HP Code Coverage Tool Reference Manual for HP Integrity NonStop NS-Series Servers

Abstract

This manual describes the Code Coverage Tool for HP Integrity NonStop NS-Series servers. It addresses application developers who use C/C++, pTAL, or COBOL to create application components for NonStop servers, and who wish either to evaluate the code coverage provided by test cases or to understand what parts of an application are used, or most heavily used, under a representative workload.

Product Version

Code Coverage Tool H06

Supported Release Version Updates (RVUs)

This publication supports H06.07 and all subsequent H-series RVUs until otherwise indicated by its replacement publication.

Part Number	Published
542684-001	August 2006

Document History

Part Number	Product Version	Published
542684-001	Code Coverage Tool H06	August 2006

HP Code Coverage Tool Reference Manual for HP Integrity NonStop NS-Series Servers

Glossary | Index | Figures | Tables

4. Running the Application

5. Converting Raw Data Files to DPI Files

6. Running the Code Cover Utility

What's New in This Manual

Manual Information

Abstract

This manual describes the Code Coverage Tool for HP Integrity NonStop NS-Series servers. It addresses application developers who use C/C++, pTAL, or COBOL to create application components for NonStop servers, and who wish either to evaluate the code coverage provided by test cases or to understand what parts of an application are used, or most heavily used, under a representative workload.

Product Version

Code Coverage Tool H06

Supported Release Version Updates (RVUs)

This publication supports H06.07 and all subsequent H-series RVUs until otherwise indicated by its replacement publication.

Part Number	Published
542684-001	August 2006

Document History

Part Number	Product Version	Published
542684-001	Code Coverage Tool H06	August 2006

New and Changed Information

This is a new manual.

About This Manual

Notation Conventions

Hypertext Links

Blue underline is used to indicate a hypertext link within text. By clicking a passage of text with a blue underline, you are taken to the location described. For example:

> This requirement is described under Backup DAM Volumes and Physical Disk Drives on page 3-2.

General Syntax Notation

This list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS. Uppercase letters indicate Guardian keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

```
file-name
```

computer type. `Computer type` letters within text indicate C and Open System Services (OSS) keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
myfile.c
```

italic computer type. *`Italic computer type`* letters within text indicate C and Open System Services (OSS) variable items that you supply. Items not enclosed in brackets are required. For example:

```
pathname
```

[ ] Brackets. Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$terminal-name
```

```
INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list can be arranged either vertically, with aligned brackets on

each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
FC [ num  ]
   [ -num ]
   [ text ]

K [ X | D ] address
```

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }

ALLOWSU { ON | OFF }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

… Ellipsis. An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address [ , new-value ]...

[ - ] {0|1|2|3|4|5|6|7|8|9}...
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char..."
```

Punctuation. Parentheses, commas, semicolons, and other symbols not previously described must be typed as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;

LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must type as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing. Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In this example, no spaces are permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing. If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] LINE

   [ , attribute-spec ]...
```

!i and !o. In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT (  segment-id                    !i
                         , error        ) ;              !o
```

!i,o. In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```
error := COMPRESSEDIT ( filenum ) ;                      !i,o
```

!i:i. In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```
error := FILENAME_COMPARE_ (  filename1:length           !i:i
                            , filename2:length ) ;       !i:i
```

!o:i. In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```
error := FILE_GETINFO_ (  filenum                        !i
                        , [ filename:maxlen ] ) ;        !o:i
```

Notation for Messages

This list summarizes the notation conventions for the presentation of displayed messages in this manual.

Bold Text. Bold text in an example indicates user input typed at the terminal. For example:

```
ENTER RUN CODE
?123
CODE RECEIVED:      123.00
```

The user must press the Return key after typing the input.

Nonitalic text. Nonitalic letters, numbers, and punctuation indicate text that is displayed or returned exactly as shown. For example:

```
Backup Up.
```

lowercase italic letters. Lowercase italic letters indicate variable items whose values are displayed or returned. For example:

p-register

process-name

[ ] Brackets. Brackets enclose items that are sometimes, but not always, displayed. For example:

```
Event number = number [ Subject = first-subject-value ]
```

A group of items enclosed in brackets is a list of all possible items that can be displayed, of which one or none might actually be displayed. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
proc-name trapped [ in SQL | in SQL file system ]
```

{ } Braces. A group of items enclosed in braces is a list of all possible items that can be displayed, of which one is actually displayed. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
obj-type obj-name state changed to state, caused by
{ Object | Operator | Service }
```

```
process-name State changed from old-objstate to objstate
{ Operator Request. }
{ Unknown.          }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
Transfer status: { OK | Failed }
```

% Percent Sign. A percent sign precedes a number that is not in decimal notation. The % notation precedes an octal number. The %B notation precedes a binary number. The %H notation precedes a hexadecimal number. For example:

```
%005400
```

```
%B101111
```

```
%H2F
```

```
P=%p-register E=%e-register
```

Change Bar Notation

Change bars are used to indicate substantive differences between this manual and its preceding version. Change bars are vertical rules placed in the right margin of changed portions of text, figures, tables, examples, and so on. Change bars highlight new or revised information. For example:

The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).

The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

1 Introduction to the HP Code Coverage Tool

This manual describes the **Code Coverage Tool** for HP Integrity NonStop NS-series servers. It addresses application developers who use C/C++, pTAL, or COBOL to create application components for NonStop Servers, and who wish either to evaluate the code coverage provided by test cases or to understand what parts of an application are used, or most heavily used, under a representative workload.

Features of the Tool

The code generator used by COBOL, pTAL, and C/C++ compilers on NS-series servers now has the capability to create **instrumented** object files. Such object files contain extra code that records which functions and blocks are executed, and how many times each is executed. The Code Coverage Tool uses this information to produce a report indicating what code in a program file or DLL was actually executed during one or more invocations. The **code coverage report** is a set of HTML files that you can view with any standard HTML browser.

Note. The Code Coverage Tool is intended for data generation and collection in a test environment only. The use of instrumented code is not recommended for production environments. Applications compiled with code coverage instrumentation will experience greatly reduced performance.

No source code changes are needed to instrument an application. The only required changes are in the commands used to compile and link the application. If you choose to instrument only a subset of your application, you specify code-coverage compiler options for only that subset of your source files.

Compilation can occur on the Guardian, NonStop Open System Services (OSS), or Windows platform. Execution must occur on either the Guardian or the OSS platform. The **Profile Merge Utility (profmrg)** and the **Code Cover Utility (codecov)**, used to assemble and present the code coverage data, run on the Windows platform.

You can instrument all or part of any type of application, for instance:

- OSS processes
- Guardian processes
- Active and passive process pairs
- Mixed language processes
- Processes with embedded NonStop SQL/MP or NonStop SQL/MX

Required Hardware and Software

The Code Coverage Tool includes two products

- T0746: Code Cover Utility (codecov)
- T0747: Profile Merge Utility (profmrg)

and support for code-coverage instrumentation in H06.07 and later versions of the following compilers:

- T0356: COBOL compiler (ECOBOL)
- T0549: C/C++ compiler for Guardian (CCOMP, CPPCOMP)
- T8164: C/C++ compiler for OSS (c89)
- T0561: pTAL compiler (EpTAL)
- T1246: Compiler backend

The codecov and profmrg products and compatible compilers are provided on the site update tape (SUT). To use codecov and profmrg, you must first install them on a Windows workstation, as described in Section 2, Installing the Code Coverage Tool.

Usage Overview

To measure code coverage for an application:

1. Compile application components on the NonStop server or a workstation. For parts of the application that you wish to instrument for code coverage, use compiler options described in Section 3, Building the Application.

   The compiler generates an instrumented binary file and a **static profiling information (SPI) file**.

2. Run the instrumented application on the NonStop server.

   The instrumented application creates a **raw data file**. Each time you run the instrumented application, a unique raw data file is created, either in the current directory for an OSS application or in the current subvolume for a Guardian application.

3. Run profmrg on a Windows workstation to merge all the raw data files into one **dynamic profiling information (DPI)** file.

   The profmrg utility consolidates all runs and therefore all raw data pertaining to code coverage for the application. It creates the DPI file, required as input to codecov.

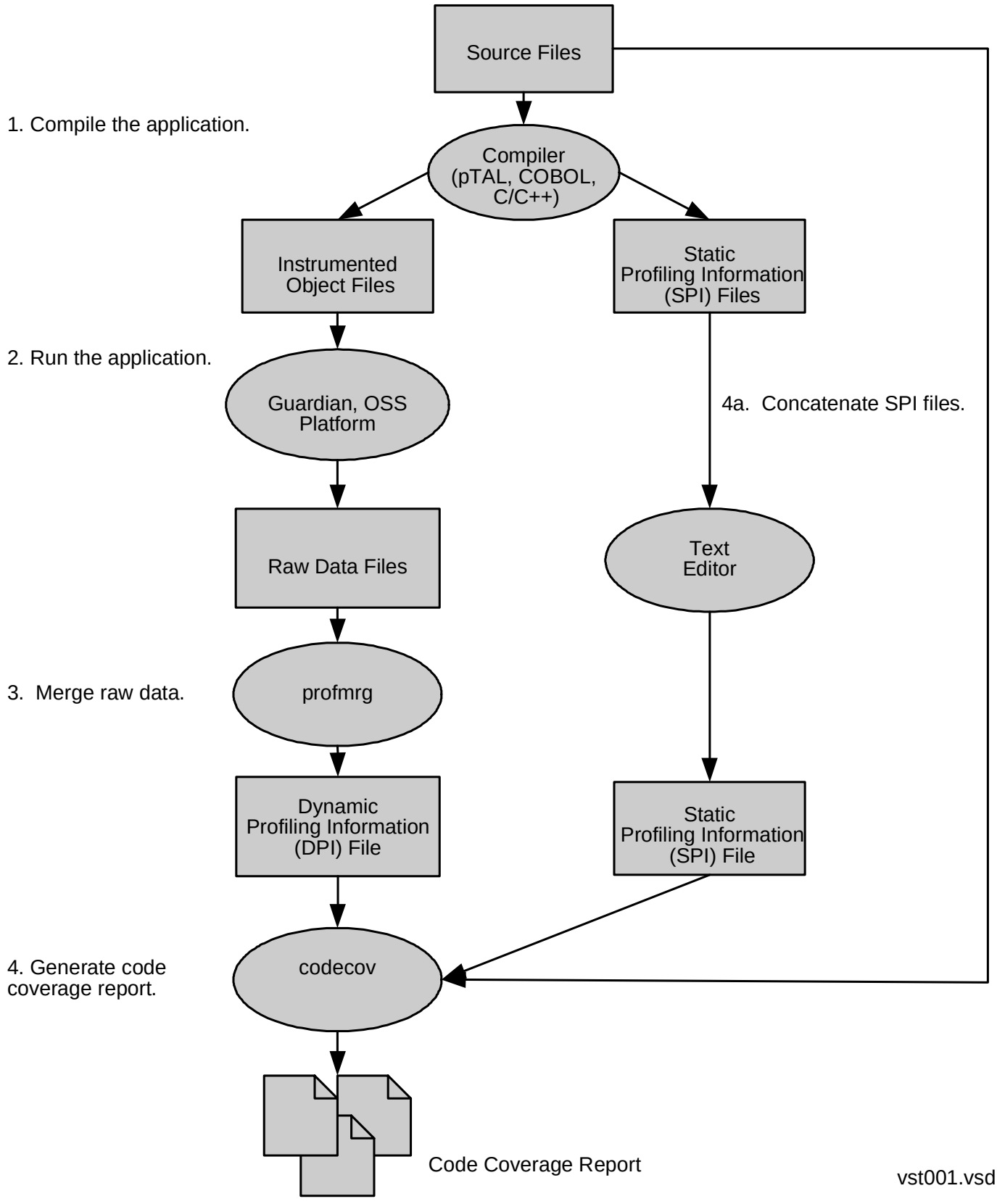
4. Run codecov on a Windows workstation to produce the code-coverage report.

   The codecov utility uses the SPI file, the DPI file, and the original source files to create a report that you can view with any HTML browser.

5. Evaluate the report, and take any appropriate action.

Figure 1-1 illustrates these steps.

Figure 1-1. Code Coverage Task Overview



Topics This Manual Covers

The rest of this manual provides

- Instructions for installing the Code Coverage Tool on a workstation
- Instructions for creating instrumented object files and SPI files from source programs in C/C++, pTAL, or COBOL
- Instructions for ensuring that a run of the application has generated the expected raw data files
- Instructions for using the profmrg utility to create DPI files from raw data files
- Instructions for running the codecov utility on a workstation
- A description of the report generated by the codecov utility
- Usage scenarios and considerations

2

Installing the Code Coverage Tool

The Code Coverage Tool consists of two products

- T0746: Code Cover Utility (codecov)
- T0747: Profile Merge Utility (profmrg)

and support for code-coverage instrumentation in H06.07 and later versions of the following compiler products:

- T0356: COBOL compiler (ECOBOL)
- T0549: C/C++ compiler for Guardian (CCOMP, CPPCOMP)
- T8164: C/C++ compiler for OSS (c89)
- T0561: pTAL compiler (EpTAL)
- T1246: Compiler backend

The Code Coverage Utility, the Profile Merge Utility, and the compilers are on the site update tape (SUT). The compilers are also delivered on CDs.

Copying the Code Coverage Tool to Your Workstation

The codecov and profmrg programs run on a Windows workstation but are delivered on a SUT. To install these programs on your workstation, proceed as follows:

Installing codecov

1. After installing the H06.07 or later RVU, locate the file $SYSTEM.ZCODECOV.T0746SET.
2. Copy T0746SET to your workstation.
3. Change the name of T0746SET to `setup.exe.`
4. Run `setup.exe`.

   A file named `Hewlett-Packard\CodeCoverage\codecov.exe` is created within the `program files` folder. `codecov.exe` is the executable codecov program.

Note. You might wish to put the location of `codecov.exe` in your `PATH` variable.

Installing profmrg

1. After installing the H06.07 or later RVU, locate the file $SYSTEM.ZCPS.T0747SET.

2. Copy T0747SET to your workstation.
3. Change the name of T0747SET to `setup.exe.`
4. Run `setup.exe`.

   A file named `Hewlett-Packard\CodeCoverage\profmrg.exe` is created within the `program files` folder. `profmrg.exe` is the executable codecov program.

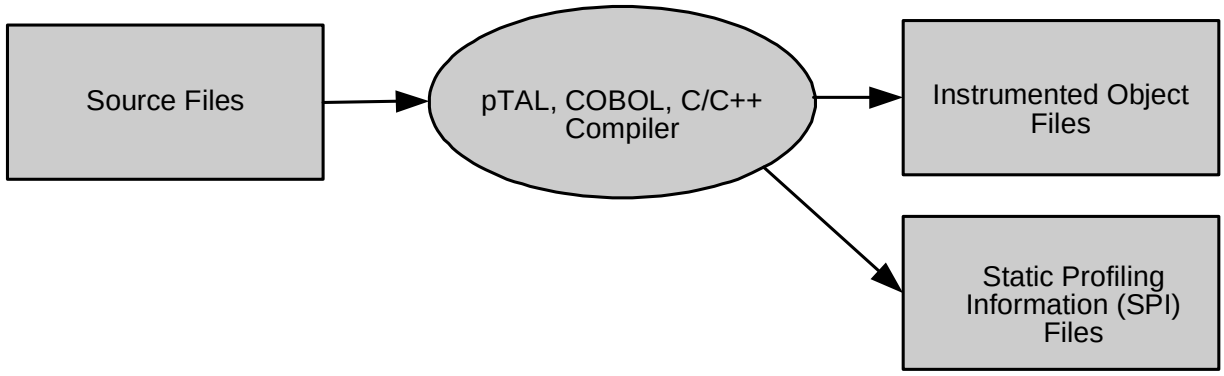
Note. You might wish to put the location of `profmrg.exe` in your `PATH` variable.

3 Building the Application

Task Overview

The first step in measuring code coverage is to generate instrumented object files for all parts of the application that you want to measure. To do this, you must compile and link the source files, specifying compiler and possibly linker options to support code coverage analysis. In addition to instrumented object files, the compiler creates SPI files (which you will later combine) for subsequent input to the codecov utility.

Figure 3-1. Compiling the Application to Generate Instrumented Object Files



vst003.vsd

Prepare to Compile

Selecting Source Files

Locate all the files you want to instrument. Some reasons you might want to measure only a subset of the application are that

- You have already studied some components and now want to study others. For example, you might want to study components recently added to the application.
- The application is large and you want to economize on data space and other resources.
- Only certain source files belong to you.

You can instrument source files written in different languages and compiled on different platforms as part of the same application.

Understanding Code Coverage Concepts

Code coverage reports are based on a model in which the application consists of source files, which contain functions, which contain **basic blocks**. A basic block is a sequence of code that is entered only at the beginning and exited at the end. Once code enters a basic block, it executes the entire block unless an exception is raised within the block. The Code Coverage Tool considers a basic block to be **covered** if it is entered, and also counts how many times such a block is entered during a test run. Procedure calls within otherwise straight-line code are also considered block boundaries. Therefore, the report tells when a procedure did not return to its caller, because the block after the procedure call is uncovered, or has a smaller count than the block leading up to the procedure call.

It is possible for multiple basic blocks to have the same source code location, with some blocks covered and others not. The Code Coverage Tool considers such locations to be **partially covered**. For example, if overflow_traps is enabled, and your source code contains a statement with an expression that could overflow when executed (but never does), then that statement would be considered partially covered because the basic blocks generated to handle the overflow for the statement are never executed even though the expression for the statement is executed.

Certain sequences of straight-line code can result in multiple blocks; for example, `if` statements can result in multiple blocks. In some cases, multiple blocks can result from code generated by the compiler (and thus, not visible at the source level), such as a check for overflow.

Using a different version of the compiler or running the compiler with different options, such as different optimization levels, can result in different blocks being generated from the same source code. For example, if the compiler determines that a block of source code is never executed, it might not generate any object code for that source code.

Usage Considerations

For important information about characteristics and limitations of instrumented program files and dynamic-link libraries (DLLs), see Compilation Issues on page 9-1.

Cleaning Up From Previous Runs

The first time you compile a program with the code-coverage option, the compiler creates a SPI file. This SPI file is one of the input files to the codecov tool described in Section 6, Running the Code Cover Utility. If compilation occurs in an OSS directory or in a Windows folder, the name for the file is pgopti.spi. If compilation occurs in a Guardian subvolume, the name for the file is pgospi. Subsequent compilations using the codecov option within the same directory, folder, or subvolume update or add information to this file.

To create a new SPI file instead of adding to an existing one, you can:

- Move or rename the old file.

- Compile the application in a different directory, folder, or subvolume.

Build the Application

Compiling the Source Files

Table 3-1 shows, for each supported platform and compiler, the options you specify to achieve code coverage.

Table 3-1. Compiler Options Related to Code Coverage

Platform	Compiler(s)	Option to Generate Instrumented Object File and SPI File
Guardian	CCOMP, CPPCOMP, ECOBOL, EPTAL	`CODECOV`
OSS, Windows	c89, ecobol	`-Wcodecov`
Windows	eptal	`-codecov`

CODECOV (or codecov) Option

This option directs the compiler to create an instrumented object file and to create or add to an existing SPI file.

Verify the Output

After compiling your program, verify that a new object file and a new or modified SPI file exist in the compilation location:

- Naming conventions for the instrumented object file are the same as for uninstrumented object files, so check the timestamp to verify that a new object file was created in the OSS directory, Windows folder, or Guardian subvolume.
- On the NonStop server, the SPI file is a type 180 text file. On OSS or Windows, it has the extension .spi. Check for a file with the name pgopti.spi in the OSS or Windows environment, or the name pgospi in the Guardian environment.

Linking the Object Files

To link instrumented object files, you must specify the option `-l pgo` if invoking the linker directly. If the linker is invoked through the compiler and you've specified the `CODECOV` or `codecov` option, the compiler automatically inserts the `-l pgo` option. (EpTAL does not support invoking the linker through the compiler.)

HP Enterprise Toolkit (ETK) Considerations

For each source file that is to generate instrumented object code at compile time, you must add the appropriate compiler driver flags to the Additional box on the compiler's General page. You can add the information at the project level, in which case it applies to all source files in the project, or at the file level to select a subset of files to be instrumented. Enter the flags exactly as you would enter them on the command line invoking a compiler driver.

In addition, you must specify the dynamic-link library zpgodll as input at link time. Enter `pgo` in the Dynamic and Static Libraries text box on the Linker Input property page. If the box already specifies other libraries, append `pgo` to the list, using a semicolon (;) as the separator.

Example

See Build the Application on page 8-3.

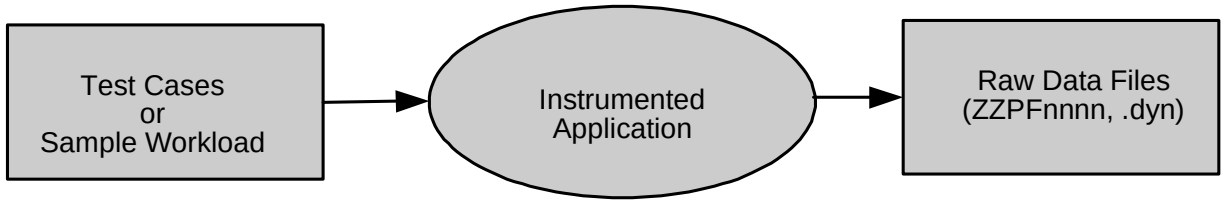
4 Running the Application

Task Overview

In this step, you run the application with test data or a representative workload. A program or DLL containing instrumented code generates a distinct **raw data file** each time it completes execution. This raw data file is the record of what functions and basic blocks were executed during the test run.

If an application is built by linking many source files and DLLs, the run generates one raw data file that contains code coverage data from all instrumented components.

Figure 4-1. Producing the Raw Data for Code Coverage Analysis



vst004.vsd

Prepare the Application for Testing

Preparing Test Cases

Select a set of test cases or a sample workload to use for your study.

Do not run your test in a production environment, because instrumented code runs more slowly than uninstrumented code, and users could experience degradation in throughput or response time.

Insulating the Application

The raw data file is placed in the default subvolume (or current working directory) when the program or DLL is initiated. To keep the raw data files for different runs and different applications separate:

- Run each instrumented application in a different subvolume or working directory.
- Archive raw data files that represent earlier versions of the source code. Remove such files from the default subvolume or current working directory.
- Use a *different* subvolume or working directory for different runs of the same source code if you want to compare the code coverage provided by the various

runs. Use the *same* subvolume or working directory for different runs of the same source code if you want to measure the code coverage provided by all runs combined.

An existing raw data file is never overwritten. If a raw data file already exists in the subvolume or directory, the instrumented program will create an additional file with a different name.

Run the Application

Run the application. An instrumented application will run more slowly than it would without instrumentation.

To produce a raw data file from an application that does not ordinarily terminate, stop the application manually.

If any errors occur during execution of the instrumented application, they are written to a file named ZZPELOG. After execution, you should check for the existence of this file (it is created only if errors occurred). It can be useful to HP support personnel, should you require their assistance in determining the cause of the errors.

Verify Output

Check the default subvolume or current working directory for one or more files with names of the form ZZPF* in the Guardian environment or the extension .dyn in the OSS environment.

On the NonStop server, a raw data file is a type 180 binary file.

Example

See Run the Application on page 8-4.

5

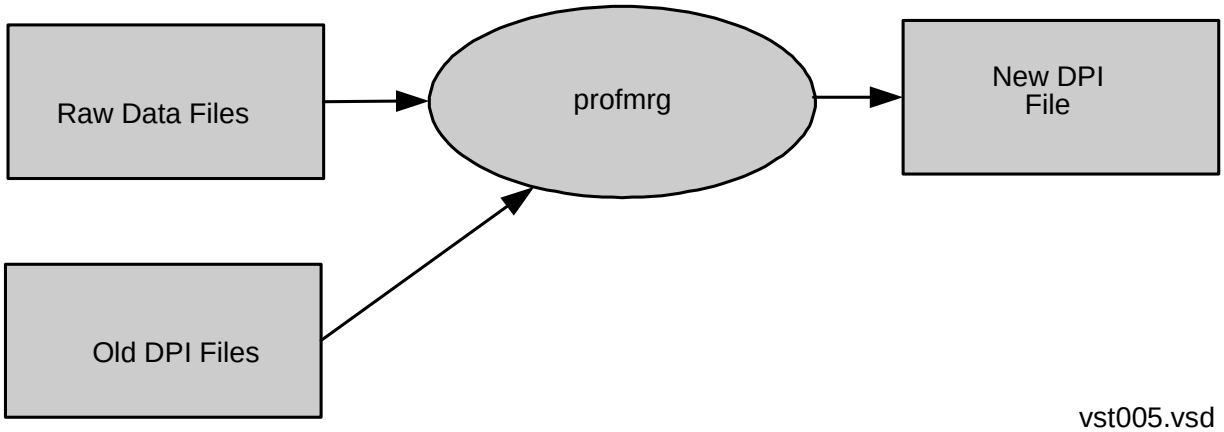
Converting Raw Data Files to DPI Files

Task Overview

In this step, you combine the raw data files from application runs to produce a dynamic profiling information (DPI) file for input to the codecov utility. If you have previously measured code coverage for the same application, you can combine the DPI files from the previous runs with raw data files from the latest run to produce a new DPI file.

To perform this task, you use the profmrg utility, which runs on a Windows workstation.

Figure 5-1. Using profmrg to Combine Raw Data Files



Prepare profmrg Input Files

The profmrg utility requires the following inputs:

- Zero or more raw data files, generated by runs of instrumented code
- Zero or more DPI files, created by previous runs of profmrg

The output of profmrg is a DPI file that combines the information from all the input files.

Although you normally use profmrg to combine new data and produce a new DPI file, it can sometimes make sense to run profmrg with no input raw data files and no DPI files, other than an existing DPI file that is also the output file. For example, the `-dump` option, described in Run profmrg, produces a text dump of the input files.

If you are running profmrg solely to update or dump an existing DPI file, you need not provide any other input.

The meaningfulness of the code coverage report depends on the consistency of the SPI file, the DPI file, and the source files provided as input. Specifically, for the results of a study to be meaningful, all inputs should reflect the same versions of the source files. The profmrg utility performs no validation in this regard.

Assembling Raw Data Files

Names of raw data files have one of two possible forms:

- ZZPF* if the program was run in the Guardian environment, or if the program was run in the OSS environment and the current directory is a Guardian subvolume
- *.dyn if the program was run in the OSS environment and the current directory is an OSS directory

To use raw data files as input to profmrg, you must first use FTP or some other mechanism to copy the files from the NonStop server to the workstation where profmrg will run.

Saving DPI Files from Previous Runs

If you want to save the file generated by a previous version of profmrg, to prevent the tool from replacing the existing file, you can:

- Move the existing file to a different location, or rename it. (The default name for the DPI file is pgopti.dpi.)
- Use the `-prof_dpi` runtime option, described in Run profmrg, to specify a name for the new DPI file

If a file of the same name as the output file already exists, profmrg displays a warning message on the standard error file and replaces the file. If the file is specified for the `-a` option, it is used as an input DPI file.

For additional considerations, see The Output DPI File.

Including DPI Files from Previous Runs

You can submit DPI files from previous profmrg runs, to base your code coverage analysis on cumulative data. The `-a` runtime option, described in Run profmrg, lets you specify the set of DPI files to use as input.

Note. If you plan to use the `-a` option to specify the set of DPI files to use, be sure to verify in advance that each of those files exists.

Do not use files from previous runs if you've changed the source code since those runs. The profmrg tool does not verify that its input files reflect the same source code versions.

Where to Put the Input Files

The profmrg utility looks for all the input files and creates the output file relative to some folder. By default, profmrg uses the current folder, but you can change the location with the `-prof_dir` option of the profmrg command, as described in Run profmrg.

The profmrg utility automatically uses, as input, any file whose name has the form ZZPF* or *.dyn. To exclude a file from processing, rename it or remove it from the folder profmrg will use.

Run profmrg

To run profmrg, use a command line of the form:

```
profmrg options
```

where *options* can be any set of options from the following list. Options must be in lowercase.

Note. A filename specified in a profmrg command option must not begin with a slash(/).

`-a` *dpi_file_list*

specifies the DPI files to use as input. The list can consist of any number of file names, separated by spaces. There is no rule for the format of a filename; for example, it need not contain the string .dpi. If the `-prof_dir` option is also present in the command, then each name specified for the `-a` option is concatenated with the name given in the `-prof_dir` option.

The `-a` option must be the last option on the command line. Any options specified after it are ignored.

`-dump`

produces a text dump of the contents of the input files. The profmrg utility writes the dump to the standard output file (stdout). You can redirect it to any convenient location.

The `-dump` option produces text output even if profmrg does not re-create the DPI file. (For a discussion of cases in which profmrg does not re-create the DPI file, see The Output DPI File.) Therefore, this option is useful for gathering information about an existing DPI file.

`-help`

displays brief descriptions of syntax options.

Do not specify any other options on the command line. When you specify the `-help` option, any other options on the command line are either ignored or produce command syntax error messages.

`-nologo`

suppresses the banner that profmrg would otherwise display.

`-prof_dir` *`directory`*

specifies the name of a folder relative to which profmrg looks for its input files and creates its output file. By default, profmrg uses the current folder. Filenames specified in the `-prof_dpi` and `-a` options are concatenated with the directory name specified in this option.

`-prof_dpi` *`filename`*

specifies the name of the DPI file that profmrg creates, overriding the default name pgopti.dpi. There is no rule for the format of the filename; for example, it need not contain the string .dpi. If the `-prof_dir` option is also present in the command, the filename is concatenated with the name given in the `-prof_dir` option.

Verify Output

The desired output from profmrg is a DPI file. Verify that such a file has been produced. Also, check the standard error file for status messages and the standard output file if you requested a dump with the `-dump` option.

The Output DPI File

The profmrg utility creates an output DPI file whose name is pgopti.dpi, unless you specify a different name with the `-prof_dpi` option.

Usually, if profmrg finds an existing DPI file with the same name as the output file, it re-creates the DPI file, on the assumption that doing so will result in the latest usage information. However, in certain cases, profmrg decides that an existing DPI File would have the same content if re-created; in such cases, profmrg leaves the existing file alone. Specifically, this behavior arises if the existing DPI file is newer than the input raw data files, and no other DPI files are specified as input. In other words, profmrg assumes that the existing DPI file was created from the same set of input files and is not just some other file that happened to be copied to the same location.

On the other hand, if you specify the `-a` option, profmrg does re-create the output DPI file, even if the existing file was newer than all the raw data files and DPI files provided as input. In other words, profmrg does not assume that DPI files specified with `-a` are already reflected in an existing DPI file in the input folder.

profmrg Use of Standard Error and Output Files

The profmrg utility writes all its output to the standard error file (stderr), except that output from the `-dump` option is written to standard output (stdout).

If profmrg does not recognize an option on the command line, it displays a message to that effect and a brief syntax message. Similarly, if an option that requires a parameter is the last token on the command line, profmrg displays an error message and a brief syntax message.

Following any such messages is a banner consisting of one line that displays the name and VPROC of the tool, and a second line stating the copyright. For example:

```
profmrg - T0747
Copyright 2006 Hewlett-Packard Company.
```

If a raw data file has the wrong format--for instance, if some other type of file has a name that resembles a DPI file name--profmrg writes a warning message to the standard error file and ignores the invalid input file. A warning message also results if a DPI file specified with the `-a` option is invalid.

Example

See Produce the DPI File on page 8-4.

6 Running the Code Cover Utility

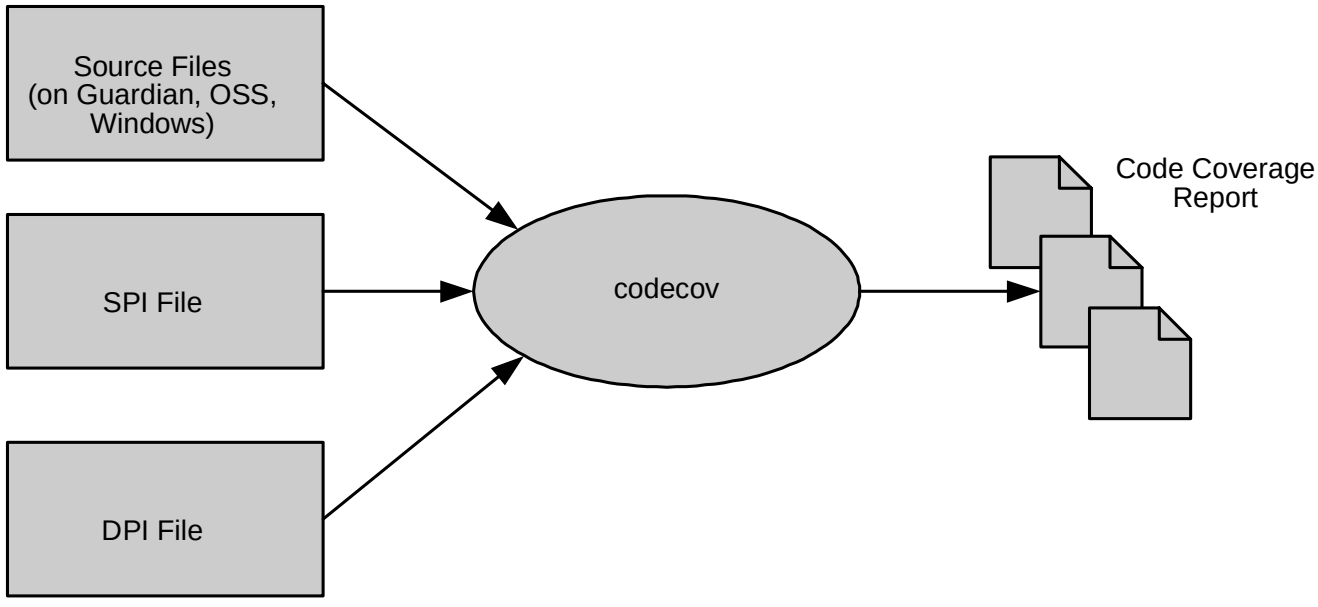
Task Overview

In this step, you run the Code Cover Utility (codecov), making available as input:

- The source files you compiled, as described in Section 3, Building the Application
- The SPI file you created, as described in Section 3, Building the Application
- The DPI file you created, as described in Section 5, Converting Raw Data Files to DPI Files

The Code Coverage Tool generates the code coverage report in a set of HTML files you can view with any standard browser. The report includes color-coded, annotated, source code listings that distinguish among various kinds of covered and uncovered code. Command options let you specify which colors are used for which purposes. For a full discussion of the report, see Section 7, Interpreting the Code Coverage Report.

Figure 6-1. Running the Code Cover Utility to Create the Code Coverage Report



Prepare to Run codecov

The meaningfulness of the code coverage report depends on the consistency of the SPI file, the DPI file, and the source files provided as input. Specifically, for the results of a study to be meaningful, all inputs should reflect the same versions of the source files. The codecov utility performs no validation in this regard.

Providing for Source File Retrieval

You need not specify source file names explicitly to the codecov utility, because the files are listed in the SPI and DPI files. You must, however, ensure that the files are in the locations reflected in the SPI and DPI files and are accessible on disk or over the network.

Part of the function of codecov is to retrieve and examine the source files to be included in the analysis, wherever those source files happen to be. The filename listed in a SPI or DPI file indicates to codecov whether the source file is on the Guardian, OSS, or Windows platform. If a source file name found in the SPI file is a Guardian or OSS file name, codecov uses an FTP session to fetch a copy of the source file to the workstation. To enable this behavior, you specify the `-host` and `-login` options on the codecov command line. You may also specify the `-passwd` option to provide the password in advance; otherwise, you will be prompted (on the standard output file) to type the password (on the standard input file.) The password must be the one associated with the name specified in the `-login` option. For security reasons, codecov does not echo the password when you type it.

You can have a mix of source files:

- On different NonStop servers in the same network
- In the Guardian or OSS environment
- On the workstation where codecov is running

However, all source files fetched from the NonStop server must be in the same network as the machine specified with the `-host` option. Also, Guardian source files must be of code 101 or 180. Be sure to set up any required permissions or remote passwords to allow codecov access to your source files.

If codecov is unable to fetch a particular source file, it writes a warning message, and the code coverage report excludes data about that file.

Preparing the SPI File

The codecov utility requires a single SPI file. The default name for the SPI file is:

- pgopti.spi if the program was compiled on Windows
- pgopti.spi if the program was compiled in the OSS environment and the current directory is an OSS directory
- pgospi if the program was compiled in the Guardian environment
- pgospi if the program was compiled in the OSS environment and the current directory is a Guardian subvolume

You can use the `-spi` option to specify a different name. codecov always looks for the SPI file in the current folder; move the file from the compilation folder to the workstation, renaming it if necessary.

The SPI file is a text file that contains information about the various source files. For each source file, the file includes information about the functions in that source file.If your application is compiled in multiple directories or subvolumes and therefore more than one SPI file exists for it, you must manually concatenate the files into one SPI file for input to codecov. The codecov utility ignores any duplicate information.

Creating a single SPI file from multiple SPI files is not complex. SPI files are text files, and all you need to do is concatenate them. However, when a SPI file is created on the Guardian platform, it is somewhat cumbersome to edit because it is a code 180 file and, in general, cannot be converted to an edit file, because the lines are too long. For convenience, move SPI files to the OSS or Windows platform and concatenate them there.

Preparing the DPI File

Identify the latest DPI file produced as output from profmrg. The default name for the DPI file is pgopti.dpi; you can use the `-dpi` option to specify a different name. The codecov utility always looks for the DPI file in the current folder.

The DPI file contains the names of source files whose code was actually executed during the test run. If a given source file name occurs in the SPI file but not the DPI file, codecov deduces that none of the code in that source file was covered, and the code coverage report reflects that conclusion. Similarly, if a source file name occurs in the DPI file but not in the SPI file, codecov ignores the profile information about that file, because the report includes information only for source files listed in the SPI file.

These assumptions make it necessary for corresponding source file names in the SPI files and DPI files to match. They should automatically match, unless you moved or renamed them between the time you compiled them, producing a SPI file, and the time you ran them, producing a raw data file (from which profmrg produces the DPI file).

Cleaning Up from Previous Runs

The HTML files containing the code coverage report are created in the folder where you run codecov. (For information about how the files are organized, see The Code Coverage Report.)

New files overwrite existing files of the same names. If you have previously run codecov in the current folder, archive the old code coverage report before you continue.

Run codecov

To run codecov, use a command line of the form:

```
codecov options
```

where `options` can be any set of options from the following list. Options must be in lowercase.

`-bcolor` *`color`*

specifies the name or hexadecimal code of the HTML color used in reports to show uncovered basic blocks within a function for which some basic blocks were covered and some were not. The default value is #ffff99, which is yellow.

`-ccolor` *`color`*

specifies the name or hexadecimal code of the HTML color used in reports to show the basic blocks that were covered (that is, executed during the test run). The default value is #ffffff, which is white (no color).

`-counts`

causes execution counts to be included in the code coverage report. A basic block executed once has a count of 1, a basic block executed twice has a count of 2, and so on.

`-dpi` *`filename`*

specifies the name of the DPI file, overriding the default name pgopti.dpi.

`-fcolor` *`color`*

specifies the name or hexadecimal code of the HTML color used in reports to show functions that were **uncovered** (never called). The default is #ffcccc, which is pink.

`-h or -help`

causes codecov to stop processing the command line, print out a syntax description of all options that it supports, and terminate.

`-host` *`string`*

provides a host address to use for access to a NonStop server. The codecov utility uses the name when fetching source files from the NonStop server. The string could be a DNS name--for example, orgdiv.arn.acorp.com--or an IP address in dotted decimal format. You must specify this option if the application includes source files on NonStop servers.

`-login` *`string`*

provides a login name for access to a NonStop server. The codecov utility uses this name when fetching source files from the NonStop server. You must specify this option if the application includes source files on NonStop servers. The name must be a valid login name on the machine specified by the `-host` option.

`-maddr` *`email`*

specifies a destination for email sent from the code coverage report. The codecov utility places a link at the bottom of each screen of the report. When you click the link, a window for sending email appears, with the address specified by the `-maddr` option. If you omit this option, the mail is sent to *nobody*.

`-mname` *`message`*

specifies the text of the link used to invoke the mail window. If you omit the `-mname` option but include the `-maddr` option, the text of the link is the same as the address specified in the `-maddr` option. If neither option is present, so that `-maddr` defaults to *nobody*, then `-mname` defaults to *Nobody*.

`-nopartial`

specifies that, if multiple basic blocks are generated for a single source position, codecov should consider them all to be covered if any one of them was covered. In the report, such code appears in the color for covered code rather than partially covered code.

`-nopmeter`

suppresses the **progress meter**, which codecov would normally write to the standard output file during its operation. The progress meter reports the percentage of functions analyzed so far. For example, if a program had only four functions, codecov would print 25%, 50%, 75%, and finally 100%. If a program contains ten or more functions, codecov prints the percentage each time it completes analysis of one-tenth of the functions, so the progress meter is updated at most ten times. Progress messages appear on the same line unless interrupted by other messages, such as warning messages about source files that codecov cannot find.

`-passwd` *`string`*

specifies the password for the user name given in the `-login` option. If codecov must fetch source files from the NonStop server and the `-passwd` option is not present on the command line, codecov prompts for the password.

`-pcolor` *`color`*

specifies the name or hexadecimal code of the HTML color used in reports to show partially covered code. If the `-nopartial` option is present on the command line, the `-pcolor` option is meaningless. The default value is #fafad2, which is light brown.

`-prj` *`title`*

specifies a title to be included at the top of the top-level HTML file in the code coverage report. For example, if you specified the value `CallDistribution`, the full title printed at the top of the report would read “Coverage Summary of CallDistribution.” If you omit this option, the top-level HTML file bears the title Coverage Summary.

`-spi` *`filename`*

specifies the name of the SPI file, overriding the default name. The default name is:

- pgopti.spi if the program was compiled on Windows
- pgopti.spi if the program was compiled in the OSS environment and the current directory is an OSS directory
- pgospi if the program was compiled in the Guardian environment
- pgospi if the program was compiled in the OSS environment and the current directory is a Guardian subvolume

`-ucolor` *`color`*

specifies the name or hexadecimal code of the HTML color used in reports to show source for which no code was generated. Examples are comments, statements that include header files, and variable declarations. The default value is #ffffff, which is white (no color)

Verify Output

The primary output from codecov is a set of HTML files constituting the code coverage report. Verify the existence of those files, and check the standard output file for status messages

The Code Coverage Report

The code coverage report consists of a hierarchy of HTML files. The top-level file is named CODE_COVERAGE.HTML and is created in the same folder where codecov runs. All the other HTML files are in a subfolder named CodeCoverage. These files also have uppercase names with the extension .HTML. Any existing files of the same names are overwritten.

Among the HTML files that codecov creates are some that correspond to individual source files.

▲ **WARNING.** The relation between source file names and HTML file names is not guaranteed to be the same in future versions of this product.

Hyperlinks among the files within the CodeCoverage folder use simple pathnames. The top-level file, CODE_COVERAGE.HTML, uses hyperlinks to point to these other files. The hyperlinks have the format

```
Code_Coverage/filename.HTML
```

This convention makes it easy to move the entire set of files to another location, placing the top-level file in one directory or folder, and the other files in a subdirectory or subfolder named Code_Coverage. Windows, OSS, and UNIX all support this naming convention.

codecov Use of Standard Error and Output Files

The codecov utility sends all its output--messages, help syntax, password prompts, and so on--to the standard output file (stdout), except that it writes two banner lines to the standard error file (stderr). The banner consists of one line that displays the name and VPROC of the tool, and a second line stating the copyright.

Note. Observe that codecov uses the standard output file for status messages, whereas profmrg uses the standard error file.

When codecov processes the command line, it detects an error if an option requires a parameter of a certain form and the parameter is invalid. The utility also detects an error if an option that requires a parameter is the last token on the command line. In such cases, codecov emits an error message and terminates.

If you specify an option that codecov does not recognize, codecov ignores the option and continues processing the command. Therefore, if results are different from what you expect, check your command line for typographical errors.

If you specify the same option several times, with different parameters, the last instance on the command line supersedes earlier instances.

Example

See Measure Code Coverage on page 8-5.

7

Interpreting the Code Coverage Report

Opening the Code Coverage Report

To examine the code coverage report, use a browser to open the file named CODE_COVERAGE.HTML. It displays a top-level summary of the code coverage for all relevant source files, as in the following example:

Figure 7-1. Coverage Summary

Coverage Summary

Files				Functions				Blocks			
total	cvrd	uncvrd	cvrg%	total	cvrd	uncvrd	cvrg%	total	cvrd	uncvrd	cvrg%
5	3	2	60.00	103	12	91	11.65	3,681	252	3,429	6.85

Covered Files

Name	Functions			Blocks		
	total	cvrd	cvrg%	total	cvrd	cvrg%
QA24.$DATA05.TEST.ELFDUMPC	42	9	21.43	2,338	246	10.52
QA24.$DATA05.TEST.NSKELFH	3	1	33.33	9	4	44.44
QA24.$DATA05.TEST.PLATDEPH	43	2	4.65	192	2	1.04

Web-Page Owner: Nobody

Uncovered Files

Name	Functions	Blocks
	total	total
QA24.$DATA05.TEST.PLATDEPC	1	3
QA24.$DATA05.TEST.YOSELFC	14	1,139

Web-Page Owner: Nobody

The **coverage summary** indicates how many files, functions, and basic blocks were covered or uncovered. The number of files is the number that the report covers. It includes files that were listed in the SPI file and that codecov was able to find when it ran. When the report says that a file or function is covered, it means that at least some part of the file or function was covered, not necessarily that it was completely covered.

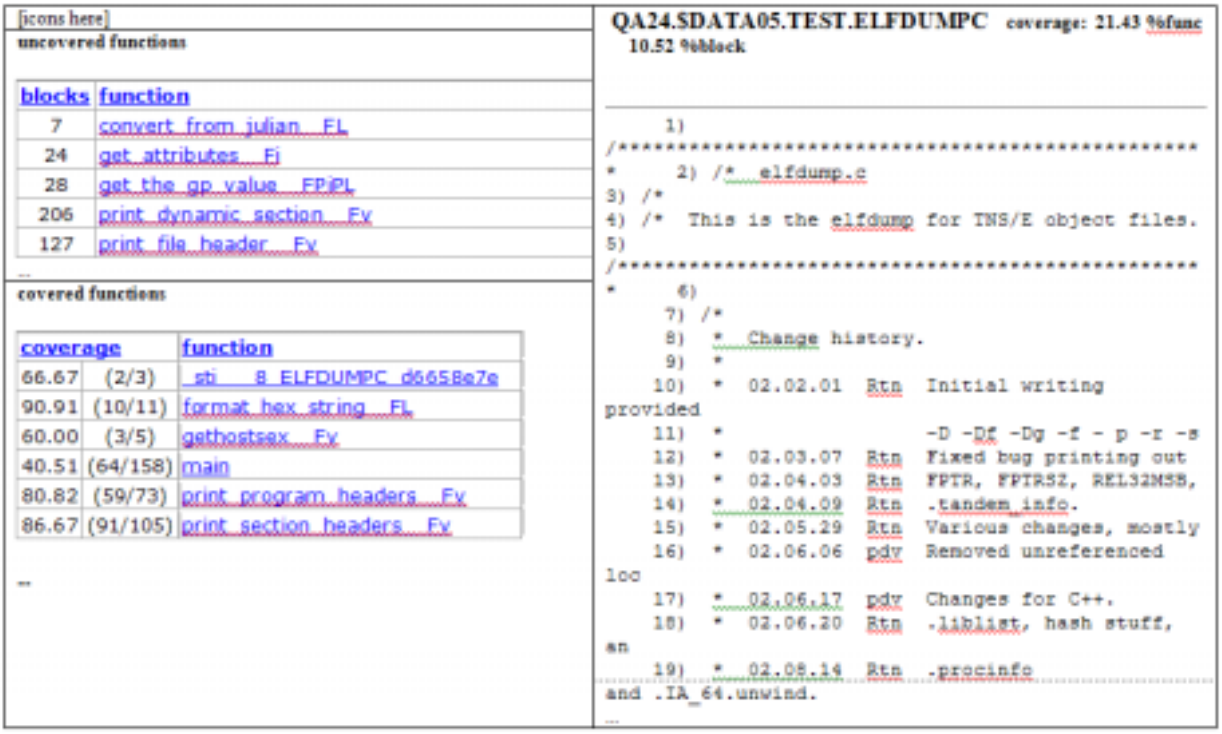
Below the coverage summary are lists of covered and uncovered files, with counts of covered or uncovered functions and basic blocks for each file.

When you click a filename in the list of covered or uncovered files, the browser displays information about that particular file.

Code Coverage Display for a Source File

The code coverage display for a source file is divided into left and right halves, as in the following example:

Figure 7-2. Coverage Display for a Source File



The left half lists the functions that were completely uncovered and the functions that were at least partially covered. For each function that was at least partially covered, the display tells how many basic blocks were or were not covered. In these lists, the names of C++ functions are mangled.

To sort the information by function name or coverage amount and switch between ascending and descending order, click the word **function** or **coverage** in the list of covered functions .

The right half of the display contains a color-coded source listing, as shown in Figure 7-2. This source listing is created by copying in the source file itself, so function names are not mangled here. Sequential line numbers (not edit line numbers) are shown alongside the source code. If you select a function name on the left side of the display, the listing on the right side scrolls until the start of that function is at the top of the window. You can use the usual scroll keys to move around this window or the ctrl-F key sequence to search for words.

The colors on this display are the ones you specified when you ran codecov, or the default values, as described in Run codecov on page 6-4.

Execution Counts in the Source Display

If you specify the `-counts` option in the codecov command (as described in Run codecov on page 6-4), a count of how many times a particular basic block was entered

appears under the code directly beneath the source position where the block begins. The count is immediately preceded by a circumflex (^). If more than one basic block is generated for the code at a source position, the number of generated basic blocks and the number of executed basic blocks follow the execution count. For example, if two basic blocks were generated at that place and only one was executed, you'll see 1/2 as part of the information presented, as in the following example:

Figure 7-3. Source Code Display Including Execution Counts

```
if ((n==1) || (==2))
^ 10(1/2)
```

Representation of #include Files

When an entire function is pulled into a compilation from an #include file, the SPI file lists that function under the name of the source file that contains the function, not under the name of the main file for the compilation. Therefore, it is possible to compile a single source file for code coverage and obtain a report that lists several source files: the compiled file and each file from which functions were imported with #include statements. Examples of program elements that might be included in this way are C++ templates and member functions whose bodies are in the class declarations.

In contrast, when an #include statement brings in only a part of some larger function, the SPI file does not include any information about the included source file (although the included source file could be listed in the SPI file for some other reason).

Understanding Color Coding in the Code Coverage Report

Table 7-1 summarizes the colors used in the code coverage report.

Table 7-1. Color Coding

Default Color	Meaning
yellow	Indicates uncovered basic blocks in a function for which some basic blocks were covered and others were not.
white (no color)	Indicates basic blocks that were covered.
pink	Indicates functions that were uncovered.
light brown	Indicates partially covered code.
white (no color)	Indicates source for which no code was generated.

In general, the recommended way to use color coding in reports is simply to get a general view of which code was covered (executed) and which code was not. You can then determine whether more testing is needed to cover those portions of the code that were not executed. For example, you might look for large pink or yellow areas, indicating uncovered portions of code, then develop additional tests to execute that code.

Getting more detailed information from color coding can be more difficult. To accurately interpret the colors that appear in a code coverage report, you should understand the basic concepts discussed under Understanding Code Coverage Concepts on page 3-2.

In general, all code within a block is given the same color. If the block was executed at least once, that block was “covered”, so it has no color (by default). If a block was never executed, it is either yellow or pink.

Different Colors Within the Same Block

However, there are exceptions to the generalization that all code within a block is given the same color. In some cases, the coloring changes in the middle of a line. This occurs because there might be several different blocks on the same line, and the compiler and codecov are not always precise in determining where the blocks begin and end. In some cases, a line, or part of a line, will inherit a color from the preceding block.

Thus, the colors in the report do not always convey the correct information. But you can better understand the report by observing where the colors change. The new color after the change might be correct for the entire line, or even for previous lines, even though those lines have a different color in the report.

As an example of how color coding can be misleading, consider the following code sequence. The example shows the color coding that would appear in the report.

```
void a() {
        return;
         c();
}
```

In this example, procedure a is called and has an unconditional return in the second line. Thus, the two lines following the return are uncovered, and you would expect them to be colored yellow. In fact, only the last line (the closing brace) is yellow. The compiler did not generate code for the call to `c`, so there are no blocks on that line. Accordingly, the call to `c` is not colored (the same as the two preceding lines), because codecov believed that the block containing the `return` statement also included the call to `c`, and gave it the same color. The report shows the call to `c` as covered even though it actually was not.

For another example, consider the following code sequence:

```
if (*p=='\001")
        answer = DATAL;
else
        answer = DATAM;
```

Now assume that the first assignment was never executed and the second assignment was executed. As expected, the report shows the first assignment as uncovered (yellow). However, the report also shows the `else` statement as yellow, even though it was executed. That happens because the compiler generated no blocks on the `else`, so that line “inherited” the color from the previous line.

These same considerations for individual lines also apply to entire functions. Consider a sequence consisting of two functions. Assume that the first function was completely uncovered (pink in the report) and that the second function has portions that were covered and other portions that were uncovered. You would expect the covered portions to have no color, and the uncovered portions to be yellow. However, some of the pink from the first function might appear at the beginning of the second function. In particular, if comments appear at the beginning of a function, those comments inherit the color of the previous function (if any) and not the color of the function to which they apply.

Using the -counts Option

Using the `-counts` option can help you understand where the basic blocks begin and end. The `-counts` option causes block execution counts to be included in the report.

The count, specifically, the caret (^), appears directly under the point where codecov "thinks" the block begins. For example:

```
int gethostsex (void) {
    ^ 2
```

This procedure was executed, so it should have been uncolored. However, the `int` at the beginning of the line is shown as partially covered (light brown).The report also indicates that the first block generated for the line starts with `gethost`, and was executed twice. You might expect that the entire line would be given the same color (no color). However, the color coding for this line shows the portion of the line starting with `gethost` as uncolored; `int` has a color inherited from previous lines.

As explained in Understanding Code Coverage Concepts on page 3-2, control flow structures, such as `if` and `return` statements, can result in multiple blocks. Moreover, multiple blocks can have the same source code location, with some blocks covered and others not. These source code locations are considered to be "partially covered." You can eliminate this potential source of confusion by specifying the `-nopartial` option, which specifies that if multiple basic blocks are generated for a single source position, codecov should consider them all to be covered if any one of them was covered.

When `-nopartial` is specified, you can still use the `-counts` option to obtain information about blocks. Consider the following example of a partial codecov output:

```
while (i < 14)
   ^28(3)
{
        Func(i);
          ^14
        ++9i;
}
```

The count of 14 below the call to `Func` indicates that the `while` loop was executed 14 times. Since that is the only point within the loop that shows a count, that means that the entire body of the loop generated a single block. However, the notation `28(3)` indicates that three other blocks were generated for the line containing the `while` statement, and collectively, those three blocks were executed 28 times and that each block was executed at least once. (When just a single value, such as (3), and not two values separated by a slash, such as (1/3), is shown in parentheses, that means that either all blocks were executed or none were executed.) You cannot determine from these values how many times each block was executed.

For another example, consider the following partial codecov output:

```
if (n==1)||(n++2)
^10(1/2)
```

In this example, the notation `(1/2)` indicates that two blocks were generated for the `if` statement, and only one of them was covered; the covered block was executed 10 times, and the other block was not executed.

Using the -ucolor Option

The `-ucolor` option specifies the color to use for source code for which no object code was generated. However, as shown in the preceding examples, codecov always assumes that a block ends just before the next block begins and, therefore, does not know which source code has no object code. In actual practice, the `-ucolor` option is useful only for lines at the beginning of each source file, preceding the first block.

.

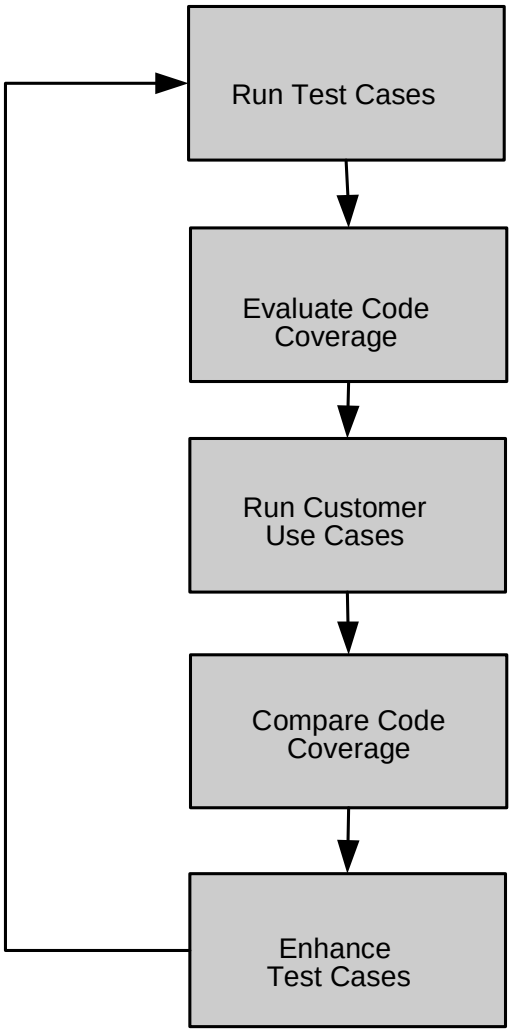
8 Usage Scenario

The most obvious application of code coverage technology is for a quality assurance group to determine what parts of a program were executed by some set of test runs. The goal, in this case, is for the test suite to be as comprehensive as possible, ensuring that the code has been well exercised before release to users.

In this scenario, one would typically rebuild the product, perform the test runs, and generate the reports in quick succession, probably not retaining intermediate files for very long, because the files become obsolete as soon as the source code changes. A quality assurance group might go through this process once a month.

Figure 8-1 illustrates a typical scenario in which a group develops a software product, establishes the code coverage provided by a set of test cases, runs an actual customer use case or sample workload, and uses the coverage as a basis for enhancing the test library.

Figure 8-1. Strategic Use of Code Coverage Technology



vst011.vsd

Build the Application

The application in this example is called elfdump. It is a program that dumps information about object files on a NonStop server.

1. This display shows the build directory for the elfdump application. The application requires three source files (which have names ending in C) and four header files (which have names ending in H). The application will be built and executed in the Guardian environment.

```
$DATA05.TEST 2> files


$DATA05.TEST


ELFCOMH   ELFDUMPC  NSKELFH   PLATDEPC  PLATDEPH  YOSELFC
YOSELFH
```

2. These command lines invoke the compiler and linker with options related to code coverage. The `codecov` option causes the compiler to create an instrumented object file. The generated SPI file will have the name PGOSPI. The `-lpgo` option is required to link instrumented object files.

```
cppcomp /in elfdumpc, out elfdumpl/ elfdumpo; codecov
cppcomp /in platdepc, out platdepl/ platdepo; codecov
cppcomp /in yoselfc, out yoselfl/ yoselfo; codecov
eld /out linkout/ $system.system.ccplmain elfdumpo platdepo
   yoselfo -lcrtl -lcre -lpgo -o elfdump
```

3. This display shows the build directory after the build. Notice the new instrumented object files (which have names ending in O), the SPI file PGOSPI, and the lock file PGOSPL

Note. The lock file PGOSPL is created by the compiler to ensure that concurrent compilations do not attempt to read or write to the SPI file at the same time; access to the SPI file must be synchronized.

```
$DATA05.TEST


ELFCOMH   ELFDUMP   ELFDUMPC  ELFDUMPL  ELFDUMPO  LINKOUT
NSKELFH   PGOSPI    PGOSPL    PLATDEPC  PLATDEPH  PLATDEPL
PLATDEPO  YOSELFC   YOSELFH   YOSELFL   YOSELFO
```

Run the Application

1. These command lines run the application twice, each time with different input. (In each case, $system.system.eld is the input file, but the -h and -p options cause the runs to vary.)

```
elfdump /out output1/ -h $system.system.eld
elfdump /out output2/ -p $system.system.eld
```

2. This display shows the working directory, with the new raw data files (which have names starting ZZPF). Each raw data file represents one run of the application.

```
$DATA05.TEST


ELFCOMH   ELFDUMP   ELFDUMPC  ELFDUMPL  ELFDUMPO  LINKOUT
NSKELFH   OUTPUT1   OUTPUT2   PGOSPI    PGOSPL    PLATDEPC
PLATDEPH  PLATDEPL  PLATDEPO  YOSELFC   YOSELFH   YOSELFL
YOSELFO   ZZPFO9UJ  ZZPFU7
```

Produce the DPI File

1. This display shows the directory containing the input files for profmrg. The SPI file and raw data files have already been moved to the workstation from the server, where the build and test runs occurred. The SPI file has been renamed from the Guardian file name PGOSPI to the Windows file name pgopti.spi.

```
$ ls
pgopti.spi  zzpfo9uj  zzpfu7
```

2. This command runs the profmrg utility. The output DPI file will have the default name pgopti.dpi.

```
profmrg
```

3. The following display shows the directory, which now includes the DPI file to be submitted as input to codecov.

```
$ ls
pgopti.dpi  pgopti.spi  zzpfo9uj  zzpfu7
```

Measure Code Coverage

1. This display shows the directory containing the input files for codecov.

```
$ ls
pgopti.dpi  pgopti.spi  zzpfo9uj  zzpfu7
```

2. This command runs the codecov tool, specifying the host address and login information required to retrieve the elfdump source files from the Guardian file system. The prompt for a password would occur in practice but is omitted from this example. The `-counts` option causes execution counts to appear in the code coverage report.

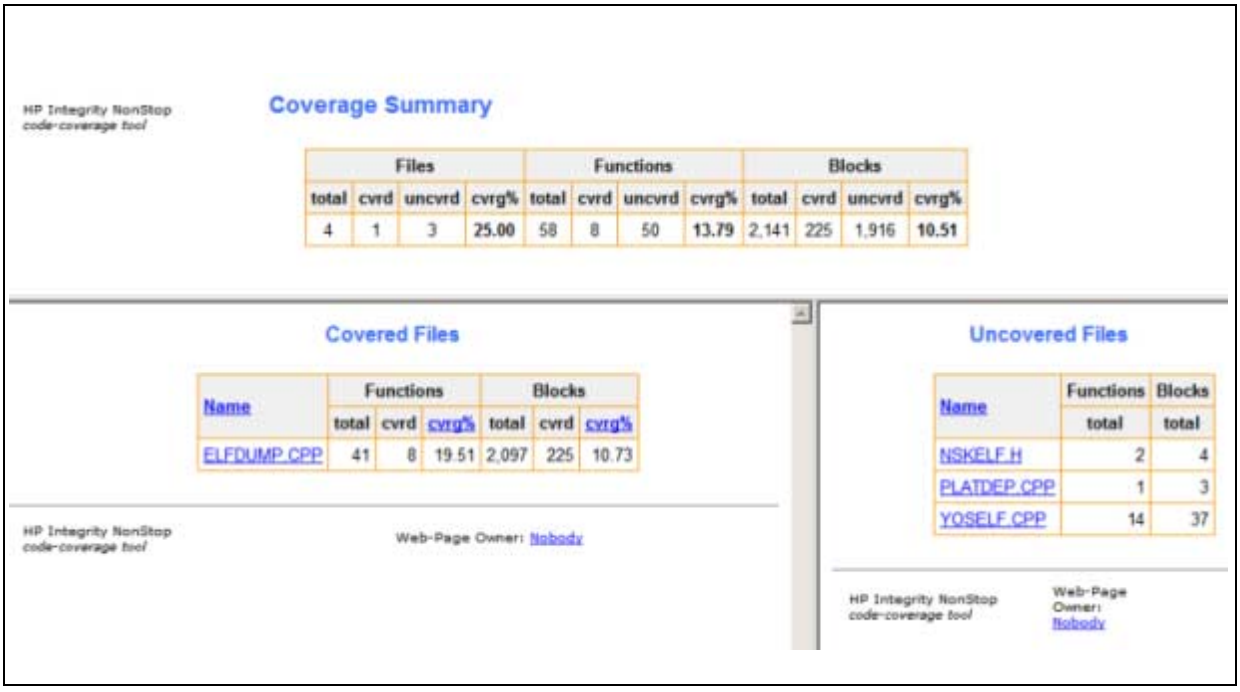
```
codecov -counts -host 16.107.174.143 -login super.super
```

3. This display shows the input directory, which now includes the code coverage report, in the file CODE_COVERAGE.HTML and the folder named CodeCoverage.

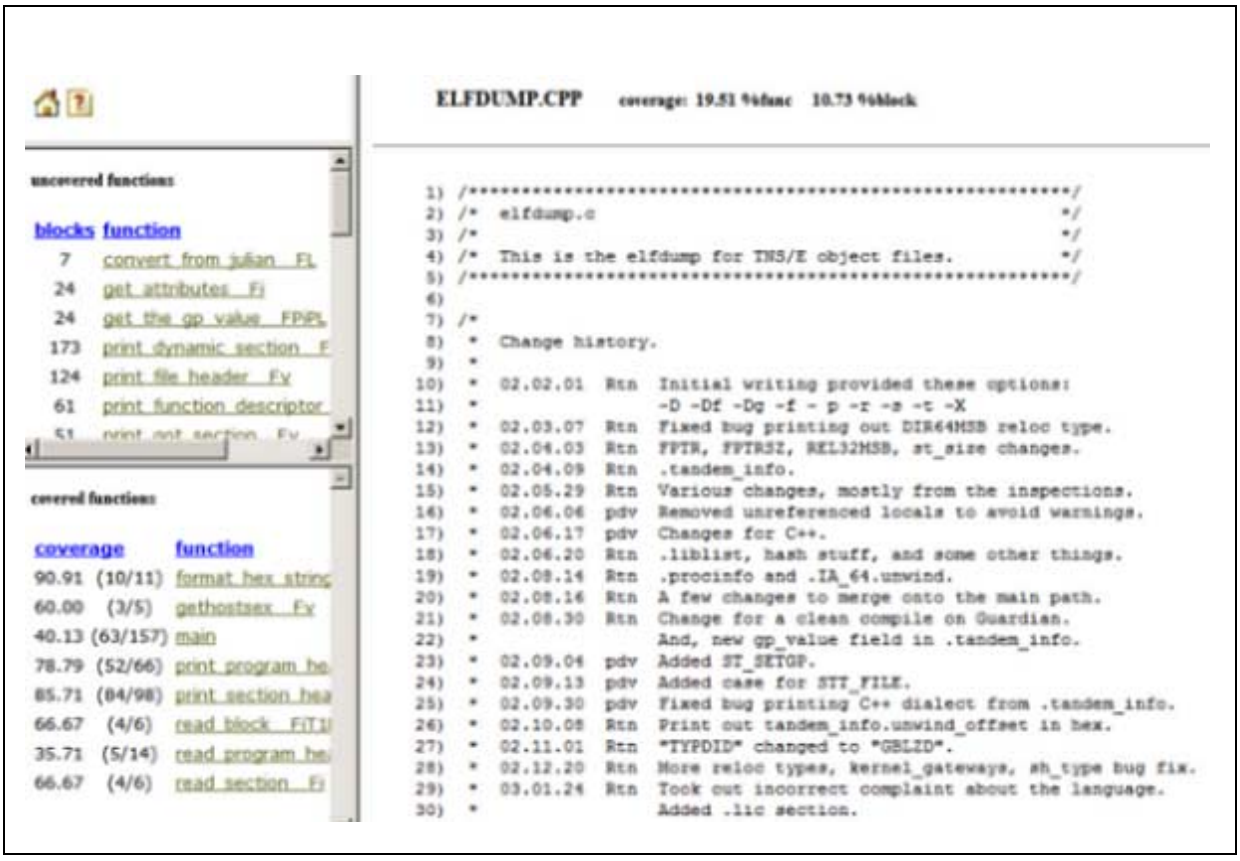
```
$ ls
CODE_COVERAGE.HTML  CodeCoverage  pgopti.dpi  pgopti.spi
zzpfo9uj  zzpfu7
```

Evaluate the Code Coverage Report

1. This screen is a summary of code coverage for the two test runs:



2. This screen shows specific coverage information for the source file ELFDUMPC:



3. Scrolling down the right side of the detailed display, you see the source code, with color coding to show what was covered and what was not. The following example shows a portion of the detailed display:

```
111) int gethostsex (void){
         ^ 2
112)    int     x;
113)    char *  p;
114)    int     answer;
115)
116)    x = 1;
117)    p = (char *) &x;
118)    if (*p == '\001')
119)       answer = ELFDATA2LSB;  /* little endian */
           ^ 0
120)    else
121)       answer = ELFDATA2MSB;  /* big endian */
           ^ 2
122)    return (answer);
        ^ 2
123)
124) } /* gethostsex */
     ^ 0
```

Note the following:

- Most of the lines are uncolored, meaning that they were executed.
- The “2” below line 111 indicates that the procedure was called twice (once for each execution of elfdump).
- Looking at the `if` statement in line 118, the first assignment is yellow, indicating that it was not executed. The second assignment is uncolored, indicating that it was executed. The “^ 0” under line 119 indicates that that line was executed zero times (in neither execution of the procedure), and the “^ 2” under line 121 indicates that that line was executed twice (once for each execution of the procedure).
- Line 120 is yellow because the compiler did not generate any code for the `else` statement, so that statement inherited its color from the preceding line.

For more information about how to interpret a display like this, see Section 7, Interpreting the Code Coverage Report

9 Usage Considerations

This section summarizes usage considerations for the Code Coverage Tool. Most of these issues are discussed in other sections of this manual but are repeated here for your convenience.

Compilation Issues

To support code coverage analysis, you must recompile existing program files or DLLs with the appropriate compiler and linker options, described in Section 3, Building the Application. When a source file changes, the code coverage information for that source file is incorrect until you recompile that source file with the code coverage compiler options and repeat the later steps to generate code coverage information.

An instrumented program file or DLL has the following characteristics:

- Some code optimizations--for example, partial redundancy elimination, function inlining, and loop unrolling--are limited or disabled.
- Instrumented code can be much larger than noninstrumented code.
- The maximum supported size for an object file has not increased. Thus, a very large program file, compiled with instrumentation, could exceed the supported object size.
- The maximum supported data area for an object file has not increased. Instrumenting a program file or DLL adds counters to the data area. The total size of data for these counters is proportional to the size (number of functions and basic blocks) in the instrumented code. Thus, a program file that already uses a large data area could, when compiled with instrumentation, exceed the maximum supported data area.
- Dynamically unloading an instrumented DLL is not supported.
- Code coverage analysis is not available for source code that is textually included within the body of a function. This limitation exists for all languages, but this style of source inclusion is commonly used only in COBOL programs.
- Compiling an application to produce an instrumented object file can take longer than a conventional compilation.

Application Performance

The code coverage tool is intended for data generation and collection in a test environment only. The use of instrumented code is not recommended for production environments. Applications compiled with the code coverage instrumentation will run much more slowly than noninstrumented code.

Glossary

basic block. A sequence of code that is entered only at the beginning and exited at the end. Once code enters a block, it executes the entire block unless an exception is raised within the block. Although the terms “basic block” and “block” are often used interchangeably, “block” is more generic and usually refers to any block (or sequence) of instructions.

code coverage. Information about which parts of the source code of a program file were executed during runs of the program file.

covered. Executed during a test run. A block is covered if it was entered in the course of a test run.

DPI file. Dynamic profiling information file, default name pgopti.dpi. This file is produced by profmrg from raw data file(s), existing DPI files, or both, and used as input by codecov.

instrumented code. Code into which the compiler inserted additional instructions to provide information about how the code behaved at runtime, such as which source lines were executed.

partially covered. Partially executed during a test run. When multiple basic blocks have the same source code location, with some blocks covered and others not, that location is said to be partially covered.

raw data files. Files produced by instrumented object code when it runs on the NonStop server and moved to the workstation for use as input by profmrg. These files have default names of the following form:

- ZZPF* if the program was run in the Guardian environment
- ZZPF* if the program was run in the OSS environment and the current directory is a Guardian subvolume
- *.dyn if the program was run in the OSS environment and the current directory is an OSS directory.

SPI file. Static profiling information file, having the name pgospi in the Guardian environment and pgopti.spi in the OSS and Windows environments. This file is produced by compilations and used as input by codecov.

uncovered. Never executed during a test run.

uncovered.

Index

H

I

L

M

N

O

P

R

S

T

U

Special Characters