ] manager.Manager::PlaceOrder - MANAGER::PLACEORDER.002.rangeIter2
[        ]   Expected: CheckTotal = [17]
[        ]   Actual:   CheckTotal = [18]
Test Execution Complete
[========] 2 tests ran.
[ PASSED ] 1 tests.
[        ] the following 1 tests failed:
[ FAILED ]   manager.Manager::PlaceOrder - MANAGER::PLACEORDER.002
[        ]   manager.Manager::PlaceOrder - MANAGER::PLACEORDER.002.rangeIter2
 Completed Batch Execution processing
```

Quiet Output

When set to quiet (CONSOLE_REPORT_QUIET), minimal test/run information is shown in the console output.

Compound Test Output

Compound test execution is displayed in a similar manner. The beginning of the compound is listed at the top with no status in the brackets. The end of the compound test execution is listed with the status ("PASS" or "FAIL"). The individual slots in the compound are also listed with a status, but do not add to the number of tests executed, of course.

The following output demonstrates the default display for a compound test with three slots:

- an <<INIT>> test case that compares a global string and is failing
- PLACE_ORDER.001 that passes
- and PLACE_ORDER.002 that fails in the second range iteration

```
Environment is Open
[        ] <<COMPOUND>> - <<COMPOUND>>.001
[  FAIL  ] manager.<<INIT>> - <<INIT>>.001.slot1
[        ] Expected: VECTORCAST_STR1 = [expected string]
[        ] Actual: VECTORCAST_STR1 = [Init]
[  PASS  ] manager.Manager::PlaceOrder - MANAGER::PLACEORDER.001.slot2
[  FAIL  ] manager.Manager::PlaceOrder - MANAGER::PLACEORDER.002.slot3
[  PASS  ] manager.Manager::PlaceOrder - MANAGER::PLACEORDER.002.slot3.rangeIter1
[  FAIL  ] manager.Manager::PlaceOrder - MANAGER::PLACEORDER.002.slot3.rangeIter2
[        ]   Expected: CheckTotal = [17]
[        ]   Actual:   CheckTotal = [18]
[  FAIL  ] <<COMPOUND>> - <<COMPOUND>>.001
[========] 1 tests ran.
[ PASSED ] 0 tests.
[        ] the following 1 tests failed:
[ FAILED ]   <<COMPOUND>> - <<COMPOUND>>.001
[        ]   manager.<<INIT>> - <<COMPOUND>>.001.slot1 - <<INIT>>.001
[        ]   manager.Manager::PlaceOrder - <<COMPOUND>>.001.slot3 -
MANAGER::PLACEORDER.002
[        ]   manager.Manager::PlaceOrder - MANAGER::PLACEORDER.002.slot3.rangeIter2
```

APPENDIX G: CLICAST - TOOL OPTIONS REFERENCE

This reference contains an alphabetized list of VectorCAST's configurable options.

Tool Options

ASM_FUNCS_BEHAVE_AS_INLINES

```
clicast -lc Option ASM_FUNCS_BEHAVE_AS_INLINES True | False
```

Category: Coverage Options

Description: 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 shold be set to true. This option is used when determining couples for control coupling. The default value is FALSE except for DIAB and Green Hills compilers.

ASSEMBLER_CMD

```
clicast -lc Option ASSEMBLER_CMD <assembler command>
```

Category: Language Options

Description: Assembler command and options.

C_ALT_COMPILE_CMD

```
clicast -lc Option C_ALT_COMPILE_CMD <compile command>
```

Category: Language Options

Description: Command used to compile C files in C++ environments.

C_ALT_EDG_FLAGS

```
clicast -lc Option C_ALT_EDG_FLAGS <parser flags>
```

Category: Language Options

Description: Flags to pass to the EDG parser when parsing a C file in a C++ environment.

C_ALT_PREPROCESS_CMD

```
clicast -lc Option C_ALT_PREPROCESS_CMD <preprocessor command>
```

Category: Language Options

Description: Command used to preprocess C files in C++ environments.

C_COMPILE_CMD

```
clicast -lc Option C_COMPILE_CMD <compile command>
```

Category: Language Options

Description: Command used to compile C/C++ harness files.

C_COMPILE_CMD_FLAG

```
clicast -lc Option C_COMPILE_CMD_FLAG <compile command flag>
```

Category: Language Options

Description: This flag is used to detect compilation commands when parsing build output in the Compiler Integration Wizard.

C_COMPILE_EXCLUDE_FLAGS

```
clicast -lc Option C_COMPILE_EXCLUDE_FLAGS <compile exclude flags>
```

Category: Language Options

Description: Arguments matching these flags will not be captured by the Compiler Integration Wizard. 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.

C_COMPILER_CFG_SOURCE

```
clicast -lc Option C_COMPILER_CFG_SOURCE <source of default compiler
options>
```

Category: Language Options

Description: Source of default compiler options.

C_COMPILER_FAMILY_NAME

```
clicast -lc Option C_COMPILER_FAMILY_NAME <compiler family name>
```

Category: Language Options

Description: Indicates the compiler family being used. This field is automatically generated by the VectorCAST compiler configuration utility.

C_COMPILER_HIERARCHY_STRING

```
clicast -lc Option C_COMPILER_HIERARCHY_STRING <compiler menu string>
```

Category: Language Options

Description: Used to create a cascading menu in the C/C++ tab in the Options dialog. This string is parsed and used to group compilation systems hierarchically.

C_COMPILER_OUTPUT_FLAG

```
clicast -lc Option C_COMPILER_OUTPUT_FLAG <compiler output flag>
```

Category: Language Options

Description: Command line option for the compiler to create an object file.

C_COMPILER_PY_ARGS

```
clicast -lc Option C_COMPILER_PY_ARGS <compiler python args>
```

Category: Language Options

Description: Compiler arguments for python configurator.

C_COMPILER_TAG

```
clicast -lc Option C_COMPILER_TAG <compiler tag>
```

Category: Language Options

Description: Used to create a cascading menu in the C/C++ tab.

C_COMPILER_VERSION_CMD

```
clicast -lc Option C_COMPILER_VERSION_CMD <compiler version command>
```

Category: Language Options

Description: Command used to determine the version of the compiler.

C_DEBUG_CMD

```
clicast -lc Option C_DEBUG_CMD <debug command>
```

Category: Language Options

Description: Command used to invoke C/C++ debugger.

C_DEBUG_HELP_FILE

```
clicast -lc Option C_DEBUG_HELP_FILE <file name>
```

Category: Language Options

Description: Instructions for debugging the test harness.

C_DEFINE_FLAG

```
clicast -lc Option C_DEFINE_FLAG <#define flag>
```

Category: Language Options

Description: Command line option used to specify values for C/C++ preprocessor variables.

C_DEFINE_LIST

```
clicast -lc Option C_DEFINE_LIST <list of definitions>
```

Category: Language Options

Description: List of preprocessor variables and definitions to use when compiling C/C++ files. The macro VCAST_CUSTOM_END can be added to the defined variables list, which enables a user-defined function to be called at the very end of test harness processing, right before the **`exit()`** function is called. For example:

```
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);'`**.

C_EDG_FLAGS

```
clicast -lc Option C_EDG_FLAGS <parser flags>
```

Category: Language Options

Description: Flags to pass to the EDG parser.

C_EXEC_HELP_FILE

```
clicast -lc Option C_EXEC_HELP_FILE <file name>
```

Category: Language Options

Description: Instructions for executing the test harness.

C_EXECUTE_CMD

```
clicast -lc Option C_EXECUTE_CMD <execute command | flag>
```

Category: Language Options

Description: Command (or special VectorCAST/Target flag) used to indicate method of executing harness.

C_INCLUDE_FLAG

```
clicast -lc Option C_INCLUDE_FLAG <search directory flag>
```

Category: Language Options

Description: Command line option used to specify search directories when compiling C/C++ files.

C_LINK_CMD

```
clicast -lc Option C_LINK_CMD <command>
```

Category: Language Options

Description: Command used to link object files into an executable C/C++ test harness. The command is a quoted string.

C_LINK_OPTIONS

```
clicast -lc Option C_LINK_OPTIONS <linker options>
```

Category: Language Options

Description: Additional command line options used when linking C/C++ harness.

C_LINKER_VERSION_CMD

```
clicast -lc Option C_LINKER_VERSION_CMD <linker version command>
```

Category: Language Options

Description: Command used to determine the version of the linker.

C_OBJECT_EXT

```
clicast -lc Option C_OBJECT_EXT <object file extension>
```

Category: Language Options

Description: File extension used by C/C++ compiler to specify object files.

C_OUTPUT_FLAG

```
clicast -lc Option C_OUTPUT_FLAG <output file specifier>
```

Category: Language Options

Description: Command line option used to specify name of executable created by linker.

C_PREPROCESS_CMD

```
clicast -lc Option C_PREPROCESS_CMD <preprocessor command>
```

Category: Language Options

Description: Command used to preprocess C/C++ source files.

C_PREPROCESS_FILE

```
clicast -lc Option C_PREPROCESS_FILE <preprocessor file template>
```

Category: Language Options

Description: Template of name of file created by preprocessor. Consists of preprocessor output file name with '?' in place of source file name. For example, if the file 'manager.c' is preprocessed into

'manager.l', the value for this option would be '?.l'

CASE_FALLTHROUGH_BRANCH_COVERAGE

```
clicast -lc Option CASE_FALLTHROUGH_BRANCH_COVERAGE True | False
```

Category: Coverage Options

Description: Setting this option will cause a case statement reached by fallthrough to be treated as a covered branch. The default value of this option is False.

COMREADER_BAUD

```
clicast -lc Option COMREADER_BAUD <baud rate>
```

Category: Target Options

Description: Serial port baud rate.

COMREADER_COMPORT

```
clicast -lc Option COMREADER_COMPORT <comport name>
```

Category: Target Options

Description: Name of serial port (e.g. 'COM1').

COMREADER_DATA_BITS

```
clicast -lc Option COMREADER_DATA_BITS <number of data bits>
```

Category: Target Options

Description: Number of data bits in communication protocol.

COMREADER_ENABLED

```
clicast -lc Option COMREADER_ENABLED True | False
```

Category: Target Options

Description: Use utility to read data from serial port.

COMREADER_PARITY

```
clicast -lc Option COMREADER_PARITY Y | N
```

Category: Target Options

Description: Communication protocol uses parity bits.

COMREADER_STOP_BITS

```
clicast -lc Option COMREADER_STOP_BITS number of stop bits
```

Category: Target Options

Description: Number of stop bits in communication protocol.

COVERAGE_ANIMATION_PLAY_SPEED_RATIO

```
clicast -lc Option COVERAGE_ANIMATION_PLAY_SPEED_RATIO <floating point
number>
```

Category: Coverage Options

Description: 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.

COVERAGE_IO_TYPE

```
clicast -lc Option COVERAGE_IO_TYPE Real_Time | Buffered | Animation
```

Category: Coverage Options

Description: Type of coverage I/O that the instrumented harness uses to record coverage data.

- Real time coverage I/O gathers coverage data and writes it to the results file immediately.
- Buffered coverage I/O gathers coverage data and buffers it in memory until the end of the test case, or on test harness termination if the option VCAST_DUMP_COVERAGE_AT_EXIT is set to TRUE. For Cover environments, buffered I/O gathers coverage data and buffers it in memory until VCAST_DUMP_COVERAGE_DATA() is called. If the option VCAST_DUMP_COVERAGE_AT_EXIT is set to TRUE then the data is instead written to the results file when the application terminates and no call to VCAST_DUMP_COVERAGE_DATA() is needed.
- Animation coverage I/O writes data to the results file in the order it was encountered, and does not remove duplicates. Animation coverage I/O is required for Basis Path coverage, and supports replaying the order of execution of the code in the Coverage Viewer.

COVERAGE_TARGET

```
clicast -lc Option COVERAGE_TARGET <integer number>
```

Category: Coverage Options

Description: Coverage is considered to Fail if it doesn't meet specified percentage.

COVERAGE_TYPE

```
clicast -lc Option COVERAGE_TYPE None | STATEMENT+MC/DC | STATEMENT+BRANCH
| FUNCTION+FUNCTION_CALL | FUNCTION | MC/DC | BASIS_PATHS | BRANCH |
STATEMENT
```

Category: Builder Options

Description: Type of coverage instrumentation to perform immediately after building environment. By setting this option to a value other than None coverage will be initialized each time you build a new environment.

EVENT_LIMIT

```
clicast -lc Option EVENT_LIMIT <integer number>
```

Category: Execute Options

Description: Maximum number of events for the harness to process. The harness will abort execution when this limit is exceeded. This is useful for testing code with infinite loops. If this limit is 0, no results data will be captured (coverage data will still be captured).

EXECUTABLE_EXTENSION

```
clicast -lc Option EXECUTABLE_EXTENSION <extension>
```

Category: Language Options

Description: File extension of executables created by VectorCAST.

EXECUTE ALL

```
clicast -e <env> EXecute All [<report filename>]
```

Category: Execute Commands

Description: This command executes all test cases in the specified environment. Regardless of the

pass or fail status of the test execution, if a report file name is given, then additionally a Testcase Management Report is generated with the given filename. The exit status from the 'execute all' command is zero if all tests pass and non-zero otherwise.

FLOATING_POINT_TOLERANCE

```
clicast -lc Option FLOATING_POINT_TOLERANCE <floating point number>
```

Category: Execute Options

Description: Percentage that a floating point value can vary from its expected value and still be considered a match.

FORCE_EXPLICIT_TEMPLATE_ARGS

```
clicast -lc Option FORCE_EXPLICIT_TEMPLATE_ARGS True | False
```

Category: Builder Options

Description: When VectorCAST generates a call to a template-based function, explicit template arguments are used if calls to the function in the original code used explicit template arguments. If there were no calls to the function in the original code, then generated calls to the function will use explicit arguments only if this option is enabled.

GLOBALS_DISPLAY

```
clicast -lc Option GLOBALS_DISPLAY Each_event | Range_Iteration | Slot_
Iteration | Testcase
```

Category: Report Options

Description: When to display / check global values in execution results: at each event; at the end of each range iteration; at the end of each slot iteration; at the end of each slot.

INST_LINK_FILE

```
clicast -lc Option INST_LINK_FILE <file name>
```

Category: Language Options

Description: Path and name of link file for uut_inst executable. A copy of this file is copied into the environment directory as inst.lkf during environment build.

LIBRARY_INCLUDE_DIR

```
clicast -lc Option LIBRARY_INCLUDE_DIR <library include directory>
```

Category: Language Options

*Description:*Directories listed here are loaded by default into the Create New Environment wizard. Add or delete testable source directories, library include directories, or type-handled directories to make it more convenient to build a new environment. Changes made here do not have any effect on an existing environment.

LIBRARY_TESTING

```
clicast -lc Option LIBRARY_TESTING Source | Library
```

Category: Builder Options

Description: Default setting for Library Testing option in the environment builder.

LINES_PER_PAGE

```
clicast -lc Option LINES_PER_PAGE <integer number>
```

Category: Report Options

Description: Number of lines between form-feeds in extracted reports. A value of 0 indicates no pagination will take place.

LINK_FILE_TEMPLATE

```
clicast -lc Option LINK_FILE_TEMPLATE <file name>
```

Category: Language Options

Description: Path and name of link file template that is copied into the environment when using the Cosmic compiler. The copy occurs during environment building. If not specified, the cosmic.cmd (or cosmicp.cmd for paged) file in the environment files option will be used.

MANAGE_ENVIRONMENT_VARIABLE

```
clicast -lc Option MANAGE_ENVIRONMENT_VARIABLE
```

Category: Language Options

Description: Environment variables used when building, executing, importing, or executing python scripts.

MANAGE_LIBRARY_INCLUDE_DIR

```
clicast -lc Option MANAGE_LIBRARY_INCLUDE_DIR <library include directory>
```

Category: Language Options

Description: Directories listed here are loaded by default into the Create New Environment wizard. Add or delete testable source directories, library include directories, or type-handled directories to make it more convenient to build a new environment. Changes made here do not have any effect on an existing environment.

MANAGE_ROOT_SOURCE_DIR

```
clicast -lc Option MANAGE_ROOT_SOURCE_DIR <root source directory>
```

Category: Language Options

Description: The common root directory of environment source files.

MANAGE_TESTABLE_SOURCE_DIR

```
clicast -lc Option MANAGE_TESTABLE_SOURCE_DIR <testable source directory>
```

Category: Language Options

Description: Directory containing source code files that you would like to test or stub.

MANAGE_TYPE_HANDLED_SOURCE_DIR

```
clicast -lc Option MANAGE_TYPE_HANDLED_SOURCE_DIR <type handled source
directory>
```

Category: Language Options

Description: Directory containing source files that you would like to parse for type information. Any entities residing on this list will not be defined by VectorCAST and therefore must be linked in through a library.

MAX_VARY_RANGE

```
clicast -lc Option MAX_VARY_RANGE <integer number>
```

Category: Target Options

Description: Maximum number of scalars that can be varied in one test case. Changes to this value

will take effect after the environment is rebuilt. Reducing this value to 0 will completely remove list and range processing from the test harness, significantly reducing the size of the harness.

METRICS_TOTAL_LEVEL_DISPLAY

```
clicast -lc Option METRICS_TOTAL_LEVEL_DISPLAY No_Filtering | Subprogram_
Detail | Unit_Detail
```

Category: Report Options

Description: This option allows you to specify which results to filter out.

NUMBER_OF_DATA_PARTITIONS

```
clicast -lc Option NUMBER_OF_DATA_PARTITIONS <positive number>
```

Category: Execute Options

Description: Number of iterations to span an entire range of a scalar parameter. Used with Range Values to allow specification of iterations instead of range delta.

PAGE_WIDTH

```
clicast -lc Option PAGE_WIDTH <integer number>
```

Category: Report Options

Description: Number of columns to use when building reports.

PRECOMPILE_CMD

```
clicast -lc Option PRECOMPILE_CMD <precompile command>
```

Category: Language Options

Description: Command called before compiling C/C++ 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.

PRECOMPILE_EXT

```
clicast -lc Option PRECOMPILE_EXT <extension>
```

Category: Language Options

Description: Extension of files resulting from the pre-compile command.

RANGE_CHECK

```
clicast -lc Option RANGE_CHECK Full | Disable | None
```

Category: Execute Options

Description: Indicates whether range checking should be performed for input and expected test case values. 'All', means that type ranges will be enforced, and out-of-range test data will not be accepted. 'Disable' allows out-of-range test data values to be used. 'None' disables all range processing. This is useful when target communication is slow.

REPORT_OBJECTS

```
clicast -lc Option REPORT_OBJECTS Never | Always
```

Category: Report Options

Description: When preparing execution reports, if this option is set, VectorCAST will display ALL global objects. If the option is not set, then only global objects that are part of the test case (as input or expected values) will be displayed.

REPORT_PARAMETERS

```
clicast -lc Option REPORT_PARAMETERS Never | Always
```

Category: Report Options

Description: When preparing execution reports, if this option is set, VectorCAST will display ALL parameters for each test event. If the option is not set, then only parameters that are part of the test case (as input or expected values) will be displayed.

SBF_LOC_MEMBER_IN_NSP

```
clicast -lc Option SBF_LOC_MEMBER_IN_NSP DECL_NAMESPACE | DEF_NAMESPACE |
GLOBAL_NAMESPACE
```

Category: Language Options

Description: This option tells VectorCAST where to define SBF objects when stubbing member functions declared and defined within a namespace. Different compilers require different locations. The options are DECL_NAMESPACE, DEF_NAMESPACE, or GLOBAL_NAMESPACE.

SBF_LOC_MEMBER_OUTSIDE_NSP

```
clicast -lc Option SBF_LOC_MEMBER_OUTSIDE_NSP DECL_NAMESPACE | DEF_
NAMESPACE | GLOBAL_NAMESPACE
```

Category: Language Options

Description: This option tells VectorCAST where to define SBF objects when stubbing member functions declared within a namespace, but defined outside of it. Different compilers require different locations. The options are DECL_NAMESPACE, DEF_NAMESPACE, or GLOBAL_NAMESPACE.

SBF_LOC_NONMEMBER_IN_NSP

```
clicast -lc Option SBF_LOC_NONMEMBER_IN_NSP DECL_NAMESPACE | DEF_NAMESPACE
| GLOBAL_NAMESPACE
```

Category: Language Options

Description: This option tells VectorCAST where to define SBF objects when stubbing nonmember functions declared and defined within a namespace. Different compilers require different locations. The options are DECL_NAMESPACE, DEF_NAMESPACE, or GLOBAL_NAMESPACE.

SBF_LOC_NONMEMBER_OUTSIDE_NSP

```
clicast -lc Option SBF_LOC_NONMEMBER_OUTSIDE_NSP DECL_NAMESPACE | DEF_
NAMESPACE | GLOBAL_NAMESPACE
```

Category: Language Options

Description: This option tells VectorCAST where to define SBF objects when stubbing nonmember functions declared within a namespace, but defined outside of it. Different compilers require different locations. The options are DECL_NAMESPACE, DEF_NAMESPACE, or GLOBAL_NAMESPACE.

SCORE_DEBUG_CMD

```
clicast -lc Option SCORE_DEBUG_CMD <SCORE debug command>
```

Category: Target Options

Description: Command used to debug on a SCORE target.

SCORE_EXECUTE_CMD

```
clicast -lc Option SCORE_EXECUTE_CMD <SCORE execution command>
```

Category: Target Options

Description: Command used to execute on a SCORE target.

SHOW_NOTES_IN_FULL_REPORT

```
clicast -lc Option SHOW_NOTES_IN_FULL_REPORT True | False
```

Category: Report Options

Description: This option will add a section to the Full Report listing test notes and requirements.

SOURCE_EXTENSION

```
clicast -lc Option SOURCE_EXTENSION <source file extension>
```

Category: Language Options

Description: This option serves two purposes. It puts the VectorCAST parser into C or C++ mode, and it changes the file extensions of the generated source files to .c and .cpp respectively.

STANDARD_ERROR

```
clicast -lc Option STANDARD_ERROR Normal | Redirect
```

Category: Execute Options

Description: If this option is set, any standard error from the test will be captured to a file for inclusion in the test execution report.

STANDARD_OUTPUT

```
clicast -lc Option STANDARD_OUTPUT Normal | Redirect
```

Category: Execute Options

Description: If this option is set, any standard output from the test will be captured to a file for inclusion in the test execution report.

STUB_DEPENDENCIES

```
clicast -lc Option STUB_DEPENDENCIES Yes | No | Custom
```

Category: Builder Options

Description: Default setting for Stub Dependencies option in the environment builder. ALL means stub all dependencies. NONE means do not stub any dependencies.

SUBSTITUTE_CODE_FOR_C_FILE

```
clicast -lc Option SUBSTITUTE_CODE_FOR_C_FILE True | False
```

Category: Target Options

Description: When constructing a test harness, some VectorCAST-generated source files may #include other source files that contain function definitions. Some debuggers are not able to show source code for functions defined in #included files. Enable this option to have VectorCAST replace such #includes with the content of the included file before compiling. Rebuild the environment after enabling this option. Its default value is false.

TARGET_BOARD_NAME

```
clicast -lc Option TARGET_BOARD_NAME <target name>
```

Category: Target Options

Description: Name of target board. For vxWorks this is the name of the target server, for other targets it is the hostname or IP address of the target.

TARGET_BOOT_HOSTNAME

```
clicast -lc Option TARGET_BOOT_HOSTNAME <host name>
```

Category: Target Options

Description: For vxWorks. This is the name that the target knows the boot host as. If this option is not set, VectorCAST assumes that the hostname of the machine it is running on, is the boot host.

TARGET_IO_DIRECTORY

```
clicast -lc Option TARGET_IO_DIRECTORY <directory>
```

Category: Target Options

Description: Directory where Input/Output files for target test execution will be stored. This option should be used when the target does not have read/write permission for the test environment directory.

TARGET_SIM_CMD

```
clicast -lc Option TARGET_SIM_CMD <simulator command>
```

Category: Target Options

Description: Command used to execute on the target or simulator.

TARGET_TFTP_DIR

```
clicast -lc Option TARGET_TFTP_DIR <directory>
```

Category: Target Options

Description: This is the directory that the executable files will be copied to so that the target can be booted via tftp.

TEST_CASE_TIMEOUT

```
clicast -lc Option TEST_CASE_TIMEOUT <integer number>
```

Category: Target Options

Description: Option TEST_CASE_TIMEOUT <integer number>. Maximum amount of time, in seconds, to wait before VectorCAST terminates a test execution. The default value is 0 which means wait 'forever'.

TESTABLE_SOURCE_DIR

```
clicast -lc Option TESTABLE_SOURCE_DIR <testable source directory>
```

Category: Language Options

Description: Directory containing source code files that you would like to test or stub.

TI_CC_DSP_BIOS_SUPPORT

```
clicast -lc Option TI_CC_DSP_BIOS_SUPPORT True | False
```

Category: Language Options

Description: This option enables DSP BIOS support for the TI Code Composer target you have selected.

TI_CC_TCF_CMD

```
clicast -lc Option TI_CC_TCF_CMD <TI CC DSP BIOS TCF Command>
```

Category: Language Options

Description: This option indicates the tconf command to be used in conjunction with the TI CC TCF Filename Option to generate the BIOS specific files.

TI_CC_TCF_FILENAME

```
clicast -lc Option TI_CC_TCF_FILENAME <TI CC DSP BIOS TCF Filename>
```

Category: Language Options

Description: This option indicates the TCF filename that has been selected to configure the desired TI DSP BIOS options. Select the path to the tcf file here and then hit the Generate button which will create the necessary source files, compile them and relink with the object files and command files.

TYPE_HANDLED_SOURCE_DIR

```
clicast -lc Option TYPE_HANDLED_SOURCE_DIR <type handled source directory>
```

Category: Language Options

Description: Directory containing source files that you would like to parse for type information.Any entities residing on this list will not be defined by VectorCAST and therefore must be linked in through a library.

UNCON_ARRAY_SIZE

```
clicast -lc Option UNCON_ARRAY_SIZE <integer number> | <integer
number>:<integer number>
```

Category: Builder Options

Description: Default size for unconstrained arrays where the index range is unknown or very large. May be specified as [lower bound]:[upper bound] or just [upper bound].

UUT_LINK_FILE

```
clicast -lc Option UUT_LINK_FILE <file name>
```

Category: Language Options

Description: Path and name of link file for uut_inte executable. A copy of this file is copied into the environment directory as uut.lkf during environment build.

VARIANT_LOGICS_PATH

```
clicast -lc Option VARIANT_LOGICS_PATH <path to variant logics file>
```

Category: Execute Options

Description: Path to .py file containing test case variant logic definitions written in Python. A Logic from this file can be assigned to one or more testcases, and is evaluated prior to execution to determine if the test is eligible for test execution. Path can be absolute or relative, and can contain environment variables, using the syntax $(ENV_VAR) or %ENV_VAR%.

VCAST_ACCESS_NON_SEARCH_GLOBALS

```
clicast -lc Option VCAST_ACCESS_NON_SEARCH_GLOBALS TRUE | FALSE
```

Category: Builder Options

Description: This option directs VectorCAST/C++ to display global variables declared in non-search directory header files in the Parameter Tree. When the option is False, such variables are not shown, but can be accessed via user code.

VCAST_ADA_FILE_EXTENSIONS

```
clicast -lc Option VCAST_ADA_FILE_EXTENSIONS <ada file extensions>
```

Category: Language Options

Description: Use this option to specify the ada file extensions.

VCAST_ALWAYS_DO_STUB_PROCESSING_IN_TI

```
clicast -lc Option VCAST_ALWAYS_DO_STUB_PROCESSING_IN_TI True | False
```

Category: Builder Options

Description: This option controls whether stubs called during test object initialization are fully enabled. Test object initialization refers to the initialization of objects as designated in the parameter tree. When this option is enabled, stubs called during test object initialization (such as stubbed constructors) will perform all stub-related processing. When this option is disabled, stubs called during test object initialization will skip most stub-related processing, but configure stubs user code and object initialization user code can still be executed. Note that when the option is enabled, an execution report is more likely to show stub events before the call to the function under test.

VCAST_APPEND_TO_TESTINSS

```
clicast -lc Option VCAST_APPEND_TO_TESTINSS True | False
```

Category: Coverage Options

Description: This option tells VectorCAST to open the coverage data file for appending, rather than always creating the data file.

VCAST_ASSEMBLY_FILE_EXTENSIONS

```
clicast -lc Option VCAST_ASSEMBLY_FILE_EXTENSIONS <assembly file
extensions>
```

Category: Language Options

Description: List of file extensions indicating assembly code. These extensions are used to determine whether any startup files should be assembled.

VCAST_ASSIGN_WITHOUT_COPY_CTOR

```
clicast -lc Option VCAST_ASSIGN_WITHOUT_COPY_CTOR <True | False>
```

Category: Builder Options

Description: When creating the harness, VectorCAST will use the assignment operator to record return values from functions when the class lacks an accessible copy constructor but has a visible assignment operator and default constructor.

VCAST_AUTO_CLEAR_TEST_USER_CODE

```
clicast -lc Option VCAST_AUTO_CLEAR_TEST_USER_CODE True | False
```

Category: Execute Options

Description: This option tells VectorCAST to clear test user code prior to executing tests. It is helpful when you need to keep the executable size small.

VCAST_AUTO_CONCRETE_CLASS_GENERATION

```
clicast -lc Option VCAST_AUTO_CONCRETE_CLASS_GENERATION True | False
```

Category: Builder Options

Description: When this option is set, VectorCAST will generate concrete subclasses of abstract classes found in the code under test.

VCAST_AUTO_TEST_INSTANTIATE_CLASS

```
clicast -lc Option VCAST_AUTO_TEST_INSTANTIATE_CLASS True | False
```

Category: Builder Options

Description: Enable this clicast-only option to have automatically-generated tests (such as basis path and MC/DC tests) include instantiation of the class instance when the test is for a non-static member function. Disable this option when the automatic instantiation does not properly setup the instance. With this option disabled, the class instance will need to be setup in some other way, probably via a user-created automatic initialization test. Automatic tests must be regenerated after toggling this option.

VCAST_AVOID_COMMA_OPERATOR

```
clicast -lc Option VCAST_AVOID_COMMA_OPERATOR True | False
```

Category: Coverage Options

Description: 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. It has no effect when using the legacy instrumenter.

VCAST_BASIS_PATHS_FOR_CONSTANT_BRANCHES

```
clicast -lc Option VCAST_BASIS_PATHS_FOR_CONSTANT_BRANCHES True | False
```

Category: Coverage Options

Description: 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.

VCAST_BUFFER_OUTPUT

```
clicast -lc Option VCAST_BUFFER_OUTPUT True | False
```

Category: Target Options

Description: This option indicates that you are running tests on a target board that does not have 'STDIN' capability and cannot write to files. In this case, VectorCAST will compile and link the test case data into the test harness, so that no data has to be 'read' by the test harness. Output data is stored in a buffer in the test harness.

VCAST_C_FILE_EXTENSIONS

```
clicast -lc Option VCAST_C_FILE_EXTENSIONS <c file extensions>
```

Category: Language Options

Description: List of file extensions indicating C source code. Typical extensions are supported by default; this option is only needed when source files do not follow normal coding conventions.

VCAST_CLASS_INST_SHARING

```
clicast -lc Option VCAST_CLASS_INST_SHARING True | False
```

Category: Builder Options

Description: Use this option to share class instances across multiple units under test. Only objects that are legal to declare will be shared.

VCAST_CMD_AFTER_INSTRUMENTATION

```
clicast -lc Option VCAST_CMD_AFTER_INSTRUMENTATION <cmd>
```

Category: Coverage Options

Description: This command will be run after VectorCAST has instrumented a file.

VCAST_CMD_DURING_ENV_CLOSE

```
clicast -lc Option VCAST_CMD_DURING_ENV_CLOSE <cmd>
```

Category: Target Options

Description: Command to be executed while closing a unit test environment, where **<cmd>** is an executable application or script to be executed in the environment directory. The command is executed just as the environment begins to close. If running a script, the interpreter should be specified.

Note that when using clicast, the user should be aware that every command taking the argument **-e <env>** opens and closes the environment. Therefore, the options can be set immediately before calling the clicast command of interest, and unset afterwards.

VCAST_CMD_DURING_ENV_OPEN

```
clicast -lc Option VCAST_CMD_DURING_ENV_OPEN <cmd>
```

Category: Target Options

Description: Command to be executed while opening a unit test environment, where **<cmd>** is an executable application or script to be executed in the environment directory. The command is executed right before building the range data during environment open. If running a script, the interpreter should be specified.

Note that when using clicast, the user should be aware that every command taking the argument **-e <env>** opens and closes the environment. Therefore, the options can be set immediately before calling the clicast command of interest, and unset afterwards.

VCAST_COLLAPSE_STD_HEADERS

```
clicast -lc Option VCAST_COLLAPSE_STD_HEADERS Collapse_None | Collapse_
System_Headers | Collapse_Non_Search_Headers
```

Category: Language Options

Description: 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. Choose "None" if your code contains macros defined in search directory files but which affect compilation of code in non-search directory headers. Choose "System headers" or "Non-search directory headers" if the headers contain code that cannot be compiled unless physically located in its original file location. The default value is set by the compiler template. Consult VectorCAST Technical Support before changing this option.

VCAST_COMMAND_AFTER_C_OPTIONS

```
clicast -lc Option VCAST_COMMAND_AFTER_C_OPTIONS <cmd>
```

Category: Coverage Options

Description: 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.

VCAST_COMMAND_LINE_DEBUGGER

```
clicast -lc Option VCAST_COMMAND_LINE_DEBUGGER True | False
```

Category: Language Options

Description: This option causes VectorCAST to bring up a shell window to run the debugger.

VCAST_COMPACT

```
clicast -lc Option VCAST_COMPACT True | False
```

Category: Report Options

Description: This option causes VectorCAST to compress the execution results for scalar array elements that are the same. If a five element array has all values = 0, this option will enable the report to have one output line of (1..5) rather than 5 lines of one value per line.

VCAST_COMPILER_SUPPORTS_CPP_CASTS

```
clicast -lc Option VCAST_COMPILER_SUPPORTS_CPP_CASTS True | False
```

Category: Builder Options

Description: Enabling this option allows VectorCAST to use C++-style casts in a C++ test harness. When this option is disabled only C-style casts are used.

VCAST_COMPILER_TEMPLATE_SECTION

```
clicast -lc Option VCAST_COMPILER_TEMPLATE_SECTION True | False
```

Category: Report Options

Description: This option will add a section to the bottom of the Full Report that lists the Compiler and Linker settings for the environment.

VCAST_CONSIDER_UNCALLED_TEMPL_FUNCTIONS_TESTABLE

```
clicast -lc Option VCAST_CONSIDER_UNCALLED_TEMPL_FUNCTIONS_TESTABLE True |
False
```

Category: Builder Options

Description: This option will consider uncalled template member functions of an already testable template class to be testable. Consider a class A<T> which contains two template functions A<T>::b and A<T>::c. If A<int>::b is instantiated, then A<int>::c would become testable as well.

VCAST_CONSOLE_REPORT_FORMAT

```
clicast -lc VCAST_CONSOLE_REPORT_FORMAT CONSOLE_REPORT_DEFAULT | CONSOLE_
REPORT_LEGACY | CONSOLE_REPORT_VERBOSE | CONSOLE_REPORT_QUIET
```

Category: Report Options

Description: This clicast-only option by default (CONSOLE_REPORT_DEFAULT), shows command line output with passes and failures at the iteration level, and Expected/Actual value mismatch at the iteration level. After all tests have run, a test summary is shown along with a list of all failing tests. With Legacy output (CONSOLE_REPORT_LEGACY), the previous style of console output is shown. Verbose output (CONSOLE_REPORT_VERBOSE) prints both Default and Legacy Output. When set to quiet (CONSOLE_REPORT_QUIET), minimal test/run information is shown in the console output.

VCAST_CONTROL_FLOW_GRAPH_CONFIG

```
clicast -lc Option VCAST_CONTROL_FLOW_GRAPH_CONFIG <config text>
```

Category: Report Options

Description: This clicast-only option allows the customization of the nodes and text length in control flow graphs generated by the custom tool named "Control Flow Graph". It customizes the control flow graph by setting one or more options to control what nodes are displayed, and how long the expressions are that are displayed. **<config text>** is a comma-separated list, enclosed in curly braces, of one or more options:

"show_branch_ids":"FALSE" or "TRUE" - Display the branch/decision numbers in the graph. The default value is "FALSE".

"show_only_branches":"FALSE" or "TRUE" - Display only branch nodes in the graph to simplify it. The default value is "FALSE", or show all nodes.

"max_cond_expression_len":<num>, where **<num>** is the maximum length in characters of the expression shown in conditions. Default value is 18 characters. Use 0 for unlimited.

"max_expression_len":<num>, where **<num>** is the maximum length in characters of the expression shown in non-branch or non-condition nodes. The default value is 24 characters. Use 0 for unlimited.

The following example restricts the graph to only branch/condition nodes and shows the branch/condition numbers in those nodes, and increases the maximum length of the condition's text to 30 characters:

```
clicast -lc option VCAST_CONTROL_FLOW_GRAPH_CONFIG '{"show_branch_
ids":"TRUE","show_only_branches":"TRUE","max_cond_expression_len":30}'
```

VCAST_CONVERT_OCTAL_HEX_STRINGS_TO_ASCII

```
clicast -lc Option VCAST_CONVERT_OCTAL_HEX_STRINGS_TO_ASCII True | False
```

Category: Report Options

Description: This option is used to determine whether or not to convert "unprintable" strings encoded in octal or hexidecimal notation to ASCII encoding in the report output. It does not affect the actual string comparisons in the test execution.

VCAST_COVER_CATCH_AS_BRANCH

```
clicast -lc Option VCAST_COVER_CATCH_AS_BRANCH True | False
```

Category: Coverage Options

Description: Enabling this option will provide code coverage for C/C++ catch blocks as if they are branches.

VCAST_COVER_OVERLOADED_LOGICAL_OPERATORS

```
clicast -lc Option VCAST_COVER_OVERLOADED_LOGICAL_OPERATORS True | False
```

Category: Coverage Options

Description: Enabling this option will provide code coverage for expressions using overloaded logical and comparison operators which return bool or integer values as if they are comparison operators. The default value is True.

VCAST_COVER_SEPARATE_TYPES_FILES

```
clicast -lc Option VCAST_COVER_SEPARATE_TYPES_FILES <ada file extensions>
```

Category: Coverage Options

Description: Put unit coverage utilities in separate file.

VCAST_COVER_STATEMENTS_BY_BLOCK

```
clicast -lc Option VCAST_COVER_STATEMENTS_BY_BLOCK True | False
```

Category: Coverage Options

Description: Enabling this option will change coverage instrumentation to instrument blocks for statement coverage. This reduces the amount of program memory used.

VCAST_COVERAGE_COLLAPSE_ALL

```
clicast -lc Option VCAST_COVERAGE_COLLAPSE_ALL True | False
```

Category: Language Options

Description: This option tells VectorCAST to replace all #included files in preprocessed files with appropriate #includes before adding instrumentation. It is not used if the "Provide code coverage in header files" option is enabled or if "Stub by Function" or Visual C++ white box is in effect.

VCAST_COVERAGE_DATABASE_CACHE_SIZE

```
clicast -lc Option VCAST_COVERAGE_DATABASE_CACHE_SIZE <integer number>
```

Category: Coverage Options

Description: 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 re-opening an environment.

VCAST_COVERAGE_DATABASE_SIMPLIFIED_LOCKING

```
clicast -lc Option VCAST_COVERAGE_DATABASE_SIMPLIFIED_LOCKING True | False
```

Category: Coverage Options

Description: Certain file system configurations, especially some using NFS, do not have a properly functioning POSIX locking implementation. Enabling this option makes the database layer use an alternate, simpler locking method which should allow the database to function on any file system. However, when enabled, only one user or instance of VectorCAST will be able to access the database at a time.

VCAST_COVERAGE_FIELD_WIDTH

```
clicast -lc Option VCAST_COVERAGE_FIELD_WIDTH <integer number>
```

Category: Report Options

Description: 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 has a large number of statements or branches.

VCAST_COVERAGE_FOR_AGGREGATE_INIT

```
clicast -lc Option VCAST_COVERAGE_FOR_AGGREGATE_INIT True | False
```

Category: Coverage Options

Description: Setting this option to true causes coverage instrumentation to be performed for aggregate initialized variable declarations in functions in C units. (Such declarations are always instrumented in C++ units.) This option should be set to false if your compiler does not accept function calls in aggregate initializations.

VCAST_COVERAGE_FOR_DECLARATIONS

```
clicast -lc Option VCAST_COVERAGE_FOR_DECLARATIONS INSTRUMENT_VARIABLE_
DECLARATIONS_NONE | INSTRUMENT_VARIABLE_DECLARATIONS_INITIALIZATIONS |
INSTRUMENT_VARIABLE_DECLARATIONS_ALL
```

Category: Coverage Options

Description: Enabling this option on causes coverage instrumentation to be performed for initialized variable declarations in functions in C units. (Such declarations are always instrumented in C++ units.)

VCAST_COVERAGE_FOR_HEADERS

```
clicast -lc Option VCAST_COVERAGE_FOR_HEADERS True | False
```

Category: Coverage Options

Description: Enabling this option will provide code coverage for C/C++ header files found in search directories.

VCAST_COVERAGE_FOR_LAMBDAS

```
clicast -lc Option VCAST_COVERAGE_FOR_LAMBDAS True | False
```

Category: Coverage Options

Description: Enabling this option causes coverage instrumentation and basis path analysis to be performed for lambda functions. The default value is True, except for the Visual Studio 2010 compiler template.

VCAST_COVERAGE_POINTS_AS_MACROS

```
clicast -lc Option VCAST_COVERAGE_POINTS_AS_MACROS True | False
```

Category: Coverage Options

Description: 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.

VCAST_CPP_FILE_EXTENSIONS

```
clicast -lc Option VCAST_CPP_FILE_EXTENSIONS <file extensions>
```

Category: Language Options

Description: List of file extensions indicating C++ source code. Typical extensions are supported by default; this option is only needed when source files do not follow normal coding conventions.

VCAST_CREATE_INIT_OBJECTS_DYNAMICALLY

```
clicast -lc Option VCAST_CREATE_INIT_OBJECTS_DYNAMICALLY True | False
```

Category: Builder Options

Description: When this option is enabled, VectorCAST will create initialization objects that are constructed dynamically. When disabled, initialization objects are defined at file scope and will be constructed at static initialization time (before main is invoked). Note this option applies only to initialization objects used for stubbed constructor member initializers. Initialization objects for stubbed global variables are never constructed dynamically.

VCAST_CUSTOM_REPORT_FORMAT

```
clicast -lc Option VCAST_CUSTOM_REPORT_FORMAT HTML | TEXT
```

Category: Report Options

Description: Output format for VectorCAST reports: HTML or text.

VCAST_CYGPATH_LINE_DIRECTIVES

```
clicast -lc Option VCAST_CYGPATH_LINE_DIRECTIVES True | False
```

Category: Language Options

Description: This option controls whether line directives in preprocessor output are converted from Cygwin-style paths to Windows-style paths. Enable this option when using a compiler that produces Cygwin-style paths. This option requires that the cygpath executable can be found via the PATH environment variable. The default value is True for GNU Native compilers version 5.0 or greater on Windows. For all other compilers, the default value is False.

VCAST_DEFAULT_SCRIPT_HEADER

```
clicast -lc Option VCAST_DEFAULT_SCRIPT_HEADER <Path to default header text
file>
```

Category: Report Options

Description: This option allows you to enter a path to a text file that contains a custom header for all generated test scripts.

VCAST_DEPENDENCY_CACHE_DIR

```
clicast -lc Option VCAST_DEPENDENCY_CACHE_DIR <directory path>
```

Category: Builder Options

Description: This directory is used to cache the dependency data (VCAST.QIK) files. Specify this option if you do not have write access to the search directories.

VCAST_DETECT_THROWN_EXCEPTION_TYPES

```
clicast -lc Option VCAST_DETECT_THROWN_EXCEPTION_TYPES True | False
```

Category: Builder Options

Description: Enable this option to allow test cases to specify expected exceptions of any type thrown directly from the body of the function. If a function definition uses an exception specification, this option is ignored and only the types from the specification can be expected exceptions. When this option is disabled, only functions with exception specifications can have test cases with expected exceptions.

VCAST_DETECT_UNUSED_EXPECTED_UC

```
clicast -lc Option VCAST_DETECT_UNUSED_EXPECTED_UC True | False
```

Category: Execute Options

Description: When enabled, this option detects test-specific expected user code that has not been executed during a test. Such code is most likely to be associated with a stub that has not been called. When True and some testcase or parameter user code has not been executed, the test fails, and the Execution Report displays the unused values, the unit, and whether that unit is stubbed or not-stubbed.

Disable this option if you have added parameter user code to a stub that you expect to never get called. (You might add code to an uncalled stub if you want the test to show a failure if the stub ever gets called.)

VCAST_DISABLE_ASM_FUNC_TESTING

```
clicast -lc Option VCAST_DISABLE_ASM_FUNC_TESTING True | False
```

Category: Builder Options

Description: When this option is selected, assembly functions will not be considered testable.

VCAST_DISABLE_AUTO_BASIS_PATH_GEN

```
clicast -lc Option VCAST_DISABLE_AUTO_BASIS_PATH_GEN True | False
```

Category: Coverage Options

Description: When this option is set, VectorCAST will not generate Basis Path information unless specifically requested by the user.

VCAST_DISABLE_BOOLEAN_CAST

```
clicast -lc Option VCAST_DISABLE_BOOLEAN_CAST True | False
```

Category: Coverage Options

Description: Select this option to prevent VectorCAST from casting conditional expressions to Boolean when instrumenting for branch or MC/DC coverage.

VCAST_DISABLE_CPP_EXCEPTIONS

```
clicast -lc Option VCAST_DISABLE_CPP_EXCEPTIONS True | False
```

Category: Builder Options

Description: Disable the catching of unhandled exceptions in the harness. Check this if your compiler does not handle exceptions.

VCAST_DISABLE_DIRECT_ARRAY_INDEXING

```
clicast -lc Option VCAST_DISABLE_DIRECT_ARRAY_INDEXING True | False
```

Category: Builder Options

Description: This option is used to allow VectorCAST to know how to display the subscripts of an array in the Parameter Tree and in the test script. For example, if you have an array indexed with an enumerated value, if this option is set, then the array subscripts are displayed with the enumerated constants. If this option is not set (default), VectorCAST displays the subscripts with the enumerations.

VCAST_DISABLE_STD_CONTAINER_DETECTION

```
clicast -lc Option VCAST_DISABLE_STD_CONTAINER_DETECTION True | False
```

Category: Builder Options

Description: This option is used to allow VectorCAST to detect ANSI C++ standard container types. Normally, VectorCAST allows the user to enter a container directly. When this option is on, User

Code must be used.

VCAST_DISABLE_STD_SMART_PTR_DETECTION

```
clicast -lc Option VCAST_DISABLE_STD_SMART_PTR_DETECTION True | False
```

Category: Builder Options

Description: This option is used to allow VectorCAST to detect ANSI C++ standard smart pointer types. Normally, VectorCAST allows the user to initialize smart pointers with the test case parameter tree. When this option is on, User Code must be used. The default value is False, meaning detect smart pointers in source code.

VCAST_DISABLE_STD_STRING_DETECTION

```
clicast -lc Option VCAST_DISABLE_STD_STRING_DETECTION True | False
```

Category: Builder Options

Description: This option is used to allow VectorCAST to detect ANSI C++ standard string types. Normally, VectorCAST allows the user to enter a string directly. When this option is on, User Code must be used.

VCAST_DISABLE_STD_WSTRING_DETECTION

```
clicast -lc Option VCAST_DISABLE_STD_WSTRING_DETECTION True | False
```

Category: Builder Options

Description: This option is used to allow VectorCAST to detect ANSI C++ standard wstring types. Normally, VectorCAST allows the user to enter a wstring directly. When this option is on, User Code must be used.

VCAST_DISABLE_TEST_MAINTENANCE

```
clicast -lc Option VCAST_DISABLE_TEST_MAINTENANCE True | False
```

Category: Builder Options

Description: This clicast-only option enables Test Script Maintenance by default and will try to update renamed parameters, subprograms, and other parts of test scripts to reflect changes in source code when importing test scripts. The default value is False. To disable the utility from running set the option to True.

VCAST_DISABLE_TI_BITFIELD

```
clicast -lc Option VCAST_DISABLE_TI_BITFIELD True | False
```

Category: Target Options

Description: This option disables type processing for bit-fields in order to reduce the size of the harness. User code is required to set and check values of parameters, returns, and global objects.

VCAST_DISABLE_TI_STRING

```
clicast -lc Option VCAST_DISABLE_TI_STRING True | False
```

Category: Target Options

Description: This option disables type processing of char* types as 'strings' in order to reduce the size of the harness. You can still use array mode to set the individual characters of a char* type.

VCAST_DISPLAY_CONST_BRANCH_CONDITIONS

```
clicast -lc option VCAST_DISPLAY_CONST_BRANCH_CONDITIONS TRUE | FALSE
```

Category: Report Options

Description: This 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)". The default value is False.

VCAST_DISPLAY_EMPTY_COVERAGE

```
clicast -lc Option VCAST_DISPLAY_EMPTY_COVERAGE <True | False>
```

Category: Report Options

Description: Display annotated source code in reports even if no run-time coverage data exists. If this option is disabled, the message "No Coverage Data Exists" will be displayed instead.

VCAST_DISPLAY_FUNCTION_COVERAGE

```
clicast -lc Option VCAST_DISPLAY_FUNCTION_COVERAGE <True | False>
```

Category: Report Options

Description: Extra column that reports if a function was entered. This option is only used if the coverage type is not Function.

VCAST_DISPLAY_UNINST_EXPR

```
clicast -lc Option VCAST_DISPLAY_UNINST_EXPR <True | False>
```

Category: Report Options

Description: When True, this option adds a 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 user may have turned off MC/DC instrumentation by setting options such as "Instrument logical expressions in assignment statements" to False, or the uninstrumented expressions may be due to an instrumentation bug in VectorCAST. Contact VectorCAST Technical Support for clarification on the effects of instrumentation options on the content of this report.

VCAST_DO_COMBINATION

```
clicast -lc Option VCAST_DO_COMBINATION True | False
```

Category: Execute Options

Description: When this option is on, and multiple parameters have a range of values, all combinations of the range values will be used as input data. (e.g. Two parameters being varied from 1..5 will result in 25 calls to the UUT) If this option is grey, it means that the open environment does not support combination testing. Rebuild the environment to use this feature.

VCAST_DO_NOT_REDEFINE_MAIN

```
clicast -lc Option VCAST_DO_NOT_REDEFINE_MAIN True | False
```

Category: Builder Options

Description: Using this option will suppress the output #define main VCAST_main in a UUT harness file where the original source file contains the main routine.

VCAST_DUMP_BUFFER

```
clicast -lc Option VCAST_DUMP_BUFFER True | False
```

Category: Target Options

Description: If this option is set, VectorCAST will dump the test result data using a single printf call at the end of the test execution (or when the buffer is full). On some targets, reducing the output to a single "write" will result in a dramatic speed improvement.

VCAST_DUMP_COVERAGE_AT_EXIT

```
clicast -lc Option VCAST_DUMP_COVERAGE_AT_EXIT True | False
```

Category: Coverage Options

Description: When using Buffered Coverage I/O, setting this option forces the coverage data to get dumped to the TESTINSS.DAT file when the harness or instrumented application terminates. Note, destructor coverage data for global objects is included in the dump because the mechanism used is the system atexit() callback. For Cover environments, you no longer need to call vcast_dump_ coverage() from your source code when this option is on.

VCAST_EDG_PREPROCESS_FLAGS

```
clicast -lc Option VCAST_EDG_PREPROCESS_FLAGS <flags>
```

Category: Language Options

Description: Use this option to pass additional command-line arguments to the EDG preprocessor used when VCAST_USE_EDG_PREPROCESSOR is TRUE or when VectorCAST performs macro detection.

VCAST_EMPTY_STATEMENTS_COVERABLE

```
clicast -lc Option VCAST_EMPTY_STATEMENTS_COVERABLE True | False
```

Category: Coverage Options

Description: When this option is true, the empty statement, ';', is considered coverable.

VCAST_EMPTY_TESTCASE_FAIL

```
clicast -lc Option VCAST_EMPTY_TESTCASE_FAIL True | False
```

Category: Execute Options

Description: If this option is set, then empty test cases will be marked as failed.

VCAST_ENABLE_DATA_CLEAR_API

```
clicast -lc Option VCAST_ENABLE_DATA_CLEAR_API True | False
```

Category: Coverage Options

Description: Setting this option enables the VCAST_CLEAR_COVERAGE_DATA() function in the

harness or instrumented application. You can call this function while the instrumented application is running to clear the currently collected coverage data. This function can be combined with the VCAST_DUMP_COVERAGE_DATA() function in such a way that you can dump the coverage data, then clear it, and repeat the process until you've collected all the necessary coverage data.

VCAST_ENABLE_FUNCTION_AND_CALL_COVERAGE

```
clicast -lc Option VCAST_ENABLE_FUNCTION_AND_CALL_COVERAGE True | False
```

Category: Coverage Options

Description: 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.

VCAST_ENABLE_FUNCTION_CALL_COVERAGE

```
clicast -lc Option VCAST_ENABLE_FUNCTION_CALL_COVERAGE True | False
```

Category: Coverage Options

Description: 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.

VCAST_ENABLE_PROBE_POINTS

```
clicast -lc Option VCAST_ENABLE_PROBE_POINTS True | False
```

Category: Builder Options

Description: Set this option to enable the use of probe points.

VCAST_ENHANCED_FOR_LOOP_COVERAGE

```
clicast -lc Option VCAST_ENHANCED_FOR_LOOP_COVERAGE True | False
```

Category: Coverage Options

Description: When this option is true, 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. When this option is not set, only the True condition is checked.

VCAST_ENVIRONMENT_FILES

```
clicast -lc Option VCAST_ENVIRONMENT_FILES <files>
```

Category: Target Options

Description: These files will be copied into the environment directory during an environment build or rebuild.

VCAST_ESCAPE_LINE_DIRECTIVES

```
clicast -lc Option VCAST_ESCAPE_LINE_DIRECTIVES True | False
```

Category: Language Options

Description: Enable this option if your preprocessor does not escape backslashes in line directives in preprocessed files.

VCAST_EXECUTE_WITH_STDIO

```
clicast -lc Option VCAST_EXECUTE_WITH_STDIO
```

Category: Target Options

Description: This option tells VectorCAST to execute the test harness with standard input mapped to file: VCAST_STDIN.DAT, and standard output mapped to VCAST_STDOUT.DAT. The normal Execute Command is still used to execute the test.

VCAST_EXECUTE_WITH_STDOUT

```
clicast -lc Option VCAST_EXECUTE_WITH_STDOUT
```

Category: Target Options

Description: This option tells VectorCAST to compile input test data into the harness, and map standard output to VCAST_STDOUT.DAT. The normal Execute Command is still used to execute the test.

VCAST_EXPECTED_BEFORE_UUT_CALL

```
clicast -lc Option VCAST_EXPECTED_BEFORE_UUT_CALL True | False
```

Category: Report Options

Description: If a stub is called before the first UUT call, allow comparisons of expected values during the stub execution.

VCAST_FAR_STDIN_DATA

```
clicast -lc Option VCAST_FAR_STDIN_DATA True | False
```

Category: Target Options

Description: This option should be used when executing on small-memory targets if your compiler supports the __far segment modifier.

VCAST_FIELD_DOT_NOTATION

```
clicast -lc Option VCAST_FIELD_DOT_NOTATION True | False
```

Category: Report Options

Description: This option is used to tell VectorCAST's execution results and test case data listing reports to use field dot notation whenever a field name is printed.

VCAST_FILE_ENCODING

```
clicast -lc Option VCAST_FILE_ENCODING <encoding format>
```

Category: Builder Options

Description: Indicates what file encoding is used for source files being processed by VectorCAST. <encoding format> is one of the following:

WCEM=h for Hex ESC encoding

WCEM=u for Upper half encoding

WCEM=s for Shift-JIS encoding

WCEM=e for EUC encoding

WCEM=8 for UTF-8 encoding

WCEM=b for Brackets encoding

The default is empty, which means "Hex ESC encoding".

VCAST_FILE_INDEX

```
clicast -lc Option VCAST_FILE_INDEX True | False
```

Category: Target Options

Description: Select this option to enable VectorCAST to use a file indexing mode that significantly reduces file I/O required for running coverage or unit tests.

VCAST_FILE_PREFIX

```
clicast -lc Option VCAST_FILE_PREFIX <file prefix>
```

Category: Target Options

Description: This text will be prepended to any filename that the harness opens.

VCAST_FILE_VERSION_COMMAND

```
clicast -lc Option VCAST_FILE_VERSION_COMMAND <command>
```

Category: Report Options

Description: This command will be executed for each source file in the test environment, and a section will be added to the Full report and Test Case Management report 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 of the report will not be created.

VCAST_FLOAT_PRECISION

```
clicast -lc Option VCAST_FLOAT_PRECISION <integer number>
```

Category: Report Options

Description: Controls the number of digits used when the test harness prints floating point Actual Values for the Execution Report or Range Data. This option's setting is significant in the comparison of Expected Values for floating point numbers as well as determining the min and max values.

In a C/C++ environment, a non-zero precision value causes floating point numbers to print in the report with no more than the given number of significant digits, and a zero precision value usually results in a maximum of 6 digits.

VCAST_FORCE_ELAB_TYPE_SPEC

```
clicast -lc Option VCAST_FORCE_ELAB_TYPE_SPEC True | False
```

Category: Builder Options

Description: Set this option to tell VectorCAST to use elaborated type tags for all classes, structs, and unions. This is necessary when the type is hidden by a non-type entity but some compilers do not allow the tag with certain built-in types.

VCAST_FORCE_NO_USERGLOBALS

```
clicast -lc Option VCAST_FORCE_NO_USERGLOBALS True | False
```

Category: Target Options

Description: This option will omit the User Globals Unit from the test harness, which will greatly reduce the test harness size.

VCAST_FORCE_NOTSUPPORTED_PROC

```
clicast -lc Option VCAST_FORCE_NOTSUPPORTED_PROC True | False
```

Category: Builder Options

Description: This option attempts to reduce the size of the test harness by forcing all types to be considered NOT SUPPORTED. This requires User Code to enter test case data but will reduce a significant amount of code generation. (For embedded targets with limited code space.)

VCAST_FORCE_REPORT_DATES

```
clicast -lc Option VCAST_FORCE_REPORT_DATES True | False
```

Category: Report Options

Description: This option will force all the dates and times in the report configuration table and the management table to all be the same. This helps diff the reports much more easily.

VCAST_FREE_HARNESS_DATA

```
clicast -lc Option VCAST_FREE_HARNESS_DATA True | False
```

Category: Target Options

Description: By default, the C/C++ test harness does not automatically free memory allocated (via malloc) for test input data specified via the parameter tree. On most systems, this memory is automatically reclaimed by the OS when the harness process exits.

Enable this option on systems where explicit freeing is required prior to exit to make memory available for subsequent processes. Enabling this option also defines a harness function VCAST_Free_Allocated_Data() that can be called by user code to free harness data during test execution, which may be helpful in compound tests that would otherwise exceed memory limits.

This option has no effect when the system malloc is not in use, such as when VCAST_NO_MALLOC or VCAST_NO_STDLIB is enabled.

The default value for this option is False.

VCAST_FULL_STRINGS

```
clicast -lc Option VCAST_FULL_STRINGS True | False
```

Category: Execute Options

Description: When this option is set, all elements of a character array (including non-printable characters) are displayed using their numeric representation. As a result, the entire array will be shown (the NULL character does not terminate the string). Use the VCAST_HEX_NOTATION option to choose between octal and hexadecimal representation. Note: This option also affects how strings are compared during test case execution.

VCAST_FUNCTION_POINTER_SUPPORT

```
clicast -lc Option VCAST_FUNCTION_POINTER_SUPPORT True | False
```

Category: Builder Options

Description: Set this option to allow setting and checking function pointer and std::function variables and parameters in the test case parameter tree. When this option is disabled, such items can be set and checked only via user code.

VCAST_FUNCTION_PROBE_BEFORE_DECL

```
clicast -lc Option VCAST_FUNCTION_PROBE_BEFORE_DECL True | False
```

*Category:*Coverage Options

Description: 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, meaning insert the function probe in the default location, after the declarative section.

VCAST_FUNC_PTR_INLINES_C

```
clicast -lc option VCAST_FUNC_PTR_INLINES_C TRUE | FALSE
```

*Category:*Builder Options

Description: Enable this option to allow inline functions to be selectable for setting or checking function pointer parameters in C units when VCAST_FUNCTION_POINTER_SUPPORT is enabled. Disable this option if inline functions do not have a standalone definition whose address can be taken.

VCAST_GEN_ALL_EXECUTION_REPORT_FORMATS

```
clicast -lc Option VCAST_GEN_ALL_EXECUTION_REPORT_FORMATS True | False
```

Category: Report Options

Description: This option is used to determine whether or not to generate the execution reports in all the available formats (e.g. HTML or TEXT).

VCAST_GH_INT_FILE

```
clicast -lc Option VCAST_GH_INT_FILE <intex filename>
```

Category: Target Options

Description: 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 'Starit' line with the value 'true'.

VCAST_GH_INTEX_CMD

```
clicast -lc Option VCAST_GH_INTEX_CMD <intex command>
```

Category: Target Options

Description: When set, VectorCAST will invoke the Green Hills intex utility with this command immediately after linking the test harness. Refer to the VCAST_GH_INT_FILE environment variable for information on how to specify a custom integrate file. Example: intex -kernel=C:\GHS\int408\sim800\kernel -bspdir=C:\GHS\int408\sim800 -target=C:\GHS\int408\sim800\default.bsp vcast.int.

VCAST_GLOBAL_BUFFER_OPTIMIZATIONS

```
clicast -lc option VCAST_GLOBAL_BUFFER_OPTIMIZATIONS [True | False]
```

Category: Coverage Options

Description: 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.

VCAST_GNU_SYSTEM_MARKER

```
clicast -lc Option VCAST_GNU_SYSTEM_MARKER True | False
```

Category: Language Options

Description: This option controls whether VectorCAST inserts a GNU-style system header line marker at the start of instrumented files and other files containing expanded header content. Enable this option when using GNU-based or clang-based compilers and the compiler issues errors in VectorCAST-generated files for unmodified code expanded from a system header. The default value is False.

VCAST_HANDLE_TERNARY_OPERATOR

```
clicast -lc Option VCAST_HANDLE_TERNARY_OPERATOR True | False
```

Category: Coverage Options

Description: 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.

VCAST_HAS_LONGLONG

```
clicast -lc Option VCAST_HAS_LONGLONG True | False
```

Category: Builder Options

Description: Enable harness support for long long and unsigned long long types.

VCAST_HEADER_FILE_EXTENSIONS

```
clicast -lc Option VCAST_HEADER_FILE_EXTENSIONS <header file extensions>
```

Category: Language Options

Description: 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. To include headers with no extension, the value <<NO_EXTENSION>> may be used.

VCAST_HEX_NOTATION

```
clicast -lc Option VCAST_HEX_NOTATION True | False
```

Category: Execute Options

Description: If a string contains a non-printable character (excluding NULL), all characters are displayed using octal numeric representation. If this option is set, the characters will be displayed

using hexadecimal notation. To display the terminating NULL character and all following elements of a character array, use the VCAST_FULL_STRINGS option.

VCAST_IGNORE_ERROR_WHEN_READ_FOLLOWS_ADDRESS_OF

```
clicast -lc Option VCAST_HEX_NOTATION True | False
```

Category: Coverage Options

Description: When this option is enabled, data couple access errors for reads following an address of operation will be ignored. The default value is False.

VCAST_IGNORE_INCOMPLETE_TESTS

```
clicast -lc Option VCAST_IGNORE_INCOMPLETE_TESTS True | False
```

Category: Execute Options

Description: When building Basis Path or MC/DC test cases there are three outcomes for each test: Complete, Partial, and Template. When this option is on, VectorCAST will discard the Partial and Template tests, and only load the Complete tests.

VCAST_IGNORE_PSC_RAM_PAYLOAD_REBOOT_STATUS

```
clicast -lc Option VCAST_IGNORE_PSC_RAM_PAYLOAD_REBOOT_STATUS True | False
```

Category: Target Options

Description: This option tells VectorCAST to treat a non-zero target reboot status as a warning and to not require user interaction to continue.This option is only used with the vxWorks Platform for Safety Critical (PSC), and only if the VectorCAST options VCAST_PSC_USE_ADDRESS_SPACE and VCAST_PSC_RAM_PAYLOAD_REBOOT_CMD are also set.

VCAST_INHERITED_INLINES_ALWAYS_TESTABLE

```
clicast -lc Option VCAST_INHERITED_INLINES_ALWAYS_TESTABLE True | False
```

Category: Builder Options

Description: This option is used to determine whether or not inherited inline functions will always appear as testable in the test case tree. If enabled, inherited inlines will appear as testable even if they are not on the search list.

VCAST_INST_FILE_MAX_LINES

```
clicast -lc Option VCAST_INST_FILE_MAX_LINES <integer number>
```

Category: Builder Options

Description: Set this option if your debugger or compiler cannot handle source files larger than a certain size. If this option is non-zero, VectorCAST will split modified source files (created for code coverage, stub by function, or Visual Studio whitebox) that have greater than this many lines into smaller files that are #included together.

VCAST_INSTANTIATE_ALL_TEMPLATE_FUNCTIONS

```
clicast -lc Option VCAST_INSTANTIATE_ALL_TEMPLATE_FUNCTIONS True | False
```

Category: Language Options

Description: Set this option to cause VectorCAST to instantiate template functions that have been referenced and all member functions of template classes that have been referenced when parsing source files.

VCAST_INSTRUMENT_ASSIGNMENTS

```
clicast -lc Option VCAST_INSTRUMENT_ASSIGNMENTS True | False
```

Category: Coverage Options

Description: 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.

VCAST_INSTRUMENT_DEAD_CODE

```
clicast -lc option VCAST_INSTRUMENT_DEAD_CODE True | False
```

Category: Coverage Options

Description: 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.

VCAST_INSTRUMENT_PARAMETERS

```
clicast -lc Option VCAST_INSTRUMENT_PARAMETERS True | False
```

Category: Coverage Options

Description: Setting this option will cause parameters in function calls to be covered by branch and MC/DC coverage.

VCAST_INSTRUMENT_SEPARATES

```
clicast -lc Option VCAST_INSTRUMENT_SEPARATES True | False
```

Category: Coverage Options

Description: This option will cause separate source files which are included into the original source unit to be instrumented.

VCAST_LIBRARY_STUBS

```
clicast -lc Option VCAST_LIBRARY_STUBS <library functions>
```

Category: Builder Options

Description: List of library functions to stub by default (such as malloc) in the Create New Environment Wizard. If you add a library function to this list when an environment is open, you must enable it explicitly by going to the Update Environment dialog, Step 6 "Choose UUTs & Stubs". Select the Library Stubs tab (scroll to the right to find it) then check the box next to the library function to stub it in this environment.

VCAST_MAIN

```
clicast -lc Option VCAST_MAIN True | False
```

Category: Target Options

Description: This option changes the name of the 'main' function of the VectorCAST test harness from 'main' to 'vcast_main'. This is useful when your target environment requires some startup processing prior to the start of the test. If you set this option, then you must provide your own 'main()' that calls 'vcast_main()' as part of its processing.

VCAST_MASKING_MCDC

```
clicast -lc Option VCAST_MASKING_MCDC True | False
```

Category: Coverage Options

Description: Use masking MC/DC to determine pairs for C and C++ files instead of unique cause MC/DC.

VCAST_MAX_CAPTURED_ASCII_DATA

```
clicast -lc Option VCAST_MAX_CAPTURED_ASCII_DATA <integer number>
```

Category: Coverage Options

Description: When this option is unchecked, VectorCAST automatically calculates the buffer size for the ASCII data. This option should be increased only when the ASCII buffer overflows when using MCDC coverage. See the file INVALID_COVERAGE_LINES.LOG for more information.

VCAST_MAX_COVERED_SUBPROGRAMS

```
clicast -lc Option VCAST_MAX_COVERED_SUBPROGRAMS <integer number>
```

Category: Coverage Options

Description: 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.

VCAST_MAX_HEAP_SIZE

```
clicast -lc Option VCAST_MAX_HEAP_SIZE <integer number>
```

Category: Target Options

Description: When using the VectorCAST Heap in place of the syslib heap, This value controls the pre-allocated size of the heap.

VCAST_MAX_MCDC_ABORT_INST

```
clicast -lc Option VCAST_MAX_MCDC_ABORT_INST True | False
```

Category: Coverage Options

Description: While instrumenting with MC/DC, if a conditional statement contains more conditions than possible to instrument, then abort 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.

VCAST_MAX_MCDC_CONDITIONS

```
clicast -lc Option VCAST_MAX_MCDC_CONDITIONS <integer number>
```

Category: Coverage Options

Description: 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.

.

VCAST_MAX_MCDC_STATEMENTS

```
clicast -lc Option VCAST_MAX_MCDC_STATEMENTS <integer number>
```

Category: Coverage Options

Description: If 'Use static memory allocation' is True, VectorCAST must allocate a specific amount of space to store coverage data each time an MC/DC expression is encountered. This number tells VectorCAST how many MC/DC expressions for which it should set aside memory.

VCAST_MAX_STRING_LENGTH

```
clicast -lc Option VCAST_MAX_STRING_LENGTH <integer number>
```

Category: Target Options

Description: This specifies the size of temporary character arrays that VectorCAST creates in the test harness by setting the VCAST_MAX_STRING_LENGTH define. It also sets the maximum length of a string that can be used as an input value in the parameter tree. If you are running in a target environment with limited heap and stack resources, making this value smaller will reduce the VectorCAST test harness use of heap and stack. Changes to this value will take effect after the environment is recompiled.

VCAST_MAX_TABLE_SUBCONDITIONS

```
clicast -lc Option VCAST_MAX_TABLE_SUBCONDITIONS <integer number>
```

Category: Coverage Options

Description: 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.

VCAST_MAX_TARGET_FILES

```
clicast -lc Option VCAST_MAX_TARGET_FILES <integer number>
```

Category: Target Options

Description: This option limits the total number of files that the test harness is allowed to open while running a test. If set too low, test execution for that test does not start. This option is used for compound tests with more than 12 slots; it automatically sets the VCAST_MAX_FILES macro in the harness. Environment must be recompiled for changes to take effect.

VCAST_MICROSOFT_LONG_LONG

```
clicast -lc Option VCAST_MICROSOFT_LONG_LONG True | False
```

Category: Builder Options

Description: Enable harness support for Visual C++ __int64 syntax. VCAST_HAS_LONGLONG also needs to be set to True to use this option.

VCAST_MINIMAL_TERMINATION

```
clicast -lc Option VCAST_MINIMAL_TERMINATION True | False
```

Category: Target Options

Description: This option removes most of the clean-up processing normally performed at the end of the test harness 'main()'. Omitted processing includes closing of files, printing status messages and calling of 'exit()'. Omitting this processing will reduce the size of the harness. This option is only honored when using STDIO or STDOUT mode for harness I/O.

VCAST_MONITOR

```
clicast -lc Option VCAST_MONITOR True | False
```

Category: Target Options

Description: This option indicates that the VectorCAST host based monitor utility is being used to control the I/O to the target.

VCAST_MULTIBYTE_CHARACTERS

```
clicast -lc Option VCAST_MULTIBYTE_CHARACTERS True | False
```

Category: Builder Options

Description: Enable this option if source code files are encoded using Shift JIS.

VCAST_NO_EXIT

```
clicast -lc Option VCAST_NO_EXIT True | False
```

Category: Target Options

Description: This option disables the trapping of calls by the code under test to the syslib 'exit()' function which will reduce the size of the harness.

VCAST_NO_FFLUSH

```
clicast -lc Option VCAST_NO_FFLUSH True | False
```

Category: Target Options

Description: This option indicates that the stdio.h function fflush, is not defined for the compiler you are using.

VCAST_NO_FLOAT

```
clicast -lc Option VCAST_NO_FLOAT True | False
```

Category: Target Options

Description: This option disables type processing for 'float' types in order to reduce the size of the harness. User code is required to set and check values of parameters, returns, and global objects.

VCAST_NO_LIMITS

```
clicast -lc Option VCAST_NO_LIMITS True | False
```

Category: Target Options

Description: This option should be turned on if your compiler does not have a 'limits.h' header file.

VCAST_NO_LONG_DOUBLE

```
clicast -lc Option VCAST_NO_LONG_DOUBLE True | False
```

Category: Builder Options

Description: This option indicates that your compiler does not support the long double type.

VCAST_NO_MALLOC

```
clicast -lc Option VCAST_NO_MALLOC True | False
```

Category: Target Options

Description: This option removes the harness dependency on syslib 'malloc()'.

VCAST_NO_SETJMP

```
clicast -lc Option VCAST_NO_SETJMP True | False
```

Category: Target Options

Description: This option indicates that your compiler does not provide a stdjmp.h file.

VCAST_NO_SIGNAL

```
clicast -lc Option VCAST_NO_SIGNAL True | False
```

Category: Target Options

Description: This option should be turned on if your compiler does not have a 'signal.h' header file.

VCAST_NO_SPRINTF

```
clicast -lc Option VCAST_NO_SPRINTF True | False
```

Category: Target Options

Description: This option directs the C/C++ unit test harness to build with the macro VCAST_NO_ SPRINTF. Set this option to True when the library function **`sprintf()`** is not defined in the compiler being used. The default value is False.

VCAST_NO_STANDARD_PKG_USAGE

```
clicast -lc Option VCAST_NO_STANDARD_PKG_USAGE True | False
```

Category: Builder Options

Description: By default, the Multiunit Whitebox harness redefines TRUE and FALSE by using STANDARD.BOOLEAN. This option disables that override, necessary when user source files use 'STANDARD' in a different context.

VCAST_NO_STDIN

```
clicast -lc Option VCAST_NO_STDIN True | False
```

Category: Target Options

Description: This option indicates that you are running tests on a target board that does not have 'STDIN' capability. In this case, VectorCAST will compile and link the test case data into the test harness, so that no data has to be 'read' by the test harness.

VCAST_NO_STDLIB

```
clicast -lc Option VCAST_NO_STDLIB True | False
```

Category: Target Options

Description: This option should be turned on if your compiler does not have a 'stdlib.h' header file.

VCAST_NO_STD_FILES

```
clicast -lc Option VCAST_NO_STD_FILES True | False
```

Category: Target Options

Description: This option indicates that the standard file handles STDIN, STDOUT, and STDERR, are not defined for the compiler you are using.

VCAST_NO_TYPE_SUPPORT

```
clicast -lc Option VCAST_NO_TYPE_SUPPORT True | False
```

Category: Target Options

Description: This option disables ALL type processing in order to reduce the size of the harness. User code is required to set all data items. Only use on VERY small targets (less than 32k program space / 2k RAM) Setting this option automatically sets the options to omit BitField, String, and Float Types.

VCAST_NOTES_SECTION_TEMPLATE

```
clicast -lc Option VCAST_NOTES_SECTION_TEMPLATE <Path to Template text
file>
```

Category: Report Options

Description: This option allows you to enter a path to a text file that contains a template for the Notes

section of the test case.

VCAST_OLD_STYLE_MANAGEMENT_REPORT

```
clicast -lc Option VCAST_OLD_STYLE_MANAGEMENT_REPORT True | False
```

Category: Report Options

Description: When set, this option reverts the Testcase Management Report to the behavior found in VectorCAST version 6.0 and prior, in which the main body of the report combined the Expecteds and Control Flow in one fraction in the "Pass/Fail" column.

VCAST_ONLY_SHOW_GLOBAL_OBJECTS_IN_ONE_UNIT

```
clicast -lc Option VCAST_ONLY_SHOW_GLOBAL_OBJECTS_IN_ONE_UNIT True | False
```

Category: Builder Options

Description: By default, global objects are shown in the GUI under every unit in which they are accessible. When this option is enabled, global objects will only be displayed from the first unit in which they are referenced.

VCAST_OPTIMIZED_MCDC_STORAGE_THRESHOLD

```
clicast -lc Option VCAST_OPTIMIZED_MCDC_STORAGE_THRESHOLD <integer number>
```

Category: Coverage Options

Description: 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.

VCAST_OUTPUT_BUFFER_SIZE

```
clicast -lc Option VCAST_OUTPUT_BUFFER_SIZE <integer number>
```

Category: Builder Options

Description: Size of buffer allocated for test results in the harness. This value is used if VCAST_ BUFFER_OUTPUT is set.

VCAST_PACK_INSTRUMENTATION_STORAGE

```
clicast -lc Option VCAST_PACK_INSTRUMENTATION_STORAGE True | False
```

Category: Coverage Options

Description: 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.

Enabling or disabling this option requires all source code to be re-instrumented. The default value is True, meaning to use a bit array.

VCAST_PARSE_FUNCTION_TEMPLATES

```
clicast -lc Option VCAST_PARSE_FUNCTION_TEMPLATES True | False
```

Category: Language Options

Description: Process 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. The default value is True.

VCAST_PASS_SUBPROGRAM_NAME_TO_UC

```
clicast -lc Option VCAST_PASS_SUBPROGRAM_NAME_TO_UC True | False
```

Category: Builder Options

Description: This option will cause the test harness to pass the unit and subprogram names as a string literal to the vCAST_STUB_PROCESSING functions. An environment rebuild is necessary for this option to take effect.

VCAST_POST_PREPROCESS_COMMAND

```
clicast -lc Option VCAST_POST_PREPROCESS_COMMAND <command>
```

Category: Language Options

Description: 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.

VCAST_POST_RUN_DELAY

```
clicast -lc Option VCAST_POST_RUN_DELAY <integer number>
```

Category: Target Options

Description: Number of seconds to delay after execution completes before result processing begins.

VCAST_PRE_EXECUTE_CMD

```
clicast -lc Option VCAST_PRE_EXECUTE_CMD <command to execute before test
execution>
```

Category: Target Options

Description: The specified executable or batch file script is executed before the test harness is called during test case execution. Useful for rebooting your target, for example. If a relative path to this command is used, it should be relative to the environment directory.

VCAST_PRECOMPILE_SCRIPT

```
clicast -lc Option VCAST_PRECOMPILE_SCRIPT <script>
```

Category: Builder Options

Description: This command is executed prior to full compilations of the test harness. It can be used to make modifications to harness files before the initial compilation. Note that this command may also be called for subsequent harness compilations, but it is not called before every compilation.

VCAST_PREPEND_TO_PATH_DIRS

```
clicast -lc Option VCAST_PREPEND_TO_PATH_DIRS <path>
```

Category: Builder Options

Description: These directories will be prepended to the PATH environment variable value. This can be used to put your compiler on the PATH. Multiple directories should be separated by ';' (Windows) or ':' (Linux). Note that some compilers require other environment variables to be set as well in order for the compiler to work properly.

VCAST_PREPROCESS_PREINCLUDE

```
clicast -lc Option VCAST_PREPROCESS_PREINCLUDE <file path>
```

Category: Language Options

Description: File will be added as a #include prior to source files during preprocessing.

VCAST_PROBE_POINTS_AS_EVENTS

```
clicast -lc Option VCAST_PROBE_POINTS_AS_EVENTS True | False
```

Category: Report Options

Description: Set this option to have an event generated at each probe point. The default value is False.

VCAST_PSC_RAM_PAYLOAD

```
clicast -lc Option VCAST_PSC_RAM_PAYLOAD True | False
```

Category: Target Options

Description: This option is only used with the vxWorks Platform for Safety Critical (PSC), and only if the VectorCAST option: VCAST_PSC_USE_ADDRESS_SPACE is also set. This option tells VectorCAST that it should that it can run a test case by rebooting the PSC address space, via the partitionModeSet command. If this option is NOT set, then VectorCAST will start the partition using the arincSchedSet command to run each test case.

VCAST_PSC_RAM_PAYLOAD_REBOOT_CMD

```
clicast -lc Option VCAST_PSC_RAM_PAYLOAD_REBOOT_CMD <command to reboot
target before execution>
```

Category: Target Options

Description: This option tells VectorCAST what command to issue to reboot the target hardware when it is required to do so.This option is only used with the vxWorks Platform for Safety Critical (PSC), and only if the VectorCAST option VCAST_PSC_USE_ADDRESS_SPACE is also set.

VCAST_PSC_USE_ADDRESS_SPACE

```
clicast -lc Option VCAST_PSC_USE_ADDRESS_SPACE True | False
```

Category: Target Options

Description: This option is only used with the vxWorks Platform for Safety Critical (PSC) This option causes VectorCAST to build a monolithic boot image for the target, with the test harness located in an address space, as opposed to the coreOS.

VCAST_RECURSIVE_DIRECTORY_ADD_LIMIT

```
clicast -lc Option VCAST_RECURSIVE_DIRECTORY_ADD_LIMIT <integer number>
```

Category: Builder Options

Description: The maximum allowed increment of directories that can be added recursively.When this limit is reached, you have the choice to add more directories or stop.

VCAST_REFERENCED_GLOBALS

```
clicast -lc Option VCAST_REFERENCED_GLOBALS True | False
```

Category: Execute Options

Description: When this option is on, the VectorCAST test case editor will display only those global objects and stubs that are referenced by the function under test.

VCAST_RELATIVE_INCLUDES

```
clicast -lc Option VCAST_RELATIVE_INCLUDES True | False
```

Category: Builder Options

Description: Setting this option indicates that relative paths to included files should be maintained.

VCAST_RELINK_PROMPT_FOR_PSC_RAM_PAYLOAD

```
clicast -lc Option VCAST_RELINK_PROMPT_FOR_PSC_RAM_PAYLOAD True | False
```

Category: Target Options

Description: By default, when the RAM Payload partition flag is set, VectorCAST relinks the environment upon opening to ensure that the target boot image matches the current environment. If this option is set to TRUE, VectorCAST will prompt the user before performing these steps. If the user knows the target is properly loaded, they can skip this step.

VCAST_REMOVE_PREPROCESSOR_COMMENTS

```
clicast -lc Option VCAST_REMOVE_PREPROCESSOR_COMMENTS True | False
```

Category: Language Options

Description: 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.

VCAST_REPORT_ON_IMPLICIT_DEFAULT_CASE

```
clicast -lc Option VCAST_REPORT_ON_IMPLICIT_DEFAULT_CASE True | False
```

Category: Coverage Options

Description: 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.

VCAST_REPOSITORY

```
clicast -lc Option VCAST_REPOSITORY <directory path>
```

Category: Language Options

Description: This is the path to a directory containing a repository of settings.

VCAST_RPTS_AUTOMATIC_BGCOLOR

```
clicast -lc Option VCAST_RPTS_AUTOMATIC_BGCOLOR <color>
```

Category: Report Options

Description: This color is used for text indicating automatic initialization and finalization tests.

VCAST_RPTS_BODY_BGCOLOR

```
clicast -lc Option VCAST_RPTS_BODY_BGCOLOR <color>
```

Category: Report Options

Description: Primary background color for HTML reports.

VCAST_RPTS_COMPLEXITY_COLUMN_WIDTH

```
clicast -lc Option VCAST_RPTS_COMPLEXITY_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of complexity columns in text reports.

VCAST_RPTS_COVERAGE_RESULT_COLUMN_WIDTH

```
clicast -lc Option VCAST_RPTS_COVERAGE_RESULT_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of coverage result columns in text reports.

VCAST_RPTS_CUSTOM_CONFIG

```
clicast -lc Option VCAST_RPTS_CUSTOM_CONFIG <config text> | <config file>
```

Category: Report Options

Description: Custom report configuration. If this option points to an existing file, then the contents of that file is used as the custom report configuration, otherwise the contents of the option is used.

VCAST_RPTS_CUSTOM_CSS

```
clicast -lc Option VCAST_RPTS_CUSTOM_CSS <css>
```

Category: Report Options

Description: 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.

VCAST_RPTS_DATE_COLUMN_WIDTH

```
clicast -lc Option VCAST_RPTS_DATE_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of date columns in text reports.

VCAST_RPTS_DEFAULT_FONT_COLOR

```
clicast -lc Option VCAST_RPTS_DEFAULT_FONT_COLOR <color>
```

Category: Report Options

Description: Default text color for HTML reports.

VCAST_RPTS_DEFAULT_FONT_FACE

```
clicast -lc Option VCAST_RPTS_DEFAULT_FONT_FACE <font>
```

Category: Report Options

Description: Default font for HTML reports.

VCAST_RPTS_DEFAULT_FONT_SIZE

```
clicast -lc Option VCAST_RPTS_DEFAULT_FONT_SIZE <font size>
```

Category: Report Options

Description: Default font size for HTML reports.

VCAST_RPTS_EMPTY_DATA_STRING

```
clicast -lc Option VCAST_RPTS_EMPTY_DATA_STRING <string>
```

Category: Report Options

Description: Text to show in blank cells in the Execution and Test Case Data report sections. The default value is unset.

VCAST_RPTS_DELIMITER

```
clicast -lc Option VCAST_RPTS_DELIMITER <delimiter>
```

Category: Report Options

Description: Character separator used in two CLICAST report commands: cover report cvs_metrics and reports alternate. To enter a tab, use \t.

Valid delimiters are ? , ' ; | { } [ ] @ ~ # $ _

VCAST_RPTS_FAIL_COLOR

```
clicast -lc Option VCAST_RPTS_FAIL_COLOR <color>
```

Category: Report Options

Description: This color is used for text indicating failing test cases or failing totals.

VCAST_RPTS_HEADER

```
clicast -lc Option VCAST_RPTS_HEADER <text to prepend to title> | <header
text> | <html file>
```

Category: Report Options

Description: 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 ** ** 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>
```

VCAST_RPTS_HEADING_FONT_SIZE

```
clicast -lc Option VCAST_RPTS_HEADING_FONT_SIZE <font size>
```

Category: Report Options

Description: Heading font size for HTML reports.

VCAST_RPTS_NOTES_COLUMN_WIDTH

```
clicast -lc Option VCAST_RPTS_NOTES_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of notes columns in text reports.

VCAST_RPTS_PASS_COLOR

```
clicast -lc Option VCAST_RPTS_PASS_COLOR <color>
```

Category: Report Options

Description: This color is used for text indicating passing test cases or passing totals.

VCAST_RPTS_PRETTY_PRINT_HTML

```
clicast -lc Option VCAST_RPTS_PRETTY_PRINT_HTML True | False
```

Category: Report Options

Description: This option enables whitespace indentation in HTML and XML report-related files. It is off by default to allow better performance and reduced memory usage.

VCAST_RPTS_REQUIREMENTS_COLUMN_WIDTH

```
clicast -lc Option VCAST_RPTS_REQUIREMENTS_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of requirements columns in text reports.

VCAST_RPTS_RESULT_COLUMN_WIDTH

```
clicast -lc Option VCAST_RPTS_RESULT_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of result columns in text reports. The default value is 16.

VCAST_RPTS_SECTION_TITLE_BGCOLOR

```
clicast -lc Option VCAST_RPTS_SECTION_TITLE_BGCOLOR <color>
```

Category: Report Options

Description: Background color for section titles in HTML reports.

VCAST_RPTS_SELF_CONTAINED

```
clicast -lc Option VCAST_RPTS_SELF_CONTAINED True | False
```

Category: Report Options

Description: When True, HTML reports are created as a single, self-contained file with an embedded stylesheet and embedded images (for all built-in, non user-modified reports). Set the option to True (default value) when planning to email reports. When False, HTML reports are created as multiple files, storing stylesheets and images in the same directory as the report. Set the option to False when serving reports from a webserver.

VCAST_RPTS_SHOW_VERSION

```
clicast -lc Option VCAST_RPTS_SHOW_VERSION True | False
```

Category: Report Options

Description: Show the VectorCAST version in the configuration section of the reports.

VCAST_RPTS_SUBPROGRAM_COLUMN_WIDTH

```
clicast -lc Option VCAST_RPTS_SUBPROGRAM_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of subprogram columns in text reports.

VCAST_RPTS_TABLE_BORDER_SIZE

```
clicast -lc Option VCAST_RPTS_TABLE_BORDER_SIZE <border size>
```

Category: Report Options

Description: Table border size for tables in HTML reports.

VCAST_RPTS_TABLE_CELL_PADDING

```
clicast -lc Option VCAST_RPTS_TABLE_CELL_PADDING <padding>
```

Category: Report Options

Description: Table cell padding for tables in HTML reports.

VCAST_RPTS_TABLE_DATA_BGCOLOR

```
clicast -lc Option VCAST_RPTS_TABLE_DATA_BGCOLOR <color>
```

Category: Report Options

Description: Background color for table data in HTML reports.

VCAST_RPTS_TABLE_DATA_FAIL_BGCOLOR

```
clicast -lc Option VCAST_RPTS_TABLE_DATA_FAIL_BGCOLOR <color>
```

Category: Report Options

Description: This background color is used for failing data cells in reports and tables.

VCAST_RPTS_TABLE_DATA_PARTIAL_BGCOLOR

```
clicast -lc Option VCAST_RPTS_TABLE_DATA_PARTIAL_BGCOLOR <color>
```

Category: Report Options

Description: This background color is used for partially passing data cells in reports and tables, as well as warning messages in reports.

VCAST_RPTS_TABLE_DATA_PASS_BGCOLOR

```
clicast -lc Option VCAST_RPTS_TABLE_DATA_PASS_BGCOLOR <color>
```

Category: Report Options

Description: This background color is used for passing data cells in reports and tables.

VCAST_RPTS_TABLE_DATA_TEXT_ALIGNMENT

```
clicast -lc Option VCAST_RPTS_TABLE_DATA_TEXT_ALIGNMENT left | right |
center
```

Category: Report Options

Description: Text alignment (left, right, or center) for table data in HTML reports.

VCAST_RPTS_TABLE_HEADING_BGCOLOR

```
clicast -lc Option VCAST_RPTS_TABLE_HEADING_BGCOLOR <color>
```

Category: Report Options

Description: Background color for table headings in HTML reports.

VCAST_RPTS_TESTCASE_COLUMN_WIDTH

```
clicast -lc Option VCAST_RPTS_TESTCASE_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of testcase columns in text reports.

VCAST_RPTS_UNIT_COLUMN_WIDTH

```
clicast -lc Option VCAST_RPTS_UNIT_COLUMN_WIDTH <width>
```

Category: Report Options

Description: Width (in characters) of unit columns in text reports.

VCAST_RPTS_USER_REPORTS_DIR

```
clicast -lc Option VCAST_RPTS_USER_REPORTS_DIR <directory>
```

Category: Report Options

*Description:*This option points to a directory containing additional user reports.

VCAST_RPTS_WRAP_NOTES

```
clicast -lc Option VCAST_RPTS_WRAP_NOTES True | False
```

Category: Report Options

Description: This option will cause the text in the notes section to wrap at the first space before 80 characters.

VCAST_SBF_LIBRARY_STUBS

```
clicast -lc Option VCAST_SBF_LIBRARY_STUBS True | False
```

Category: Builder Options

Description: Enable this option to allow some library stubs to be dynamically stubbable. This option applies even without SBF units. Note that library stubs for member functions and variadic functions are not affected by this option, and are never dynamically stubbable.

VCAST_SBF_NONTESTABLE_FUNCTIONS

```
clicast -lc Option VCAST_SBF_NONTESTABLE_FUNCTIONS True | False
```

Category: Builder Options

Description: Enabling this option allows dynamic stubbing of functions which are defined in an SBF unit but are not testable. For example, depending on other options, these may include functions that are inline, static, or private.

VCAST_SBF_TEMPLATES

```
clicast -lc Option VCAST_SBF_TEMPLATES True | False
```

Category: Builder Options

Description: Enabling this option allows dynamic stubbing of template functions.

VCAST_SCRIPT_ALWAYS_USES_PARAMS

```
clicast -lc Option VCAST_SCRIPT_ALWAYS_USES_PARAMS True | False
```

Category: Execute Options

Description: When True, this option causes C++ function parameter and return types to always be included with the function name in test scripts. The default value is False, which causes parameterized names to only be used when a function is treated as overloaded.

The True value is useful mainly for tools that generate test scripts independently of an environment, when it isn't certain whether a function will be treated as overloaded. Note that toggling this option can lead to import failures with test scripts that were generated before the option was toggled.

VCAST_SCRIPT_EXPAND_ARRAYS

```
clicast -lc Option VCAST_SCRIPT_EXPAND_ARRAYS True | False
```

Category: Execute Options

Description: By default, when an array of scalars is written to a test script, identical values are condensed to one script line. By setting this option, each array element will get its own script line.

VCAST_SCRIPT_IN_CREATION_ORDER

```
clicast -lc Option VCAST_SCRIPT_IN_CREATION_ORDER True | False
```

Category: Execute Options

Description: By default, test scripts are exported in alphabetical order within unit and subprogram. Setting this option will cause the script to be generated in the order that the test cases were created.

VCAST_SCRIPT_LOG_SHOW_ONLY_ERRORS

```
clicast -lc Option VCAST_SCRIPT_LOG_SHOW_ONLY_ERRORS True | False
```

Category: Report Options

Description: In script logs for large test scripts, error messages can be overwhelmed by the number of status messages. Setting this flag prevents normal status messages from being added to the log - only error messages remain.

VCAST_SHOW_INLINES_COVERED_IN_ALL_UNITS

```
clicast -lc Option VCAST_SHOW_INLINES_COVERED_IN_ALL_UNITS True | False
```

Category: Coverage Options

Description: Enabling this option will cause inline functions that appear in multiple units to show the same coverage in each.

VCAST_SHOW_ONLY_DATA_WITH_EXPECTED_VALUES

```
clicast -lc Option VCAST_SHOW_ONLY_DATA_WITH_EXPECTED_VALUES True | False
```

Category: Report Options

Description: When this option is set, global and parameter data will only be displayed in the test execution report if expected values are provided. This option will result in smaller execution reports and faster test execution in cases where there are is a lot of global and/or parameter data input.

VCAST_SHOW_ONLY_EVENTS_WITH_EXPECTED_VALUES

```
clicast -lc Option VCAST_SHOW_ONLY_EVENTS_WITH_EXPECTED_VALUES True | False
```

Category: Report Options

Description: When this option is set, events will only be displayed in the test execution report if expected values exist for that event. This option will result in smaller execution reports in cases where there are is a lot of global and/or parameter data input.

VCAST_SHOW_STDOUT_CONSOLE

```
clicast -lc Option VCAST_SHOW_STDOUT_CONSOLE True | False
```

Category: Execute Options

Description: When this option is set, a new console will be opened during test harness execution to facilitate Standard I/O. If 'Redirect standard output' or 'Redirect standard error' is set, this option is ignored. (Windows Only)

VCAST_SIMPLIFIED_CONDITION_COVERAGE

```
clicast -lc Option VCAST_SIMPLIFIED_CONDITION_COVERAGE True | False
```

Category: Coverage Options

Description: 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.

VCAST_SORT_METRICS_RPT_BY_DIR

```
clicast -lc Option VCAST_SORT_METRICS_RPT_BY_DIR <True | False>
```

Category: Report Options

Description: Sort Metrics report by directory.

VCAST_SPLIT_UC_FILES

```
clicast -lc Option VCAST_SPLIT_UC_FILES True | False
```

Category: Builder Options

Description: Enabling this option causes test-specific user code to be inserted into a separate unit in the harness. To allow visibility to UUT code, the separate unit #includes an interface file that VectorCAST generates based on the original UUT code with function and object definitions transformed into declarations. This increases the complexity of the harness and the environment building process but can speed up user code compilation in large environments. This option is compatible with "Traditional Unit Testing" and "Test-Driven Development" Testing Methods only. Note that when user code is inserted into a separate unit, user code references to entities with internal (static) linkage will be references to different instances than are used by a UUT.

VCAST_STARTUP_FILE

```
clicast -lc Option VCAST_STARTUP_FILE <file name>
```

Category: Language Options

*Description:*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.

VCAST_STDIO

```
clicast -lc Option VCAST_STDIO True | False
```

Category: Target Options

Description: This option indicates that you are running tests on a target board and do not have file I/O capability. In this case, STDIN and STDOUT will be used to perform I/O. VectorCAST will read all of the input data from stdin and write all of the output to stdout. The stdin of the target should be mapped to the file: VCAST_STDIN.DAT, and the stdout should be mapped to VCAST_STDOUT.DAT.

VCAST_STRICT_TEST_CASE_IMPORT

```
clicast -lc Option VCAST_STRICT_TEST_CASE_IMPORT True | False
```

Category: Execute Options

Description: Normally, if there are errors when importing a test script, VectorCAST will ignore any object errors. When this option is selected, errors in the script will cause the testcase to not be allowed to execute until the script is fixed or until the testcase is edited.

VCAST_STRIP_SYS_HEADER_PATH

```
clicast -lc Option VCAST_STRIP_SYS_HEADER_PATH <path>
```

Category: Language Options

Description: VectorCAST preprocesses source files before adding instrumentation, which results in header content appearing in instrumented files. Some compilers treat code from system headers differently from non-header code, so to avoid problems when header content is part of the instrumented file, VectorCAST will "collapse" some headers back to a #include directive. This #include directive will normally use the full path to the collapsed header. Some compilers will treat the header differently because of the full path in the #include directive, so this option can be used to specify a directory path prefix which VectorCAST will remove from the #include, leaving a relative path or file name.

VCAST_STUB_ALL_SPECIAL_FUNCTIONS

```
clicast -lc Option VCAST_STUB_ALL_SPECIAL_FUNCTIONS True | False
```

Category: Builder Options

Description: This option causes stubs to be built for all special functions. That is, always stub all constructors, destructors, virtual functions, and overloaded operators.

VCAST_STUB_ALL_SUBPROGRAMS

```
clicast -lc Option VCAST_STUB_ALL_SUBPROGRAMS True | False
```

Category: Builder Options

Description: This option causes stubs to be built for all function prototypes which are not defined in the uut or stub by implementation units. By default, only those functions which are called get stubbed. This option is often necessary when the code under test makes use of function pointers.

VCAST_STUB_BY_IMPLEMENTATION

```
clicast -lc Option VCAST_STUB_BY_IMPLEMENTATION True | False
```

Category: Builder Options

Description: Create stubs based on actual source code files instead of just their header files.

VCAST_STUB_WITHOUT_PROTOTYPE

```
clicast -lc Option VCAST_STUB_WITHOUT_PROTOTYPE True | False
```

Category: Builder Options

Description: Enable this option to allow automatic stubbing of called functions that are not declared with a prototype. Since the function return and parameter types can only be inferred, the resulting stub might have a different protoype than the real function. More significantly, undeclared functions normally defined by external libraries could inadvertently be stubbed, resulting in either multiple definition errors or unexpected behavior.

Disable this option to prevent automatic stubbing or these functions, or #include the proper header so that VectorCAST can identify the declaration location. With this option disabled, the best way to stub undeclared functions is to declare or #include the proper prototype, either in the original code or in Unit Appendix User Code. Alternatively, a function can be specified in Additional Stubs in the the Environment Wizard. The default value is False.

VCAST_SUBDIRS_OF_SEARCH_LIST_CONSIDERED

```
clicast -lc Option VCAST_SUBDIRS_OF_SEARCH_LIST_CONSIDERED True | False
```

Category: Builder Options

Description: When enabled, sub-directories inherit the types of their parent directories, unless they are explicitly given other types. Sub-directories of library include directories are considered library include directories, sub-directories of type-handled directories are type-handled, and sub-directories of testable directories are testable.

VCAST_SUPPRESS_COVERABLE_FUNCTIONS

```
clicast -lc Option VCAST_SUPPRESS_COVERABLE_FUNCTIONS <suppressed coverable
functions>
```

*Category:*Coverage Options

Description: To suppress files or functions from being instrumented, enter lines 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.

Changes to this option require reinstrumentation to take effect.

VCAST_SUPPRESS_IMPORT_FILE_CHECKS

```
clicast -lc Option VCAST_SUPPRESS_IMPORT_FILE_CHECKS True | False
```

Category: Builder Options

Description: This option overrides the timestamp check on source files when importing coverage results. Timestamp checks are done only on older versions of vcp cover projects and cvr files which did not use checksums. You should only use this if you are working with older results files and you are sure the source files match.

VCAST_TEMPLATE_STRUCT_AS_CLASS

```
clicast -lc Option VCAST_TEMPLATE_STRUCT_AS_CLASS True | False
```

Category: Builder Options

Description: When this option is TRUE, all template-based struct types will be treated like classes that use 'new' to allocate and for which generated harness objects will always be pointers. When this option is FALSE, template-based struct types will be treated like classes only when they contain declared or compiler-generated constructors, destructors, or virtual functions, or when VectorCAST detects an instantiation will not compile. The default value is True.

VCAST_TEST_ALL_INLINES

```
clicast -lc Option VCAST_TEST_ALL_INLINES True | False
```

Category: Builder Options

Description: This option is used to tell VectorCAST all in-lined member functions found in a header file on the search list should be testable.

VCAST_TEST_ALL_NON_MEMBER_INLINES

```
clicast -lc Option VCAST_TEST_ALL_NON_MEMBER_INLINES True | False
```

Category: Builder Options

Description: This option is used to tell VectorCAST all non-member unlined functions found in a header file on the search list should be testable.

VCAST_TEST_UNIT_USER_CODE

```
clicast -lc Option VCAST_TEST_UNIT_USER_CODE True | False
```

Category: Builder Options

Description: Enable this option to allow functions defined in unit prefix user code or unit appendix user code to be testable. Note that functions in such user code do not get coverage instrumentation, even when this option is enabled.

VCAST_TEST_VALUES_DICTIONARY

```
clicast -lc option VCAST_TEST_VALUES_DICTIONARY <path to test values
dictionary file>
```

Category: Execute Options

Description: Path to the test values dictionary file.

VCAST_TESTCASE_FAIL_ON_NO_EXP_RETURN

```
clicast -lc option VCAST_TESTCASE_FAIL_ON_NO_EXP_RETURN True | False
```

Category: Execute Options

Description: If this option is set, then testcases that have no expected return value or expected user code for the return value will be marked as failed. The default value is FALSE. Note that this option is specific to the return value of the subprogram under test only.

VCAST_TESTCASE_FAIL_ON_NO_EXPECTED

```
clicast -lc Option VCAST_TESTCASE_FAIL_ON_NO_EXPECTED True | False
```

Category: Execute Options

Description: If this option is set, any test cases that have no expected results or expected user code will be marked as failed. Its default value is FALSE.

VCAST_TI_IGNORE_QUALIFIERS

```
clicast -lc Option VCAST_TI_IGNORE_QUALIFIERS True | False
```

Category: Builder Options

Description: This option allows the testing of objects whose types normally prohibit testing because of qualifiers (i.e. const or volatile, etc).When this option is enabled, type qualifiers are ignored in the test harness, allowing the testing of objects that use these prohibitive types. When the option is disabled, the objects that use the prohibitive type require user code to test.

VCAST_TORNADO_CONSTRUCTOR_CALL_FILE

```
clicast -lc Option VCAST_TORNADO_CONSTRUCTOR_CALL_FILE True | False
```

Category: Target Options

Description: This option is used to tell VectorCAST to create a constructor call source file. This is only for use with the WindRiver Tornado or Workbench compiler with C++. If this option is not set, global class objects will not be constructed properly.

VCAST_TREAT_DC_INITIALIZATIONS_AS_WRITE

```
clicast -lc Option VCAST_TREAT_DC_INITIALIZATIONS_AS_WRITE True | False
```

*Category:*Coverage Options

Description: 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.

VCAST_TYPEOF_OPERATOR

```
clicast -lc Option VCAST_TYPEOF_OPERATOR True | False
```

Category: Target Options

Description: This option indicates that your compiler supports the typeof operator. The default value is determined by the compiler template.

VCAST_UNCOVERED_LINE_INDICATOR

```
clicast -lc Option VCAST_UNCOVERED_LINE_INDICATOR <'#' | '$' | '%' | '&' |
'+' | ' '>
```

Category: Coverage Options

Description: Character to be used to indicate an uncovered statement in a coverage report. Valid coverage indicators are:

$ % & +

VCAST_UNEXPECTED_EXCEPTIONS_FAIL

```
clicast -lc Option VCAST_UNEXPECTED_EXCEPTIONS_FAIL True | False
```

Category: Execute Options

Description: If this option is set, then if an exception is raised during the execution of a testcase and no exception was expected, the testcase is marked as failed.

VCAST_UNEXPECTED_SIGNALS_FAIL

```
clicast -lc Option VCAST_UNEXPECTED_SIGNALS_FAIL True | False
```

Category: Execute Options

Description: If this option is set, then if a signal is raised during the execution of a testcase then

testcase is marked as failed.

VCAST_UNIT_TYPE

```
clicast -lc Option VCAST_UNIT_TYPE UUT | SBF
```

Category: Builder Options

Description: Default setting for Unit type in Create New Environment Wizard. UUT allows stubbing only for functions external to unit being tested; Stub By Function (SBF) allows stubbing of functions within the unit being tested.

VCAST_UNSIGNED_LONG_MCDC_STORAGE

```
clicast -lc Option VCAST_UNSIGNED_LONG_MCDC_STORAGE True | False
```

Category: Coverage Options

Description: MC/DC instrumentation will only occur for up to 31 subconditions in C and C++ files.

VCAST_USE_BUFFERED_ASCII_DATA

```
clicast -lc Option VCAST_USE_BUFFERED_ASCII_DATA True | False
```

Category: Coverage Options

Description: Enable this option to cause VectorCAST to store the coverage data in ASCII format in memory while running the test harness or instrumented application. When using this option with VectorCAST/Cover, the coverage data is not written to the TESTINSS.DAT file; you will need to read the in-memory buffer itself. In addition, if you have the option Dump buffered coverage data on exit on, then you will need a method to read the in-memory buffer after the instrumented application has exited.

VCAST_USE_COMPOUND_FOR_BATCH

```
clicast -lc Option VCAST_USE_COMPOUND_FOR_BATCH True | False
```

Category: Target Options

Description: This option allows you to run all of your test cases during a single test driver execution. This is useful if there are many steps involved in running on your target platform. When this option is set, and you choose to Batch Execute All, VectorCAST creates a temporary compound test case to execute all tests. The temporary compound contains all Initialization test cases, followed by all simple and Compound test cases.

VCAST_USE_DEFAULT_ARGS

```
clicast -lc Option VCAST_USE_DEFAULT_ARGS True | False
```

Category: Builder Options

Description: When creating the harness, VectorCAST will set up functions to be called with their default arguments.

VCAST_USE_DIR_LIST_FOR_INCLUDES

```
clicast -lc Option VCAST_USE_DIR_LIST_FOR_INCLUDES TRUE|FALSE
```

*Category:*Language Options

Description: Use items from the source directory list as include directories when invoking the preprocessor or compiler. Source directories might be specified via CCAST_.CFG, vcshell.db, or an .env file, depending on tool configuration. Disable this option when include directories should only be specified via unit-specific options, such as when the source directory list is very long, or when different units require different orders in the arguments.

VCAST_USE_EDG_PREPROCESSOR

```
clicast -lc Option VCAST_USE_EDG_PREPROCESSOR True | False
```

Category: Language Options

Description: This option is needed for compilers whose preprocessor does not retain comments or removes whitespace in the translation unit.

VCAST_USE_OPTIMIZED_MCDC_INSTRUMENTATION

```
clicast -lc Option VCAST_USE_OPTIMIZED_MCDC_INSTRUMENTATION True | False
```

Category: Coverage Options

Description: This option enables optimized MC/DC instrumentation for smaller MC/DC 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.

VCAST_USE_RELATIVE_PATHS

```
clicast -lc Option VCAST_USE_RELATIVE_PATHS True | False
```

Category: Builder Options

Description: If set, the new environment dialog will use relative paths. The target I/O directory will also use relative paths.

VCAST_USE_SEARCH_LIST_FOR_LIBRARY_TESTING

```
clicast -lc Option VCAST_USE_SEARCH_LIST_FOR_LIBRARY_TESTING True | False
```

Category: Builder Options

Description: Use the search list instead of a header file name to determine testability in Library Test Mode. The default value is false.

VCAST_USE_STATIC_MEMORY

```
clicast -lc Option VCAST_USE_STATIC_MEMORY True | False
```

Category: Coverage Options

Description: 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.

VCAST_USE_STD_STRING

```
clicast -lc Option VCAST_USE_STD_STRING True | False
```

Category: Builder Options

Description: This build option will cause the standard C library string functions to be used instead of their respective VCAST versions.

VCAST_USE_STDOUT

```
clicast -lc Option VCAST_USE_STDOUT True | False
```

Category: Coverage Options

Description: 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.

VCAST_USE_VCPP

```
clicast -lc Option VCAST_USE_VCPP True | False
```

Category: Language Options

Description: 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.

VCAST_USE_WINDSH_DOT_BAT

```
clicast -lc Option VCAST_USE_WINDSH_DOT_BAT True | False
```

Category: Target Options

Description: Check this option to force the use of windsh.bat to connect to the target. If this option is not set, VectorCAST will use windsh.exe (if available) and windsh.bat otherwise.

VCAST_USE_WINDSH_IO_REDIRECTION

```
clicast -lc Option VCAST_USE_WINDSH_IO_REDIRECTION True | False
```

Category: Target Options

Description: This option causes VectorCAST to insert TCL commands into the windsh.scr file to force vxWorks to use a /vio device for stdout. This option can be helpful, when the target stdout is going to the target console, rather than the windsh stdout.

VCAST_USER_CODE_FILE

```
clicast -lc Option VCAST_USER_CODE_FILE <script filename>
```

Category: Builder Options

Description: Set this to a file containing environment user code (in scripting format) to get this code automatically inserted into every environment built.

VCAST_VCDB_FLAG_STRING

```
clicast -lc Option VCAST_VCDB_FLAG_STRING <flag string -I=1,-D=1
```

Category: Language Options

Description: List of extra flags to be passed to vcdb as the --flags parameter. By default, the --flags are set to -I=1,-D=1, this option lets you add additional flags such as -isystem=1.

VCAST_VERBOSE_MANAGEMENT_REPORT

```
clicast -lc Option VCAST_VERBOSE_MANAGEMENT_REPORT True | False
```

Category: Report Options

Description: This option will copy the first line from the 'Test Notes' box in the test case editor to the test case management report. One possible use for this is for test requirement numbers to be correlated to test cases in the test case management report.

VCAST_VXWORKS

```
clicast -lc Option VCAST_VXWORKS True | False
```

Category: Target Options

Description: This option indicates that you are running with the vxWorks RTOS. This causes the -D VCAST_VXWORKS compilation switch to be set to allow the test harness code to run under vxWorks.

VCAST_VXWORKS_NO_CSHELL

```
clicast -lc Option VCAST_VXWORKS_NO_CSHELL True | False
```

Category: Target Options

Description: By default VectorCAST uses the 'c-shell' interpreter when creating the windsh script that controls download and execution of the test harness. In some cases, the 'c-shell' is not supported, and the 'command-shell' syntax must be used. For example, 64-bit vxSim requires this option to be set.

VCAST_VXWORKS_RTP_MODE

```
clicast -lc Option VCAST_VXWORKS_RTP_MODE True | False
```

Category: Target Options

Description: Set this option to run VxWorks executables as a Real-Time process (via the 'rtpSp'

command) rather than as a standard process (via the 'ld' and 'sp' commands).

VCAST_WHITEBOX_SEARCH_DIRS_ONLY

```
clicast -lc Option VCAST_WHITEBOX_SEARCH_DIRS_ONLY True | False
```

Category: Builder Options

Description: When whitebox processing is on and this option is enabled, C++ access specifiers 'private' and 'protected' are transformed to 'public' only in files located in search directories. With whitebox processing on and this option disabled, C++ access specifiers 'private' and 'protected' are transformed via preprocess and compilation arguments that apply to the entire translation unit. This option has no effect on C files.

VCAST_WRAP_TEXT_REPORT

```
clicast -lc Option VCAST_WRAP_TEXT_REPORT True | False
```

Category: Report Options

Description: In some text reports, if text is too long for a table cell then this option will allow the text to wrap in a new row.

VCDB_FILENAME

```
clicast -lc Option VCDB_FILENAME <vcdb filename>
```

Category: Language Options

Description: Path to the compiler settings database created by vcshell. Path can be absolute or relative, and can contain environment variables, using the syntax $(ENV_VAR) or %ENV_VAR%.

VCDB_CMD_VERB

```
clicast -lc Option VCDB_CMD_VERB <vcdb replacement verb>
```

Category: Language Options

Description: Set this option to the compile command verb that vcshell "sees" when it processes your build settings. For example, the command line compile command may be 'gcc' but the underlying executable that the system sees might be 'cc1'. In this case, set this option to be 'cc1'.

WHITEBOX

```
clicast -lc Option WHITEBOX No | Yes
```

Category: Builder Options

Description: Default setting for Whitebox checkbox in environment builder. (See the Whitebox Processing section of User's Guide for more details.)

APPENDIX H: RGW - COMMAND LINE REFERENCE

This reference contains an alphabetized list of VectorCAST/Requirements Gateway (RGW) commands.

Requirements Gateway supports the following RGW operations from the command line:

- Import and Export
- Profile Settings
- Repository Manipulation
- Test Automation



Note: Command line operations for RGW must be run from a directory with a CCAST_.CFG file that indicates where the repository is located.

When RGW begins, it loads the CCAST_.CFG file located in the current working directory and looks for the repository path in VCAST_REPOSITORY. If it cannot locate an entry for VCAST_REPOSITORY, it returns this text: "VCAST_REPOSITORY not set in .CFG file. Please set it using 'clicast option'."

The command line syntax is:

```
clicast -lc RGw -r <command> <arguments>
```

where:

<command>	is the command to execute
<arguments>	are the arguments for the command

RGW Commands

Configure Set

Syntax	**clicast -lc RGw Configure Set <Subsystem Name> <Option Name> <Option Value>**
Description	This command sets any of the configuration options for a particular subsystem. Use Configure Test to determine the names of options that need to be set for a particular subsystem.
Example	**clicast -lc RGw Configure Set DOORS user Administrator** **Key 'user' set for Profile 'DOORS'**

Export

Syntax	`clicast -lc RGw Export <Subsystem Name>`
Description	This command exports Requirements Test Data from the RGW Repository to an external Requirements System.
Example	`clicast -lc RGw Export DOORS`

Import

Syntax	`clicast -lc RGw IMport <Subsystem Name>`
Description	This command imports Requirements into the RGW Repository from an external Requirements system. **`<Subsystem Name>`** specifies the RGW subsystem that the import will work on. The following subsystems are supported: - CSV - Polarion - Jama
Example	`clicast -lc RGw Import DOORS`

Initialize

Syntax	`clicast -lc RGw INitialize`
Description	This command creates a **`requirements_gateway`** directory with **`repository.json`** and **`settings.json`** files in the location specified by VCAST_REPOSITORY.
Example	`clicast -lc RGw INitialize`

Requirement Report

Syntax	`clicast -lc RGw Requirement REPort <Query Type> <Query ARg 1> <Query Arg2>`
Description	This command returns Requirements that match the supplied query. Query Types can be: **All** - Return all Requirements in the Repository **Pending** - Return all Requirements with pending Test Data to be exported. **Changed** - Return all Requirements that have been changed between the two specified dates in Query Arguments 1 and 2. Date must be specified in the format 'YYYY-MM-DD hh:mm:ss'.

Example	`clicast -lc RGw Requirement REPort Changed 2017-01-02 12:00:00 2017-01-03 12:00:00`

Set Gateway

Syntax	`clicast -lc RGw Set Gateway <Gateway Name>`
Description	Select an RGW integration to use.
Example	`clicast  -lc RGw Set Gateway csv`

Testcase Link

Syntax	`clicast -lc -e <env> -u <unit> -s <sub> -t <test> RGw Testcase Link <Requirement Key>`
Description	This command links a Requirement to the Test Case specified by the Test Case identification flags: `-e -u -s -t`.
Example	`clicast -lc -e ENVIRONMENT_1 -u manager -s Clear_Table -t Clear_Table.001 RGW Testcase Link FR11`

Testcase Mark as Reviewed

Syntax	`clicast -lc -e <env> -u <unit> -s <sub> -t <test> RGw Testcase MArk_as_reviewed`
Description	Where a testcase is flagged as having been impacted by a Requirements change, this command marks the test case specified as having been reviewed and approved, and clears the "Marked for Review" flag.
Example	`clicast -lc -e ENVIRONMENT_1 -u manager -s Clear_Table -t Clear_Table.001 RGW MArk_as_reviewed`

Testcase Migrate_links

Syntax	`clicast -lc -e <env> RGw Testcase Migrate_links`
Description	This command migrates all legacy test case requirement links from the specified Environment into the Requirements Repository.
Example	`clicast -lc -e ENVIRONMENT_1 RGw Testcase Migrate_links`

Testcase Report

Syntax	**clicast -lc -e <env> -u <unit> -s <sub> -t <test> RGw Testcase Report**
Description	This command returns all Requirements linked to the Test Case specified in the flags.
Example	**clicast -lc -e ENVIRONMENT_1 -u manager -s Clear_Table -t Clear_Table.001 RGW Testcase Report**

Index

*
    indicating a statement covered 491
AALINKER.LIS 164, 654
Abort button 405
aborting test case execution 405
actual results 22
Add button
    in List tab 319
add or remove search directory
    reset stubbing method 117
Add Search Directory button 115
Add Search Directory Recursively
    search limit 120
Add Search Directory Recursively button 120
additional stubs
    enable/disable 139
additional stubs environment script
    ENVIRO.ADDITIONAL_STUB
        <function> 140
Additional Stubs tab 139
aggregate coverage report 417, 499
All Values button 312
alphabetize parameter tree parameters 276
alphabetize test cases 68, 248
animated control flow 515
animation coverage type 520
animation play speed 524
Any button
    in List Values tab 319
    in Range Values tab 317
    in Scalar Values tab 314
apply to array indices 309, 315
apply to expanded array indices only 310, 315
array
    indexing using enumerals 312
array types 685
arrays
    clearing data 311, 316
    collapsing unused elements 307
    entering data 309, 315
    expanding with + 306-307
    expanding with <<Expand Indices>>
        dialog 312
    ways to expand 312
assembler command 204
ASSEMBLER_CMD 204
assembly file extensions 195
atg
    baseline-on-test 344
    baselining 342
    baselining example 342
    baselining process 342
    baselining-allow-atg-tests-in-env 344
    delete test cases 339
    generate test cases 336, 350
    no-baselining-delete-tests 344
    reload test cases 339
automate test script maintenance 331
automatic finalization 264
automatic initialization 264
automatic test case generation 336
automatically clear test user code 442
automatically generate concrete classes 209
background color for failing cells 459
background color for partially passing cells 459
background color for passing cells 458
background colors
    changing for coverage output 544
backgrounds
    printing 87

basis path
    edit script for 379
    viewing for subprogram 378, 508
basis path analysis 507
basis path output
    building test cases from 507
basis paths
    do not automatically generate 216, 379
batch execute all 400
batch file 224-226, 229, 693
binder/linker log file 164
blank cells text 457
branch coverage 480
breakpoints
    removing 518
    setting in coverage animation 518
buffered coverage I/O 520
build options
    number of jobs 108
    parallel processing 108
    set coverage 109
    whitebox 109
building test cases 21
C compile command 201
C file extensions 194
C parser flags 201
C preprocessor command 201
C standard library
    link error using 175
C_ALT_COMPILE_CMD 201
C_ALT_EDG_FLAGS 201
C_ALT_PREPROCESS_CMD 201
C_COMPILE_CMD 171
C_COMPILE_CMD_FLAG 202
C_COMPILE_EXCLUDE_FLAGS 202
C_COMPILER_CFG_SOURCE 204
C_COMPILER_FAMILY_NAME 203
C_COMPILER_OUTPUT_FLAG 171
C_COMPILER_PY_ARGS 203
C_COMPILER_TAG 168
C_DEBUG_CMD 190
C_DEBUG_HELP_FILE 203
C_DEFINE_FLAG 171
C_DEFINE_LIST 174
C_EDG_FLAGS 192
C_EXEC_HELP_FILE 203
C_EXECUTE_CMD 169
C_INCLUDE_FLAG 171
C_LINK_CMD 189
C_LINK_OPTIONS 189
C_OBJECT_EXT 189
C_OUTPUT_FLAG 188
C_PREPROCESS_CMD 170
C_PREPROCESS_FILE 172
C++
    STL support 298
C++ file extensions 194
can’t proceed without parse data 122
CBA
    aggregate coverage report 590
    change cba display color 585
    covered by analysis report 587
    edit analysis result 584
    export analysis files 586
    file format 586
    import analysis files 586
    metrics report 591
    re-instrumenting with cba data 587
    remove analysis files 587
    using coverage analysis with sfp 592
    view cba data in reports 587
    view cba in coverage viewer 585

cba editor
branch coverage 582
mcdc coverage 583
statement coverage 582
CCAST_.CFG 160
opening 86
changing application font 70
changing industry mode 33
character types 296, 684
choose compiler 105
compiler not on default search path 106
class
testable
meaning of 19
class types 292
Classes of Interest tab
in creating new environment 135
classification tree method 369
Clear Dependency Data button 121, 161
Clear Expected button 616
Clear Input button 616
clear unapplied test script data 328
ClearCaseTM
integrating regression scripts with 230
clearing default source directories for wizard 174
CLICAST
accessing in Linux 697
accessing in Windows 697
clear_default_source_dirs 174
command format 697
console report 698
cover instrument mcdc 482
cover report aggregate 500
environment build 146
environment compile unit_appendix 612
environment delete 235
environment extract build_log 165
environment extract compile errors 164
environment extract link errors 164
environment extract overview 158
environment extract undefined_entities 165
environment get_unit_options 243
environment re_build 233
environment rebuild_range_data 243
environment recompile automatic 232, 236
environment recompile create script 238
environment recompile instrumented 237
environment recompile run 242
environment relink 243
environment script create 105, 235
environment script quick 235
environment script run 235-236
environment stub_user clear 628
environment stub_user compile 627
environment unit appendix 609
environment user clear 614
environment user compile 609
execute all command 710
execute batch 400
execute clear control flow 409
execute debug 405
execute run 390
execute save control flow 409
extract probe point log 573
IF DOCPROPERTY Project C 549-550
IF DOCPROPERTY Project C 418, 501
messages 660
options
ASM_FUNCS_BEHAVE_AS_INLINES 537, 702
ASSEMBLER_CMD 204, 702
C_ALT_COMPILE_CMD 201, 702

C_ALT_EDG_FLAGS 201, 702

C_ALT_PREPROCESS_CMD 201, 702

C_COMPILE_CMD 171, 703

C_COMPILE_CMD_FLAG 202-204, 703

C_COMPILE_EXCLUDE_FLAGS 202-203, 703

C_COMPILER_CFG_SOURCE 703

C_COMPILER_FAMILY_NAME 703

C_COMPILER_HIERARCHY_STRING 704

C_COMPILER_OUTPUT_FLAG 171, 704

C_COMPILER_PY_ARGS 203, 704

C_COMPILER_TAG 704

C_COMPILER_VERSION_CMD 704

C_DEBUG_CMD 191, 705

C_DEBUG_HELP_FILE 705

C_DEFINE_FLAG 171, 705

C_DEFINE_LIST 175, 705

C_EDG_FLAGS 192, 705

C_EXEC_HELP_FILE 203-204, 706

C_EXECUTE_CMD 169, 706

C_INCLUDE_FLAG 171, 706

C_LINK_CMD 189, 706

C_LINK_OPTIONS 189, 706

C_LINKER_VERSION_CMD 707

C_OBJECT_EXT 189, 707

C_OUTPUT_FLAG 189, 707

C_PREPROCESS_CMD 170, 707

C_PREPROCESS_FILE 172, 707

CASE_FALLTHROUGH_BRANCH_COVERAGE 528, 708

COMREADER_BAUD 708

COMREADER_COMPORT 708

COMREADER_DATA_BITS 708

COMREADER_ENABLED 708

COMREADER_PARITY 709

COMREADER_STOP_BITS 709

COVERAGE_ANIMATION_PLAY_SPEED_RATIO 524, 709

COVERAGE_IO_TYPE 520, 709

COVERAGE_TARGET 710

COVERAGE_TYPE 178-181, 710

EVENT_LIMIT 433, 469, 710

EXECUTABLE_EXTENSION 189, 710

FLOATING_POINT_TOLERANCE 434, 468, 711

FORCE_EXPLICIT_TEMPLATE_ARGS 210, 711

GLOBALS_DISPLAY 450, 711

INST_LINK_FILE 711

LIBRARY 178

LIBRARY_INCLUDE_DIR 173, 711

LIBRARY_TESTING 712

LINES_PER_PAGE 712

LINK_FILE_TEMPLATE 712

MANAGE_ENVIRONMENT_VARIABLE 712

MANAGE_LIBRARY_INCLUDE_DIR 713

MANAGE_ROOT_SOURCE_DIR 713

MANAGE_TESTABLE_SOURCE_DIR 713

MANAGE_TYPE_HANDLED_SOURCE_DIR 713

MAX_VARY_RANGE 713

METRICS_TOTAL_LEVEL_DISPLAY 449, 714

NUMBER_OF_DATA_PARTITIONS 714

PAGE_WIDTH 714

PRECOMPILE_CMD 204, 714

PRECOMPILE_EXT 205, 714

RANGE_CHECK 433, 715

REPORT_OBJECTS 715

REPORT_PARAMETERS 715

reports extract 75

SBF_LOC_MEMBER_IN_NSP 715

SBF_LOC_MEMBER_OUTSIDE_NSP 716

SBF_LOC_NONMEMBER_IN_NSP 716

SBF_LOC_NONMEMBER_OUTSIDE_NSP 716

SCORE_DEBUG_CMD 716

SCORE_EXECUTE_CMD 716

show non-search directory globals 220

SHOW_NOTES_IN_FULL_REPORT 454, 717

SOURCE_EXTENSION 192, 717

STANDARD_ERROR 435, 717

STANDARD_OUTPUT 435, 717

STUB_DEPENDENCIES 176, 717

SUBSTITUTE_CODE_FOR_C_FILE 191, 718

TARGET_BOARD_NAME 718

TARGET_BOOT_HOSTNAME 718

TARGET_IO_DIRECTORY 718

TARGET_SIM_CMD 718

TARGET_TFTP_DIR 719

TEST_CASE_TIMEOUT 719

TESTABLE_SOURCE_DIR 173, 719

TI_CC_DSP_BIOS_SUPPORT 719

TI_CC_TCF_CMD 719

TI_CC_TCF_FILENAME 720

TYPE_HANDLED_SOURCE_DIR 173, 720

UNCON_ARRAY_SIZE 221, 720

UUT_LINK_FILE 720

VARIANT_LOGICS_PATH 182, 721

VCAST_ACCESS_NON_SEARCH_GLOBALS 721

VCAST_ADA_FILE_EXTENSIONS 721

VCAST_ALT_WB_METHOD 219

VCAST_ALWAYS_DO_STUB_PROCESSING_IN_TI 216, 721

VCAST_APPEND_TO_TESTINSS 721

VCAST_ASSEMBLY_FILE_EXTENSIONS 195, 722

VCAST_ASSIGN_WITHOUT_COPY_CTOR 722

VCAST_AUTO_CLEAR_TEST_USER_CODE 443, 722

VCAST_AUTO_CONCRETE_CLASS_GENERATION 210, 722

VCAST_AUTO_TEST_INSTANTIATE_CLASS 723

VCAST_AVOID_COMMA_OPERATOR 532-533, 723

VCAST_BASIS_PATHS_FOR_CONSTANT_BRANCHES 532, 723

VCAST_BUFFER_OUTPUT 723

VCAST_C_FILE_EXTENSIONS 194, 724

VCAST_CLASS_INST_SHARING 207, 724

VCAST_CMD_AFTER_C_ OPTIONS 540

VCAST_CMD_AFTER_ INSTRUMENTATION 539, 724

VCAST_CMD_DURING_ENV_ CLOSE 724

VCAST_CMD_DURING_ENV_ OPEN 724

VCAST_COLLAPSE_STD_ HEADERS 199, 725

VCAST_COMMAND_AFTER_C_ OPTIONS 725

VCAST_COMMAND_LINE_ DEBUGGER 191, 725

VCAST_COMPACT 726

VCAST_COMPILER_SUPPORTS_ CPP_CASTS 217, 726

VCAST_COMPILER_TEMPLATE_ SECTION 453, 726

VCAST_CONSIDER_UNCALLED_ TEMPL_FUNCTIONS_ TESTABLE 214, 726

VCAST_CONSOLE_REPORT_ FORMAT 698, 726

VCAST_CONTROL_FLOW_GRAPH_ CONFIG 727

VCAST_CONVERT_OCTAL_HEX_ STRINGS_ASCII 452

VCAST_CONVERT_OCTAL_HEX_ STRINGS_TO_ASCII 727

VCAST_COVER_CATCH_AS_ BRANCH 530, 727

VCAST_COVER_OVERLOADED_ LOGICAL_OPERATORS 728

VCAST_COVER_SEPARATE_TYPES_ FILES 728

VCAST_COVER_STATEMENTS_BY_ BLOCK 527, 728

VCAST_COVERAGE_COLLAPSE_ ALL 728

VCAST_COVERAGE_DATABASE_ CACHE_SIZE 538, 729

VCAST_COVERAGE_DATABASE_ SIMPLIFIED_LOCKING 729

VCAST_COVERAGE_FIELD_ WIDTH 538, 729

VCAST_COVERAGE_FOR_ AGGREGATE_INIT 729

VCAST_COVERAGE_FOR_ DECLARATIONS 729

VCAST_COVERAGE_FOR_ HEADERS 531, 730

VCAST_COVERAGE_FOR_ LAMBDAS 730

VCAST_COVERAGE_POINTS_AS_ MACROS 531, 730

VCAST_CPP_FILE_EXTENSIONS 195, 730

VCAST_CREATE_INIT_OBJECTS_ DYNAMICALLY 210, 731

VCAST_CUSTOM_REPORT_ FORMAT 459, 731

VCAST_CYGPATH_LINE_ DIRECTIVES 197, 731

VCAST_DEFAULT_SCRIPT_ HEADER 456, 731

VCAST_DEPENDENCY_CACHE_ DIR 181, 732

VCAST_DETECT_THROWN_ EXCEPTION_TYPES 213, 732

VCAST_DETECT_UNUSED_ EXPECTED_UC 443, 732

VCAST_DISABLE_ASM_FUNC_ TESTING 208, 732

VCAST_DISABLE_AUTO_BASIS_ PATH_GEN 217, 732

VCAST_DISABLE_BOOLEAN_CAST 733

VCAST_DISABLE_CPP_EXCEPTIONS 213, 733

VCAST_DISABLE_DIRECT_ARRAY_INDEXING 209, 733

VCAST_DISABLE_STD_CONTAINER_DETECTION 215, 733

VCAST_DISABLE_STD_SMART_PTR_DETECTION 215, 734

VCAST_DISABLE_STD_STRING_DETECTION 214, 734

VCAST_DISABLE_STD_WSTRING_DETECTION 215, 734

VCAST_DISABLE_TEST_MAINTENANCE 734

VCAST_DISABLE_TI_BITFIELD 735

VCAST_DISABLE_TI_STRING 735

VCAST_DISPLAY_CONST_BRANCH_CONDITIONS 735

VCAST_DISPLAY_EMPTY_COVERAGE 445, 735

VCAST_DISPLAY_FUNCTION_COVERAGE 446, 735

VCAST_DISPLAY_UNINST_EXPR 445, 736

VCAST_DO_COMBINATION 437, 471, 736

VCAST_DO_NOT_REDEFINE_MAIN 736

VCAST_DUMP_BUFFER 736

VCAST_DUMP_COVERAGE_AT_EXIT 737

VCAST_EDG_PREPROCESS_FLAGS 198, 737

VCAST_EMPTY_STATEMENTS_COVERABLE 737

VCAST_EMPTY_TESTCASE_FAIL 737

VCAST_ENABLE_DATA_CLEAR 521

VCAST_ENABLE_DATA_CLEAR_API 737

VCAST_ENABLE_FUNCTION_AND_CALL_COVERAGE 526, 738

VCAST_ENABLE_FUNCTION_CALL_COVERAGE 526, 738

VCAST_ENABLE_PROBE_POINTS 218, 738

VCAST_ENHANCED_FOR_LOOP_COVERAGE 525, 738

VCAST_ENVIRONMENT_FILES 200, 738

VCAST_ESCAPE_LINE_DIRECTIVES 196, 739

VCAST_EXECUTE_WITH_STDIO 739

VCAST_EXECUTE_WITH_STDOUT 739

VCAST_EXPECTED_BEFORE_UUT_CALL 452, 472, 739

VCAST_FAR_STDIN_DATA 740

VCAST_FIELD_DOT_NOTATION 740

VCAST_FILE_ENCODING 740

VCAST_FILE_INDEX 740

VCAST_FILE_PREFIX 741

VCAST_FILE_VERSION_COMMAND 455, 741

VCAST_FLOAT_PRECISION 455, 741

VCAST_FORCE_ELAB_TYPE_SPEC 741

VCAST_FORCE_NO_USERGLOBALS 741

VCAST_FORCE_NOTSUPPORTED_PROC 742

VCAST_FORCE_REPORT_DATES 742

VCAST_FREE_HARNESS_DATA 742

VCAST_FULL_STRINGS 441, 466, 742

VCAST_FUNC_PTR_INLINES_C 220

VCAST_FUNCTION_POINTER_SUPPORT 212, 743

VCAST_FUNCTION_PROBE_BEFORE_DECL 743

VCAST_GEN_ALL_EXECUTION_REPORT_FORMATS 743

VCAST_GH_INT_FILE 744

VCAST_GH_INTEX_CMD 190, 744

VCAST_GH_INTEX_FILE 190

VCAST_GLOBAL_BUFFER_OPTIMIZATIONS 529, 744

VCAST_GNU_SYSTEM_MARKER 200, 744

VCAST_HANDLE_TERNARY_OPERATOR 745

VCAST_HAS_LONGLONG 745

VCAST_HEADER_FILE_EXTENSIONS 194, 745

VCAST_HEX_NOTATION 435, 467, 745

VCAST_IGNORE_ERROR_WHEN_READ_FOLLOWS_ADDRESS_OF 529, 746

VCAST_IGNORE_INCOMPLETE_TESTS 438, 443, 746

VCAST_IGNORE_PSC_RAM_PAYLOAD_REBOOT_STATUS 746

VCAST_INHERITED_INLINES_ALWAYS_TESTABLE 746

VCAST_INST_FILE_MAX_LINES 746

VCAST_INSTANTIATE_ALL_TEMPLATE_FUNCTIONS 208, 747

VCAST_INSTRUMENT_ASSIGNMENTS 747

VCAST_INSTRUMENT_DEAD_CODE 747

VCAST_INSTRUMENT_PARAMETERS 747

VCAST_INSTRUMENT_SEPARATES 748

VCAST_INSTRUMENT_UNREACHABLE_CODE 525

VCAST_LIBRARY_STUBS 183, 748

VCAST_MAIN 748

VCAST_MASKING_MCDC 748

VCAST_MAX_CAPTURED_ASCII_DATA 521, 749

VCAST_MAX_COVERED_SUBPROGRAMS 522, 749

VCAST_MAX_HEAP_SIZE 749

VCAST_MAX_MCDC_ABORT_INST 749

VCAST_MAX_MCDC_CONDITIONS 535, 749

VCAST_MAX_MCDC_STATEMENTS 523, 750

VCAST_MAX_STRING_LENGTH 750

VCAST_MAX_TABLE_SUBCONDITIONS 750

VCAST_MAX_TARGET_FILES 750

VCAST_MICROSOFT_LONG_LONG 751

VCAST_MINIMAL_TERMINATION 751

VCAST_MONITOR 751

VCAST_MULTIBYTE_CHARACTERS 30, 193, 751

VCAST_NO_EXIT 752

VCAST_NO_FFLUSH 752

VCAST_NO_FLOAT 752

VCAST_NO_LIMITS 752

VCAST_NO_LONG_DOUBLE 208, 752

VCAST_NO_MALLOC 753

VCAST_NO_SETJMP 753

VCAST_NO_SIGNAL 753

VCAST_NO_SPRINTF 753

VCAST_NO_STANDARD_PKG_ USAGE 753

VCAST_NO_STD_FILES 754

VCAST_NO_STDIN 754

VCAST_NO_STDLIB 754

VCAST_NO_TYPE_SUPPORT 754

VCAST_NOTES_SECTION_ TEMPLATE 456, 754

VCAST_OLD_STYLE_MANAGEMENT_ REPORT 454, 755

VCAST_ONLY_SHOW_GLOBAL_ OBJECTS_IN_ONE_UNIT 208, 755

VCAST_OPTIMIZED_MCDC_ STORAGE_THRESHOLD 536, 755

VCAST_OUTPUT_BUFFER_SIZE 755

VCAST_PACK_INSTRUMENTATION_ STORAGE 528, 756

VCAST_PARSE_FUNCTION_ TEMPLATES 193, 756

VCAST_PASS_SUBPROGRAM_ NAME_TO_UC 756

VCAST_POST_PREPROCESS_ COMMAND 198, 756

VCAST_POST_RUN_DELAY 757

VCAST_PRE_EXECUTE_CMD 757

VCAST_PRECOMPILE_SCRIPT 221, 757

VCAST_PREPEND_TO_PATH_ DIRS 757

VCAST_PREPROCESS_ PREINCLUDE 198, 758

VCAST_PROBE_POINTS_AS_ EVENTS 452, 569, 758

VCAST_PSC_RAM_PAYLOAD 758

VCAST_PSC_RAM_PAYLOAD_ REBOOT_CMD 758

VCAST_PSC_USE_ADDRESS_ SPACE 758

VCAST_RECURSIVE_DIRECTORY_ ADD_LIMIT 72, 759

VCAST_REFERENCED_GLOBAL 68, 275

VCAST_REFERENCED_GLOBALS 759

VCAST_RELATIVE_INCLUDES 759

VCAST_RELINK_PROMPT_FOR_PSC_ RAM_PAYLOAD 759

VCAST_REMOVE_PREPROCESSOR_ COMMENTS 196, 759

VCAST_REPORT_ON_IMPLICIT_ DEFAULT_CASE 527, 760

VCAST_REPOSITORY 170, 760

VCAST_RPTS_AUTOMATIC_ BGCOLOR 760

VCAST_RPTS_BODY_BGCOLOR 760

VCAST_RPTS_COMPLEXITY_ COLUMN_WIDTH 462, 760

VCAST_RPTS_COVERAGE_RESULT_ COLUMN_WIDTH 462, 761

VCAST_RPTS_CUSTOM_CONFIG 72, 761

VCAST_RPTS_CUSTOM_CSS 80, 460, 761

VCAST_RPTS_DATE_COLUMN_WIDTH 461, 761
VCAST_RPTS_DEFAULT_FONT_COLOR 762
VCAST_RPTS_DEFAULT_FONT_FACE 762
VCAST_RPTS_DEFAULT_FONT_SIZE 762
VCAST_RPTS_DELIMITER 762
VCAST_RPTS_EMPTY_DATA_STRING 457, 762
VCAST_RPTS_FAIL_COLOR 763
VCAST_RPTS_HEADER 457, 763
VCAST_RPTS_HEADING_FONT_SIZE 763
VCAST_RPTS_NOTES_COLUMN_WIDTH 462, 763
VCAST_RPTS_PASS_COLOR 763
VCAST_RPTS_PRETTY_PRINT_HTML 764
VCAST_RPTS_REQUIREMENTS_COLUMN_WIDTH 764
VCAST_RPTS_RESULT_COLUMN_WIDTH 461, 764
VCAST_RPTS_SECTION_TITLE_BGCOLOR 764
VCAST_RPTS_SELF_CONTAINED 80, 764
VCAST_RPTS_SHOW_VERSION 454, 765
VCAST_RPTS_SUBPROGRAM_COLUMN_WIDTH 461, 765
VCAST_RPTS_TABLE_BORDER_SIZE 765
VCAST_RPTS_TABLE_CELL_PADDING 765
VCAST_RPTS_TABLE_DATA_BGCOLOR 765
VCAST_RPTS_TABLE_DATA_FAIL_BGCOLOR 459, 766
VCAST_RPTS_TABLE_DATA_PARTIAL_BGCOLOR 459, 766
VCAST_RPTS_TABLE_DATA_PASS_BGCOLOR 459, 766
VCAST_RPTS_TABLE_DATA_TEXT_ALIGNMENT 766
VCAST_RPTS_TABLE_HEADING_BGCOLOR 766
VCAST_RPTS_TESTCASE_COLUMN_WIDTH 461, 767
VCAST_RPTS_UNIT_COLUMN_WIDTH 460, 767
VCAST_RPTS_USER_REPORTS_DIR 767
VCAST_RPTS_WRAP_NOTES 767
VCAST_SBF_LIBRARY_STUBS 218, 767
VCAST_SBF_NONTESTABLE_FUNCTIONS 217, 768
VCAST_SBF_TEMPLATES 768
VCAST_SCRIPT_ALWAYS_USES_PARAMS 441
VCAST_SCRIPT_EXPAND_ARRAYS 768
VCAST_SCRIPT_IN_CREATION_ORDER 768
VCAST_SCRIPT_LOG_SHOW_ONLY_ERRORS 438, 769
VCAST_SHOW_INLINES_COVERED_IN_ALL_UNITS 530, 769
VCAST_SHOW_ONLY_DATA_WITH_EXPECTED_VALUES 451, 472, 769
VCAST_SHOW_ONLY_EVENTS_

WITH_EXPECTED_VALUES 451, 473, 769

VCAST_SHOW_STDOUT_CONSOLE 439, 770

VCAST_SIMPLIFIED_CONDITION_COVERAGE 534, 770

VCAST_SORT_METRICS_RPT_BY_DIR 444, 446, 770

VCAST_SPLIT_UC_FILES 211, 770

VCAST_STARTUP_FILE 205, 770

VCAST_STDIO 771

VCAST_STRICT_TEST_CASE_IMPORT 442, 771

VCAST_STRIP_SYS_HEADER_PATH 771

VCAST_STUB_ALL_SPECIAL_FUNCTIONS 771

VCAST_STUB_ALL_SUBPROGRAMS 772

VCAST_STUB_BY_IMPLEMENTATION 220, 772

VCAST_STUB_WITHOUT_PROTOTYPE 772

VCAST_SUBDIRS_OF_SEARCH_LIST_CONSIDERED 216, 772

VCAST_SUPPRESS_COVERABLE_FUNCTIONS 542, 773

VCAST_SUPPRESS_IMPORT_FILE_CHECKS 773

VCAST_TEMPLATE_STRUCT_AS_CLASS 213, 773

VCAST_TEST_ALL_INLINES 207, 774

VCAST_TEST_ALL_NON_MEMBER_INLINES 207, 774

VCAST_TEST_UNIT_USER_CODE 212, 774

VCAST_TEST_UNIT_VALUES_DICTIONARY 774

VCAST_TEST_VALUES_DICTIONARY 182, 291

VCAST_TESTCASE_FAIL_ON_NO_EXP_RETURN 438, 775

VCAST_TESTCASE_FAIL_ON_NO_EXPECTED 442, 775

VCAST_TI_IGNORE_QUALIFIERS 209, 775

VCAST_TORNADO_CONSTRUCTOR_CALL_FILE 775

VCAST_TREAT_DC_INITIALIZATIONS_AS_WRITE 533, 775

VCAST_TYPEOF_OPERATOR 776

VCAST_UNCOVERED_LINE_INDICATOR 523, 776

VCAST_UNEXPECTED_EXCEPTIONS_FAIL 442, 776

VCAST_UNEXPECTED_SIGNALS_FAIL 438, 776

VCAST_UNIT_TYPE 177, 777

VCAST_UNSIGNED_LONG_MCDC_STORAGE 535, 777

VCAST_USE_BUFFERED_ASCII_DATA 520

VCAST_USE_BUFFERED_ASCII_DATAX 777

VCAST_USE_COMPOUND_FOR_BATCH 777

VCAST_USE_DEFAULT_ARGS 211, 778

VCAST_USE_DIR_LIST_FOR_INCLUDES 175, 778

VCAST_USE_EDG_PREPROCESSOR 197, 778

VCAST_USE_OPTIMIZED_MCDC_INSTRUMENTATION 536, 778

VCAST_USE_RELATIVE_PATHS 183, 778
VCAST_USE_SEARCH_LIST_FOR_LIBRARY_TESTING 212, 779
VCAST_USE_STATIC_MEMORY 522, 779
VCAST_USE_STD_STRING 206, 779
VCAST_USE_STDOUT 779
VCAST_USE_VCPP 197, 780
VCAST_USE_WINDSH_DOT_BAT 780
VCAST_USE_WINDSH_IO_REDIRECTION 780
VCAST_USER_CODE_FILE 780
VCAST_VCAST_MASKING_MCDC 534
VCAST_VCDB_FLAG_STRING 780
VCAST_VERBOSE_MANAGEMENT_REPORT 453, 781
VCAST_VXWORKS 781
VCAST_VXWORKS_NO_CSHELL 781
VCAST_VXWORKS_RTP_MODE 781
VCAST_WHITEBOX_SEARCH_DIRS_ONLY 218, 782
VCAST_WRAP_TEXT_REPORT 398, 782
VCDB_CMD_VERB 182, 782
VCDB_FILENAME 181, 782
WHITEBOX 182, 782
probe point apply 562
probe point remove all 567
reports custom actual 394, 415
reports custom cba 419, 590
reports custom coverage 417, 500
reports custom equivalence_matrices 512
reports custom full 415
reports custom function call 418
reports custom management 416
reports custom test 411
reports diagnostic 52, 387
rgw testcase mark as reviewed 642
set_default_source_dirs 173
template 106, 169
test compound_only 267
test csv2tst create 369
test csv2tst load 366
test csv2tst template 362
test delete 252-253
test script create 327
test script log 328
test script run 328
test script template 329
test unapplied clear 328
test user clear 616
tools auto_test_generation 259
tools basis_path 379, 508
tools cover <coverage-type> 486
tools cover disable 487, 489
tools cover enable 488
tools cover level_B 485
tools cover none 488
tools cover statement 480
tools execute_commands 226
tools export cba 586
tools import_coverage 550
tools regression_test 225, 229
tools update_basis_path 379
clicast
options
VCAST_ENABLE_PROBE_POINTS 557
clicast actuals to expected 272, 410
closing environment 222

closure concept 19
ignoring units 20
minimizing 19
code complexity 378, 508
code coverage summary 503
coded tests 344
generate test cases 344, 350
collapse all subprograms 496
collapsing arrays 307
colors 87
combination testing 435, 469
command line debugger 191
compare expected before UUT call 452, 472
compile command 171
compile command flag 202
compile command line
parse for library includes and defined variables 184
compile error
disable assembly function testing 208
compile flag to exclude 202
compile log file 164
compile test-specific user code in separate unit 211
compiler
testing settings of 185
compiler family name 203
compiler lacks long double 208
compiler python args 203
compiler template
choosing between C and C++ 168
components
of test harness 18
compound Only test cases 267
compound test cases
automatic initialization finalization 264
building 261
deleting test case from 266
enable/disable reporting for slots 270
executing 398
find slot in test case tree 266
nested 267
opening Coverage Viewer from 266
opening test case from 265
rearranging order of test cases in 265
search editor 266
conditional statement longer than 5000 chars 487
configuration file
created by options dialog 376
how to use several of 376
opening 86
configuration management 224
configure stubs 623
how to edit 624
consider uncalled template functions testable 214
consider unit user code functions testable 212
console report
compount test output 700
default ouput 699
format 698
legacy output 699
quiet output 700
verbose output 699
constrained array types 306, 685
contacting
technical support 53
continuation character 688
control flow
animated 515
clearing 409
edit 405
edit a control flow 406

edit context menu 406
edit subprogram name 407
executing a test case with a control flow that has changed 407
expand/collapse subprogram tree 405
manually creating a control flow 406
saving 409
subprogram identifier 407
subprograms panel 405
test case editor 405
test execution report 407
view 405
convert octal/hex-encoded strings to ASCII 452
copying text 88
coverage
aggregate coverage report 417, 499
animation play speed 524
before using 478
control instrumentation using pragmas 487
coverage viewer 490
customizing display 496
deleting imported results 550
disabling 488
enabling 488
exporting 547
importing 550
importing from VectorCAST/Cover 545
in status bar 43
initializing branch 480
initializing custom 485
initializing DO-178B Level A 483
initializing DO-178B Level B 485
initializing DO-178B Level C 480
initializing EN-50128 SIL 1/2 480
initializing EN-50128 SIL 3 485
initializing EN-50128 SIL 4 483
initializing function 483
initializing function + function call 484
initializing IEC61508 SIL 1/2 480
initializing IEC61508 SIL 3 485
initializing IEC61508 SIL 4 483
initializing IEC-62304 Class A 480
initializing IEC-62304 Class B 485
initializing IEC-62304 Class C 483
initializing ISO-26262 ASIL A 480
initializing ISO-26262 ASIL B/C 485
initializing ISO-26262 ASIL D 483
initializing MC/DC 481
initializing statement 479
initializing Statement 480
initializing Statement+BRANCH 485
initializing Statement+MC/DC 483
instrument unreachable code 525
not instrumenting sections of source code 487
opening test case which caused 497
pairs status 482, 501
recompile instrumented code 237
removing data 499
results navigation 495
show true and false in for loops 524
uncovered line indicator 523
uninstrument 488
xml aggregate coverage report 500
coverage analysis editor 581
coverage animation
setting breakpoints 518
toolbar 516-517
coverage data in ASCII format
compared to buffered I/O 520
coverage data used in reports 410
Coverage menu 478
Export Script 547
Import Results 545

Initialize Custom 485
Remove All Coverage Data 499
Uninstrument 488
View Import Log 547
coverage reports
printing colors 87
coverage type
set default 109
coverage viewer 489-490
branch coverage 492
function + function call coverage 493
function coverage 492
mcdc coverage 494
numbers along left-hand side 491
open coverage viewer 489
statement coverage 491
viewing cba lines 585
COVERAGE_IO_TYPE 519
covered by analysis
see CBA 581
covered by analysis report 419, 587
covered lines 543
create CSV template
CSV options panel 356
parameter tree 356
create csv template file 355
create initialization objects dynamically 210
create new environment
from a JSON Compilation Database 111
from vcshell.db 110
select a variant logic file 112
create tab delmited template file 355
create test
from CSV map 365
creating
regression test 224
creating new reports 81
creating reports 22
css 72
csv
mapping a column 364
CSV
edit map test case 366
csv file 355
CSV map
create from existing file 362
csv map
create tests 365
mapping view 364
rows and columns in file 362
table.csv 362
csv mapping
save 365
csv options
Load all rows 364
Load rows from to 364
Row containing column headings 363
Rows per testcase 364
Rows to skip 363
Test name column 364
Test notes column 364
CSV options panel 356
CSV Options panel
Create Template type selection 357
csv options panel
comma separted values 358
tab separted values 358
value seperator type 357
csv template
create template file 360
populate with data 361
csv template headers 359

csv template map
modify column order 359
custom
not stubbed 128
stubbed (by implementation) 128
stubbed (by prototype) 128
custom css
add 79
add from command line 80
add from gui 79
create 78
embedding style sheets 80
custom end macro 175
custom script header text file 456
custom style sheet 78
custom templates 75
configure 75
custom tools
adding 381
default tools 380
deleting 384
import export 385
settings for 384
sharing 385
testing 383
customer number
obtaining 53
customize reports
add footer 76
add logo 77
add section 73
create new 81
custom css 78
existing 72
order section 74
remove section 73
replace section 74
report components 81
run new report 83
sections 76
style 78
templates 75
customized stubbing
specify 128
customizing Coverage Viewer 496
customizing report sections 76
customizing report style 78
customizing reports 72
cutting text 88
cygpath line directives 197
data
display full string 439, 464
entering directly in MDI window 273
entering for parameters 314
entering for test cases 277
entering in Scalar Values tab 314
entering lists 284
entering on List Values tab 319
entering on Range Values tab 317
entering ranges 284
for parameters 617
global 283
ranges
testing all combinations 435, 469
two ways to enter 272
data types 284
class 292
constrained arrays 306
entering number for enumeration 285
entering number in different bases 286
entering real numbers in scientific notation 287

enumeration 284
numeric 285
of type char 296
of type string 296
pointer types 313
string 297
structures 291
unconstrained arrays 313
unions 292
void pointer types 293
debug help file 203
debugger command 190
Default button 376
default compiler options 204
default coverage type 178
default search list for wizard 170
default string display mode 70
define flags 171
defined variable 174
parse out of compile command line 184
deleting an environment 234
deleting test cases 252-253
deleting test cases, part of compound 252
delta value
in ranges 317
dependency data directory 181
dependent unit
meaning of 18
smart stubs 18
detect thrown exception types 212
disable assembly function testing 208
disable coverage 488
disable direct array indexing 209
disable exception handling 213
display actual value with user code 620
display coverage in reports (always) 445
display full string data 439, 464
display only referenced globals 67, 274
display theme 62
display uninstrumented mcdc expressions in reports 445
do not automatically generate basis paths 216, 379
do not detect ANSI C++ standard containers 215
do not detect ANSI C++ standard string types 214
do not detect ANSI C++ standard wstring types 214
docking window
floating 37
moving 36
returning to default layout 37-38
Down button
in List tab 320
driver program
meaning of 18
duplicate names in exported scripts 327
duplicating test cases 250
EDG parser 192
edit existing map test case 366
Edit Harness Source 377
edit link options 242
delete an entry 147, 149, 242
select libraries 242
Edit menu
Copy 88
Cut 88
Find 89
Find again 91
Goto Line 98
Paste 88
Redo 88
Undo 88

enable coverage 488
enable disable a test case 250
enable function pointer type support 211
enable probe points 218
enable sbf capability for library stubs 217
enable sbf capability for non-testable functions 217
enable sbf capability for template functions 218
Enumerals 312
enumerals
  execution reports 313
  test case editor 313
  test case scripts 313
enumeration types 284, 684
  entering number for 285
ENVIRO.ADDITIONAL_TESTABLE_FUNCTION 664
ENVIRO.ADDITIONAL_UNIT_BODIES 670
ENVIRO.CLASS_OF_INTEREST 137, 664
ENVIRO.COVERAGE_TYPE 663
ENVIRO.DRIVER_PREFIX_USER_CODE 665
ENVIRO.DRIVER_PREFIX_USER_CODE_FILE 665
ENVIRO.END 671
ENVIRO.INDUSTRY MODE 671
ENVIRO.LIBRARY_INCLUDE_DIR 662
ENVIRO.NAME 661
ENVIRO.NEW 661, 671
ENVIRO.SEARCH_LIST 662
ENVIRO.STUB 663
  ALL_BY_PROTOTYPE 663
ENVIRO.STUB_BY_FUNCTION 661
ENVIRO.STUB_DEPEND_USER_CODE_FILE 670
ENVIRO.STUB_ENTRY_USER_CODE 669
ENVIRO.STUB_EXIT_USER_CODE 669
ENVIRO.STUB_USER_CODE_FILE 669
ENVIRO.SUPPRESS_STUB 664
ENVIRO.SUPPRESS_TESTABLE_FUNCTION 142
ENVIRO.TEST_VALUES_DICTIONARY 662
ENVIRO.TYPE_HANDLED_DIRS_ALLOWED 663
ENVIRO.TYPE_HANDLED_SOURCE_DIR 662
ENVIRO.UNIT_APPENDIX_USER_CODE 665
ENVIRO.UNIT_APPENDIX_USER_CODE_FILE 665
ENVIRO.UNIT_COMPILATION_ARGUMENTS 135, 664
ENVIRO.UNIT_PREFIX_USER_CODE 665
ENVIRO.UNIT_PREFIX_USER_CODE_FILE 665
ENVIRO.USER_CODE_CAPTURE 668
ENVIRO.USER_CODE_DEPENDENCIES 667
ENVIRO.USER_CODE_INITIALIZE 668
ENVIRO.USER_CODE_OBJECTS 667
ENVIRO.USER_CODE_ONE_SHOT_INIT 667
ENVIRO.USER_CODE_ONE_SHOT_TERM 668
ENVIRO.USER_CODE_STUB_PROCESSING 668
ENVIRO.USER_CODE_TIMER_START 671
ENVIRO.USER_CODE_TIMER_STOP 671
ENVIRO.USER_GLOBALS 665
ENVIRO.USER_PARAMETERS 666
ENVIRO.UUT 661
ENVIRO.VARIANT_LOGICS_PATH 662
ENVIRO.WHITE_BOX 663
environment
  build options 109
  building new 145
  closing 222
  constructor 21
  coverage type 109
  deleting 234

editing 222
files created during building 159
get unit options 243
incorporating source code changes into 232
library include directories 115
loading 107
meaning of 17
naming 107
opening 85, 221
rebuilding 232
rebuilding range data for 243
regression test, creating 224
renaming 222
scripting language 661
search directories 115
sending to Tech Support 53
setting working dir before loading script with relative paths 107
source directories list 114
stubbed-by-function unit(s) under test 125
type-handled directories 115
un-deleting 235
unit(s) under test 124
updating 222
user code 596
whitebox 109
whitebox option 110
environment builder log file 165
environment files 199
Environment menu
Configure Stubs 596, 623
Clear 627
Rebuild Environment 232
Rebuild range data 243
Rebuild Range Data 331
Recompile
Automatic 236
Create Script 237
Instrumented Code 237
Run Script 241
Relink 243
Scripting
Create 105, 235
Run 235-236
Update Environment 222
User Code 596
Clear 614
Compile 609, 627
Edit 600
View
Compile Errors 164
Environment Build Log 165
Link Errors 164
Metrics Report 418, 482, 500
Overview 157
Test Case Management Report 416, 482
Undefined Entities Log 165
environment name 107
duplicate name 107
environment script
opening 85
environment scripts
saving user code in 665
environment variables
setting 691
VCAST_ALTERNATE_REPORT 691
VCAST_DEBUG_LEVELS 691
VCAST_DEBUG_SCRIPT 691

VCAST_DISABLE_FILE_DELETE 691
VCAST_END_WAIT 691
VCAST_FORCE_OBJECT_EXTENSION 692
VCAST_GLOBAL_DEFINE_LIST 692
VCAST_IGNORE_COMPILE_ERRORS 692
VCAST_IGNORE_FILE_EXISTS_CHECK_FOR_COVERABLE 692
VCAST_IGNORE_TYPEMARKS 692
VCAST_LEGACY_CFG_FILE_INHERITANCE 692
VCAST_MAX_LINE_LENGTH 487, 692
VCAST_NEVER_STUB_LIST 663, 693
VCAST_NO_PARSE 693
VCAST_NUM_JOBS 693
VCAST_QIK_ASSUME_SRC_HAS_NOT_CHANGED 693
VCAST_RECURSION_DEPTH 693
VCAST_REGRESSION_COMMAND 693
VCAST_RPTS_PRETTY_PRINT_HTML 693
VCAST_RSP_USER_TEMPLATE 693
VCAST_TORNADO_OLD_SCRIPT 694
VCAST_WHITEBOX 694
VCV_EDIT_LINE_NUMBER 694
VCV_ENVIRONMENT_DIR 694
VCV_ENVIRONMENT_NAME 694
VECTOR_LICENSE_FILE 695
VECTORCAST_DIR 695
environment wizard
additional stubs 139
additional testable functions tab 141
choose compiler 105
choose UUTs & stubs 123
classes of interest tab 135
customized stubbing 128
driver prefix user code 144
get properties of a dependent unit 132
load an existing environment 106
load environment script 107
name environment 106
show dependency view 129
stub none using object files 126
suppressed stubs 140
suppressed testable functions tab 142
testing method 107
turn off data type support 142
turn off support for a data type 143
unit appendix user code 145
unit prefix user code 144
unresolved globals tab 133
update environment 107
user code 143-144
user globals 144
user params 144
equivalence matrices 511
escape backslashes in line directives 196
event
defined 18
event limit 433, 468
EVENT_LIMIT 433, 468, 686
EVENT_LIMIT_TYPE 434, 469
examples
user code in test script 681
executable file extension 189
EXECUTABLE_EXTENSION 189
execute all command 710
execute command 169
execute commands, CLICAST 400
execute help file 203
execute option
automatically clear test user code 442
fail if no expected return 437

executing compound test cases 398
executing test cases 21, 390
executing test cases with your debugger 405
execution report event header
    nested compound test case 268
execution reports
    enumerals 313
execution status viewer 58
exiting 27
exiting VectorCAST 27
expand all subprograms 496
expanding arrays 306-307
expected results
    range of values 688
    unused 396
    using ANY in list of values 688
expected user code comparisons
    width of 395
expected values
    any (placeholder) 314, 322
    defined 272, 277
    entering as a list 284
    entering as a range 284
    entering as a scalar 277
    range of 321
    set expected to actual 271, 409
export test scripts 327
external browser
    setting up 66
    using when viewing html files 410
external editor
    scrolling to first line of function 65
    setting up 63
    using in user code 599
    using in user code and Notes 282
    using when viewing text files 410
fail if no expected return 437
fail on unexpected exceptions 442
fail on unexpected signals 438
File menu
    Close 45
    Close All 48
    Close Environment 222
    Delete Environment 234
    Exit 27
    New 104
    new document 85
    Open 85, 222
    Print 87
    Print Setup 86
    Recent Environments 221-222
    Save 86
    Save All 86
    Save As 86
    Set Working Directory 103, 222
file version command 455
files
    .CFG 376
    .vcast-qt 38, 67-68, 70, 72
    AALINKER.LIS 164, 654
    CCAST_.CFG 160
    configuration 376
    created by environment 159
    environment_name.bat 224
    environment_name.cvr 224
    environment_name.env 224, 235
    environment_name.sh 224
    environment_name.tst 224
    exporting scripts 327
    GLOBALS.C 665
    importing 599
    importing in user code and Notes 281

importing scripts 328
opening 85
QuickParse 160
saving 86
saving automatically 71, 391
scripting log 328
TEST_DELETE.SAV 253-254
user code 600
USER_CODE_VCAST 609
VCAST_REBUILD_TEMP.env 233
viewing in external editor 63
filter
parameter tree 92
test case tree 92
filter Metrics report 446
find
tear off menu 93
find banner 89
finding text
in a text file 89
FLEXlm 652
floating point digits of precision 454, 467
floating point tolerance 434, 468
FLOATING_POINT_TOLERANCE 434, 468
fonts
changing for coverage output 543
default for coverage output 543
force explicit template arguments 210, 216-217
function
vcast_probe_assert 564
vcast_probe_print 565
vcast_probe_print_float 565
vcast_probe_print_int 565
vcast_probe_print_unsigned 565
vcast_probe_verbose 565
vcast_test_name_equals 565
function + function call coverage 484
function call report 418
function coverage 483
function entry probe points 553
function exit probe points 554
function return parameters 686
gcc 196
generate vct map 369
get properties of a dependent unit 132
global data
about 283
initializing in INIT test case 255
global objects 17, 686
global values
when to display 449
GLOBALS_DISPLAY 449
gridlines in test case, showing/hiding 68
groups
cascading 45
closing 44
closing all 48
explained 43
opening 44
switching between 44
gui options
display only referenced globals 67, 274
header and footer 87
header file extensions 193
Help menu
About 53
Available Licenses 51
Diagnostic
Test Compiler Settings 188
Diagnostics
Create Diagnostic Report 387
Troubleshooting 387

Tech Support 53
Tech Support 53
User Guides 52
hex notation for unprintable chars 435, 452, 466
import test script 328
importing
contents of file 599
importing coverage results 545
include directory
adding to an environment 222
include flags 171
incremental rebuild
incremental rebuild report 475
perform 475
indexing
enumerals 312
industry mode
automotive 30
avionics 30
changing 33
changing from branch, basis path or MC/DC 33
default 30
industrial 30
medical 30
railway 30
reports 32
setting the mode 30
status bar 32
INIT test cases
using to initialize global data 255
Initialize button 486
initializing global data
in INIT test case 255
input values 17
<<keep>> 284
defined 272, 277
entering as a list 284
entering as a range 284
entering as a scalar 277
understanding 278
Insert button
in List tab 319
instantiate all template functions during parsing 208
instrument unreachable code 525
instrumented code
recompiling 237
job status viewer 57
jobs monitor 55
execution status viewer 58
job status viewer 57
JSON Compilation Database
create new environment in wizard 111
keyboard navigation
Parameter Tree 275
shortcuts 48
Test Case Tree 246
language mode 192
Level A coverage 483
Level B coverage 480, 485
library include directories
definition 115
parse out of compile command line 184
library include directory
adding to source directories list 118
library interface testing 108, 149
environment wizard 149
Link Options in wizard 149
library stubs 137, 183
enable/disable 138
gray background 138

library stubs environment script
ENVIRO.LIBRARY_STUBS
<function> 138
Library Stubs tab
in creating new environment 137
license messages 652
licenses
determining which your organization has 51
limitations 690
link errors 164-165
linker command 189
linker errors
resolving 165
linker options 189-190
list
<<keep>> 284
entering data as 284
list of values for expected results 688
List Values tab 319
Load button 107
logic assignment 402
logic editor 401
logic file basics 405
logics file 401
looping 687
macro expansion longer than 5000 chars 487
map line selection 496
map test case 356
create test script 367
edit test script 367
export 356
import 356
mapping view 364
margins 87
max size of ASCII coverage data buffer 520
maximize button 45
maximum MC/DC Conditions 535
maximum number of test cases 248
maximum varied parameters 318
MC/DC 178-181, 480-481
maximum conditions 535
understanding analysis 510
viewing 379
MC/DC tests
automatically generate 513
MDI
symbolic contants window 288
MDI window
closing all files in 48
entering data directly into 273
Notes tab 281
Parameter Tree 272
member functions
making visible with Whitebox 110
message window
collapse 33
error tab 33
expand 33
undocking and docking 36
messages
<environment name> could not be created 654
<environment name> could not be deleted 654
<environment name> could not be opened 654
<environment> deleted successfully 658
<type mark> could not be found 655
Abnormal Termination of Compile and Link 653
after exporting coverage script 547
after importing coverage results 546
Building Driver 653

Building Harness for unit_name 653
Building Min Mid Max test cases 659
Building test case data 655
Building test case script file 657
Building test case script template 658
Building the CLI script 657
Calling QuickParse Utility for <directory_ name> 652
Cannot create min-mid-max or max test cases 659
Cannot Overwrite Environment 654
Clicast failed - parameter inconsistency 660
Clicast failed with unknown exception 660
Compile Failed, stubbed unit
unit name 653
Compile Failed, unit
unit name 653
Compiler not supported 660
Compiling unit name 653
Could not build test history 660
Could not open environment 658
Could not read ENVIRO.AUX, '<path>' is not an environment directory 656
Could not save Variant Logics path to environment 657
Could not update status for testcase logic assignment 656
Could not update test case name in Logic Assignment due to missing name 656
Could not update test case status in Logic Assignment due to missing name <logic name> 656
Creating environment script 658
Data interface could not be invoked 654
Data Program did not link 654
Deleting files from environment build 654
Determining basis paths 658
Driver could not be invoked 654
Driver Program did not link 654
ENVIRO.AUX path not set or does not exist 657
Environment build but not compiled 654
Environment build but not linked 654
Environment Built Successfully 653
Environment creation failed 654
Environment data not expanded 659
Environment Directory Creation Failed 653
Environment has not been compiled 659
Environment has unresolved compile errors 659
Environment has unresolved link errors 659
Environment must be specified on command line 660
Environment name is invalid on command line 660
Error compiling instrumented file 659
Error computing basis paths 658
Error evaluating Variant Logics file, evaluate file for details 657
Error evaluating Variant Logics file, refer to Logic Editor for details 656
Error initializing coverage 658
Error Must specify a variant logic. The variant logics currently available are <logic list> 657
Error retrieving Variant Logics Path from environment 656
FLEXlm Cannot find license file 652
FLEXlm Feature has expired 652
FLEXlm License failure 652
FLEXlm Licensed number of users already reached 652
Getting size and range data 658
Harness has run-time errors 659
Illegal numeric entry 655
instrumentation data line <number> 660

Instrumented Harness did not link 654
Invalid environment name 658
Invalid logic name entered, reset to previous value 656
Invalid test case name 655
Linking Environment 653
Linking Instrumented Harness 658
Logic Editor failed to save 657
Logic file changes detected, reloading logics 656
Logics reload failed 656
No change made to test data 655
No files specified 660
No results exist 655
Not in a VectorCast environment directory, thus Logics file not found 657
Parsing UUT body for whitebox conversion 652
Processing unit name 653
Processing UUT unit name 653
Processing whitebox data 652
QuickParse Utility Completed 652
Rebuilding instrumented harness 659
Script building completed 658
Sorting instrumentation data line <number> 660
Subprogram must be specified on command line 660
Subprogram name is invalid 660
Test case must be specified on command line 660
Test case name is invalid 660
test case variants 656
Test execution failed 655
Test Variant Logic Name '<logic name>' not found 657
Type not supported 655
Updating aggregate coverage report 659
Updating coverage data 659
Value out of range 655
Variant Logic '<logic name>' assigned to test case '<test case>;' 657
Variant Logic unassigned from test case ‘<test case>’ 657
Variant Logics path saved to CCAST_.CFG file 656
Variant Logics path saved to environment 656
VECTORCAST_DIR Environment variable is not set 652
metrics report 418, 500
METRICS_TOTAL_LEVEL_DISPLAY 446
min-mid-max 255
mixed-language environment 201
mixing C and C++ source files in one environment 195, 200
modifications
to source code and relinking 243
multiple value syntax 320
multi-return spans compound iterations 463
multi-return spans range iterations 463
nested compound
double-click nested test 268
nested compound test case 267
drop of compound test into itself error 269
execution report event header 268
recursion detection error 270
nested compound tests 267
new generation preprocessor 197
no change made to data 655
NO_FILTERING 446
Not Supported Types tab 142
notes
entering for test cases 281
Notes section template 282
notes section template 456

Notes tab 453
number of data partitions 434, 469
NUMBER_OF_DATA_PARTITIONS 434, 469
numeric types 285, 684
object file testing
environment wizard 146
Link Options in wizard 147
object file extension 189
object file testing 108, 146
only show errors in script logs” 438
only show global objects in one unit 207
open a console for standard I/O 439
opening
environment 221
files, scripts, environments, etc. 85
opening test cases 248
options
add a gnu system header marker 200
allow inlines for function pointers in c 220
alphabetize parameters 276
alphabetize test cases 68
always display coverage in reports 445
animation coverage type 520
animation play speed 524
ask before closing multiple tabs 71
assembler command 204
assembly file extensions 195
automatically generate concrete classes 209
automatically save test cases 71, 391
blank cells text 457
buffered coverage I/O 520
builder 205
test all member inlined functions 136
C compile command 201
C compile command flag 202
C compile flag to exclude 202
C file extensions 194
C parser flags 201
C preprocessor command 201
C/C++ 168
compile command 171
linker options 189-190
C++ file extensions 194
change application font 70
change background color 544
change fonts 543
change text color 544
combination testing 435, 469
command line debugger 191
command verb 181
compare expected before UUT call 452, 472
compile test-specific user code in separate unit 211
compiler args 203
compiler family name 203
Compiler Integration tab 201
compiler lacks long double 208
compiler supports c++style casts 217
compiler template 168
consider uncalled template functions testable 214
consider unit user code functions testable 212
convert octal/hex-encoded strings to ASCII 452
coverage 518
default fonts 543
coverage field width 537
coverage I/O types 519
coverage type 178
covered lines 543

create initialization objects dynamically 210
custom end macro 175, 705
database file 181
debug help file 203
debugger command 190
default compiler 204
default script header file 456
default string display mode 70
define flags 171
defined variables 174, 705
dependency data directory 181
deprecated
  C_INCLUDE_LIST 205
  VCAST_FORCE_NO_USERGLOBALS 221
  VCAST_STUB_ALL_SUBPROGRAMS 221
  VCAST_TORNADO_CONSTRUCTOR_CALL_FILE 221
detect thrown exception types 212
detect unused expected user code 443
disable assembly function testing 208
disable direct array indexing 209
disable exception handling 213
display full string data 439, 464
display global data after 449
display subprogram coverage 446
display uninstrumented mcdc expressions in reports 445
do not automatically generate basis paths 216, 379
do not detect ansi c++ smart pointers 215
do not detect ANSI C++ standard containers 215
do not detect ANSI C++ standard string types 214
do not detect ANSI C++ standard wstring types 214
EDG flags 192
enable function pointer type support 211
enable probe points 218
enable sbf capability for library stubs 217
enable sbf capability for non-testable functions 217
enable sbf capability for template functions 218
environment files 199
escape line directives 196
event limit 433, 468
executable file extension 189
execute 431
  floating point tolerance 434, 468
execute command 169
executehelp file 203
expand scalar array elements 443
external browser 66
external editor 63
fail on no expected values 442
fail on unexpected exceptions 442
fail on unexpected signals 438
file version command 455
filter Metrics report 446
floating point precision 454, 467
force explicit template arguments 210, 216-217
harness precompile script command 221
header file extensions 193
hex notation for unprintable chars 435, 452, 466
ignore incomplete tests 438
Ignore type qualifiers for testable objects 209
include flags 171
instantiate all template functions during parsing 208
instrument unreachable code 525

language mode 192
Language tab 191
library include directory 172
library stubs 183
linker command 189
Linker/Debug tab 188
max size of ASCII coverage data buffer 520
maximum covered subprograms 522
maximum directories added recursively 72
maximum MC/DC conditions 535
maximum MC/DC expressions 523
Misc tab 195
Mixed C/C++ tab 200
multibyte characters 30
notes section template 282, 456
number of data partitions 434, 469
object file extension 189
only show errors in script logs 438
only show global objects in one unit 207
open a console for standard I/O 439
output file flags 188
parse function templates 193
partially covered lines 543
post-process instrumented files command 538
post-process vcast_c_options.h command 539
precompile command 204
precompile extension 204
preprocessor command 170
preprocessor file 172
Preprocessor/Compiler tab 170
provide code coverage in header files 530
range check 432
real-time coverage I/O 519
redirect standard output 435
reduce memory overhead using global buffers 529
remember window sizes per environment 72
remove preprocessor comments 196
report 443, 457
report complexity column width 461
report coverage results column width 462
report date column width 461
report format 459
report header 456, 459
report notes column width 462
report results column width 461
report subprogram column width 460
report testcase column width 461
report unit column width 460
save data in ASCII format in memory 520
script always uses params 441
setting custom css 459
share class instances across UUTs 207
shift jis character encoding 30
show compiler/linker settings in the Full report 453
show constructors during test-object init 216
show gridlines in test case 68
show inlines as covered in all unit 530
show non-search directory globals 220
show only data with expected results 451, 472
show only events with expected results 451, 472
show true and false in for loops 524
show version of VectorCAST in reports 454
sort Metrics report by directory 444-445, 449
strict test case importing 442
stub by implementation 216, 219
stub dependencies 176
substitute code for c file 191

test all member inline functions 207
test all non-member inline functions 207
test values dictionary file 182
testable source directory 172
testing method 177
text report options 460
treat c++ catch blocks as branches 530
treat template-based structs as classes 213
type handled source directory 172
unconstrained array size 220
uncoverable lines 543
uncovered line indicator 523
uncovered lines 543
use alternate whitebox method 219
use as text file viewer 64
use cygpath to convert cygwin line directives 197
use default parameter arguments 211
use directory list for include directories 175
use DOS formatted files 64
use new generation preprocessor 197
use relative paths 182
use search list for library testing 212
use standard C string functions 206
use static memory allocation on the target 522
use Vector preprocessor 197
variant logics path 182
vcdb flag string 202
verbose management report 453, 679
whitebox 182
whitebox search directories only 218
wrap text at column width 397
wrap text in Notes 398, 462
Options tab
for test cases 686
original source code line numbers 496
output file flags 188
output values 17
overloaded member functions
naming of 246
overloaded subprograms
working with 689
overview 17
PAGE_WIDTH 395
pairs status 482, 501
parameter dialog box 314
parameter tree 356
alphabetize parameters 276
defined 272
display only referenced globals 67, 274
filter 92
filter indicators 92
keyboard navigation 275
parameter checkboxes 357
with stubbed-by-function unit 273
parameters
entering data for 314
function return 686
user code 596
Parse Command Line button 184
parse data
missing for a unit 122
parse function templates 193
parser 192
partially covered lines 543
pasting text 88
performance testing
real-time vs. instrumented 479
pointer types 313, 685
pointers 685
precompile command 204
precompile extension 204

precompile script command 221
PRECOMPILE_CMD 204
PRECOMPILE_EXT 204
preprocess file
    diagnostic uses of 188
preprocessor command 170
preprocessor file 172
print
    print preview 88
print preview 88
print setup
    coverage 87
    header and footer 87
    margins 87
    use backgrounds 87
Print Setup
    reverse foreground and background 87
print unit-specific arguments 243
printing
    files 87
    setting up 86
probe point events 567
probe point listing 572
probe points 552
    activate probe point 565
    apply all 562
    capture variable values 569
    deactivate probe point 565
    delete all probe points 566
    disable 579
    edit a probe point 563
    enable 578
    fault injection 570
    function entry probe points 553
    function exit probe points 554
    function probe points 552
    functions and macros 564
    import a probe point 576-577
    inject spurious values 570
    insert a file scope probe point 558
    insert a probe point 557
    matching line index 574
    order of execution 555
    patch faulty code 571
    probe id number 575
    probe point api 573
    probe point editor 556
    probe point file 573
    remove 579
    remove probe point 566
    saving and applying 561
    status buttons 563
    test compile 559
    xml report 579
prototype stubbing
    meaning of 19
Quickparse file 160
quitting 27
range
    entering data as 284
range check 432
range data
    rebuilding 243
range of values for expected results 688
range of values for test creation 687
range testing 318
range values tab 317
RANGE_CHECK 432
ranges as expected results 318
real-time coverage I/O 519
rebuild environment 232

recompiling
automatic 236
create script 237
instrumented code 237
run script 241
redirect standard output 435
redoing an edit action 88
regression scripts
creating 224
integrating with ClearCaseTM 230
post-processing automatically 228
regression test 224
relative paths 182
in Create New Environment dialog 120
relinking
test harness 243
remember last working directory 103
Remove All button
in List tab 320
Remove button
in List tab 320
Remove Directory button 121
remove preprocessor comments 196
removing
breakpoints 518
removing a custom tool 384
renaming
environment 222
renaming a test case 248
renaming test cases 248
Replace button
in List tab 319
report complexity column width 461
report components 81
layout template 82
report file 82
section file 82
report coverage results column width 462
report customization 72
report date column width 461
report format 459
report header 456, 459
report notes column width 462
report results column width 461
report subprogram column width 460
report testcase column width 461
report unit column width 460
reports
aggregate coverage report 417, 499
controlling the test execution report 396
coverage summary 22
creating 22
diagnostic 387
execution history 22
expand report objects 396
expand report parameters 396
expanding information included 396
full 415
metrics 418, 500
overview of environment 157
scripting log 328
summary of results 22
test case management 416, 679
test comparison 423
test data 22
viewing in external browser 67
requirements
displaying in report 679
setting up template to document 282
requirements gateway
authentication tab 632
command line 784

configure export 647
example workflow 643
export 648
export tab 635
import requirements 646
import tab 633
initialize repository 644
link requirements 638, 646
requirements repository 632
requirements tab 636
run test case 647
script tab 637
set configuration options 644
set repository 644
supported subsytems 642
tracking requirement change impacts 640
using rgw 631
requirements gateway commands
configure set 784
export 785
import 785
initialize 785
requirement report 785
set gateway 786
testcase link 786
testcase mark as reviewed 786
testcase migrate_links 786
testcase report 787
Reset Expected button
in Parameter User Code tab 617
Reset Input button
in Parameter User Code tab 617
restoring deleted test cases 254
rgw
see requirements gateway 631
rgw database location 169
right-click menu
(Array) Properties 308
Close (tab) 45
Collapse Unused Children 308
Compound Only 267
copy/clear text in Message window 35
Delete 252-253
Delete, from a Compound 266
Deselect Coverage 489
Duplicate Test Case 250
Execute 390, 400
Execute with Debug 405
Expand All Array Indices 308
Expand First Array Index 308
find slot in test case tree 266
Import file contents 281, 599
Import Template 282
Insert Basis Path Test Cases 257
Insert Min Mid Max 256
Insert user code tag 598
Insert usercode tag 615
Invoke external editor 281, 599
Invoke External Editor 282
Move Down, in a Compound 265
Move Up, in a Compound 265
Open Source 254
Open Test Case 248
Open, from a Compound 265
Rename 248
Select Coverage 489
Show (tab) 48
String Display Mode 297
toggle Message window 34
View Coverage 254
View Coverage, from a Compound 266

run new reports 83
from command line 83
from gui 83
Save and Link button 605, 626
save backup copies 253
Save button 105
save control flow 409
save data in ASCII format in memory 520
Save Only button 605
saving
files 86
test cases automatically 71, 391
saving a file to another name 86
SBF 125
meaning of 19
scalar values tab 314
scientific notation
supported for real numbers 287
script always uses function parameters 441
scripting 326
exporting for regression test 547
scripting utility
building test cases with scripts 85, 326
clear unapplied test script data 328
duplicate names in exported scripts 327
exporting scripts 327
importing scripts 328
opening scripts 85
setting test values 683
scripts
environment, creating 85
test case, creating 85
search
case sensitive 90
close 91
execute 89
match case 90
next instance 90
previous instance 90
repeat 91
search directories
definition 115
troubleshooting 122
search directory
adding to an environment 222
adding to source directories list 115
search list
adding directories from here down 120
removing parse data from 121
source files have not changed 122
searching 89
segmentation violation
in execution results 259
set display theme 62
set expected values to actual values 271, 409
set working directory 103
setting default source directories for wizard 172-174
setting the industry mode 30
share class instances across UUTs 207
shell script 224-226, 229, 693
shortcut keys 48
show constructors called during test-object init 216
show inlines as covered in all units 530
show only data with expected results 451, 472
show only events with expected results 451, 472
show true and false in for loops 524
show version of VectorCAST in reports 454
SHOW_NOTES_IN_FULL_REPORT 454
smart stubs 18
sort Metrics report by directory 444-445, 449
sort test cases alphabetically 248

source code
incorporating changes of into environment 232
making modifications to and relinking 243
source code file
opening 86
source directories
already present in Wizard 115
clearing defaults for wizard 174
definition 115
setting defaults for wizard 172, 174
setting defaults for wizard, command line 173
source directories list
adding library include directory 118
adding search directory 115
adding type-handled directory 119
source files have not changed checkbox 122
SOURCE_EXTENSION 192
standard I/O 439
STANDARD_ERROR 435
STANDARD_OUTPUT 435
starting VectorCAST 24
starting VectorCAST and opening environment 24
starting VectorCAST and opening options dialog 24
starting VectorCAST in C or Ada mode 26
starting VectorCAST, building and opening environment 24
startup file 205
start-up messages 652
statement coverage 479
Statement coverage 480
STATEMENT+BRANCH 485
Statement+Branch coverage 485
STATEMENT+MC/DC 483
Statement+MC/DC coverage 483
static functions
making visible with Whitebox 110
status bar 43
STL
examples 298
vectors, maps, lists, pairs, etc. 298
STL containers
adding elements to parameter tree 299
behaviors 305
list 299
standard element names 300
STL Containers
back 300
first 300
front 300
second 300
top 300
strict test case importing 442
string display mode 297
string types 296-297, 684
structure types 291, 684
Stub by function
meaning of 19
stub by implementation 219
stub dependencies 176
all (by prototype) 125
all by prototype 160
choosing 125
custom 125
none 125
stub none using object files 126
stub user code
compiling 627
stubbed-by-function
expected values 279
how to specify stubbed function 279

in Create New Env wizard 125
input values 279
Parameter Tree of 273
stubs
meaning of 18
sub-directories of the Search List are considered 216
SUBPROGRAM_DETAIL 447
subprograms
overloaded 689
viewing coverage of 254
SUBSTITUTE_CODE_FOR_C_FILE 191
support for C++ STL Types 298
suppressed coverable functions 540
suppressed stubs environment script
ENVIRO.SUPPRESS_STUB <function> 141
Suppressed Stubs tab 140
symbolic constants
test values dictionary 290
test-only 290
window 287
T
indicating a branch covered 492
tab delimited file 355
tabs
closing 45
closing all 48
closing the current 45
showing from behind 48
tech support
troubleshooting VectorCAST 387
technical support
contacting 53
FTP environment to 53
template
for exported test scripts 456
for Notes tab 456
for user code 456
using in user code and Notes 282
template file
creation 356
csv 356
tab delimited 356
templates 106, 329
test all member inline functions 207
test all member inlined functions 136
test all non-member inline functions 207
Test button 187
test case
(MAP) 356
ATG 336
automatic test case generation 336
coded tests 344
create from CSV Map 365
CSV File 355
global constants 287, 290
import/exprt map test case 356
macros 287, 290
map test 356
meaning of 18
tab delimited file 355
test values dictionary 290
VCT editor 369
test case editor
enumerals 313
test case execution
detect unused expected user code 443
expand scalar array elements 443
fail on unexpected signals 438
ignore incomplete tests 438

no expected values 442
strict importing 442
test case notes
displaying in report 679
in MDI window 281
test case options
multi-return spans compound iterations 463
multi-return spans range iterations 463
test case script
opening 85
test case scripts
enumerals 313
Test Case Tree 35
filter 92
navigation 246
test case tree
alphabetize 249
changing search term for filter 94
coverage viewer 489
filter indicators 93
filter test case reports 94
filter to selectively apply coverage 98
filtering to export test scripts 96
green and red arrows 391
sort 249
sort alphabetically 250
sort subprograms 249
test cases
aborting execution 405
automatic range testing 687
automatically naming 248
batch execute all 400
building 21
building compound 261
building from basis path analysis 507
compound only 267
creating INIT 255
creating regression test for 224
creating simple 247
definition of compound 247
definition of simple 247
deleting 252-253
deleting from Compound 266
deleting test case part of compound 252
duplicating 250
entering data 272
entering data for 277
executing 21, 390
executing compound 398
executing with debugger 405
find slot, from Compound 266
generating automatically for full basis path 257
management report 416
maximum number of 248
Min-Mid-Max 255
nested compound 267
opening for editing 248
opening, from Compound 265
rearranging order of, in Compound 265
renaming 248
restoring deleted test cases 254
saving automatically 71, 391
saving control flow 409
scripting language 674
select next passing or failing test case 391
selecting multiple 247
selecting subprogram 247
sorting alphabetically 248
specifying input values in scripting 683
specifying options in scripting 686
specifying return parameters in scripting 686

types of 247
user code 596
viewing 410
viewing coverage of 254
viewing coverage of, from Compound 266
viewing execution results 412
Test Compile button 599, 603, 610, 625
Test Compile Expected button 615
in Parameter User Code tab 617
Test Compile Input button 615
in Parameter User Code tab 617
test data editor 369
edit vct 370
export import 375
generate automated tests 374
generate map 369
generate test case data 372
test data summary 419
test driven development 108, 151
example 151
test driven environment
add implementation 155
create 152
source file checkbox 156
subprogram icon 153
test driver
meaning of 18
test execution
animated 515
test execution report
controlling data displayed 273, 277, 396
test executions
specifying the number of iterations of 262
test harness
components 18
creating a recompile script 237
edit link options 242
editing of 377
incorporating source code changes into 232
meaning of 17
recompiling 236
recompiling without reinstrumenting 237
relinking 243
running a recompile script 241
Test menu
Batch Execute All 400
Control Flow
Clear 409
Save 409
Delete All 253
Execute 390
Scripting
Duplicate names in exported scripts 327
Export Script 327
Import Script 328
Template 329
User Code
Add All 616
Clear All 616
View 410
Aggregate Coverage Report 417, 499
Execution Results 412
Full Report 415
Raw Data Set 431
Scripting Log 328
Test Case Data 411
test menu
coverage data used in reports 410
test requirements 281
test script header template 456

Test Script Log 328
test scripts
automating maintenance 331
comparing for different versions of source code 327
conversion processing 336
convert multiple solutions 332
convert rebuilt environments with no test cases 334
delete translation 334
internal organization 327
language 326
sorted alphabetically 327
test script converter 331
view messages from last translation 335
view test script log 334
view test script maintenance difference listing 335
Test Settings button 185
test values
spanning multiple lines 688
test values dictionary 290
TEST.ADD 674
TEST.ATTRIBUTES 674
TEST.AUTOMATIC_FINALIZATION 674
TEST.AUTOMATIC_INITIALIZATION 675
TEST.BASIS_PATH 675
TEST.COMPOUND_ONLY 675
TEST.CSV_COLUMN_INFO 675
TEST.CSV_DELIMITER 676
TEST.CSV_FILENAME 676
TEST.CSV_HEADER_ROWS 676
TEST.CSV_NAME_COL 676
TEST.CSV_NOTES_COL 676
TEST.CSV_ROWS_PER_TESTCASE 676
TEST.END 676
TEST.EXPECTED 676
TEST.EXPECTED_USER_CODE 677
TEST.FLOATING_POINT_TOLERANCE 677
TEST.FLOW 678
TEST.IMPORT_FAILED 678
TEST.MCDC_BASIS_PATH 678
TEST.NAME 678
TEST.NEW 674
TEST.NOTES 679
TEST.REMOVE 674
TEST.REPLACE 674
TEST.REQUIREMENT_KEY 679
TEST.SLOT 679
TEST.STUB (by function) 680
TEST.STUB_EXP_USER_CODE 680
TEST.STUB_VAL_USER_CODE 680
TEST.SUBPROGRAM 680
TEST.UNIT 681
TEST.VALUE 681, 683
TEST.VALUE_USER_CODE 681
TEST.VARIANT_LOGIC_NAME 683
TEST_DELETE.SAV 253-254
testable class
meaning of 19
testcase
options
COMPARE_BEFORE_UUT 452, 472
EVENT_LIMIT 433, 469
FLOATING_POINT_DIGITS_OF_PRECISION 455, 468
FLOATING_POINT_TOLERANCE 434, 468, 678
GLOBAL_DATA_DISPLAY 450
HEX_NOTATION_FOR_UNPRINTABLE_CHARS 467
MULTI_RETURN_SPANS_RANGE 463

MULTI_RETURN_SPANS_TESTCASES 464
NUMBER_OF_DATA_PARTITIONS 435
SHOW_ONLY_DATA_WITH_EXPECTED_RESULTS 451, 472
SHOW_ONLY_EVENTS_WITH_EXPECTED_RESULTS 451, 473
testing custom tool 383
testing method 177
library interface testing 108
object file testing 108
test driven development XE 108
traditional unit testing 107
test-only symbolic constants 290
text colors
changing for coverage output 544
tiling groups 46
toolbar
button 288
coverage animation 516-517
New 104
Open button 85, 221
Print button 87
print preview 88
Redo button 88
Save All button 86
Save button 86, 605, 608, 611, 626
Undo button 88
Tools
MC/DC
View Analysis Report 379
options 376
Tools menu
Basis Path
Edit Script 379
View Analysis Report 378, 508
Coverage
Delete Imported Results 550
Disable 488
Enable 488
Import Script 550
Custom Tools 380
Edit Harness Source 377
MC/DC
Test Case Analysis 513
View Equivalence Matrices 511
menu selections 377
Options dialog 376
traditional testing method 107
treat c++ catch blocks as branches 530
treat template-based structs as classes 213
troubleshooting
<<unknown>> values in template 330
cannot backup directory 234
environment data not expanded 330
error while parsing or preprocessing 161
exit is undefined 175
parse error during rebuild 233
search directories 122
segmentation violation caused by Min, Mid, or Max values 259
spaces in working directory 104
suppressing some statement in MC/DC coverage 482
VectorCAST 387
true branch 492
turn off data type support 142
type-handled directories
definition 115
type-handled directory
adding to source directories list 119

types
array 685
character 684
enumeration 684
numeric 684
pointer 685
standard containers 215
standard string 214
standard wstring 214
string 684
structure 684
unconstrained array size 220
unconstrained array types 685
unconstrained arrays 313
uncoverable lines 543
uncovered line indicator 523
reinstrument 523
uncovered lines 543
undefined entities log file 165
undefined symbol VCAST_exit 175
understanding input and expected values 278
undoing an edit action 88
uninstrument coverage 488
union types 292
unit appendix user code
how to edit 609
Unit Options tab
in creating new environment 134
Unit Under Test
choosing 124
meaning of 18
UNIT_DETAIL 448
units
viewing coverage of 254
units under test 125
unresolved globals tab 133
Up button
in List tab 320
updating an environment 222
use alternate whitebox method 219
use as text file viewer 64
use default parameter arguments 211
use DOS Formatted Files 64
Use Relative Paths checkbox 120
use search list for library testing 212
use standard C string functions 206
user code
add all 616
adding to a test script 681
clear all 616
configure stubs
saving and linking 626
Stub Dependency column 624
test compiling 625
display actual value in execution report 620
editing 600
entering for test cases 615
environment
additional unit bodies 602
additional unit specs 602
clearing 614
compiling 609
data 601
global data 606
harness init 601
harness term 602
header 601
saving and linking 605
stub entry 601
stub exit 602

stub processing 601
test case init 601
test case term 602
test compiling 603
UUT timer start 601
UUT timer stop 601
importing contents of a file 599
order of execution 596
stub
clearing 627
entering 623
types of 596
unit appendix
test compiling 610
user appendix 609
using external editor 599
using external editor for 66
User Code tab
for a parameter 617
user code template 456
user globals
adding 606
changing the default 606
default 606
how to edit 606
introduction 606
saving and linking 608
test compiling 607
User Globals in environment scripts 665
user guides
HTML 52
PDF filenames for 52
USER_CODE_VCAST 609
UUT 125
meaning of 18
V(g) 378, 508
variants
clicast messages 657
control test case execution using variant logic 400
enable disable a test case 250
error messages 656
information messages 656
logic assignment 402
logic file 400
logic file basics 405
variant logic 400
VCAST.QIK 160
VCAST_ACCESS_NON_SEARCH_GLOBALS 220
VCAST_ALTERNATE_REPORT 691
VCAST_ASSEMBLY_FILE_EXTENSIONS 195
VCAST_C_FILE_EXTENSIONS 194
VCAST_COMMAND_LINE_DEBUGGER 191
VCAST_CONSOLE_REPORT_FORMAT 698
VCAST_CONVERT_OCTAL_HEX_STRINGS_TO_ASCII 452
VCAST_COVERAGE_FOR_DECLARATIONS 531
VCAST_COVERAGE_IO_ANIMATION 520
VCAST_COVERAGE_IO_BUFFERED 520
VCAST_COVERAGE_IO_REAL_TIME 519
VCAST_CPP_FILE_EXTENSIONS 194
VCAST_CUSTOM_END 705
VCAST_CUSTOM_REPORT_FORMAT 459
VCAST_CYGPATH_LINE_DIRECTIVES 197
VCAST_DEBUG_LEVELS 691
VCAST_DEBUG_SCRIPT 691
VCAST_DETECT_UNUSED_EXPECTED_UC 443
VCAST_DISABLE_FILE_DELETE 691
VCAST_DISABLE_STD_SMART_PTR_DETECTION 215

VCAST_DISPLAY_EMPTY_COVERAGE 445
VCAST_DISPLAY_FUNCTION_ COVERAGE 446
VCAST_DISPLAY_UNINST_EXPR 445
VCAST_DO_COMBINATION 435, 469
VCAST_DO_MCDC 482
VCAST_DONT_DO_MCDC 482
VCAST_DONT_INSTRUMENT_END 487
VCAST_DONT_INSTRUMENT_START 487
VCAST_ENABLE_PROBE_POINTS 557
VCAST_END_WAIT 691
VCAST_ENVIRONMENT_FILES 199
VCAST_ESCAPE_LINE_DIRECTIVES 196
VCAST_exit, undefined symbol 175
VCAST_EXPECTED_BEFORE_UUT_ CALL 452, 472
VCAST_FILE_VERSION_COMMAND 455
VCAST_FLOAT_PRECISION 454, 467
VCAST_FORCE_OBJECT_EXTENSION 692
VCAST_FULL_STRINGS 439, 464
VCAST_FUNC_PTR_INLINES_C 220
VCAST_GH_INTEX_CMD 190
VCAST_GLOBAL_DEFINE_LIST 692
VCAST_GNU_SYSTEM_MARKER 200
VCAST_HEADER_FILE_EXTENSIONS 193
VCAST_HEX_NOTATION 435, 466
VCAST_IGNORE_COMPILE_ERRORS 692
VCAST_IGNORE_FILE_EXISTS_CHECK_ FOR_COVERABLE 692
VCAST_IGNORE_INCOMPLETE_TESTS 438
VCAST_IGNORE_TYPEMARKS 692
VCAST_LEGACY_CFG_FILE_ INHERITANCE 692
VCAST_MAX_LINE_LENGTH 487, 692
VCAST_NEVER_STUB_LIST 663, 693
VCAST_NO_PARSE 693
VCAST_NOTES_SECTION_TEMPLATE 282, 456
VCAST_NUM_JOBS 693
VCAST_OLD_STYLE_MANAGEMENT_ REPORT 453
vcast_probe_print 564-565
vcast_probe_print_float 565
vcast_probe_print_int 565
vcast_probe_print_unsigned 565
VCAST_QIK_ASSUME_SRC_HAS_NOT_ CHANGED 693
VCAST_REBUILD_TEMP.env file 233
VCAST_RECURSION_DEPTH 693
VCAST_REGRESSION_COMMAND 693
VCAST_REMOVE_PREPROCESSOR_ COMMENTS 196
VCAST_REPOSITORY 169
VCAST_RPTS_COMPLEXITY_COLUMN_ WIDTH 461
VCAST_RPTS_COVERAGE_RESULT_ COLUMN_WIDTH 462
VCAST_RPTS_CUSTOM_CSS 459
VCAST_RPTS_DATE_COLUMN_WIDTH 461
VCAST_RPTS_EMPTY_DATA_STRING 457
VCAST_RPTS_HEADER 456, 459
VCAST_RPTS_NOTES_COLUMN_WIDTH 462
VCAST_RPTS_PRETTY_PRINT_HTML 693
VCAST_RPTS_RESULT_COLUMN_ WIDTH 461
VCAST_RPTS_SHOW_VERSION 454
VCAST_RPTS_SUBPROGRAM_COLUMN_ WIDTH 460
VCAST_RPTS_TESTCASE_COLUMN_ WIDTH 461
VCAST_RPTS_UNIT_COLUMN_WIDTH 460
VCAST_RPTS_WRAP_NOTES 398, 462
VCAST_RSP_USER_TEMPLATE 693
VCAST_SCRIPT_ALWAYS_USES_ PARAMS 441
VCAST_SCRIPT_EXPAND_ARRAYS 443

VCAST_SCRIPT_LOG_SHOW_ONLY_ERRORS 438
VCAST_SHOW_ONLY_DATA_WITH_EXPECTED_VALUES 451, 472
VCAST_SHOW_ONLY_EVENTS_WITH_EXPECTED_VALUES 451, 472
VCAST_SHOW_STDOUT_CONSOLE 439
VCAST_SORT_METRICS_RPT_BY_DIR 444-445, 449
VCAST_STRICT_TEST_CASE_IMPORT 442
vcast_test_name_equals 565
VCAST_TESTCASE_FAIL_ON_NO_EXPECTED 442
VCAST_TORNADO_OLD_SCRIPT 694
VCAST_UNEXPECTED_EXCEPTIONS_FAIL 442
VCAST_UNEXPECTED_SIGNALS_FAIL 438
VCAST_USE_DIR_LIST_FOR_INCLUDES 175
VCAST_USE_EDG_PREPROCESSOR 197
VCAST_USE_VCPP 197
VCAST_VCDB_FLAG_STRING 202
VCAST_VERBOSE_MANAGEMENT_REPORT 453
VCAST_WHITEBOX 694
VCAST_WRAP_TEXT_REPORT 397
vcdb flag parameter 202
vcshell
  create new environment in wizard 110
VCT editor 369
  edit vct 370
  export import 375
  generate automated tests 374
  generate map 369
  generate test case data 372
VCV_EDIT_LINE_NUMBER 65, 694
VCV_ENVIRONMENT_DIR 694
VCV_ENVIRONMENT_NAME 694
Vector classification tree editor 371
Vector preprocessor 197
VECTOR_LICENSE_FILE 695
VectorCAST
  exiting 27
  starting 24
  starting and opening environment 24
  starting and opening options dialog 24
  starting in C or Ada mode 26
  starting, building and opening environment 24
  troubleshooting 387
VECTORCAST_DIR 652, 695, 697
verbose management report 453, 679
version control 224
version number
  obtaining 53
View menu
  Dock control 34-35
viewing
  execution results 412
  full report 415
  raw data file 431
  test cases 410
  test comparison report 423
visible subprogram
  meaning of 18
void pointer types 293
whitebox 182
  changes made to source 110
  whitebox search directories only 218
Window menu
  Cascade 45
  Group 44
  Tile 46-48
windows
  closing all 48

windshell script
    opening 85
working directory 222
    in status bar 43
    in Use Relative Paths checkbox 120
    invalid 103
    remembering last 103
    setting 103
    uses of 103
    when starting from command line 103
    when starting from Start menu 103
wrap text at column width 397
wrap text in Notes 398, 462