
HP Branch Validator for AxLS C

User's Guide



HP Part No. B1418-97000
Printed in U.S.A.
March 1994

Edition 5

Notice

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

Copyright 1990,1991, 1994 Hewlett-Packard Company.

This document contains proprietary information, which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

HP is a trademark of Hewlett-Packard Company.

UNIX (R) is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

Torx is a registered trademark of Camcar Division of Textron, Inc.

Hewlett-Packard Company

P. O. Box 2197

1900 Garden of the Gods Road

Colorado Springs, CO 80901-2197, U.S.A.

RESTRICTED RIGHTS LEGEND Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software Clause in DFARS 252.227-7013. Hewlett-Packard Company, 3000 Hanover Street, Palo Alto, CA 94304, U.S.A. Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19 (c) (1,2).

Printing History

New editions are complete revisions of the manual. The date on the title page changes only when a new edition is published.

A software code may be printed before the date; this indicates the version level of the software product at the time the manual was issued. Many product updates and fixes do not require manual changes, and manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual revisions.

Edition 1	B1419-97000, May 1990
Edition 2	B1419-97001, June 1990
Edition 3	B1419-97002, November 1990
Edition 4	B1419-97003, November 1991
Edition 5	B1418-97000, March 1994

Using This Manual

Organization

The HP Branch Validator is based on the HP Basis Branch Analyzer product, which has been encapsulated within the HP SoftBench Interface. This product can function in the HP SoftBench environment with an OSF/Motif type interface. In addition, it has the capabilities of the previous BBA product in that it can operate in a standard-in/standard-out format. Throughout this manual, the HP Branch Validator product will be referred to as the BBA.

This manual is designed to show you how to use the BBA to create valid tests and use them to improve the quality of your product software. The BBA can be operated through the following three interfaces: (1) an emulation interface, (2) a debugger interface to an emulator or simulator, and (3) an emulator or simulator working within the HP SoftBench environment. This manual provides all of the information you will need to operate BBA with any one of these three interfaces.

- Chapter 1** Discusses general testing with the BBA. The tests made with the BBA determine the extent of execution of branches in your source program. Networked use of the BBA is also discussed in this Chapter.
- Chapter 2** Covers use of the BBA with an emulator or a debugger. It describes a sequence in which the BBA is used to develop a comprehensive test set to thoroughly test a software routine under development.
- Chapter 3** Covers use of the BBA through the HP SoftBench interface. It shows you how the test sequence performed in Chapter 2 can be performed more easily by using the capabilities of the HP SoftBench interface.

- Chapter 4** Shows you how to use the preprocessing routine to prepare your source files for testing with the Basis Branch Analyzer. Shows you how to use the options available to obtain specific information during your test runs.
- Chapter 5** Covers use of BBA with an emulator. It shows you how to use the command that builds the readable file of branch analysis information, after you have completed your tests.
- Chapter 6** Covers use of BBA with a debugger. It shows you how to use the command that builds the readable file of branch analysis information, after you have completed your tests.
- Chapter 7** Shows you how to obtain reports of measurement results and use these to develop a comprehensive test suite during software development.
- Chapter 8** Shows you how to eliminate redundancy in dump files to keep your dump file to its smallest possible size, without losing information.
- Chapter 9** Shows you how to create and use Makefiles when using BBA with an emulator or a debugger. This chapter also discusses regression testing. Finally, this chapter shows you how to use command files to run Makefiles and perform automatic regression testing with BBA and an emulator.
- Chapter 10** Provides the details you need to understand to obtain the best possible usage of BBA within the HP SoftBench interface.
- Appendix A** Lists each error message that is specific to the use of the Basis Branch Analyzer, discusses its meaning, and shows you the steps to take to clear the problem.

Contents

1 General Information

bbacpp	1-1
BBA Unload	1-2
bbarep	1-3
bbamerge	1-3
bba	1-3
Compatibility with Operating Systems Software	1-4
Compatibility with HP SoftBench Software	1-4
Compatibility with HP Development System Software	1-5
Networking	1-6

2 Getting Started Using BBA In An Emulator Or A Debugger (Walkthrough)

Networked Use Of BBA	2-2
The Test Program (And Directories)	2-4
The Driver Routine	2-5
The Makefile	2-6
Special Requirements Of The HP 64749A Emulator For 68331 And 68332	2-10
The Initial Test Set	2-10
Gathering Data	2-11
The Default Report	2-15
The Source-Reference Report	2-19
Avoiding Manual Re-verification Of Known-good Results	2-21
Using Ignore Files	2-25
Logically Dead Code	2-27
The Final Report - Net Benefits	2-35

3	Getting Started With The HP Branch Validator (BBA) In The SoftBench User Interface	
	Before You Start The HP Branch Validator (BBA) SoftBench User Interface	3-3
	Starting The Demo With SoftBench	3-4
	Starting The Demo Without SoftBench Broadcast Message Server	3-5
	The Main Branch Validator Window	3-6
	Pull Down Menu Bar	3-6
	Command Buttons	3-6
	Test Report Area	3-6
	Menu Mnemonics	3-6
	Menu Item Accelerators	3-8
	Customizing Mnemonics and Accelerators	3-8
	Displaying The Test Reports	3-8
	Details Of The Histogram Display	3-9
	Details Of The Summary Report	3-10
	Details Of The Results Only Display	3-12
	Details Of The File History Report	3-14
	Ignoring A File	3-16
	Displaying The Source Of A File Or Function With Unexecuted Branches Identified	3-16
	Ignoring A Branch	3-18
	Adding A Pragma	3-19
	Printing And Saving The Results	3-20
	How To Exit The HP Branch Validator (BBA) SoftBench User Interface	3-21
4	Details Of bbacpp	
	What bbacpp Does	4-4
	Amount Of Code Added	4-5
	Example Of bbacpp's Operation On A Simple Source File	4-5
	Explanation Of Lines In Figure 4-3	4-6
	How To Invoke bbacpp	4-8
	Relocating The BBA Constants	
	(-DBBA_OPTc<constname> Option)	4-10
	Relocating The BBA Data Array	
	(-DBBA_OPTd<dataname> Option)	4-11
	Details Of How bbacpp Modifies Your Code, And The -DBBA_OPTO=<options> Options	4-12

Default Instrumenting Of Branches (No -DBBA_OPTO=<options>)	4-14
Instrumenting Conditional Assignments (-DBBA_OPTO=a Option)	4-17
Instrumenting Do-While Statements (-DBBA_OPTO=d Option)	4-18
Instrumenting Case Statements (-DBBA_OPTO=e Option)	4-18
Instrumenting An If With No Else (-DBBA_OPTO=i Option)	4-20
Instrumenting A Switch With No Default (-DBBA_OPTO=s Option)	4-20
Instrumenting The Third Expression In A For Statement (-DBBA_OPTO=f Option)	4-21
Detecting A While Loop Always Executed (-DBBA_OPTO=w Option)	4-22
Using All -DBBA-OPTO Options Together (-DBBA_OPTO=A Option)	4-22
How (And Why) To Combine -DBBA_OPTO= Options	4-23
Changing The Map File Suffix (-DBBA_OPTM<character> Option)	4-26
Suppressing Creation Of The Map File (-DBBA_OPTS Option)	4-28
What Is A Pragma?	4-30
Detailed Explanation Of The BBA_IGNORE Pragma	4-30
Why Use Ignore Branches?	4-32
bbarep And BBA_IGNORED Branches	4-33
Detailed Explanation Of The BBA_IGNORE_ALWAYS_EXECUTED Pragma	4-34
Detailed Explanation Of The BBA_ALERT Pragma	4-35
Why Use BBA_ALERT	4-35
Increasing Push-Back-Line Memory (-DBBA_OPTp<lines> Option)	4-36
Reserved Words (Symbols)	4-36
Pitfalls With bbacpp And AxLS cc<COMP>	4-37
cc<COMP> -h (emulator only)	4-37
cc<COMP> -s	4-37
cc<COMP> -u	4-37
The ASM Pragma Of cc<COMP>	4-38
ccv20 -Q, ccv33 -Q, and cc8086 -Q	4-38

Linking Array And Data Sections	4-39
Pitfalls With bbacpp And MRI Compilers	4-39

5 Details Of bbaunload or Unload_BBA

What BBA Unload Does	5-1
Read Only If Using M68020, M68030, or M68040	5-4

6 Details Of bbarep

What bbarep Does	6-4
The BBAPATH Environment Variable	6-5
How To Set BBAPATH	6-6
Getting A Short Summary Report (-S Option)	6-7
Getting The Default Report (-s Option)	6-8
Getting Line Numbers In The Report (-l Option)	6-9
Explanation Lines For -l, -bN, And -aN Options	6-12
Showing Lines Before The Unexecuted Line (-bN Option)	6-17
Showing Lines After The Unexecuted Line (-aN Option)	6-19
Details Of Actions When -bN And -aN Options Are Combined	6-21
Showing Character Positions In The Report (-p Option)	6-22
Ignoring By Use Of The -i<ignorefile> Option	6-23
How To Ignore All Functions In A File	6-24
How To Ignore A Function	6-24
How To Ignore All Branches Generated By A Macro	6-25
How To Ignore All Branches Controlled By A Specific Statement	6-26
Selecting The Report Content With The -u <usefile> Option	6-31
Getting Reports From Other Dump Files (-d <file> Option)	6-32
How To Obtain Separate Reports Per Test Suite, And Then Combine Them	6-32
Specifying Spaces In A Tab (-e <tabs> Option)	6-33
Getting Reports That Include Older Dump Data (-o Option)	6-34
Suppressing Footnotes In Reports (-F Option)	6-35
Ignoring Branches In Include Files (-I option)	6-35
Appending Additional Information To Reports (-D[fvt] Option)	6-36
Listing Totals (-Dt Option)	6-36
Listing Files (-Df Option)	6-37
Listing Version Numbers (-Dv Option)	6-39

7	Details Of bbamerge	
	What bbamerge Does	7-1
	When To Use bbamerge	7-3
8	Tips On More Effective Testing Using BBA	
	Makefiles (make)	8-1
	Makefile Without Branch-Analysis Capability	8-3
	Makefile With Simple Branch-Analysis Capabilities	8-6
	Makefile With And Without Branch-Analysis Capabilities	8-7
	Automatic Makefile With And Without BBA	8-10
	Automatic/Efficient Makefile With And Without BBA	8-12
	Automatic Regression Testing	8-14
	Detailed Example Of Automatic Regression Testing	8-16
	Scripts to Run Emulator Command Files	8-17
	Makefile To Run Emulator Command Files	8-19
	Advantages Of Automatic Regression Testing	8-20
9	Details Of The HP Branch Validator (BBA) In The HP SoftBench Interface	
	Customizing The HP Branch Validator	9-1
	Details About The .bbarc File	9-3
	Options Controlling General BBA Operation	9-3
	Special BBA Options And Their Default Values	9-4
	BBA Options Used By The Actions Pull Down Menu	9-7
	Options Used When Not Using HP SoftBench Broadcast Message Server	9-10
	Before Starting BBA	9-11
	Starting BBA And Using The SoftBench Broadcast Message Server	9-12
	Invoking BBA From The Command Line	9-12
	Starting From The Tool Manager	9-12
	Starting BBA, But Not Using The SoftBench Broadcast Message Server	9-13
	Using The Four Test Report Displays	9-14
	The Summary Test Report	9-14
	The Histogram Test Report	9-15
	The Results Only Test Report	9-15
	The File History Test Report	9-15
	The BBAPATH Environment Variable	9-16
	How To Display Source Files	9-16

Ignoring A Branch In The Source Window	9-17
Exiting The Source Files Display	9-20
Selecting A Set Of Active Files And Functions To Appear In Test Reports	9-21
Displaying Error And Warning Messages	9-22
How To Ignore A File, Function, Or Branch	9-22
Adding Pragmas	9-25
Using Build	9-27
How To Control A Run Of Your Test	9-29
Accessing A File To Edit	9-30
Using Print And Save	9-31
Setting The Dump Data File And Retaining Old Data	9-32
Using Help	9-33
Exiting BBA	9-33
Restrictions When Using The HP Branch Validator (BBA) In The HP SoftBench Interface	9-34

A Error And Warning Messages

bbacpp Messages	A-2
BBA Unload Messages	A-10
bbarep Messages	A-13

B Installing The HP Branch Validator

Installation On HPUX Systems	B-1
Installation On SUN Sparc Systems With SunOS 4.X	B-2
Installation On SUN Sparc Systems With SunOS 5.X (Solaris)	B-2



General Information

Introduction

The quality of your product is directly related to the extent that you tested your product. The Hewlett-Packard Branch Validator (called BBA) is a software tool you can use to analyze your testing, create more complete test suites, and quantify your level of testing.

This chapter gives an overview of testing using the BBA. The tests made with the BBA determine the extent of execution of the branches in your source program. Branches are optional paths in a source program.

By doing branch analysis on the example in figure 1-1, you can obtain a record showing which (if either, or both) of the two possible branches were taken during any execution of the example program.

To do branch analysis testing, a BBA array must be defined and BBA statements must be inserted in your source code file(s) before the corresponding executable file is generated. See figure 1-2. These are inserted automatically by a BBA preprocessor routine (called `bbacpp`).

`bbacpp`

The first step in a typical compiler sequence is to submit the source file to a "C" language preprocessor. This preprocessor performs tasks to get the file ready for parsing by the compiler, such as "including" files specified by the "# include" commands, and the processing of "# ifdef" statements. The BBA uses a special "C" language preprocessor (called `bbacpp`) that is substituted for the normal "C" language preprocessor. This special "`bbacpp`" preprocessor does everything that the normal preprocessor does, and additionally, it inserts array definitions and BBA statements in the source file to record which branches were executed in the file. `Bbacpp` also creates a "map file", which contains

```
if (statement)
    statement 1;      /* branch 1 */
else
    statement 2;      /* branch 2 */
```

Figure 1-1. Source Program Without BBA Statements

```
static short _bA_array[23];

if (statement)
    { _bA_array[1]=1;statement 1; }
else
    { _bA_array[2]=1;statement 2; }
```

Figure 1-2. Source Program With BBA Statements Added

information necessary to map a specific branch with the associated source file.

When you run your executable file, the BBA statements write to the arrays to keep track of which branches were executed.

BBA Unload

When you have run your program as many times as you want, you can use the BBA unload command (bbaunload in the emulator or Unload_BBA in the debugger) to gather the data into an appropriate file for interpretation. You issue this command while still in your emulation or debugger session. This command unloads the data in the BBA arrays and stores it in a file called "bbadump.data". The "bbadump.data" file is formatted to be read by the routine that generates reports of branch analysis information ("bbarep").

1-2 General Information

bbarep

Now you obtain test reports by running "bbarep" from the shell. The bbarep command offers several options to generate reports based on the data contained in the "bbadump.data" file. You can use "bbarep" to derive information in a variety of report formats, each designed to give you a specific kind of information about the execution of your program. For example, you can measure the percentage of your program's branches that were executed during your series of tests. You can identify all of the branches that were not executed so that you can write additional tests to execute those branches. By this method, you can also identify unnecessary branches (if any) in your program.

bbamerge

The bbamerge routine can take large bbadump.data files and compress their information into smaller "bbadump.data" files. By using "bbamerge", you can reduce the overall execution time of a "bbarep" when you are running "bbarep" several times on a large "bbadump.data" file. You can also use bbamerge to recover some disk space when you are storing large "bbadump.data" files.

bba

BBA is a SoftBench tool that provides a complete environment in which to do basis branch analysis. It is an interactive, mouse-driven tool that helps you rapidly determine which branches of a program have not been taken. With the missed branches identified, you can modify your regression tests to take those branches (or choose to ignore the missed branches) to improve the percentages on your final branch coverage report.

BBA primarily uses bbarep and the SoftBench Encapsulator to produce the following types of output:

- Branch coverage summaries, illustrating file and function coverage.
- Branch coverage histogram for all functions.
- Source of file or function, with missed branches indicated and explained.



- Branch coverage file history, indicating the file dates and bbacpp options under which they were compiled.
- Results only summary, indicating total files, functions, branches, and coverage percents.
- List of the files and functions used to produce the final report.

Compatibility with Operating Systems Software

To use HP Branch Validator (BBA) on HP-UX systems, you must have an HP-UX operating system with software version number 8.0 (or higher).

To use HP Branch Validator on Sun SPARCstations, you must have SunOS Release 4.1 (or higher), and Sun OpenWindows 2.0 (or higher).

Compatibility with HP SoftBench Software

To use HP SoftBench and HP Branch Validator together on HP-UX systems, you must have similar versions of HP SoftBench and HP Branch Validator. This version of the HP Branch Validator software will work with HP Softbench version A.02.XX or B.00.XX or higher.

If you have a software version number of HP SoftBench that does not meet the requirements listed above, contact your HP Sales/Service Office for an update of your HP SoftBench Software.

If you do not have HP SoftBench, the SoftBench-style interface described in this manual will still operate, but in a slightly limited manner (discussed later in this manual).

Compatibility with HP Development System Software

The HP Branch Validator (BBA) will operate properly when it is used with an HP AxLS C compiler, HP emulator and/or HP debugger. In general, the HP Branch Validator will work with all HP AxLS C compilers with revision codes of A.03.10, or greater.

In addition, the HP Branch Validator will work with MRI compilers. In particular, it has been tested with the following MRI compilers:

MRI Compiler	Version
mcc68k	4.3A
mcc86	2.2A
ccc68k	1.1
mcc960	2.2

Networking

The HP Branch Validator (BBA) is supported over a network where software development is done on one host and emulation or debugger operation is done on another. Consult the manual "Networking HP64000 Software Development and Execution Environments" for details. This manual is supplied with the compiler manual for your HP AxLS C compiler, and with the assembler manual for your assembler.

If BBA is networked with an emulator or debugger on another host, the following files must be moved before operation:

1. One file must be moved from the software development host which hosts BBA to the emulator/debugger host. It is the **bbacpp.spec** file that is specific to your emulator/debugger. You will find its file name in table 1-1. Move this file once, before you start to use BBA. System privileges are required.
2. One file must be moved from the emulator/debugger host to the software development host. It is the file: **bbadump.data**. The "bbadump.data" file must be moved each time new test results are to be analyzed.

Table 1-1. List Of bbacpp.spec Files

Emulator In Use	Name Of File To Be Moved To Emulation Host
68000/302	\$HP64000/lib/68000/bbacpp.spec
68020	\$HP64000/lib/68020/bbacpp.spec
68030	\$HP64000/lib/68030/bbacpp.spec
68040	\$HP64000/lib/68040/bbacpp.spec
68332/340/360	\$HP64000/lib/68332/bbacpp.spec
8086-Series	\$HP64000/lib/8086/bbacpp.spec
80960-Series	\$HP64000/lib/80960/bbacpp.spec
V20-Series	\$HP64000/lib/v20/bbacpp.spec
V33-Series	\$HP64000/lib/v33/bbacpp.spec

Getting Started Using BBA In An Emulator Or A Debugger (Walkthrough)

Introduction

Frank Wiz is working on a routine. He is designing it to read two hardware registers that reflect which lines are down on a 5X5 grid of buttons, and map that information into a keycode reflecting which button was pressed. In this chapter, we will watch Frank create a test-set to thoroughly test that routine.

The files used in this chapter can be found in the directory named **\$HP64000/demo/bba/demo1**. In that directory are several sub-directories (e.g., **start**, **mod1**, etc.) that show the program and test-set data in several stages of development. To read a more complete description of the problem, see the introductory comments in the file **\$HP64000/demo/bba/demo1/start/getkey.c**.

In this chapter you will see examples of:

1. A quick 'driver' program to test a function.
2. A Makefile to compile programs either for branch analysis or regular execution.
3. Several different bbarep reports, showing when each is useful.
4. How to use BBA_IGNORE pragmas.
5. How to extend test-set data based on the coverage information.
6. How to avoid manual re-verification of known-good results.
7. How to use ignore files.

Note



Make sure your current PATH variable includes the path to the **hp64000/bin** directory. Typically, this is **/usr/hp64000/bin** or **/opt/hp64000/bin**.

Networked Use Of BBA

Follow the instructions in this paragraph only if you are using an emulator in a networked system, such as shown in figure 2-1. If you are using an emulator but not in a networked system (if your software host and emulation host are the same, or your emulator is on the LAN), **skip this paragraph**, and go directly to the paragraph titled, The Test Program (And Directories).

If you are using a networked system, you must have a login on both the software development host (SW host) and the emulation host (HW host). Both hosts must be running Networked File Services (NFS) for this walkthrough to work as presented. This walkthrough assumes you have logged in to the HW host.

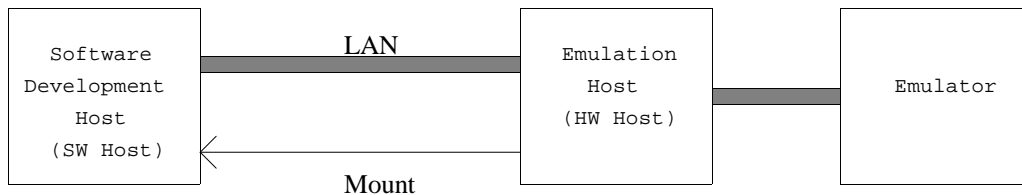


Figure 2-1. Networking Block Diagram

2-2 Getting Started (BBA/Emulator Walkthrough)

The system administrator for your networked system must enter the following commands from the HW host:

1. Make a home directory for the special BBA file on the HW host:

```
$mkdir $HP64000/lib/<emulator>
```

2. Copy special BBA file (listed in table 2-1) to the HW host:

```
$rcp <SW_host>:/<bba spec file from table 2-1>  
$HP64000/lib/<emulator>/bbacpp.spec
```

Table 2-1. List Of bbacpp.spec Files

Emulator In Use	Name Of File To Be Moved To HW Host
68000/302	\$HP64000/lib/68000/bbacpp.spec
68020	\$HP64000/lib/68020/bbacpp.spec
68030	\$HP64000/lib/68030/bbacpp.spec
68040	\$HP64000/lib/68040/bbacpp.spec
68332/340/360	\$HP64000/lib/68332/bbacpp.spec
8086-Series	\$HP64000/lib/8086/bbacpp.spec
80960-Series	\$HP64000/lib/80960/bbacpp.spec
V20-Series	\$HP64000/lib/v20/bbacpp.spec
V33-Series	\$HP64000/lib/v33/bbacpp.spec

3. Have "Root" on the SW host export the home_directory, and mount your files on the HW host.

4. Run this walkthrough from the HW Host.

Note



Rcp(1) is a Berkeley service.
Mount(1), as it is used here, is an NFS service.
Refer to the appropriate UNIX system manual for details of administration and usage.

The Test Program (And Directories)

Command lines beginning with the shell prompt "\$" are commands that you are to type into the shell during this demonstration. Command lines that do not begin with the "\$" prompt are to be typed into the emulator or debugger command line.

If you would like to follow along with this chapter, create an empty directory and copy all of the files from the directory **\$HP64000/demo/bba/demo1/start** to your empty directory. Adapt the commands shown in figure 2-2 to copy the files to your empty directory.

```
$ cd # go to your home directory
$ mkdir examples # make an empty directory
$ cd examples # go to the empty directory
$ cp $HP64000/demo/bba/demo1/start/* . # copy all files
```

Figure 2-2. Initial Setup For Walkthrough

Note that each subdirectory of **\$HP64000/demo/bba/demo1** has all of the files instead of just the files that change from the "previous" subdirectory. This is a convenience in case you want to jump to the middle of this walkthrough. At the beginning of each section, a comment will tell you which files should be copied to your current directory, if you want to start at the beginning of that section (if you do that, some warning messages may be different from those shown in this manual).

Edit "Makefile.old" and "Makefile" to specify the correct compiler invocations (CC=) and emulation environment (ENVFLAG=). If you start somewhere in the middle of these procedures, make sure you edit "CC=" and "ENVFLAG=" in the Makefile in the directory where you begin. Instructions for editing these parameters are contained in the text of Makefile.

Figures 2-3 and 2-4 show how the Makefile will appear after editing is complete.

2-4 Getting Started (BBA/Emulator Walkthrough)

Note



The makefiles supplied for this demonstration procedure contain instructions for modifying the emulation environment file "ENVFLAG=" when the HP Branch Validator is used on an emulator or a debugger/emulator.

The Driver Routine

(We are using the files in the directory named **\$HP64000/demo/bba/demo1/start**).

Sometimes it is useful to have a 'driver' routine to help when testing a function. A 'driver' routine is a program that fetches input (in a friendly way!) from a test-set (file), sends that data to a function (or functions) that you want to test, and then presents the results in such a way as to make it easy to decide whether or not the tested functions passed or failed the tests.

The driver routine in this example is in the file named "driver.c".

File "driver.c" creates a **main()** routine so we can use the default linker command file. We will not have to bother with creating a new linker command file. (Remember, this is to be used for testing, so the less time we have to spend here, the better. On the other hand, in many cases the ability to quickly tell if a test has passed or failed is important, so the test program may become quite elaborate - it just depends on the circumstances.)

This driver routine takes input from a file on the host system (via Simulated I/O), and sends the results to another file (again via Simulated I/O). This is a typical test method - it makes it easy to re-run the tests later when the program has been modified. You'll see this used later in this chapter.

This driver opens a file named "input" and sends the output to a file named "output". Both "input" and "output" are files located in the directory where the emulator or debugger is running.

The Makefile

(We are still using the files in the directory named **\$HP64000/demo/bba/demo1/start**.)

The UNIX program "make" is an indispensable aid to program development. Properly used, it will track changes made to source files and compile, assemble, and link only those files that need it (e.g., if you have 500 source files and modify only three of them, only the three modified files will be re-compiled and assembled, and then all 500 will be linked together to form your executable. This is quite a timesaver over always having to compile all files, and it is a lot safer than trying to remember exactly which files you changed!)

"Make" uses a description file (typically named "Makefile") to determine which source files are needed to create the executable. When Frank started out (i.e., before he wanted to use the BBA to help him test his routine), his Makefile looked like the one shown in figure 2-3 (also called "Makefile.old" in the directory **\$HP64000/demo/bba/demo1/start**).


```

#       Example Makefile
#
#       Target      Action
#       runtest    creates test program (runtest)
#
# Specify the appropriate compiler CC=ccXXXXX
CC=cc68000
#   Use CC=cc68000 if using a 68000, or 68302 processor.
#   Change to CC=cc68020 if using a 68020 processor.
#   Change to CC=cc68030 if using a 68030 processor.
#   Change to CC=cc68040 if using a 68040 processor.
#   Change to CC=cc68332 if using a 68332, 68340, or 68360 processor.
#   Change to CC=cc8086 if using an 8086-Series processor.
#   Change to CC=ccv20 if using a V20-Series processor.
#   Change to CC=ccv33 if using a V33-Series processor.
#
# Specify the environment variable (ENVFLAG) according to your emulator
ENVFLAG=-r hp64742
# See your compiler user's guide for the correct value
# A few of the values are listed below:
#   Change to ENVFLAG="-r hp64742" if using a 68000 emulator
#   Change to ENVFLAG="-r hp64746" if using a 68302 emulator
#   Change to ENVFLAG="-r hp64748" if using a 68020 emulator
#   Change to ENVFLAG="-r hp64747" if using a 68030 emulator
#   Change to ENVFLAG="-r hp64749A" if using a 68332A emulator
#   Change to ENVFLAG="-r hp64749B" if using a 68332B emulator
#   Change to ENVFLAG="-r hp64751" if using a 68340 emulator
#   Change to ENVFLAG="-r hp64780" if using a 68360 emulator
#
CFLAGS= -gw $(ENVFLAG)
SHELL=/bin/sh

#       Source files
CSRC = convert.c driver.c getkey.c multibits.c
#       Object files
COBJ = $(CSRC:.c=.o)

#       Main target. Create a program to test the 'getkey.c' file.
runtest : $(COBJ)
          $(CC) $(CFLAGS) -o runtest $(COBJ)

.SUFFIXES: .o .c

.c.o: *.c
          $(CC) $(CFLAGS) -c *.c

```

Figure 2-3. Makefile Without Branch Analysis Additions

Frank wants to be able to switch quickly between normal compilation and compilation with BBA. He changes the Makefile so that it has two targets: runtest, and bbatest, shown in figure 2-4.



Note



Frank has chosen one of the fancier options for modifying his Makefile. The paragraph on MAKEFILES in Chapter 8 shows simpler changes he might have made.

The Makefile shown in figure 2-4 (which is the file "Makefile") will automatically remember if the programs were last compiled with or without BBA statements, so it is totally automatic (which Frank thinks is a great idea!).

```
# Example Makefile
#
# Target      Action
# runtest    creates test program (runtest) w/o bba
# bbatest    creates test program (bbatest) w/bba
#
# NOTE: The file .NORMAL is present if the last compile did not use bba.
#       The file .BBA is present if the last compile did use bba.
#
# Specify the appropriate compiler CC=ccXXXXX
CC=cc68000
# Use CC=cc68000 if using a 68000, or 68302 processor.
# Change to CC=cc68020 if using a 68020 processor.
# Change to CC=cc68030 if using a 68030 processor.
# Change to CC=cc68040 if using a 68040 processor.
# Change to CC=cc68332 if using a 68332, 68340, or 68360 processor.
# Change to CC=cc8086 if using an 8086-Series processor.
# Change to CC=ccv20 if using a V20-Series processor.
# Change to CC=ccv33 if using a V33-Series processor.
#
```

Figure 2-4. Makefile With Branch Analysis Added

2-8 Getting Started (BBA/Emulator Walkthrough)

```

# Specify the environment variable (ENVFLAG) according to your emulator
ENVFLAG=-r hp64742
# See your compiler user's guide for the correct value
# A few of the values are listed below:
# Change to ENVFLAG="-r hp64742" if using a 68000 emulator
# Change to ENVFLAG="-r hp64746" if using a 68302 emulator
# Change to ENVFLAG="-r hp64748" if using a 68020 emulator
# Change to ENVFLAG="-r hp64747" if using a 68030 emulator
# Change to ENVFLAG="-r hp64749A" if using a 68332A emulator
# Change to ENVFLAG="-r hp64749B" if using a 68332B emulator
# Change to ENVFLAG="-r hp64751" if using a 68340 emulator
# Change to ENVFLAG="-r hp64780" if using a 68360 emulator
#

BBACMD= -b
BBAOPT= -DBBA_OPTO=A
CFLAGS= -gw $(ENVFLAG)
SHELL=/bin/sh

# Source files
CSRC = convert.c driver.c getkey.c multibits.c
# Object files
COBJ = $(CSRC:.c=.o)

# Normal make
# First check to see if the last time we compiled we
# used bba. If so, delete the object files and remind
# ourselves that we are compiling without bba.
runtest ::
    -if [ ! -f .NORMAL ] ; then rm $(COBJ) .BBA; touch .NORMAL; fi

# Now compile any out-of-date object files
runtest :: $(COBJ)
    # link the object files into the file 'runtest'
    $(CC) $(CFLAGS) -o runtest $(COBJ)

# BBA make
# If last time we compiled without bba remove all .o files;
# then compile any out-of-date files.
# Note that making runobjs forces all sources to be compiled with
# BBA options.
bbatest ::
    -if [ ! -f .BBA ] ; then rm $(COBJ) .NORMAL; touch .BBA; fi
    $(MAKE) CFLAGS="$(CFLAGS) $(BBACMD) $(BBAOPT)" CC="$(CC)" runobjs

bbatest :: $(COBJ)
    $(CC) $(CFLAGS) -o bbatest $(COBJ)

runobjs : $(COBJ)
    touch runobjs

.SUFFIXES: .o .c

.c.o: *.c
    $(CC) $(CFLAGS) -c *.c

```

Figure 2-4. Makefile With Branch Analysis Added (Cont)

Special Requirements Of The HP 64749A Emulator For 68331 And 68332

The HP 64749A emulator for the 68331 and 68332 processors has no emulation memory. All of the BBA software must be loaded into your target system memory (it will occupy about 40K of target system memory). In order to use the HP 64749A emulator, you will need to create your own linker command file (.k) to allocate target system memory. For an example linker command file, refer to **[io]linkcom.k** in the directory **\$HP64000/env/hp64749A**. This example linker command file was written for use in a system with 64k of target system memory beginning at address 0.

You will also need to edit the Makefile (figure 2-4). The ENVFLAG entry in Makefile is ignored when you modify "CFLAGS" so that it invokes the correct options for the HP 64749A Emulator. Edit Makefile to change CFLAGS to:

```
CFLAGS= -gw -k<your linker command ".k" file name>
```

The Initial Test Set

(For this section, we are still using the files in directory **\$HP64000/demo/bba/demo1/start**.)

For Frank's first attempt at a test-set, he decides to try the equivalent of hitting all of the buttons in the first column, and all of the buttons on the first row. (He knows this will cause the function to have been run with all bits set in both "horiz" and "vert".) Because our driver program reads the file named "input", this is shown as "input" in the "start" directory. See figure 2-5.

```
1,1
2,1
4,1
8,1
16,1
1,2
1,4
1,8
1,16
```

Figure 2-5. The First Test-Set Data

Gathering Data

(For this section, we are still using the files in directory **\$HP64000/demo/bba/demo1/start.**)

To gather the branch analysis data for this first set of test data, Frank does the procedure shown in figure 2-6. (If you are following along, you should do the numbered instructions in figure 2-6, also.) Don't be alarmed if some of the "rm" commands invoke messages about non-existent files; it just means we attempted to remove a file that wasn't there, as a precaution).

As you do these procedures, you may see warning messages about /bbatest.Ys/bldmessages.WY. Ignore these warning messages, also.

1. Make the absolute file that is tooled for branch analysis:

```
$ make bbatest
```

2. Delete any data left over from earlier BBA tests by using the command:

```
$ rm bbadump.data
```

Figure 2-6. Instructions List #1

3. Copy the emulation configuration file for your processor from the appropriate "/env" directory to your current directory. Use a command like:

```
$cp $HP64000/env/hp<emulator number>/ioconfig.EA .
```

Where <emulator number> is the HP product number of your emulator.

For example, if you have a 68020 emulator, use the command:

```
$cp $HP64000/env/hp64748/ioconfig.EA .
```

The configuration file assumes the monitor you linked in supports simulated I/O, and the program can be loaded into the emulator without problems (i.e., the memory map in the emulator is correct).

4. If networked, login to the HW host and access the "examples" directory:

```
$rlogin <HW_host>
```

```
$cd examples
```

5. Start the emulator interface or debugger interface by using the appropriate command, listed below:

```
emul700 <logical_name> (if starting an emulator)
```

```
dbxxxxx -e <logical_name> (if starting a debugger)
```

Where <logical_name> is the logical emulator name in the HP 64700 device table file (**\$HP64000/etc/64700tab.net**), and **dbxxxxx** is the debugger invocation (examples: **db68k**, **db86**, **db80960**).

Figure 2-6. Instructions List #1 (Cont'd)

6. Load the emulation configuration by using the following command:

File→Load→Emulator Config...

Select your file **.../ioconfig.EA** in the dialog box, and press OK.

7. Load the program into memory using the command:

File→Load→Executable...

Select your file **.../bbatest.x** in the dialog box, and press OK.

(Ignore any warnings about duplicate symbols)

8. Run the program using the command:

Execution→Run→from Transfer Address

9. When the program finishes (Prog end, returned 0 appears), Frank unloads the Branch Validator data using the command:

File→Store→BBA Data...

Enter the name of the BBA Dump file **bbadump.data** in the dialog box and press OK.

Figure 2-6. Instructions List #1 (Cont'd)

Then Mr. Frank G. W. looks at the output file which was created when the testing program ran. He wants to see if the output is correct. The output file looks like figure 2-7.

To do this, he opens a terminal window and then within the terminal window, he displays the output file:

File→Term...

\$ more output

As Frank looks through the data shown in figure 2-7, he sees that, yes, "1,1" should indeed return a "10" (because "10" is the keycode for the "a" key, as shown in keymap.h), and that "2,1" returns keycode "11" correctly, and so on.

```
( 1, 1) -> 10
( 2, 1) -> 11
( 4, 1) -> 12
( 8, 1) -> 13
(16, 1) -> 14
( 1, 2) -> 15
( 1, 4) -> 1
( 1, 8) -> 6
( 1, 16) -> 20
```

Figure 2-7. "output" File Using First Test-Set Data

Frank leaves the terminal window open. That is where he will get his reports.

The Default Report

(In this part of the procedure, we are still using the files in \$HP64000/demo/bba/demo1/start.)


Reassured that the function took the test data correctly, Frank now runs the "bbarep" program within the terminal window by entering the following command:

\$ bbarep

He wants to focus on functions that are only lightly tested, so he selects the default output (shown in figure 2-8).

```
_hit_ total_ %_ IA_ function_ file_
  8 /   12 ( 66.67)  keyconvert  convert.c
  6 /    9 ( 66.67)  bitpos    convert.c
  0 /    1 (  0.00)  error     driver.c
  3 /    5 ( 60.00)  getkeyvalue  getkey.c
  5 /    7 ( 71.43)  twobits    multibits.c
22 out of 34 retained branches executed (64.71%)
[ 23 branches were ignored ]
```

Figure 2-8. Default Branch Analysis Output



The first thing that Frank notices is that a function from his driver program (**error**) was included in the report. Frank doesn't care about testing his driver program. The appearance of this function annoys him so he goes into his driver.c program and adds a BBA_IGNORE pragma in **error** so it won't show up in the report anymore. (Chapter 4 explains the use of the BBA_IGNORE pragma.) Frank had remembered to do that for his other routines, but he forgot it for this one.

If you are following along, this change is reflected in the driver.c file found in the directory **\$HP64000/demo/bba/demo1/mod1**. Copy the file "driver.c" from that directory to your current directory by using the following command within the terminal window:

```
$ cp $HP64000/demo/bba/demo1/mod1/driver.c .
```

Then Frank reruns his tests as shown in figure 2-9.

1. Within the terminal window, make new version of program
tooled for branch analysis:

\$ make bbatest

2. Within the emulator or debugger interface, load the new
absolute file:

File→Load→Executable...

Select your file **.../bbatest.x** in the dialog box, and press OK.

(Ignore any warnings about duplicate symbols)

3. Run the program using the command:

Execution→Run→from Transfer Address

4. When the program finishes (Prog end, returned 0 appears),
Frank unloads the Branch Validator data using the command:

File→Store→BBA Data...

Enter the name of the BBA Dump file **bbadump.data** in the
dialog box and press OK.

5. From within the terminal window, Frank gets another default
report with the command:

\$ bbarep

Figure 2-9. Instructions List #2

The output of the bbarep command is shown in figure 2-10.

```
bbarep: warning: skipping data from older version of file driver.c
date of skipped file is 10/05/1987 12:46
_hit_total_%_IA_function_file
 8 / 12 ( 66.67) keyconvert convert.c
 6 / 9 ( 66.67) bitpos convert.c
 3 / 5 ( 60.00) getkeyvalue getkey.c
 5 / 7 ( 71.43) twobits multibits.c
22 out of 33 retained branches executed (66.67%)
[ 24 branches were ignored ]
```

Figure 2-10. Default Branch Analysis Output #2

Oh yes, Frank remembers, he forgot to remove the old branch analysis data (bbadump.data) before he ran the new version. Therefore, the report generator warned him that it was ignoring some data from an old version of the driver. No problem - Frank doesn't care about the data from driver.c, anyway.

The Source-Reference Report

(Now we are using the files from **\$HP64000/demo/bba/demo1/mod1**.)

The next item that catches Frank's eye is that the function 'getkeyvalue' has a rather low coverage (60%). Frank decides to do a better job of testing 'getkeyvalue'. To find out which branches were not hit during the test, Frank gets a source-reference listing of getkeyvalue. Frank likes to see four lines before the control statement, and four lines after the first non-executed statement, so he uses the following command and gets the output shown in figure 2-11.

```
$ bbarep -a4 -b4 getkeyvalue # Frank's command
```

"Well", says Frank, "I guess I should test the conditions of no bits set in either horiz or vert, as well as the case of multiple bits set." He modifies the file "input" by adding the lines shown in figure 2-12. His modifications are reflected in the file

```
$HP64000/demo/bba/demo1/mod2/input;
```

to get it, use a command like:

```
$ cp $HP64000/demo/bba/demo1/mod2/input .
```

"Wait a minute!", Frank says. "If I rerun the test with the new data, it'll re-write the file 'output', and I'll have to re-verify the first nine sets of data! Isn't there some way around that?"

(The way around this extra work is discussed in the next paragraph.)

```

bbarep: warning: skipping data from older version of file driver.c
date of skipped file is 10/05/1987 09:05

getkeyvalue  getkey.c
(1) 'then' part of 'if' was never executed
49  {
50      int keycode;          /* keycode we will return */
51
52      /* First test: if both horiz and vert are 0, just return KEY_NONE */
53      if ((horiz == 0) && (vert == 0))
54      {
55 ->          return(KEY_NONE);
56      }
57
58      /* Second: if more than one key pressed, return an error */
59      if ((twobits(horiz) != 0) || (twobits(vert) != 0))

(1) 'then' part of 'if' was never executed
55          return(KEY_NONE);
56      }
57
58      /* Second: if more than one key pressed, return an error */
59      if ((twobits(horiz) != 0) || (twobits(vert) != 0))
60 ->          return(KEY_NONE);
61
62          /* Do the conversion! */
63          keycode = keyconvert(horiz, vert);
64

3 out of 5 branches executed (60.00%)

```

Figure 2-11. Source-Reference Branch Analysis Output

```

0,0
3,1
2,7

```

Figure 2-12. First Additions To "input" File

2-20 Getting Started (BBA/Emulator Walkthrough)

Avoiding Manual Re-verification Of Known-good Results

(We are using the files in directory
\$HP64000/demo/bba/demo1/mod2.)

When a test is run on a set of test data, the output is checked (manually) to make sure it is correct (i.e., the test-set "passed"). Wouldn't it be nice to never have to manually check that test-set's output again? After all, if you have a lot of output to check, you might easily miss a significant error.

UNIX helps again. There is a program (called 'diff') that will show differences between two ASCII files.

Frank uses the following command to copy the current "output" file (which contains data that he knows is OK) to a file he calls "output.good":

\$ cp output output.good

Then Frank re-runs the tests by using the sequence of instructions in figure 2-13.

1. Remove the current dump data file (to avoid those nasty warning messages):

\$ rm bbadump.data

2. From within the emulator or debugger interface, Frank executes the test with the new input data (no need to reload the program; it has not been changed):

Execution→Run→from Transfer Address

4. When the program finishes, Frank stores the new Branch Validator data using the command:

File→Store→BBA Data...

Enter the name of the BBA Dump file **bbadump.data** in the dialog box and press OK.

Figure 2-13. Instructions #3

Now Frank needs to see if the new output is good. He uses the 'diff' command within the terminal window so that any differences will leap out at him. The command is shown below. Its output is shown in figure 2-14.

```
$ diff output.good output # Frank's command
```

Figure 2-14 shows that the only differences found by "diff" were that lines 10 thru 12 in "output" were added after line 9 in "output.good". That means that the first nine lines - the previous output that Frank had manually verified - didn't change. There is no need to even look at the first nine sets of input that was passed to the function under test. (Refer to the manual page on "diff" in your UNIX system for more information on how to read "diff" outputs. Use the command "man diff" to do this.)

At any rate, the additional tests show good news for Frank: "0,0" should, indeed, return a "-1" (because -1 is the keycode for "KEY_NONE"), and the same goes for "3,1" and "2,7" because they both show multiple keys being down.

Because this output is "good", Frank copies it to output.good so that he doesn't have to check it over again, later.

```
$ cp output output.good
```

```
9a10,12
> ( 0, 0) -> -1
> ( 3, 1) -> -1
> ( 2, 7) -> -1
```

Figure 2-14. Display Of "diff" Output

Now Frank wants to get a report that shows his new test coverage so he runs the default bbarep command.

\$ bbarep

The summary report produced by the bbarep command is shown in figure 2-15.

```
_hit_ total_ %_ IA_ function_ file_
  8 /   12 ( 66.67)  keyconvert      convert.c
  6 /    9 ( 66.67)  bitpos          convert.c
  5 /    5 (100.00)  getkeyvalue     getkey.c
  6 /    7 ( 85.71)  twobits         multibits.c
25 out of 33 retained branches executed (75.76%)
[ 24 branches were ignored ]
```

Figure 2-15. Summary Report After Changing Test-Set

Great! The test input achieves 100% coverage of the **getkeyvalue** routine. Now Frank decides to find the untested parts of another low-coverage routine, "bitpos". Frank issues the following bbarep command, and obtains the report shown in figure 2-16:

\$ bbarep -a4 -b4 bitpos # Frank's command

```
bitpos  convert.c
(1) 'case' code was never executed
71      return(2);
72      case 8 :
73      return(3);
74      case 16 :
75      return(4);
76      case 32 :
77 ->    return(5);
78      case 64 :
79      return(6);
80      case 128 :
81      return(7);

(1) 'case' code was never executed
73      return(3);
74      case 16 :
75      return(4);
76      case 32 :
77      return(5);
78      case 64 :
79 ->    return(6);
80      case 128 :
81      return(7);
82      }
83  }

(1) 'case' code was never executed
75      return(4);
76      case 32 :
77      return(5);
78      case 64 :
79      return(6);
80      case 128 :
81 ->    return(7);
82      }
83  }
(eof)

6 out of 9 branches executed (66.67%)
```

Figure 2-16. BBA Source-Reference For bitpos

Using Ignore Files

(At this point, we are using the files in directory **\$HP64000/demo/bba/demo1/mod2.**)

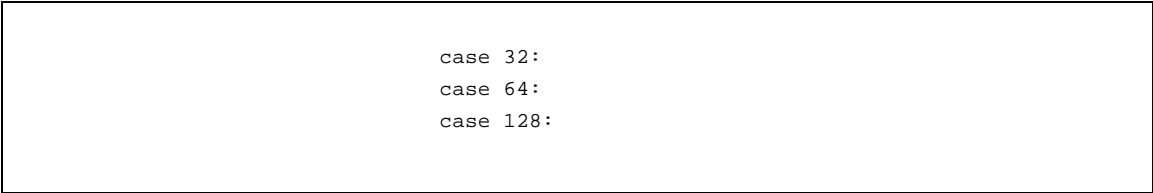
"Ah ha!", Frank exclaims. "I can never get to cases 32, 64, and 128 (for legal input!) because only bits 0 thru 4 are legal". Upon investigation, however, he discovers that Shirley (another programmer in his group) would like to use his routine to do the same task, but she needs all eight bits to be returned. To save memory space, Frank and Shirley decide to make the routine work for both applications. However, for his comfort, Frank wants his BBA test reports to ignore the branches he won't need. He asks Shirley if he can place "BBA_IGNORE" pragmas in his source file for those cases.

"Surely you jest!", she replies. "I'll need to test those, and if you put BBA_IGNORE pragmas in those cases, I won't ever see whether or not they are executed!"

"O.K.," Frank says. "There's another way to do this, and it'll work for both of us." He creates the file "ignorelist" (which is already in the **mod2** directory). It looks like figure 2-17.

If you do not have the 'ignorelist' file in your directory, copy it to your current directory by using the following command:

```
$ cp $HP64000/demo/bba/demo1/mod2/ignorelist .
```



```
case 32:  
case 64:  
case 128:
```

Figure 2-17. The "ignorelist" File

Here is Frank's new (summary) bbarep command. The output it produces is shown in figure 2-18.

\$ bbarep -i ignorelist # Frank's command

Frank notes that **bitpos** now has 100% coverage, but the report notifies him that some branches in the routine were ignored.

```
_hit__total__%__IA__function_____file_____
 8 / 12 ( 66.67)  keyconvert                convert.c
 6 /  6 (100.00) * bitpos                  convert.c
 5 /  5 (100.00)  getkeyvalue              getkey.c
 6 /  7 ( 85.71)  twobits                   multibits.c
25 out of 30 retained branches executed (83.33%)
[ 27 branches were ignored ]

NOTE:
A '*' in the 'I' column means this function had one or more
    branches that were ignored
```

Figure 2-18. BBA Report Using An Ignore File

Logically Dead Code

(We are still using the files in directory \$HP64000/demo/bba/demo1/mod2.)

Frank next looks at the unexecuted branches in keyconvert. (See figure 2-20.) He uses the following command:

```
$ bbarep -a4 -b4 keyconvert # Frank typed this
```

"Hmm," Frank says. "The first one is pretty easy. I'll just add the equivalent of pressing the "0" key (16, 8). But how can the conditional of the 'do while' be both 'never TRUE' and 'never FALSE'?!?! Oh, I see. I always executed the 'return' statement on line 45 (and therefore I never got to the conditional of the 'do while'). Maybe I'd better add some cases to exercise the while conditional." After some thinking, Frank adds the lines shown in figure 2-19 to the file "input".

To get this new version of the file "input", use the command:

```
$ cp $HP64000/demo/bba/demo1/mod3/input .
```



```
16,8
2,16
16,16
```

Figure 2-19. Additional Lines To "input"

```

keyconvert  convert.c
(1) 'then' part of 'if' was never executed
32         keycode = bitpos(horiz) + KEY_1;
33         break;
34     case 8: /* pressing fourth column of keys */
35         keycode = bitpos(horiz) + KEY_6;
36         if (keycode > KEY_9) /* special case for 0 key */
37 ->             keycode = KEY_0;
38         break;
39     case 16: /* pressing fifth column of keys */
40         keycode = KEY_RUN;
41         do

(1) conditional of 'if' was never FALSE (no 'else' statement) [control lines]
40         keycode = KEY_RUN;
41         do
42         {
43             /* If lowest bit is set, return its keycode */
44 ->         if (horiz == 1)
45             return(keycode);
46             horiz = horiz >> 1;
47             keycode++;
48         }

(1) conditional of 'do while' was never TRUE [control lines]
45             return(keycode);
46             horiz = horiz >> 1;
47             keycode++;
48         }
49 ->         while (horiz != 0);
50     }
51     return(keycode);
52 }
53 }

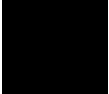
(1) conditional of 'do while' was never FALSE [control lines]
45             return(keycode);
46             horiz = horiz >> 1;
47             keycode++;
48         }
49 ->         while (horiz != 0);
50     }
51     return(keycode);
52 }
53 }

8 out of 12 branches executed (66.67%)

```

Figure 2-20. BBA Source-Reference For keyconvert

Now Frank runs the test program again with the new data. The steps he follows are shown in figure 2-21.



1. Remove the current dump data (it's a good habit):
\$ rm bbadump.data
2. Within the emulator or debugger interface, rerun the program. (No need to load the program; it has not changed):
Execution→Run→from Transfer Address
3. When the program finishes, Frank unloads the Branch Validator data using the command:
File→Store→BBA Data...
Enter the name of the BBA Dump file **bbadump.data** in the dialog box and press OK.
4. From within the terminal window, Frank verifies that the output file is correct by using the command:
\$ diff output.good output
(Refer back to figure 2-14 and its supporting text to understand the output you get at this point.)
5. The above command automatically checks the "known good" (i.e. previously verified) data, and Frank manually checks the additional values.
6. Now copy the output file to "output.good" so you can preserve the new "known good" data:
\$ cp output output.good
7. Get another detailed report for keyconvert:
\$ bbarep -a4 -b4 keyconvert

Figure 2-21. Instructions #4

The new report is shown in figure 2-22. After some reflection, Frank realizes that the while loop will only become FALSE if the value of `horiz` either (A) has more than one bit set, or (B) is zero on entry to **keyconvert**. Neither one of these cases are legal input, but Frank wonders if these cases are dealt with appropriately. (The problem definition states that for non-legal input, the program should return `KEY_NONE`. Frank wonders if it does that correctly.) Looking at the routine **getkeyvalue** (in `getkey.c`), Frank sees that both cases are trapped prior to calling **keyconvert**.

Therefore, the test "**while (horiz != 0)**", in **keyconvert**, doesn't make a whole lot of sense; **horiz** can never be 0!

```
keyconvert  convert.c
(1) conditional of 'do while' was never FALSE [control lines]
45                                     return(keycode);
46                                     horiz = horiz >> 1;
47                                     keycode++;
48                                     }
49 ->                                 while (horiz != 0);
50                                     }
51
52     return(keycode);
53 }
```

11 out of 12 branches executed (91.67%)

Figure 2-22. Another Source-Reference Output

To tell the truth, this loop was only added to point out that you can discover "logically dead" code by using the BBA. Frank would not have written the code this way (Frank made us put that disclaimer in the manual). The natural way to have written **case 16** in **keyconvert** is shown in figure 2-23.

Frank now changes convert.c to the way shown in figure 2-23. To see Frank's rewrite, enter the following command:

```
$ cp $HP64000/demo/bba/demo1/mod4/convert.c .
```

```
case 16: /* pressing fifth column of keys */
    keycode = bitpos(horiz) + KEY_RUN;
```

Figure 2-23. Natural Way To Write Case 16's Code

Frank now recompiles, runs the tests, and gets yet another report by using the steps shown in figure 2-24. The report is shown in figure 2-25.

1. Make the new version of the test:

```
$ make bbatest
```

2. Remove the current dump data (it's a good habit to develop):

```
$ rm bbadump.data
```

3. Within the emulator or debugger interface, load the new absolute file:

```
File→Load→Executable...
```

Select your file `.../bbatest.x` in the dialog box, and press OK.

(Ignore any warnings about duplicate symbols)

4. Run the program using the command:

```
Execution→Run→from Transfer Address
```

5. When the program finishes, Frank unloads the Branch Validator data using the command:

```
File→Store→BBA Data...
```

Enter the name of the BBA Dump file `bbadump.data` in the dialog box and press OK.

6. From within the terminal window, Frank gets another summary report, using the ignore file:

```
$ bbarep -i ignorelist
```

Figure 2-24. Instructions #5

```

_hit   total   %   IA   function   file
-----
  8 /    8 (100.00)  keyconvert   convert.c
  6 /    6 (100.00) * bitpos     convert.c
  5 /    5 (100.00)  getkeyvalue  getkey.c
  6 /    7 ( 85.71)  twobits      multibits.c

```

25 out of 26 retained branches executed (96.15%)
[27 branches were ignored]

NOTE:

A '*' in the 'I' column means this function had one or more
branches that were ignored

Figure 2-25. Report Using "-i ignorelist"

Continuing to track down the unexecuted branches, Frank issues the command:

```
$ bbarep -i ignorelist -a4 -b4 multibits.c
```

(Note: Frank didn't need the **-i ignorelist** in his last command because the ignorelist file doesn't contain any code pertaining to the file **multibits.c**. On the other hand, using the "ignorelist" file didn't hurt him, either.)

As Frank looks at his source-reference output (figure 2-26), he sees that the while loop is only skipped when the routine is entered and **bitmap** is 0. The routine **getkeyvalue** checks for 0 values before this is called. It might make more sense to change this to a "do-while" loop instead of a "while" loop.

```
twobits  multibits.c
(1) body of 'while' loop was never skipped
20
21         /* This loop continues only while there is at least
22             one more bit set in bitmap
23         */
24         while (bitmap != 0)
25         {
26 ->             if ((bitmap & 1) != 0)
27                 numbits++; /* increment count of bits that are set */
28                 bitmap = bitmap >> 1; /* test next bit */
29         }
30

6 out of 7 branches executed (85.71%)
```

Figure 2-26. Another Source-Reference Output

Remember, Shirley wants to use this routine, too. She wants the function's documentation to show that it will return a '0' only if the passed parameter has exactly one bit set (she thinks that the documentation implies that it returns a '0' if 1 or 0 bits are set).

Therefore, Frank changes the documentation and adds "while (bitmap != 0)" to the "ignorelist" file. These changes are shown in the following files. Copy them to your directory using the following commands:

```
$ cp $HP64000/demo/bba/demo1/finish/multibits.c .  
$ cp $HP64000/demo/bba/demo1/finish/ignorelist .
```

The Final Report - Net Benefits

(We are now using the files in the directory
\$HP64000/demo/bba/demo1/finish.)

Frank now recompiles, runs the tests, and gets the last report by using the steps shown in figure 2-27. The last report is shown in figure 2-28.

1.Recompile:

```
$ make bbatest
```

2.Remove the current dump data:

```
$ rm bbadump.data
```

3.Within the emulator or debugger interface, load the new absolute file:

```
File→Load→Executable...
```

Select your file **.../bbatest.x** in the dialog box, and press OK.

(Ignore any warnings about duplicate symbols)

4.Run the program using the command:

```
Execution→Run→from Transfer Address
```

5.When the program finishes, Frank unloads the Branch Validator data using the command:

```
File→Store→BBA Data...
```

Enter the name of the BBA Dump file **bbadump.data** in the dialog box and press OK.

6.From within the terminal window, Frank gets another summary report, using the ignore file:

```
$ bbarep -i ignorelist
```

Figure 2-27. Instructions #6

```

_hit   total   %   IA   function   file
 8 /    8 (100.00)  keyconvert   convert.c
 6 /    6 (100.00) * bitpos       convert.c
 5 /    5 (100.00)  getkeyvalue  getkey.c
 3 /    3 (100.00) * twobits    multibits.c
22 out of 22 retained branches executed (100.00%)
[ 31 branches were ignored ]

NOTE:
A '**' in the 'I' column means this function had one or more
      branches that were ignored

```

Figure 2-28. Frank's Last Report

Frankly, this was a lot of work. Let's see what Frank has gained:

1. Frank now has high confidence that his routine works.
2. If Frank needs to change the routine, he does not need to create his tests all over again. He can use his old tests and only write new tests.
3. When Frank changes a routine in the future, he does not worry that he may 'break' something. He can just run these tests over again and still have high confidence that everything is working.
4. If Shirley has to take over maintenance of these routines, she is assured that these routines have been well tested. She won't be nervous about having to support them.
5. Both Frank and Shirley can share code, knowing that if any changes are made to it, they can run the old tests and discover any unanticipated side effects (i.e., somebody adds some functionality, but their "addition" changes some output that it shouldn't have changed. Re-running these tests will quickly locate the problems).

Notes



Getting Started With The HP Branch Validator (BBA) In The SoftBench User Interface

Introduction

Before you perform the procedures in this chapter, do the getting started procedures in Chapter 2. The procedures in Chapter 2 will show you how BBA is used to develop a complete test suite. In this chapter, you will see how the HP Branch Validator (BBA) SoftBench User Interface can simplify many aspects of the branch analysis development process.

The HP Branch Validator (BBA) is a SoftBench tool that provides a complete environment in which to do basis branch analysis testing. It utilizes the mouse and menus to provide a point-and-click method for doing basis branch analysis. This interface will dramatically reduce the amount of time you need to spend to analyze BBA test reports, create ignore files, and add pragmas. In addition, the HP Branch Validator (BBA) SoftBench User Interface provides many parameters that you can define to further increase the speed at which operations are done.

This interface is available on any HP-UX operating system with software version number 8.0 (or higher). You must have HP SoftBench or HP SoftBench Framework to use this interface. HP SoftBench must be software version number B.00.00 or greater. Note that HP SoftBench Framework is included with the HP-UX B1418 Branch Validator. Refer to the installation instructions in Appendix B for proper installation.

This interface is available on Sun SPARCstations with SunOS Release 4.1 (or higher), and Sun OpenWindows 2.0 (or higher). When used on Sun SPARCstations, HP SoftBench (version B.00.00 or higher) is not required, but it is highly recommended. The HP Branch Validator installation on SUN SPARCstations will automatically install the needed HP SoftBench files if they do not exist.

In this chapter you will see examples of:

- How to start the HP Branch Validator (BBA) SoftBench User Interface
- How to display BBA test reports
- How to ignore a file
- How to display the source code of a file or function, identifying its unexecuted branches
- How to ignore a branch
- How to add a pragma
- How to print test results
- How to Quit the HP Branch Validator (BBA) SoftBench User Interface

Note



Make sure your current PATH variable includes the paths to **/softbench/bin** and **hp64000/bin**. If not, add those directories to your current PATH variable before starting BBA. Typically, the path to **/softbench/bin** is the first path in your PATH shell variable.

Before You Start The HP Branch Validator (BBA) SoftBench User Interface

Before starting these demonstration procedures, you need to gain access to the directory supplied by HP that contains the files for these demonstrations. Then you will enter commands to make sure the demonstration files are clean, and to install the appropriate configuration file in your home directory.

BBA obtains its branch coverage information from a bbadump.data file and its associated ".M" (map) files. The ".M" files are created when a program is compiled using the BBA preprocessor "bbacpp" with an HP AxLS or Host compiler. The bbadump.data file is created when you unload it from your emulator or simulator.

The demonstrations in this chapter use a bbadump.data file and corresponding map files that were created by HP and supplied in directory **\$HP64000/demo/bba/demo2**. To access the demonstration directory, use the following command:

```
cd $HP64000/demo/bba/demo2
```

Before starting the HP Branch Validator (BBA) SoftBench User Interface, enter the following commands to ensure that the demonstration will run properly:

```
Demo_clean # Cleans up the demo
```

```
Demo_install # Installs the appropriate configuration file in your $HOME directory
```

If you do not have HP SoftBench, or you do not intend to use the HP SoftBench Broadcast Message Server, enter:

```
Demo_install no_softbench. This command sets up the demo to not utilize the HP SoftBench Broadcast Message Server for the execution of Edit, Build, and other commands.
```

Starting The Demo With SoftBench

Begin by starting SoftBench. Enter the following command:

softbench

Move the cursor into OK in the Tool Manager copyright statement, and click the command select mouse button.

Start BBA from within the SoftBench Tool Manager window, as follows:

Move the cursor into the Tool pull down and click on Start.

A new window will open, showing a list of tools available in the SoftBench Tool Manager. You may have to scroll down this window to find BBA. Click on BBA in this list. This selects BBA as the tool to be started. To start the BBA tool, press the Start button. You could also have started BBA by double-clicking the command select mouse button (two quick presses) with the cursor on BBA in the list of tools.

Close the Tool Manager - Start window by moving the cursor to the "Close" button and clicking the command select mouse button.

BBA can also be invoked without using the HP SoftBench Tool Manager. This is done by entering `bba` on the command line of your host system (like any UNIX command) and pressing the return key. If SoftBench Broadcast Message Server is already running, the `bba` invocation will connect with the SoftBench Broadcast Message Server. The SoftBench Broadcast Message Server is used to pass the Edit, Build and Help commands of BBA. If SoftBench Broadcast Message Server is not running, BBA will still start and run, but the Edit, Build, and Help features of the interface will not be available. Enter the command:

bba

Starting The Demo Without SoftBench Broadcast Message Server

Make sure you have the X window system running, and you have run the "Demo_install" command with the "no_softbench" option. Enter the command:

Demo_install no_softbench

Start the HP Branch Validator (BBA) from the command line by entering the following command:

bba

The message "Starting the Basis Branch Analyzer Interface..." will appear on your display and BBA will start in a few moments.

Additionally, you may see a message like:

```
encaprun: cannot find a message server
  Make sure that there is a message server running and
  that either $DISPLAY or $MSERVE were set correctly,
  both when SoftBench was started, and also in the current environment.
```

This message is simply a warning message and can be disregarded. When properly set up, this mode of operation will still allow you to use all of the BBA commands in the pull down menus, with the exception of the Help command. The Help command uses resources within the SoftBench Broadcast Message Server.

Note



If you have HP SoftBench installed, you must add "/usr/softbench/bin" to your PATH variable even if you are not using the SoftBench Broadcast Message Server. If you do not have HP SoftBench installed, do not add the above to your PATH variable.

The Main Branch Validator Window

The main Branch Validator window will be on screen. See figure 3-1. It is labeled "Softbench - Branch Validator". It will contain a BBA test report about files named **convert.c**, **driver.c**, **getkey.c**, and **multibits.c**. The main Branch Validator window presents three sets of control selections for controlling tests and formatting test reports:

1. The Pull Down Menu Bar.
2. The Command Buttons Row.
3. The Test Report Area.

Pull Down Menu Bar

The Pull Down Menu Bar (with pull downs hidden under File, Actions, Edit, Settings, and Help) sets up the environment in which basis branch analysis can be performed. For example, you can redefine the location of the bbadump.data file by setting the context directory of the bbadump.data file (the location of this file must be defined before you make any BBA tests).

For a complete BBA test report, choose **File→Show Summary** or **File→Show Histogram**.

Command Buttons

The Command Buttons row (row of on-screen buttons labeled Next, Previous, Ignore, Source, Update, and Errors) selects and/or ignores functions and files listed in the test report area.

Test Report Area

The test report area allows you to scroll through a BBA test report and find the files or functions that have low test coverage.

Menu Mnemonics

The HP Branch Validator supports Pull Down Menu Mnemonics. These are shortcuts for selecting items in pull down menus. The mnemonic for a menu category is the underscored letter in the text of the menu category.

To make a selection, press the <Alt> key and the mnemonic key for the pull down menu category (i.e. F, A, E, S, or H, corresponding to File, Actions, Edit, Settings, or Help).

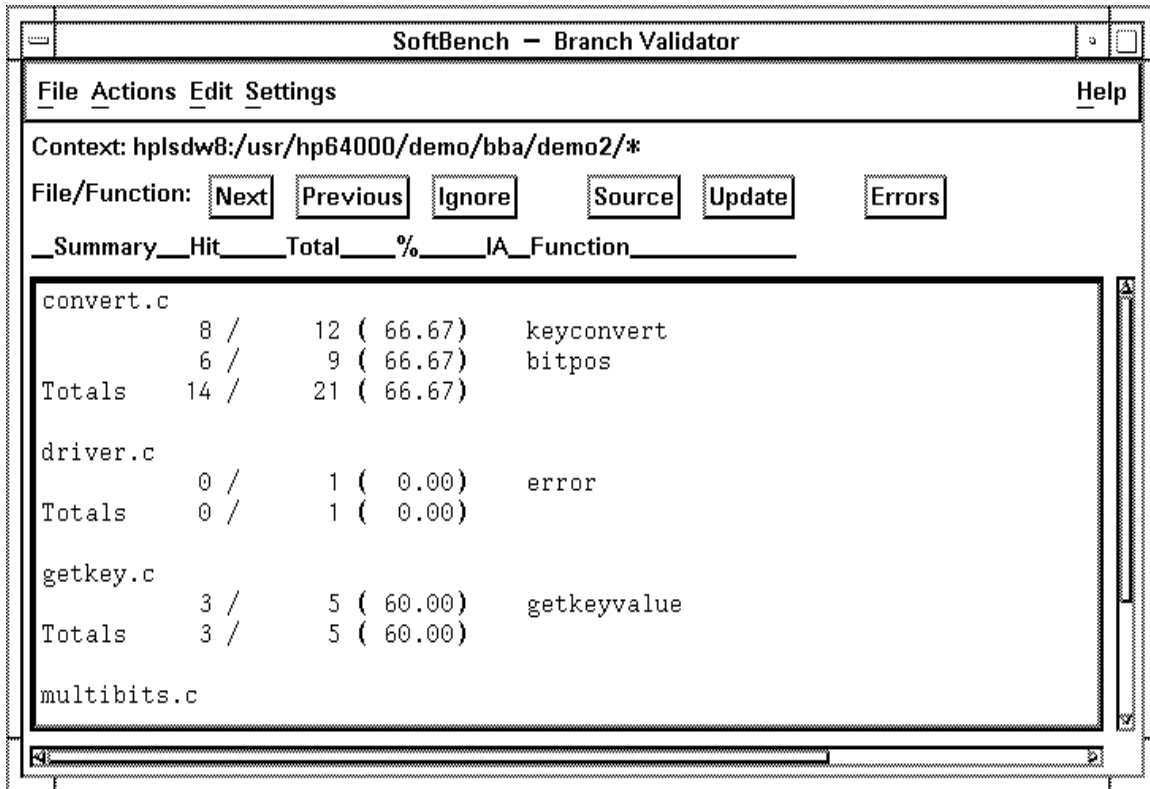


Figure 3-1. The Main BBA Window

Note 

On HP-UX systems, the <Alt> key is the <Extend char> key.
 On Sun SPARCstations, the <Alt> key is the diamond-shaped key to the left of the space bar.

For example, pressing <Alt> F will open the File pull down menu. Each item in the pull down menu also has a mnemonic. To activate an item in the pull down menu, press the designated key. For example, press "h" to display a histogram in the test report area. Be sure to release the <Alt> key before pressing the new key ("h" in this example).

To open a different pull down menu category, press <Alt><Key> where <Key> is the mnemonic character for the menu category.

To close a pull down menu without making a selection, repeat the <Alt><Key> you used to open it. For example, <Alt> F opens the File pull down menu, and <Alt> F will close it.



Menu Item Accelerators

Some pull down menu items also have key accelerators associated with them. These key accelerators allow the action for a menu item to be obtained without going through the pull down menu system. The key sequence for all accelerators is <Shift><Alt><Key>. Later paragraphs in this chapter show how to use some of the key accelerators you may use most often.

Customizing Mnemonics and Accelerators

Mnemonics and accelerators can be customized by adding resource entries to your .Xdefaults file. The file `$HP64000/lib/X11/app-defaults/BBA` contains the default settings. It can be copied or appended to your `$HOME/.Xdefaults` and modified. If BBA is running, it must be restarted before the new resource settings will be recognized.

Displaying The Test Reports

Use the commands described below to obtain test reports available in the HP Branch Validator SoftBench User Interface.

Note



Use the *command select* mouse button to click on selections in the pull down menus.

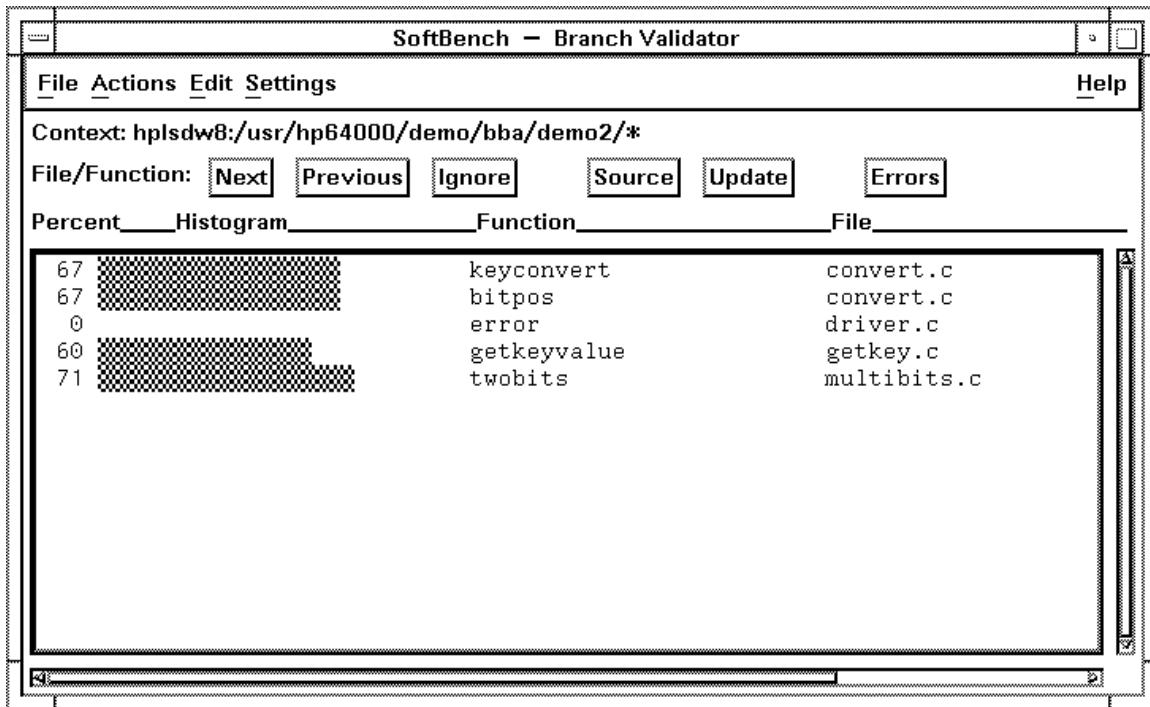


Figure 3-2. BBA Histogram

Details Of The Histogram Display

Choose **File**→**Show Histogram**. The test report area will show a histogram of the branch coverage of the functions tested. See figure 3-2.

This display shows the BBA function coverage, and each graphic bar represents the percent of the branches that were hit (executed) in the associated function during the test. The histogram shows the following:

- Numerical percent of branch coverage.
- Histogram bars representing percent of branches hit.
- Name of function, and name of file.

Details Of The Summary Report

Choose **File**→**Show Summary**. The test report area will show a file and function summary of BBA test coverage. See figure 3-3. The following specific information is shown in this report:

1. Name of file.
2. On a function-by-function basis:
 - a. Number of retained branches hit (executed).
 - b. Total number of retained branches in function.
 - c. Percent of retained branches that were hit.
 - d. Existence of "ignores" in function.
 - e. Existence of "alerts" in function.
 - f. Name of function.
3. For each file, the total branch-test values for all retained functions in the file.

Note



Retained branches are the branches remaining after removal of the branches designated to be ignored or excluded from the analysis. Retained functions are the functions remaining after removal of the functions designated to be ignored or excluded from the analysis.

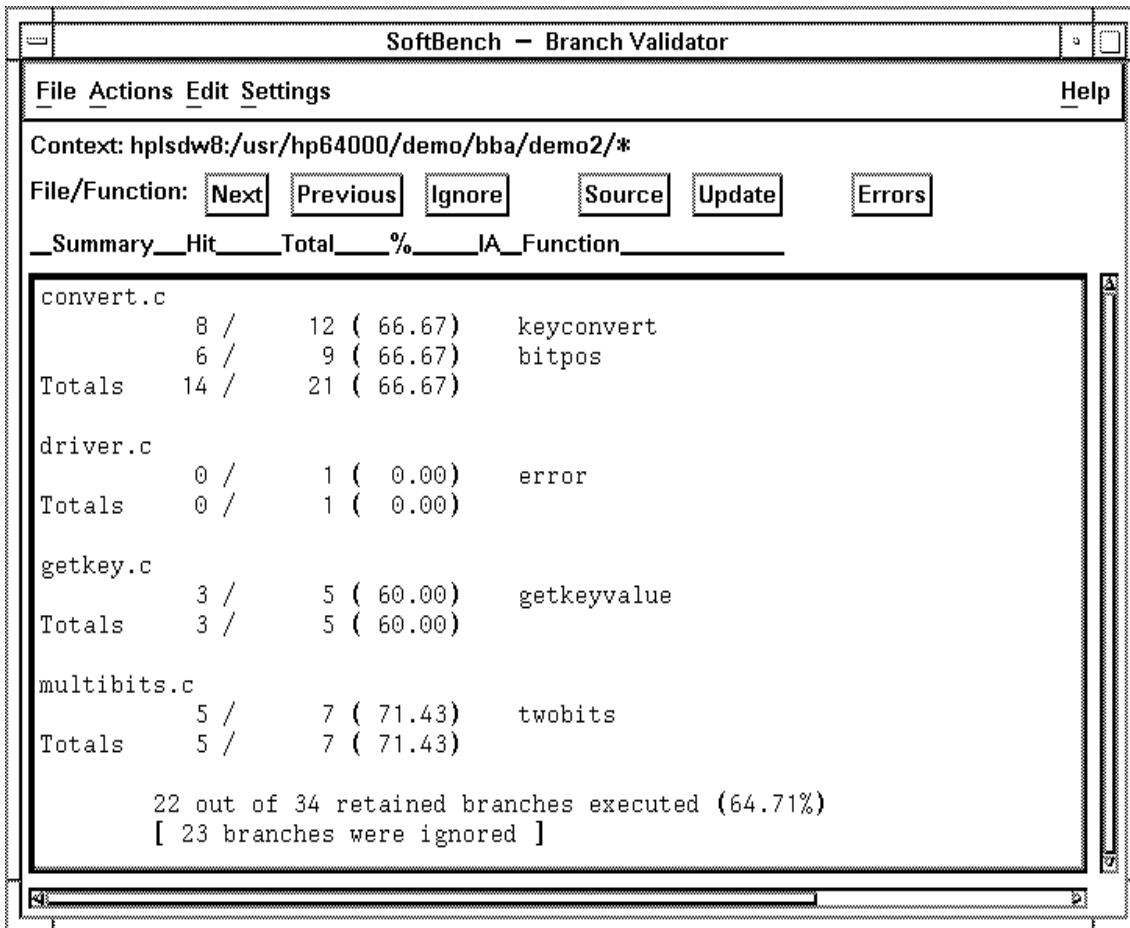


Figure 3-3. BBA Summary

Details Of The Results Only Display

Choose **File**→**Show Results Only**. The test report area will show a summary of the branch coverage during the BBA tests. See figure 3-4. The information shown in the Results Only report is listed below:

- Number of files tested.
- Number of functions tested.
- Number of all branches in the files and functions.
- Number and percent of retained branches that were hit (values exclude ignored branches).
- Number of branches ignored.

Note



Number of branches is equal to total retained branches plus total ignored branches (e.g. 57 branches = 34 retained branches + 23 ignored branches).

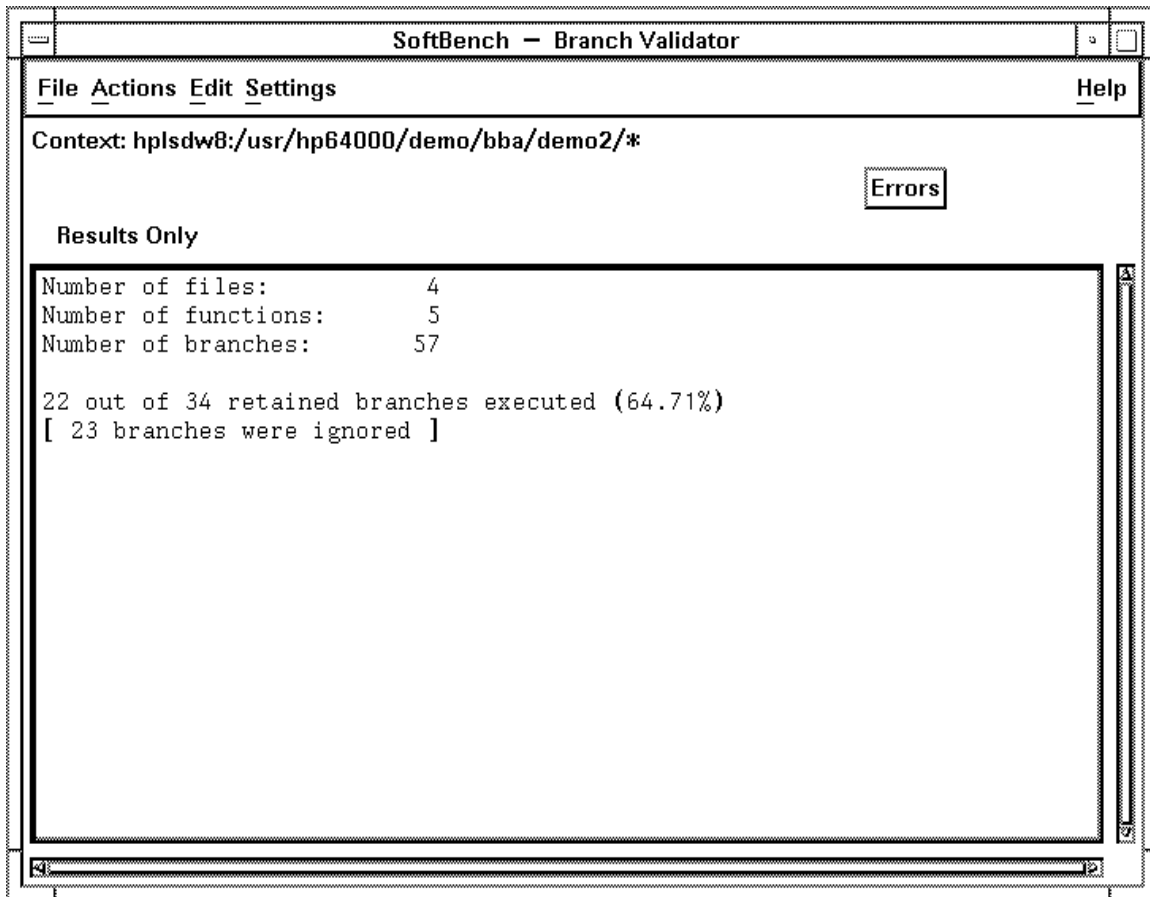


Figure 3-4. BBA Results Only

Details Of The File History Report

Choose **File**→**Show File History**. The test report area will show a list of all of the files that were tested, the dates when the files were last compiled, and the "bbacpp" preprocessor options with which the files were compiled. See figure 3-5. This command is identical to the BBA report command "bbarep -Dft". It shows the following information:

- File names.
- File modification dates.
- BBA preprocessor options with which the files were compiled.
- Number of unloads within the dumpfile.
- Full path name of each file.
- Total number of files and functions compiled.
- Total number of branches.

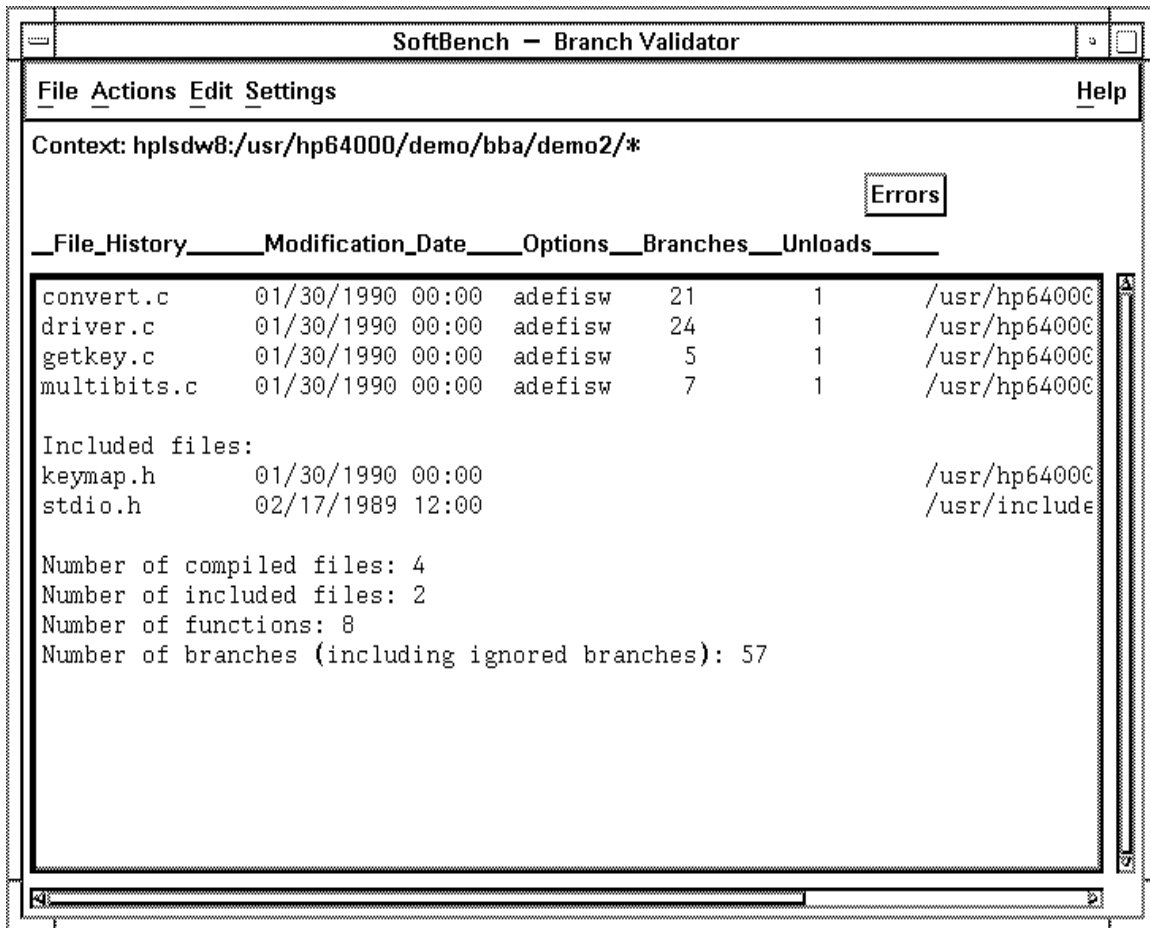


Figure 3-5. BBA File History Report

Ignoring A File

Return again to the Histogram display. As a shortcut, type <Shift><Alt>H.

You may remember that during the getting started procedures in Chapter 2, Frank wanted to ignore the function named **error** in the **driver.c** file. One method to ignore this function would be to add "error" to the ignore file. Let's do that in this interface.

Move the cursor to the **error** line on the report and click the *command select* mouse button. This selects the **error** function (the line containing **error** is highlighted).

Move the cursor to the "Ignore" command button and click the command select mouse button. This causes BBA to ignore the indicated file (or function).

To see a new report with the **error** function ignored, click on the "Update" command button. The new report will ignore the error function.

Displaying The Source Of A File Or Function With Unexecuted Branches Identified

The normal use of BBA test results consists of scanning the list of files or functions in the Summary or Histogram reports until you find a function with a test coverage that is too low. After selecting the low-coverage function, you can view its associated source file and see its unexecuted branches by clicking on the "Source" command button. The selected source file is displayed in a new window with the unexecuted branches highlighted. See figure 3-6. To see how this works, enter the following commands:

Choose **File**→**Show Summary** or type <Shift><Alt>S.

Select the **convert.c** file in the report by moving the cursor into the **convert.c** line and clicking the command select mouse button.

Now move the cursor to the "Source" command button and click the command select mouse button, or simply double click on **convert.c** to open the source window.

A second window opens (figure 3-6) and displays the **convert.c** source file. The first unexecuted branch in **convert.c** is highlighted.

In the source window, click on the "Next" button. The source display will scroll to the next unexecuted branch in the source file.

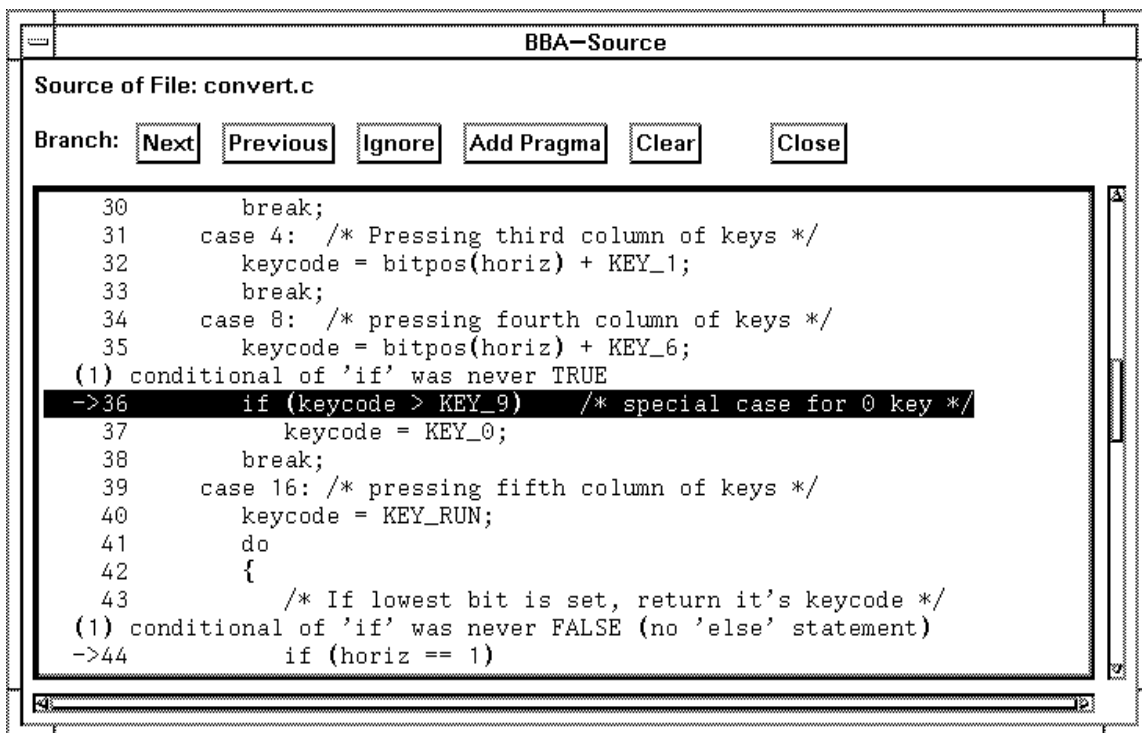


Figure 3-6. Source Of Selection Window

In the source window, click on the "Previous" button. The source display will scroll back to the previous unexecuted branch in the source file.

In the main Branch Validator window, click on the "Next" button. The source file for the next file or function in the Summary report will appear in the source window.

In the main Branch Validator window, click on "Previous" to see the previous source file or function in the Summary report.

In the main Branch Validator window, you can click on any file or function in the test report area and see its source file appear in the source window.

Ignoring A Branch

In the main Branch Validator window, select the function **bitpos**. Its source file will appear in the source window.

In the getting started procedures of Chapter 2, Frank needed to ignore the three cases of this routine that were used by another programmer. Here is an easy way he could have ignored these three branches if he had been using this interface.

With the first branch highlighted in the source window (case 32), click on the "Ignore" button.

The highlighting advances to the next unexecuted branch in the file.

Click on the "Ignore" button two more times to ignore all three branches.

Adding A Pragma

With the function bitpos still selected and its source file still on screen, the following paragraphs will show you how to remove the "ignore" designations and add BBA_IGNORE pragmas to the three branches.

Frank might have decided to add BBA_IGNORE pragmas to the last three branches in the bitpos routine if he had not heard that his fellow worker would be using the routine, also. (A BBA_IGNORE pragma is a compiler directive that prevents bbacpp from instrumenting the branch for branch analysis. Refer to the paragraph titled "What is a pragma?" in Chapter 4 for more information.)

First remove the ignore designations from the ignore file by selecting the first ignored branch in the source file "case 32" using the "Previous" command button. Click on "Clear" to remove the ignore designation.

The next ignored branch will be highlighted in the source window. Click on "Clear" again.

Click on "Clear" one more time. This removes the ignore designation from the last of the three branches.

Now select (highlight) the first branch to be ignored and click on "Add Pragma".

The highlighting advances to the next branch.

Click on "Add Pragma" two more times. This adds pragmas to all three branches. The P^ indicates that the pragma will appear above this line in the source file.

The pragma is not actually inserted into your source code at this point. To get the pragmas inserted into your source code, you will have to go to the Actions pull down and click on Add Pragmas to Source Files. Thus, you can use "Add Pragma" and "Clear" as many times as you want and your source file will be unaffected unless you use the Actions pull down and click on Add Pragmas to Source Files.

Once you have added the pragmas to your source file, you can recompile and link your file, and rerun your test sequence.

Printing And Saving The Results

You can print the four displays of measurement results, a listing of the present content of a selected source file, or a list of active files and functions. For example, choose **File→Print→Summary**.

The summary file is passed as a parameter to the command defined by `BBA_PRINTER_COMMAND`. As a default, the `BBA_PRINTER_COMMAND` pipes the summary file to `lp`. (`lp` can be modified as discussed in Chapter 10 under the heading, "Using Print And Save.")

You can save in a file the four displays of measurement results, a listing of the present content of a selected source file, or a list of active files and functions. For example, choose **File→Save in file→Histogram**.

A new window will open. In it, you can specify the name of the file where you want your histogram saved. When you enter the file name and click OK, the content of your histogram will be appended to the specified file.

How To Exit The HP Branch Validator (BBA) SoftBench User Interface

Choose **File**→**Quit** or type <Shift><Alt>Q.

Note



You can quit BBA at any time and later resume your analysis and continue right where you left off. The branches you marked to be ignored or in which you entered pragmas are retained in files in your home directory. When you restart the BBA process, the contents of those files will be used to generate your BBA test reports.

Notes



Details Of bbacpp

Introduction

Chapter 1 briefly explained the purpose of bbacpp. This chapter will explain how to invoke bbacpp, the options available to bbacpp, the effect bbacpp has on your code, and how and why to insert BBA_IGNORE, BBA_IGNORE_ALWAYS_EXECUTED, and BBA_ALERT pragmas into your source code.

BBACPP QUICK REFERENCE

HP AxLS C Cross Compilers

Bbacpp is invoked by using the "-b" command-line option to "cc<COMP>" (where <COMP> = your specific compiler number; e.g., **cc68000**, **cc8086**, **cc68030**, etc).

In addition to the options that cpp<COMP> will accept (e.g., -P, -C, -I<directory>, etc), bbacpp<COMP> also accepts the options listed in this quick reference figure. (See the next two pages.)

MRI C/C++ Cross Compilers

Bbacpp is invoked by replacing your normal compile command with a script. As an example, replace **mcc68k** with **mcc68kbba**. If you are using a makefile, this can typically be done by changing **CC=mcc68k** to **CC=mcc68kbba**. In addition, the following scripts are available **mcc86bba**, **mcc960bba**, and **ccc68kbba**. These scripts accept most of the same options as the normal compiler, with a few exceptions. Refer to the man pages for complete details.

Figure 4-1. Quick Reference To "bbacpp"

BBACPP QUICK REFERENCE (Cont'd)

-DBBA_OPTO=<insert_options>

<insert_options>:

A - turn on all of the following options.

a - insert code to tell if the true and false expressions of a conditional assignment statement were ever executed.

d - insert code to tell if the conditional of a 'do' statement was evaluated as true.

e - insert code to tell if empty 'case' statements (i.e., those with no statements or only a "break" statement) were ever executed.

i - insert code to tell if an 'if' statement that has no 'else' was never evaluated as false.

s - insert a 'default' statement for 'switch' statements that do not have a 'default'.

f - insert code to tell if the third expression in a 'for' statement was ever executed.

w - insert code to tell if the body of a 'while' statement was never skipped.

Figure 4-1. Quick Reference To "bbacpp" (Cont'd)

BBACPP QUICK REFERENCE (Cont'd)

-DBBA_OPTc<constname>

Place the constant data that bbacpp creates in a CONST SECTION named <constname>. The default is to place the constant data into the CONST SECTION that is valid at the end of the file. This option is not available when using an MRI compiler.

-DBBA_OPTd<dataname>

Place the array that bbacpp creates in a DATA SECTION named <dataname>. The default is to place the array into the DATA SECTION that is valid at the beginning of the file. This option is not available when using an MRI compiler.

-DBBA_OPTM<character>

Use <character> as the suffix for the map file (instead of "M").

-DBBA_OPTp<lines>

Increase the amount of "push-back-line" memory.

-DBBA_OPTS

Do not generate a mapping file.

Figure 4-1. Quick Reference To "bbacpp" (Cont'd)

What bbacpp Does

Bbacpp prepares a C source file to have branch analysis data generated when it is executed. It also creates a "map file" that is used by bbarep to associate branches with the source statements that created the branches.

To do this, bbacpp performs the following:

1. Creates an array with one entry for each branch. Each entry in the array is initialized to 0, which is the default for the C language.
2. For each branch that is detected:
 - a. An array entry is assigned to the branch.
 - b. An assignment statement is inserted as the first executable statement within the branch. (The assignment statement sets the associated array entry to 1, showing that it was executed. Also, the assignment statement is inserted on the same line as the first executable statement so that emulators and debuggers can report the correct line numbers of the source statements.)
 - c. An entry in the map file is created for that array entry. The entry specifies what type of branch it was (e.g., an if statement or a for loop).
3. At the end of the newly created source file, an "environment" data area is created. This data prevents mismatching of mapping files with data, and allows changes in the source file to be monitored. The data area specifies (among other things):
 - a. Which **-DBBA_OPTO=** options were used.
 - b. The number of branches detected in this file.
 - c. The suffix of the map file.
 - d. The date that the source file was last modified.

The BBA unload routine (**File**→**Store**→**BBA_Data**) will use a symbol database to locate the array and the environment data area, then use the emulator or debugger to transfer the values in the array and data area to your host's disk. The symbol database is created automatically when you load your program into the emulator or debugger.

Amount Of Code Added

The "environment" data (step 3 in the preceding paragraph) adds a little more than 20 bytes to your absolute file. The exact number of bytes depends primarily upon the length of the full path name of the current file.

The "per-branch" data (step 2 in the preceding paragraph) adds between five and eight bytes (depending on optimization and register usage), plus one byte in the data section (the array, itself).

Experiments have shown total code expansion anywhere between 1% and 100%, depending on coding practices.

Example Of bbacpp's Operation On A Simple Source File

Figure 4-2 shows a simple C source file. Figure 4-3 shows that same source file after it has been tooled by bbacpp. Line numbers have been inserted on the left-hand side for use in the discussion that follows.

```
1  extern int goodbye;
2
3  yousay(hello)
4  int hello;
5  {
6      if (hello == goodbye)
7          goodbye = hello + 3;
8      else
9          goodbye = hello - 5;
10 }
```

Figure 4-2. Simple C Source File

Explanation Of Lines In Figure 4-3

Lines 1 and 5 are lines that C preprocessors generate for the compiler. They identify the file and line where the text originated (so that **include** files can be tracked correctly).

Line 2 sets the SECTION for the branch data array to be in section **bbadata**. For this to have happened, the **-DBBA_OPTdbbadata** option was used with **bbacpp**. Refer to the paragraph that discusses the details of the **-DBBA_OPTd<dataname>** OPTION appearing later in this chapter for more information. If no **-DBBA_OPTd<dataname>** option had been used, this line would not be present.

Line 3 is the declaration of the array where the branch data is kept. Note that the bizarre name (**_bA_array**) is unlikely to conflict with variables you use. In fact, all variables that **bbacpp** creates begin with the characters **_bA_**.

Line 4 sets the SECTION information back to what it was prior to line 2. If no **-DBBA_OPTd<dataname>** option had been used, this line would not be present.

Note



MRI compilers do not support options to change the data and constant sections. Therefore, program SECTION directives are not inserted.

```

1 # 1 "t.c"
2 #pragma SECTION DATA=bbadata CONST=bbaconst
3 static unsigned char _bA_array[3      ] = {0};
4 #pragma SECTION UNDO
5 # 1 "t.c"
6 extern int goodbye;
7
8 yousay(hello)
9 int hello;
10 {
11     _bA_array[0]=1;if (hello == goodbye)
12     {_bA_array[1]=1;goodbye = hello + 3;}
13     else
14     {_bA_array[2]=1;goodbye = hello - 5;}
15 }
16 #pragma SECTION DATA=bbadata CONST=bbaconst
17 struct _bA_probe_struct_ {
18     unsigned char insertprotocol;
19     char mapsuffix;
20     char sourcemodtime[9];
21     unsigned char options[4];
22     int numentries;
23 };
24 const struct _bA_probe_struct_ _bA_t_nam_crs_abb_ph_ =
25 {
26     6,
27     'M',
28     {'B', 'w', 'x', '\'', 'q', 'j', '\'', 'v', '\'},
29     {0xa7, 0x05, 0x02, 0x00},
30     3,
31 };
31 #pragma SECTION UNDO

```

Figure 4-3. Simple C Source File Tooled By bbacpp

Lines 6 thru 10 are exact copies of lines 1 thru 5 in the source file.

Line 11 contains the first executable statement of the function **yousay**. Therefore, an array assignment expression is inserted. The array entry for this function is 0, something you don't care about, but bbarep does.

Line 12 shows that another array assignment was generated. Note that bbacpp has inserted "{" and "}" correctly so that no logical change to the code was made.

Line 13 is a copy of line 8 in the source file.

Line 14 is another array assignment.

Line 15 is the last line of the source file.

Details Of bbacpp 4-7

Line 16 sets the SECTION for the following constant definition to be **bbconst**. Refer to the paragraph that discusses the details of the **-DBBA_OPTc<constname>** OPTION appearing later in this chapter, for more information. If no **-DBBA_OPTc<constname>** option had been used, this line would not be present.

Lines 17 thru 30 were inserted by bbacpp to define the data area that keeps track of the environment that the file was compiled under. The name defined on line 24 (**_bA_t_nam_crs_abb_ph_**) will be unique for each file (it has to be, because it is global), and will always start with **_bA_** so it will not conflict with any of your symbols.

Line 31 sets the SECTION information back to what it was prior to line 16. This is not strictly necessary, but clean, in case you have additional processing you wish to do.

How To Invoke bbacpp

To use bbacpp with your AxLS C compiler, you simply add the **-b** option to your **cc<COMP>** command line (where **<COMP>** is your specific compiler number; e.g., **68000**, **8086**, **68030**, etc.).

This tells **cc<COMP>** to replace the normal C preprocessor (**cpp<COMP>**) with the BBA preprocessor that is specific for your compiler (**bbacpp<COMP>**). This preprocessor has the same predefined variables (e.g. **__<COMP>**) and search paths (e.g., **\$HP64000/include/<COMP>**) as **cpp<COMP>**, but adds the BBA capabilities.

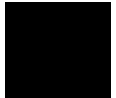
For example, if you normally compile a program by use of a command such as:

```
$ cc<COMP> -c myfile.c
```

you can modify your command to use bbacpp by adding the **-b** option, as follows:

```
$ cc<COMP> -c -b myfile.c
```

To use bbacpp with your MRI compiler, use the appropriate script command in place of your normal compile command. For example, if you normally compile a program by using a command like **mcc68k -c myfile.c**, you can modify your command to use bbacpp with a modification like **mcc68kbba -c myfile.c**



Relocating The BBA Constants (-DBBA_OPTc <constname> Option)

Bbacpp generates a constant data structure for each file (refer to DETAILS OF HOW BBACPP MODIFIES YOUR CODE later in this chapter for more information on this structure).

Normally this data structure is placed in the same section that contains the rest of your constants.

You can override placement of this data structure in the normal section by using the -DBBA_OPTc<constname> option. There must be no space between the c and the constant name.

This will allow you to separate the constants that bbacpp generated from the constants you generate.

Normally, the <constname> section is placed after the "const" section. You can also use a linker command file to place the sections where you want them. For example, if you usually use a command like the following:

```
$ cc<COMP> -b blue.c
```

you can force bbacpp to put the data structure for **blue.c** in the section **myconst** by using the command:

```
$ cc<COMP> -b -DBBA_OPTcmyconst blue.c
```

Note



Due to limitations in the file formats, the use of -DBBA_OPTc<constname> option to BBA does not make sense when using the cc<COMP> -h flag. For more information, refer to the paragraph titled PITFALLS WITH BBACPP AND CC<COMP> at the end of this chapter.

This option is not available with MRI compilers.

Relocating The BBA Data Array (-DBBA_OPTd <dataname> Option

Bbacpp normally places the branch analysis data (the array) in the same data section with the rest of your data.

You can override placement of the array in the normal section by using the **-DBBA_OPTd<dataname>** option. There must be no space between the **d** and the data name.

This will allow you to separate the data that bbacpp generated from the data that you generated.

Normally, the **<dataname>** section is placed after the "data" section. You can also use a linker command file to place the sections anywhere else that you want them. For example, if you usually use a command such as:

```
$ cc<COMP> -b green.c
```

you can force bbacpp to put the data structure for **green.c** in the section named **mydata** by using the command:

```
$ cc<COMP> -b -DBBA_OPTdmydata green.c
```

Note



Due to limitations in the file formats, use of the **-DBBA_OPTd<dataname>** option to BBA does not make sense when using the **cc<COMP> -h** flag. For more information, refer to the paragraph titled PITFALLS WITH BBACPP AND CC<COMP> at the end of this chapter.

This option is not available with MRI compilers.

Details Of How bbacpp Modifies Your Code, And The -DBBA_OPTO= <options> Options

This part of the manual explains which branches are detected and instrumented by using different **-DBBA_OPTO=<options>** options. (The term "instrumented" means that the branch is detected, and an array assignment statement is inserted. Examples in the following subparagraphs show how the code is instrumented by using different options to this command.

Normally, the option **-DBBA_OPTO=A** is used. It requests bbacpp to detect all branches (therefore, bbarep reports on all branches). However, you may not wish to recognize certain branches based on previous experience or lack of microprocessor address space. If this is the case, you can still obtain the equivalent of the **-DBBA_OPTO=A** option by making multiple compilations and test-set executions (refer to the paragraph that discusses HOW TO COMBINE -DBBA_OPTO= OPTIONS later in this chapter for more information). To use any (or all) of the **-DBBA_OPTO=<options>**, simply add the options you want on the same command line that contains the **-b** option.

Examples:

```
$CC -b -DBBA_OPTO=iaf -c hello.c
```

```
$CC -b -DBBA_OPTO=A -c hello.c
```

The examples in the following subparagraphs refer to the sample function in figure 4-4. Line numbers have been added on the left-hand side of the sample function for reference in the discussions that follow.

```

1  /* Sample function showing many types of branches */
2  sample()
3  {
4      int a, b, c;
5
6      /* if statement with an else */
7      if (a == 0)
8          b++;
9      else
10         c++;
11
12     /* if statement with no else */
13     if (b == 0)
14         b++;
15
16     /* Switch statement with empty case, a case with fall-thru
17        execution, and a default */
18     switch(a)
19     {
20     case 0:
21     case 1:  b++;
22             break;
23     default: b++;
24     }
25
26     /* switch statement with no default */
27     switch(a)
28     {
29     case 0:  b++;      /* non-empty, no fall-thru */
30             break;
31             /* no default */
32     }
33
34     /* while statement */
35     while(b != 0)
36         b--;
37
38     /* For loop */
39     for (c = 0; c < b; c++)
40         b--;
41
42     /* Conditional assignment */
43     c = (a == 0) ? 123 : 321;
44
45     /* do-while loop */
46     do
47         b++;
48     while (b != 0);
49 }

```

Figure 4-4. Sample Function

**Default
Instrumenting Of
Branches
(No -DBBA_OPTO=
<options>)**

When no **-DBBA_OPTO=<options>** are used, bbacpp detects the following:

1. **Functions**: An array assignment is added as the first executable statement of any function.
2. **if** statements: An array assignment is added as the first executable statement of the 'then' part of any **if** statement.
3. **else** statements: An array assignment is added as the first executable statement of the 'else' part of any **if** statement for which you wrote the **else**.
4. **while** statements: An array assignment is added as the first executable statement within any **while** loop.
5. **for** statements: An array assignment is added as the first executable statement within any **for** loop.
6. **case** and **default** statements: An array assignment is added as the first executable statement following a **case** or **default** within a **switch** statement.

(To understand the purpose of the "array assignments" described above, refer to the paragraphs entitled, "WHAT BBACPP DOES" and, "EXAMPLE OF BBACPP'S OPERATION ON A SIMPLE SOURCE FILE", earlier in this chapter.)

Running the sample through bbacpp with no **-DBBA_OPTO=** options (but with the **-C** option so that comments are retained) will produce a listing as shown in figure 4-5. We removed the **_bA_array** declaration and the file's data structure from the top and bottom to emphasize the **_bA_array** assignment statement insertions.

Notes On Figure 4-5

1. On line 7, the array assignment was inserted because the **if (a == 0)** statement is the first statement of the function. The first statement of any function is always preceded with an array assignment so the BBA can tell if the function was ever called.
2. An insertion was done on lines 8 and 14 to reflect the **if** statements.
3. An insertion was done on line 10 because the example program had an **else** statement. No **else** insertion was done for the **if** statement on line 13 because the example program did not have an "else" (refer to the explanation of the **-DBBA_OPTO=i** option later in this chapter).
4. Insertions were done on lines 21, 23, and 29 in response to code after **case** or **default** statements. Note that on line 21, `array[4]` will be set even if **a** is 0. Refer to the discussion of **-DBBA_OPTO=e** later in this chapter). Also note that if the switch statement at line 27 was entered with "a not equal to 0", no notice would be given; however, refer to the discussion of **-DBBA_OPTO=s** later in this chapter.
5. An insertion was done on line 36 because that was the first statement of a while loop.
6. An insertion was done on line 40 because that was the first statement of a for loop.

The **-DBBA_OPTO=** options add to the branches that are detected. The examples that follow will only show differences caused by adding the options.

```

1  /* Sample function showing many types of branches */
2  sample()
3  {
4      int a, b, c;
5
6      /* if statement with an else */
7      _bA_array[0]=1;if (a == 0)
8          {_bA_array[1]=1;b++;}
9      else
10         {_bA_array[2]=1;c++;}
11
12     /* if statement with no else */
13     if (b == 0)
14         {_bA_array[3]=1;b++;}
15
16     /* Switch statement with empty case, a case with fall-thru
17        execution, and a default */
18     switch(a)
19     {
20     case 0:
21     case 1: _bA_array[4]=1;b++;
22             break;
23     default: _bA_array[5]=1;b++;
24     }
25
26     /* switch statement with no default */
27     switch(a)
28     {
29     case 0: _bA_array[6]=1;b++;      /* non-empty, no fall-thru */
30             break;
31             /* no default */
32     }
33
34     /* while statement */
35     while(b != 0)
36         {_bA_array[7]=1;b--;}
37
38     /* For loop */
39     for (c = 0; c < b; c++)
40         {_bA_array[8]=1;b--;}
41
42     /* Conditional assignment */
43     c = (a == 0) ? 123 : 321;
44
45     /* do-while loop */
46     do
47         b++;
48     while (b != 0);
49 }

```

Figure 4-5. Results Of bbacpp -C

4-16 Details Of bbacpp

Instrumenting Conditional Assignments (-DBBA_OPTO=a Option)

This option will cause bbacpp to detect and instrument conditional assignments, that is, code will be inserted to detect whether or not the condition in the conditional assignment was ever TRUE and was ever FALSE. By adding this option to the sample program, line 43 changes as shown in figure 4-6.

When you have used this option, bbarep will automatically report whether or not the condition (in this example, the `(a == 0)` expression) was never TRUE or never FALSE.

Note



When you use this option, you may receive warnings from the compiler about "illegal combination of pointer and integer". This happens when the left-hand side (lhs) of the conditional assignment is a pointer, and either the TRUE or FALSE expression was 0 or NULL. For example, you will receive the warning for the following code fragment:

```
char *ptr1, *ptr2;  
ptr1 = (a == 0) ? ptr2 : NULL;
```

To avoid this warning, typecast the 0 or NULL to be the same type as the lhs. For example:

```
char *ptr1, *ptr2;  
ptr1 = (a == 0) ? ptr2 : (char *) NULL;
```

```
42     /* Conditional assignment */  
43     c = (a == 0) ? (_bA_array[9]=1,123) : (_bA_array[10]=1,321);  
44
```

Figure 4-6. Results Of bbacpp -C -DBBA_OPTO=a

```

45     /* do-while loop */
46     do
47         b++;
48     while ((b != 0)?(_bA_array[9]=1,1):(_bA_array[10]=1,0));

```

Figure 4-7. Results Of `bbacpp -C -DBBA_OPTO=d`

Instrumenting Do-While Statements (-DBBA_OPTO=d Option)

This option is used to instrument do-while statements. Obviously, the code in a do-while is always executed (at least once). This adds code to determine if the do-while is ever executed more than once. Adding this option to the sample program changes lines 46 thru 48 as shown in figure 4-7.

Note that if there was a break within the do loop that was always being executed, `bbarep` would also report that the conditional of the while was never FALSE.

Instrumenting Case Statements (-DBBA_OPTO=e Option)

This option actually does several things to case statements:

1. If a **case** or **default** statement has no code, this will instrument the case statements so that the BBA can tell when the **switch** statement went to that case. In our example function, line 20 has a **case** with no code.
2. If a **case** or **default** statement is not preceded by a **break** statement, code is inserted so that you can tell if the **switch** statement did not go to that case, but the case's code was executed because the previous code "fell through" and executed the code. For example, the `b++`; on line 21 of the sample function will be executed even if `a` is never 1 (but is ever a 0).

Figure 4-8 shows an example of the inserted code when the `-DBBA_OPTO=e` option is used:


```
16     /* Switch statement with empty case, a case with fall-thru
17     execution, and a default */
18     switch(a)
19     {
20     case 0:
21     _bA_array[4]=1; goto _bA_switch_label_1_1;case 1:
22         _bA_array[5]=1;_bA_switch_label_1_1:b++;
23         break;
24     default: _bA_array[6]=1;_bA_switch_label_1_2:b++;
25     }
```

Figure 4-8. Results Of bbacpp -C -DBBA_OPTO=e

Note that if the switch is entered with a value of 0, array[4] will be set, but array[5] will not be set, even though the code associated with **case 1** will still be executed because of fall through.

```

12     /* if statement with no else */
13     if (b == 0)
14         {_bA_array[3]=1;b++;}else _bA_array[4]=1;
15

```

Figure 4-9. Results Of bbacpp -C -DBBA_OPTO=i

Instrumenting An If With No Else (-DBBA_OPTO=i Option)

This option allows you to detect when an **if** statement that has no **else** is always TRUE. It does this by inserting an **else** statement and an array assignment. Then bbarep checks to see if the array value is 0, and if it is, informs you that the conditional of the **if** was never FALSE.

For example, using the **-DBBA_OPTO=i** option with our sample function, the code is modified as shown in figure 4-9.

Instrumenting A Switch With No Default (-DBBA_OPTO=s Option)

This option is used to determine if a **switch** that has no **default** was entered with a value that caused none of the **case** statements to be executed. It creates a **default** statement and inserts an assignment statement. By using this option, our example function will be modified as shown in figure 4-10. As you can see, a **default** and associated array assignment statement have been added at line 32, at the end of the **switch** statement on line 27.

```

26     /* switch statement with no default */
27     switch(a)
28     {
29         case 0: _bA_array[6]=1;b++; /* non-empty, no fall-thru */
30             break;
31         /* no default */
32     default:_bA_array[7]=1; }

```

Figure 4-10. Results Of bbacpp -C -DBBA_OPTO=s

Instrumenting The Third Expression In A For Statement (-DBBA_OPTO=f Option)

This option is used to tell if the third expression in a **for** statement is ever executed. By default, bbacpp inserts statements to tell if the code within a **for** loop was ever executed, but it is possible for the code within a **for** loop to be executed and the third expression to never be executed (for example, this could be caused by a **break** or a **return** that is always executed in the code controlled by the **for**).

Using **-DBBA_OPTO=f** to tool the sample function yields code as shown in figure 4-11.

In the example of figure 4-11, the `c++` is the third expression of the **for** on line 39.

```
38     /* For loop */
39     for (c = 0; c < b; _bA_array[8]=1,c++)
40         {_bA_array[9]=1;b--;}
41
```

Figure 4-11. Results Of bbacpp -C -DBBA_OPTO=f

```

34  /* while statement */
35  _bA_temp_[0]=0;while(b != 0)
36  {_bA_array[7]=1;_bA_temp_[0]=1;b--;} if(_bA_temp_[0]==0)_bA_array[8]=1;
37

```

Figure 4-12. Results Of bbacpp -C -DBBA_OPTO=w

**Detecting A While
Loop Always
Executed
(-DBBA_OPTO=w
Option)**

This option allows you to detect when the body of a **while** loop is always executed. It does this by adding a "temporary" variable, setting it to "0" prior to the loop, and then after the loop, checking to see if the temporary variable is still 0. This is shown in figure 4-12.

Note



The **_bA_temp_** array is created only if needed (which is why you haven't seen it in this manual before).

**Using All
-DBBA-OPTO
Options Together
(-DBBA_OPTO=A
Option)**

The **-DBBA_OPTO=A** option implies all of the previous **-DBBA_OPTO=** options. Therefore, it is identical to **-DBBA_OPTO=adeisfw**.

How (And Why) To Combine -DBBA_OPTO= Options

Generally, you will want to use the **-DBBA_OPTO=A** option. However, there are at least two occasions when you may not want to use it:

1. Your organization chooses not to (for example, your organization may want to shorten testing time, recognizing the trade-off in test quality).
2. Your system does not have enough memory to load the entire program in memory with all the branch-analysis options enabled (**-DBBA_OPTO=A**).

For the first case, just choose the options you want, and that's the end of it.

The second case only applies when using an emulator. You can usually use the memory mapping in the emulator to expand your system's memory by placing some emulation memory after your system's memory. However, if you are constrained by your addressing space (i.e., your program just about fills the processor's addressing capability), there is another method.

In this case, compile all of your files with two or more **-DBBA_OPTO=** options. Load the program. Run the tests, and dump the data to "bbadump.data". Then do the compile-load-run-dump loop with different **-DBBA_OPTO=** options. When you run "bbarep" or "bbamerge", the branch data will automatically be "merged" to give you the same effect as if you had used all of the **-DBBA_OPTO=** options and had only run the tests one time.

For example, assume that when you used **-DBBA_OPTO=A**, you could not fit your program into your system's memory space, but when you used the options **-DBBA_OPTO=ade**, and **-DBBA_OPTO=isfw**, the program fit.

You would first compile all of your programs using

```
$CC -c -b -DBBA_OPTO=ade -DBBA_OPTM1 <C source files>
```

This enables the "ade" insertion options, and sets the extension of the map file to "1" instead of "M". The map file extension was changed so that when you compile with different options, you won't overwrite the mapping file created with the previous options. That would restrict the type of reports you could get with "bbarep".



Note



Details of how to use the -DBBA_OPTM option to change the map file extension are given under the next heading in this chapter.

Now run your tests and create the "bbadump.data" file with the **File→Store→BBA_Data** command.

Re-compile all of your programs using the command:

```
$ CC -c -b -DBBA_OPTO=isfw -DBBA_OPTM2 <C source files>
```

This enables the "isfw" insertion options, and sets the mapping file extension to "2" instead of "M".

Run your tests, and use the **File→Store→BBA_Data** command (which always appends the data to the previous dump, so you don't lose previous data). When you run "bbarep", you will get the same results as if you had been able to use the -DBBA_OPTO=A option.

Note



The process described above will work with "bbarep", the command line report generator. It will not work for BBA in the HP SoftBench interface (where only ".M" map files can be used).

Two requirements must be met before the preceding example will work:

1. The modification dates of your source files must all be the same during each compilation of your program. Don't go in and modify your source files half-way through and expect to merge the results of all the tests!
2. Each time you compile tests with a new option, be sure to give your "mapping file" a different file extension (discussed next in this chapter). Otherwise, each time you invoke tests, your new (.M) map will overwrite the existing (.M) map, and the map information required by "bbarep" will only be able to show information from the most recent run of your executable file.



Changing The Map File Suffix (-DBBA_OPTM <character> Option)

This option allows you to override the default suffix that is used for the map file. Refer to the detailed discussion of THE -DBBA_OPTS OPTION appearing later in this chapter for information on the use of the map file, and information on how to suppress the creation of the map file, if necessary.

Typically, the name of the map file is the same as the name of the C source file, except that the file extension ".c" is replaced with ".M". If your source file name does not end with ".c", but does end with "<character>", the <character> is replaced with an "M". However, bbacpp will not overwrite a file unless it is either zero length or is a bbacpp map file (so don't worry that bbacpp may overwrite one of your significant files!).

The map file is always placed in the same directory as the C source file.

To change the map suffix from **M** to something else, add the **-DBBA_OPTM<character>** option to the same command line that has the **-b** option. There must be no space between the **-DBBA_OPTM** and the <character>.

There are several reasons why you might want to use the **-DBBA_OPTM<character>** option:

1. You already have files (that you want to keep) with the name of the C source file, but with the **.M** extension.
2. Your organization may reserve the **.M** extension for a particular type of file, and you don't want any exceptions to that rule.
3. You need to compile your program several times with different **-DBBA_OPTO=** options. When you do this, you must specify different map file extensions for each compilation. Refer to the discussion entitled "HOW (AND WHY) TO COMBINE **-DBBA_OPTO=** OPTIONS" appearing earlier in this chapter for reasons to use several compilations with different **-DBBA_OPTO=** options.

When the BBA unload command **File→Store→BBA_Data** unloads the data area associated with each file, it will store the map suffix (along with other information) into the "bbadump.data" file. Bbarep will look for map files with the specified suffix in order to generate reports. Refer to the next paragraph (the **-DBBA_OPTS** option), for the list of bbarep output options that require the map files.

Note



If you use the **-DBBA_OPTM** option, you will not be able to use BBA in the HP SoftBench Interface.

Suppressing Creation Of The Map File (-DBBA_OPTS Option)

This option allows you to prevent creation of the mapping file. If you use this option, some of the functions of the bbarep command will not work.

To use this option, simply add the **-DBBA_OPTS** option to the same command line that has the **-b** option.

You might use this option if you are concerned about limited disk space, or if you only care about over-all coverage percentage (and have no BBA_IGNORE or BBA_ALERT pragmas in your source code).

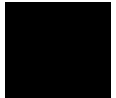
Note



If you use the **-DBBA_OPTS** option, you will not be able to use BBA in the HP SoftBench Interface.

Bbarep uses the map file(s) to:

1. Find out what type of branch was executed or not executed (used by the **-l**, **-aN** and **-bN** options to bbarep).
2. Determine the "scope" of a branch (so you don't get 500 reports of unexecuted branches, if a single function that had 500 branches in it was never executed). Used by the **-l**, **-aN** and **-bN** options to bbarep.
3. Determine if a branch had a `BBA_IGNORE` or `BBA_ALERT` pragma within it (used by all reporting options of bbarep).
4. Determine what function a branch is associated with (used by all reporting options of bbarep).
5. Determine which source lines generated the branch, and which source lines would be unexecuted if a branch were not taken. Used by the **-l**, **-aN** and **-bN** options to bbarep.



What Is A Pragma?

First, let's define "pragma" because it is not defined in the K&R book.

In ANSI-standard C, a "pragma" is defined to be the way to extend directives in an implementation-dependent manner. A pragma can go anywhere that a "#if" operator can go - i.e., the '#' must be the first non-whitespace on the line, and the entire line is generally not considered part of the C source file for grammatical purposes. Further, all implementations of C ignore unrecognized pragmas.

We have defined three pragmas for use with the BBA:

1. BBA_IGNORE.
2. BBA_IGNORE_ALWAYS_EXECUTED.
3. BBA_ALERT.

The names were chosen to make it highly unlikely that any other implementation of C would recognize these pragmas (thus, these should be ignored).

Detailed Explanation Of The BBA_IGNORE Pragma

The BBA_IGNORE pragma causes all branches to be ignored at, or within, the scope where the pragma is located - i.e., the reports generated by bbarep will not show whether or not the ignored branches were hit (although the number of ignored branches will be shown in a summary). As a special case, a BBA_IGNORE pragma that is outside of any function will act as if it were located within the next function (but only the next function; it will not cause any of the other functions from that point on to be ignored!).

In figure 4-13, lines 21 through 24 are ignored (i.e., if **a** was never zero, you would not see it in the number of unexecuted branches). Further, because the statement **if (j == 0)** is "within" the same scope as the BBA_IGNORE pragma, it is also ignored. The 'if' statement on line 26 is not within the same scope as line 22. Therefore, it will not be ignored (i.e., if **j** is never 5, you will see an unexecuted branch in the report).

```

20     if (a == 0)
21     {
22     #         pragma BBA_IGNORE
23         if (j == 0)
24             k++;
25     }
26     if (j == 5)
27     {
28         k--;
29     }
30     if (b == 5)
31     {
32         j++;
33     #     pragma BBA_IGNORE
34     }

```

Figure 4-13. Ignoring Lines 21 Thru 24 And 31 Thru 34

Lines 31 through 34 are also ignored because that is the scope of BBA_IGNORE, even though the pragma itself is toward the end of the scope.

In figure 4-14, the TRUE condition of **(j == 0) ?** is ignored while the FALSE condition is not ignored (because the FALSE part is at a different scoping level - see below). Note that the pragma can go anywhere a white space can go, but it must be on a line by itself. Because C always expands macros on a single line, no pragmas can be used in macros. (Specifically, you cannot ignore a portion of the branches generated by a macro; however, you can ignore all branches generated by a macro by using the **-i <ignorefile>** option to bbarep.)

Note that C comments are legal on the pragma line; however, they should not continue on to other lines.

```

945     k = (j == 0) ?
946 # pragma BBA_IGNORE /* It is ok to do this */
947         hello->dolly + 67 : 487 - well;

```

Figure 4-14. Ignoring The TRUE Condition

```

76     if (j == 4)
77     #     pragma BBA_IGNORE
78         drag_cursor(LEFT);
79     else
80         drag_cursor(RIGHT);

```

Figure 4-15. Ignore Pragma Under If Statement

In figure 4-15, we see that the pragma is not part of C's grammar; i.e., the object of the statement **if (j == 4)** is the statement on line 78, not the pragma on line 77. Also, since the pragma is not considered part of the grammar, it was not necessary to put braces ({ and }) around lines 77 and 78. It may be wise to insert the braces, anyway, for readability reasons.

One final note about `BBA_IGNORE`. For the 'if' statement that has an 'else' associated with it, a `BBA_IGNORE` in the 'then' part does not ignore the 'else' part. If you want to also ignore the 'else' part, you must put a `BBA_IGNORE` pragma in the 'else' statement, as well. However, for the case of an 'if' statement with no 'else', and when `-DBBA_OPTO=i` is used, a `BBA_IGNORE` in the 'then' part will cause the (inserted) 'else' to also be ignored. In the example in figure 4-15, line 78 is ignored but not line 80. If you want line 80 to be ignored, add a `BBA_IGNORE` pragma after the else statement.

Why Use Ignore Branches?

You might ask, "Why would I want to ignore branches?" Good question! The answer is that there may be conditions that you want to check, but that are impossible (or very difficult) to test. For example, a motor-controller program might be designed to go to a "fail-safe" condition if the motor overheats. You probably would not want to ruin a motor for each test! On the other hand, you don't want to be "penalized" by the test report for doing a complete job of programming, and not being able to test it as completely!

Also, you might want to include code in your product that is only executable in a "maintenance mode", and yet you might not want to test that code.

bbarep And BBA_IGNOREd Branches

In the percent summary (bbarep -S), the number of ignored branches is subtracted from the total number of branches, and the number of ignored branches that were hit is subtracted from the total number of hit branches. However, the number of ignored branches is printed (at the end of the report).

In the function summary (bbarep -s), the number of ignored branches is subtracted from the total number of branches in each function, and the number of ignored branches that were hit in each function is subtracted from the total number of branches hit in each function. Any function that was completely ignored (i.e., there was a BBA_IGNORE pragma prior to the function declaration) will not show up in the listing. Any function which was not completely ignored, but which did have one or more BBA_IGNORE pragmas within it, will have a star next to it in the 'I' column.

When you request a list of unexecuted branches (bbarep -a, -b), no ignored branch will be printed, whether it was hit or not (unless it was also a BBA_ALERT branch, described later in this chapter).

Refer to the discussion of DETAILS OF BBAREP OPTION "-i" in Chapter 6 for more information.

Detailed Explanation Of The BBA_IGNORE_ ALWAYS_ EXECUTED Pragma

The `BBA_IGNORE_ALWAYS_EXECUTED` pragma is used only within while loops. When the `-DBBA_OPTO=w` option is used with `bbacpp`, while loops that are never skipped are reported. If you want to ignore that condition, but not ignore any other branches within the while statement, you must use the `BBA_IGNORE_ALWAYS_EXECUTED` pragma; it ignores ONLY the "while loop never skipped" condition.

```
132     while (ptr != NULL)
133     {
134 #       pragma BBA_IGNORE_ALWAYS_EXECUTED
135         optr = ptr;
136         ptr = ptr->next;
137         if (ptr->string != NULL)
138         {
139             free(ptr->string);
140         }
141         free(optr);
142     }
```

Figure 4-16. BBA_IGNORE_ALWAYS_EXECUTED Example Listing

In figure 4-16, if `ptr` was always not equal to `NULL` the first time the while statement on line 132 was executed, you would normally get the message "while loop never skipped" when you run `bbarep`. With the pragma on line 134 present, that message is suppressed because that condition will be ignored. However, if the "if" statement on line 137 is never `TRUE`, you will still get the message "'then' part of 'if' was never executed".

If the pragma on line 134 was `BBA_IGNORE`, the "while loop never skipped" would still be ignored, but the if statement on line 137 would also be ignored.

Detailed Explanation Of The BBA_ALERT Pragma

The BBA_ALERT pragma causes bbarep to 'flag' you when a branch containing the BBA_ALERT pragma is executed. Specifically, if code which is in the same scope as the BBA_ALERT pragma is executed, bbarep will alert you. Refer to Chapter 7 for details of the bbarep options -s, -aN, -bN, and -l.

The BBA_ALERT pragma can go anywhere the BBA_IGNORE pragma can go. Further, both BBA_ALERT and BBA_IGNORE can be combined in any order on the same #pragma line. See figure 4-17.

```
if (a != OMEGA)
{
#   pragma BBA_ALERT BBA_IGNORE /* should never get here.... */
}
```

Figure 4-17. Combining BBA_ALERT And BBA_IGNORE

Why Use BBA_ALERT

Sometimes when a program is written, conditions are identified as "impossible". However, as specifications change, the case that was "impossible" may become "possible". Or, you may not be in a position to write code to deal with a "possible" condition now, but you want to be reminded to write that code later. A method for dealing with this is to test for the condition, but not have any code for the condition, except whatever is necessary to alert you that the condition has occurred.

Alert branches are a method of notifying you when a branch (which you designate) has been executed.

By using the alert pragma, you do not need to write any code to alert yourself, because the BBA will alert you! In fact, any bbarep report of a function or file that contains executed "alert" branches will contain a notification; the output was designed to immediately catch your attention.

Increasing Push-Back-Line Memory (-DBBA_OPTp <lines> Option)

This option is only necessary when a previous run of bbacpp gives you the following error message:

```
Out of push-back-line memory; use -DBBA_OPTpNNN, where NNN is greater than XXX
```

Where **XXX** is a decimal integer. This message warns you that your source code exceeded an internal limit, and bbacpp could not finish processing your file. The **XXX** will be the current internal limit, and if you specify a number larger than **XXX**, bbacpp will probably succeed. (If it doesn't succeed, it will give you the error message again, and you should try an even larger number).

Reserved Words (Symbols)

Bbacpp generates symbols of the form `_bA_XXXXXXXX`. If your source file has any symbols of that form, the error message below will be printed to stderr:

```
symbol 'xxxxxx' conflicts with bba symbol
```

where **xxxxxx** is the symbol in your source file. In order to avoid these messages, and other problems, do not use bbacpp with any file that gives you this message, or else change the name of the offending symbol to begin with characters other than `'_bA_'`.

Pitfalls With bbacpp And AxLS cc<COMP>

Keep the following considerations in mind when using the BBA with cc<COMP>; where <COMP> represents your compiler number; e.g. 68000, 8086, 68030, etc.

cc<COMP> -h (emulator only)

If you use the **-h** option (to generate HP 64000 format files), and additionally you use **-DBBA_OPTc<constname>** or **-DBBA_OPTd<dataname>**, **bbaunload** may not be able to unload some file data.

If, in addition to the **-h** option, you use the compiler **-d** option, you will not be able to unload the BBA data from the file. This problem is caused by the mapping of compiler segments into HP File Format regions. Only the three HP File regions "prog", "data", and "common" can be used by BBA to store BBA information. If any of the BBA information (**_bA_array** and **_bA_info**) is loaded into the absolute HP File Format region, **bbaunload** will be inoperable. A possible work around is to use **-DBBA_OPTc<constname>** and **-DBBA_OPTd<dataname>** and set both to the name of a segment that will be loaded into the HP File Format data or common regions.

In the situation where you are using the compiler option **-d**, you could define **-DBBA_OPTcdata** and then be able to obtain a BBA dump from this file (provided you do not define other segments).

If you use the Debugger, do not use the **-h** option.

cc<COMP> -s

Do not use the **-s** option for cc<COMP>. The **-s** option causes all symbol information to be stripped. When this happens, the **bbaunload** emulation command or **Unload_BBA** debugger command cannot locate the data arrays.

cc<COMP> -u

Be very careful when using the **-u** option for cc<COMP>. The **-u** option prevents the setting of initialized data to zeros. Getting valid BBA data requires the data arrays to be 0 before the program starts. If you use the **-u** option, you should manually force all of the memory to 0's before loading your program.

The ASM Pragma Of cc<COMP>

The **ASM** pragma of **cc<COMP>** is used to insert assembly language code into your C source file. Branches within the assembly language area will not be recognized by the BBA. The entire block of code between **ASM** and **END_ASM** will be treated as a single C statement.

If the **ASM** pragma is the only statement of an **if** or **else** branch, you must use curly brackets around it. Example:

```
if (a>0)
# pragma ASM
    moveq #1,D3
# pragma END_ASM
```

must be written as

```
if (a>0)
{
# pragma ASM
    moveq #1,D3
# pragma END_ASM
}
```

ccv20 -Q, ccv33 -Q, and cc8086 -Q

If you are using a V20-series, V33-series, or 8086-series microprocessor and you need to use the **-Q** compiler option, you will need to copy the file named **bbacppQ.spec** for your emulator to **bbacpp.spec**. (The normal **bbacpp.spec** file is not compatible with the **-Q** option.)

1. Change to the appropriate directory.
 - a. For 8086-series, use: **cd \$HP/lib/8086**
 - b. For V20-series, use: **cd \$HP64000/lib/v20**
 - c. For V33-series, use: **cd \$HP64000/lib/v33**
2. Save the current version of the **bbacpp.spec** file. You will need it when you make tests and measurements without the **-Q** option.

```
cp bbacpp.spec bbacppNOQ.spec
```

3. Copy the "Q" version of the ".spec" file to **bbacpp**:

```
cp bbacppQ.spec bbacpp.spec
```

As long as you use the **-Q** option, you will need to use the "Q" version of the file **bbacpp.spec**. In your later measurements, if you do not use the **-Q** option, you will need to switch back to use of the file you saved as **bbacppNOQ.spec**.

Note: Only the file named **bbacpp.spec** is actually read during the BBA test process.

Linking Array And Data Sections

In order for the **File→Store→BBA_Data** command to work correctly, the sections that contain the array data and the constant data must have the same "function code". Refer to "WHAT BBA UNLOAD DOES", in Chapter 5 of this manual, for an example linker command file and discussion of how the BBA Unload command reacts to different linking strategies.

Pitfalls With bbacpp And MRI Compilers

Keep the following considerations in mind when using the BBA with MRI compilers:

Not all compiler options are supported. Refer to the appropriate man pages `mcc68kbba(1)`, `mcc86bba(1)`, `mcc960bba(1)`, and `ccc68kbba(1)`.

The HP Branch Validator, when used with the MRI C++ Compiler, will show additional branches that are generated by the translation of C++ to the C language. You can reduce the number of these additional branches by proper selection of options for `BBA_OPTO`.

The HP Branch Validator, when used with the MRI C++ Compiler will not allow you to ignore MACROs by entering the name of the MACRO. You can still ignore the branches generated by a MACRO by individually ignoring each branch.

Notes



Details Of bbaunload or Unload_BBA

Introduction

Chapter 1 briefly explained the purpose of BBA unload. This chapter will explain exactly how to invoke the BBA unload, and it will also describe each of the options available to use with each of the BBA unload commands (**bbaunload** or **Unload_BBA**). Figure 5-1 is provided for quick reference to the emulator **bbaunload** command. Figure 5-2 provides a quick reference to the debugger **Unload_BBA** command. The **File→Store→BBA Data** pulldown menu item can be used in either interface to unload the BBA data.



What BBA Unload Does

The BBA unload command appends all the branch-execution data from the target system RAM or the emulation RAM to a file on the host disc. (Branch execution data is created when files are compiled using the HP Branch Validator preprocessor. Refer to Chapter 4 for more information on how to use **bbacpp**).

The file on the host disc where the branch data is sent can be optionally specified. This is the file that **bbarep** uses to prepare reports on the number of branches hit during execution of the code. Refer to Chapter 6 for details of the **bbarep** command and its options. The name of the dump file defaults to **bbadump.data**, unless otherwise specified.

BBAUNLOAD QUICK REFERENCE

Bbaunload is invoked by using the "bbaunload" softkey when you are in an emulation session. This command unloads the data in the branch-analysis arrays from the emulator memory (either emulation or target memory) to a file named "bbadump.data". Bbaunload always appends the data to the file instead of writing over it. Error messages, if any, are sent to the status line.

Softkey tracking is available to help you enter a correct command because "bbaunload" is executed from within an emulation session.

Usage:

```
bbaunload [ function codes ] load_file [ dump_file ]
```

Examples:

```
bbaunload myprogram
```

```
bbaunload fcode USER_DATA programundertest mydumpfile
```

Figure 5-1. Quick Reference To "bbaunload"

UNLOAD_BBA QUICK REFERENCE

Unload_BBA is invoked via the Memory Unload_BBA command when you are in a debugger/emulation or debugger/simulation session. This command unloads the data in the branch-analysis arrays from the debugger memory to either a file named "bbadump.data", or a file name of your choice. Unload_BBA always appends the data to the file instead of writing over it. Error messages, if any, are sent to the journal window.

Examples:

Memory Unload_BBA All - Unload BBA information from all absolute files loaded, placing data in file bbadump.data.

Memory Unload_BBA Load_File "bbatest" - Unload BBA information from absolute file bbatest.x, placing data in file bbadump.data.

Memory Unload_BBA All to "mydumpfile" - Unload BBA information from all absolute files loaded, placing data in file mydumpfile.

Memory Unload_BBA Load_File "bbatest" to "mydumpfile" - Unload BBA information from absolute file bbatest.x, placing data in file mydumpfile.

Figure 5-2. Quick Reference To Unload_BBA

Read Only If Using M68020, M68030, or M68040

The remaining paragraphs in this chapter apply only if you are using Motorola M68020, M68030, or M68040 emulators because these emulators support separate address spaces selected by using optional function codes. These function codes appear on softkeys when using these emulators.

Link the DATA section for the BBA (the same DATA section used in your source files by default), and the CONST section for the BBA (the same CONST section used in your source files by default) next to each other in your executable file. This ensures that the branch data will unload properly.

Assuming that you used the options **-DBBA_OPTcbbconst** and **-DBBA_OPTdbbadata** when compiling all of the source files, the example linker command file in figure 5-3 will be a good guide for you when you are preparing your own linker command file.

In figure 5-3, **bbadata** and **bbaconst** are together in the address ranging from FFFEA000 up. If we assign that address range to USER_DATA, and invoke **bbaunload** with the function code USER_DATA, the unload operation will proceed correctly. If we move **bbaconst** to address 400 and put it right after const, for example, and the addresses from 400 up are either SUPER_PROG or USER_PROG (it is a program and/or constant section), then the data accessed by **bbaunload** is split into two different function codes and the unload operation will not work properly.

Function codes appear on softkeys when the absolute code has been loaded into a memory configuration that uses the function codes. Depending on the memory configuration selected, there are different modes for function codes:

- They may be enabled (full use of function codes).
- They may be disabled (no use of function codes).
- They may be partially disabled (only PROGRAM/DATA spaces recognized).

If the function codes are disabled (even partially), they will be masked

5-4 Details Of bbaunload In An Emulator

off and ignored during the access of memory by **bbaunload**. When they are enabled, only two choices for function code are valid for the **bbaunload** command: SUPER_DATA and USER_DATA (the branch data always resides in the data area of the program). Function codes are not required when they are disabled in the selected memory configuration. They don't even appear on softkeys. When a function

```
* Example of a linker command file
* In this file, <COMP> = your specific microprocessor number
*                               (e.g. 68020, 68030, etc.).
*                               <PROD> = your specific HP product number
*                               (e.g. 64747, 647480, etc.).
*
CHIP <COMP>
SECT env=$400          * Load address for program/const sections
ORDER env,prog,const,lib,libc,libm

SECT mon =$20000      * Load address for emulation monitor sections
ORDER mon,mondata    * This must be mapped to emulation memory

SECT stack=$7FFF8000 * Load address for stack section

SECT envdata=$FFFEA000 * Load address for data sections
ORDER envdata,data,bbadata,bbaconst,libdata,libcdata,libmdata,heap
*****
* Set register A5 to the beginning address of the data section + 32k
* so that the A5-relative address mode may be used.  If this directive
* is omitted, ?A5 has an undefined value.
*****
INDEX ?A5,data,$8000
LOAD test.o,sub1.o,sub2.o
LOAD $HP64000/env/hp<PROD>/crt1.o
LOAD $HP64000/lib/<COMP>/libc.a
LOAD $HP64000/lib/<COMP>/lib.a
LOAD $HP64000/env/hp<PROD>/monitor.o
LOAD $HP64000/env/hp<PROD>/env.a
END
```

Figure 5-3. Example Of A Linker Command File

code is valid, but not explicitly specified, SUPER_DATA is the default.



Details Of bbarep

Introduction

Chapter 1 briefly explained the purpose of bbarep. This chapter will explain exactly how to invoke bbarep, describe the options available to bbarep, and show you how to interpret the reports produced by bbarep.

BBAREP QUICK REFERENCE

Bbarep is invoked by using the "bbarep" command from the shell. It normally reads data from the "bbadump.data" file (which can be overridden with the "-d" option, see below). All output, except errors, are written to stdout; errors go to stderr.

Usage for C: **bbarep [options] [file | function ...]**

Usage for C++: **bbarep [options] [file | function ...] | c++filt**

Options to bbarep:

- S Print only the percentage of coverage, the number of branches, and the number of ignored branches. This option may be used with style options "-s", "-l", "-b", "-a" and "-D*".
- s Print a summary of coverage information for each function. This is the default. This option may be used with style options "-S" and "-D*".

Figure 6-1. Quick Reference To "bbarep"

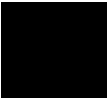
- 
- l** Print the line numbers of unexecuted code for each function.
This option may be used with style options "-S" and "-D*".
 - bN** Print "N" lines prior to the line which generated the branch that was never taken. A value of 0 will print out only the control statement. This option may be used with style options "-S", "-aN" and "-D*".
 - aN** Print "N" lines after the first line that was never executed. A value of 0 will print only the first line. If "-bN" is also specified, the printout will also include all lines from the control statement thru the unexecuted lines. This option may be used with style options "-S", "-bN" and "-D*".
 - D[ftv]** These options are helpful when documenting which files and versions were used in the testing. This option may be used with any of the previous style options.
The documentation options are:
 - f** - list all files used to generate a report, the modification date/time of the files, and which -DBBA_OPTO= options were used when the files were compiled.
 - t** - list total number of compiled files, included files, functions, and branches.
 - v** - report version of "bbacpp" used when files were compiled, and current version of "bbarep".
- The following options change some aspect of the report, but not the style of the report:
- F** Do not print the explanatory footnotes at the end of the report.
 - d<file>** Use <file> as the dumpfile instead of the default bbadump.data.
 - u<usefile>** Only include files or functions whose names are in <usefile>. Note that only one function name or file name should be on each line of <usefile>.

Figure 6-1. Quick Reference To "bbarep" (Cont'd)

- i<ignorefile>** The **<ignorefile>** is a file that contains control statements, file names, and/or function names (one per line). In addition to code which contains the BBA_IGNORE pragma, the BBA will ignore code whose function name, file name, or control statement appears in this file.
- I** Ignore branches associated with include files. This option ignores all branches that are generated through an include file. This is most useful with C++.
- p** Print the column numbers in addition to line numbers of unexecuted code for each function. Note that this is useful only if "-l" was used.
- o** Retain data associated with older sources in the dumpfile. Normally, if the dumpfile contains data from older sources, the old data is ignored (and a warning is printed to stderr). If this is used, however, the BBA will attempt to retain the old data.
- e<tabs>** Expand tabs to **<tabs>** tabstops. The default is one tabstop every 8 spaces. For example, **-e2** will set tabstops every 2 spaces.

Examples:

```
bbarep  
bbarep -l -d old.data -Df  
bbarep -a4 -b4 -e2 proc1 file2.c  
bbarep -S  
bbarep -s -Dtvf
```

Figure 6-1. Quick Reference To "bbarep" (Cont'd)

What bbarep Does

Bbarep reads the dump file that the **File→Store→BBA_Data** command created, and generates any of several reports based on the data. For most reports, the "mapfiles" (which are usually generated when bbacpp runs) are also used, and for the source-reference reports, the source files are also used.

If there are file names or function names on the command line, only those functions listed (or functions within the listed files) will show up in the report. Refer to the discussion of the **-u <usefile>** option, later in this chapter.

The "bbadump.data" files are ASCII files. You may catenate the "bbadump.data" files together if you wish. In fact, the **File→Store→BBA_Data** command normally appends data to data already in the bbadump.data file. Also, you may add lines to the bbadump.data files as long as they do not begin with a colon (:) in the first column. Lines that do not start with a colon are ignored by bbarep.

Normally, results from multiple **File→Store→BBA_Data** commands are ORed together to form a single report. However, if a source file has been modified and compiled between **File→Store→BBA_Data** commands, the data from the newest version is retained and the data from the oldest is skipped. Refer to the discussion of the **-o** option for a way to override this, if desired.

Each of the following sections describes one of the options that bbarep will accept, what it does, and some reasons you might want to use it. Certain of the options (such as **-I** and **-aN**) cannot be used together. If you want reports with multiple styles of output, run bbarep with each style as an option. See the man page to determine legal combinations of command-line flags (or bbarep will warn you if you use an illegal combination). The options **-D[ftv]**, **-S**, **-F**, **-d<file>**, **-u<usefile>**, **-i<ignorefile>**, **-I**, and **-e<tabs>** can be combined with any of the other options.

The BBAPATH Environment Variable

When bbarep needs to use a map file or a source file, it looks for the file in the directory where that file was compiled. If the file is not there, bbarep will generate an error message telling you that it could not find the file. When this happens, the output you wanted may not be available (such as a source reference report).

If the file exists, but it is in a different directory from the one where it was compiled, you can direct bbarep to find it using the BBAPATH environment variable. The format of the BBAPATH variable is the same as the PATH variable; directory names are separated by colons.

For example, if BBAPATH contained

```
"/users/harry/src:/users/harry/src2:/users/harry/src3",
```

bbarep will look for files that it cannot find in their original directories in these three directories: "/users/harry/src", "/users/harry/src2", and "/users/harry/src3". When bbarep cannot find a file, it will search the directories mentioned in the BBAPATH variable in the order they are specified; it will stop when it finds a file with the name it is looking for.

For example, if bbarep needs the file "/users/hairy/mysource/main.M" but that file does not exist, and BBAPATH is set to the example above, bbarep will look for the files:

```
/users/harry/src/main.M  
/users/harry/src2/main.M  
/users/harry/src3/main.M
```

If none of those exist, you will get the error message telling you that "/users/hairy/mysource/main.M" could not be found.

How To Set BBAPATH

In sh(1) or ksh(1), the following commands will set BBAPATH:

```
BBAPATH=<directory>:<directory>:....  
export BBAPATH
```

(The 'export' is necessary to make the BBAPATH variable available to programs executed from the shell.)

Example:

```
BBAPATH=/users/harry/src:/users/harry/src2:/users/harry/src3
```

In csh(1), the command is:

```
setenv BBAPATH <directory>:<directory>:....
```

Getting A Short Summary Report (-S Option)

This option prints only a short summary of the results, including the percentage of coverage, the number of branches, and the number of alert/ignored branches (if the mapping files are available). See figure 6-2.

This shows that of the 33 branches that were supposed to be executed (i.e., 33 branches were not ignored), only 22 were actually executed, for a 66.67% coverage. There were 24 branches that were ignored (i.e., out of 57 branches, 24 were ignored either because their control statement occurred in an ignore file, or because they had #pragma BBA_IGNORE in the source file).

This output format is useful for quickly getting a measurement of how well the complete set of software is being tested.

Note



If a file or function is ignored by a statement in the ignore file, its branches are not included in the summary.

```
22 out of 33 retained branches executed (66.67%)
[ 24 branches were ignored ]
```

Figure 6-2. Example Of bbarep -S Output

Getting The Default Report (-s Option)

This option prints a function-by-function summary of the results, showing the percentage of coverage (and number of branches) for each function. It also includes explanatory footnotes.

This is the default report, which means if no options are given to `bbarep`, it is equivalent to `bbarep -s`. If you want to combine the `-s` option with (for example), `-Dt`, you must include the `-s`.

The `-s` report also includes the information provided by the `-S` option, at the end of the listing.

Figure 6-3 shows four functions (`keyconvert`, `bitpos`, `getkeyvalue`, and `twobits`) in three files (`convert.c`, `getkey.c`, and `multibits.c`).

In `keyconvert`, for example, 8 branches out of a total of 12 retained branches were executed (we know that there were some ignored branches in `keyconvert` because a star (*) appears in the I column next to `keyconvert`'s name). That came to 66.67% coverage for the function 'keyconvert'. If there had been a star (*) in the A column, we would know that at least one `BBA_ALERT` branch in `keyconvert` had also been executed (as there was in `getkeyvalue`).

The footnotes - explaining what the stars meant - can be disabled by using the `-F` option (discussed later in this chapter).

```
_hit_ total_ % IA_ function_ file_
  8 / 12 ( 66.67) * keyconvert convert.c
  6 / 9 ( 66.67) bitpos convert.c
  3 / 5 ( 60.00) *getkeyvalue getkey.c
  5 / 7 ( 71.43) twobits multibits.c
22 out of 33 retained branches executed (66.67%)
[ 24 branches were ignored ]
```

NOTE:

A '*' in the 'I' column means this function had one or more branches that were ignored
A '*' in the 'A' column means this function had a `BBA_ALERT` which was executed

Figure 6-3. Example Of `bbarep -s` Output

6-8 Details Of `bbarep`

This output format is useful when you know you want to enhance your test suite, and want to find out which functions have low test coverage (on the idea that it is better to have all functions tested at N% instead of some functions tested at 100% and some at 0%, even though that might result in the same over-all coverage).

Getting Line Numbers In The Report (-l Option)

This option prints the line numbers and types of branches for each "outer unexecuted" branch. Only the outer one is printed to avoid tons of meaningless output. For example, consider a function with 300 branches in it. If the function is never called, you might get 300 lines of output! By using the **-l** option, you would just get one line, telling you that the function was never called, but you would be notified that there were 300 branches beneath it (see the function `bonzo`, in figure 6-4, below).

The listing in figure 6-4 shows that the function "bonzo" (which is in file "globals.c") was never called, and that there were 300 branches in that function.

The "range of code" (which is 12 through 345 for "bonzo") identifies the line numbers that contain the code that was not executed.

```
range_of_code____#_explanation_____
bonzo      globals.c
  12 -> 345    300 function was never called
keyconvert  convert.c
  37 -> 37     1 'then' part of 'if' was never executed
  44 -> 44     c 1 conditional of 'if' was never FALSE (no 'else' statement)
  49 -> 49     c 1 conditional of 'do while' was never TRUE
  49 -> 52     *** code with BBA_ALERT was executed
bitpos     convert.c
  77 -> 87     5 'case' code was never executed
 194 -> 198    2 'then' part of 'if' was never executed
```

Figure 6-4. Example Of bbarep -l Output

If a "c" is between the line numbers and the number of branches (such as in "keyconvert", line 44), the line numbers refer to the controlling statement, not the lines of unexecuted code. This is necessary, for example, in order to clearly report about an 'if' statement which had no 'else' when the condition of the 'if' was never TRUE.

The three stars (for lines 49 through 52 of function keyconvert) indicate that these lines were a block of code that had a BBA_ALERT pragma, and that code was executed. The stars should catch your attention.

The numbers in the '#' column indicate the number of branches that were never executed. The number in this column can be greater than one when the branch listed has other branches nested within it. For example, consider the code fragment in figure 6-5.

If **a** was always non-zero, you would get a line in the report that looks like:

```
22  -> 30          3 'then' part of 'if' was never executed
```

```
20     if (a == 0)
21     {
22         if (b == 0)
23         {
24             c++;
25         }
26         else
27         {
28             c--;
29         }
30     }
```

Figure 6-5. Code Fragment To Explain The # Column

The '#' column shows the following three branches:

1. One for the **if (a == 0)** statement itself.
2. One for the **if (b == 0)** statement (which could not be executed, because **a** was never **0**).
3. One for the else associated with the **if (b == 0)** statement.

This output is useful when you want to increase test coverage for a large routine. It identifies the branches that "hide" the most branches (so you can concentrate on those branches first). It is also useful if you want to quickly find the executed "BBA_ALERT" branches.

If you have several instrumented branches on a single line, the **-p** option to `bbarep` is useful. Refer to the discussion of DETAILS OF THE **-p** OPTION, later in this chapter for more information.

Note that if a branch is "ignored", it will not show up in this listing unless it was also a "BBA_ALERT" branch and the branch was executed.



Explanation Lines For -l, -bN, And -aN Options

The explanations shown with the **-l** option are usually the same as the explanations shown with the **-aN** and **-bN** options (see below).

Here is a complete list of explanation strings, and their meanings:

function was never called

The function referenced was never called.

conditional of 'if' was never TRUE ' then' part of 'if' was never executed

Both of these tell you that the conditional of an 'if' statement was never evaluated to be TRUE; however, in the second case, the line numbers refer to the unexecuted code, and in the first case, the line numbers refer to the 'if' statement itself. The first form of the message will be used whenever the **-bN** is used without the **-aN** option. The second form will be used for all other reports.

conditional of 'if' was never FALSE ('else' never executed) ' else' part of 'if' was never executed

Both of these tell you that the conditional of an 'if' statement was never evaluated to be FALSE; however, in the second case, the line numbers refer to the unexecuted code, and in the first case, the line numbers refer to the 'if' statement itself.

conditional of 'if' was never FALSE (no 'else' statement)

This tells you that the file was compiled with the -DBBA_OPTO=i option. The lines referenced were an 'if' statement that had no 'else'. Further, the conditional of the 'if' statement was never evaluated to be FALSE (so the 'then' part of the 'if' statement was never unexecuted).

switch never went to 'case'

switch never went to 'default'

switch never went to 'inner case'

switch never went to 'inner default'

'case' code was never executed

'default' code was never executed

All six of these refer to 'case' or 'default' statements inside a 'switch' statement. In the first two, the lines refer to unexecuted code. In the last four, the lines refer to the 'case' or 'default' statement itself. The 'inner case' mentioned in the third message refers to a 'case' statement that is within the scope of another case statement. An 'inner case' example is shown in figure 6-6.

In figure 6-6, **case 1** is a normal case, while **case 2** is an 'inner case', and the **default** is an 'inner default'.

```
switch(a)
{
case 1 : if (i == 4)
        {
            case 2: i++;
              j--;
              default: k++;
        }
}
```

Figure 6-6. 'inner case'/'inner default' Explanation

switch never went to 'case' (no executable statements)
switch never went to 'default' (no executable statements)

In these two messages, the referenced 'case' or 'default' had no executable statements, but the file was compiled with the -DBBA_OPTO=e option so the BBA detected that the 'switch' statement was never executed with a value that caused it to go to the specified 'case' or 'default'.

switch never went to 'default' (code executed by fall-thru)
switch never went to 'case' (code executed by fall-thru)
switch never went to 'inner default' (code executed by fall-thru)
switch never went to 'inner case' (code executed by fall-thru)

These four messages advise you that although the code associated with a 'case' or a 'default' was executed, the switch statement never went to the 'case' or 'default'. This happens when a previous 'case' or 'default' was not followed by a 'break' statement. This is detected when the -DBBA_OPTO=e option is used. Refer to the paragraph associated with figure 6-6 for an explanation of 'inner case' and 'inner default'.

switch went to 'default' (default not defined in code)

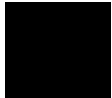
This shows that even though a 'switch' statement did not have a 'default' statement within it, the 'switch' was entered with a value that did not match any of the 'case' statements. This is detectable only when the -DBBA_OPTO=s option is used when compiling the file.

**conditional of 'while' was never TRUE
body of 'while' loop was never executed**

Both of these describe what happens when the conditional of a 'while' statement was never TRUE. The first has line numbers that refer to the 'while' statement itself. The line numbers in the second message refer to the unexecuted code controlled by the 'while' statement.

**conditional of 'while' was never false first time thru
body of 'while' loop was never skipped**

These both refer to cases where the body of a 'while' loop was always executed at least once. For the first message, the lines refer to the 'while' statement itself; for the second message, the lines refer to the unexecuted code controlled by the 'while' statement. This case can only be detected when the file was compiled with the **-DBBA_OPTO=w** option.



**conditional of 'for' was never TRUE
body of 'for' loop was never executed**

Both of these describe what happens when the conditional in a 'for' statement is never TRUE; however, in the second case, the line numbers refer to the unexecuted code; and in the first case, the line numbers refer to the 'for' statement itself.

3rd expression of 'for' was never executed

This means that the third expression in a 'for' command was never reached. This can happen, for example, when a 'return' statement inside a for loop is always executed the first time the 'for' is executed. This can only be detected when the file was compiled with the **-DBBA_OPTO=f** option.

**conditional of 'conditional assignment' was never TRUE
true part of 'conditional assignment' was never executed**

Both of these describe what happens when the conditional in a conditional assignment statement is never TRUE. (The conditional is the part prior to the question mark). In the second case, the line numbers refer to the unexecuted code; and in the first case, the line numbers refer to the conditional statement itself.

**conditional of 'conditional assignment' was never FALSE
false part of 'conditional assignment' was never executed**

Both of these describe what happens when the conditional in a conditional assignment statement is never FALSE; however, in the second case, the line numbers refer to the unexecuted code; and in the first case, the line numbers refer to the conditional statement itself.

**conditional of 'do while' was never TRUE
conditional of 'do while' was never FALSE**

Pretty clear - the line numbers always refer to the line(s) that carry the 'while' statement. It is only possible to get this if the file was compiled with the **-DBBA_OPTO=d** option.

Showing Lines Before The Unexecuted Line (-bN Option)

This option requests N lines to be printed before the line(s) that have the controlling statement. If you use **-b0**, only the line(s) of the controlling statement will be printed.

For example, refer to the code fragment in figure 6-7.

```
15     /* execute if we have not timed out yet */
16     timer_count++;
17     if (timer_count < TIME_OUT)
18     {
19         int i;
20         for (i = 0; i < active_ports; i++)
21             *port[i] = NO_IO;
22     }
```

Figure 6-7. Example Code Fragment

If **timer_count** was never less than **TIME_OUT**, these are the reports that different **-bN** values will give you:

"-b0"

```
(2) conditional of 'if' was never TRUE
17 ->     if (timer_count < TIME_OUT)
```

The "explanation text" precedes the actual source reference. The number in parenthesis indicates the number of branches that were never executed (i.e., they are identical to the numbers in the '#' column for the **-l** report).

The line numbers being printed are to the left; the line referred in the "explanation" text is highlighted with an arrow (pretty useless in the case where only one line is printed, but very useful for the case where multiple lines are printed.)

Any tabs in the source file are expanded as per the **-eN** option, described later in this chapter.

"-b2"

```
(2) conditional of 'if' was never TRUE
15      /* execute if we have not timed out yet */
16      timer_count++;
17 ->    if (timer_count < TIME_OUT)
```

Each time the output changes from one file or function to another, a line is printed showing the function and file before the first condition.

When code containing the "BBA_ALERT" pragma was executed, the output will have stars in the first column. Again, this is done to make it easy to see these cases. For example:

```
*** code with BBA_ALERT was executed
*18      array[i]--;
*19 ->    if (array[0] == 1)
```

If there are two or more branches of the type referenced in the explanation, the branch of interest will be pointed out by carets (^) printed beneath the line:

"-b3"

```
(1) conditional of 'if' was never TRUE
25      /* execute if we have not timed out yet */
27 ->    if (j < k) { j++; } else {if (k > q) q--; else k++; }
           ^^^^^
```

The listing above indicates that **j** was never less than **k**.

The **-bN** output is useful when you want to see the branches that were not executed, but you want to exclude the code associated with these branches

Showing Lines After The Unexecuted Line (-aN Option)

This option requests N lines to be printed after the first line that was not executed. If you use **-a0**, only the first unexecuted line will be printed.

For the following examples, refer to the code fragment shown in figure 6-7. Here are some example listings:

"-a0"

```
(2) 'then' part of 'if' was never executed
20  ->      for (i = 0; i < active_ports; i++)
```

"-a2"

```
(2) 'then' part of 'if' was never executed
20  ->      for (i = 0; i < active_ports; i++)
21           *port[i] = NO_IO;
22           }
```

Again, the "explanation text" precedes the source reference. The number in parenthesis indicates the number of branches that were never executed (i.e., they are identical to the numbers in the '#' column for the **-l** report).

Note that the output may contain lines that were executed (consider the report you would have if you used **-a15**, and an unexecuted branch only controlled three lines).

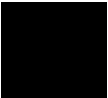
If there are fewer than N lines in the file after the first unexecuted line, **(eof)** will be printed at the end of the listing.

Again, any executed code that had a "BBA_ALERT" pragma will be highlighted by stars:

```
*** code with BBA_ALERT was executed
*21 ->      i++;
*22      #    pragma BBA_ALERT /* Shouldn't get here for version 1.5! */
*23      }
```

Carets are also used to avoid confusion when two or more branches of the same type are on one line:

"-a1"



```
(1) true part of 'conditional assignment' was never executed
20 ->      j = (i < 0) ? ++i : (i < 30) ? i-- : 0;
           ^^^
21      /* what a mess!  check I/O ports to see if
```

The output obtained by "-aN" is useful to see the code that was not executed (and therefore, the program behaviors that were not tested).

Details Of Actions When -bN And -aN Options Are Combined

When you use both the **-bN** and **-aN** options, the listing includes lines before the control statement, after the first unexecuted statement, and some lines in between. The number of lines printed that are "between" the control statement and first executable statement is about the same as the smallest N value used for **-a** or **-b**.

For example:

"-a2 -b2"

```
(2) 'then' part of 'if' was never executed
15      /* execute if we have not timed out yet */
16      timer_count++;
17      if (timer_count < TIME_OUT)
18      {
19          int i;
20  ->      for (i = 0; i < active_ports; i++)
21              *port[i] = NO_IO;
22      }
```

In the above example:

1. Two lines prior to the controlling statement are printed (lines 15 and 16).
2. The controlling statement is printed (line 17).
3. Two lines between the controlling statement and the first executable statement are printed (lines 18 and 19).
4. The first unexecuted statement is printed (line 20).
5. Two lines after the first unexecuted statement are printed (lines 21 and 22).

Note that the arrow points at the line referenced in the explanation. Compare this with the examples discussing the **-bN** option, earlier in this chapter.

This is, perhaps, the most useful output because it shows not only which conditions must occur for a branch to be executed, but also what the unexecuted code would have done.

Showing Character Positions In The Report (-p Option)

This option is only useful when the **-l** option is used. It causes the character positions in addition to the line numbers to be printed to the left of the explanation for each outer unexecuted branch. See figure 6-8.

Figure 6-8 shows that the 'then' code for the first unexecuted branch started in character position 3 on line 15, and ended in character position 2 on line 17. Character positions are defined as the number of characters (not number of columns!) from the start of the line. Each tab counts as 1 character (regardless of how it expands when printed). The first character on a line is character position 1 (not 0).

```
____range_of_code_____#_explanation_____
main  dump1.c
      15/3 -> 17/2      1 'then' part of 'if' was never executed
      24/4 -> 26/3      1 'then' part of 'if' was never executed
```

Figure 6-8. Example Report Using -p Option

Ignoring By Use Of The -i<ignorefile> Option

This option provides another method of ignoring branches (in addition to the `BBA_IGNORE` pragma). It also provides a method of ignoring macros (which can't be ignored with the `BBA_IGNORE` pragma), as well as a convenient method for ignoring all functions in a file.

To use the option, you must have a file (the "ignorefile") which contains lines defining what is to be ignored. There are four types of lines that can be put in an ignore file:

1. File names (ignore all functions in the named file).
2. Function names (ignore all branches in the named function).
3. Macro names (ignore all branches generated by the named macro).
4. Conditions (ignore all branches controlled by the condition).

Note



With C++, the names must be the C++ file names (`.cxx`), the function names must be the C encrypted function names, and the conditions must be the C encrypted conditions from the intermediate file. From a practical point of view, only C++ filenames can easily be entered.

How To Ignore All Functions In A File

A file name is contained on this line. This is usually an "absolute path" name, i.e., a file name beginning with a slash ("/"). However, if the files do not begin with a slash, they are assumed to be paths relative to the directory where the bbarep command is presently running.

Examples:

```
file1.c
/procl/src/omega.c
test_src/gamma.c
```

The first line requests that all functions in the file "<cwd>/file1.c" be ignored; <cwd> is the directory where bbarep is presently running.

The second line requests that all functions in the file "/procl/src/omega.c" be ignored.

The third line requests that all functions in the file "<cwd>/test_src/gamma.c" be ignored; <cwd> is the directory where bbarep is presently running.

How To Ignore A Function

This is done by entry of a line that contains a function name, or contains a "full-path file name: function name". It is exactly equivalent to having a BBA_IGNORE pragma before a function.

Examples:

```
function1
main.c
/users/leslie/project/file1.c: function2
```

The first two lines cause all branches in the functions "function1" and "main" to be ignored. These entries will ignore ALL functions that have the name (a function name can be duplicated if no more than one of the instances of the function name is declared **static**).

The third line causes all branches in the function "function2" in the file "/users/leslie/project/file1.c" to be ignored.

Note



A file name preceding a function must be a full-path file name.

How To Ignore All Branches Generated By A Macro

This is done by entering a macro name on a line. This will only cause those branches that are generated by "outer invocations" of the macro to be ignored. Any macro that is not invoked inside another macro is an "outer invocation".

For example, consider the macro definitions, and macro invocations shown in figure 6-9, and in the following subparagraphs.

```
11 # define MAC1(a)          if (a == 0) j++; else j--;
12 # define MAC2(b,c)      if (b != c) MAC1(c) else MAC1(b)
13 # define MAC3(a)        (a == 0) ? -1 : 1;
14
15 func(j,k,l,m)
16 {
17     MAC1(j);
18     MAC2(k,l);
19     m = MAC3(j);
20     MAC2(MAC3(j), k)
```

Figure 6-9. Macro Definitions

If an "ignorefile" contains the following line:

```
MAC1
```

then the branches generated on line 17 of figure 6-9 will both be ignored. However, although MAC1 is invoked indirectly on lines 18 and 20, those branches will not be ignored.

If an "ignorefile" contains the following line:

```
MAC2
```

then all of the branches generated on lines 18 and 20 will be ignored because MAC2 is the "outer macro".

If an "ignorefile" contains the following line:

```
MAC3
```

then the branches on line 19 will be ignored (assuming you compiled your source files with the **-DBBA_OPTO=A** option!). Although MAC3 is mentioned on line 20, it is not the "outer invocation" so the ignorefile will not ignore it.

You could ignore all branches on lines 17 through 20 by an ignorefile that looks like:

```
MAC1
MAC2
MAC3
```

If you want to ignore MAC1 in only one function, you can enter a line like:

```
../full path name./file1.c: function2: MAC1
```

This is called scoping. A macro can be ignored throughout all of the code by just including the macro name, or it can be ignored on a functional basis by providing the file and function scoping (full-path file: function: MACRO). These are the only two methods of ignoring a macro.



How To Ignore All Branches Controlled By A Specific Statement

This is done by entering a control statement (or part of it) in the ignorefile. In all files and functions, branches that are controlled by that statement will be ignored. For example, if an ignorefile contains

```
if (a == 0)
```

then all if statements that compare **a == 0** will be ignored. Note that this might make the BBA ignore lots of important branches. On the other hand, if you commonly have code that is only executed under circumstances you don't want to test (such as a "maintenance mode"), this can be very useful.

Further, lines in an ignorefile are compared against both the pre-processed and post-processed C source lines (i.e., they are compared both against the text you wrote, and against the results of any macro expansion). Used with care, this might ignore specific branches of macros.

Even if the source line that is the control code spans several lines, the lines must be reduced to one line for each control statement in the ignore file. White space in C language programs (i.e. spaces, tabs, and comments) is ignored when comparing each control statement in the source files with the line in the ignore file.

For example, consider the following code fragment:

```
133  if (beta(8) <
134      omega(90))
135  {
136      printf("oops\n");
137  }
138
139  j = (k == 9) ? /* degenerate case */ 0 : 9;
```

All branches in the above code fragment can be ignored by using the following ignore file:

```
if ( beta(8) < omega(90))
(k==9)?
/* don't really care */
```

Note again that the above statements will ignore all occurrences of these control statements in all of your source files.

By adding the file and function name to these branches, you can confine the ignoring to the specified file and functions. To do this, include the file name, function name, and controlling branch separated by ":"s" all on one line in the ignore file.

Consider the following statements:

```
../full path name../file1.c: function2: if (beta(8) < omega(90))
../full path name../file1.c: function3: (k == 9)?
```

The first line will ignore all branches associated with each occurrence of "if (beta(8) < omega(90))", but only in function "function2" in file "file1.c".

The second line will ignore each occurrence of "(k == 9)?", but only in function "function3", in file "file1.c".

In the following examples, remember that each controlling statement can be scoped to a function by adding file: function: controlling statement.

Note



Scoping will only work on a function level. You must enter both "full-path file:" and "function:" for this scoping to work.

Entries of "file: : controlling statement" and ": function: controlling statement" will not work.

The following examples and explanations will help you to decide exactly what text to insert into an ignore file for each type of branch.

Ignoring An 'if' Statement (And 'else' Statement)

The text to ignore an 'if' statement starts with the 'if' text and stops with the closing parenthesis of the 'if's condition.

Examples:

```
if (j++ <= omicron(890 + falsetto))
if (MAC3(j) == 4)
```

Note



When an 'if' statement appears in an ignorefile, the 'else' statement is also ignored. There is no way to specify ignoring only the 'else' or only the 'then' part of an 'if' statement. However, you can ignore just one of these by using the BBA_IGNORE pragma, discussed in Chapter 4 of this manual.

Ingoring 'case' And 'default' Statements

The text to ignore a 'case' statement starts with the word 'case' and ends with the cases's colon (:).

Examples:

```
case 28:
case HELP_CHARACTER:
case 'Q' :
```


Ignoring An Inserted Default

Defaults are inserted by using the **-DBBA_OPTO=s** option.

The entry required to ignore a default that isn't defined in the code (but was generated when the file was compiled with the **-DBBA_OPTO=s** option) is the name of the switch statement that didn't have the default. It starts with the word 'switch' and ends with the closing parenthesis of the switch's condition.

Examples:

```
switch (input_character)
switch(j)
```

Ignoring A 'while' Statement

The text required in an ignore file to ignore branches generated by a 'while' statement begins with the word 'while' and ends with the closing parenthesis of the 'while's condition.

Examples:

```
while (b != 0)
while(payment > stocks)
```

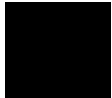
Ignoring 'do-while' Statements

The text required to ignore the branches in a 'do-while' statement is only the condition of the while (no other text). To ignore branches in the following statement:

```
do { j++; } while (j < 5);
```

the ignorefile's line is:

```
j < 5
```



Ignoring A 'conditional assignment' Statement

The text required to ignore a conditional assignment starts with the opening parenthesis of the conditional assignment's condition, and closes with the question-mark (?).

Examples:

```
( j == 4 ) ?  
( k <= j ) ?
```

Note



There is no way in an ignorefile to ignore only the TRUE or FALSE portions of a conditional assignment. If a conditional assignment is specified in an ignorefile, both the TRUE and FALSE branches will be ignored. The BBA_IGNORE pragma, however, can be used to selectively ignore the TRUE or FALSE branch.

Ignoring A 'for' Loop

The text required in an ignorefile to ignore the body of a 'for' loop begins with the word 'for' and ends with the closing parenthesis.

Examples:

```
for ( i = 0; i < sizeof(array); i++)  
for ( /* no init*/; j <= omega; j+= 45)
```

When compiling with the **-DBBA_OPTO=f** option, you may ignore whether or not the third expression of a 'for' statement is executed by inserting the controlling code into the ignorefile. For example, to ignore both the **i++** and **j+=45** shown above, the ignorefile should contain:

```
i < sizeof ( array )  
j <= omega
```

Selecting The Report Content With The -u <usefile> Option

This option specifies a file that contains names of files or functions that are to be included in a report. Only those files or functions named in the <usefile> will be in the report.

The effect is identical to having the files or functions on the command line, but it may be easier to put the list in a <usefile> than to type it on the command line.

The format of the <usefile> is simple. Each line contains the name of a function or file. File names are assumed to be relative to the directory where the bbarep program is running, unless they begin with a slash (/).

For example, if the file "myfiles" contains:

```
gram/lex.c
gram/parse.c
parseerror
```

then the command **bbarep -u myfiles** would report on all functions in the files <cwd>/**gram/lex.c** and <cwd>/**gram/parse.c** (<cwd> is the directory where bbarep is being run), and all functions named **parseerror**.

This is useful when several people are working on a single program, but you only want your report to show the files for which you have responsibility.

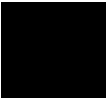
Getting Reports From Other Dump Files (-d <file> Option)

This option tells bbarep to use the file <file> instead of the file "bbadump.data" to obtain the data for the report. Normally, the **bbunload** emulation command or the **Unload_BBA** debugger command writes the coverage data to a file named "bbadump.data". This can be overridden.

Example:

```
bbarep -a5 -b5 -d mydata
```

requests bbarep to generate a source reference listing using data contained in the file **mydata** instead of bbadump.data.



How To Obtain Separate Reports Per Test Suite, And Then Combine Them

You may want to save the coverage from one test set and then run another test-set, and get separate reports on each of them. Perhaps, you might want another report that merges both sets of coverage data). To do this, you might:

1. Remove the old bbadump.data files (to avoid inadvertent merging of coverage information). Use a command such as:

```
rm bbadump.data coverage.1 coverage.2
```
2. Run your first test suite, and use the **File→Store→BBA_Data** command to store the coverage data into a file named coverage.1.
3. Reload your program (to set the coverage data arrays to 0). Then run your second test suite. Use the **File→Store→BBA_Data** command to save the data in a file named coverage.2.

4. Run `bbarep` twice, once for each **File→Store→BBA_Data** command given:

```
bbarep -d coverage.1
```

```
# get report on first test suite's coverage
```

```
bbarep -d coverage.2
```

```
# get report on second test suite's coverage
```

5. To get a merged report (i.e., with the coverage from both test suites ORed together), create a file that is a combination of `coverage.1` and `coverage.2`, using the UNIX `cat` command:

```
cat coverage.1 coverage.2 > coverage
```

and to get the report of the merged data, use the following command:

```
bbarep -d coverage
```

Specifying Spaces In A Tab (-e <tabs> Option)

When the `-aN` and `-bN` options are used, your source files are printed out (in addition to other information). When a tab character is found in your source code, it is expanded the same way that "vi" (and other UNIX utilities) would expand it.

With the `-e <tabs>` option, you can set the number of columns between tabstops (`-eN` is equivalent to the `vi` command `:set tabs=N`).

In order to make the `bbarep` listing have the same indentation as your source file when it is displayed in your editor, you will want to set the `<tabs>` to the same value used by your editor. The default value for `<tabs>` is 8 (i.e., there is a tabstop every 8 characters).

For example, to cause `bbarep` to expand tabs as if there were tabstops every four columns, use the following command:

```
bbarep -a3 -b5 -e4
```

Getting Reports That Include Older Dump Data (-o Option)

This option disables some of the error checking that bbarep does when reading a bbadump.data file prior to generating the reports.

In the bbadump.data file, the data from each file also has the modification date of that file (detected at the time bbacpp was run) associated with it. Normally, bbarep does not combine data when it detects two BBA unload's with different modification dates in the same bbadump.data file. However, the **-o** option tells bbarep to go ahead and combine the data.

If the number of branches has changed (for identical **-DBBA_OPTO=** options), the **-o** option is ineffective. You have made a significant change and the data cannot be reliably merged. In this case, the data associated with the newest file modification time will be retained and you will be notified via a warning message.

This option is useful when you have inadvertently changed the modification dates of the files, but have not changed the branches. For example, if you added a comment, or changed the name of a variable, you might still want to combine old coverage data with new coverage data.

Example of use:

1. File "hello.c" is compiled with the **-b** option. The absolute file is created, run, and a **File→Store→BBA_Data** command is given.
2. File "hello.c" is edited, changing only a comment.
3. File "hello.c" is compiled and linked. The absolute is loaded into the emulator or debugger and run, and a **File→Store→BBA_Data** command is given.

Normally, running bbarep will result in the warning:

```
bbarep: warning: skipping data from older version of file hello.c
date of skipped file is 11/01/87 12:32:48
```

and the coverage data from the first run of "hello.c" is not used in creating the report. However, if you include the **-o** option in your command, the data from both files will be ORed together to form that report, and you'll get the warning:

```
bbarep: warning: using data from multiple versions of file hello.c
```

Suppressing Footnotes In Reports (-F Option)

This option suppresses the footnotes, which are printed after **-s**, **-aN**, **-bN**, and **-I** listings. If you use bbarep often, you may feel that the footnotes are not necessary. For a new user, the footnotes help in understanding the output.

This option is useful to generate listings with a minimum of extraneous output.

Ignoring Branches In Include Files (-I option)

This option ignores all branches that are generated from include files. Each branch is examined to determine if it was generated by a source or include file. Branches associated with include files are ignored. This is the same as choosing **Settings**→**Ignore**→**Include Files...** in the bba SoftBench interface.

This feature is most useful with C++ programs where many branches may come from include files.

Appending Additional Information To Reports (-D[fvt] Option)

This option causes bbarep to append various information after all of the previous options output. This information is useful when you wish to archive information about the test to make it easier to repeat the test, or to define exactly what was tested.

All of the options (**-Df**, **-Dv**, and **-Dt**) may be combined. The following paragraphs describe the output of each option and how to interpret the listings.

Listing Totals (-Dt Option)

This causes a listing of "totals" to be printed to stdout. The following is an example of this list:

```
Number of compiled files: 3
Number of included files: 1
Number of functions: 7
Number of branches (including ignored branches): 56
```

The above list shows the following information:

1. Data for three files were contained in the bbadump.data file from which bbarep generated the report. This implies that these three files are .c files that had been compiled with the **-b** option to **cc<COMP>**, where **<COMP>** = your specific compiler number; e.g., **cc68000**, **cc8086**, **cc68030**, etc.
2. The three files had included (via the **#include <file>** directive) one file. The "Number of included files" is the number of unique files that were included. In this example, either one, two, or three of the compiled files had included the same file.
3. There were a total of seven functions (defined in the three compiled files).
4. There were a total of 56 branches, including any branches that were "ignored".

file	modification date	options	branches	unloads	full path of file
dump1.c	10/14/1987 17:13	a fi	23	7	/hp/dump1.c
Included files:					
dumper.h	10/10/1987 09:58				/hp/dumper.h

Figure 6-10. Report Of bbadump.data Content

**Listing Files
(-Df Option)**

This option gives a listing for each file that was in "bbadump.data". An example output is shown in figure 6-10.

For each .c file, figure 6-10 shows the following:

1. The file's basename (the last column is the file's full path).
2. The modification date of the file (i.e., the last date/time that the file was modified).
3. The **-DBBA_OPTO=** options that were used when the file was compiled.
4. The number of branches (including "ignored" branches) in the file that were instrumented.
5. The number of emulation or debugger BBA unload commands that were given where the file was present.

For each included (.h) file, figure 6-10 shows:

1. The basename of the file (the file's full path is in the last column).
2. The modification date of the file (i.e., the last date/time the file was modified).

Since the "dumper.h" file was included (via the **#include** directive), perhaps in several files, no other data can be shown. The **#unloads** and **-DBBA_OPTO=** options are determined by the file where it was included.

file	modification date	options	branches	unloads	full path of file
dump2.c	10/14/1987 17:13		9	2	/hp/dump2.c
		a	11	3	
		fi	13	2	

Figure 6-11. Multiple Runs With Different -DBBA_OPTO=

In cases where multiple runs were made with different **-DBBA_OPTO=** options, the output may look like figure 6-11.

Figure 6-11 shows that **dump2.c** was compiled three times (once with no **-DBBA_OPTO=** option, instrumenting nine branches, once with **-DBBA_OPTO=a** options, instrumenting 11 branches, and once with **-DBBA_OPTO=fi** options, instrumenting 13 branches).

Further, the version that was compiled with no **-DBBA_OPTO=** options was unloaded twice, the version that was compiled with **-DBBA_OPTO=a** was unloaded three times, and the version that was compiled with **-DBBA_OPTO=fi** was unloaded twice (which implies that the same test suite had NOT been executed with all three versions).

If you used the **-o** option to **bbarep**, and a file had multiple modification dates, the date field would show a star (*), and a line would be added in the report showing the modification dates.

See figure 6-12.

Figure 6-12 shows that the file **dump1.c** was compiled twice with the **-DBBA_OPTO=fi** option, and was edited between compilations. Therefore, there were two modification dates: **11/04/1987** and **10/31/1987**.

```
bbarep: warning: using data from multiple versions of file dump1.c

file      modification date  options  branches  unloads  full path of file
dump1.c   11/04/1987 15:01      a        12        2        /hp/dump1.c
          *              fi        12        3
* = 11/04/1987 15:01, 10/31/1987 15:00
```

Figure 6-12. Report Files With Multiple Modifications

Only the first modification date for an included file is retained.
Therefore, only one date will be displayed beside it.

**Listing Version
Numbers
(-Dv Option)**

By itself, the **-Dv** option simply prints the version of bbarep.

When used with **-Df**, it prints the version number of the bbacpp used to compile the files. Again, this option is designed to make it easier to document the environment in which the tests were executed.



Notes



Details Of bbamerge

Introduction

Chapter 1 briefly explained the purpose of bbamerge. This chapter will explain exactly how to invoke bbamerge, describe the effect of bbamerge on your file system, list the options available to bbamerge, and tell you when and why to use the bbamerge command.

What bbamerge Does

Bbamerge reads the dump file that the BBA unload command generated, and reduces the amount of redundant information present. For example, each BBA unload command will append the name of each C source file (that was compiled with bbacpp) to the dumpfile, along with information associated with that file. The "bbamerge" command will reduce the number of instances of each C source file's name by combining the 'associated information' as much as possible.

If the dump file contains data from different versions of a C source file, only the data associated with the most recent version of the C source file will be retained.

Use the **-o** option to this command if you want to retain data from older versions of C source files.

If you have placed measurement results into separate dump files, you can first cat the files together, and then run bbamerge to compress the results to conserve disk space.

BBAMERGE QUICK REFERENCE

Bbamerge is invoked by executing the "bbamerge" command from the shell. It normally reads data from the "bbadump.data" file (this can be overridden with the "-i" option, see below).

Bbamerge reduces the size of the "bbadump.data" file. The output is normally sent to "bbadump.data", unless you specify a different output file name.

Usage:

```
bbamerge [ options ] [ output-file ]
```

Options to bbamerge:

-o Retain data from older C sources in the dumpfile. Normally, if the dumpfile contains data from older versions of C source files, the old data will be removed when the merged dumpfile is created (and a warning will be printed to stderr). If "**-o**" is used, however, bbamerge will retain the old data in the merged dumpfile.

-i <input-dumpfile> Use the file <input-dumpfile> for input instead of the file named "bbadump.data". The input file and output file may have the same name.

Examples:

```
bbamerge
```

```
bbamerge -i infile.data
```

```
bbamerge -o -i infile.data outfile.data
```

```
bbamerge -o
```

Figure 7-1. Quick Reference To bbamerge

When To Use bbamerge

Use the **bbamerge** command to reduce the size of your "bbadump.data" file. This file gets large after many executions of BBA unload commands. For example, if your "bbadump.data" file is 4 Kbytes after a single use of **File→Store→BBA_Data**, the "bbadump.data" file could be as large as 400 Kbytes after 100 executions of **File→Store→BBA_Data**. The **bbamerge** command can reduce the size of that 400-Kbyte "bbadump.data" file to 6 or 7 Kbytes, depending on how many times you have modified your C source files, etc.

Note that the only way to remove obsolete data from a "bbadump.data" file (when you have not touched your C source files and recompiled them) is to remove the "bbadump.data" file and re-run your tests.



Notes



Tips On More Effective Testing Using BBA

Introduction

This chapter discusses "Makefiles". It shows you how to create them, and how to use them. This chapter also discusses regression testing. Finally, this chapter shows you how to use command files to run Makefiles and perform automatic regression testing.

Makefiles (make)

The most common use of Makefiles is to list all of the files that must be included in an absolute file, and let the computer decide which files are newer than the absolute file, and, therefore, need to be recompiled and linked in order to update the absolute file.

This chapter is not intended to completely explain what **make** is capable of doing or how it works. This chapter is only provided to help you use Makefiles with BBA. See the UNIX manual entry on **make** (in section 1 of the UNIX manual set) for a more complete description of **make**.

The UNIX program called **make** looks for a file of the name "Makefile" or "makefile" in your current directory, and reads it in order to determine what actions to take.

Note



If you use "Makefile" as your file's name, the file will show up first in any directory listing where the rest of the file's names are in lower case. This is useful, because it makes it clear to everybody who looks in the directory that a Makefile has already been written for that directory.

The following paragraphs describe how to create a Makefile, and how to use it to perform tasks.

In the discussion of Makefiles, the following terms will be used: "macros", "targets", "dependencies", and "actions". These terms are defined as follows:

1. Macro - identified by a token beginning in column 1 and followed by an equals sign (=) and value. Macros are referenced by \$(<token>), which replaces the \$(<token>) with its value. Both **COBJ** and **CC** in figure 9-1 are macros.
2. Target - generally anything beginning in column 1, and followed by a colon. In figure 8-1, **program** is a target.
3. Dependency - whatever follows the colon after a target (objects and \$(COBJ)). Dependencies are what the associated target depends on. In figure 8-1, the target called **program** depends on the expansion of \$(COBJ). Also, there may be implied dependencies. For example, **make** understands that any <file>.o depends on some <file>.c.
4. Action - anything that doesn't begin in column 1 (preceded by at least one tab), and follows a target. These are the actions to take when the target is found to be out of date with respect to its dependencies. Actions must be preceded by tabs, NOT SPACES.

```
COBJ= cprog.o csub.o
CC=cc68030 # Replace with your compiler number; e.g., cc68000, cc8086, cc68030, etc.
CFLAGS=
SHELL=/bin/sh

program : $(COBJ)
         $(CC) $(CFLAGS) -o program $(COBJ)
```

Figure 8-1. Example Makefile Without BBA Entries

Makefile Without Branch-Analysis Capability

A simple Makefile is shown in figure 8-1. It will give you an idea of how **make** works. The "Makefile" in figure 8-1 is a Makefile without branch-analysis entries. It simply automates the task of making an up-to-date executable file. It will check to see if any of the object files are older than their corresponding source files, and it will compile new object files for any it finds that have more recent source files. Then it will link all object files together and deliver an up-to-date executable file.

Note that any text following a pound sign (#) is a comment.

To execute this form of makefile and obtain the executable file, all you need to do is type **make** on the command line and press RETURN. See figure 8-1.

Make will go through the following actions:

1. The file shown in figure 8-1 will be read in.
 - a. The macro **COBJ** is defined to be the string **cprog.o csub.o**.
 - b. The macro **CC** is defined to be the string **cc68030** (or your compiler invocation, if different). The macro **CC** is used in **make**'s default rule for creating **.o** files from **.c** files. Refer to letter "a" under number 2, below.
 - c. The macro **CFLAGS** is used in **make**'s default rule for creating **.o** files from **.c** files. Refer to letter "a" under number 2, below.

- d. The macro **SHELL** is defined to be the string `/bin/sh`. The macro **SHELL** is the shell used by **make** when it executes commands.
- e. The target **program** is found. It depends on the files **cprog.o** and **csub.o** (because the **COBJ** macro was expanded). Further, because this is the first target found, this is the "default" target, which is what will be made when no targets are mentioned on **make**'s command line.
- f. The action for the target **program** is read in. It is the string
cc68030 -o program cprog.o csub.o
because the **CC** macro was expanded to become **cc68030**, and the **COBJ** was expanded to the object file names.

2. **Make** seeks to create its target (**program**).

- a. The first dependency of **program** is **cprog.o**. Now **make** tries to figure out if **cprog.o** depends on anything. Although there is no explicit rule (i.e., **cprog.o** is not a target in this makefile), **make** has a built-in rule that says "`<any_file>.o` depends on `<any_file>.c`". Now **make** looks for the file **cprog.c**. When it finds **cprog.c**, it compares the modification dates of **cprog.o** and **cprog.c**. If **cprog.o** is older than **cprog.c**, **make** will execute its default rule for creating a new **.o** file from the **.c** file. The default rule is:

\$(CC) \$(CFLAGS) -c <any_file>.c

Therefore, in this case, the command

cc68030 -c cprog.c

is executed. After that, **cprog.o** is declared to be up-to-date with respect to all of its dependencies (**cprog.c**).

- b. The second dependency of **program** is **csub.o**. The same rules are followed as in step a above. When finished, **csub.o** is declared to be up-to-date with respect to all of its dependencies.
- c. Now the modification date of the file **program** is compared with the modification dates of **cprog.o** and **csub.o**. If **program** is newer, no action is taken. If **program** is older than either (or both) of its dependencies, the defined action is executed (i.e., the following command is executed):
cc68030 -o program cprog.o csub.o
When this command is executed, **make** is finished.



```
COBJ= cprog.o csub.o
CFLAGS= -b -DBBA_OPTO=A
CC=cc68030 # Replace with your compiler number; e.g., cc68000, cc8086, cc68030, etc.
SHELL=/bin/sh

program : $(COBJ)
          $(CC) $(CFLAGS) -o program $(COBJ)
```

Figure 8-2. Example Makefile Configured For BBA

Makefile With Simple Branch-Analysis Capabilities

When you are ready to test your program using the BBA, you can simply modify your Makefile. The appropriate modifications for the Makefile shown in figure 8-1 are illustrated in figure 8-2.

In figure 8-2, **CFLAGS** was modified to add the **-b** option. This causes **cc68030** to use the branch-analysis preprocessor. Also, **-DBBA_OPTO=A** was added. This will be passed to **bbacpp**, requesting that all branches be instrumented.

If you have previously compiled **cprog.o** and **csub.o**, you will need to force their **.o** files to appear to be "out of date" (with respect to their source files) in order to get **make** to recompile the files with the branch analysis information. There are two popular ways to do this:

1. Remove the **.o** files with a command like:

```
rm *.o
```

The command in method 1 actually removes ALL of the **.o** files in your current directory.

2. You can "touch" the **.c** files so they appear to have been edited. The following is a command to touch the files:

```
touch cprog.c csub.c
```

No matter which method you use, when **make** is run, it will execute the following commands:

```
cc68030 -b -DBBA_OPTO=A -c cprog.c
cc68030 -b -DBBA_OPTO=A -c csub.c
cc68030 -b -DBBA_OPTO=A -o program cprog.o csub.o
```

Note that the text created by expanding macro **CFLAGS** is used in each of the invocations of **cc68030**. This is because of the default rule used to convert **.c** files to **.o** files, and because **\$(CFLAGS)** was used on the action line to create **program**.

In this instance, neither **-b** nor **-DBBA_OPTO=A** were needed in the command to link the file. On the other hand, the presence of **-b** and **-DBBA_OPTO=A** didn't hurt; it is considered good practice to use **\$(CFLAGS)** wherever **\$(CC)** is used.

When you have finished doing your testing with branch analysis, you need to remove the branch-analysis structures from your makefile. This returns the file to its former arrangement (figure 8-1).

Makefile With And Without Branch-Analysis Capabilities

Figure 8-3 shows an example makefile that can be used to make an executable file with or without the branch-analysis structures. Figure 8-3 makes the object files with different **CFLAGS**, depending on whether or not you want to make an executable file that has branch analysis, or one that has no branch analysis.

It is almost as simple to write this makefile as it is to write the preceding two makefiles (figures 8-1 and 8-2). You can use it to make an executable file that includes all of the branch-analysis structures, and then use it again to create an executable file that has none of the branch-analysis features. You don't have to make any changes to the content of this makefile to obtain either one of the executable files. All you have to do is change the command you type on the command line when you invoke the makefile, depending on the kind of executable file you want.

If you want an executable file that has no branch-analysis features, type: **make RETURN**. The **make** routine will search for the first target in the makefile (the default), and make the executable file according to that target. The first target is **program**, which produces an executable file with no branch-analysis features.

```

# Target      Action
# program    creates 'program' without branch analysis
# bba       creates 'program' with branch analysis
# object     forces the .o files to be up-to-date with the .c files

COBJ= cprog.o csub.o
CC=cc68030 # Replace with your compiler number; e.g., cc68000, cc8086, cc68030, etc.
CFLAGS=
SHELL=/bin/sh

program : objects
        $(CC) $(CFLAGS) -o program $(COBJ)

objects : $(COBJ)

bba :
        $(MAKE) CFLAGS="$(CFLAGS) -b -DBBA_OPTO=A" objects
        $(CC) $(CFLAGS) -o program $(COBJ)

```

Figure 8-3. Example Makefile That Makes Either File

If you want an executable file that has all of the branch-analysis features, type: "**make bba**" RETURN. The **make** routine will find your makefile and search through it until it finds the **bba** target. Then it will do whatever work is necessary to get the **bba** target up-to-date with respect to its dependencies. The commands under the **bba** target make the executable file with branch-analysis features.

The Makefile in figure 8-3 shows the following new ideas:

1. A target (`bba`) can have no dependencies. In this case, **make** assumes that the target is up-to-date if the file with the name of the target exists (and its actions are not run). If the file does not exist, the actions are executed.
2. It is OK to invoke **make** from within **make**, as long as you avoid infinite recursion. In this case, the re-invocation of **make** occurs when "`bba`" is the target to be made. The macro **MAKE** is (by default) the text "**make**" so when "**make bba**" is executed, the first action taken is:

```
make CFLAGS="-b -DBBA_OPTO=A" objects
```

This causes another **make** to do the actions necessary to get "`objects`" up-to-date with respect to its dependencies. This leads to the next "new idea".

3. A target (`objects`) that is never created is assumed to always be out of date with respect to its dependencies. In this case, trying to get **objects** up to date causes **cprog.c** and **csub.c** to be recompiled (assuming they need to be) - by adding **-b -DBBA_OPTO=A** to **CFLAGS**!

From the above discussion, you know that in order for this makefile to work, you must force the `.o` files to become out of date with respect to their `.c` files (via **touch**, for example) whenever you have been compiling for one target and want to switch to another target. The advantage of this is that if you make changes and want to compile again using the same target, it's easy to do. The disadvantage is that if you forget to **touch** the `.c` source files when you switch to the other target, you may be annoyed with the result.

Another disadvantage of the Makefile in figure 8-3 is that when the **make bba** command is given, the following commands will always be executed:

```
make CFLAGS="-b -DBBA_OPTO=A" objects  
cc68030 -o program cprog.o csub.o
```

even if none of the `.c` files have been modified.

Automatic Makefile With And Without BBA

The Makefile shown in figure 8-4 will 'remember' whether you compiled with branch analysis or without branch analysis the last time **make** was run. This means that you can just type **make bba** if you want the branch analysis version and not have to worry about whether or not to **touch** your source files, etc.

The Makefile shown in figure 8-4 has three new ideas:

1. There is a primitive macro-replacement facility in **make**. In the case of figure 8-4, the macro **CSRC** was defined to be our C source files, and **COBJ** was defined to be the text of **CSRC**, but with the ".c" entries replaced with ".o" entries.
2. If a target is followed by two colons (e.g., **program ::**), then **make** will permit multiple instances of that target; each instance of the target is executed sequentially.
3. The full power of the UNIX shell is available in the actions; in this case, the shell's **if** statement and file-existence test (**[-f <file>]**) is a shell command to return TRUE if **<file>** exists. Refer to the UNIX manual pages on **sh** and **test**, both in Section 1, for more information).

When **make bba** is executed, the following command is executed:

```
if [ ! -f .BBAOBJ ]; then rm *.o .NORMOBJ ; touch .BBAOBJ; fi
```

This command checks for the existence of the file **.BBAOBJ**. If it does not exist, then all **.o** and **.NORMOBJ** files are deleted, and the file **.BBAOBJ** is created. This is done so that the next time **make bba** is executed, the file **.BBAOBJ** will exist and the **.o** files will not be deleted. The files **.BBAOBJ** and **.NORMOBJ** are used to 'remember' which **CFLAGS** were last used during compilation. These two file names begin with a dot (.) so that they don't clutter up a normal listing (but you can force **ls** to show them by using the **-a** flag. Refer to the UNIX manual page on **ls**, in Section 1).

Then the following command line is executed:

```
make CFLAGS="-b -DBBA_OPTO=A" objects
```

This command causes any out-of-date **.o** files to be made.

```

# Target      Action
# program    creates 'program' without branch analysis
# bba       creates 'program' with branch analysis
# object     forces the .o files to be up-to-date with the .c files

CSRC= cprog.c csub.c
COBJ= $(CSRC:.c=.o)
CC=cc68030 # Replace with your compiler number; e.g., cc68000, cc8086, cc68030, etc.
CFLAGS=
SHELL=/bin/sh

program ::
    if [ ! -f .NORMOBJ ]; then rm *.o .BBAOBJ ; touch .NORMOBJ; fi

program :: objects
    $(CC) $(CFLAGS) -o program $(COBJ)

bba ::
    if [ ! -f .BBAOBJ ]; then rm *.o .NORMOBJ ; touch .BBAOBJ; fi
    $(MAKE) CFLAGS="$(CFLAGS) -b -DBBA_OPTO=A" objects

bba :: objects
    $(CC) $(CFLAGS) -o program $(COBJ)
    touch bba

objects : $(COBJ)

```

Figure 8-4. Example Makefile #4

Finally, the modification date of the BBA file is compared with the modification dates of **cprog.o** and **csub.o**. If they are newer, the following commands are executed:

```
cc68030 -o program cprog.o csub.o
touch bba
```

The first line causes **program** to be relinked, and the second causes the file **bba** to be touched so that if another **make bba** command is given (without any source files having been changed) another link is avoided.

The only problem with the makefile in figure 8-4 is that if you frequently go between non-branch-analysis links and branch-analysis links, you will be recompiling ALL of your source files each time you do so.

Automatic/Efficient Makefile With And Without BBA

The Makefile shown in figure 8-5 has several advantages:

1. It will minimize the number of compilations done for each target (non-bba and bba).
2. It will automatically re-compile files that have been changed since the last time they were compiled for the same target (non-bba and bba).

The automatic Makefile has the disadvantage of potentially using a lot of disk space!

The Makefile shown in figure 8-5 uses two subdirectories: **.bbaobj**, and **.normobj**. When you go from using branch analysis to not using it, for example, the current **.o** files (which were created with branch analysis) are copied to the directory **.bbaobj**, then removed from the current directory. The file **.BBAOBJ** is removed, preventing any further copying of files down to **.bbaobj**.

The **.o** files in **.normobj** are copied to the working directory, and the **.NORMOBJ** file is created.

If the next **make** command creates **program**, the file **.NORMOBJ** exists, so the only actions are to update the **.o** files and perhaps to re-link.

```

# Automatic Makefile Example.
# Target      Action
# program     compiles & links without bba
# bba         compiles & links with bba

CSRC=cprog.c csub.c      # list of source files
COBJ=$(CSRC:.c=.o)       # list of object files we want (derived from source list)
CC=cc68030               # Replace with your compiler number; e.g., cc68000, cc8086, cc68030, etc.
CFLAGS=
SHELL=/bin/sh

program ::
    if [ ! -f .NORMOBJ ]; then \
        if [ -f .BBAOBJ ]; then \
            rm .bbaobj/*.o ; ln *.o .bbaobj; rm .BBAOBJ; \
        fi; rm *.o ; ln .normobj/*.o .; touch .NORMOBJ; \
    fi; exit 0

program :: objects
    $(CC) $(CFLAGS) -o program $(COBJ)

bba    ::
    if [ ! -f .BBAOBJ ]; then \
        if [ -f .NORMOBJ ]; then \
            rm .normobj/*.o ; ln *.o .normobj; rm .NORMOBJ; \
        fi; rm *.o ; ln .bbaobj/*.o .; touch .BBAOBJ; \
    fi; exit 0

bba ::
    $(MAKE) CFLAGS="$(CFLAGS) -b -DBBA_OPTO=A" objects
    $(CC) $(CFLAGS) -o program $(COBJ)

objects : $(COBJ)

```

Figure 8-5. Fully Automatic Makefile

Automatic Regression Testing

The problem of verifying that software is correct grows geometrically with the size and complexity of the program under development. You can increase the coverage of your tests by doing module-based testing. This form of testing can be more precise, and cover more possibilities. The traditional method of testing code is to wait until the code is complete and then test it as a whole. This makes it difficult, if not impossible, to measure the performance of sections of the code that are deeply embedded within the program. For example, if a module of code is intended to convert the values of a Gray-code counter to a linear count, and then send the result to an LED display, you can only test its performance by putting different values into the Gray-code hardware and then observing the display. This method is time consuming and prone to error.

The following paragraphs will show you a way to:

1. Isolate such modules (or functions) for testing.
2. Automate the testing process itself.

Test automation is easy when you use the programmability of UNIX and the controlling power inherent in an emulator or debugger. In fact, you can create "regression tests" that start when nobody is around and store the results for reporting later. This means changes in one section of code that cause unexpected behavior in another section of code can be detected and fixed early in the project.

The basis of this test method is to write the tests at the same time you write the code module. In this way, the tests tend to be more complete because you are most familiar with the code module when you are writing it. This is called "white-box testing". Also, writing tests for a single module at the same time as writing the module itself is relatively painless when compared with the task of writing all of the tests at the end of the project.

The tests are written in the form of UNIX shell scripts combined with HP64000-UX or debugger command files.

The emulator or debugger command files do the following tasks:

1. Load the program (or perhaps a portion of the program) in an emulator.
2. Set up the initial test data.
3. Run the program from start to finish.
4. Write the resulting data out to a file.
5. Repeat the preceding test with new data. This repetition continues until the module under test has operated on a sufficient variety of data to give a high degree of confidence in its performance. (i.e., bbarep shows a sufficiently high coverage percentage).

The UNIX shell script executes command files, and compares the outputs generated with 'known good' output.

Alternatively, a Makefile can be written to execute the command files and compare the outputs.

When combined with the UNIX "crontab" command, the UNIX shell scripts can be automatically run on a regular schedule, and the results of the tests can be "mailed" to you (or to a group of people) via UNIX "mail".



Detailed Example Of Automatic Regression Testing

Suppose you are writing a function to convert a 64-bit integer into an ascii decimal string. Call the function "i64toa". Your function is normally invoked after an A/D has sampled a line, then added an offset factor, and then multiplied the result by a scale factor. To test this function, you decide to send it a series of selected values, and check that the output is correct for each value. You choose inputs that will inspect the boundary conditions (or, perhaps, you start with some input data and let bbarep guide you in creating more complete data).

If you were testing without an emulator or debugger, you would have to figure out what voltage to send to the A/D, and determine the appropriate scale factors to use in order to test this function. By having an emulator or debugger in your test equipment, you can simply set the necessary values in the data areas read by i64toa, and let the module run.

For an emulator, figure 8-7 shows an example command file that could be used to make the example test with data in figure 8-6.

input, hex	output, ASCII
0000000000000000	0
0000000000000001	1
0000000000007FFF	32767
0000000000008000	32768
0000000000008FFF	36863
000000000000FFFF	65535
0000000000010000	65536
0000000000010000	1048576
.	.
.	.
.	.
1000000000000000	1152921504606846976
7FFFFFFFFFFFFFFF	9223372036854775087
8000000000000000	9223372036854775088
FFFFFFFFFFFFFFF	18446744073709551615

Figure 8-6. Input/Output Selected For Example Test


```

# FILE: i64toa.cmd
#
load memory bigprog          # load in the big program
# input value is 0000000000000000H
modify memory i64toa_msword to 0
modify memory i64toa_mshword to 0
modify memory i64toa_mslword to 0
modify memory i64toa_lsword to 0
run from i64toa until i64toa end
wait measurement_complete
copy memory i64toa_ascii thru i64toa_ascii + 20 absolute bytes
                           to i64toa.lst.2

# input value is 0000000000000001H
modify memory i64toa_msword to 0
modify memory i64toa_mshword to 0
modify memory i64toa_mslword to 0
modify memory i64toa_lsword to 1
run from i64toa until i64toa end
wait measurement_complete
copy memory i64toa_ascii thru i64toa_ascii + 20 absolute bytes
                           to i64toa.lst.2

... and so on.

```

Figure 8-7. Command File To Run Emulator Example Test

Scripts to Run Emulator Command Files

Figure 8-8 shows an example UNIX shell script that runs the example test.

Note that the example shell script in figure 8-8 uses the same format, no matter which modules are being tested. To create new shell scripts when you need them, you can probably copy your most recent shell script and change the names to identify the new command files. After you have finished testing a module, you can include its test in the master-testing file (`test_master`) which is run automatically each day. Figure 8-9 shows an example of a master-test shell script.

```

# FILE: i64toa
#
#     test the i64toa program

rm -f i60toa.lst.2          # remove old attempt

# Start the emulator, telling it to execute commands from
# the i64toa command file:
emul700 -c i64toa.cmd m68000

# Compare the 'known good' output with what we just got:
# Put the comparison results in a temporary file so we can
# print them without having to run diff again ....

diff i64toa.lst i64toa.lst.2 > .tmpdiff$$ 2>&1

# If the diff program found a difference, print an error message;
# else report that the program completed successfully
if [ $? != 0 ]
then
    echo "ERROR: i64toa.cmd: the 64-bit to ASCII conversion function failed!"
    cat .tmpdiff$$
else
    echo "OK: i64toa.cmd: the 64-bit to ASCII conversion function passed!"
fi
rm -f .tmpdiff$$

```

Figure 8-8. Shell Script To Run Example Command File

```

# FILE: test_master
#
#     This file invokes all regression tests.
#     It is run from cron each night and the results
#     are mailed to me each morning

i64offset          # test offset function
i64scale           # test the scaling system
i64toa             # test conversion to ASCII

```

Figure 8-9. Master-Test Shell Script Example

8-18 Effective Testing Using BBA

Makefile To Run Emulator Command Files

You can also use a Makefile to run the tests and compare output. This method tends to hold the tests together better (i.e. centralize and standardize the tests, because everybody knows how to run **make**).

Figure 8-10 shows a Makefile that is analogous to both the **test_master** and **i64toa** files used in this example (figures 8-7 and 8-8).

Note that when you add new tests, all you have to do is copy the actions of **i64toa** and change the test **i64toa** to the name of the new test. This is very easy to do in most editors!

```
# Testing Makefile
# target      action
# all         execute all tests
# i64toa      test the function i64toa()

EMUL=m68000      # emulator's name
DIFF=diff        # text comparator program
SHELL=/bin/sh

all : i64toa i64offset i64scale

i64toa :
  rm -f i64toa.lst.2          # remove old attempt
  emul700 -c i64toa.cmd $(EMUL) # execute command file
  if $(DIFF) i64toa.lst i64toa.lst.2 > .tmpdiff; then \
    echo "i64toa.cmd: the 64-bit to ASCII conversion function passed!"; \ else
    echo "i64toa.cmd: the 64-bit to ASCII conversion function failed!";
    cat .tmpdiff; fi
```

Figure 8-10. Makefile To Run Emulator Command Files

Advantages Of Automatic Regression Testing

Performing automatic regression testing during the software-development phase of a project offers several advantages. These are listed below:

1. Tests are written as code is developed, increasing test coverage.
2. The risk of propagating undetected defects is decreased.
3. The effort of designing tests is spread out over the entire project instead of piling up at the end.
4. Unexpected side effects of code changes are detected quickly.
5. The task of developing a quality-assurance test routine for the entire program is greatly simplified.

Details Of The HP Branch Validator (BBA) In The HP SoftBench Interface

Introduction

This chapter discusses all of the detailed information you will need to operate the HP Branch Validator (BBA) within the HP SoftBench User Interface. The HP Branch Validator (BBA) SoftBench User Interface will operate (with some limitations) even if you have not purchased HP SoftBench.

Customizing The HP Branch Validator

Perform the following steps to ensure proper operation of the HP Branch Validator in the HP SoftBench Interface.

1. For each user of BBA, copy the BBA configuration file (**\$HP64000/bba/config/bbarc**) to the users \$HOME directory and name it **.bbarc**. Use a command like:

```
cp $HP64000/bba/config/bbarc /users/clarkkent/.bbarc
```

The configuration file can be modified to customize the BBA interface for the needs of each user. In particular, if you are not using HP SoftBench Broadcast Message Server, set **BBA_USE_SOFTBENCH=False** in each user's **.bbarc**. In addition, if you are using MRI C++, you must set **BBA_C_PLUS_PLUS=True**.

If you are using HP SoftBench Broadcast Message Server, each user who has a **.softinit** file should add the BBA tool to their SoftBench initialization table. To add the BBA tool, copy the BBA line from the **softinit** file (**/usr/softbench/config/softinit**) to the users **.softinit** file.

The line will look like:

```
BBA TOOL DIR * %Local% bba -host %Host% -dir  
%Directory% -toolmgr
```

2. You may need to modify the command that enables printing to reflect your system's printing mechanism. Printing is controlled by the script file **\$HP64000/bba/config/cmd_print**.

When a print command is issued, this script file receives the name of the file to be printed. The script file then executes its default "**lp \$1**". If this print command needs to be changed for use in your system, such as "**lp -dlaser \$1**", you will need to edit this file.

3. Finally, you may want to copy the BBA Xdefaults to a users **\$HOME/.Xdefaults** file. Copying the BBA Xdefaults file to a users file is only necessary if you want to edit the file to change the default keyboard accelerators for BBA. The default Xdefaults are located in the file **\$HP64000/lib/X11/app-defaults/BBA**.

Details About The .bbarc File

The **.bbarc** file customizes the BBA interface to a users needs. This file consists of about 23 options which are read when the HP Branch Validator (BBA) SoftBench User Interface is started. If you change one of these options while the HP Branch Validator (BBA) SoftBench User Interface is running, you will have to click on Read Configuration File in the Settings pull down menu before the option you just changed will take effect.

Options are included in the **.bbarc** file to:

1. Define the general operation of BBA.
2. Define default parameters needed by BBA.
3. Define the actions of various BBA Pull Down Menu items.

Options Controlling General BBA Operation

The first three options in the .bbarc file determine the general operation of BBA. These are BBA_USE_SOFTBENCH, BBA_C_PLUS_PLUS, and BBA_PRINTER_COMMAND.

BBA_USE_SOFTBENCH

This option specifies whether or not you are using the HP SoftBench Broadcast Message Server.

When BBA_USE_SOFTBENCH=True, Edit, Build, and Help requests are directed to the SoftBench Broadcast Message Server, and ultimately to the tools servicing these requests.

When BBA_USE_SOFTBENCH=False, the BBA interface does not use the SoftBench Broadcast Message Server. Instead, the BBA interface executes edit and build commands by using the Edit and Build options discussed in the paragraph titled "Options If Not Using HP SoftBench Broadcast Message Server", later in this chapter. The help feature is disabled when BBA_USE_SOFTBENCH=False.

Default: BBA_USE_SOFTBENCH=True

If you are not using HP SoftBench Broadcast Message Server, set this option False:

BBA_USE_SOFTBENCH=False.

BBA_C_PLUS_PLUS

This option specifies whether or not you are using C++ files. This option should be set to true if you are examining C++ files, or combinations of C and C++ files. For faster performance, this option should be set to False when examining only C files.

Default: BBA_C_PLUS_PLUS=False

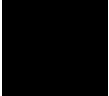
Note that setting BBA_C_PLUS_PLUS=True will invoke a c++filt(1) routine to demangle C++ function names. Make sure your PATH shell variable can find c++filt before setting this to True.

BBA_PRINTER_COMMAND

This is the command that directs printing of BBA reports.

Modify either the **\$HP64000/bba/config/cmd_print** file, or this command directly to specify how to print BBA ascii reports. The first parameter passed is the name of the file containing the report to print.

Default: BBA_PRINTER_COMMAND=
\$HP64000/bba/config/cmd_print



Special BBA Options And Their Default Values

The following options specify default values for various BBA parameters.

BBA_IGNORE_FILE

Name of the file containing the list of ignored branches, files, and functions.

Default: \$HOME/BBA_IGNORE_FILE

BBA_PRAGMA_FILE

Name of the file containing a list of pragmas to be inserted into the users source file.

Default: \$HOME/BBA_PRAGMA_FILE

BBA_DUMPDATA_FILE

Name of the file pointing to the BBA dump file.

Default: bbadump.data

BBA_SOURCE_TAB_WIDTH

Expand tabs in the source listings to the specified number of spaces. Enter a value between 1 and 8.

Default: 8

BBA_RETAIN_OLD_DATA

Retain data associated with older sources in the dumpfile. Normally, if the dumpfile contains data from older sources, the old data is ignored. If this is set True, the BBA will attempt to retain the older data.

Default: False

BBA_IGN_COMMENT_TYPE

Defines the type of comment to be added with each branch, file, or function that is ignored. "Unique Comments" query you each time for an appropriate comment. "Universal Comment" is appended after each ignore. "No Comment" does not append any comment with ignores.

Default: No Comment



BBA_IGNORE_COMMENT

Defines the default comment string to be included with each branch, file, or function that is ignored when you have selected "Unique Comments" or "Universal Comment" in BBA_IGN_COMMENT_TYPE.

Default: # Enter Comment

BBA_PRAG_COMMENT_TYPE

Defines the type of comment to be added with each pragma in the source file. Pragma comments are added to the line just before the line where the pragma will be added. "Unique Comments" query you each time for an appropriate comment. "Universal Comment" is included with each pragma. "No Comment" does not include any comment with your pragma.

Default: No Comment

BBA_PRAGMA_COMMENT

Defines the default comment string to be included with each pragma when you have selected "Unique Comments" or "Universal Comment" in BBA_PRAG_COMMENT_TYPE.

Default: /* Enter Comment */

BBA_PRAGMA_TYPE

Defines the type of pragma to be added to your source file. The pragmas that can be added include: BBA_IGNORE, BBA_ALERT, both BBA_IGNORE and BBA_ALERT together, and BBA_IGNORE_ALWAYS_EXECUTED. Enter one of the following five values as your pragma to add to the source file:

Ignore
Alert
Ignore Always Executed
Ignore and Alert
Prompt for Pragma

The "Prompt for Pragma" selection will prompt you for one of the four pragma types when you select "Add Pragma" in the source window.

Default: Ignore

BBA Options Used By The Actions Pull Down Menu

The following options are executed when you select items in the Actions pull down menu bar:

BBA_MAKEFILE_NAME

Name of makefile if you select "BBA Build". This option is not used if BBA_USE_SOFTBENCH=False.

Default: BBA_MAKEFILE_NAME=Makefile

BBA_MAKEFILE_OPTIONS

List of options to be used with Makefile when "BBA Build" is selected. This option is not used if BBA_USE_SOFTBENCH=False.

Default: BBA_MAKEFILE_OPTIONS=all.

BBA_UNLOAD_COMMAND

Command that executes an unload operation if you select "BBA Unload". Typically, this command is used to execute a script to unload an emulator or simulator. Refer to Chapter 5 for details of the BBA unload command.

BBA_MERGE_COMMAND

Command that controls merge operations if you select "BBA Merge". Typically, this option will execute "bbamerge". Refer to Chapter 7 for details of the bbamerge command.

BBA_RUN_TEST_COMMAND

Command that executes a test if you select "Run Test". This may be a script to run an emulator or simulator. Note that the currently selected file, function, and line number will be passed to this option if they are selected.

Note



Do NOT add a change directory command directly to the following option strings: BBA_RUN_TEST_COMMAND, BBA_STOP_TEST_COMMAND, BBA_MOD_TEST_COMMAND, and BBA_ADD_PRAG_COMMAND.

You should enter only a simple command or a script command. (You can have a change directory command in the script command, if desired. Do NOT do this:

```
"BBA_RUN_TEST_COMMAND=cd /users/clarkkent;  
my_run_command")
```

BBA_STOP_TEST_COMMAND

Command to halt a running test if you select "Stop Test". This may be a script to stop an emulator or simulator. Note that the currently selected file, function, and line number will be passed to this option if they are selected.

BBA_MOD_TEST_COMMAND

Command that opens a test file for modification if you select "Modify Test". This may be used to modify a specified test. Note that the currently selected file, function, and line number will be passed to this command if they are selected.

BBA_ADD_PRAG_COMMAND

Command that writes all present pragmas into the designated source files if you select "Add Pragmas to Source Files". This command is passed three parameters:

1. The first parameter is the complete list of source files to be updated with new pragmas.
2. The second parameter is the name of the file that lists all of the pragmas to be added.
3. The third parameter is the file that returns (to BBA) the names of the files that have had pragmas successfully added.

Normally, you will not want to modify BBA_ADD_PRAG_COMMAND unless you want pragmas to be inserted into your source files in a different manner.



Options Used When Not Using HP SoftBench Broadcast Message Server

The following options are only available if you are not using the HP SoftBench Broadcast Message Server (BBA_USE_SOFTBENCH=False).

BBA_BUILD_COMMAND

Command that performs a build of your executable file if you select "BBA Build". This option should include the complete command to build your BBA executable. You may want to make this a script that executes the desired make commands. Note that options BBA_MAKEFILE_NAME and BBA_MAKEFILE_OPTIONS are not used here (they are only available if you are using the HP SoftBench Broadcast Message Server).

Note



Do NOT add a change directory command directly to the BBA_BUILD_COMMAND or BBA_EDIT_FILE_COMMAND option strings. You should enter only a simple command or a script command.
(You can have a change directory command in the script command, if desired. Do NOT do this:
"BBA_BUILD_COMMAND=cd /users/clarkkent; make bbatest")

BBA_EDIT_FILE_COMMAND

Command that starts an edit session of a file if you select an Edit pull down menu item. Parameters passed to this command are a file name and a line_number. By default, this command invokes the vi editor. You can modify the script in `$HP64000/bba/config/cmd_edit` to specify a different editor if you do not want to use the vi editor.

Default: `set -m;$HP64000/bba/config/cmd_edit`

The "set -m;" in front of this command prevents BBA from exiting until the BBA_EDIT_FILE_COMMAND command completes. This avoids killing your edit session when you quit BBA. Note that BBA

will not leave the screen until the edit completes.
(You can iconify BBA to remove it, if desired.)

Before Starting BBA

BBA can be run with or without using the HP SoftBench Broadcast Message Server. Before starting BBA, you need to decide whether or not you will use the SoftBench Broadcast Message Server. To make this decision, consider that the SoftBench Broadcast Message Server provides all the capabilities under the Edit pull down menu, the "BBA Build" function in the Actions pull down menu, and all of the Help menu items. In certain CPU systems that have limited memory available, using the HP SoftBench Broadcast Message Server may reduce the performance of your system. HP recommends at least 12 Mbytes of RAM when using HP SoftBench.

If you want to use the SoftBench Broadcast Message Server, set `BBA_USE_SOFTBENCH=True`. If you do not want to use the SoftBench Broadcast Message Server, set `BBA_USE_SOFTBENCH=False`. Even with a "False" setting, you can still use the Edit and Build pull down menu items if you follow instructions in the paragraph titled "Options Used When Not Using HP SoftBench Broadcast Message Server", above.

The `BBA_USE_SOFTBENCH` option is located in your BBA configuration file, which must be named `.bbarc` in your `$HOME` directory.



Starting BBA And Using The SoftBench Broadcast Message Server

When the SoftBench Broadcast Message Server is running, BBA can be invoked in one of two ways: (1) it may be invoked from the command line, just like any other UNIX command, and (2) it may be invoked by using the SoftBench Tool Manager.

Invoking BBA From The Command Line

On the command line, enter **bba** and press return. The SoftBench Broadcast Message Server will connect with BBA. The message "Starting the Basis Branch Analyzer Interface..." will appear on your display, and BBA will start in a few moments. You may see the following message:

```
encaprun: cannot find a message server
  Make sure that there is a message server running and
  that either $DISPLAY or $MSERVE were set correctly,
  both when SoftBench was started, and also in the current environment.
```

Appearance of the above message indicates that the SoftBench Broadcast Message Server is NOT running and it needs to be started. BBA will still work, with the exception that Build, Edit, and Help commands will be inoperable. To get the SoftBench Broadcast Message Server running, exit BBA, start the SoftBench Broadcast Message Server by executing "softbench", and then restart BBA. Refer to your SoftBench manuals titled, "Installing HP SoftBench" or "Exploring HP SoftBench: A Beginner's Guide."

Starting From The Tool Manager

Move the cursor into the Tool pull down and click on Start.

A new window will open, showing a list of the tools available in the SoftBench Tool Manager. You may have to scroll down this window to find BBA. Click on BBA in this list. This selects BBA as the tool to be started. To start the BBA tool, press the Start button (by moving the cursor into the "Start" button and clicking the command select mouse button). You can also start BBA by placing the cursor on BBA in the list of tools and double-clicking the command select mouse button.

Starting BBA, But Not Using The SoftBench Broadcast Message Server

You can use BBA and not use the SoftBench Broadcast Message Server if you wish. This will give you a little better performance on CPU systems that have less than the recommended amount of RAM (12 Mbytes for HP SoftBench installations) because you will not be using as many SoftBench Tools.

First, verify that you have set option `BBA_USE_SOFTBENCH=False` in your configuration file. Then start BBA from the command line by entering `bba` and pressing return. The message "Starting the Basis Branch Analyzer Interface..." will appear on your display and BBA will start in a few moments. In addition, you will probably see this message:

```
encaprun: cannot find a message server
Make sure that there is a message server running and
that either $DISPLAY or $MSERVE were set correctly,
both when SoftBench was started, and also in the current environment.
```

This message is simply a warning message and can be disregarded. When properly set up, this mode of operation will still allow you to use all of the BBA commands in the pull down menus, with the exception of the Help command. The Help command uses resources within the SoftBench Broadcast Message Server.

Note



If you have HP SoftBench installed, you must add `"/usr/softbench/bin"` to your `PATH` variable even if you are not using the SoftBench Broadcast Message Server.

Using The Four Test Report Displays

There are four test reports that can be displayed in the bottom half of the main Branch Validator window. These are labeled: Summary, Histogram, Results Only, and File History. If your Xdefaults file is properly set up, you can cycle through these test reports by typing <Shift><Alt>S, <Shift><Alt>H, <Shift><Alt>R, and <Shift><Alt>F, respectively.

Note



On HP-UX systems, <Alt> is the <Extend char> key. On Sun SPARCstations, <Alt> is the diamond-shaped key to the left of the space bar. Refer to Chapter 3 for more details on pull down menu accelerators and mnemonics.

The following paragraphs assume you have a bbadump.data file to examine. If this is not the case, you may want to set the context to **\$HP64000/demo/bba/demo2** and use the bbadump.data file supplied for the "Getting Started" demonstration. Do this by choosing **File→Set Context...** and then entering the proper context in the new window that pops up.



The Summary Test Report

The Summary test report (**File→Show Summary** or <Shift><Alt>S) presents a BBA summary that shows file-by-file and function-by-function branch-coverage results. Information in the summary test report includes: percent of test coverage, and number of retained (not ignored) branches, both total and hit. In addition, if an asterisk "*" appears in the "I" column in the report, some branches in the associated file or function were ignored. An "*" in the A column of the report indicates that a branch that contained a BBA_ALERT pragma was executed. At the end of the summary test report is a cumulative total of the branch coverage results.

The "Totals" line reflects the totals of all of the listed functions. It may not be an accurate representation for the total of the file. In particular, functions that are ignored will not be counted in the totals. Instead, the branches in these ignored functions will be counted in the summary listing at the bottom of the display (XX branches were ignored).

Use the vertical scroll bar to scroll through the list of files and functions to examine branch-coverage test results.

You can select a file or function in the report by moving the cursor into the desired line and clicking the *command select* mouse button. Use the Next and Previous keys to select the next or previous file or function in the report. If you select a file, Next and Previous will select the next and previous files. If you select a function, Next and Previous will select the next and previous functions (even between different files).

The Histogram Test Report

The histogram test report (**File**→**Show Histogram** or <Shift><Alt>H) presents a BBA histogram that shows function-by-function branch-coverage results.

Information displayed includes: percent of coverage, a histogram display of the coverage percent, the function name, and the file name.

Use the vertical scroll bar to scroll through the list of functions to examine branch-coverage test results. To select a function in the display, move the cursor to the desired line and click the *command select* mouse button. Use the Next and Previous keys to select the next or previous function in the display.

The Results Only Test Report

The results only test report (**File**→**Show Results Only** or <Shift><Alt>R) presents a BBA Results Only listing. This listing is a cumulative total of the branch coverage test results, along with the total number of files, functions, and branches instrumented for BBA.



The File History Test Report

The file history test report (**File**→**Show File History** or <Shift><Alt>F) presents the following information about the files that make up the executable that was subjected to the BBA tests:

1. The files that were tested.
2. The dates when the files were compiled.
3. The BBA options with which the files were compiled.
4. The number of branches in each file.

5. The number of times the data from each file was unloaded.
6. The complete path of each file.

This display is identical to "bbarep -Dft". Refer to Chapter 6 for a more complete description about this display.

Use the vertical scroll bar to scroll through the list of files. Use the horizontal scroll bar to view the complete path of each file.

The BBAPATH Environment Variable

The BBA Interface will search for map and source files in the directories where they were originally compiled, and it will report an error if it cannot find a file. When you have moved map and/or source files out of the directories where they were originally compiled, you can use the BBAPATH Environment Variable to define the new paths to the map and source files. To set up this variable, refer to the paragraph titled "BBAPATH Environment Variable" in Chapter 6 of this manual.

You can also set the BBAPATH with the **Settings→Source Directories** command.



How To Display Source Files

With the Summary or Histogram test report on screen, select a file or function by clicking on (highlighting) its line in the report. Press the "Source" button. A new window will open and it will contain the source listing of the selected file or function. The source file listing will identify the branches that were not executed (or hit) when your program executed. The first branch that was not hit will be on screen in the listing. Included will be an explanation of why the branch was not executed, preceded in parenthesis by a count of the number of branches affected because this branch was not executed. Chapter 6 in this manual has a complete description of the generated messages. All of these messages correspond to use of the **bbarep -bN** option. Use the Next and Previous keys in the Source file window to display other

branches that were not hit during the run of your executable file. Also, use the vertical scroll bar to roll the source listing in the window.

You can have the source file listing shown for a different file or function by selecting the desired file or function in the Summary or Histogram window with the *command select* mouse button or with the Next and Previous keys. Each time a new file or function is selected, its source file listing will appear in the source window. If you want to look at all of the branches in all of the functions, one at a time, select the Histogram display and use the Next button in the Histogram display to sequence through each function.

If you want to look at all of the branches missed on a file level, use the Summary display and select a file of interest. Then use the Next and Previous keys in the main Branch Validator window to obtain the next or previous files in the source window.

If you want your source file listing shown with a tab width other than TAB=8 characters, you can specify the desired number of characters to expand each tab by using "Settings: Source Tab Width" in the main BBA box. Simply enter a value between 1 and 8 in the dialog window and then redisplay your source by clicking on the desired file or function.

Ignoring A Branch In The Source Window

In the source window, you can set the BBA to ignore an unexecuted branch. You can ignore the branch in one of two ways: (1) ignore the branch directly, or (2) insert an ignore pragma.

Ignoring Branches Directly

To ignore the branch directly, you must write the file:function:branch identifier in the BBA Ignore file. To do this, highlight the branch to be ignored and click on "Ignore". This branch will now be ignored in your next summary display. The "I" indicator will appear in front of the ignored branch, and the next unexecuted branch will be displayed in the Source window.

If multiple branches are on one line, then ignoring that line will ignore all of the branches on that line. If you decide not to ignore the branch you just ignored, you can use the "Clear" button to clear the ignore from the ignore file. Select the line having the ignored branch and then click the *command select* mouse button on "Clear".

Note



If a function contains two branches with identical text (e.g. "if (k == 1)" appears, and then "if (k == 1)" appears again later in the same function), ignoring one of the branches will cause both branches to be ignored. The ignore symbol "I" will initially appear only on the branch you ignored. When you redisplay the source (by clicking again on the function name in the main BBA window), the ignore symbol will appear beside both branches. The only way to ignore one of the two branches without ignoring the other is to add an ignore pragma to the selected branch.

Ignoring Branches Using A Pragma

To ignore a branch by using a pragma, you will need to write an appropriate pragma into your source file listing. A pragma is a compiler directive that tells the BBA preprocessor to do something special with the associated branch or file. If you insert a BBA_IGNORE pragma (default case), the associated branch or function will not be instrumented for BBA, thereby ignoring the branch. For a more complete description of the various BBA pragmas and what they do, refer to "What is a pragma?" in Chapter 4 (Details Of bbacpp) in this manual.

To add a pragma to a branch, select the branch in the source file listing, and press the "Add Pragma" button. The location where the pragma will be added will be indicated by "P^". The "^" indicates that the pragma will be placed on a new line between where the P is located and the line above it. If an ignore is to be placed on the same line where a pragma is to be inserted, the symbol shown in your source list will be "PI" without the caret. In any case, the pragma will be inserted on a new line in between the line where the P is shown and the line above it. Note that the pragma is not actually added to your source file at this time. The name of the source file and the location of the pragma (line number) are simply stored in a file to be added later, when you choose **Actions→Add Pragmas to Source Files**. Therefore, you can use "Clear" to remove the pragma (provided you have not used **Actions→Add Pragmas to Source Files**).

To clear a pragma from a source line, click on either the pragma or the branch associated with the pragma, and click on "Clear". The "P^" symbol will be removed and the pragma will no longer be in the source file.

Problems Inserting Pragmas

The BBA determines the location where the pragma will be inserted by examining the preprocessor-generated ".M" files. In some cases, it is not clear where to add the pragma to ignore a particular branch. Typically, this occurs when the branch and the first line of code to be executed for the branch are on the same line. The pragma insertion program tries to add the pragma after the branch and before the first executable statement associated with the branch. If the branch and the first executable statement of the branch are both on the same line, the pragma insertion routine will not know where to add the pragma, and it will indicate that the pragma will be inserted on the line above the branch. This is typically not where you want to add the pragma. In this case, you will probably want to remove this pragma and insert it manually. To insert the pragma manually, locate a line where the pragma should be added in the associated statements of the branch and select "Add Pragma". The pragma will now be inserted one line above the selected line (provided the selected line is not another branch).

Example Of Problem Inserting Pragma

If the line where you add a pragma manually contains a branch, the pragma will be inserted at the appropriate location to ignore that branch; it will not be on the line you selected manually.



Listing showing the problem:

```
1     if (i == 10) { j = 11; /* trying to ignore this line */
2         if (k >= 10) { /* manually insert pragma here */
3             j = 12; /* P^ appears on this line */
4             m = 10;
5         }
6         z = 12;
7     }
```

In the above example, the user wanted to add a pragma to ignore the "if (i == 10)" branch. By using the "Add Pragma" command, the pragma was inserted on the line above the branch statement "if (i == 10)". This occurred because BBA could not determine exactly where to add the pragma (the first executable statement "j = 11;" was on the same line as the branch). The user then removed this pragma with the "Clear" command and tried to manually insert the pragma on the "if (k >= 10) {" line. If the "P^" appeared on this line, the desired ignore would be achieved, but instead the pragma appeared on the "j = 12;" line. This is because the line where the user manually tried to insert the pragma was a branch control statement, also. If the user left this as it is, the "if (k >= 10)" branch would be ignored and not the desired "if (i == 10)" branch. Finally, the user corrected this problem by clearing the pragma and selecting another statement in the scope of the "if (i == 10)" branch. In this case, the desired ignore can be achieved by selecting the "z = 12;" statement and manually inserting the pragma there. This will place the ignore pragma one line above the "z = 12;" statement. As long as the pragma is inserted within in the scope of the branch, it will affect the entire segment of the branch.

Listing showing the way the problem was solved:

```
1     if (i == 10) { j = 11; /* trying to ignore this branch */
2         if (k = 10) {
3             j = 12;
4             m = 10;
5         }
6         z = 12; /* manually insert pragma P^ here */
7     }
```

Exiting The Source Files Display

Use the "Close" button to close the Source window when you want to remove it from the screen. Do not use the "Close" selection in the pull down menu of your window manager because this may exit BBA.

Selecting A Set Of Active Files And Functions To Appear In Test Reports

You can identify a set of files and functions to appear in BBA test reports (excluding all others) by choosing **Settings→Active Files and Functions**. This opens a new window named "Set Active Files and Functions". All of the files and functions you have instrumented for BBA will appear in this window. Use the vertical scroll bar to examine the list. You can select a file or function in this window, and use the "Ignore" button to ignore (or remove) the selected file or function from your test report. The line you selected will show the "Ign->" symbol, indicating that the selected file or function has been added to the BBA Ignore file. If you ignore a file, all of the functions of the ignored file will also be ignored, and this will be indicated by (Ign->), an implied ignore. The (Ign->) symbol indicates that the function will be ignored even though it is not listed in the BBA Ignore file.

If you change your mind and decide to have a previously ignored file or function included in your BBA report, select the line with that file or function and press "Clear". Whenever you press either "Clear" or "Ignore", the selecting line on the display advances to the next function or file. If the selected line is a function with an implied ignore "(Ign->)", you cannot clear it this way. You'll have to clear its file.

If you only want to examine BBA coverage on one or two files or functions in a long list, use the "Ignore All" button to add all of the files and functions to the BBA Ignore file. Then use "Clear" to clear only the desired files and functions from the ignore file. Note that you must clear a file before you can clear the functions within the file.

The "Clear All" button will remove all of the files and functions from the BBA Ignore file. No file or function will be ignored (including any you previously ignored through the main Branch Validator window and the source window).

Use the "Close" button to close the Active Files and Functions window when you want to remove it from the screen. Do not use the "Close" selection in the pull down menu of your window manager because this will exit BBA. This window may also be iconified separately from the main Branch Validator window, if desired.

Displaying Error And Warning Messages

The BBA Errors window will open when you click on the "Errors" button in the main Branch Validator window. The Errors window lists all of the error and warning messages that have been generated since executing the most recent BBA command. Serious error messages will bring this window up, automatically. If you do not see what you expect to see when you execute a BBA command, click on the "Errors" button to see if any errors have occurred while executing your last command. All of the commands associated with BBA (including the user-definable commands in the Actions pull down) are capable of writing errors to this window. For help in understanding a BBA error message, refer to Appendix A near the end of this manual.

How To Ignore A File, Function, Or Branch

Pressing the "Ignore" button in the main Branch Validator window, Source window, or Active Files and Functions window will ignore the selected file, function, or branch. In the main Branch Validator window, pressing the "Ignore" button after selecting a file or function will write the specified file or function to the BBA Ignore file. Note that the display will indicate that a function is ignored by placing the string "Ignore" in the same line as the function. If you ignore a file, all of the functions in the file will be preceded by the string "Ignore" to indicate that they are ignored.

The Active Files and Functions window allows you to define a set of files and functions to include in the BBA test report. All files or functions in this window that begin with "Ign->" are listed in the BBA Ignore file. You can add or remove files and functions within the ignore file by pressing "Ignore" and "Clear".

The Source window allows you to ignore branches and functions. Simply select the desired branch and click on "Ignore". You can remove these branches from the BBA Ignore file by selecting the desired branch again, and clicking on "Clear".

Commenting Ignore Entries

You may add comments to each file, function, and branch that is ignored by choosing **Settings→Ignore→Comment Type**. The three types of comments you can include with each ignore are: "No comment", "Universal comment", and "Unique comments". By selecting "No comment", you turn off the commenting feature and do not include a comment with an ignore entry. By selecting "Universal comment", your selected comment text will be included with each new Ignore entry in the ignore file. By selecting "Unique comments", BBA will query you after each new "Ignore" selection for a comment to add to the ignore statement in the BBA Ignore file. All Ignore comments should begin with a "#" symbol as the first character. If your comment does not begin with a "#" symbol, a "#" symbol will be added automatically.

The Ignore File

The appropriate statements required to ignore files, functions, and branches are stored in the BBA Ignore file, which by default is defined to be \$HOME/BBA_IGNORE_FILE. You can rename the BBA Ignore file, if desired, by either choosing **Settings→Ignore→File...** or by defining the configuration file option "BBA_IGNORE_FILE".

You may edit the ignore file to remove or add the names of files, functions, and branches, provided you leave the file content in a format usable by BBA. If BBA does not understand a particular statement in the BBA Ignore file, that statement will be treated as a comment. To edit the BBA Ignore file, choose **Edit→Ignore File**.



Format Of Ignore File Contents

The contents of the BBA Ignore file can include files, functions, branches, and comments. The format of the contents in the Ignore file is important if you desire the HP Branch Validator (BBA) SoftBench User Interface to use this file properly. In order to ignore a file, enter the complete path of the file. To ignore a function, enter the complete path of the file that contains the function, followed by a ":", and then followed by the function name. To ignore a branch, enter the full path of the file, the function, and the branch-control statement, all separated by ":". Refer to Chapter 4 for an explanation of the branch-control statements required to ignore a particular type of branch. The comment associated with a file, function, or branch will then appear immediately following the branch in the file.

Consider the following Ignore File segment:

```
/usr/hp64000/demo/bba/demo2/driver.c : error
# Routine is a hardware driver, don't need to test
/usr/hp64000/demo/bba/demo2/convert.c : bitpos : case 128 :
# Case impossible to reach
/usr/hp64000/demo/bba/demo2/multibits.c
```

From the above segment, you can see that the function "error", the branch "case 128:" and the file "multibits,c" are being ignored. Note that the function and the branch statements have comments added. Also note that the contents of the BBA Ignore file can be in any order, except that the comment statement about a file, function, or branch should follow the file, function, or branch.

Note



When not using the HP softBench interface, BBA will allow you to enter ignore-file statements without the complete scoping of the file and function and without the complete path to the source file. Bbarep will work properly with both unscoped and fully scoped ignore statements. The HP Branch Validator (BBA) SoftBench User Interface will not work properly unless each ignore-file entry includes complete scoping, as shown above.

Adding Pragmas

To add a pragma to a branch, open the Source window, select the desired branch, and click on "Add Pragma". The pragma will be added at the indicated line (refer to the discussion about the source window, earlier in this chapter). This step actually adds a pragma tag (or pragma) to the \$HOME/BBA_PRAGMA_FILE. Pragma tags are read when you choose **Actions**→**Add Pragmas to Source Files**, and the pragmas are actually inserted at that time.

The name of the pragma tag file is \$HOME/BBA_PRAGMA_FILE. You may rename the BBA Pragma file to a name of your choice by either choosing **Settings**→**Pragma**→**File...** or by defining the configuration file option "BBA_PRAGMA_FILE" in your .bbarc file.

Four types of pragmas can be added to a particular branch. They are: BBA_IGNORE, BBA_ALERT, BBA_IGNORE_ALWAYS_EXECUTED, and BBA_IGNORE plus BBA_ALERT in combination. To specify the type of pragma to be included with a particular branch, choose **Settings**→**Pragma**→**Type** and enter your selection. Refer to Chapter 4 for a discussion of why you might want to enter pragmas, and the various types of pragmas available. In addition, you may add comments to each pragma.

Comments are inserted by choosing **Settings**→**Pragma**→**Comment Type**. The three types of comments you can include with each pragma are "No comment", "Universal comment", and "Unique comments". Selecting "No comment" turns off the commenting feature and does not add a comment to each pragma entry. Selecting "Universal comment" causes BBA to ask you to enter a comment that it can include with each subsequent pragma selection. Selecting "Unique comments" causes BBA to query you after each use of "Add Pragma" for a unique comment to include with the pragma in the source file. All pragma comments must begin with "/*" and end with "*/", like normal C comments.

The BBA Pragma file stores the results of using the "Add Pragma" command, along with the pragma comments. You may edit this file and remove pragmas or add new pragmas, provided you leave the file content in a format usable by BBA. To edit the BBA Pragma file, choose **Edit**→**Pragma File**.

The format of each statement in the BBA Pragma file includes: the full path name of the file, the line number where the pragma will be added, the type of pragma to add (I,E,A or C), and a comment to be included with the pragma, if desired. Each of these fields is separated by a ":" and there are no blank spaces.

Note



This format is critical. If it is not followed exactly, the associated pragmas will not be added to your source files properly. Consider the following Pragma file segment:

```
/usr/hp64000/demo/bba/demo2/driver.c:105:I:  
/usr/hp64000/demo/bba/demo2/convert.c:79:A:/* Let me know if enters */  
/usr/hp64000/demo/bba/demo2/convert.c:77:C:/* Case not possible */
```

From the above segment, you can see that the file "driver.c" will have a BBA_IGNORE pragma added before line 105. The file convert.c will have two pragmas added (a BBA_ALERT before line 79 and a combination of BBA_ALERT and BBA_IGNORE before line 77). Note that the Pragma file entry shows only the file and line number where the pragma will be added. If the source file is edited and the line numbers change before you insert the pragmas with the **Actions→Add Pragmas to Source Files** command, the pragmas will not be inserted in the proper locations. Edit the Pragma file and delete all pragmas that are to be inserted into the modified file before you actually add the pragmas to the other non-edited files.

The routine that actually adds the pragmas is controlled by the script file assigned to the configuration file option BBA_ADD_PRAG_COMMAND in the .bbarc configuration file. When you execute this command (by using **Actions→Add Pragmas to Source Files**), it will display a terminal window and show you the results of adding the pragmas to each of the files. When the command is complete, you can roll this window to examine the results of adding the pragmas. The script can be edited to check-out versioned files before adding the pragmas, to log the results of adding the pragmas to a file, or to more(1) the resulting files to the screen, if desired. The only thing that BBA needs back from executing this script is a list of the files where pragmas have been successfully

added. This list of files is used to delete the pragma tags in the BBA Pragma File.

Using Build

The BBA Build feature provides an automated method of building an up-to-date executable file to run in your BBA test environment. BBA Build operates in one of two ways, depending on how you have set the BBA_USE_SOFTBENCH option in your \$HOME/.bbarc configuration file. If BBA_USE_SOFTBENCH=True, choosing **Actions**→**BBA Build** will open a series of dialog boxes where you enter a build directory, Makefile name, and Makefile options. The default value for the build directory is your current context directory. The default values for the Makefile name and Makefile options are read from your BBA_MAKEFILE_NAME and BBA_MAKEFILE_OPTIONS options in your configuration file (.bbarc).

If you have the AxLS C-Cross BBA, and you have the HP SoftBench Builder properly set up to build your Cross Compiler executables, you could enter the appropriate Build directory, Makefile name, and Makefile options, and have the HP SoftBench Builder create an executable file for you.

If you are not using the SoftBench Broadcast Message Server (BBA_USE_SOFTBENCH=False), choosing **Actions**→**BBA Build** will execute the command defined by option BBA_BUILD_COMMAND in your configuration file. No questions will be asked and no options will be passed. Be sure to define this command exactly as you want it to occur. This command will be passed to a hidden shell exactly as you have it defined. Error messages from the hidden shell will be sent to the Errors window. If you want to see the complete results of the command, you may want to create a script file similar to the following:

```
make -f Makefile bbatest > /tmp/results 2 > &1
hpterm -e more /tmp/results
mailx -s "Build Results" clarkkent < /tmp/results
```

This command will save the results of the make in a temporary file, and then more(1) the build results to the screen. In addition, it will mail the results to the specified user "clarkkent". Once the build is complete, you will need to rerun your executable to obtain the new BBA dump data.

Note



Do NOT add a change directory command directly to a BBA_BUILD_COMMAND option string. You should enter only a simple command or a script command.

You can have a change directory command in the script command, if desired. Do NOT do this:

```
"BBA_BUILD_COMMAND=cd /users/clarkkent; make bbatest"
```

How To Control A Run Of Your Test

The Actions pull down offers the following three user-definable actions: "Run Test", "Stop Test", and "Modify Test". When you click on one of the three user-definable pull down items, the user-defined command in your configuration file (.bbarc) will be executed. Refer to table 9-1 to see which command option runs when a pull down menu item is selected.

Table 9-1. User-Definable Commands

Actions Pull Down Menu	Command To Run In Hidden Shell
Run Test	BBA_RUN_TEST_COMMAND
Stop test	BBA_STOP_TEST_COMMAND
Modify Test	BBA_MOD_TEST_COMMAND

Each of the commands shown in table 9-1 will be executed in a hidden shell (no standard output results will be sent to the screen). Standard error messages will be sent to the BBA-Errors window. Up to three parameters will be passed to each of the commands, depending on what you have selected on your screen. If you have selected a file in the Summary display, each of the above commands will be passed the name of the file as the first parameter. If you have selected a function in the Summary or Histogram display, your command will be passed the file name as the first parameter and the function name as the second parameter. If you have selected a line of source code in the Source window, your command will be passed the line number as the third parameter. You can do anything with these parameters that you want. They may be used as qualifiers for running, stopping, or modifying your test. The mechanism for doing this qualification is user definable; it depends on your tests, file locations, etc. You may be able to implement the qualifications with a look-up table.

Accessing A File To Edit

The Edit pull down menu offers a method for editing a function or file listed in the Summary or Histogram test report, editing the Ignore file, the Pragma file, the Configuration file (\$HOME/.bbarc), or a file of your own choosing. If you are using the SoftBench Broadcast Message Server (BBA_USE_SOFTBENCH=True), your edit menu selection will be passed to the SoftBench Broadcast Message Server as a request to edit the specified file at the specified line number.

If you are not using the SoftBench Broadcast Message Server (BBA_USE_SOFTBENCH=False), your edit menu selection will invoke the command defined by BBA_EDIT_FILE_COMMAND. This command will be passed a file name as the first parameter and a line number as the second parameter. By default, an Edit menu selection (with BBA_USE_SOFTBENCH=False) will invoke the vi editor in a terminal window. If this is not the editor you want to use, you can enter a new edit command in the script file **\$HP64000/bba/config/cmd_edit**, or you can modify the configuration file option BBA_EDIT_FILE_COMMAND to invoke your desired editor.

If you would like to edit a source line that appears in the Source window, click on "Edit: Selected Summary File". The source line number will be passed along with the edit request. A new window will open. It will show the file that is in the Source window, and be positioned directly to the line of interest.

If you edit the configuration file (\$HOME/.bbarc), BBA will not automatically reread your new version of configuration file variables. Use "Settings: Read Configuration File" to make BBA reread the values of the configuration file.

Using Print And Save

The **File**→**Print**→ and **File**→**Save in file**→ commands allow you to document your BBA test results. You can print the content of the "Summary", "Histogram", "Results Only", "File History", "Source of Selection" or "Active Files and Functions" displays, or you can save the selected contents in a file. Each of these results is identical to the corresponding display you can obtain on your screen.

Note



The files you print or save may not match the display on screen. The resulting outputs are created fresh with each execution of the "Print" or "Save" command; thus, they will not match exactly what is on your screen if your screen has not been updated to contain the latest information.

The Print command creates a file of the requested output and then passes its file name as the first parameter to the command defined by `BBA_PRINT_COMMAND`. By default, the `BBA_PRINT_COMMAND` executes a script, "`$HP64000/bba/config/cmd_print`" which executes the command `lp $1` (where `$1` is set equal to the requested file name). The results of this print command are then captured and presented to the screen in a small dialog box. If you do not want this small dialog box coming up on the HP Branch Validator (BBA) SoftBench User Interface, you can send the results of the print operation to `/dev/null`. In this case, no dialog box will be displayed.

The "Save in file" command appends the requested output to a file of your choice. If the file does not exist, it will be created. Each "Save in file" command will append the current output to the end of the contents of the file.

Setting The Dump Data File And Retaining Old Data

In order for BBA to operate properly, the location of the bbadump.data file must be specified. You can specify its location in one of two ways: (1) specify the dump data file directly, or (2) modify BBA_DUMP_DATA_FILE in your .bbarc configuration file.

To directly specify the location of the dump data file, choose **Settings→Dump Data File...** This opens a dialog box where you can specify the directory and the name of the BBA dump data file (by default, called "bbadump.data"). After entering the name of the file, select "Update" to get a new test report display. If the specified dump file does not exist, you will get an appropriate error message in the Errors window.

Note



Entering a new BBA dump data file name will not automatically update your screen. Select the desired type of test report, or click on "Update" to view the summary from the new BBA dump data.

The other method of selecting a different bbadump.data file involves editing your .bbarc configuration file to redefine the BBA_DUMP_DATA_FILE option. If you specify a file name without a complete path, BBA will expect to find the BBA dump file in the current context where the BBA tool is running. Note that you cannot specify an incomplete path dump file by choosing the **Settings→Dump Data File...** (method discussed above). The dialog box will always return a full path file name, even if you only put in a partial pathname file.

Choose **Settings→Retain Old Dump Data** to retain older dump data along with more recent dump data when producing the BBA reports. Refer to "bbarep -o" (in Chapter 6) to understand more about why you might want to use this option.

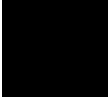
Using Help

Help windows are only available if you are using the SoftBench Broadcast Message Server (BBA_USE_SOFTBENCH=True), and the Message Server is running. If you choose **Help**→**Application Help**, a window opens to show a complete listing of all "help" topics available for the BBA. If you choose **Help**→**Item Help**, a window opens to describe how to get help on individual screen items. You can get item help about any button, any pull down menu item, and most of the display regions. For more information about the Help facility, refer to the manual titled, "Exploring HP SoftBench: A Beginner's Guide".

Exiting BBA

To exit BBA, choose **File**→**Quit**. If you are using the default keyboard accelerators, type <Shift><Alt>Q. You can quit BBA at any time and return later to resume where you left off. The files, functions, and branches you have ignored will be retained in the BBA Ignore file. The pragmas you were prepared to insert will be retained in the BBA Pragma file.

There is only one potential problem when quitting BBA. When the BBA interface exits, it will attempt to kill all windows that are associated with the BBA interface. This includes any windows or commands that were started by using any of the Actions menu items. In addition, if you are NOT using the SoftBench Broadcast Message Server, this kill signal will be sent to any edit processes and build processes that were started from the BBA interface. This kill signal will abort any of the above listed Actions or Edits unless "set -m;" was issued as part of the command. By using "set -m;", BBA is prevented from exiting until the specified command completes. In this case, BBA will appear to hang until the command completes. If this happens, you may want to close and exit the windows that BBA is waiting for, or you may want to iconify BBA to get it out of the way and allow the action in the BBA initiated window to complete.



Restrictions When Using The HP Branch Validator (BBA) In The HP SoftBench Interface

Within the HP SoftBench User Interface, BBA uses the `bbarep` command to obtain test results to display. Consequently, there are a few restrictions and limitations imposed by BBA on the `bbarep` and `bbacpp` commands.

1. You cannot use multiple map files with the `bbacpp -DBBA_OPTM` option. In this interface, BBA assumes that all preprocessor-generated map files are labeled with a ".M" file name extension.
2. BBA will not work correctly if a file name contains a ":" character as part of the filename.
3. BBA may not always find the proper location to insert pragmas. Pragma insertion will be a problem when the pragma needs to be inserted in the middle of a statement, or when a branch and the first executable statement of the branch are both on the same line.
4. If you add pragmas to a branch and then later add pragmas to the same branch before choosing **Actions→Add Pragmas to Source Files**, then the last (and only the last) pragma added to the branch will actually be inserted into the source file. Stated differently, if a branch contains multiple pragma tags, only the most recent tag will actually be added to the branch when you choose **Actions→Add Pragmas to Source Files**.
5. When you ignore a branch by using a statement in the ignore file, both the if and the else conditions of the branch will be ignored. This is identical to the operation of `bbarep`.

- Alert warnings are placed before the affected branch. This sometimes makes it difficult to determine which portion of a multiple-branch statement activated the Alert warning. For example:

```
*** code with BBA_ALERT was executed
->53   if ((horiz == 0) && (vert == 0))
54     {
55         if (k == 12)
56         {
57             j = 12;
58             # pragma BBA_ALERT
59         }
60         return(KEY_NONE);
61     }
62     else
63     {
64         # pragma BBA_ALERT
65     }
```

In the above listing, the BBA Alert warning was activated by line 64. If the warning had been activated by line 58, then the warning would be placed immediately before the branch on line 55.

- Yacc and Lex source files will not work properly with BBA. Bbarep will work better with yacc and lex output, but neither will show you the original yacc and lex source where the branches were generated.
- Dialog windows will always pop the main Branch Validator window to the top of the display when using the Motif Window Manager (Mwm). If this behavior disturbs you, you will have to position the Branch Validator source and main windows in such a manner that the effect is minimized, or use a different window manager.
- Pressing the keyboard accelerators one after another too rapidly (such as <Shift><Alt>H for a histogram display, depending on your version of HP SoftBench software) may cause the display to become out of sync. Wait until the display is valid before pressing the next keyboard accelerator.

10. If the HP Branch Validator (BBA) SoftBench User Interface is exited by a non-standard method (power failure, kill signals, etc.), temporary files may not be properly removed. When this happens, HP suggests that you remove the **bba*** and **analc*** files in the **/usr/tmp** and **/tmp** directories.



Error And Warning Messages

Introduction

This appendix contains a list of error and warning messages that you may see when using the BBA. The list in this appendix is separated into the following three sections:

1. Messages that may be generated during preprocessing.
2. Status-line messages that may appear while using the **bbaunload** command with an emulator, and journal window messages that may appear when using the **Unload_BBA** command with a debugger.
3. Messages that may be generated when running **bbarep**.

Conventions used in this appendix are described below:

1. **<sfx>** stands for a specific single character. In the actual error message, it will be replaced by the actual suffix.
2. **<mapfile>** stands for the name of a map file (such as `/hp/goodyear/blimp.M`).
3. **<file>** stands for a file name. The type of file depends on the context of the message.
4. **<flag>** stands for a command-line flag (such as **-DBBA_OPTS**).
5. **<num>** stands for a decimal integer (such as **64**).
6. **<line>** stands for a line number in **<file>**.

7. **<sym>** stands for a C symbol.

8. **<date>** stands for a date and time, in the current time zone.

bbacpp Messages

Bbacpp<COMP> always sends its messages to stderr. In addition to the error messages that cpp<COMP> can generate, bbacpp<COMP> can generate the following messages:

Note



In place of "bbacpp<COMP>", substitute your specific compiler number, e.g., bbacpp68000, bbacpp8086, bbacpp68030, etc.

```
<file>: <line>: BBA_IGNORE_ALWAYS_EXECUTED is not immediately within a while loop
```

This message indicates that a `BBA_IGNORE_ALWAYS_EXECUTED` pragma is not at the same scope as a while statement. Move the pragma that is at line <line> so that it is within the control of the while statement, but not within the control of any branches that are within the if statement. For example:

```
15     while(i != 0)
16     {
17         if (j == i)
18         {
19             #           pragma BBA_IGNORE_ALWAYS_EXECUTED
20                 j++;
21         }
22     }
```

is incorrect, and should be changed as shown in one of the two following examples:

Example 1:

```
15     while(i != 0)
16     {
17 #           pragma BBA_IGNORE_ALWAYS_EXECUTED
18           if (j == i)
19           {
20               j++;
21           }
22     }
```

Example 2:

```
15     while(i != 0)
16     {
17           if (j == i)
18           {
19               j++;
20           }
21 #           pragma BBA_IGNORE_ALWAYS_EXECUTED
22     }
```

"<file>", <line>: Illegal operand type combination for '?:'.

This message actually comes from cc<COMP>. If you do not normally see this message when you compile your program, refer to Chapter 4 for more information about "the -DBBA_OPTO=a option".

<file>: <line>: ERROR: cscanorigtext: unexpected \\0 when looking for a \\002

This message should never happen. It implies that the temporary file was truncated between passes.

```
<file>: <line>: No string delimiter
```

A string was started (with either a single quote or double quote), and an end-of-line was found before the matching quote was found. Note that you can end a line with a backslash (\) and continue a string on a following line with no problems.

```
<file>: <line>: Out of push-back-line memory; use -DBBA_OPTpNNN, where NNN is greater than <num>
```

Refer to Chapter 4 for more information about DETAILS OF THE -DBBA_OPTpNNN OPTION.

```
<file>: <line>: Syntax error
```

bbacpp has detected a syntax error in **<file>** at line **<line>**. Use the cc<COMP> compiler without the **-b** option for a more complete syntax/semantic check (bbacpp does not do a semantic check).



```
<file>: <line>: Too many syntax errors
```

bbacpp detected too many syntax errors, and is giving up its attempt to parse the file.

A-4 Error And Warning Messages

<file>: <line>: Unexpected end of comment

The end of the C source file was found before the end of a comment.

<file>: <line>: Unknown preprocessor directive

A line beginning with # was found that bbacpp could not parse. Note that

pragma <anything>

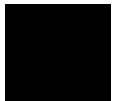
is always legal!

<file>: <line>: symbol '<sym>' conflicts with bba symbol

The symbol <sym> begins with **_bA_**, which is reserved for use by the BBA. You can either change the name of the symbol, or choose not to use bbacpp on files that reference that symbol.

<file>: <line>: Yacc stack overflow

This message indicates that an expression on line <line> has become too complex. Simplify your expression.



```
bbacpp<COMP>: ERROR: cgets: unexpected EOF while searching for a \003
```

This message can happen if you have the illegal characters ^A, ^B, or ^C in your file (probably in comments or strings).

```
bbacpp<COMP>: cannot generate map file '<mapfile>' because that would overwrite an  
unknown-type file (use the -DBBA_OPTM<suffix> to choose a suffix different from '<sfx>')
```

This message is warning you that the mapfile will not be generated. Refer to Chapter 4 for the discussion of the DETAILS OF THE -DBBA_OPTS OPTION for the implications of not having a map file. The mapfile is not generated because a file already exists that has the same name as the mapfile, and this already existing file is not a BBA mapfile.

```
bbacpp<COMP>: cannot generate map file '<mapfile>' because that would overwrite the  
source file (use the -DBBA_OPTM<suffix> to choose a suffix different from 'c')
```

This message warns you that the mapfile will not be generated. Refer to Chapter 4 for the discussion of DETAILS OF THE -DBBA_OPTS OPTION for the implications of not having a map file.

```
bbacpp<COMP>: cannot generate map file because file name '<file>' is too long to append
.<sfx>
```

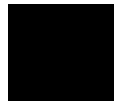
This message warns you that the mapfile will not be generated. Refer to Chapter 4 for the discussion of DETAILS OF THE -DBBA_OPTS OPTION for the implications of not having a map file. The mapfile is not generated because the file has no extension and there are not enough characters left in the file name to uniquely append a suffix.

```
bbacpp<COMP>: cannot open map file <mapfile> for output
```

This message warns you that the mapfile will not be generated. Refer to Chapter 4 for the discussion of DETAILS OF THE -DBBA_OPTS OPTION for the implications of not having a map file. The mapfile is not generated because a file already exists that has the same name as the mapfile, but the permissions of the existing file will not allow bbacpp to write to it.

```
bbacpp<COMP>: cannot read file offset
```

This message should never happen. It implies that the temporary file was truncated between passes. If this occurs, somebody else probably deleted the file accidentally while you were running bbacpp<COMP>. Try again.



```
bbacpp<COMP>: cannot reposition input
```

This message should never happen. It implies that the temporary file was truncated between passes.

```
bbacpp<COMP>: could not open <file> for output
```

The post-cpp file that bbacpp<COMP> was requested to create is write protected. This should not happen, because cc<COMP> requests output to a file in /usr/tmp or /tmp (and temporary files should never be write protected).

```
bbacpp<COMP>: exceeded maximum scoping (<num>)
```

This message states that more nested blocks were found than could be handled by bbacpp<COMP>. This should not happen because bbacpp can handle the same number of nesting as cc<COMP>.

```
bbacpp<COMP>: pass2 could not open pass1's output file
```

bbacpp<COMP> operates in two passes. The first pass does the cpp processing, and the second pass parses the post-cpp output. The output of the first pass is stored in a temporary file in the directory /usr/tmp or /tmp. If this message appears, it implies that something has deleted the temporary file (which should never happen).

A-8 Error And Warning Messages

bbacpp<COMP>: unable to write output file

bbacpp<COMP> attempted to write to a file and failed, either due to the file system being full or to a hardware (disk I/O) error.

bbacpp<COMP>: unknown flag <flag>

An unknown flag was used on the command line and passed to bbacpp<COMP>. This does not affect processing. It is an advisory message, only.



BBA Unload Messages

The **bbaunload** command in emulation always sends its error messages to the status line. The **Memory Unload_BBA** command in the debugger always sends its error messages to the journal window.

`<file> can't be appended`

The file "bbadump.data", or dumpfile **<file>**, is write protected.

`<file> can't be opened`

(Emulator Only) This shows that **<file>** (which is your absolute file) could not be opened. Since it must be valid for the emulator to have loaded your absolute, either you have entered an incorrect file name, or the file has been deleted.

`<file> is not currently loaded`

(Debugger Only) This shows that **<file>** (which is the base name of your absolute file) is not loaded. Most often, this is the result of an incorrectly spelled file name.

Can't look up name of symbol (<sym>)

This should not happen. It shows that the symbol database is corrupt.

Can't search the symbol database

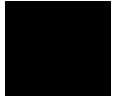
This should not happen. It shows that the symbol database is corrupt.

Incorrectly formatted dump file

The file "bbadump.data", or dumpfile **<file>**, is either not a dumpfile, or parts of the file have been deleted.

No root symbol

This should not happen. It shows that the symbol database is corrupt.



dump protocol <num> is newer than known protocol <num>

The version of bbacpp is newer than the version of your emulator or debugger. You must get a new version of the emulator or debugger software (or use an older version of bbacpp).

insert protocol <num> is newer than known protocol <num>

The version of bbacpp is newer than the version of your emulator or debugger. You must get a new version of the emulator or debugger software (or use an older version of bbacpp).

spec file is corrupt

The file \$HP64000/lib/<COMP>/bbacpp.spec has been modified.
Reload it from the product tape.



bbarep Messages

Bbarep always sends its error messages to stderr.

```
bbarep: <file>: <line>: error in dump time
```

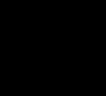
This shows that the bbadump.data file **<file>** became corrupt between the time the BBA unload command wrote it, and bbarep tried to read it. bbarep cannot generate any reports when this happens.

```
bbarep: <file>: <line>: expected ':array' line; did not find it
```

This shows that the bbadump.data file **<file>** became corrupt between the time the BBA unload command wrote it, and bbarep tried to read it. bbarep cannot generate any reports when this happens.

```
bbarep: <file>: <line>: illegal ':array' line
```

This shows that the bbadump.data file **<file>** became corrupt between the time the BBA unload command wrote it, and bbarep tried to read it. bbarep cannot generate any reports when this happens.



```
bbarep: <file>: <line>: illegal ':dump' line
```

This shows that the bbadump.data file **<file>** became corrupt between the time the BBA unload command wrote it, and bbarep tried to read it. bbarep cannot generate any reports when this happens.

```
bbarep: <file>: <line>: illegal ':file' line
```

This shows that the bbadump.data file **<file>** became corrupt between the time the BBA unload command wrote it, and bbarep tried to read it. bbarep cannot generate any reports when this happens.

```
bbarep: <file>: <line>: no ':dump' line
```

This shows that the bbadump.data file **<file>** became corrupt between the time the BBA unload command wrote it, and bbarep tried to read it. bbarep cannot generate any reports when this happens.



```
bbarep: <file>: <line>: no ':dump' line, or ':dump' line out of order
```

This shows that the bbadump.data file **<file>** became corrupt between the time the BBA unload command wrote it, and bbarep tried to read it. bbarep cannot generate any reports when this happens.

A-14 Error And Warning Messages

```
bbarep: <file>: <line>: no ':file' lines; no branch data in dump file
```

This shows that the bbadump.data file **<file>** became corrupt between the time the BBA unload command wrote it, and bbarep tried to read it. bbarep cannot generate any reports when this happens.

```
bbarep: <file>: <line>: unexpected :array line
```

This shows that the bbadump.data file **<file>** became corrupt between the time the BBA unload command wrote it, and bbarep tried to read it. bbarep cannot generate any reports when this happens.

```
bbarep: <file>: <line>: unexpected end of file
```

This shows that the bbadump.data file **<file>** became corrupt between the time the BBA unload command wrote it, and bbarep tried to read it. bbarep cannot generate any reports when this happens.

```
bbarep: <file>: <line>: unknown line
```

This shows that the bbadump.data file **<file>** became corrupt between the time the BBA unload command wrote it, and bbarep tried to read it. bbarep cannot generate any reports when this happens.



```
bbarep: <mapfile>: <line>: error in modification date
```

This shows that the mapfile became corrupt between the time bbacpp wrote it and bbarep tried to read it. This causes bbarep to act as if the **<mapfile>** does not exist. Refer to Chapter 4 for the discussion on DETAILS OF THE -DBBA_OPTS OPTION for the implications of not having a map file.

```
bbarep: <mapfile>: <line>: illegal ':option' line
```

This shows that the mapfile became corrupt between the time bbacpp wrote it and bbarep tried to read it. This causes bbarep to act as if the **<mapfile>** does not exist. Refer to Chapter 4 for the discussion on DETAILS OF THE -DBBA_OPTS OPTION for the implications of not having a map file.

```
bbarep: <mapfile>: <line>: illegal ':probe' line
```



This shows that the mapfile became corrupt between the time bbacpp wrote it and bbarep tried to read it. This causes bbarep to act as if the **<mapfile>** does not exist. Refer to Chapter 4 for the discussion on DETAILS OF THE -DBBA_OPTS OPTION for the implications of not having a map file.

A-16 Error And Warning Messages


```
bbarep: <mapfile>: <line>: illegal ':source' line
```

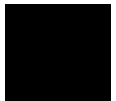
This shows that the mapfile became corrupt between the time bbacpp wrote it and bbarep tried to read it. This causes bbarep to act as if the **<mapfile>** does not exist. Refer to Chapter 4 for the discussion on DETAILS OF THE -DBBA_OPTS OPTION for the implications of not having a map file.

```
bbarep: <mapfile>: <line>: missing at least one :probe line
```

This shows that the mapfile became corrupt between the time bbacpp wrote it and bbarep tried to read it. This causes bbarep to act as if the **<mapfile>** does not exist. Refer to Chapter 4 for the discussion on DETAILS OF THE -DBBA_OPTS OPTION for the implications of not having a map file.

```
bbarep: <mapfile>: <line>: no ':options' line
```

This shows that the mapfile became corrupt between the time bbacpp wrote it and bbarep tried to read it. This causes bbarep to act as if the **<mapfile>** does not exist. Refer to Chapter 4 for the discussion on DETAILS OF THE -DBBA_OPTS OPTION for the implications of not having a map file.



```
bbarep: <mapfile>: <line>: no ':protocol' line, or ':protocol' line out of order
```

This shows that the mapfile became corrupt between the time bbacpp wrote it and bbarep tried to read it. This causes bbarep to act as if the **<mapfile>** does not exist. Refer to Chapter 4 for the discussion on DETAILS OF THE -DBBA_OPTS OPTION for the implications of not having a map file.

```
bbarep: <mapfile>: <line>: unknown control line
```

This shows that the mapfile became corrupt between the time bbacpp wrote it and bbarep tried to read it. This causes bbarep to act as if the **<mapfile>** does not exist. Refer to Chapter 4 for the discussion on DETAILS OF THE -DBBA_OPTS OPTION for the implications of not having a map file.

```
bbarep: Unable to read protocol level (<num>) of dump file <file> (current maximum  
protocol level for dumpfiles is <num>)
```



This shows that the BBA unload command generated a dumpfile that bbarep could not read because the software version of your emulator or debugger is newer than the software version of bbarep.

A-18 Error And Warning Messages

```
bbarep: Unable to read protocol level (<num>) of map file <mapfile> (current maximum
protocol level for mapfiles is <num>)
```

This shows that bbacpp generated a mapfile that bbarep could not read because the version of bbacpp is newer than the version of bbarep. This causes bbarep to act as if the **<mapfile>** does not exist (because you did not invoke bbarep with the **-o** option). Refer to Chapter 4 for the discussion on **DETAILS OF THE -DBBA_OPTS OPTION** for the implications of not having a map file.

```
bbarep: cannot derive a mapping file name for <file>
```

This message warns you that bbacpp was run on a file for which no mapping file could be generated (you would have also been warned of this while bbacpp was running).

```
bbarep: cannot find <mapfile>, or it is not a mapping file
```

This implies that the mapfile either does not exist, is not readable, or is not of the correct type.

```
bbarep: cannot open <file>, or it is not a Basis Branch Analysis dumpfile
```

Either **<file>** does not exist, it is not readable, or it is not the correct format for a bbadump.data file.

```
bbarep: error: cannot merge data for <file> (different -O= options have identical
map-suffixes; no analysis done for file)
```

The bbadump.data file contains at least two entries for **<file>** which show different -DBBA_OPTO= options, but the map suffix for different -DBBA_OPTO= options are identical. This means that one of the mapfiles was overwritten, so there is no reliable way to merge the data. Bbarep will act as if the file was not compiled using bbacpp.

```
bbarep: error: cannot merge data for <file>
(no map-suffix information; no analysis done for file)
```

The bbadump.data file contains at least two entries for **<file>** which show different -DBBA_OPTO= options. However, at least one of the mapfiles does not exist (or isn't readable), so there is no reliable way to merge the data. Bbarep will act as if the file was not compiled using bbacpp.

```
bbarep: warning: ignoring mapsuffix of '<sfx>' for source file '<file>'
```



The bbadump.data file contains at least two entries for **<file>**. Both of these entries show that **<file>** had the same -DBBA_OPTO= options, both have the same number of branches, and both have the same modification date, but they have different mapfile suffixes. In this case, bbarep will search for a mapfile with the first suffix, and ignore the mapfile with the second suffix.

A-20 Error And Warning Messages

```
bbarep: warning: mapfile '<mapfile>' has different number of probes
than the dump file shows; some functions disallowed for <file>
```

This shows that **<mapfile>** is out-of-date with respect to data in the bbadump.data file. This causes bbarep to act as if the **<mapfile>** does not exist. Refer to Chapter 4 for the discussion on DETAILS OF THE -DBBA_OPTS OPTION for the implications of not having a map file.

```
bbarep: warning: mapfile '<mapfile>' shows different source date than dumpfile
```

The source modification date stored in the **<mapfile>** is different from the date stored in the dumpfile; however, the number of branches detected and the options used to create the mapfile are the same. This warns you that some data may not be correct, but you invoked bbarep with the **-o** option (forcing it to retain the old data).

```
bbarep: warning: mapfile '<mapfile>' shows different source date than dumpfile
some functions disallowed for <file>
```

The source modification date stored in the **<mapfile>** is different from the date stored in the dumpfile; however, the number of branches detected and the options used to create the mapfile are the same. This causes bbarep to act as if the **<mapfile>** does not exist (because you did not invoke bbarep with the **-o** option). Refer to Chapter 4 for the discussion on DETAILS OF THE -DBBA_OPTS OPTION for the implications of not having a map file.



```
bbarep: warning: skipping data from file <file> showing <num> branches
date of both files is the same
```

The bbadump.data file contains at least two entries for **<file>**. Both of these entries show that **<file>** had the same -DBBA_OPTO= options, and both have the same modification date, but they have a different number of branches. In this case, the data associated with the fewer number of branches is ignored.

```
bbarep: warning: skipping data from file <file> showing <num> branches
date of file skipped is <date>
```

The bbadump.data file contains at least two entries for **<file>**. Both of these entries show that **<file>** had the same -DBBA_OPTO= options, but they have different branches and the source file has different modification dates. In this case, the data associated with the oldest modification date is ignored.

```
bbarep: warning: using data from multiple version of file <file>
```



The bbadump.data file contains at least two entries for **<file>**. Both of these entries show that **<file>** had the same -DBBA_OPTO= options, both have the same number of branches, but they have different source modification dates. This warns you that some data may not be correct, but you invoked bbarep with the **-o** option (forcing it to retain the old data).

A-22 Error And Warning Messages

Installing The HP Branch Validator

Introduction

If you are going to use HP SoftBench and HP Branch Validator (BBA) together, install HP softBench before you install the HP Branch Validator (BBA). If you install BBA before installing HP SoftBench, you should reinstall BBA after installing HP SoftBench. Refer to Chapter 1 for information about software version numbers that are appropriate for your system.

The following steps describe the correct sequence of installation.

1. Install HP SoftBench (optional).
2. Verify that HP SoftBench is properly installed and working (optional).
3. Install BBA.

Installation On HPUX Systems

If HP SoftBench exists on your system and it is not installed under the /usr/softbench directory, you will need to define the SOFTBENCH shell variable to point to the directory where softbench exists.

As an example, if using sh(1) or ksh(1):

```
SOFTBENCH=/usr/softbench  
export SOFTBENCH
```

The HP Branch Validator product is shipped with SoftBench filesets that should only be installed if SoftBench does not exist on your system. These SoftBench filesets are:

B3283 SoftBench Filesets	For HP9000 series 700 Rev B.00.01
B3281 SoftBench Filesets	For HP9000 series 300 Rev B.00.01

If you have HP SoftBench installed on your system, do not install the above filesets.

You can avoid the need to install the above fileset by using `/etc/update` and unselecting the fileset from the list of filesets to be loaded.

Installation On SUN Sparc Systems With SunOS 4.X

If HP SoftBench exists on your system and it is not installed under the `/usr/softbench` directory, you will need to define the `SOFTBENCH` shell variable to point to the directory where `softbench` exists.

As an example, if using `csh(1)`:
`setenv SOFTBENCH /usr/softbench`

The HP Branch Validator product is shipped with SoftBench components that are required for operation on your system. If you already have SoftBench on your system, the components will already be there. If you do not have SoftBench on your system, the components will be installed automatically when you install the HP Branch Validator product. The installation process automatically checks to determine if the HP SoftBench components are needed.

Installation On SUN Sparc Systems With SunOS 5.X (Solaris)

The HP Branch Validator SoftBench interface (`bba`) is only available if you have HP SoftBench installed. You must purchase separately the HP SoftBench Framework to use this interface.

If HP SoftBench exists on your system and it is not installed under the `/opt/softbench` (Solaris) directory, you will need to define the `SOFTBENCH` shell variable to point to the directory where `softbench` exists.

As an example, if using `csh(1)`:
`setenv SOFTBENCH= /opt/softbench`

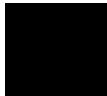
Index

- A**
 - accelerators in Main BBA Window, customizing, **3-8**
 - action used in makefiles, **8-2**
 - active files and functions display, how to use, **9-21**
 - <Alt> key, which key is it on the keyboard, **3-7**
 - application-defaults file, what is its name, **9-2**
 - array and data sections, linking, **4-39**
 - ASM pragma (bbacpp), **4-38**
 - automating regression testing, **8-14**
 - avoiding manual re-verification of known-good results, **2-21**

- B**
 - _bA_ symbols (bbacpp), **4-36**
 - BBA effect on code expansion, **4-5**
 - BBA general information, **1-1**
 - BBA Unload, **1-2**
 - BBA, benefits of using, **2-35, 2-37**
 - BBA, the pitfalls of using certain options, **4-37**
 - BBA/emulator getting started procedure, **2-1**
 - BBA_ADD_PRAG_COMMAND option in Actions menu, **9-9**
 - BBA_ALERT pragma (bbacpp), details, **4-35**
 - BBA_ALERT pragma and bbarep -aN or -bN (bbarep), **6-20**
 - BBA_ALERT pragma and bbarep -l (bbarep), **6-18**
 - BBA_ALERT warning, which branch caused it, **9-35**
 - BBA_BUILD_COMMAND option in configuration file, **9-10**
 - BBA_C_PLUS_PLUS option in configuration file, **9-4**
 - BBA_DUMPDATA_FILE option, **9-5**
 - BBA_EDIT_FILE_COMMAND option in configuration file, **9-10**
 - BBA_IGN_COMMENT_TYPE option, **9-5**
 - BBA_IGNORE (bbacpp), why use it?, **4-32**
 - BBA_IGNORE and else statements, **4-32**
 - BBA_IGNORE pragma (bbacpp), details of, **4-30**
 - BBA_IGNORE pragma, operation and use, **6-23**
 - BBA_IGNORE used in getting started procedure, **2-16**
 - BBA_IGNORE_ALWAYS_EXECUTED, details of, **4-34**
 - BBA_IGNORE_COMMENT option, **9-6**
 - BBA_IGNORE_FILE option, **9-4**
 - BBA_MAKEFILE_NAME option in Actions menu, **9-7**

- BBA_MAKEFILE_OPTIONS option in Actions menu, **9-7**
- BBA_MERGE_COMMAND option in Actions menu, **9-8**
- BBA_MOD_TEST_COMMAND option in Actions menu, **9-8**
- BBA_PRAG_COMMENT_TYPE option, **9-6**
- BBA_PRAGMA_COMMENT option, **9-6**
- BBA_PRAGMA_FILE option, **9-5**
- BBA_PRAGMA_TYPE option, **9-7**
- BBA_PRINTER_COMMAND option in configuration file, **9-4**
- BBA_RETAIN_OLD_DATA option, **9-5**
- BBA_RUN_TEST_COMMAND option in Actions menu, **9-8**
- BBA_SOURCE_TAB_WIDTH option, **9-5**
- BBA_STOP_TEST_COMMAND option in Actions menu, **9-8**
- BBA_UNLOAD_COMMAND option in Actions menu, **9-7**
- BBA_USE_SOFTBENCH option in configuration file, **9-3**
- bbacpp,
 - branches instrumented with -DBBA_OPTO=f, **4-21**
 - changing the name of the constant data section, **4-10**
 - changing the name of the data data section, **4-11**
 - DBBA_OPTc option, **4-10**
 - default instrumenting of branches, **4-14**
 - detailed discussion, **4-1**
 - error and warning messages, **A-2**
 - generally, **1-1**
 - how it works, **4-4**
 - how to invoke it, **4-8**
 - insertions, an example, **4-5**
 - instrumenting : ? statements, **4-17**
 - instrumenting a switch with no default, **4-20**
 - instrumenting all options, how to, **4-22**
 - instrumenting do-while loops, **4-18**
 - instrumenting else-less if statements, **4-20**
 - instrumenting fall-through case and default statements, **4-18**
 - instrumenting switch statements, **4-14**
 - instrumenting the third expression of a for loop, **4-21**
 - instrumenting while loops), **4-22**
 - not initializing the data area), **4-37**
 - quick reference, **4-4**
 - removing symbols from the absolute file, **4-37**
 - using BBA_ALERT pragma, **4-35**
 - what it inserts into your code, **4-4**
 - what the array is for, **4-4**

- bbacpp (continued),
 - what the assignment statement is for, **4-4**
 - what the constant data is for, **4-4**
 - what the mapfile is for, **4-4**
- bbacpp.spec file to move when networking, **1-6**
- bbacppQ.spec for use with -Q option, **4-38**
- bbamerge,
 - generally, **1-3**
 - described in detail, **7-1**
 - quick reference, **7-2**
 - what it does, **7-1**
 - when to use, **7-3**
- BBAPATH, detailed description, **6-5**
- bbarc configuration file for HP Branch Validator/SoftBench, **9-3**
- bbarep,
 - aN option, **6-19**
 - bN option, **6-17**
 - default report in getting started, **2-15**
 - Df option, **6-37**
 - Dt option, **6-36**
 - Dv option, **6-39**
 - error and warning messages, **A-13**
 - detailed discussion, **6-1**
 - F option, **6-35**
 - generally, **1-3**
 - l option, **6-9**
 - o option, **6-34**
 - p option, **6-22**
 - quick reference, **6-3**
 - s option, **6-8**
 - S option, **6-7**
 - source-reference report in getting started, **2-19**
 - what it does, **6-4**
- bbaunload,
 - cannot locate the data arrays, **4-37**
 - error and warning messages, **A-10**
 - generally, **1-2**
 - used in an emulator, **5-1**
 - what it does, **5-1**
- before starting HP Branch Validator/SoftBench, **9-11**
- body of 'while' loop was never executed, **6-15**



body of for loop was never executed (bbarep), **6-15**
body of while loop was never skipped (bbarep), **6-15**
branches in source listing, shown in HP Branch
Validator/SoftBench, **3-16**
branches,
 how to ignore, **9-22**
 how to ignore in HP Branch Validator/SoftBench, **3-18**
 how to ignore in source file listing, **9-17**
 retained defined, **3-10**
build, as it is used in HP Branch Validator/SoftBench, **9-27**

C case code was never executed (bbarep), **6-13**
case statements, how to instrument, **4-18**
changing the mapfile's suffix from .M, **4-26**
code expansion caused by bbacpp, **4-5**
code that is logically dead, how to find it, **2-27**
combining -aN and -bN (bbarep), **6-21**
combining -DBBA_OPTO= options, **4-23**
command buttons row in Main BBA Window, **3-6**
command file, example linker, **5-4**
command files run by UNIX scripts, **8-15**
commenting ignore entries, **9-23**
comments that can be used in pragmas, **9-25**
compatible software version numbers, **1-4**
conditional assignments, how to instrument, **4-17**
conditional of 'while' was never TRUE, **6-15**
conditional of conditional assignment was never FALSE (bbarep), **6-16**
conditional of conditional assignment was never TRUE (bbarep), **6-16**
conditional of do while was never FALSE (bbarep), **6-16**
conditional of do while was never TRUE (bbarep), **6-16**
conditional of for was never TRUE (bbarep), **6-15**
conditional of if was never FALSE (else never executed)
(bbarep), **6-12**
conditional of if was never FALSE (no else statement) (bbarep), **6-13**
conditional of if was never FALSE (no else), **6-13**
conditional of if was never TRUE (bbarep), **6-12**
conditional of while was never false first time thru (bbarep), **6-15**
configuration file details for HP Branch Validator/Softbench, **9-3**

Configuration file options:

- BBA_ADD_PRAG_COMMAND, **9-9**
- BBA_BUILD_COMMAND, **9-10**
- BBA_C_PLUS_PLUS, **9-4**
- BBA_DUMPDATA_FILE, **9-5**
- BBA_EDIT_FILE_COMMAND, **9-10**
- BBA_IGNORE_FILE, **9-4**
- BBA_MAKEFILE_NAME, **9-7**
- BBA_MAKEFILE_OPTIONS, **9-7**
- BBA_MERGE_COMMAND, **9-8**
- BBA_MOD_TEST_COMMAND, **9-8**
- BBA_PRAGMA_FILE, **9-5**
- BBA_PRINTER_COMMAND, **9-4**
- BBA_RUN_TEST_COMMAND, **9-8**
- BBA_SOURCE_TAB_WIDTH, **9-5**
- BBA_STOP_TEST_COMMAND, **9-8**
- BBA_UNLOAD_COMMAND, **9-7**
- BBA_USE_SOFTBENCH, **9-3**

constant data structure, override its placement, **4-10**

controlling an HP Branch Validator/SoftBench test run, **9-29**

copying demonstration files in getting started, **2-4**

- D** data and array sections, linking, **4-39**
- data area not initialized, **4-37**
- data section, changing its name, **4-11**
- dead code, how to find it, **2-27**
- dealing with the 'Out of push-back-line memory' error, **4-36**
- default code was never executed (bbarep), **6-13**
- definition of branches, **1-1**
- definition of pragma, **4-30**
- demo directory used by HP Branch Validator/SoftBench, **3-3**
- demo files and directories in getting started, **2-1**
- Demo_clean, what it does, **3-3**
- Demo_install, what it does, **3-3**
- demonstration, starting without SoftBench, **3-5**
- dependency used in makefiles, **8-2**
- description of BBA, **1-1**
- diff used in getting started, **2-21**
- directory used in getting started, **2-1**
- \$DISPLAY, **3-5, 9-12 - 9-13**
- display out of sync after pressing key, **9-35**
- displaying source file listings, **9-16**

do-while loops, how to instrument, **4-18**
documenting the test environment, **6-36**
driver routine in getting started, **2-5**
dumpdata file, how to set it and retain data, **9-32**

- E**
- editing a file, how to access the file, **9-30**
 - else part of if was never executed (bbarep), **6-12**
 - emulator 68331 or 68332, special requirements, **2-10**
 - emulator use of bbaunload, **5-1**
 - emulator/BBA getting started procedure, **2-1**
 - encaprun: cannot find a message server message, **3-5, 9-12 - 9-13**
 - equipment software version numbers, **1-5**
 - error messages, **A-1**
 - error messages displayed in HP Branch Validator/SoftBench, **9-22**
 - example of bbacpp insertions, **4-5**
 - example of the -eN option (bbarep), **6-17**
 - exiting HP Branch Validator/SoftBench, **9-33**
 - exiting the HP Branch Validator/SoftBench interface, **3-21**
 - expansion of code space caused by bbacpp, **4-5**
- F**
- false part of conditional assignment was never executed (bbarep), **6-16**
 - file format for pragma files, **9-26**
 - file format, ignore file, **9-24**
 - File History report, discussed in getting started, **3-14**
 - File History test report, discussed in detail, **9-15**
 - file name with ":" causing failure, **9-34**
 - file, example linker command, **5-4**
 - file, how to access and edit, **9-30**
 - file, how to set and retain dump data, **9-32**
 - files and directories used in getting started, **2-1**
 - files of HP Branch Validator/SoftBench improper, **9-36**
 - files, how to ignore, **9-22**
 - files, merging to save space, **7-1**
 - final report in getting started procedure, **2-35**
 - footnotes, how to suppress them in reports, **6-35**
 - for statement, instrumenting the third expression, **4-21**
 - function was never called (bbarep), **6-12**
 - functions performed by bbacpp, **4-4**
 - functions, how to ignore, **9-22**

- G** getting started,
 - files and directories, **2-1**
 - problem BBA/emulator, **2-1**
 - test program, **2-4**
 - the final report, **2-35**
 - using ignore files, **2-25**
 - with BBA and a 68030 emulator, **2-1**
 - with BBA/emulator, **2-1**
 - with HP Branch Validator in SoftBench, **3-1**
- good results, how to avoid reverifying, **2-21**
- H** Help
 - how to use it in HP Branch Validator/SoftBench, **9-33**
 - messages not available, **9-11**
- Histogram
 - discussed in getting started, **3-9**
 - test report, discussed in detail, **9-15**
- HP 64000 format files, **4-37**
- HP Branch Validator
 - appears to hang when you try to exit, **9-33**
 - files with improper format, **9-36**
 - in SoftBench, getting started procedure, **3-1**
 - starting but not using Message Server, **9-13**
- HP Branch Validator/SoftBench
 - before starting the demo, **3-3**
 - configuration file (.bbarc), **9-3**
 - controlling a run of a test, **9-29**
 - how to exit or quit, **9-33**
 - how to invoke from command line, **9-12**
 - interface, how to quit, **3-21**
 - restrictions when using, **9-34**
 - starting the demo procedure, **3-4**
 - things to do before starting, **9-11**
 - use of build, **9-27**
- I** if statements with no else, how to instrument, **4-20**
- Ign-> symbol discussed, **9-21**
- ignore,
 - file, function, or branch, how to, **9-22**
 - capabilities and options, **4-31**
 - entries, how to add comments, **9-23**
 - file contents, format of, **9-24**



- ignore (continued),
 - file details, **9-23**
 - file format, **9-24**
 - files used in getting started, **2-25**
 - files in detail, **6-23**
- ignoring,
 - all functions in a file (bbarep), **6-24**
 - branch in HP Branch Validator/SoftBench, **3-18**
 - branch in source file listing, **9-17**
 - 'case' and 'default' statements, **6-28**
 - 'conditional assignment' statement, **6-30**
 - data in BBA reports, **2-18**
 - elements during a test, **6-23**
 - 'for' loop, **6-30**
 - 'if' statement, **6-28**
 - inserted default, **6-29**
 - macro (bbarep), **6-25**
 - single function (bbarep), **6-24**
 - specific statements (bbacpp), **4-30**
 - specific statements (bbarep), **6-26**
 - TRUE or FALSE branches, **6-30**
 - 'while' statement, **6-29**
- illegal combination of pointer and integer, **4-17**
- initial test set in getting started, **2-10**
- insertion of pragmas incorrect in source file, **9-19**
- instrumented branches, definition, **4-12**
- invoking HP Branch Validator/SoftBench from the command line, **9-12**
- K**
 - keyboard accelerator makes display unstable, **9-35**
 - keyboard accelerators, file to edit, **9-2**
 - keyboard letters to select test reports, **9-14**
 - known-good results, how to avoid reverifying, **2-21**
- L**
 - letters on keyboard to select test reports, **9-14**
 - Lex source files with HP Branch Validator and SoftBench, **9-35**
 - line numbers shown in report (bbarep), **6-9**
 - linking array and data sections, **4-39**
 - logically dead code, how to find it, **2-27**

M macros used in makefiles, **8-2**
Main BBA Window,
discussed in getting started, **3-6**
pops to top of screen, **9-35**
makefile,
changes for BBA/emulator, **2-6**
described in getting started, **2-6**
discussed in detail, **8-1**
how to create and use, **8-2**
with BBA in getting started, **2-8**
with simple BBA capabilities, **8-6**
with/without BBA, automatic, **8-7, 8-10**
with/without BBA, automatic and efficient, **8-12**
without BBA capability, **8-3**
without BBA in getting started, **2-7**
map file,
how to change its filename extension, **4-26**
how to prevent its creation, **4-28**
its purpose in bbacpp, **4-4**
incomplete? this may be why, **4-25**
".M" when using HP Branch Validator/SoftBench, **9-34**
the information it gives to bbarep, **4-29**
memory,
how to increase push-back-line memory, **4-36**
if not enough, here is a workaround, **4-23**
menu item accelerators in Main BBA Window, **3-8**
menu mnemonics in Main BBA Window, **3-6**
merging dumpdata files, **7-1**
Message Server,
how to turn it on, **9-12**
not used with HP Branch Validator/SoftBench, **9-13**
options when not using it, **9-10**
whether or not to use it, **9-11**
message strings explained, **6-12**
messages
displayed in HP Branch Validator/SoftBench, **9-22**
error and warning, **A-1**
mnemonics in Main BBA Window, customizing, **3-8**
\$MSERVE, **3-5, 9-12 - 9-13**
multiple dumps in a single bbadump.data file, **6-4**

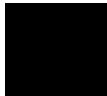
- N**
 - networked system described in getting started, **2-2**
 - networking, **1-6**
 - networking requirements, **2-2**
- O**
 - old data included in reports (bbarep), **6-34**
 - option h for generating HP 64000 format files, **4-37**
 - options when not using SoftBench Broadcast Message Server, **9-10**
 - overview of bbacpp, **1-1**
 - overview of makefiles, **8-1**
- P**
 - P^, what it means in source file listing, **9-18**
 - PATH variable,
 - made specifically for BBA, **6-5**
 - required additional file, **3-2**
 - PI, what it means in the source file listing, **9-18**
 - pragma,
 - definition, **4-30**
 - file format, **9-26**
 - how to add one in source windows, **9-25**
 - how to insert in HP Branch Validator/SoftBench, **3-19**
 - in wrong place when using HP Branch Validator/SoftBench, **9-34**
 - not correctly inserted in source file, **9-19**
 - only one gets added to source file, **9-34**
 - types of comments they can use, **9-25**
 - preprocessor used by BBA, detailed discussion, **4-1**
 - preventing creation of the mapfile, **4-28**
 - print,
 - how to use in HP Branch Validator/SoftBench, **9-31**
 - HP Branch Validator/SoftBench test results, **3-20**
 - version of bbacpp, **6-39**
 - program used in getting started procedure, **2-4**
 - pull-down menu bar in Main BBA Window, **3-6**
 - purpose of BBA, **1-1**
 - push-back-line memory, how to increase it, **4-36**
- Q**
 - quitting HP Branch Validator/SoftBench, **9-33**
 - quitting the HP Branch Validator/SoftBench interface, **3-21**

- R** regression testing,
 - advantages, **8-20**
 - automatic, **8-14**
 - testing example, **8-16**
- report,
 - default form, **6-8**
 - default form in getting started, **2-15**
 - from other than 'dumpfile.data', **6-32**
 - includes old data, **6-34**
 - function-by-function summary, **6-8**
 - generator, detailed discussion, **6-1**
 - listing each file included, **6-37**
 - only specific files (bbarep), **6-4, 6-31**
 - separate reports combined, **6-32**
 - short summary, **6-7**
 - showing ***, **6-10**
 - showing line numbers, **6-9**
 - showing N lines after the unexecuted statement, **6-19**
 - showing N lines before the unexecuted statement, **6-17**
 - showing totals printed to stdout, **6-36**
 - showing lines before and after statement, **6-21**
 - source-reference, **2-19**
- reserved words for BBA, **4-36**
- restrictions when using HP Branch Validator/SoftBench, **9-34**
- Results Only,
 - display, discussed in getting started, **3-12**
 - test report, discussed in detail, **9-15**
- results, known-good, how to avoid reverifying, **2-21**
- retained branches, definition of, **3-10**
- reverifying known good results, avoiding, **2-21**
- running your HP Branch Validator/SoftBench test, **9-29**
- S** save, how to use in HP Branch Validator/SoftBench, **9-31**
- saving HP Branch Validator/SoftBench test results in a file, **3-20**
- set -m;, **9-10**
- set of files and functions in report, how to compose, **9-21**
- setting tabs for source-reference reports (bbarep), **6-33**
- setting tabs=spaces in source file listings, **9-17**
- short summary
 - (bbamerge), **7-3**
 - report (bbarep), **6-7**



- SoftBench Broadcast Message Server,
 - how to turn it on, **9-12**
 - whether or not to use it, **9-11**
- SoftBench with HP Branch Validator, getting started procedure, **3-1**
- SoftBench, command to use if not installing, **3-3**
- software,
 - compatibility, **1-5**
 - version numbers, **1-4**
- source file,
 - displays, how to obtain, **3-16**
 - gets only one pragma added to it, **9-34**
 - listings, how to display, **9-16**
- source-reference report, **2-19**
- special requirements: 68331 or 68332 emulator, **2-10**
- specifying the dump file's name, **6-32**
- starting HP Branch Validator/SoftBench from Tool Manager, **3-4, 9-12**
- Summary test report,
 - discussed in detail, **9-14**
 - discussed in getting started, **3-10**
- switch never went to case (bbarep), **6-13**
- switch never went to case (code executed by fall-thru) (bbarep), **6-14**
- switch never went to case (no executable statements) (bbarep), **6-14**
- switch never went to default (bbarep), **6-13**
- switch never went to default (code executed by fall-thru) (bbarep), **6-14**
- switch never went to default (no executable statements) (bbarep), **6-14**
- switch never went to inner case (bbarep), **6-13**
- switch never went to inner case (code executed by fall-thru) (bbarep), **6-14**
- switch never went to inner case (no executable statements) (bbarep), **6-14**
- switch never went to inner default (bbarep), **6-13**
- switch never went to inner default (code executed by fall-thru) (bbarep), **6-14**
- switch never went to inner default (no executable statements), **6-14**
- switch statements, how to instrument, **4-18**
- switch went to 'default' (default not defined), **6-14**
- switch with no default, how to instrument, **4-20**
- symbols stripped out of the absolute file, **4-37**
- symbols used by BBA, **4-36**

- T**
 - 3rd expression of for was never executed (bbarep), **6-15**
 - tab spaces, defining for source-reference reports, **6-33**
 - tabs=spaces, how to set in source file listings, **9-17**
 - target used in makefiles, **8-2**
 - test,
 - advantages of regression testing, **8-20**
 - automatic regression testing, **8-14**
 - avoid reverifying known-good results, **2-21**
 - environment, how to document, **6-36**
 - example automatic regression test, **8-16**
 - report area in Main BBA Window, **3-6**
 - reports selected by typing keyboard letters, **9-14**
 - with BBA efficiently, **8-1**
 - then part of if was never executed (bbarep), **6-12**
 - Tool Manager used to start HP Branch Validator/SoftBench, **3-4, 9-12**
 - true part of conditional assignment was never executed (bbarep), **6-16**
 - types of comments used with pragmas, **9-25**
- U**
 - unexecuted branches, how to display in source file list, **3-16**
 - Unload_BBA, **1-2**
 - cannot locate the data arrays, **4-37**
 - error and warning messages, **A-10**
 - unloading dump data from an emulator, **5-1**
 - usefile explained in detail, **6-31**
 - using BBA/emulator in getting started, **2-1**
 - using BBA_IGNORE to ignore functions, **4-30**
 - using bbacpp from the command line, **4-8**
- V**
 - version number,
 - for associated software, **1-4**
 - of bbarep, how to print, **6-39**
 - of compatible software, **1-5**
- W**
 - warning messages, **A-1**
 - displayed in HP Branch Validator/SoftBench, **9-22**
 - what to do when out of memory, **4-23**
 - while loop never skipped, how to ignore, **4-34**
 - while loops, how to instrument, **4-22**
- X**
 - Xdefaults files, where they are located, **9-2**
- Y**
 - Yacc source files with HP Branch Validator and SoftBench, **9-35**





Notes

