

# Wind River® Workbench Code Coverage Analyzer

USER'S GUIDE

3.0

---

Copyright © 2007 Wind River Systems, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means without the prior written permission of Wind River Systems, Inc.

Wind River, the Wind River logo, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. Any third-party trademarks referenced are the property of their respective owners. For further information regarding Wind River trademarks, please see:

<http://www.windriver.com/company/terms/trademark.html>

This product may include software licensed to Wind River by third parties. Relevant notices (if any) are provided in your product installation at the following location:  
*installDir\product\_name\3rd\_party\_licensor\_notice.pdf*.

Wind River may refer to third-party documentation by listing publications or providing links to third-party Web sites for informational purposes. Wind River accepts no responsibility for the information provided in such third-party documentation.

---

#### **Corporate Headquarters**

Wind River Systems, Inc.  
500 Wind River Way  
Alameda, CA 94501-1153  
U.S.A.

toll free (U.S.): (800) 545-WIND  
telephone: (510) 748-4100  
facsimile: (510) 749-2010

For additional contact information, please visit the Wind River URL:

<http://www.windriver.com>

For information on how to contact Customer Support, please visit the following URL:

<http://www.windriver.com/support>

# Contents

<b>1</b>	<b>Introduction .....</b>	<b>1</b>
1.1	Introduction .....	1
	Code Coverage Analyzer Overview .....	2
	Workflow Highlights .....	3
1.2	Architectural Summary .....	4
1.3	Features .....	5
<b>2</b>	<b>Getting Started .....</b>	<b>7</b>
2.1	Introduction .....	7
2.2	Requirements .....	8
	VxWorks .....	8
	Linux .....	8
	All Targets .....	9
2.3	Creating a Project .....	10
2.4	Instrumenting and Compiling Your Code .....	11
2.5	Starting Code Coverage Analyzer .....	11
	Connect to Target Dialog Box .....	12
	Start the Target Test Code .....	15

2.6	Viewing Coverage Data .....	17
	Coverage Summary View .....	17
	Source Code Viewer .....	18
	Trend Graph View .....	19
	Distribution Graph View .....	19
	Console View .....	20
	Color Configuration .....	21
	Create HTML Report Dialog Box .....	21
	Stopping Data Collection .....	23
2.7	Example Code Coverage Analyzer Session .....	24
3	Instrumenting Source Code .....	27
3.1	Introduction .....	27
3.2	Specifying Instrumentation Parameters .....	28
	Coverage Types Tab .....	29
	Data Storage Tab (Linux Only) .....	30
	Covered Files Tab .....	31
	Other Options Tab .....	33
	Saving the Instrumented Code Files .....	35
3.3	Instrumenting and Compiling .....	35
	From Workbench .....	36
	From Your Own Makefile .....	37
	From a Command-Line Window .....	37
	Instrumentation Issues .....	38
3.4	Downloading Your Object Code .....	39
4	Viewing Output .....	43
4.1	Introduction .....	43
4.2	Starting Data Collection .....	44
4.3	Viewing Live Coverage Data .....	45
	4.3.1 Coverage Summary View .....	45

	Saving Output Data .....	48
	Deleting Saved Data Files .....	48
4.3.2	Source Code Viewer .....	49
	Benefits of Multiple Coverage Selection .....	50
	Searching for Source Code Files .....	51
	Specifying Source Paths .....	52
4.3.3	Trend Graph View .....	54
4.3.4	Distribution Graph View .....	55
4.3.5	Coverage Report .....	56
4.3.6	Merge Data Files .....	58
	Merge Log .....	60
	Saving Merged Files .....	60
4.4	Viewing Saved Data .....	61
4.5	Exporting Data .....	62
<b>5</b>	<b>Using the Command-Line Interface .....</b>	<b>65</b>
5.1	Introduction .....	65
5.2	Commands .....	66
	coverageupload .....	66
	coverageconvert .....	68
5.3	Procedures .....	70
5.4	Example Script Files .....	72
	Shell Script File: testcovdemo.sh .....	72
	TCL Script File: runcoveragetests.tcl .....	73
	TCL Script File: killtgtsvrs.tcl .....	74
<b>6</b>	<b>Troubleshooting .....</b>	<b>75</b>
6.1	Introduction .....	75
6.2	GUI Messages .....	75

	Status .....	76
	Error .....	76
<b>6.3</b>	<b>Command-Line Interface Messages .....</b>	<b>76</b>
	Warning .....	77
	Error .....	77
<b>6.4</b>	<b>Troubleshooting Tips .....</b>	<b>79</b>
	Issues with Instrumentation .....	79
	Issues with the Target .....	79
	Issues with the GUI .....	79
<b>A</b>	<b>Code Coverage Types .....</b>	<b>81</b>
<b>A.1</b>	<b>Introduction .....</b>	<b>81</b>
<b>A.2</b>	<b>Purpose of Code Coverage .....</b>	<b>82</b>
<b>A.3</b>	<b>Types of Coverage .....</b>	<b>83</b>
	Function .....	83
	Function Exit .....	83
	Block .....	84
	Decision .....	86
	Condition .....	88
<b>A.4</b>	<b>Coverage Type Hierarchy .....</b>	<b>89</b>
<b>B</b>	<b>Performance Metrics .....</b>	<b>91</b>
<b>B.1</b>	<b>Introduction .....</b>	<b>91</b>
<b>B.2</b>	<b>Supported Targets Data .....</b>	<b>92</b>
<b>C</b>	<b>Glossary .....</b>	<b>95</b>
	<b>Index .....</b>	<b>99</b>

# 1

## *Introduction*

1.1	Introduction	1
1.2	Architectural Summary	4
1.3	Features	5

### 1.1 Introduction

Wind River Code Coverage Analyzer is a run-time code test coverage analyzer from Wind River for use in developing embedded software. Wind River Code Coverage Analyzer tracks your target source code, and determines the percentage of code that has been executed by a software test case, pointing you to the sections of code that have not been fully tested. The real-time output display reports on the branches and blocks in a code segment that are, and are not, actually traversed during the software test. Code Coverage Analyzer provides you with a definable degree of confidence that your code has been thoroughly tested before you make it available to your customers.



---

**NOTE:** This document contains background information and process descriptions only. Detailed help with user interface operations is available by pressing the help key for your host while running Code Coverage Analyzer.

---

## Code Coverage Analyzer Overview

This section provides an overview of the Code Coverage Analyzer operation.

### Mode of Operation

Code Coverage Analyzer is started directly from the Workbench development platform. It runs as a graphical user interface (GUI) view, integrated with all the other debugging tools and views available in Workbench. All parameters affecting the workflow are set up in dedicated Workbench menus, and are saved and made available for on-going testing projects.

### Types of Coverage

During the execution of a software test case, Code Coverage Analyzer monitors the execution of every branch and block of statements in your code. Its color-coded graphics and corresponding text output, displayed in real time, enable you to determine the parts of your code that have (and just as importantly, have *not*) been covered by the test case.

The following conditions in your code can be explicitly identified and reported:

- When each block of non-branching code has been executed.
- When a function has been entered.
- If each function exit has been taken.
- If both the **TRUE** and **FALSE** branches of a decision statement are taken.
- If each subexpression in a Boolean statement has separately evaluated to both the **TRUE** and **FALSE** conditions.

These software test case coverages give you positive proof that thorough software testing has, or has not, taken place. With this information, you can modify the test cases to include missed areas as you work toward code that is 100% tested.

### Dynamic Coverage Analysis

Code Coverage Analyzer gives you a dynamic real-time view of the performance of your software with a graph of the percentage of code block elements that have been traversed in a software test case over a given time period. The use of color coded output highlights the results as the software test progresses. The capability of displaying the source code containing a block in question can also help determine why it was or was not covered in the software test case. To maximize the degree of assurance you are seeking, Code Coverage Analyzer lets you



incorporate only the types of coverage you want into the instrumented and compiled source code.

### Post-Analysis Report

When the software test is complete, a comprehensive Code Coverage Analyzer report can be generated, allowing you to review analyses of coverage percentages by function, as well as for each type of coverage. This HTML-formatted report can help you to redesign your test case to ensure the complete and comprehensive testing of your software product.

### Command Line Interface

A command-line interface (CLI) version of Code Coverage Analyzer is also available. This implementation is designed to support users with a very large code base that has memory requirements exceeding the capacity of the GUI, and it allows you to collect and analyze the same coverage data without having to use the GUI. It also provides an easy means to generate script files for automated testing, or for adding to your existing test scripts. The command-line interface is described in detail in Chapter 5. [Using the Command-Line Interface](#).

### Workflow Highlights

Code Coverage Analyzer opens a GUI view where you select the coverage types you want to check, as well as other instrumentation parameters. Other optional parameters allow you to control the printing of status or error messages, set the extent of instrumentation in source code and included files, and select compile stage intermediate files to be saved for later analysis.

The specific test coverage types you can select are:

1. **Function**—Shows whether each function has been executed at least once.
2. **Function Exit**—Shows whether every exit from a function has been taken.
3. **Block**—Shows whether each non-branching block of code has been executed at least once.
4. **Decision**—Shows whether both the **TRUE** and **False** branches of each decision statement have been executed at least once.
5. **Condition**—Shows whether each subexpression in each Boolean statement has been separately evaluated to **TRUE** and **False** at least once.

When you recompile your target code, the coverage instrumentor adds tags to the code according to options you selected in the GUI. Then, back in the GUI, you use your test program to execute the instrumented and recompiled target code, viewing the coverage results in the GUI as the test progresses.

The instrumentation tags for the coverage types you selected cause data to be gathered from the code as it is executed by this software test. A variety of color-coded real-time coverage summary dialog boxes are generated and displayed as the test case progresses.

Upon completion of the test case, you can generate a formal report of the coverage analysis performed on your target code. This report can be configured to include only the data you want to present.

## 1.2 Architectural Summary

The Code Coverage Analyzer elements consist of the GUI, a compile-time instrumentation program, and an optional target agent.

The GUI enables you to carry out the following activities:

- Create, modify, and maintain instrumentation parameters.
- Start and stop data collection.
- Select from the many optional output viewing modes.
- Load or delete previously saved data files.
- View and analyze the coverage results.
- Merge coverage data files.

The code instrumentation program **coverage** is run outside the GUI and does the following:

- Adds instrumentation tags to the source code files as determined from the instrumentation parameters you selected.
- Compiles the source code files, or a selected subset of them.



**NOTE:** The coverage code instrumentor supports only ANSI C standard C/C++ code compiled with the **gcc** or Wind River compiler.

---

## **1.3 Features**

Code Coverage Analyzer includes the following features:

### **GUI Format**

GUI views are intuitive and easy to use in setting up Code Coverage Analyzer run-time parameters.

### **Color-Coded Output**

While the software test is running, Code Coverage Analyzer generates real-time output data records and graphs with the different coverage types color-coded for easy identification.

### **Web-based HTML Report**

A printable HTML text report, easily generated from the output files, can be configured to show only the data you want to see.

### **Source Code View**

Code Coverage Analyzer enables you to view source code corresponding to any file or function displayed in the coverage tree view by clicking on the screen entry. Viewing the source code can help you better understand how to modify software test case parameters to ensure that the code gets tested.



# 2

## Getting Started

- 2.1 Introduction 7
- 2.2 Requirements 8
- 2.3 Creating a Project 10
- 2.4 Instrumenting and Compiling Your Code 11
- 2.5 Starting Code Coverage Analyzer 11
- 2.6 Viewing Coverage Data 17
- 2.7 Example Code Coverage Analyzer Session 24

### 2.1 Introduction

This chapter takes you through the process of setting up and running Wind River Code Coverage Analyzer on a VxWorks or Linux target platform. It gives you enough information to begin using Code Coverage Analyzer with the demonstration program supplied. At each step references are made to the location in this manual of more detailed descriptions. For more information on using Workbench, see the *Wind River Workbench User's Guide*.

## 2.2 Requirements

You must connect to the target manager for your target in the Remote System view in order to use Code Coverage Analyzer. For more information on the Remote System view, see the *Wind River Workbench User's Guide: Remote System View*, as well as your platform User's Guide.

There are some dependencies Code Coverage Analyzer places on your host operating system for resources that are specific to the target platform, summarized in the following sections.

### VxWorks

- Code Coverage Analyzer uses the WDB target agent. The easiest way to ensure that your VxWorks Image Project (VIP) has the WDB support it needs is to make sure one of the following kernel configuration profiles is used in your project:
  - PROFILE\_COMPATIBLE
  - PROFILE\_DEVELOPMENT
  - PROFILE\_ENHANCED\_NET



---

**NOTE:** Workbench does not support using build extensions to instrument VIP-type projects. To instrument these types of projects, you must use your own makefile, or use the command-line window. For more information, see the *Wind River Workbench User's Guide*.

---

- Wind River Run-Time Analysis Tools do not support connecting to a target using a **WDB\_TIPC** connection. This means that if you are working in an **AMP** environment, you can only connect the Code Coverage Analyzer to core 0 in AMP mode.

For more information, see *Wind River Workbench User's Guide: VxWorks Image Projects*.

### Linux

In the process of building your target root file system, the binary files needed for the target you are using are copied to the following directory:

/usr/scopetools-6.0

If you should see a file in that directory with a name like the one formerly used to specify your specific architecture (that is, target type/platform/compiler, such as **ppc85xxGPP1.4gcc4.1.2**), it is an empty file and should be disregarded completely.

## All Targets

If you have not yet created a target server connection, you should do that now in the **Target Manager** view of the Workbench GUI. For detailed instructions on how to do this, see the *Wind River Workbench User's Guide: New Target Server Connections*. Once you have a target server connection established, you are ready to get started using Code Coverage Analyzer.

For the first-time user, there are a few necessary functions to be performed before Code Coverage Analyzer can begin to work on your target code. These are presented as discrete steps listed below, and each one is described in detail in the subsections that follow.

1. You must create a Workbench **project** where you will build and execute your instrumented target code (see [2.3 Creating a Project](#), p.10).



---

**NOTE:** This is a change from previous versions of Code Coverage Analyzer where a CoverageScope ".prj" project was used for instrumentation. See the *Wind River General Purpose Platform, VxWorks Edition Release Notes*, 3.5 for details.

---

2. You must **instrument** and **compile** your target code using one of three optional methods. In all three methods you will be instrumenting and compiling outside of the Code Coverage Analyzer program, but you do not need to exit Code Coverage Analyzer to do that (see [3.3 Instrumenting and Compiling](#), p.35).
3. When you have successfully carried out the first two steps above, you are then ready to **download** your target code binary files and **start** them, as well as your Code Coverage Analyzer program, running. Analysis output data will begin being displayed in the Workbench GUI (see [3.4 Downloading Your Object Code](#), p.39).

## 2.3 Creating a Project

If you do not already have a Code Coverage Analyzer project in the Workbench **Project Explorer** view, follow these steps to create a new Workbench project node:

1. Right-click anywhere in the **Project Explorer** view and select **New**, then **Example** to open the **New Example** dialog box.
2. Select **VxWorks Downloadable Kernel Module Sample Project** here, then click **Next**.
3. Select **The Code Coverage Analyzer Demonstration Program** in the **New Project Sample** dialog box that opens, then click **Finish** to complete the project creation.

Notice that a new **covdemo** node now appears in the Project Explorer view. Next you need to build the covdemo.c demonstration program.

4. To establish the path to the source code (covdemo.c), right-click your project name in the **Project Explorer** view, then select **Import** to open the **Import** wizard.
5. Select **General > File System**, then click **Next**.
6. In the **From Directory** field, click **Browse** and navigate to the directory where your source code (covdemo.c) is stored, then click **OK**.

Typically this will be:

```
WIND_SCOPETOOLS_BASE/target/src/vxworks/covdemo
```

where **WIND\_SCOPETOOLS\_BASE** (an environment variable of the same name) is the root of the tree where you installed the Run-Time Analysis Tools.

7. Select the check box for your code file (covdemo) that appears in the left panel.  
Note that all the associated elements (in the panel to the right) also become selected, including covdemo.c.
8. Click **Finish** to close the wizard.

Note that all the associated elements checked in the File System panel in this wizard now appear in your project tree in the **Project Explorer** view.

You may now wish to open Code Coverage Analyzer and review the instrumentation options you selected (see [3.2 Specifying Instrumentation Parameters](#), p.28). When this review is complete, you are ready to instrument and compile your target code. If Code Coverage Analyzer is currently running, you do not have to exit from it. Simply return to Workbench and continue with the next step below.



## 2.4 Instrumenting and Compiling Your Code

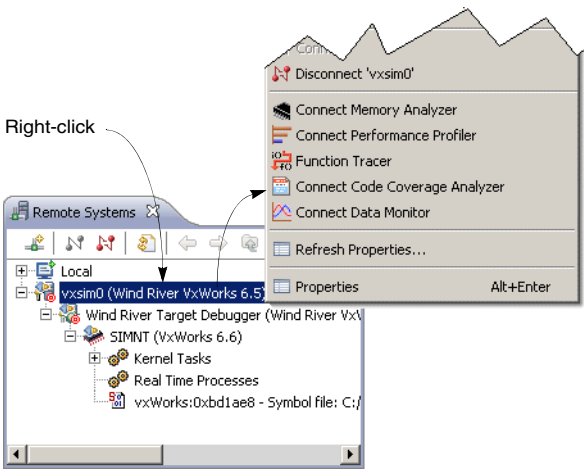
Before you run Code Coverage Analyzer on your target, your target source code must first be instrumented with tags you select to enable the desired coverage options. A complete review of the available coverage options and the kind of analysis they generate is given in [A. Code Coverage Types](#). Review this information for help in deciding on a strategy for developing test cases for your target code.

After reviewing the available coverage options, follow the steps outlined in [3.2 Specifying Instrumentation Parameters](#), p.28 to select the desired options to be compiled into your target code.

The actual process of instrumenting and compiling your target code is covered in detail in [3.3 Instrumenting and Compiling](#), p.35. Instrumentation takes place in other Workbench views outside of Code Coverage Analyzer, but it can be done without having to quit Code Coverage Analyzer if it is already running.

## 2.5 Starting Code Coverage Analyzer

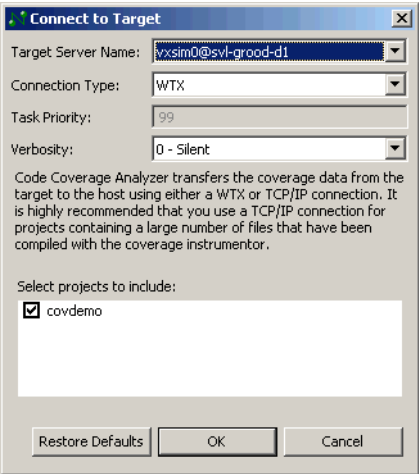
To start Code Coverage Analyzer, right-click on your target name in the **Remote Systems** view, and select **Connect Code Coverage Analyzer** in the pop-up menu, as shown here.



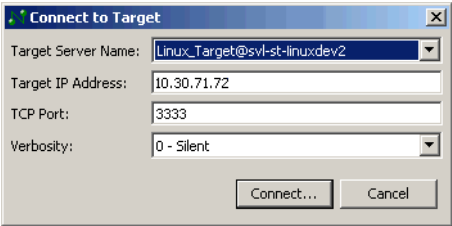
Connect to Target Dialog Box

The **Connect to Target** dialog box, shown for both VxWorks and Linux targets, opens. Here you enter parameters for your specific target, then click **Connect** to start Code Coverage Analyzer running.

VxWorks Target



Linux Target



This dialog box connects Code Coverage Analyzer to the selected VxWorks or Linux target server. The available parameters and their default values are as follows:

### Target Server

Select a target server name from the drop-down list. This list is created from the list of active target connections owned by the current user. If you do not see the target you expect to be in this list, look in the **Target Manager** view and make sure the desired target server connection is active.

### Verbosity

Specifies the volume of target status, information, and error messages written to the **Analyzer Console** view (see [Console View](#), p.20). Verbosity can be set in the range of 0-3 and has the following options:

- 0 (silent) - Displays only the most severe error message (most restrictive)
- 1 - Displays all warning, and error messages.
- 2 - Displays warning, error, and workflow messages.
- 3 (verbose) - Displays all messages (most verbose)

This verbosity setting controls the volume of messages generated by the target as well as by the host. All messages appear in the Analyzer Console view.



---

**CAUTION:** Setting verbosity to a value greater than 0 may cause the target CoverageAgent to needlessly generate large numbers of messages.

**Always use the default value of 0 for verbosity unless requested by Wind River Technical Support to help you diagnose a problem.**

---

Additional parameters for **VxWorks** targets:

### Connection Type

Use the default (TCP/IP) whenever possible because it is faster, but if you have special or unusual connection constraints, you can choose the **WTX** connection option.



---

**NOTE:** If your target is not enabled for TCP/IP communication (that is, if your VxWorks target kernel does not have the TCP/IP components), an informative message is displayed before the **Coverage Summary** view opens. In this case the **TCP/IP** option is still available, but Code Coverage Analyzer runs in WTX mode only.

---

### Task Priority

Specify the priority at which the target data collection task is to run, where **0** is highest priority, and **255** the lowest. Use the default (**99**), or set a different priority for the task on the target that handles communication between the target and the host computer. A higher priority can increase the flow of data being sent to the host, but at the cost of impacting the performance other target processes.

**Tip:** To select a reasonable priority for a TCP/IP connection, you should assign this task a priority higher than the software test you are running on the target, but lower than the system modules. The default priority is **99**.

Additional parameters for **Linux** targets:

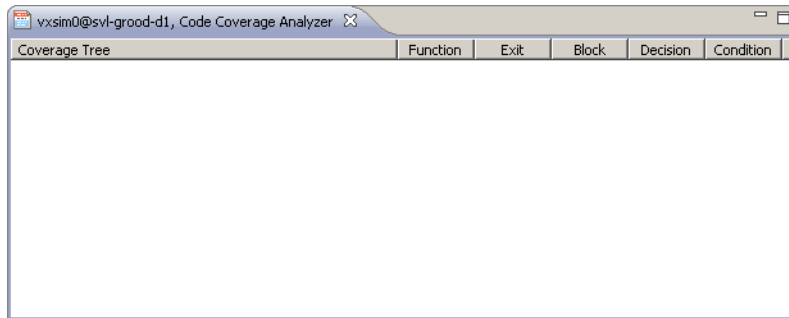
### Target IP Address

Enter the IP address of your target.

### TCP Port

Port number used by this target. Usually displays **3333**, and does not need to be changed unless you are using another specific port number.

When you click **Connect**, Code Coverage Analyzer attempts to establish the connection, and if successful, the **Coverage Summary** view opens.



This view is described in detail in [4.3 Viewing Live Coverage Data](#), p.45.

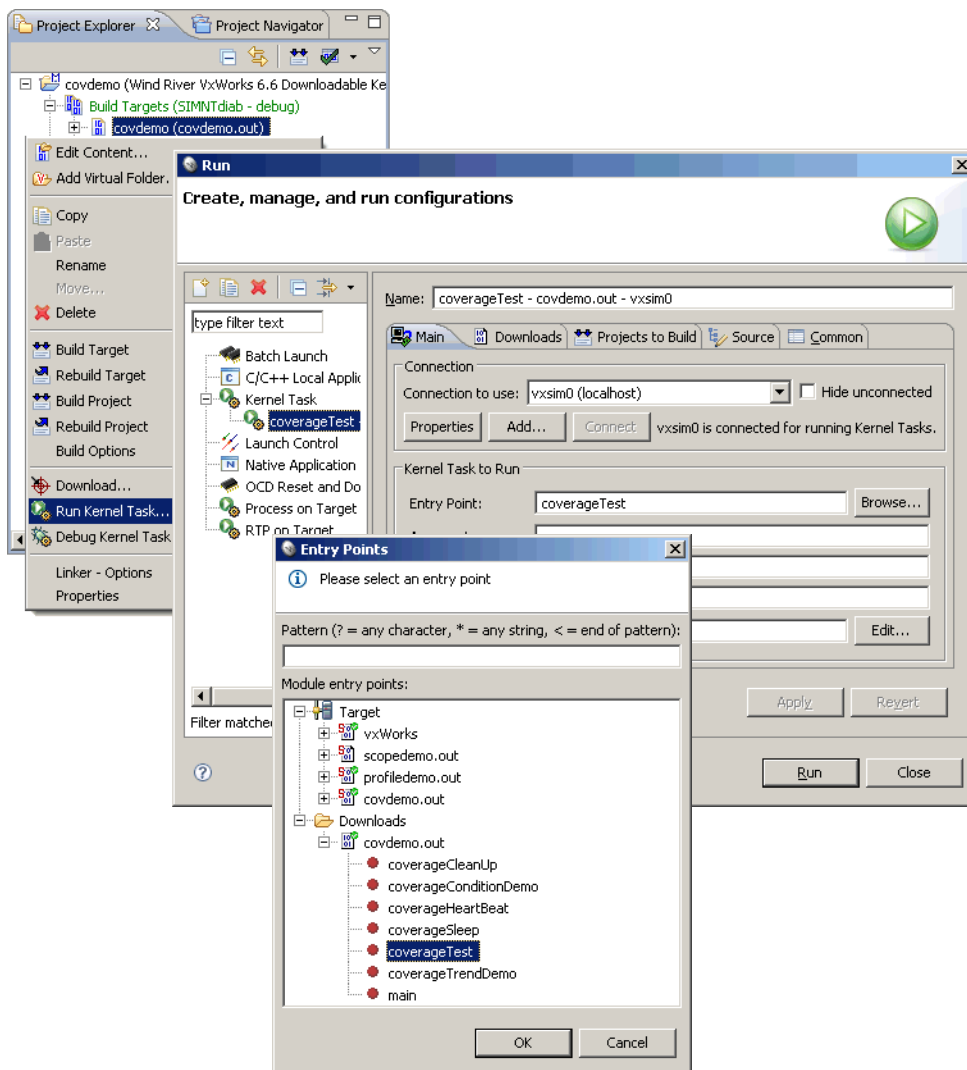


**NOTE:** When a coverage project is available, it will appear in this window with text in *italics* initially. The text will then turn **bold** when the binary files are successfully downloaded. If any code files displayed in the Coverage Summary view that you know have been downloaded do not switch from *italics* to **bold** after just a few seconds, see the troubleshooting guide, [File Errors](#), p.80 for remedies.

## Start the Target Test Code

If you have not already done so, create a coverage project now (see [2.3 Creating a Project](#), p.10), then instrument and compile your target code and download the binary files to your target, as outlined in [3. Instrumenting Source Code](#). Initially, after starting Code Coverage Analyzer, the **Coverage Summary** view is empty, as shown in the view above. When you have created a project, instrumented and compiled your test code, and finally downloaded the binary files, then you are ready to start the **covDemo** test code.

To do this, right-click **covdemo (covdemo.out)** in the **Project Explorer** view, then in the pop-up menu select **Run Kernel Task** to open the **Run** dialog box, as shown here.



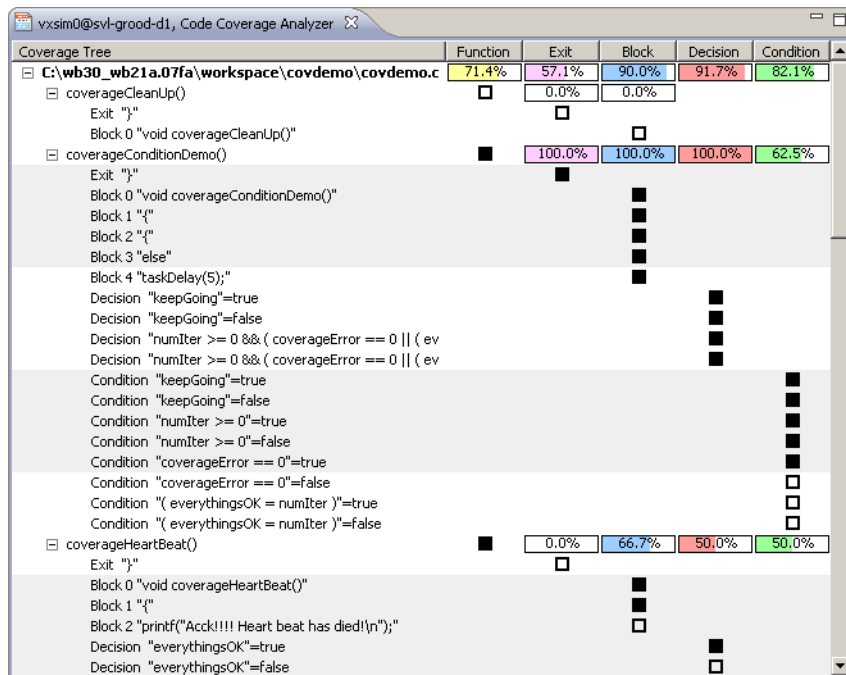
If **coverageTest** is not already displayed in the **Entry Point** field of the **Kernel Task to Run** group, click **Browse** to locate this entry point, then click **Apply** to enter it, as shown above. Click **Run** in the dialog box to start executing the **coverageTest** entry point. Output data will begin appearing in the **Coverage Summary** window, as shown in [Coverage Summary View](#), p. 17.

## 2.6 Viewing Coverage Data

This section briefly describes each of the Code Coverage Analyzer views and dialog boxes used to control the processing steps in the test code coverage analysis, and to display the results. It directs you to where you can find detailed view descriptions.

### Coverage Summary View

When you start Code Coverage Analyzer running, the **Coverage Summary** view begins to display the analyzed coverage data in a tree structure of functions and their coverage types, together with the analysis results.



It displays live coverage data when you start data collection, or saved data when you open a stored data file. Note that the top line in the data has now turned bold, indicating that the execution file was successfully downloaded. You can change the background color for any of the coverage types using the **Coverage Types**

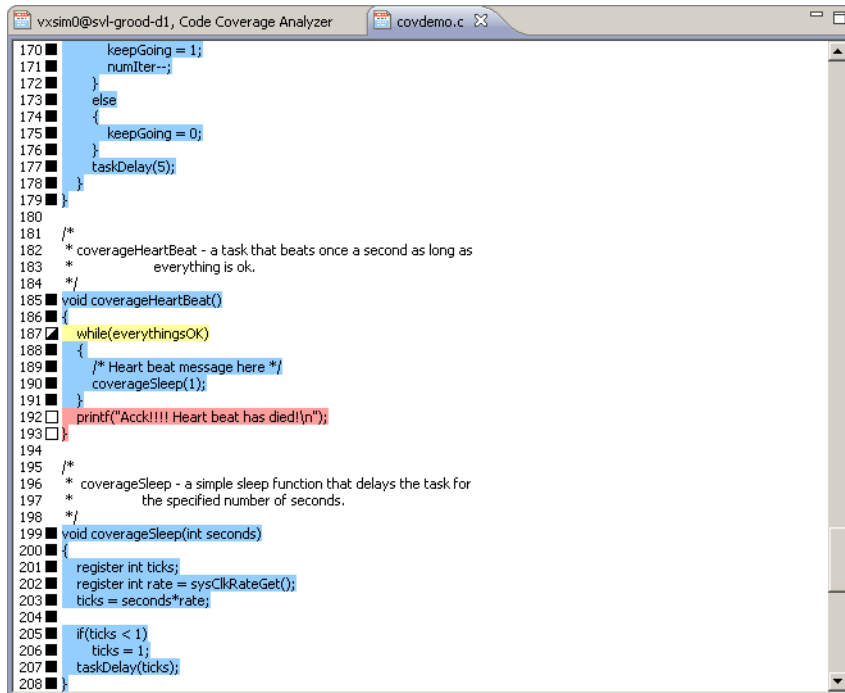
color palette in the

**Window > Preferences > Wind River > Code Coverage Analyzer** page (see [Color Configuration](#), p.21).

The **Coverage Summary** view is described in detail in [4.3 Viewing Live Coverage Data](#), p.45.

## Source Code Viewer

Double-click any data line of the coverage tree in the **Coverage Summary** view to open the **Source** view.



```
170 keepGoing = 1;
171 numIter--;
172 }
173 else
174 {
175     keepGoing = 0;
176 }
177 taskDelay(5);
178 }
179 }
180
181 /*
182  * coverageHeartBeat - a task that beats once a second as long as
183  * everything is ok.
184  */
185 void coverageHeartBeat()
186 {
187     while(everythingsOk)
188     {
189         /* Heart beat message here */
190         coverageSleep(1);
191     }
192     printf("Acck!!!! Heart beat has died!\\n");
193 }
194
195 /*
196  * coverageSleep - a simple sleep function that delays the task for
197  * the specified number of seconds.
198  */
199 void coverageSleep(int seconds)
200 {
201     register int ticks;
202     register int rate = sysClkRateGet();
203     ticks = seconds*rate;
204
205     if(ticks < 1)
206         ticks = 1;
207     taskDelay(ticks);
208 }
```

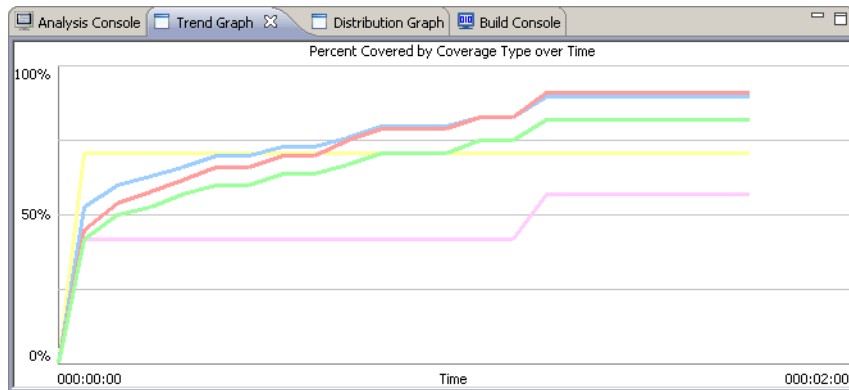
This tab view displays the entire contents of the source code file containing the selected code line, initially centered about that line. Each line in the file reflects the relative degree of coverage in two ways: a relative coverage symbol (left end, just to the right of the line number), and a colored background.



You can change the background color for any of the relative coverage types using the **Source Highlighting** color palette in the **Window > Preferences > Wind River > Code Coverage Analyzer** page (see [Color Configuration](#), p.21).

## Trend Graph View

Select the **Trend Graph** tab (below the **Coverage Summary** view) to open the **Trend Graph** view.

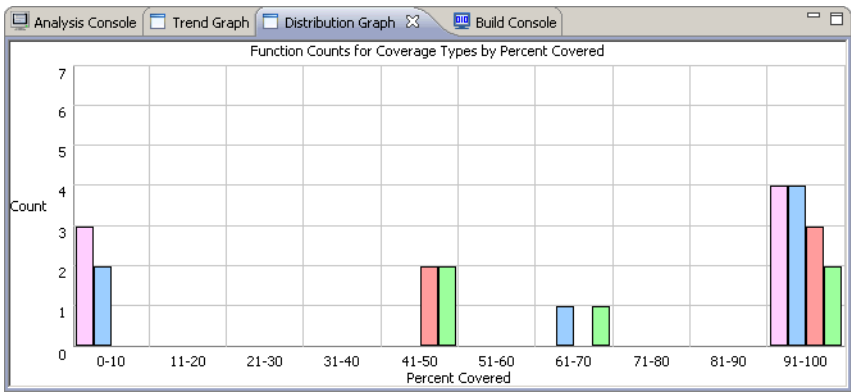


This view displays coverage data as a graph of percent coverage over time. Each selected coverage type is represented by a color-coded line. The colors key to the same colors used in the **Coverage Summary** view. You can change these colors at any time using the **Coverage Types** color palette in the **Window > Preferences > Wind River > Code Coverage Analyzer** page (see [Color Configuration](#), p.21).

The **Trend Graph** view is described in detail in [4.3.3 Trend Graph View](#), p.54.

## Distribution Graph View

Select the **Distribution Graph** tab to open the **Distribution Graph** view.



This view displays a bar graph representation of the different coverage types, using vertical bars to measure the number of functions or files, grouped in clusters or bins of percent coverage. Each of the 10 horizontal bins represents a span of 10% coverage. Right-click in the graph area and select **Display Count by File** to cause the bars representing coverage types to represent files instead. If you select it again it will read **Display Count by Function** and will toggle back to functions.

The bars are color coded by coverage type. You can change colors at any time using the color palette in the **Window > Preferences > Wind River > Code Coverage Analyzer** page (see [Color Configuration](#), p.21).

The **Distribution Graph** view is described in [4.3.4 Distribution Graph View](#), p.55.

Console View

Select the **Analysis Console** tab to open the Analysis Console view.

Time	Module	Type	Message
11:02:25 AM	Code Coverage Analy...	Info	connected to vxsim0@svl-grood-d1:2891
11:41:21 AM	Code Coverage Analy...	Info	Lost connection to target vxsim0@svl-grood-d1. CoverageScope wi
11:41:21 AM	Code Coverage Analy...	Info	Lost connection to target vxsim0@svl-grood-d1. CoverageScope wi
11:41:26 AM	Code Coverage Analy...	Info	disconnected from vxsim0@svl-grood-d1:2891
12:06:21 PM	Code Coverage Analy...	Info	connected to vxsim0@svl-grood-d1:3649

This view displays system status, as well as information and error messages generated by Code Coverage Analyzer during the session. The volume and type of messages is controlled by the **Verbosity** value entered in the Connect to Target dialog box (see [Connect to Target Dialog Box](#), p.12).

## Color Configuration

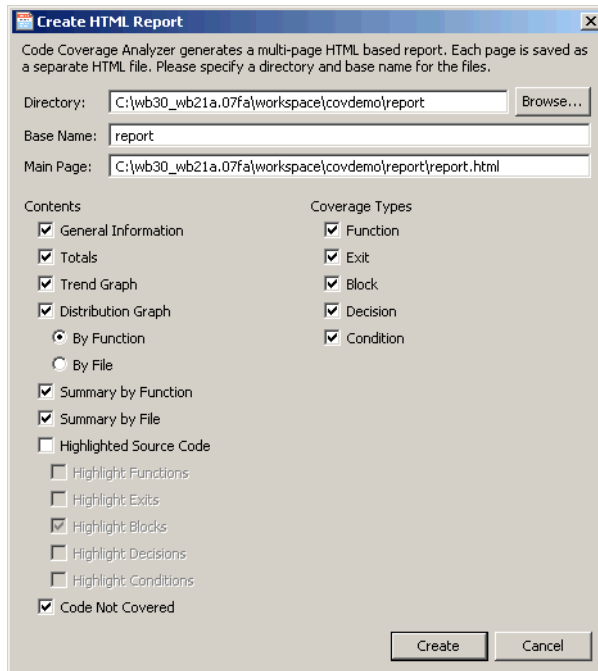
Select **Window > Preferences > Wind River > Code Coverage Analyzer** to open the **Code Coverage Analyzer** page, containing the color configuration palettes.

In this page you can select background colors for **Coverage Types**, allowing you to modify the colors used in **Coverage Summary** output data displays, as well as in the **Trend Graph** and **Distribution Graph**. It also has a **Source Highlighting** palette, allowing you to set the highlighting colors, used only in the **Source** view, separately from the other views (see [Source Code Viewer](#), p.18).

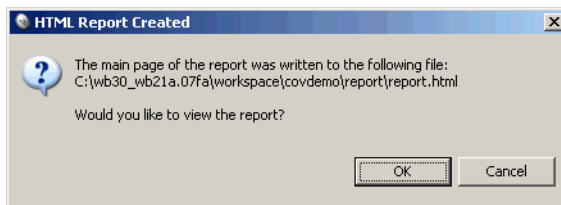
Any new colors selected will remain until changed again, even across Code Coverage Analyzer sessions. Code Coverage Analyzer is shipped with default color settings, which can be reinstated at any time using this page.

## Create HTML Report Dialog Box

Select the **Create Report** menu item from the pop-up menu in the Coverage Summary view (see [4.3 Viewing Live Coverage Data](#), p.45) to open the **Create HTML Report** dialog box.



The **Create HTML Report** dialog box allows you to pick and choose the data you want included in your coverage report, as well as the file storage location. When you have selected the desired items, click **Create** to generate the report. The **HTML Report Created** confirmation dialog box opens, where you can choose to view the report in your default browser, or click **Cancel** to not view it now.



The HTML report contains coverage data in a printable format. It is generated in HTML from coverage data as a separate step after data collection has finished, or

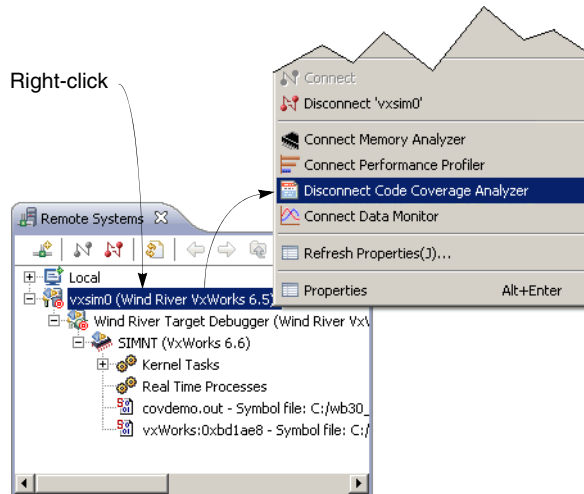
at any later time from stored coverage data. It can be displayed and printed in a browser.

A table of contents enables you to go directly to the data you're looking for. If you enabled all options while generating the report, then general information is presented first, followed by total coverage figures. Coverage is then broken down into summaries by files, then by functions. Finally there is a review of code that was not covered, first by files, then by coverage types. The report is fully linked within to allow you to easily navigate its contents.

The **Create HTML Report** dialog box is described in detail in [4.3.5 Coverage Report](#), p.56.

## Stopping Data Collection

To stop data collection, right-click on the target name and select **Disconnect Code Coverage Analyzer** in the pop-up menu.



Before exiting, Code Coverage Analyzer opens the **Save** dialog box, offering you the opportunity to save all the data collected since starting, in a file you specify.

You can also just close the **Coverage Summary** view by clicking the "X" symbol in the view's tab, but in this case the **File > Save As** option is not available to save the collected data.

## 2.7 Example Code Coverage Analyzer Session

The features of Code Coverage Analyzer can be illustrated using a simple example involving a segment of code to be tested. This example code segment, containing a C code file **covdemo.c**, is shipped with Code Coverage Analyzer. In this session you will compile the code segment with instrumentation, display the real-time results of the software test case running on the instrumented code, and analyze the output data.

Follow these steps to run the example Code Coverage Analyzer session:

### Step 1: Create a Code Coverage Analyzer project

Before you can even start Code Coverage Analyzer you must have a set of instrumented and compiled source files available. A project in which to generate and maintain these files must be created in Workbench as the first step. To create this project, follow the instructions in [2.3 Creating a Project](#), p.10.

### Step 2: Instrument and compile your target source code.

In Workbench, compile the code segment using the coverage instrumentor. To do this, follow instructions in [From Workbench](#), p.36. When you have finished instrumenting and compiling your code, come back to here and continue following these instructions.

### Step 3: Download your compiled object file(s) to the target.

This process is described in detail in [3.4 Downloading Your Object Code](#), p.39.

### Step 4: Start Code Coverage Analyzer.

Start the Code Coverage Analyzer GUI, following the instructions at [2.5 Starting Code Coverage Analyzer](#), p.11.

Observe that the file name in the **Coverage Tree** in the Coverage Summary view initially appears in italics font, but quickly appears in normal bold print, indicating that it was successfully downloaded to your target. But note that there is no progress on the coverage test yet (because we have not yet started the target code running).



---

**NOTE:** If any of the instrumented code files displayed in the Coverage Summary view do not switch from *italics* to **bold** after just a few seconds, see the troubleshooting guide, [File Errors](#), p.80 for remedies.

---

Data collection and display will begin when you actually start the instrumented test code running, as described in [Start the Target Test Code](#), p.15. The output data should quickly begin to look like the data presented in [Coverage Summary View](#), p.17.

**Step 5: View results in the different venues**

Several data presentation features and venues are available to view output from your Code Coverage Analyzer run, including the following:

- a. Right-click anywhere in the **Coverage Tree** and select **Expand** in the pop-up menu to show all the elements of the coverage tree (see [4.3.1 Coverage Summary View](#), p.45).
- b. Double-click any line in the expanded coverage tree to view the source code file containing that code line in the **Source** view (see [Source Code Viewer](#), p.18).
- c. Select the **Trend Graph** tab to view a graph of the output data over time (see [Trend Graph View](#), p.19), or the **Distribution Graph** tab to view percent coverage in bar graph form (see [Distribution Graph View](#), p.19).
- d. The **Coverage Summary** view itself shows the results of the test run on the sample software segment (see [Coverage Summary View](#), p.17). Some observations you can make from this data are:
  - Function coverage is at 71% for the file **covdemo.c** (five of the six functions were entered during the test).
  - 90% of the block elements in function **coverageHeartBeat( )**, or 9 out of the 10, tested positive for being covered. The block elements that failed are indicated by a ☐ symbol in the block column, as well as being identified in the report.
  - Scroll down the screen to see similar results that can be validated in the report.

**Step 6: Stop data collection**

Right-click your target name in the **Remote Systems** view and select **Disconnect Code Coverage Analyzer** in the pop-up menu (see [Stopping Data Collection](#), p.23).

Note that the **Coverage Summary** view remains open for further analysis. At this point you can right-click anywhere in the **Coverage Summary** view and select **Create Report**, then use the **Create HTML Report** command to generate an HTML-formatted text report from the output data.

If you connect to your target again at this point, the current coverage data being displayed in all open Code Coverage Analyzer views will be cleared before new data is displayed.

#### **Step 7: Exit Code Coverage Analyzer**

If you want to exit Code Coverage Analyzer by closing all the related views, right-click in the Code Coverage Analyzer tab itself (in the **Editor** window) and select **Close All**.

Before you exit, you might want to use the **File > Save As** Workbench menu item to save the data collected in this example (see [Saving Output Data](#), p.48). This data file can then be downloaded and examined in another Code Coverage Analyzer session.

This example shows the kind of the information you can obtain from Code Coverage Analyzer. From the output data you can determine the software segments that were not covered. You can also view the actual source code for these non-covered software segments in order to help determine how to modify testing parameters so those code segments are covered next time.



# 3

## *Instrumenting Source Code*

- 3.1 Introduction 27
- 3.2 Specifying Instrumentation Parameters 28
- 3.3 Instrumenting and Compiling 35
- 3.4 Downloading Your Object Code 39

### 3.1 Introduction

This chapter outlines the steps for instrumenting and compiling source code for a Wind River Code Coverage Analyzer project.

Before you can begin collecting data using Code Coverage Analyzer, your project source code must have instrumentation tags applied, and be recompiled using the **coverage** instrumentor. To prepare for this step you must select the instrumentor parameters that describe the coverage characteristics you want applied to your source code. After your code has been instrumented and recompiled, Code Coverage Analyzer is able to map the data from your coverage tests to the lines of original source code, allowing your coverage test results to be correlated and interpreted in many useful ways.



---

**NOTE:** The Workbench GUI manages the instrumentation and output files associated with your project, but it does not keep track of the file names and locations of your project source code files. If you move your source code *after* building it, you *must* use the **Source Path** dialog box to inform Code Coverage Analyzer where you moved it (see [Search Failure](#), p.52).

---

## 3.2 Specifying Instrumentation Parameters

As previously mentioned, prior to running the Code Coverage Analyzer you must instrument your target code with instrumentation parameter types you select to provide the desired analysis. You can always use the **default** Code Coverage Analyzer instrumentation parameters for basic coverage testing, or you can select specific parameters you want in order to customize your coverage results. If you choose to create your own instrumentation configuration, a **Code Coverage Analyzer** dialog box, available through Workbench, contains all the tools you will need.

To customize an instrumentation setup, right-click your project name in the Workbench **Project Explorer** view and select **Properties**, then select **Code Coverage Analyzer** in the tree nodes to open the **Code Coverage Analyzer** dialog box.



---

**NOTE:** When your cursor rolls over a **Coverage Types** tab view selection in the menu, a description of that coverage type appears in the **Description** field.

---

In this dialog box you can specify the instrumentation parameters you want to use (see [2.4 Instrumenting and Compiling Your Code](#), p.11).

The **Instrumentor Options** dialog box contains three tab views (four in Linux):

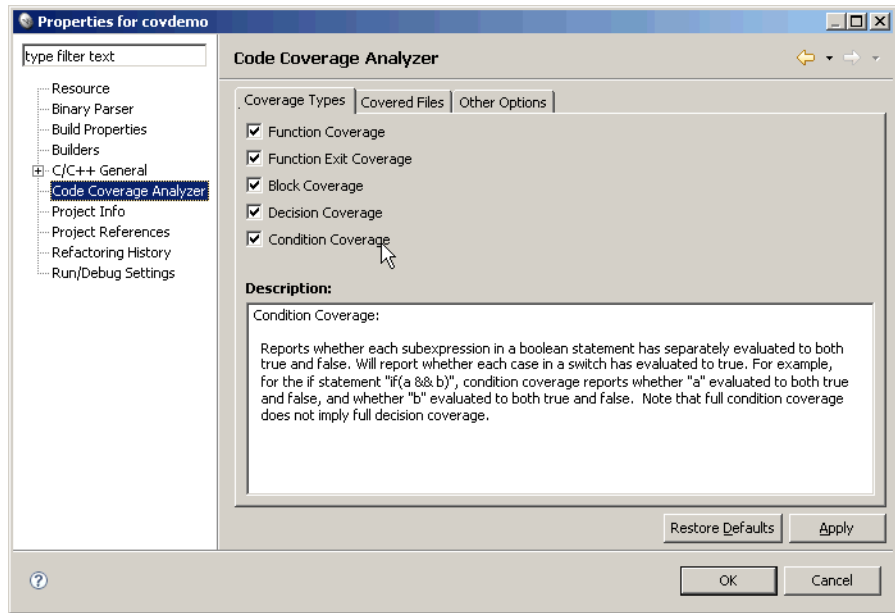
- **Coverage Types**
- **Data Storage** (Linux only)
- **Covered Files**
- **Other Options**

Each tab view contains a related set of optional configuration values you can use for instrumenting your code.

## Coverage Types Tab

When you open the **Code Coverage Analyzer** dialog box, the **Coverage Types** tab view is displayed by default.

3



In this tab view, select the coverage type(s) you want to use in your code test analysis. The default is all five coverage types selected. A detailed review of each of the five coverage types is presented in [A.3 Types of Coverage](#), p.83.

## Strategy

A common technique to maximize productivity is to initially choose parameters that will conduct a broad sweep through all your code before concentrating efforts in specific areas. For instance, function coverage alone could be used to verify that every function in every file is visited at least once. This technique causes a smaller increase in the size of the resulting object code than the selection of more than one coverage type, which is especially helpful if target memory is a concern. You can do more detailed coverage analysis of your code after you have identified areas of your code on which you want to focus your testing.

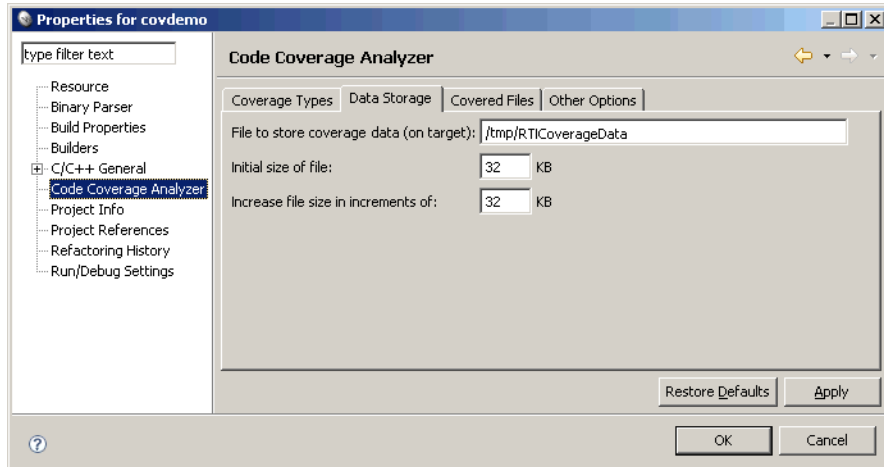
Even in more advanced coverage you can take advantage of the interrelationship of coverage types to catch some of the more subtle coverage errors. Using **Decision** and **Function** coverage together, for instance, returns more information than if both are run separately.

Remember, you will not see changes in your coverage output data until you recompile your source code.

You can click **Apply** at any time to save the current parameter selections, or you can wait until you are finished making all your changes. The current parameter values are always displayed in their respective tab views. You can modify them again at any time.

### Data Storage Tab (Linux Only)

For a Linux target only, the **Data Storage** tab view lets you specify the file name and parameters for the file on the target where coverage data is to be stored.



Use the following parameters to specify this file:

#### File to store coverage data (on target)

Enter the directory path and file name of the storage file to be created.

#### Initial size of file

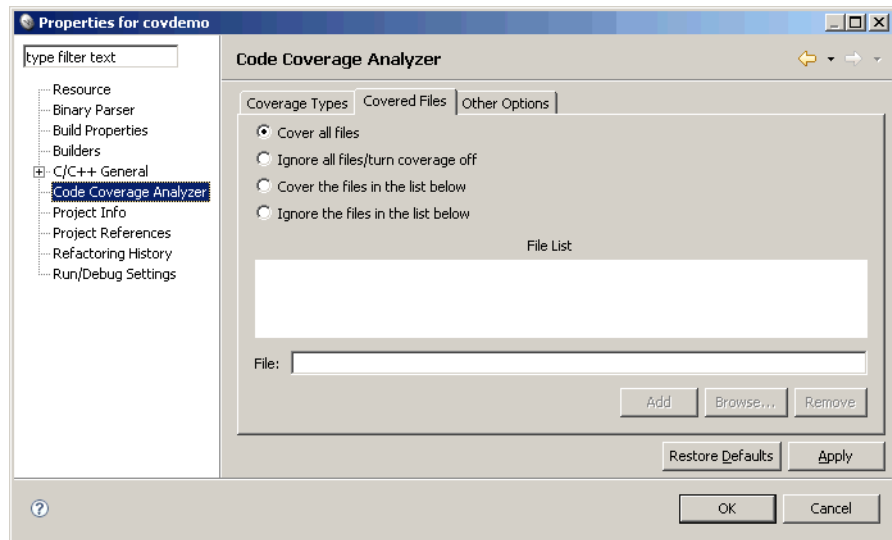
The starting file size (in KB) to be allocated.

### Increase file size in increments of

The size (in KB) to increment file when the current size will be exceeded with the next write.

## Covered Files Tab

The **Covered Files** tab view lets you specify code files to include or exclude in the process of instrumentation and compiling.



There are four file coverage choices, with one always selected:

### Cover all files

This option (the default) selects all the code files for instrumentation and compilation.

### Ignore all files/turn coverage off

This option allows you to bypass coverage testing without having to modify your makefiles again.

### Cover the files in the list below

This option allows you to specify only the source code files you want compiled with instrumentation tags. No other source code files will be instrumented.

### Ignore the files in the list below

This option allows you to specify the source code files you want to compile without adding instrumentation tags. All other source code files will be instrumented.

You can modify the **File List** table created from either of the last two options by performing any of the following:

- To add files directly, enter a directory (or a full path and file name) in the **File** field and click **Add** to add the selected pathname to the **File List** table.
- To search for, and navigate to, a file, click **Browse** to open either:
  - the **Select Files to Cover** dialog box (where you can navigate to a specific file you want covered), or
  - the **Select Files to Ignore** dialog box (where you can navigate to a specific file you want ignored),

then click **Open** to add the selected file to the **File List** table, as well as the **File** field.

In the **File List** table, you can specify full filenames, pathnames, or directories, as indicated in the following examples.

**xyz.c**

All files named **xyz.c**, regardless of where they are, will be covered (or ignored).

**c:\home\mdm\src\test\xyz.c**

The specific file "xyz.c" will be covered (or ignored).

**c:\home\mdm\src\other\**

All files in this directory, and all its subdirectories, will be covered (or ignored).

- To delete files, select any entry in the **File List** table and click **Remove** to remove it from the table.

Only files remaining in the **File List** table are included (or excluded) when the source code is instrumented and compiled.

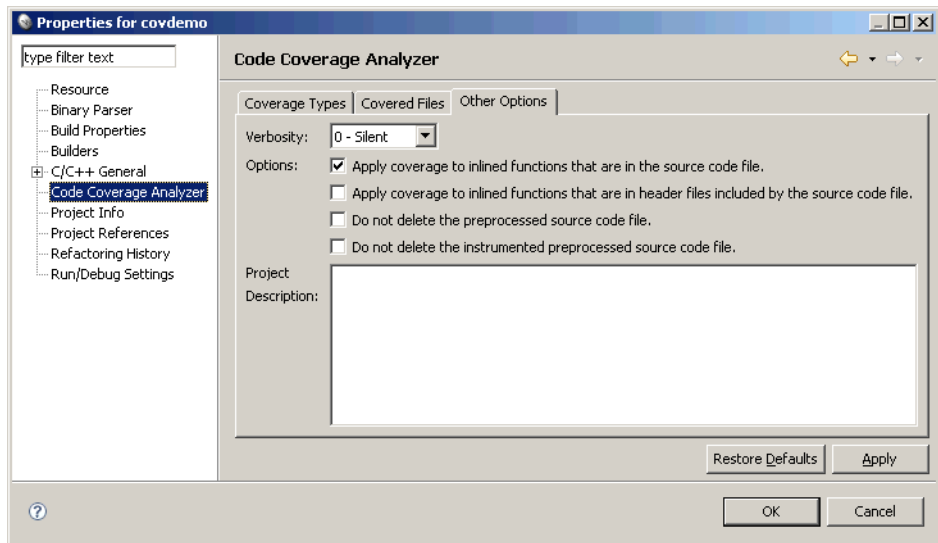
Click **Apply** to save your changes in the project configuration file. Click **Apply** whenever you change the value of any options to save the new values.

➔ **NOTE:** The **File List** table and associated **Add**, **Browse**, and **Remove** buttons only apply to the last two options. They are greyed out and not available if either of the first two (**Cover all files** and **Ignore all files/turn coverage off**) options are selected.

➔ **NOTE:** Reminder: when you make changes to your instrumentation parameters, for the changes to appear in the coverage data you *must recompile your code*.

## Other Options Tab

The **Other Options** tab view displays optional parameters that can be selected for your source code instrumentation.



The parameters available in the **Other Options** tab view are:

### Verbosity

Controls the number and type of messages displayed in the Workbench **Build Console** view during compilation of your instrumented code (see [From Workbench](#), p.36). Verbosity values can be in the range of 0-3, where 0 is least verbose, and 3 is most verbose. The results from selecting a non-zero value for

verbosity can be found at [4.2 Starting Data Collection](#), p.44. Be sure to read the Caution message there before assigning a value greater than 0 to verbosity.

## Options

Provides the following additional compile-time parameters:

### **Apply coverage to inlined functions that are in the source code file**

Check this option (default) to include these inlined source code functions in your coverage analysis. If you uncheck it, then coverage will not be reported on these functions. If there are few enough of them, or their importance to testing is minimal, you might want to consider unchecking this option. The default is **checked**.

### **Apply coverage to inlined functions that are in header files included by the source code file**

Check this option to include the header file inlined functions in coverage. The default is **unchecked**.

Inlined functions in header files are something you cannot change, and probably do not need to test. But if you do for any reason, you can check this option to include these functions in your coverage.

### **Do not delete the instrumented preprocessed source code file**

This file (with the **.i** or **.ii** extension) is the preprocessed source code file (**.csi** extension; see next item) after instrumentation tags are added by Code Coverage Analyzer. Check this option to save this file when compiling source code if you want to verify what code objects were actually tagged, or to clarify or resolve issues. The default is **unchecked**.

### **Do not delete the preprocessed source code file**

This intermediate file (with the **.csi** extension) is generated from the original source code file by the compiler preprocessor prior to being instrumented. Check this option to save this file when compiling source code if you want to compare the preprocessed source code to the instrumented preprocessed source code. The default is **unchecked**.

These optional parameters are applied when compiling. They should be carefully considered before modifying as they may cause degraded performance and increased memory and disk storage requirements.

## Project Description

This is a text entry field in which you can make any comments and annotations relevant to this project that you want. There are no restrictions on text that may be entered. Just click anywhere in the field and begin typing. Anything you enter in this field is saved when you click **Apply**.



This text information is kept with the project configuration file, and is displayed whenever this project is opened again.

### Saving the Instrumented Code Files

All the preprocessed and the instrumented preprocessed files are saved for future reference in accordance with the parameters you selected in the **Other Options** tab view (see [Other Options Tab](#), p.33).

## 3.3 Instrumenting and Compiling

The next step after setting up the Code Coverage Analyzer instrumentor options described in [2.3 Creating a Project](#), p.10 is to instrument and compile your source code. Compilation is done using the Wind River **coverage** instrumentor. This software application calls the compiler preprocessor, inserts instrumentation tags into the source code files, and finally calls the compiler to compile the instrumented code. The instrumentation parameters used are from the most recent project opened by Code Coverage Analyzer.

The compile process takes place in the same environment you have always used, and the object code generated is stored in its usual place. It is your responsibility to continue to manage this infrastructure, whether from within the Workbench GUI or in your own environment. The Workbench GUI can be used to create and manage the instrumentation and output data files associated with your project.

Your code can be instrumented and compiled from the following sources:

- Workbench
- your own makefile
- a command-line window



---

**NOTE:** Workbench does not support using build extensions to instrument VIP-type projects. To instrument these types of projects, you must use your own makefile, or use the command-line window. For more information, see the *Wind River Workbench User's Guide*.

---

## From Workbench

Follow this procedure to instrument and compile your code in Workbench:

1. Right-click your target name in the **Project Explorer** view and select **Build Options**, then select **Build with Code Coverage Analyzer**.

Before building the binaries, be sure this item is checked by right-clicking your target name and selecting **Build Options** again to verify that it is checked.



---

**NOTE:** This menu item *must* be checked before *each* time you build in order to instrument your target code with the tags you select.

---

2. Ensure that you are compiling against the correct architecture for your target by right-clicking your target name in the **Project Explorer** view and select **Properties** to open the **Properties for project** dialog box.
3. Select the **Build Properties** view, then be sure the check box for your target architecture is selected in the **Available and enabled build specs** list in the **Build Support and Specs** tab view, then click **OK** to close the dialog box.
4. Right-click the project name in the **Project Explorer** view again, and select **Rebuild Project** from the drop-down menu to start the compilation.



---

**NOTE:** You must always use the **Rebuild Project** option (rather than just **Build**) in order for the compiler to be instructed to use the special **coverage** instrumentor option (see [3.3 Instrumenting and Compiling](#), p.35).

---

Your code will be instrumented and compiled in this process. Status, information and error messages appear in the Workbench **Build Console** view. The number and types of these messages is determined by the verbosity value, described in [Other Options Tab](#), p.33. After a few seconds, if the compilation finished with no errors, you will observe a message similar to the following:

```
Build Finished in Project 'covDemo': 2006-03-30 15:10:58 (Elapsed
Time: 00:07)
```



---

**NOTE:** If you select instrumentation parameters and compile your code using the above procedure, then later you make code changes and need to compile again, you *must* select **Rebuild Project** (as in Step 5 above). If you only select **Build Project**, the build does not use the **coverage** instrumentor.

---

## From Your Own Makefile

If you compile using your own makefile, you must modify the makefile before compiling. Specifically, you must add the word **coverage** (the name of the Code Coverage Analyzer Instrumentor tool) before each compiler statement in the makefile. No other changes are required. As each compile statement in the makefile is executed, the specified instrumentation tags are added to the preprocessed source code, and the instrumented preprocessed source code is then compiled with the same options (in situ) on the compiler command-line.

For instance, if you compile with the gcc compiler, a statement to compile the source code file `xyz.c` might ordinarily look like:

```
gcc xyz.c
```

To instrument and compile this source code file under Code Coverage Analyzer, the statement would be modified to:

```
coverage gcc xyz.c
```

You can see from this example that the entire compiler statement has become an argument to the coverage instrumentor. Do not change any of your compiler options or directives, as they are transferred to the compiler when it is executed. The coverage instrumentor parameters, listed in From a Command Line Window below, can be also be specified in your makefile.



---

**NOTE:** Be sure to verify that the coverage instrumentor is accessible in the PATH statement in your makefile.

---

## From a Command-Line Window

You can instrument and compile individual files using the coverage instrumentor from the command-line, in the same location where you ran Code Coverage Analyzer. To run this application, you must specify the full pathname to the executable file when you invoke it, or you can set the **PATH** environment variable to include that pathname.

The coverage instrumentor program does not need to be run on your target, or even on your host computer. Because of cross-compilers, you can build your target code on almost any machine. The only requirement is that you must use a version of the coverage program that matches the architecture type of your build machine. Note that we only support build machines that match the host architectures indicated in Section 1.1 of the *Wind River Run-Time Analysis Tools Installation Guide and Release Notes Manual*.

The following is the command-line syntax for starting the coverage instrumentor:

```
coverage [-project=project file] [-verbosity=n] [-help]  
          compiler [compiler options]
```

where the parameters (all are optional) have the following meaning:

**-project=project file**

The instrumentor uses the specified Code Coverage Analyzer project file, where *project file* is the full pathname to the project configuration file. In the absence of this flag, the instrumentor uses the last project opened in the GUI.

**-verbosity=n** where *n* is in the range of 0 to 3

Controls the number and type of messages posted during source code instrumentation. Using the default value of 0 causes the instrumentor to post only error messages. Specifying a larger value (in the range of 1-3) creates an increasingly greater variety and volume of instrumentor messages. The general characteristics of verbosity levels are described in greater detail in [4.2 Starting Data Collection](#), p.44.

**-help**

This option displays a list of commands for Code Coverage Analyzer.

The coverage instrumentor adds instrumentation tags to your source code, then invokes the compiler to compile the instrumented code.

## Instrumentation Issues

The following issues deal specifically with instrumenting and compiling your source code, and should be reviewed before proceeding with this activity.

- The Code Coverage Analyzer instrumentor supports only **ANSI C standard C/C++** code. In addition, it currently supports only the gcc and Wind River compilers.
- Recent versions of the Wind River compiler driver programs (dcc and dplus) attempt to detect whether the language is C or C++ and compile accordingly. This mechanism enables you to compile C code with dcc or C++ code with dplus without having to actually specify which compiler to use.
- The way in which Code Coverage Analyzer invokes the compiler renders the compiler unable to dynamically guess which language is being compiled. Therefore you *must* use dcc to compile only C code, and dplus to compile only C++ code. Further, this means that if a project contains both C and C++ code, you must compile each using different compilation command-lines

- The coverage instrumentor contains a C++ parser, which can create a difficulty when there is a mixture of C and C++ code to be instrumented. The parser declares an error when a C code definition statement tries to define a variable that is a *registered keyword* in C++. For example, the C code definition:

```
typedef enum tagbool { FALSE, TRUE } bool;
```

is a legal statement in C, but would not be legal in C++, since **bool** is a registered C++ keyword. The C++ parser in the instrumentor will declare this statement an error. There are easy work-arounds available. You can use a C preprocessor macro, such as adding the flag:

```
-Dbool=Bool
```

to your compilation line for all C source code (but not for C++). Or, if you prefer, before the **typedef** line in the **superv.h** file you can add:

```
#define bool Bool
```



---

**NOTE:** Beware that if you create a code file by the name **ctdt.c**, it will be ignored by the instrumentor.

---

## 3.4 Downloading Your Object Code

When you have finished instrumenting and compiling your source code files, you must be sure to download the object files to the target. You can do this at any time.



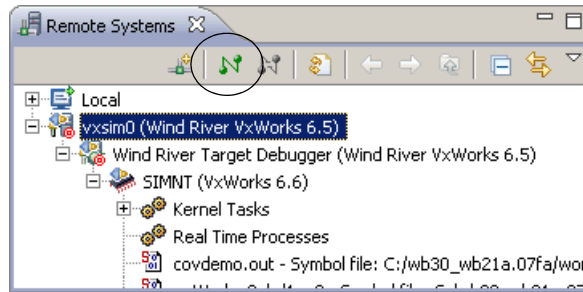
---

**NOTE:** If you recompile object files which were previously instrumented and compiled for Code Coverage Analyzer, you must first disconnect, then reconnect, any instances of Code Coverage Analyzer currently connected to that target. If you do not, you will receive incorrect coverage data for the newly created object files (that is, coverage data corresponding to the previous instrumentation parameter selection).

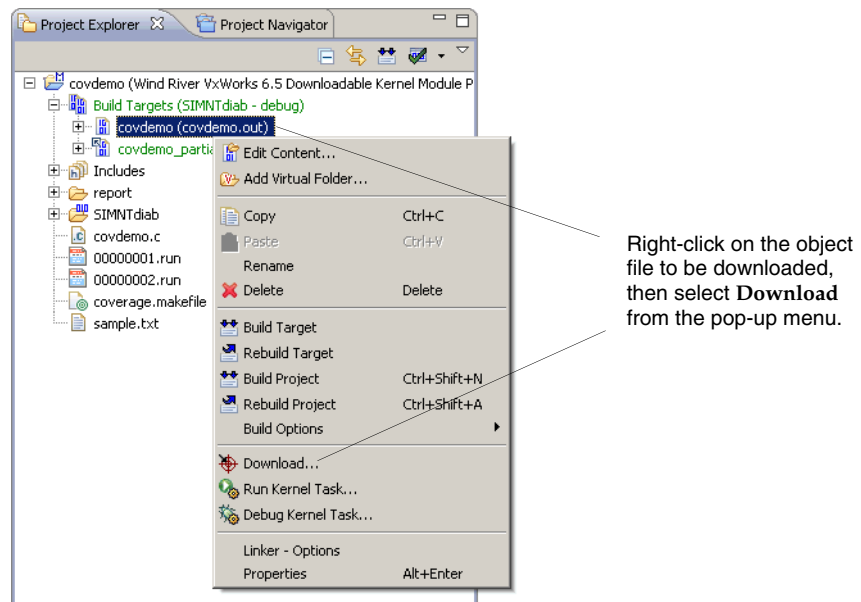
---

When you have finished instrumenting and compiling your source files in Workbench, you can download your object file(s) as follows:

- a. If you have not done so already, connect your target to the target manager by selecting the target symbol and selecting the **Connect** icon (circled below).



- b. With the target connected, in the **Project Explorer** view, right-click the object file (**covdemo.out**), then select **Download** in the pop-up menu that opens.



- c. Click **OK** in the Download dialog box that opens.

In a few seconds you should see the project data entry in the Coverage Summary view change from italics to bold if your binary files were

successfully downloaded. If this does not happen, see the troubleshooting guide, [File Errors](#), p.80 for remedies as discussed in [Connect to Target Dialog Box](#), p.12.

- d. If you have multiple object files you need to download, repeat this procedure for each object file.
- e. When all your binary files are downloaded, return to [Start the Target Test Code](#), p.15 for directions to start the code running.





# 4

## *Viewing Output*

- 4.1 Introduction 43
- 4.2 Starting Data Collection 44
- 4.3 Viewing Live Coverage Data 45
- 4.4 Viewing Saved Data 61
- 4.5 Exporting Data 62

### **4.1 Introduction**

This chapter begins by briefly discussing how to start the Code Coverage Analyzer GUI, then goes on to describe in detail the appearance, organization, and usefulness of each output view.

## 4.2 Starting Data Collection

When you have finished instrumenting and compiling your code, as outlined in [3. Instrumenting Source Code](#), you can start collecting data with Wind River Code Coverage Analyzer.

Code Coverage Analyzer reports on data gathered and stored during the running of the test code. It analyzes this data as it is being generated (in real time). If you want to observe the data from the very beginning of the test run, you will want to start Code Coverage Analyzer running prior to starting your test code. However, you can start collecting data at any time, before, during, or after you start running the test code.

When you are ready to begin data collection, right-click on your target server name in the **Remote Systems** view and select **Connect Code Coverage Analyzer** to open the **Connect to Target** dialog box (see [2.5 Starting Code Coverage Analyzer](#), p.11). After selecting desired connection parameters in the dialog box, click **OK** to open the **Coverage Summary** view, and then start your test program if you have not already done so, following directions in [Start the Target Test Code](#), p.15.

### Data Connection Problems

If the **Coverage Summary** view opens successfully, but one or more of the listed files does not change from italics to bold after just a few seconds, a potential problem with that file is indicated. Code Coverage Analyzer will begin collecting data for the remaining (good) files, but you may want to investigate the broken ones (see [File Errors](#), p.80).

The following are possible file-related problems and suggested remedies:

- If there were no instrumentor or compiler errors, check the shell for loading error messages. Try downloading the files again after correcting any errors found.
- Check to be sure that the indicated source code files were not compiled either with another project, or with coverage turned off (see [Covered Files Tab](#), p.31). In this case, rebuild your source code and try again.

### Successful Connection

When a successful connection has been made, the **Coverage Summary** view opens. If your test run is already in progress, the results gathered up to this point are displayed immediately. Any further results are displayed dynamically as they are generated.

## 4.3 Viewing Live Coverage Data

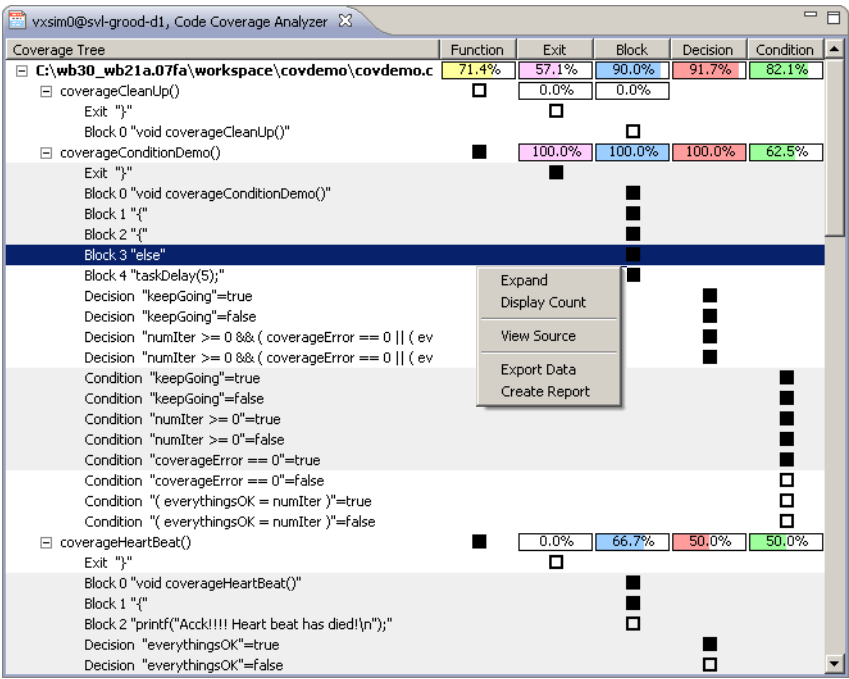
Output data can be viewed in the following forms:

- In the [Coverage Summary View](#), as an expandable/collapsible tree structure of percent coverage figures for files and functions, and as covered (or not covered) symbols for specific coverage types.
- In the [Source Code Viewer](#), as individual lines of code, with relative coverage symbols and colored highlighting based on full, partial, or no coverage.
- In the [Trend Graph View](#), as a graph of increasing percentage of coverage over test run time, for each of the different coverage types (differentiated with colored lines).
- In the [Distribution Graph View](#), as a bar graph of numbers of files or functions by percent coverage bins (10 of them), over the range of 0-100%, using colors to show the different coverage types.
- As a generated HTML [Coverage Report](#).
- Exported as a comma separated value (.csv) file for your own analysis, typically in a spreadsheet utility.

All the different forms of data display can be used to display a stored output data file that is opened, but the **Coverage Summary**, **Source** view, **Trend Graph**, and **Distribution Graph** options are also used for data currently being collected. Results can be displayed simultaneously for comparison and analysis.

### 4.3.1 Coverage Summary View

The **Coverage Summary** view opens at the beginning of data collection, or with the loading of a stored output data file.



The example **Coverage Summary** view above shows how the results of collecting data are displayed as an expandable/collapsible coverage tree structure of functions and their coverage types, together with the resulting coverage data. The tree is displayed in rows, one for each file and each function within the file, and one for the beginning statement of each coverage type within the function.

Each row of the tree is comprised of three regions, representing the following:

**Expand/Collapse**

Shows a "+" symbol (collapsed, select to expand), or a "-" symbol (expanded, select to collapse). This symbol appears only on file and function lines, and controls whether subordinate lines are made visible or hidden.

**Coverage Tree**

Displays the selected covered files and functions (and specific coverage tags, if expanded). For a file, it shows the complete pathname and filename. For a function, it shows the function name and parameters passed. For a specific coverage tag, it indicates the type, and shows the beginning code statement of

the type. For long lines, a horizontal scroll bar allows you to view the entire line.

Double-click any line in the coverage tree to open a separate view where you can view the entire source code file containing this line (see [4.3.2 Source Code Viewer](#), p.49).

### Coverage Results

Graphically shows the percent coverage in columns for each basic coverage type. Coverage results for each file and function are given in actual percent coverage (for example, a number such as "50%") displayed in text fields for each coverage type. The specific coverage tag lines show coverage in this column graphically, using the ■ symbol for **covered**, or the □ symbol for **not covered**. Overall percentage covered figures for the entire run are displayed at the top and bottom of this column.

For further graphic enhancement, the percent coverage text fields show a color in the background that increases with the percent coverage figure. To modify the colors, select

**Window > Preferences > Wind River > Code Coverage Analyzer** in the Workbench toolbar to open the **Code Coverage Analyzer** page, then select colors from the **Coverage Types** palette (see [Color Configuration](#), p.21).

By selecting **Display Count** in the pop-up menu (see below), the graphic symbols representing coverage (as described above) are replaced by the actual count of *hits* for each coverage results line in the display. This feature can be toggled on or off at any time, and its effects appear only in the **Coverage Summary** view.

### Pop-up Menu

When you right-click any row in the **Coverage Summary** view, a pop-up menu opens as shown in the figure above. The following menu items are available:

- **Expand**  
Expands the **Coverage Tree** to display all nodes.
- **Display Count**  
Toggles to show the numeric count of actual "hits" found within each group in a given coverage type, or only a symbol denoting none, partial, or full coverage.
- **View Source**

Opens the **Source** tab view display for the selected row (see [4.3.2 Source Code Viewer](#), p.49).

- **Export Data**

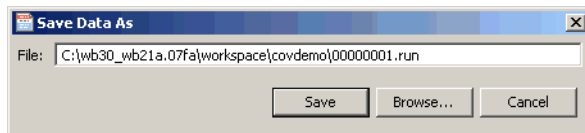
Opens the Export Data dialog where you can specify a file to save your data to be exported (see [4.5 Exporting Data](#), p.62).

- **Create Report**

Opens the **Create HTML Report** dialog box where you can custom configure a coverage report (see [4.3.5 Coverage Report](#), p.56).

## Saving Output Data

You can save the collected and analyzed coverage data to an output file at any time while the Code Coverage Analyzer GUI is open, while it is still collecting data, or even if it has been disconnected. To save your output to a file, select **File > Save As** from the Workbench toolbar to open the **Save Data As** dialog box.

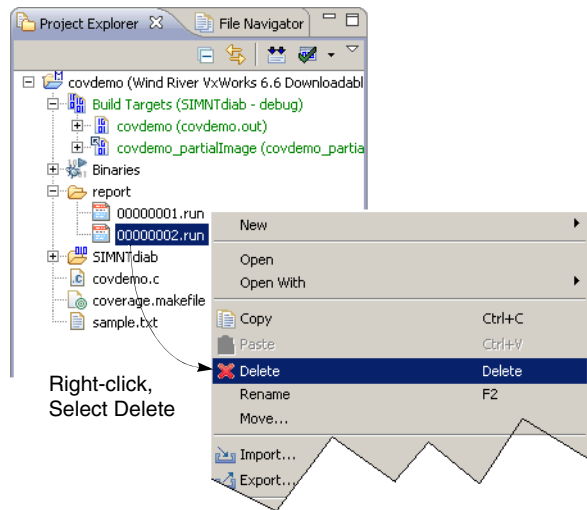


You can modify the default file name in the **File** field, or you can use the **Browse** button to navigate to another existing directory and file. When you click **Save**, a copy of all the data collected since Code Coverage Analyzer was started will be saved at the location showing in the **File** field. The saved data can be viewed as described in [4.4 Viewing Saved Data](#), p.61.

## Deleting Saved Data Files

You can delete any data files you previously saved with the **File > Save As** command from the Workbench toolbar. To do this, follow these steps:

1. Open the **report** node in your project tree in the **Project Explorer** view.
2. Right-click the data file you want to delete to open the pop-up menu.
3. In the pop-up menu, select **Delete** to delete the file.



### 4.3.2 Source Code Viewer

The **Source** view displays your original source code corresponding to the coverage analysis line selected in the **Coverage Summary** view, for visual analysis. To open this Source view, double-click any analysis line in the Coverage Summary view. The corresponding code line is initially positioned at the top of the Source view, but you can scroll up and down from there. This view is designed to allow you to scan through your code to find any sections of it that were not exercised by the test program.

Double-click on a block, decision, or condition line in the **Coverage Summary** view and the Source view will scroll to that line and highlight all the lines in that block. An example Coverage Summary view, with the Source view opened in the foreground, is shown here.

The relative coverage status (fully, partially, or not covered) for each instrumented code line is indicated in two visual ways:

- With a symbol at the left end of each instrumented code line. This symbol signifies the following relative coverage status:
  - = fully covered
  - ▣ = partially covered

□ = not covered

- The background (highlight) color of the line.

The color for each coverage type highlight can be changed by selecting **Code Coverage Analyzer** in Workbench to open the **Coverage Analyzer** page, and using the **Source\_Highlighting** color palette.

The **Source** view always opens with the coverage type selected and highlighted that corresponds to the coverage type of the line double-clicked in the **Coverage Summary** view. In addition, every code line in that category has a relative coverage symbol displayed right after the line number. In this example, all the code lines contained in the Block 3 selected in the **Coverage Summary** view are fully covered (indicated by the ■ symbol, as well as the green highlighting), as can be observed in [4.3 Viewing Live Coverage Data](#), p.45.

## Benefits of Multiple Coverage Selection

Multiple kinds of coverage can sometimes point out lack of coverage that was not apparent with just a single coverage type. For instance, in the example above it appears that all the code in the view is covered. But if you go back to the **Coverage Summary** view and select **Condition Coverage** in addition to the other coverage types already selected, then rerun Code Coverage Analyzer, the result shown below makes it obvious that the indicated portion of the code is not fully covered.

Compare this with the previous results where **Condition Coverage** is not enabled.

One or more of the subexpressions in the **if** statement at line 168 (at the arrow), in the figure above, did not get evaluated to true or false, as described in [Condition](#), p.88, whereas with only **Decision Coverage** it appeared fully covered. This situation can also arise with other types of coverage used alone.

The descriptions of **Partially Covered** (indicated by the ▣ symbol) relate to the different coverage types as follows:

- **Function** — cannot be partially covered; it either is or is not covered.
- **Block**—cannot be partially covered, however, a single line in the **Source** view may contain more than one block — typically the end of one block and the start of another. If one block in the line is covered and the other is not, the line is labeled **Partially Covered**.
- **Decision** — if there is only one decision statement on the line, it means the decision has evaluated to true or false, but not to both. If there are multiple

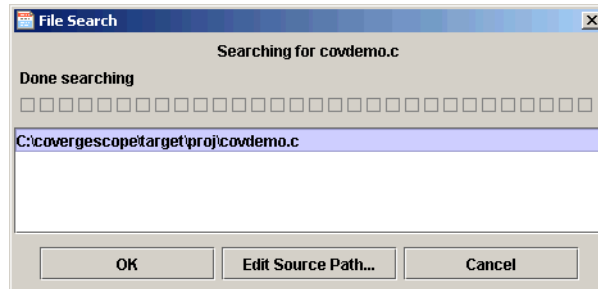


decision statements on the line, it means that at least one has evaluated to true or false, but not to both. For specific information on what the decision(s) have been evaluated to, see the coverage tree in the **Coverage Summary** view.

- **Condition** — at least one of the subexpressions in a Boolean expression has evaluated to true or false, but not to both.

## Searching for Source Code Files

The **File Search** dialog box opens when you initiate the search for a source code file by double-clicking any line in the **Coverage Summary** window.



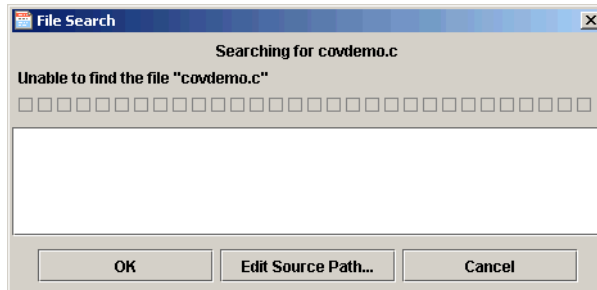
The name of the file being searched for is displayed at the top of the dialog box, and a progress bar is displayed just below it to show that the search is still working. As matching files are found they are added, along with their pathnames, to the list in the center of the dialog box. When searching is finished, **Done Searching** is displayed above the progress bar, and the progress stops. If no entries are in the list, it means none were found; edit your source paths and retry.

You can select a match in this field at any time, even while the search is continuing, and click **OK**. This closes the dialog box (and terminates the search, if still running) then opens the selected file in the **Source** view. The selected source code file will remain **found**, but only for the duration of this Code Coverage Analyzer session.

If you click **Edit Source Path** at any time, this dialog box closes (the search is terminated, if still running) and the **Source Path** dialog box opens, where you can again modify the selection of pathnames to be searched. Click **Cancel** to terminate the search and close this dialog box.

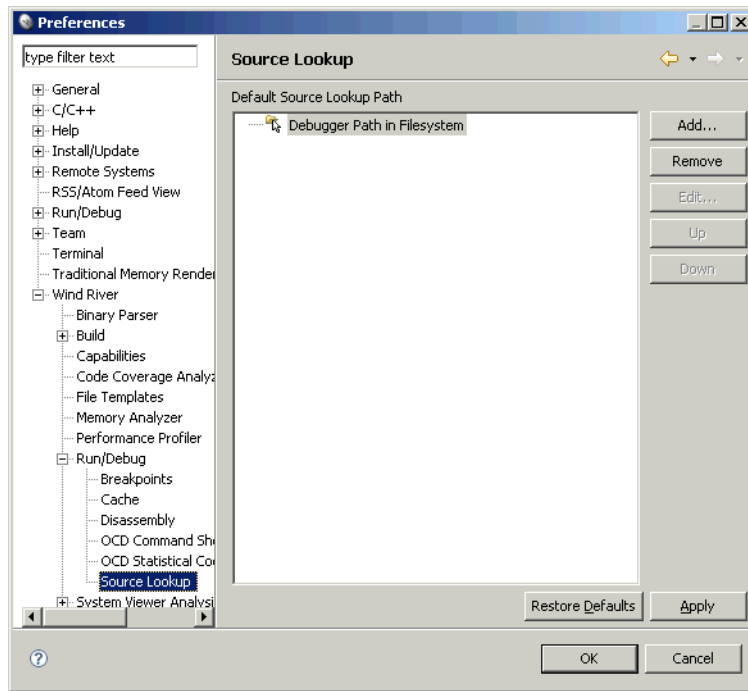
## Search Failure

If you move your source code files from the place where they were built, Code Coverage Analyzer will not be able to locate them when you try to open the **Source** view. In this case the **File Search** dialog box displays an error message, including the filename, as in this example.



## Specifying Source Paths

To modify pathnames to your source code files, select **Window > Preferences** in the Workbench toolbar to open the **Preferences** dialog box. In this dialog box, select the **Wind River > Run/Debug** node in the tree to open the **Source Lookup** page.



In this page, you can enter one or more alternate source code directories into the ordered list. Code Coverage Analyzer searches the directories in this list for instrumented and compiled source code files in the order in which the directories are listed in the dialog box (from top to the bottom).

To add a directory to the list, click **Add** to open the **Browse** dialog box, where you can navigate to and select a directory. The directory will be entered at the bottom in the **Default Source Lookup Path** list. Use the **Edit** button to open the **Folder Selection** dialog box where you can add subfolders to an existing folder.

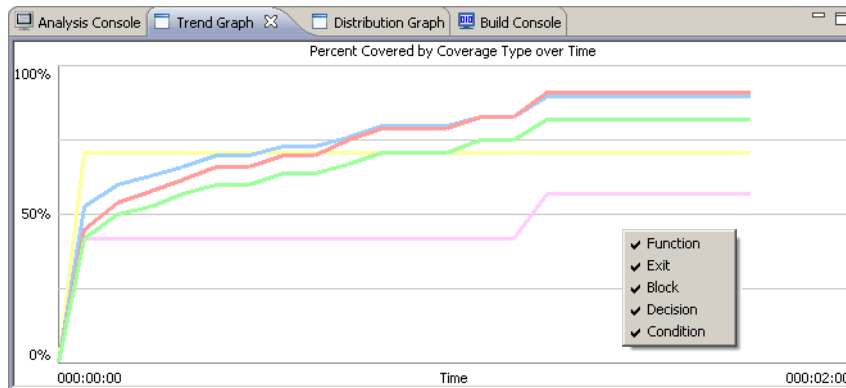
Directories can be given a different search priority by selecting the directory, then using the **Up** or **Down** button on the right to physically move the entry to a new location in the list. To remove a directory, select it in the list, then click **Remove**.

When your list is complete and prioritized as you want, click **Apply** to save any changes you made to the list of directories. Close this dialog box and start the search again as directed in [4.3.2 Source Code Viewer](#), p.49.

The list of directories you generated in the **Source Path** dialog box appear the next time you open this dialog box, and remain until you modify the list again.

### 4.3.3 Trend Graph View

Open the **Trend Graph** view by selecting the **Trend Graph** tab.



It contains a line graph of percent coverage for each of the selected coverage types as a function of time. Time is measured from when the connection is made to the target, regardless of when the test begins.

The data lines in the **Trend Graph** are color coded to differentiate between the types of coverage. To modify the colors (which will also modify the colors used in the **Coverage Summary** view), select

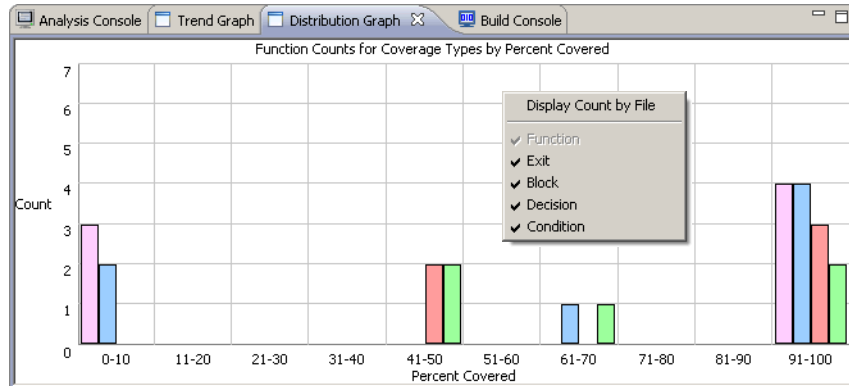
**Window > Preferences > Wind River > Code Coverage Analyzer** in Workbench to open the **Code Coverage Analyzer** page, then select colors from the **Coverage Types** palette. (see [Color Configuration](#), p.21).

#### Pop-up Menu

Right-click anywhere in the graphing area to open the pop-up menu shown in the view above. In this menu, select any entry to toggle the corresponding trace line on or off.

#### 4.3.4 Distribution Graph View

Open the **Distribution Graph** view by selecting the **Distribution Graph** tab.



The **Distribution Graph** gives you a quick visual indication of the distribution of percent coverage at the function or file level. It shows vertical bars, color coded to represent coverage types, arranged in bins of percent coverage ranges, each bin covering a 10% range. The height of each bar is the number of functions (or files) whose coverage falls within that percent range.

The data bars in the **Distribution Graph** are color coded to differentiate between the types of coverage. To modify the colors (which will modify the colors used in the **Coverage Summary** view as well), select

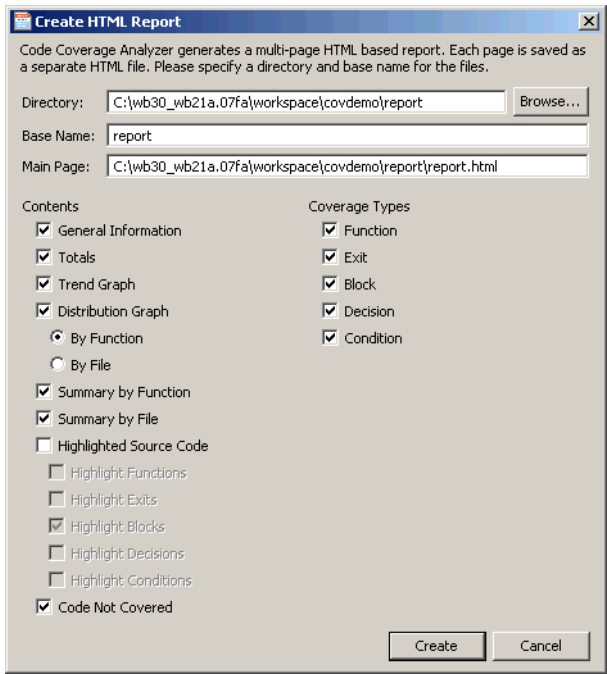
**Window > Preferences > Wind River > Code Coverage Analyzer** in Workbench to open the **Code Coverage Analyzer** page, then select colors from the **Coverage Types** palette. (see [Color Configuration](#), p.21).

#### Pop-up Menu

Right-click anywhere in the graphing area to open the pop-up menu shown in the view above. Select any entry in this menu to toggle the corresponding plot bar on or off (in this example, the Function bar display has been toggled off). In addition, You can display the view of percent coverage by **Files** or **Functions** by selecting the **Display Count by File** (or **Function**) menu item at the top of the pop-up menu. This item toggles with each selection, so the only choice is the other view. Whichever state you select, it will remain until you change it again, even across Code Coverage Analyzer sessions.

### 4.3.5 Coverage Report

Right-click anywhere in the **Coverage Summary** view, and select **Create Report** from the pop-up menu to open the **Create HTML Report** dialog box.



This is where you select parameters used to build the HTML-formatted report for the coverage results generated by this Code Coverage Analyzer test run.

The following **Contents** parameters can be selected:

#### General Information

When checked, a section containing general information is added to the report. It contains the target name, the date and time data collection began, the duration, and the coverage types selected.

#### Totals

When checked, a section containing the percent coverage for each of the coverage types selected is added to the report.

#### Trend Graph

When checked, a link to a **Trend Graph** view is provided.

### Coverage Graph

When checked, a link to a **Distribution Graph** view is provided.

#### By Function

When checked, the **Distribution Graph** shows functions.

#### By File

When checked, the **Distribution Graph** shows files.

### Summary by Function

When checked, a section listing each function, which file the function is in, and percent coverage in that function for each of the specific coverage types is added to the report.

### Summary by File

When checked, a section listing each file and the percent coverage for each of the coverage types selected in that file is added to the report.

### Highlighted Source Code

When checked, a link is provided to one or more views, each displaying a source code file with code lines highlighted, as specified in the **Highlight Functions**, **Highlight Blocks**, **Highlight Decisions**, and **Highlight Conditions** check boxes that follow.

### Code Not Covered

When checked, a detailed section listing all the objects of each selected coverage type that were found to not be covered is added to the report. The section is organized by **Function Coverage** first, followed by **Exit Coverage**, **Block Coverage**, **Decision Coverage**, and **Condition Coverage**. Each report item includes the coverage type, followed by a brief explanation of why it was not covered, the file containing it, and the line numbers (or range) included. For **Function Coverage**, the function name is given in place of the coverage type, whereas for the other coverage types, the coverage type name is first, and there is a line with the beginning code statement before the line number.

The following **Coverage Types** options can be selected:

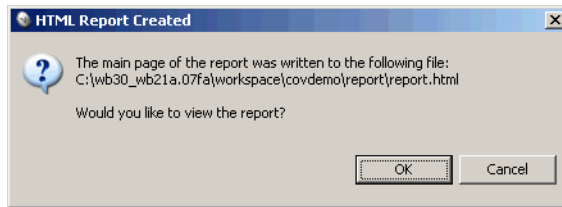
**Function, Exit, Block, Decision, Condition**

In the **Directory** and **Base Name** fields, enter the pathname and filename, respectively, where you want the report file stored. You can Click **Browse** to open the **Select Coverage Report** dialog box where you can navigate to your desired directory and click **Select** to enter the pathname in the field. Since, for any given report, Code Coverage Analyzer will likely generate the report as multiple HTML pages (files), each linked to the opening page, you may want to consider specifying

a separate directory in the **Directory** field with a descriptive name to store the collection of files for each report.

A confirmation of the full pathname for the report file you are creating is displayed in the **Main Page** field. You cannot change any part of the pathname in this field, but any changes you make in the **Directory** or **Base Name** fields are displayed here also.

When you are ready, click **Create** to begin the report creation. The **HTML Report Created** dialog box opens when the report has been successfully created.



This dialog box affirms where the report was stored and gives you the option to open the report. The report generated by this option is an internally linked HTML document. It can be viewed online in any browser, and can be printed using the print capability of the browser. An example report generated from the parameters selected in the **Create HTML Report** dialog box is shown here.

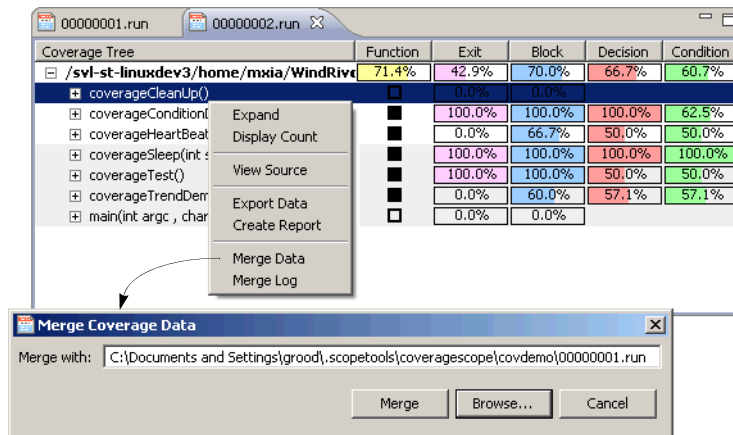
This HTML report has been tested and verified correct on both Netscape and Internet Explorer browsers. It has been executed successfully on a few other browsers, but not rigorously tested on any others.

On a Windows platform, the report is opened in your default browser when it is started from Code Coverage Analyzer. When Code Coverage Analyzer opens the report from a UNIX platform it will try to launch **Mozilla**, then **Firefox**, and then **Netscape**. If none of these browsers are installed, an error message is displayed.

#### 4.3.6 Merge Data Files

To merge output data files, right-click on any entry in a **Coverage Summary** view and select **Merge Data** to open the **Merge Coverage Data** dialog box.





This dialog box controls the process of merging saved coverage data files into the current coverage output in a **Coverage Summary** view, or with other saved coverage data files. Enter the file name (00000001.run in this example) to be merged in the **Merge with** field (or use **Browse** to navigate to a desired file), then click **Merge** to complete the file merging.

Merging coverage data is useful for constructing a more inclusive picture of code coverage when it is not possible to design tests that exercise the entire application in one pass. Separate coverage tests, run at different times, can be merged together to display results as if they had all executed at the same time, yielding a more complete code coverage analysis of the source code.

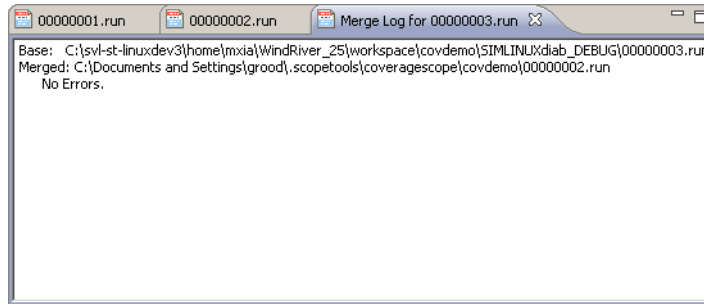
The merge process is subject to the following constraints:

- The source code for all coverage tests to be merged must not have changed.
- The same coverage types must have been selected for each test (see [Coverage Types Tab](#), p.29).
- The same options for inlined functions in source and header files must have been selected from the **Other Options** tab view of the **Instrumentor Options** dialog box for all tests (see [Other Options Tab](#), p.33).

These conditions are all checked by Code Coverage Analyzer for any selected files before they are merged together. If any conditions are not met, an appropriate error is posted in the **Merge Log** window and the merge is aborted.

## Merge Log

The Merged Log view appears as a new Editor view at the completion of a merge operation. It displays the merge status and any error messages generated in the process.

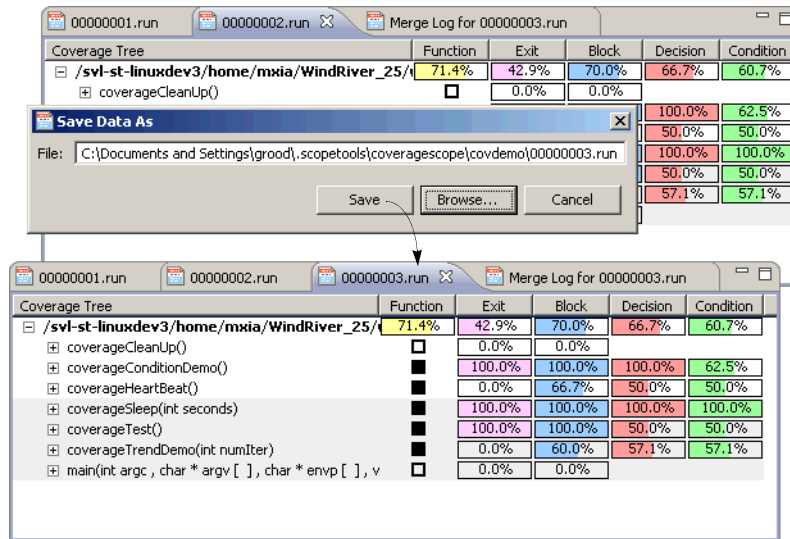


The data in the log for that window is only relevant to that window (that is, each saved data file containing merged data displays only its own **Merge Log** window). If closed, it can be opened again at any time by right-clicking anywhere in the view containing the merged file (the **00000002.run** file, in this example) and selecting **Merge Log**.

## Saving Merged Files

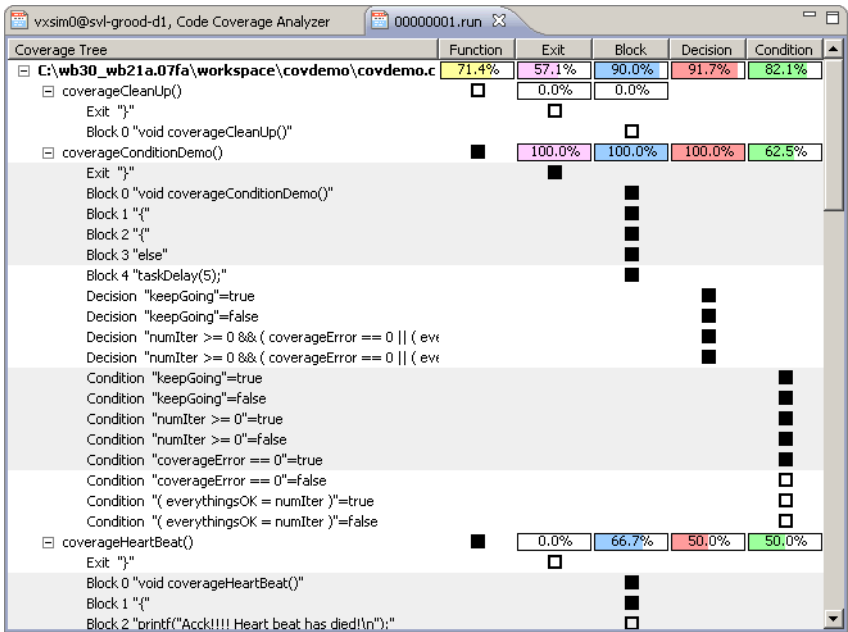
When a data file is merged into another file, the file containing the merged data is not automatically saved, and the merged data will be lost if you close the view without saving it.

Merged data can be saved using the **File > Save As** Workbench menu command from the view displaying the merged data (the **00000002.run** file, in this example).



## 4.4 Viewing Saved Data

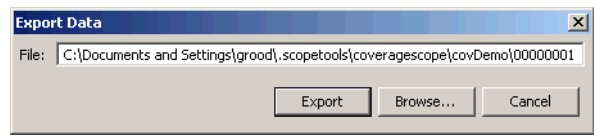
Using the Workbench toolbar command **File > Save As** to open the **Save Data As** dialog box, you can save all the output data from the beginning of the session up to the moment you select the command, in a file you select or specify (see [Saving Output Data](#), p.48). This file can then be opened in the Code Coverage Analyzer GUI for viewing at any time using the Workbench **File > Open File** menu command. This action causes the selected file tab view to open in the Editor view, with the file name appearing in the tab.



All the same functions and views available in the **Coverage Summary** view are also available in the saved data view, including the **Source view**, **Trend Graph**, **Distribution Graph**, and **Coverage Report**.

## 4.5 Exporting Data

You can create a file in Code Coverage Analyzer containing the coverage output data displayed in a **Coverage Summary** view, but formatted in ASCII for display in a spreadsheet application. In the **Coverage Summary** window, use the pop-up menu command **Export Data** (see [Pop-up Menu](#), p.47) to open the **Export Data** dialog box where you can select the pathname and filename for the exported data file. It will be saved as a comma-separated value file with the extension **.csv**.



You can use the default pathname and filename shown in this dialog box, or enter any other file name and location you want. Either enter your choice directly in the **File** field, or click **Browse** to navigate to an existing pathname and enter your filename. When you are ready to save the file, click **Export**.

**Tip:** Be sure to note the pathname and filename you have selected, because once you click **Export** this information is no longer displayed anywhere. This file is not maintained by Code Coverage Analyzer.

An example of this file is shown here, as viewed in a typical spreadsheet.

	A	B	C	D	E	F	G	H
1	File	Function	Code	Start Line	End Line	Coverage	Value	Covered
2	C:\test\cov	coverageCleanup()		79	90	function		0
3	C:\test\cov	coverageCvoid cover		79	90	block		0
4	C:\test\cov	coverageConditionDe		149	171	function		1
5	C:\test\cov	coverageCvoid cover		149	156	block		1
6	C:\test\cov	coverageC{		157	160	block		1
7	C:\test\cov	coverageC{		161	164	block		1
8	C:\test\cov	coverageCelse		165	168	block		1
9	C:\test\cov	coverageCtaskDelay(		169	171	block		1
10	C:\test\cov	coverageCkeepGoing		156	156	decision	TRUE	1
11	C:\test\cov	coverageCkeepGoing		156	156	decision	FALSE	1
12	C:\test\cov	coverageCnumlter >=		160	160	decision	TRUE	1
13	C:\test\cov	coverageCnumlter >=		160	160	decision	FALSE	1
14	C:\test\cov	coverageCkeepGoing		156	156	condition	TRUE	1
15	C:\test\cov	coverageCkeepGoing		156	156	condition	FALSE	1
16	C:\test\cov	coverageCnumlter >=		160	160	condition	TRUE	1



# 5

## *Using the Command-Line Interface*

- 5.1 Introduction 65
- 5.2 Commands 66
- 5.3 Procedures 70
- 5.4 Example Script Files 72

### 5.1 Introduction

The Wind River Code Coverage Analyzer command-line interface (CLI) provides the ability to upload and manipulate coverage data from a target without using the Code Coverage Analyzer GUI. It is designed to handle large code bases that have much greater memory requirements than the GUI can handle. If you are running the GUI on a very large instrumented code file and the progress meter stalls while loading the code on the target, you may want to try testing this code with the command-line version of Code Coverage Analyzer. By using special caching and data lookup techniques, it has 26 times the memory capability of the GUI.

In addition to featuring increased memory capacity, you can also add coverage testing to existing test scripts, or generate a coverage test script specifically to collect Code Coverage Analyzer data. This enhances the capabilities for autonomous testing, and can reduce routine and monotony, particularly when many short tests are to be run. The data is saved in a file for later analysis.

Special commands can be executed directly in a command shell, or added to a host-based test script. The commands collect and store the same data that would have been collected by the running GUI. In many cases the stored data can later be loaded and displayed by the GUI, or it can always be merged with other Code Coverage Analyzer data. You can also use the command-line interface to generate HTML reports from this stored data, as well as exporting the data in a comma-separated value (.csv) format used in spreadsheet and other similar applications.

For information on any errors encountered in using the CLI, and their resolution, see [6. Troubleshooting](#).

## 5.2 Commands



**NOTE:** Before running the CLI, the `WIND_SCOPETOOLS_BASE` environment variable must be explicitly set to the location where the Run-Time Analysis Tools were installed on your system. This is most easily accomplished by executing the command, for example:

```
wrenv -p workbench-3.0
```

This properly sets up the environment variables to allow you to use the commands described in [5.3 Procedures](#), p.70.

The interface has two commands, which can be entered from the command-line or in a script. The commands, described in the following sections, are:

- **coverageupload**
- **coverageconvert**

### coverageupload

The **coverageupload** command connects to your target and uploads requested data into the specified file, in the Code Coverage Analyzer run format.

The full command options for **coverageupload** are:

```
coverageupload [-project project_name.prj] [-tgtsvr target_server]  
               [-file file_name.run] [-verbosity n]  
               [-overwrite true/false]
```



where the command-line arguments are:

**-project** *project\_name.prj* (required)

The full path to the Code Coverage Analyzer project, including the **.prj** extension.

**-tgtsvr** *target\_server* (required)

The name of the target server to use for connection to the target. This must be a valid (fully qualified) target server name, such as **walnut@sv1-grood-d1**.

**-file** *file\_name.run* (required)

The name of the file to save the data in. Use the **.run** extension. By default, the file will be put in the directory in which you are running, but you can also enter a full pathname.

**-verbosity** *n*

A number from 0 to 3 that determines volume of status output. The default is 0.

**-overwrite** *true/false*

Specifies whether to overwrite (**TRUE**) or not to overwrite (**FALSE**) the *file\_name.run* file. The default is **FALSE**.

The GUI does not have to be running to use this command, but a target server is necessary to connect to the target. The command connects the target to the specified target server, uploads the data once, and saves it in the specified **.run** file.

A project file name, including the **.prj** extension, must be specified in order to know which data to upload from the target. You must specify the same project that was used to compile the instrumented code, and the full pathname to that project file must be entered. This pathname to the project file can be found in either of two ways:

1. Set the verbosity to 1.

This will print out the full path to the project file in the shell window.

2. Select **Open > Project**.

The **Open Project** dialog box opens, displaying the name and pathname of all your current projects. From this list you can note the full pathname to the project you are looking for.

It is recommended that you save the data in a file with a **.run** file extension to identify it as a Code Coverage Analyzer run file. By default, Code Coverage Analyzer will not overwrite an existing run file, but if you do want to overwrite an existing file, set **-overwrite** to **TRUE**.

The **coverageupload** command cannot merge results. If you want to merge the results of this Code Coverage Analyzer run with an existing **.run** file, specify a new file name for *coverageupload*, then later merge this file with the previous results using **coverageconvert**, described below.



**NOTE:** It was noted above that data is uploaded by **coverageupload** only once. Hence, the small time value for **Duration**, reported in the HTML report, reflects the time it took **coverageupload** to execute (not the time span of **coverageconvert** as it monitored the running target code).

## coverageconvert

The **coverageconvert** command manipulates Code Coverage Analyzer data previously collected by a **coverageupload** command. It can perform the following operations:

- Merge the Code Coverage Analyzer **.run** file into another Code Coverage Analyzer **.run** file.  
Only two files can be merged together at a time. To merge more than two files together, call **coverageconvert** repeatedly, specifying the same *merge\_file.run* name (following the **-merge** option) each time. If the *merge\_file.run* file does not exist, **coverageconvert** will create it and merely copy the *file\_name.run* file to it.
- Export the Code Coverage Analyzer run file into a comma-separated value (**.csv**) file.  
The **.csv** file generated with this option can be imported directly into a spreadsheet.
- Generate an HTML report from the Code Coverage Analyzer run file.  
The HTML report generated is the same report generated by Code Coverage Analyzer. Its format is described in detail in [4.3.5 Coverage Report](#), p.56.

The basic command options for **coverageconvert** are:

```
coverageconvert file_name.run [-merge merge_file.run]  
[-export csv_file.csv] [-html report_directory]
```

where the arguments are:

*file\_name.run* (required)

The name of the **coverageupload** data output file.

- merge** *merge\_file.run*  
Tells Code Coverage Analyzer to merge the contents of *file\_name.run* file into the existing *merge\_file.run* file.
- export** *csv\_file.csv*  
Tells Code Coverage Analyzer to transform data in the *file\_name.run* file into a comma-separated value (.csv) file, *csv\_file.csv*.
- html** *report\_directory*  
Causes Code Coverage Analyzer to generate an HTML report from the Code Coverage Analyzer run data collected, and save it in the (required) *report\_directory* directory.

There are optional commands available that control the HTML report format. They include the following:

- coverageType** *function/exit/block/decision/condition*  
Any of these coverage types selected are included in the report.
- generalInfo** *true/false*  
If set to **TRUE**, this option shows general information in the main page.
- totals** *true/false*  
If set to **TRUE**, this option shows totals information in the main page.
- graph** *true/false function/file*  
If set to **TRUE**, this option shows a distribution graph by functions or files.
- summaryByFunc** *true/false*  
If set to **TRUE**, this option shows a statistical summary by functions.
- summaryByFile** *true/false*  
If set to **TRUE**, this option shows a statistical summary by files.
- highlightSource** *true/false function/exit/block/decision/condition*  
If set to **TRUE**, this option highlights source code lines containing the selected coverage types.
- notCovered** *true/false*  
If set to **TRUE**, this option displays summary of source code that is not covered.



**NOTE:** By default, if none of the above optional commands are specified, it is equivalent to specifying:

```
-coverageType function block decision condition -generalInfo true
-totals true -graph true function -summaryByFunc true
-summaryByFile true -highlightSource true block -notCovered true
```

## 5.3 Procedures

This section outlines the steps to retrieve data from your VxWorks targets and save it for later analysis.

**Step 1: Execute the following instructions to set up the proper environment.**

On your Windows or UNIX host, run the `wrenv` program as follows:

```
WIND_HOME/wrenv.exe -p workbench-3.0 (Windows)
```

```
WIND_HOME/wrenv.sh -p workbench-3.0 (UNIX)
```

This will properly set up the environment variables to allow you to use the `coverageupload` and `coverageconvert` commands.

**Step 2: Start `wtxregd` in the shell.**

**Step 3: Start your target server running in the shell.**

**Step 4: Run `dfwserver`.**

To do that, follow these steps:

- a. It is recommended that you unregister any `dfwserver` left over from previous activity before continuing (it may not always be necessary, but it is a good practice). Use the following command:

```
%WIND_SCOPETOOLS%\dfw\dfwrelease\x86-win32\bin\dfwserver.exe -unregister  
-userregistry -session dfwservername -registryhost localhost
```

where:

*dfwrelease* is the latest dfw release directory, for example,

**0160j**

*dfwservername* is the dfw server name you are using, for example:

**dfw-wb30-dsmith**

- b. Start `dfwserver` to take commands from the console and sockets using the command:

```
%WIND_SCOPETOOLS%\dfw\dfwrelease\x86-win32\bin\dfwserver.exe -daemon  
-protocol mi -io stdio -io sockets -userregistry  
-session dfwservername -registryhost localhost
```

Where the same placeholders are as described above.

**Step 5: In the dfw window, connect dfw to the target server.**

To do this, follow these steps:

- a. Define both the target server name and the path to the kernel image using the command:

```
-wrs-target-define targetserver@hostname unifiedtargetplugin
-tgt "DEVICE='WTX_VXWORKS',ADDR='targetserver@hostname',
KERNEL='D:/wb26_22c23a/workspace/walnut/default/vxWorks' "
-auto-cpu-all
```

Where *targetserver@hostname* is your target server/host name, for example:

```
tgt_128.224.45.33@sv1-grood1
```

There will be a response similar to,

```
"done, defName='targetserver@hostname' "
```

- b. Connect dfw to the target, changing the name of the target to your target, using the command:

```
-target-select wrs-remote-dfw targetserver@hostname
```

There will be several responses, such as,

```
"=connected,thread-id='1',def-name='targetserver@hostname' "
```

```
"=connected,thread-id='2',def-name='targetserver@hostname'
core-name='405GP',system-context-thread-id=3"
```

```
"=modules-changed,thread-id-'2',modules=[{id='0x1',
name='D:/wb26_22c23a/workspace/walnut/default/vxWorks'
,file='D:/wb26_22c23a/workspace/walnut/default/vxWorks'
,symbols='1',reserved='0',image='1'}]
done.connected-cores=[{core-name='405GP',thread-
id='2'}]
```

**Step 6: On the target, load the full path to the module as it was compiled.**

Do this by entering:

```
ld 1,1,"D:/wb26_22c23a/workspace/covdemo/PPC405sfdiab_DEBUG/
covdemo.out"
```

**Step 7: On the target, execute coverageTest.**

In coverageTest, do the following:

- a. Use `coverageupload` to upload the coverage data from the target with the command:

```
coverageupload -tgtsvr targetserver@hostname -file coveragedata.run  
-project projectname.prj
```

Where *projectname.prj* is the pathname to your project, for example:

```
/home/grood1/.scopetools/coveragescope/default/default.prj
```

- b. Generate a coverage report with the command:

```
coverageconvert coveragedata.run -html pathnametoHTMLfile
```

Where *pathnametoHTMLfile* is the pathname to where you will store your HTML output file, for example:

```
/home/grood1/tmp/html/
```

## 5.4 Example Script Files

This section presents an example shell script that tests the **covdemo.c** program shipped with Code Coverage Analyzer. The script references two additional TCL script files, also presented.

### Shell Script File: **testcovdemo.sh**

This shell script file gives you an example of how you can use the Code Coverage Analyzer scripting feature to set up and collect coverage data from your target. Two TCL scripts are referenced within this file, one to download object files and start the tests, and the other to shut down the target servers. Finally, the script merges the output files and generates a Code Coverage Analyzer HTML report.

```
#!/bin/sh  
#  
# Example script for collecting CodeCoverageAnalyzer data from two  
# targets, merging the data, and creating a report  
#  
# Assumes that WIND_BASE and WIND_REGISTRY have been set, e.g. by  
# sourcing torVars.csh  
#
```

```

# Start the target servers
tgtsvr katmai -A -s -n katmai -c /raid0/local/rts/export/root/katmai/vxWorks
&
tgtsvr henry-2 -A -s -n henry-2 -c /raid0/local/rts/export/root/henry-
2/vxWorks &

# Call tcl script that downloads the object modules from the
# target and runs the tests
wtxtcl runcoveragetests.tcl

# Remove coverage data from the last run
rm coveredata*.run

# Use coverageupload to upload the coverage data from the targets
coverageupload -tgtsvr katmai@mammoth -file coveredata_katmai.run -project
/home/heidi/.scopetools/coveragescope/default/default.prj

coverageupload -tgtsvr henry-2@mammoth -file coveredata_henry-2.run -
project /home/heidi/.scopetools/coveragescope/default/default.prj

# Shutdown the target servers
wtxtcl killtgtsvrs.tcl

# Merge the coverage results into one run
coverageconvert coveredata_katmai.run -merge coveredata.run
coverageconvert coveredata_henry-2.run -merge coveredata.run

# Generate a coverage report
coverageconvert coveredata.run -html /home/heidi/tmp/html/

```

### TCL Script File: runcoveragetests.tcl

```

# Sleep for a bit to let target servers come up
msleep 500

# Attach to katmai
wtxToolAttach katmai@mammoth coverageTest
# Load and run tests (call coverageTest())
wtxObjModuleLoad LOAD_ALL_SYMBOLS covdemo.o
set coverageTestAddress [lindex [wtxSymFind -name coverageTest] 1]
wtxFuncCall $coverageTestAddress
# Sleep to let test run, then detach
msleep 9000
wtxToolDetach

# Attach to henry-2
wtxToolAttach henry-2@mammoth coverageTest
# Load and run tests (call coverageCleanUp())
wtxObjModuleLoad LOAD_ALL_SYMBOLS covdemo.o
set coverageCleanUpAddress [lindex [wtxSymFind -name coverageCleanUp] 1]
wtxFuncCall $coverageCleanUpAddress

```

```
# Detach  
wtxToolDetach
```

### **TCL Script File: killtgtsvrs.tcl**

```
# Shutdown katmai  
wtxToolAttach katmai@mammoth coverageTest  
wtxTsKill WTX_OBJ_KILL_DESTROY  
  
# Shutdown henry-2  
wtxToolAttach henry-2@mammoth coverageTest  
wtxTsKill WTX_OBJ_KILL_DESTROY
```



# 6

## *Troubleshooting*

- 6.1 Introduction 75
- 6.2 GUI Messages 75
- 6.3 Command-Line Interface Messages 76
- 6.4 Troubleshooting Tips 79

### 6.1 Introduction

If you get error messages, or are having problems getting Wind River Code Coverage Analyzer to work, check the error messages and troubleshooting tips in this chapter to see if they resolve your problems. If you are still unable to get Code Coverage Analyzer to work, email the Technical Support Team.

### 6.2 GUI Messages

Message traffic within the Code Coverage Analyzer GUI, and with its external parts, is formatted and displayed in a variety of places.

## Status

Status messages appear in the **Analysis Console** view, described in [Console View](#), p.20, while warning messages often appear in pop-up windows.

## Error

In the process of starting and running various Code Coverage Analyzer tasks, error messages may appear. Error messages for the Code Coverage Analyzer GUI are displayed in the **Analysis Console** view. This section lists the error messages most likely to be encountered.

### Lost Connection

If Code Coverage Analyzer loses its connection with the target, you will see the error message:

```
Lost connection to target.
```

in the status bar. Possible causes are:

- The target was rebooted.
- The network connection was interrupted or disconnected.
- The target server is busy, or the machine running the target server is busy.

### Instrumentor Errors

When an instrumentation error occurs, the **Instrumentor Errors** window, which opens automatically on top of the **Coverage Summary** window, displays the error messages from the instrumentor. If it is not open, you can select **Project > instrumentor Errors** to open it at any time.

## 6.3 Command-Line Interface Messages

The following list contains error and warning messages that the **coverageupload** and **coverageconvert** options may generate when run on the command-line interface (CLI) (see Chapter 5. [Using the Command-Line Interface](#)). These messages are sent to **stdout**. In addition, the **coverageupload** command creates a log file

containing extra status information on the data collection run, as well as any of these error or warning messages it generates. The log file is written to:

`user_home/.scopetools/coveragescope/coverage.log`

## Warning

2000 Could not find data on the target for a source file that had been instrumented for this project.

This error probably means that the code was not loaded on the target for this source file. It is reported as 0% covered.

2001 Could not load the CodeCoverageAnalyzer information file for this source code.

This file does not show up in the Code Coverage Analyzer run or report.

2002 Found a bad tag in the CodeCoverageAnalyzer information file.

The Code Coverage Analyzer data reporting could have an error for this source file.

2003 Found bad data in the CodeCoverageAnalyzer information file.

The Code Coverage Analyzer data reporting could have an error for this source file.

2004 Had a data error on coverageupload.

Either the size of the array on the target did not match what the host expected, the key did not match, or the array was unknown. Clear all data in the project from the GUI (**Project > Clear Results**) and recompile the entire project.

## Error

1000 Need to specify a target server for coverageupload.

1001 Need to specify a .run file for either the output of coverageupload or the input of coverageconvert.

1002 The run file specified after -file already exists.

Use the argument **-overwrite true** to overwrite the file or specify a different file name.

1003 Need to specify a project for coverageupload.

1004 Could not find the specified project file.

Verify that the full path to the file is specified and is correct. Also verify that the .prj extension is included.

1005 Error opening a project.

Make sure you have read permissions to the project file.

1006 Target server *tgt server* is not found.

Verify that the target server *tgt server* is up and running, and that you specified a fully qualified target server name, such as **walnut@sv1-grood-d1**.

1007 Error in the CodeCoverageAnalyzer link to the target.

The most likely cause of this error is either the target rebooting while **coverageupload** is uploading the Code Coverage Analyzer data or a crash of the target server.

1008 No CodeCoverageAnalyzer information files (.tid) were found for the project.

This error probably means no source files were instrumented for the project.

1009 Error while reading in CodeCoverageAnalyzer information files.

Either none were successfully loaded or there was another error.

1010 API\_SERVER\_NOT\_FOUND

(Same cause and remedy as error 1006 above.)

1100 Cannot find the specified .run file.

Verify that it exists.

1101 Cannot read the specified .run file.

The file could be corrupted.

1102 Do not have write permission to the file.

1103 Failed to export to the specified file.

1104 Failed to save the .run file.

Verify that you have permission to write to this file.

1105 Merging of the two runs failed.

1106 Could not create the directory for the report.

1107 Failed to generate the report.

3001 The environment variable WIND\_SCOPETOOLS is not set.

You need to set this environment variable by calling  
**WIND\_HOME/wrenv.exe -p workbench-3.0** (Windows host)  
**WIND\_HOME/wrenv.sh -p workbench-3.0** (UNIX host)

## 6.4 Troubleshooting Tips

This section organizes problem areas by the major Workbench components in which they occur.

### Issues with Instrumentation

#### Code Size and CPU Overhead

As with any procedure that tags or patches a program, Code Coverage Analyzer instrumentation adds a small amount of overhead to your target code, both in terms of code size and also execution time. Since the object of this analysis is the paths taken (or not taken) through your test code, and not on speed or timing issues, the implications of this overhead should not be a problem in most circumstances.

A summary of observed data on these overhead parameters is given in Appendix [B. Performance Metrics](#). The information presented there will help you determine the impact of the test coverage types you would like to run on your target.

### Issues with the Target

#### Target Connection Lost

A message appears alerting you to the fact.

### Issues with the GUI

#### Instrumentor Errors

You can obtain more detailed information by increasing the verbosity level, set in the **Other Options** tab view of the **Instrumentor Options** dialog box, then running the instrumentor again. Be aware, however, that specifying a larger value (in the range of 1-3) generates an increasingly greater variety and volume of instrumentor messages. The default is 0.

#### Compiler Errors

If an error occurs when compiling your instrumented code, check the compiler message log for information about the error. The compiler message log is usually

located where your compiler output is displayed. If you are running on Workbench, the compiler output typically appears in the **Build Console** view.

### File Errors

If the **Coverage Summary** view appears to open successfully, but one or more of the listed files does not change from italics to bold after just a few seconds, a potential problem with that file is indicated.

The following are possible file-related problems and suggested remedies:

- If there were no instrumentor or compiler errors, check the shell for loading error messages. Try downloading the files again after correcting any errors found.
- Check that the indicated source code files were not compiled either with another project, or with coverage turned off (see [Covered Files Tab](#), p.31). In this case, recompile your source code and try again.

# A

## *Code Coverage Types*

<a href="#">A.1 Introduction</a>	81
<a href="#">A.2 Purpose of Code Coverage</a>	82
<a href="#">A.3 Types of Coverage</a>	83
<a href="#">A.4 Coverage Type Hierarchy</a>	89

### **A.1 Introduction**

This chapter describes code coverage types and the role of a code coverage analyzer, such as Wind River Code Coverage Analyzer, as used in the software testing process.

Code coverage analysis is the process of determining which code statements have, and which have not, been exercised by a software test case. A code coverage tool provides the developer with data about how much of the code has been traversed and which specific parts of the code were not visited. This enables the developer to modify software test cases to more fully cover all areas of the code.

Various types of coverage, as discussed in [A.3 Types of Coverage](#), p. 83, provide the developer with different information about the effectiveness of the test cases.

## **A.2 Purpose of Code Coverage**

A code coverage analyzer provides the developer with data about how much of the source code has been exercised by a software test case. For instance, a code coverage analyzer may report the percentage of total functions that have been invoked, or the percentage of executable statements that have executed. This gives the developer a measure of how comprehensive the test case is. A code coverage analyzer also reports to the developer which parts of the code have not been executed, such as generating a list of functions that have not been called. The developer can then use these reports to refine the software test cases to better exercise the code.

The main point to understand about using a code coverage analyzer is that code coverage measures the quality of the software testing effort, not the quality of the software itself. A code coverage analyzer does not detect or locate bugs in the software, nor does it tell if the code meets all of its specified requirements. Code coverage results are a tool used to analyze the effectiveness of the software testing effort.

One of the primary goals of a code coverage analyzer is to find and identify those areas of code that are not being tested so the test cases themselves can be modified or augmented. The code coverage report indicates which parts of the code have not been traversed by the test. The code coverage analyzer does not generate test software for the developer, but rather points out to the developer which areas of the code were not exercised. Used properly, the code coverage results can enable the software developer to improve both the quality and scope of the software test cases.

As a secondary goal, code coverage can be used to identify redundant test cases that exercise identical code, thus shortening the testing time. Code Coverage Analyzer does this by displaying a trend graph of coverage level versus time. A flat section of the graph indicates the software tests are simply rerunning the same code and may possibly be eliminated.

Code coverage tests can be used for various levels of software testing. A software developer may use a code coverage analyzer to verify that a critical function has been fully tested. A software quality tester may use code coverage to ensure a certain level of coverage for the entire software test effort before the release of the product. Different types of coverage are useful for these different purposes, as discussed in the next section.



## A.3 Types of Coverage

Code Coverage Analyzer provides five types of code coverage, allowing you discrete steps in increasing levels of coverage capability.

The types of code coverage provided are:

- **Function**
- **Function Exit**
- **Block**
- **Decision**
- **Condition**

A

### Function

**Function** coverage reports whether or not each function in the source code has been called. A coverage tag is placed in the entry (but not the exit) of each function, and the function is considered covered if it has been entered at least once. **Function** coverage is useful as an initial coverage goal of verifying that a software test case exercises all major parts of the code.

### Function Exit

**Function Exit** coverage reports whether every exit from a function has been taken. There are one or more function exits for every function. One exit is always the end of the function, while other exits may be **return**, **exit()** or **abort()** statements.

**Function Exit** coverage reports whether or not the bottom of the function was reached. However, in some cases this end is never reached because of one or more previous exit statements. It is up to the user to prove that the final function exit is indeed unreachable and to not show **Function Exit** coverage.

For example, the following function has three exits, but only two are reachable:

```
void ExampleExitFunction(int a)
{
    if(a) {
        ...
        return ; /* First Function Exit */
    } else {
        ...
        return ; /* Second Function Exit */
    }
    /* Third Function Exit */
}
```

## Block

**Block** coverage reports whether every non-branching block of code has been executed at least once.

Block coverage provides the same information as coverage for every individual statement would. "Statement" coverage would report whether each statement has been executed. But since Code Coverage Analyzer assumes that every consecutive statement is always executed until a branch is encountered in the code, it is more concise to break the code up into blocks. Each block consists of all the statements between branches, and coverage is reported by blocks.

The rest of this section describes how Code Coverage Analyzer divides and labels blocks. This information may be helpful in interpreting the results of your Block coverage run.

Every function in the source code is divided into blocks, with the first block starting at the beginning of the function and the last block ending at the close of the function. A new block is started every time the code branches, such as an **if** statement or a **while** loop. The various types of branches in C and C++ are broken into blocks as described below.

- **if** statements  
This block begins at the opening brace of the **if** statement and ends at the closing brace. At the end of an **if** or **if-else** block, a new block is started as long as there is another statement following it. If the **if** block is the last statement in the function, it will be the last block in the function, with the block ending at the closing brace.
- **if-else-if** statements  
These are treated as nested **if** statements, but the **else** is the start of the next block, which ends at the following **if** where another new block begins.

For example, the code fragment:

```
void ExampleIfFunction(int a, int b) {  
    if(a==2)  
    {  
        printf("Hello");  
    }  
    else if(b==3 && b>a)  
    {  
        printf("World");  
    }  
    else  
    {  
        printf("\n");  
    }  
}
```

has five blocks:

```
block 1: void func(int a, intb) {  
    if(a==2)  
block 2: {  
        printf("Hello");  
    }  
block 3:else if(b==3 && b>a)  
block 4: {  
        printf("World");  
    }  
block 5:else  
    {  
        printf("\n");  
    }  
}
```

Entering the function always covers block 1. If **a** equals 2, block 2 will be covered, but not blocks 3, 4, or 5. If **a** is not 2 and **b** equals 3 and is greater than **a**, then blocks 3 and 4 will be covered, but not blocks 2 and 5. If **a** is not equal to 2 and **b** is not equal to 3, then blocks 3 and 5 will be covered, but not blocks 2 or 4.

- **loop** statements

A new block is started at the beginning of a **for** or **while** loop. **Block** coverage reports whether this loop has been executed at least once.

A new block is not started at the beginning of a **do-while** loop as this loop is always evaluated at least once.

- **switch** statements

Every non-empty switch case is the start of a new block. Note that **Block** coverage does not tell you whether any case was explicitly encountered, or if it just fell through to that case.

For example, the code fragment:

```
void ExampleSwitchFunction(int a,) {  
    switch(a) {  
        case 0:  
            printf("Hello");  
        case 1:  
        case 2:  
            printf("World");  
            break;  
    }  
}
```

has three blocks:

```
block 1: void ExampleSwitchFunction(int a,) {  
    switch(a) {  
block 2:     case 0:  
        printf("Hello");  
block 3:     case 1:  
        case 2:  
            printf("World");  
            break;  
    } }  
}
```

Block 1 is covered whenever the function is called. Block 2 is covered when **a** equals 0. Block 3 is covered if **a** equals 0,1, or 2.

- **Other branches**  
Code Coverage Analyzer always starts a new block at a **goto** or **label** statement. A new block is also started after any **break**, **return**, or **continue** statement, however, these statements should be at the end of a block already, for example at the end of an **if** statement. If they are not, the code following them will never be reached.
- **Exceptions**  
Code Coverage Analyzer **Block** coverage supports exception handling for C++. Catch blocks are marked as separate blocks for coverage, so the developer must test each catch block to ensure full block coverage.



---

**NOTE:** Catch blocks are C++ programming constructs. For more information, consult any C++ manual.

---

If an exception occurs in the middle of the block, Code Coverage Analyzer reports the block as fully covered even though the exception has caused the control flow to not finish the block.

## Decision

**Decision** coverage reports whether or not each decision statement affecting control flow has been evaluated to both **TRUE** and **FALSE**. **Decision** coverage tells you if every branch in the code has been taken, including empty branches.

The various kinds of decision, or *branching* statements are handled as follows:

- **if** statements  
For **if** statements to be fully covered, the expression in ( ) must evaluate to both **TRUE** and **FALSE**. Thus **Decision** coverage for an **if** statement is similar to **Block** coverage, except the **else** branch must also be taken, whether or not it is

explicitly present. **Decision** coverage helps make you aware of possible errors lurking when an else branch is not taken.

For example, in the code fragment:

```
char *msg;  
if (a==0)  
    msg = "Hello\n";  
printf(msg);
```

If **a** equals 0, **Decision** coverage will report full block coverage. The code actually fails when **a** does not equal 0, though, and is reported only partially covered since the **FALSE** branch (which is explicitly not present) was never taken. **Decision** coverage catches the testing error by requiring the condition to evaluate to **FALSE** as well as **TRUE**.

For **else-if** statements, **Decision** coverage is the same as block coverage.

- **loop** statements

**Decision** coverage reports whether the loop condition evaluates to both **TRUE** and **FALSE**. This differs from block coverage in that a loop may terminate from within the loop, and may never actually terminate from a loop condition evaluating to **FALSE**.

**Decision** coverage also tags and monitors **do-while**, **for**, and **while** loops. However, a loop with empty conditions (such as `for(;;)`) will not be tagged and monitored for decision coverage. Also, a loop with a hard-coded **TRUE** or **FALSE** expression (such as `"while(true){...}"`) will never get full **Decision** coverage since **TRUE** is never evaluated to **FALSE**.

- **switch** statements

Coverage of **switch** statements is similar to **Block** coverage, except that it also reports whether the default case was encountered, even if that case is not explicitly present.

For example, the **ExampleSwitchFunction** in [Block](#), p.84, has three decision branches: **a** equal 0, **a** equal 1 or 2, and **a** not equal 0, 1, or 2.

Because it is related to "decisions," the report for decision coverage shows the decisions, not the blocks that are affected by the decisions. So in the **ExampleIfFunction** in [Block](#), p.84, decision coverage reports whether the decision `a==2` evaluated to both **TRUE** and **FALSE**, and whether the decision `b==3 && b>a` evaluated to both **TRUE** and **FALSE**. In this case, because each **if** has an **else** associated with it, decision and **Block** coverage both give the same information, but the reporting is different.

## Condition

**Condition** coverage reports whether or not every subexpression in a Boolean expression evaluated to both **TRUE** and **FALSE**. At first glance it may appear that decision coverage attained that goal, but in reality, even though decision coverage catches many control flow errors that block misses, it may not catch all errors due to the short circuit operation.

For example, the code fragment:

```
if(a && (b || BuggyFunction() )) {...}
```

reports full decision coverage if evaluated with **a** equal **TRUE** and **b** equal **TRUE**, as well as with **a** equal **FALSE**. However, neither case called the buggy function.

**Condition** coverage, on the other hand, requires that each Boolean subexpression be evaluated to both **TRUE** and **FALSE**. **Condition** coverage does not require the full Boolean expression to evaluate to both **TRUE** and **FALSE**, nor does it require the full truth table to be tested. Because **Decision** coverage *does* require the full Boolean expression to evaluate to both **TRUE** and **FALSE**, **Condition** coverage and **Decision** coverage are often combined to create **Condition/Decision** coverage.

**Condition** coverage adds tags to condition subexpressions anywhere in the code, looking for any expression separated by "&&" or "||." Coverage is handled for the various condition statements as follows:

- **if** statements

In addition to tagging condition subexpressions anywhere, condition coverage adds a tag to any expression in an **if** statement, tagging subexpressions only if they exist. For example, in the **ExampleIfFunction** in [Block](#), p.84, condition coverage requires each of the subexpressions **a==2**, **b==3**, and **b>a** to evaluate to both **TRUE** and **FALSE**.

In another example, the code fragment:

```
d=a && b && c;  
if(d) {...}
```

checks **a**, **b**, and **c**, as well as **d** when it is in the **if** statement, for evaluating to both **TRUE** and **FALSE**.

In contrast, the code fragment:

```
if(a && b && c) {...}
```

checks only **a**, **b**, and **c** for this condition.

- **loop** statements  
Like the **if** statement, condition coverage reports any subexpressions found in **while** and **for** loop statements, or the whole expression if it has no subexpressions.
- **switch** statements  
**Condition** coverage reports whether each **case x** was actually encountered (in other words, did the switch condition equal **x** at least once). It also reports whether the default case was encountered, even if that case is not explicitly present.  
  
In the **ExampleSwitchFunction** in [Block](#), p.84, full coverage would require **a** to have evaluated at some point to **0**, to **1**, to **2**, and to anything but **0**, **1**, or **2**.

A

## A.4 Coverage Type Hierarchy

The coverage types described above are grouped from general to specific measures of coverage. Within these coverage types, however, there are implications that hold true for all occurrences irrespective of hierarchy. They are:

- **Full Block** coverage implies that all functions are also covered.
- **Full Decision** and **Full Function** coverage implies that blocks are also covered.
- **Condition** coverage alone does not imply that blocks or decisions are covered.





# *B*

## *Performance Metrics*

**B.1 Introduction 91**

**B.2 Supported Targets Data 92**

### **B.1 Introduction**

This appendix contains Wind River Code Coverage Analyzer data describing the increased code size for target code that has been instrumented by the **coverage** instrumentor program. Each entry in this table represents a specific brand and model of target supported by this release. A complete list of supported targets for this release is found in the following documents.

- For **VxWorks** targets:  
*Wind River Workbench for VxWorks 6.6 Release Notes*
- For **Linux** targets:  
*Wind River Workbench for Linux 1.5 Release Notes*

## B.2 Supported Targets Data

The code size overhead statistics presented in the tables below was generated using the Wind River program **cobble.c**. [Table B-1](#) presents data collected for a VxWorks target.

Because the increase in code size is highly dependent on the specific code that you instrument, these numbers should be viewed as rough estimates. As expected, the Code Coverage Analyzer run time execution of your code (cpu overhead) also increases, typically between 5% and 50%.

The key for the **% Code Size Increase** parameters in [Table B-1](#) is as follows:

- F** = Function coverage only.
- FB** = Function and Block coverage only.
- All** = All coverage types.

Table B-1 **Code Size Overhead**

Family	CPU	Target Agent Size (Kb)	% Code Size Increase		
			F	FB	All
68k	MC68000	78	5	11	16
	MC68020	78	5	11	16
	MC68040	78	5	11	16
	MC68060	78	5	11	16
	MC68LC040	78	5	11	16
ARM	ARM7TDMI	113	7	12	13
	AMARCH4	108	7	10	12
CPU32	CPU32	74	5	11	16
Coldfire	MCF5200	91	4	9	14
	MCF5400	91	4	9	14
Fujitsu FR-V	FR500	117	5	9	11

Table B-1    **Code Size Overhead** (cont'd)

Family	CPU	Target Agent Size (Kb)	% Code Size Increase		
			F	FB	All
Hitachi SH	SH7600	93	4	7	10
	SH7700	93	4	7	10
	SH7750	97	3	6	12
MIPS	MIPS32sf	122	6	12	18
	MIPS32sfr3k	126	6	12	17
	R3000	115	6	13	15
	R3000sf	115	6	13	15
	R4000	110	6	13	15
	R4000sf	110	6	13	15
	R4650	110	6	13	15
	VR4100sf	111	6	13	15
	VR5000	121	6	12	18
	VR5400	121	6	12	18
PPC	PPC403	127	5	9	10
	PPC405	128	5	9	10
	PC405f	129	5	9	10
	PPC603	128	5	9	10
	PPC604	133	5	9	11
	PPC860	127	5	9	10
	PPCEC603	128	5	9	10
RC32364	RC32364sf	111	6	13	15
Simulators	SIMNT	89	9	14	18

**B**

Table B-1    **Code Size Overhead** (cont'd)

Family	CPU	Target Agent Size (Kb)	% Code Size Increase		
			F	FB	All
	SIMSPARCSOLARIS	107	5	9	11
StrongARM /XScale	ARMSA110	113	7	12	13
	STRONGARM	108	7	10	12
	XSCALE	109	7	10	12
x86	I80386	78	5	11	17
	I80486	84	6	11	17
	PENTIUM	84	6	11	17

# C

## Glossary

### **block**

A contiguous group of non-branching code statements, which are, by definition, guaranteed to be traversed completely when entered.

### **Block Coverage**

A search for, and report on, each contiguous **block** of non-branching code statements. For further context, see [A.3 Types of Coverage](#), p.83.

### **Code Coverage Analyzer instrumentor**

The Wind River software tool that adds instrumentation tags to the source code statements, then calls the compiler. For further context, see [3.3 Instrumenting and Compiling](#), p.35.

### **condition**

Any sub expression in a **decision** statement in the code, containing one Boolean expression, or multiple Boolean expressions separated by "&&" or "||."

### **Condition Coverage**

This coverage type searches for, and reports on, whether or not every **condition** sub expression in a **decision** statement evaluated to both TRUE and FALSE. For further context, see [A.3 Types of Coverage](#), p.83.

## decision

Any testing and branching statement in the code that affects control flow. It contains one or more **condition** sub expressions.

## Decision Coverage

This coverage type reports whether or not each decision statement has been evaluated to both **TRUE** and **FALSE** overall. For further context, see [A.3 Types of Coverage](#), p.83.

## function

A self-contained code module that can accept input, execute, and produce output.

## Function Coverage

This coverage type reports whether or not each **function** in the code has been called. For further context, see [A.3 Types of Coverage](#), p.83.

## Function Exit Coverage

This coverage type reports whether every **exit** from a function has been taken. For further context, see [A.3 Types of Coverage](#), p.83.

## instrumentation parameters

The values, selected by the user, that determine the type and placement of tags in source code files by the **Coverage Instrumentor** for coverage monitoring purposes. For further context, see [3. Instrumenting Source Code](#).

## preprocessed source code

Intermediate source code file that has had its **include** files and other preprocessor commands expanded by the compiler prior to being compiled. For further context, see [3. Instrumenting Source Code](#).

## Software test case

The set of parameters and input data created by the software developer and used to test software modules in the development process. Code Coverage Analyzer provides a quantifiable measure of the effectiveness of software test cases. For further context, see [1.1 Introduction](#), p.1.

**tag information database (TID)**

A file created by the coverage instrumentor that records the instrumentation parameters in effect for the source code. The file is given the extension **.tid**. For further context, see [3. Instrumenting Source Code](#).

**target agent**

The part of Code Coverage Analyzer that runs on the target.

**truth table**

A two-dimensional matrix of all possible **TRUE** and **FALSE** values for each of the unique conditions in a **condition** statement.

**verbosity**

Controls the type and number of messages generated by:

- The coverage instrumentor during compilation (see [Other Options Tab](#), p.33). The value can range from **0** (least verbose) to **3** (most verbose).
- The target server connection process (see [4.2 Starting Data Collection](#), p.44). The value can range from **0** (least verbose) to **3**(most verbose).

For further context, see [4.2 Starting Data Collection](#), p.44.





# Index

## A

architecture  
  coverage instrumentor 4  
  GUI 4  
  summary 4

## B

block coverage 84

## C

CLI  
  coverageconvert command 68  
  coverageupload command 66  
  description and benefits 65  
  summary of commands 66  
  using shell scripts 72  
code coverage analysis  
  described 81  
  purpose 82  
Code Coverage Analyzer  
  features 5  
  how it works 3  
command-line, instrumentor 38

compiler errors 79  
compiling source code  
  from the command-line 37  
  from Workbench 36  
  from your makefile 37  
  in the example session 24  
condition coverage 88  
Console view 20  
coverageconvert command 68  
coverage instrumentor tool 35  
Coverage Summary view 17  
coverage types  
  block 84  
  condition 88  
  decision 86  
  defined 3  
  described 83  
  function 83  
  implications of 89  
Coverage Types color palette 21  
coverageupload command 66  
create a new Workbench project 10  
Create HTML Report dialog box 21

## D

decision coverage 86  
description field

Instrumentor Options dialog box [34](#)  
dialog boxes  
    Create HTML Report [21](#)  
    Select Target Server [12](#)  
Distribution Graph view [19](#)

## E

error  
    lost connection [76](#)  
    messages [76](#)  
example Code Coverage Analyzer session [24](#)

## F

features, list [5](#)  
file errors [80](#)  
function coverage [83](#)  
function defined [96](#)

## I

ignoring covered files [31](#)  
instrumenting and compiling source code  
    description [35](#)  
    from the command-line [37](#)  
    from Workbench [36](#)  
    from your makefile [37](#)  
    in the example session [24](#)  
instrumentor command line [38](#)  
instrumentor errors [76, 79](#)  
Instrumentor Options  
    Coverage Types tab view [29](#)  
    Covered Files tab view [31](#)  
    description field [34](#)  
    dialog box [28](#)  
    other options [34](#)  
    Other Options tab view [33](#)  
    verbosity [33](#)

## L

launching Code Coverage Analyzer [11](#)  
    example session [24](#)

## M

makefile modifications [37](#)  
messages  
    compiler errors [79](#)  
    error [76](#)  
    file errors [80](#)  
    instrumentor errors [76, 79](#)  
    status [76](#)  
    warning (command-line interface) [77](#)  
modifying your makefile [37](#)

## O

options  
    command-line [38](#)  
    Instrumentor [33](#)  
other options, Instrumentor Options dialog box [34](#)  
overview  
    architectural [4](#)

## P

projects  
    create new, in Workbench [10](#)

## S

scripts, CLI command examples [72](#)  
Select Target Server dialog box [12](#)  
selecting  
    Coverage Types tab view [29](#)  
    Covered Files tab view [31](#)  
    Other Options tab view [33](#)

- source code
  - compiling [35](#)
  - instrumenting [35](#)
  - viewer [18](#)
- specifying instrumentation parameters [28](#)
- start
  - data collection, example session [25](#)
- status messages [76](#)
- stop data collection, example session [25](#)

## T

- target connection
  - lost, *see* troubleshooting
  - verbosity [13](#)
- Trend Graph view [19](#)
- troubleshooting
  - Code Coverage Analyzer GUI [79](#)
  - guide [79](#)
  - target connection lost [79](#)
  - target server [79](#)

## U

- UNIX
  - displaying report in a browser [58](#)

## V

- verbosity
  - defined [97](#)
  - effect on Console view [21](#)
  - instrumentor [33](#)
  - instrumentor command-line [38](#)
  - target connection [13](#)
  - Warning [13](#)
- views
  - Console [20](#)
  - Coverage Data Colors [21](#)
  - Coverage Summary [17](#)
  - displaying report in a browser [58](#)

- Distribution Graph [19](#)
- Source code [18](#)
- Trend Graph [19](#)

## W

- warning
  - messages [77](#)
  - target verbosity [13](#)