

**IBM**

**RISCWatch Debugger  
User's Manual**

13H6964 000011

Fourteenth edition (June 2002)

This edition of *IBM RISCWatch Debugger User's Manual* applies to IBM RISCWatch Debugger Version 5.0 and to all subsequent versions of the debugger until otherwise indicated in new versions or technical newsletters.

**The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS MANUAL "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

IBM does not warrant that the contents of this publication or the accompanying source code examples, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying source code examples are error-free.

This publication could contain technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or program(s) described in this publication at any time.

It is possible that this publication may contain references to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Any reference to an IBM licensed program in this publication is not intended to state or imply that you can use only IBM's licensed program. You can use any functionally equivalent program instead.

No part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the written permission of IBM.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

Forms for user's and reader's comments are provided on page xvii and page xix, respectively. You may also address written comments about this publication to:

IBM Corporation  
Department YM5A  
P.O. Box 12195  
Research Triangle Park, NC 27709

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

©Copyright International Business Machines Corporation 1997, 2002. All rights reserved.

Printed in the United States of America.

4 3 2 1

Notice to U.S. Government Users—Documentation Related to Restricted Rights—Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

## Patents and Trademarks

IBM may have patents or pending patent applications covering the subject matter in this publication. The furnishing of this publication does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 208 Harbor Drive, Stamford, CT 06904, United States of America.

The following terms are trademarks of IBM Corporation:

- AIX
- AIX/Windows
- IBM
- OS Open
- PowerPC
- PowerPC Architecture
- RISC System/6000
- RISCTrace
- RISCWatch

The following term is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited:

- UNIX

Windows is a trademark of Microsoft Corporation.

Other terms which are trademarks are the property of their respective owners.



---

# Contents

<b>Contents</b> .....	<b>v</b>
<b>Figures</b> .....	<b>xiii</b>
<b>Tables</b> .....	<b>xv</b>
<b>User's Comments Form</b> .....	<b>xvii</b>
<b>Reader's Comments Form</b> .....	<b>xix</b>
<b>About This Book</b> .....	<b>xxi</b>
Who Should Use This Book .....	xxi
How To Use This Book .....	xxi
Conventions Used In This Book .....	xxii
Numeric Notation and Input Conventions.....	xxii
Highlighting Conventions.....	xxii
Syntax Diagram Conventions.....	xxiii
Where to Find More Information .....	xxiv
Related IBM Publications .....	xxiv
<b>Introducing the RISCWatch Debugger</b> .....	<b>1-1</b>
Embedded System Software Development .....	1-1
Programming Languages .....	1-1
Features.....	1-1
<b>Quick Start</b> .....	<b>2-1</b>
Starting the Debugger.....	2-1
Entering Commands .....	2-2
Loading the Demo Program.....	2-2
Scrolling Through Source Code.....	2-3
Setting Breakpoints.....	2-5
Stepping Through the Code.....	2-7
Altering and Displaying Variables .....	2-9
Debugging at the Assembly Level .....	2-13
<b>Using the RISCWatch Debugger</b> .....	<b>3-1</b>
Debugger Facilities .....	3-1
Environment Resources.....	3-5
Core + ASIC Resources .....	3-8
Processors, Cores and Chip Resources .....	3-9
Processor Configuration File (PCF) .....	3-9

File Management .....	3-10
File Syntax .....	3-10
REFER Definitions.....	3-11
MACRO Definitions .....	3-11
CHIP Definitions .....	3-12
INCLUDE Definitions.....	3-13
EXEC Definitions.....	3-14
FIELD Definitions .....	3-15
NAME Definitions .....	3-16
PVR Definitions .....	3-16
REG Definitions.....	3-17
REGALIAS Definitions .....	3-18
REGFLD Definitions.....	3-19
REV Definitions .....	3-19
PCF Compiling .....	3-20
PCF Example .....	3-20
MEMACC Command.....	3-23
Use of MEMACC ADD .....	3-24
Practical Application Example.....	3-25
Window Descriptor File .....	3-26
RISCWatch Programming Interface (RWPI).....	3-26
Multi-Processor Resources.....	3-28
PCF File Syntax For Multi-Core Processors .....	3-28
MPS File Syntax.....	3-29
Board Definitions .....	3-29
MPS Debugging .....	3-31
MPS Context.....	3-34
MPS Windows.....	3-35
Invoking the Debugger.....	3-35
JTAG Ethernet Targets and the RISCWatch Processor Probe.....	3-38
Main Window Resources .....	3-40
Menus .....	3-41
File Menu.....	3-43
Source Menu .....	3-43
Hardware Menu .....	3-43
Window Menu.....	3-43
Utilities Menu .....	3-43

Help Menu .....	3-44
Command Line Usage.....	3-44
Command History Usage .....	3-44
Message Window .....	3-45
Running Your Programs .....	3-45
Preparing the Program for Debug .....	3-45
Loading Files .....	3-45
Loading Boot and Boot Image Files .....	3-47
Executing the Program .....	3-48
Following Program Execution Flow .....	3-48
Input Line Usage .....	3-49
Source Level Debugging.....	3-52
Source Window .....	3-52
Scrolling Source Window Contents Using the Keyboard .....	3-55
Assembly Debug Window .....	3-55
Programs Window .....	3-59
Callers Window .....	3-60
Files Window .....	3-62
Functions Window .....	3-62
Load Memory Window.....	3-64
OS Open Debugging.....	3-67
Managing Breakpoints .....	3-71
Using Software Breakpoints .....	3-72
Using Hardware Breakpoints.....	3-73
Breakpoints Window.....	3-74
Breakpoint Select Window .....	3-76
Reading and Writing Program Variables.....	3-77
Local Variables Window .....	3-78
Global Variables Window .....	3-79
Inspect Variable Windows .....	3-81
Variable Configuration Window .....	3-83
Change Variable Window.....	3-85
Formatting Examples .....	3-88
Expansion/Contraction from Locals or Globals Window .....	3-88
Displaying ASCII Strings.....	3-90
Handling Multiple Data Elements Referenced by a Single Pointer.....	3-91
Changing Multiple Instances of a Variable Within an Array .....	3-93
Type Casting a Variable .....	3-101

Source Variable Command Support .....	3-102
Reading and Writing Memory .....	3-103
Memory Coherency Window (JTAG Targets Only) .....	3-104
ASCII Memory Window .....	3-107
Custom Memory Window .....	3-109
Cache Windows (JTAG Targets Only) .....	3-112
Save Memory Window .....	3-113
Reading and Writing Registers .....	3-115
Register Windows .....	3-115
Register Field Windows .....	3-117
User-Defined Windows .....	3-118
File Syntax .....	3-118
Keyword Definition/Syntax .....	3-119
Creating the Window .....	3-121
Example .....	3-121
Command Files .....	3-123
Using Shell Scripts to Execute Command Files .....	3-124
Startup Command File .....	3-124
Special Command File Commands .....	3-124
Blank Lines and Comments in Command Files .....	3-125
Command File Programming .....	3-125
Command File Special Expressions .....	3-127
Command File Parameters .....	3-128
Command File Pseudo-Variables .....	3-129
Command File Programming Example .....	3-130
Running a Command File .....	3-131
Command File Window .....	3-132
Processor Resources .....	3-134
Processor Reset Window (JTAG Targets Only) .....	3-134
General Resources .....	3-135
Window Layout .....	3-135
Output Window .....	3-136
Window List .....	3-138
User Created Variables Window .....	3-138
Log Files .....	3-139
Logging Control .....	3-140
Logging User Comments .....	3-140
Screen Capture .....	3-141



Calculator Window .....	3-142
Online Help .....	3-143
<b>Using Processor-Specific Debug Features.....</b>	<b>4-1</b>
PowerPC 400Series MMU Implementation Notes .....	4-1
Managing Hardware Breakpoints and Trace Events .....	4-2
Using RISCTrace (400Series JTAG Processor Probe Only) .....	4-2
RISCTrace Operational Notes .....	4-3
RISCTrace Output .....	4-3
Trigger/Trace Window (400Series Only) .....	4-7
RISCTrace Controls .....	4-10
Compound Trigger/Trace Window (403Series Only) .....	4-11
Memory Resources.....	4-14
Translation Lookaside Buffer Window (Applicable Processors Only) .....	4-14
Processor Resources.....	4-15
<b>Debugger Command Reference.....</b>	<b>5-1</b>
Processors Currently Supported.....	5-1
Reading the Syntax Diagrams .....	5-1
Using RISCWatch Debugger Commands.....	5-1
Window Quick Reference .....	5-2
Command Quick Reference .....	5-3
asmstep.....	5-11
assign .....	5-12
assm.....	5-14
attach.....	5-16
beep .....	5-17
bot .....	5-18
bp .....	5-19
bpmode .....	5-23
callstep .....	5-25
capture .....	5-26
cfss.....	5-28
color.....	5-29
config.....	5-31
create .....	5-32
delay.....	5-34
detach.....	5-35
dis.....	5-36

down.....	5-38
end .....	5-40
exec.....	5-41
expr .....	5-42
fctrl .....	5-43
file.....	5-45
find .....	5-46
findb .....	5-48
finde .....	5-50
focus.....	5-52
fold .....	5-53
fprdisp .....	5-54
fprint .....	5-55
freeze .....	5-58
funcdisp.....	5-59
goto .....	5-61
halt .....	5-62
hidewins .....	5-63
ip .....	5-64
jtag .....	5-65
kill_thread.....	5-66
line.....	5-67
linestep.....	5-68
load .....	5-69
log .....	5-72
logging.....	5-73
logoff .....	5-74
memacc.....	5-75
memchk.....	5-78
memcoh .....	5-79
memcpy.....	5-81
memfill.....	5-82
memfind .....	5-83
memrwait.....	5-85
memwwait .....	5-86
mpsset.....	5-87
pagedn .....	5-88
pageup .....	5-89

parms .....	5-90
poll.....	5-91
post.....	5-92
prefer .....	5-93
print .....	5-97
quit.....	5-98
read .....	5-99
reg .....	5-102
reset .....	5-103
restart .....	5-104
retstep .....	5-105
run .....	5-106
save.....	5-108
set.....	5-110
shell .....	5-114
showip .....	5-115
socket.....	5-116
srcdisp.....	5-117
srchpath.....	5-118
srcline .....	5-120
start_thread .....	5-121
stop.....	5-122
stuff.....	5-123
timer .....	5-124
top .....	5-125
trace .....	5-126
unassign.....	5-128
uncreate .....	5-129
unload.....	5-130
up .....	5-131
varinfo.....	5-133
varvis .....	5-135
view .....	5-137
window .....	5-138
write.....	5-140
<b>Interfacing RISCWatch to a Target Board.....</b>	<b>A-1</b>
IEEE 1149.1 (JTAG) Port.....	A-1
Trace Status Port (400Series JTAG Processor Probe Only) .....	A-4

JTAG and Trace Connector Requirements .....	A-6
Target Monitor Debugging .....	A-9
<b>Register Definition File (Outdated) .....</b>	<b>B-1</b>
Register Definition File .....	B-1
File Syntax .....	B-1
DCR Register Definitions .....	B-2
SPR Register Definitions.....	B-3
MMIO Register Definitions .....	B-4
ALIAS Definitions .....	B-4
Register Field Definitions .....	B-5
<b>Index .....</b>	<b>X-1</b>

---

# Figures

Figure 2-1. Sample Main Window .....	2-1
Figure 2-2. Sample Files Window .....	2-3
Figure 2-3. Sample Source Window .....	2-4
Figure 2-4. Sample Breakpoints Window.....	2-5
Figure 2-5. Sample Functions Window .....	2-6
Figure 2-6. Sample Callers Window .....	2-7
Figure 2-7. Sample Locals Window .....	2-9
Figure 2-8. Sample Variable Configuration Window .....	2-10
Figure 2-9. Change Display Information .....	2-11
Figure 2-10. Change Base Variable.....	2-12
Figure 2-11. Sample Assembly Debug Window .....	2-14
Figure 3-1. Sample MPS Window .....	3-31
Figure 3-2. Sample MPS Window .....	3-32
Figure 3-3. Sample Main Window .....	3-40
Figure 3-4. Main Window Menu Options.....	3-42
Figure 3-5. Sample Input Line Displayed.....	3-50
Figure 3-6. Sample Source Window .....	3-52
Figure 3-7. Sample Assembly Debug Window .....	3-56
Figure 3-8. Sample Programs Window .....	3-59
Figure 3-9. Sample Callers Window .....	3-61
Figure 3-10. Sample Files Window .....	3-62
Figure 3-11. Sample Functions Window .....	3-63
Figure 3-12. Load Memory.....	3-65
Figure 3-13. Sample OS Open Window.....	3-67
Figure 3-14. Sample Breakpoints Window.....	3-74
Figure 3-15. Sample Breakpoint Select Window .....	3-76
Figure 3-16. Sample Locals Window .....	3-78
Figure 3-17. Sample Globals Window .....	3-80
Figure 3-18. Sample Inspect Window .....	3-82
Figure 3-19. Sample Variable Configuration Window .....	3-83
Figure 3-20. Change Variable Window .....	3-85
Figure 3-21. Sample Unexpanded Structure Variable .....	3-88
Figure 3-22. Sample Expanded Structure Variable .....	3-88
Figure 3-23. Further Structure Variable Expansion .....	3-88
Figure 3-24. Single-Element Structure Variable Expansion.....	3-89
Figure 3-25. Structure Variable Contraction .....	3-89
Figure 3-26. Sample Pointer Variable.....	3-90
Figure 3-27. Sample ASCII String Display.....	3-90
Figure 3-28. Sample Character Array .....	3-90
Figure 3-29. Sample Array Element Display.....	3-90
Figure 3-30. Sample <b>struct record</b> Pointer Display .....	3-91

Figure 3-31. Sample Initial <b>struct record</b> Pointer Expansion .....	3-91
Figure 3-32. Changing Pointer Variables.....	3-92
Figure 3-33. Sample Pointer Variable Shown as an Array .....	3-92
Figure 3-34. Sample Expanded Pointer Variable Shown as an Array .....	3-93
Figure 3-35. Sample <b>char</b> Array Display .....	3-94
Figure 3-36. Changing Multiple Elements of a Variable Array.....	3-95
Figure 3-37. Updated Display of Variable Array .....	3-96
Figure 3-38. Sample Multi-Element, Multilevel Variable Display .....	3-97
Figure 3-39. Updated Multi-Element, Multilevel Variable Display.....	3-98
Figure 3-40. Sample Change Value Display.....	3-99
Figure 3-41. Sample Result of Change Value Update.....	3-100
Figure 3-42. Sample Variable Type Cast.....	3-101
Figure 3-43. Sample Memory Access Window.....	3-104
Figure 3-44. Sample ASCII Memory Window .....	3-107
Figure 3-45. Custom Memory Window .....	3-109
Figure 3-46. Sample Data Cache Window .....	3-112
Figure 3-47. Save Memory .....	3-114
Figure 3-48. Sample Register Window .....	3-116
Figure 3-49. Sample Register Field Window .....	3-117
Figure 3-50. Sample User-Defined Window .....	3-123
Figure 3-51. Sample Command File Window .....	3-132
Figure 3-52. Sample Processor Reset Window.....	3-135
Figure 3-53. Sample Output Window.....	3-136
Figure 3-54. Sample User Created Variables Window .....	3-138
Figure 3-55. Sample Log Comment Window.....	3-141
Figure 3-56. Sample Calculator Window .....	3-142
Figure 4-1. Sample Trace Output File .....	4-5
Figure 4-2. Sample Trigger/Trace Window with Trace Supported.....	4-8
Figure 4-3. Sample Compound Trigger/Trace Window with Trace Supported .....	4-12
Figure 4-4. Sample TLB Window.....	4-14
Figure A-1. JTAG Header Connector (top view).....	A-1
Figure A-2. RISCTrace Header (top view).....	A-4
Figure A-3. RISCTrace 2 x10 and RISCWatch 2 x 8 Headers .....	A-6

---

## Tables

Table 3-1. Quick Reference for the RISCWatch Debugger .....	3-2
Table 3-2. Input Line Functions .....	3-49
Table 3-3. Keyboard Options for Scrolling .....	3-55
Table 4-1. Quick Reference for Processor-Specific Debug Features .....	4-1
Table 5-1. Syntax Summary for Debugger Commands .....	5-5
Table 5-2. Windows that support capture and total .....	5-27
Table A-1. PowerPC 400Series JTAG Interface Connections and Resistors .....	A-2
Table A-2. PowerPC 6xx/7xx JTAG Interface Connections and Resistors .....	A-3
Table A-3. RISCTrace Header Pin Description .....	A-5
Table A-4. Mictor Connector Signal Assignments .....	A-6
Table A-5. Mictor Connector Signal Assignments for 440 .....	A-7





# User's Comments Form

We hope you are delighted with this product, but only you can tell us! Your comments and suggestions will help us improve our products. Please take a few minutes to let us know what you think by completing this form.

If you wish to fax this form, please send to the following number care of 'PowerPC Embedded Tools Software Feedback':

FAX: (919) 543-7575

If you wish to send your comments softcopy, please send to the following Internet address:

INTERNET: ppcsupp@us.ibm.com

You can also contact us at our web page:

INTERNET: <http://www-3.chips.ibm.com/chips/products/powerpc/tools/riscwatc.html>

Please indicate which product you are commenting on by marking the appropriate box:

- OS Open Real-Time Operating System
- IBM PowerPC Evaluation Board Kit
- High C/C++ Compiler
- RISCWatch Debugger

In order for us to properly process your information, please also include the version number for the product you indicated above. **Version:** \_\_\_\_\_

Please check the appropriate boxes below, to describe your host, target and application:

<b>Host Platform</b>	RS/6000	<input type="checkbox"/>	PC (Win98/NT/2000/ME/XP)	<input type="checkbox"/>	Sun (Solaris)	<input type="checkbox"/>
	Red Hat Linux	<input type="checkbox"/>		<input type="checkbox"/>		<input type="checkbox"/>
<b>Target Processor</b>	403	<input type="checkbox"/>	405	<input type="checkbox"/>	440	<input type="checkbox"/>
	603e/ev	<input type="checkbox"/>	604e	<input type="checkbox"/>	740/750	<input type="checkbox"/>
	750cx	<input type="checkbox"/>	Other: _____			
<b>Target Platform</b>	IBM Evaluation Board	<input type="checkbox"/>	Other Evaluation Board (please specify): _____			
	Other Platform: _____					
<b>Target Application</b>	IBM ROM Monitor	<input type="checkbox"/>	OS Open	<input type="checkbox"/>	Own ROM Monitor	<input type="checkbox"/>
	Other: _____					

<b>Interface Used</b>	JTAG				
	Ethernet				

1. Please rate the characteristics of the product on a scale of 1 to 5 (1 being the best):

ease of installation	1	2	3	4	5
ease of use	1	2	3	4	5
amount of function provided	1	2	3	4	5
level to which it helped you do your job	1	2	3	4	5
reliability (frequency of failure)	1	2	3	4	5
performance	1	2	3	4	5
error messages	1	2	3	4	5
IBM problem support and service	1	2	3	4	5
price, considering value received	1	2	3	4	5

2. What is your overall impression of the product?

overall	1	2	3	4	5
---------	---	---	---	---	---

Please include additional comments below. PLEASE BE AS SPECIFIC AS POSSIBLE.

---



---



---

Please tell us how we can improve this product:

---



---



---

Please tell us what you especially liked about the product:

---



---

Thank you for your response. When you send information to IBM, you grant IBM the right to use or distribute the information without incurring any obligation to you. You of course retain the right to use the information in any way you choose.

Please provide the following information should it be necessary for us to contact you for any reason in order to properly address your input:

Name: \_\_\_\_\_

Company: \_\_\_\_\_

Phone: \_\_\_\_\_ Internet Address: \_\_\_\_\_

---

# Reader's Comments Form

We hope you find this publication useful, readable and technically accurate, but only you can tell us! Your comments and suggestions will help us improve our technical publications. Please take a few minutes to let us know what you think by completing this form.

If you wish to fax this form, please send to the following number care of 'PowerPC Embedded Tools Software Feedback':

FAX: (919) 543-7575

If you wish to send your comments softcopy, please send to the following Internet address:

INTERNET: [ppcsupp@us.ibm.com](mailto:ppcsupp@us.ibm.com)

You can also contact us at our web page:

INTERNET: <http://www-3.chips.ibm.com/chips/products/powerpc/tools/riscwatc.html>

Please indicate which publication you are commenting on by marking the appropriate box:

<input type="checkbox"/>	High C/C++ Language Reference
<input type="checkbox"/>	High C/C++ Compiler, ELF Linker and Assembler
<input type="checkbox"/>	OS Open User's Guide
<input type="checkbox"/>	OS Open Programmer's Reference Volume 1
<input type="checkbox"/>	OS Open Programmer's Reference Volume 2
<input type="checkbox"/>	PowerPC Evaluation Board Kit User's Guide
<input type="checkbox"/>	RISCWatch Debugger User's Guide

In order for us to properly process your information, please also include the edition number and date for the book you indicated above (on the back of the title page, at the top).

**Edition and Date:** \_\_\_\_\_

1. Please rate the characteristics of the book on a scale of 1 to 5 (1 being the best).

accurate	1	2	3	4	5
complete	1	2	3	4	5
well laid out	1	2	3	4	5
well organized	1	2	3	4	5
easy to understand	1	2	3	4	5

applies to your tasks    1       2       3       4       5

has enough examples    1       2       3       4       5

2. What is your overall impression of the book?

overall                    1       2       3       4       5

For additional comments, either attach a marked-up hardcopy (if applicable) or include your comments below. PLEASE BE AS SPECIFIC AS POSSIBLE AND INCLUDE THE PAGE NUMBER AND SECTION OF THE PUBLICATION WHERE YOU HAVE A COMMENT.

Specific Comments or Problems:

---

---

---

---

---

---

Please tell us how we can improve this book:

---

---

---

---

---

---

Please tell us what you especially liked about the book:

---

---

---

---

Thank you for your response. When you send information to IBM, you grant IBM the right to use or distribute the information without incurring any obligation to you. You of course retain the right to use the information in any way you choose.

Please provide the following information should it be necessary for us to contact you for any reason in order to properly address your input:

Name: \_\_\_\_\_

Company: \_\_\_\_\_

Phone: \_\_\_\_\_ Internet Address: \_\_\_\_\_

---

## About This Book

This book describes the IBM® RISCWatch™ Debugger, its windowing environment, and its debugging facilities and commands. This publication contains the information needed to use RISCWatch, a hardware and software development tool for PowerPC™ processors.

The RISCWatch Debugger supports numerous PowerPC processors and versions. For more information on current processors supported and other up to date information, please refer to the README file included with the product, or visit our web site at <http://www-e.ibm.com/chips/products/powerpc/tools/riscwatc.html>

Support for additional PowerPC processors and targets is planned for future RISCWatch releases.

---

## Who Should Use This Book

This book is for:

- Programmers and engineers who will use the RISCWatch Debugger to develop embedded applications using PowerPC processors

Users should understand:

- Functions, architecture, and features of their host systems
- PowerPC instruction set architecture and assembler programming
- C programming

For information concerning features and operations of a specific PowerPC processor, please refer to the document set for each individual device.

---

## How To Use This Book

This manual describes the RISCWatch debugger facilities, windows, and functions provided specifically to support PowerPC processors in embedded applications. This book is divided into the following chapters:

- Chapter 1, "Introducing the RISCWatch Debugger," describes RISCWatch debugger functions and features.
- Chapter 2, "Quick Start," introduces the RISCWatch Debugger by means of a brief demo with descriptions of the main windows and debugger functions.
- Chapter 3, "Using the RISCWatch Debugger," shows debugging tasks in relation to sample debugger windows and some specific features of the debugger.

- Chapter 4, "Using Processor-Specific Debug Features," describes RISCWatch features and windows applicable to specific PowerPC processors.
- Chapter 5, "Debugger Command Reference," provides detailed descriptions of the debugger commands.
- Appendix A, "Interfacing RISCWatch to a Target Board," describes the required connections for interfacing RISCWatch to a PowerPC processor on a target development board.
- Appendix B, "Register Definition File (Outdate)," describes the file format for the Register Definition File (RDF) which is used to add custom register definitions to the RISCWatch debugger.

For detailed information about installing and configuring the RISCWatch Debugger, consult the accompanying *RISCWatch Debugger Installation Guide*.

## Conventions Used In This Book

This book follows numeric and highlighting notation conventions based on those used in the RISC System/6000™ and Advanced Interactive Executive (AIX™) publications.

### Numeric Notation and Input Conventions

In general, numbers are used exactly as shown. Unless noted otherwise, all numbers are in decimal, and, if entered as part of a command, are entered without format information.

The hexadecimal digits A through F typically appear in uppercase. Hexadecimal numbers are preceded by "0x" as shown below:

0x1A7

### Highlighting Conventions

In code examples, this book uses no highlighting.





This book uses the following highlighting conventions:

- The names of invariant objects known to RISCWatch appear in bold type. In some text, however, such as in lists, no special typographic treatment is used. Examples of such objects include:
  - File and command names
  - Data types and structures
  - Constants and flags
- Variable names that are supplied by user programs appear in italic type. In some text, however, such as in lists, no special typographic treatment is used. Examples of these objects include arguments and other parameters.

Names of objects and keywords known to the RISCWatch Debugger must be entered exactly as written.

## Syntax Diagram Conventions

Throughout this book, diagrams illustrate the syntax for string formats and commands. The following list shows how to read these diagrams:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.
- A  symbol begins a diagram.
- A  symbol indicates continuation of a diagram on the next line.
- A  symbol indicates continuation of a diagram from the previous line.
- A  symbol terminates a diagram.
- Keywords are in regular type, and variables are in italics. Keywords must be typed exactly as shown.
- Keywords or variables on the main path of a diagram are required.

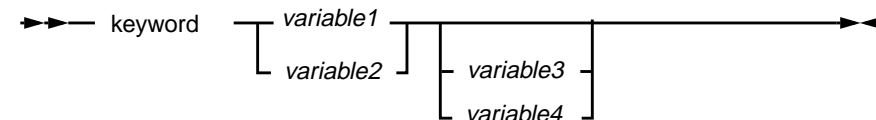


- Keywords or variables shown on branches below the main path are optional.

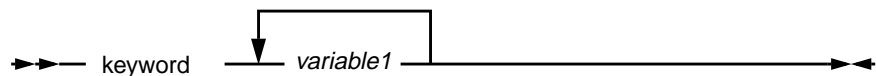


- Keywords or variables can appear in a stack, indicating that only one item in a stack can be chosen. If an item in a stack is on the main path, you must choose an item from the stack. If all items in a stack are below the main path, you may choose an item from the stack.

For example, in the following syntax diagram, you must choose either *variable1* or *variable2*. However, because *variable3* and *variable4* are below the main path, neither is required.



- A repeat separator is a returning arrow that surrounds a syntax element or group and shows that the element or group can be repeated.



---

## Where to Find More Information

The following sections list sources of information about or related to RISCWatch.

### Related IBM Publications

This book refers to the following publications, which are available from your IBM Microelectronics representative:

- **RISC System/6000 Publications**

*IBM RISC System/6000: POWERstation and POWERserver Hardware Technical Information General Architectures*, SA23-2643

- **AIX Publications**

This book refers to the following AIX publications. The words “IBM AIX Version 4 for RISC System/6000” are actually part of the title of each book; however, in all references to these books, those words are omitted.

*Assembler Language Reference*, SC23-2642

*Commands Reference*, Volume 1, SC23-2537

*Commands Reference*, Volume 2, SC23-2538

*Commands Reference*, Volume 3, SC23-2539

*Commands Reference*, Volume 4, SC23-2540

*Commands Reference*, Volume 5, SC23-2639

*Commands Reference*, Volume 6, SC23-2640

*Editing Concepts and Procedures*, GC23-2212

*Files Reference*, GC23-2200

- **XL C Compiler/6000 Publications**

*XL C Language Reference*, SC09-1260

*XL C User's Guide*, SC09-1259

- **IBM High C/C++ Publications**

The following list includes the books in the IBM High C/C++ library:

*IBM High C/C++ Programmer's Guide for PowerPC*, 92G6920

*IBM High C/C++ Language Reference for PowerPC*, 92G6923

*IBM ELF Assembler User's Guide for PowerPC*, 92G6921



*IBM ELF Linker User's Guide for PowerPC, 92G6922*

*PowerPC Embedded Application Binary Interface*

A copy of the EABI specification can be found at the RISCWatch web page at:

<http://www-e.ibm.com/chips/products/powerpc/tools/riscwadc.html>

- **OS Open Publications**

The following list includes the books in the OS Open library:

*IBM OS Open Programmer's Reference, Volume 1, 92G6911*

*IBM OS Open Programmer's Reference, Volume 2, 92G6912*

*IBM OS Open User's Guide, 92G6897*

- **PowerPC 400Series User's Manuals**

Visit

[http://www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC\\_Microprocessors\\_and\\_Embedded\\_Controllers](http://www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC_Microprocessors_and_Embedded_Controllers)

- **PowerPC 6xx/7xx User's Manuals**

Visit

[http://www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC\\_Microprocessors\\_and\\_Embedded\\_Controllers](http://www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC_Microprocessors_and_Embedded_Controllers)

- **PowerPC**

*PowerPC Microprocessor Family: The Programming Environments, MPRPPCFPE-01*



---

# Chapter 1. Introducing the RISCWatch Debugger

The IBM RISCWatch Debugger provides a powerful, flexible debugging environment to support hardware and software development using PowerPC processors in embedded applications.

---

## Embedded System Software Development

Embedded systems are typically developed in a cross-development environment consisting of host computers and target systems. The host computers provide software and project management tools for embedded system application developers. The developers are not restricted to the limited computing resources typically available on the target embedded system.

Developers write, compile, and debug embedded application programs on the host computers. When appropriate, the application programs are loaded on the target embedded system, where they run and are tested in the target operating environment.

Embedded system development is an iterative process; the application programs are refined on the host computers and tested on the target system until the programs meet the functional and performance requirements of the application. Eventually, the application programs are shipped as part of an embedded system.

## Programming Languages

Application programs for PowerPC processors are typically written in C/C++ and assembler. Formats currently supported include ELF/DWARF/DWARF2 (SVR4 ABI and PowerPC Embedded ABI) and XCOFF/STABS.

---

## Features

RISCWatch is a development and debug tool for PowerPC processors. RISCWatch employs a graphical user interface allowing complete access to all of the PowerPC processor functions. Following is a list of RISCWatch features:

- Robust source program debug capability
- Low-level hardware program debug (assembly level)
- Read, modify and write of all processor registers
- Read, modify and write of processor register fields
- User defined registers (DCR, SPR, Memory mapped)

- Read, modify and write of all processor memory (single, multi-byte access) with memory fill and write verification testing
- Memory loading of many types of file formats (ELF, XCOFF, Motorola 32-bit, and straight binary)
- Save/load processor memory image to/from file
- Save/load processor register values to/from file
- Command file execution, including nesting capabilities
- Command file execution with user-created variables, programming constructs, expressions and **printf**-like function
- Command file single-step execution
- Batch mode command file execution
- Program assembler and disassembler allowing memory read and write capability
- Single-step execution (assembly or source level) of loaded programs
- Set/clear of multiple-event breakpoints
- Saving and loading of customized window layout
- User-defined windows consisting of register, register field, memory, disassembly, command execution and status interfaces
- Processor reset functions
- Logging of all commands and messages
- File browsing
- Operating System command execution capability
- On-line help for all screens including extensive processor register definitions
- Multiprocessor support with User-defined board configurations
- Resizable windows with configurable user interface control colors
- User-defined Core+ASIC interface for customized chip support
- RISCTrace interface for PowerPC 400Series real-time trace debug
- Debug via remote host using C/PERL Programming Interface

---

## Chapter 2. Quick Start

Included with the RISCWatch debugger are some example files that can be used to quickly demonstrate some of the capabilities of the tool. They include all of the source, object, and executable files necessary to proceed with the following tutorial. The sections are designed to be performed sequentially, but the actions described in each can be applied at various stages of the debug session.

In general, the windows and descriptions will appear exactly as stated in the text. However, there may be slight differences in what is pictured versus what the user will actually see when running through the demonstration. For example, if the program is loaded in a location other than that specified in the **load** command, any addresses shown in the window might not match what appears in the document. However, the functions performed are equivalent.

---

### Starting the Debugger

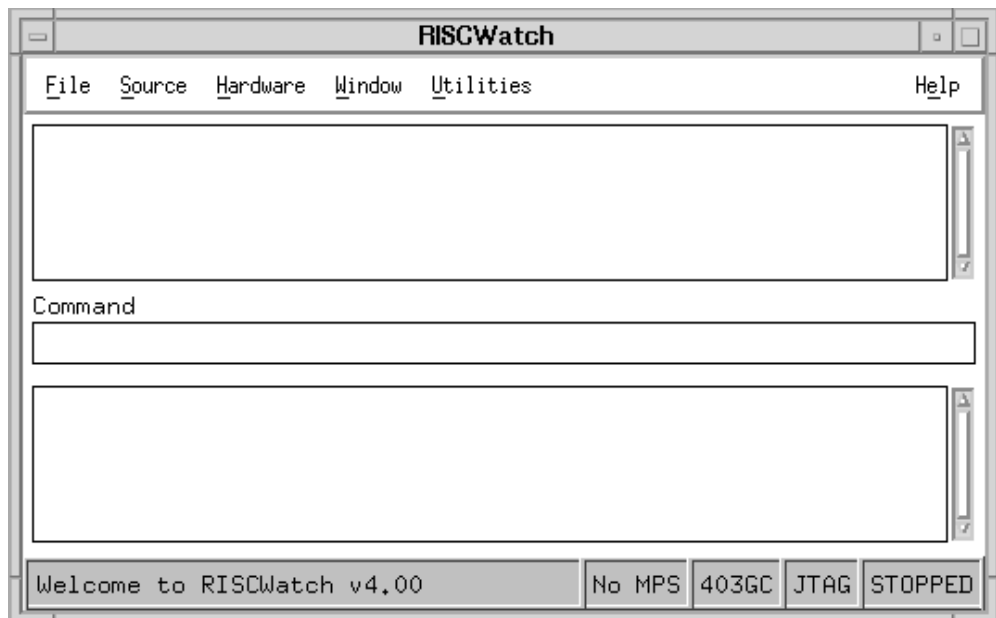


Figure 2-1. Sample Main Window

The Main Window, as illustrated in figure 2.1, is the first window seen when RISCWatch is started. Perform the following steps to display this window:

- Edit the “rwppc.env” file to designate the “target processor”, “target type”, “target name”, and RISCWatch directory, as described in “Environment Resources” on page 3-5 and “Invoking the Debugger” on page 3-35. Edit any additional environment resources required for your specific setup.
- From a RISC System/6000 or Sun workstation or Linux PC running Motif, change to the installation directory, and type “rwppc”.
- From a PC running Windows, double-click on the RISCWatch icon created during program installation.

---

## Entering Commands

To enter debugger commands from the command line of the Main window, single-click on the Command area to give it ‘focus’, type in the desired command, and then press “Enter”. See “Command Quick Reference” on page 5-3 for the complete list of valid commands.

For the demonstration program, enter the command “srchpath set xxxx”, where xxxx is the fully qualified directory path where the source files for ‘demo’ reside.

Note that when the command is entered, it is displayed in the command history window. It is also displayed, along with any associated messages, below the command line in the message window.

---

## Loading the Demo Program

Enter from the command line:

```
load file demo t=0x35000 d=0x37000
```

**Note:** An address of 0x35000 will work for most ROM Monitor targets. Refer to the Eval Kit User’s Guide (section 7.5) for instructions on how a valid address can be determined for ROM Monitor targets.

---

## Scrolling Through Source Code

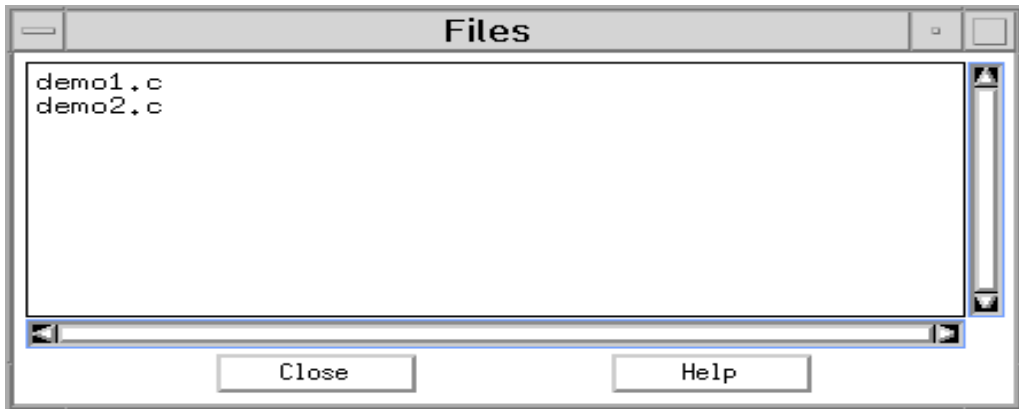


Figure 2-2. Sample Files Window

Now that the program has been loaded, the next step is to bring up the source files. Move the cursor to the to the "Source" menu bar entry on the Main screen and single click the left mouse button. Then, single click on the "Files" choice. Figure 2-2 shows the sample Files display.

Move the cursor to the to the "Source" menu bar entry on the Main screen and single click the left mouse button. Then, single click on the "Source" choice. The Source window should be displayed.

Single-click the left mouse button on the "demo1.c" entry in the Files window. It will become highlighted, and the following will appear in the Source Window:

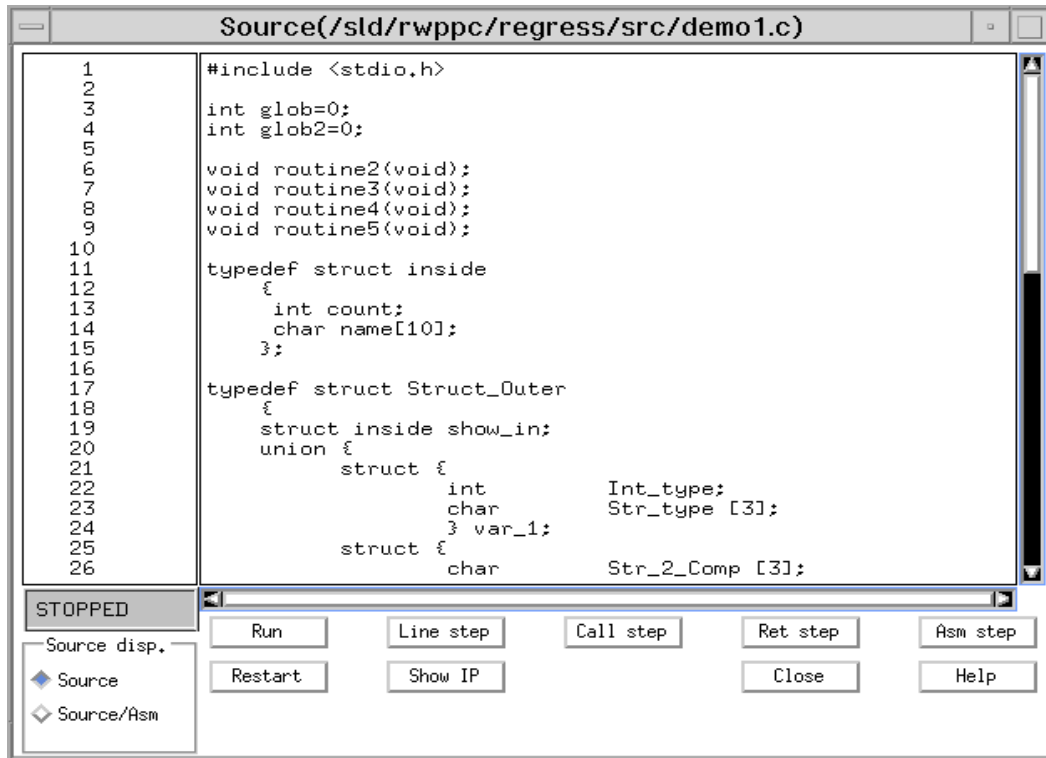


Figure 2-3. Sample Source Window

Move the cursor to the Main window, and single-click the left mouse button in the Command area to enable the command line.

Enter “pagedn source” on the command line. The source window will scroll down one page.

Enter “pageup” on the command line. The source window will scroll up one page. Notice that “source” wasn’t specified this time because the last command stored it for subsequent commands to use.

Move the cursor back to the Source window, and place the cursor on the down arrow found on the scroll bar area on the right side of the window. Hold down the left mouse button. The source code will scroll down a line at a time while the button is being held down. The scroll bar will also move down along the right side of the screen.

Move the cursor to the area above the scroll bar, placing it between the bar and the up arrow. Press the left mouse button once. This will move the source code up one page.

Move the cursor to the scroll bar itself. Hold down the left mouse button and move the mouse up and down. The source code will scroll up and down with the movement of the mouse.



Move the cursor back to the Main window, and single-click the left mouse button in the Command area to enable the command line.

Enter "top" on the command line. The Source window will scroll to the top of the source file.

---

## Setting Breakpoints

Move the cursor back to the Source window, and scroll down through the code until line 39 is in view.

Single-click the left mouse button, in the Source window left side which shows the line numbers, at the line 39 entry. A "BP" indicator will appear next to the line number 39. This means a breakpoint has been set.

Move the cursor to the "Source" menu bar entry on the Main screen and single-click the left mouse button. Then, single-click on the "Breakpoints" option. Figure 2-4 shows the display.

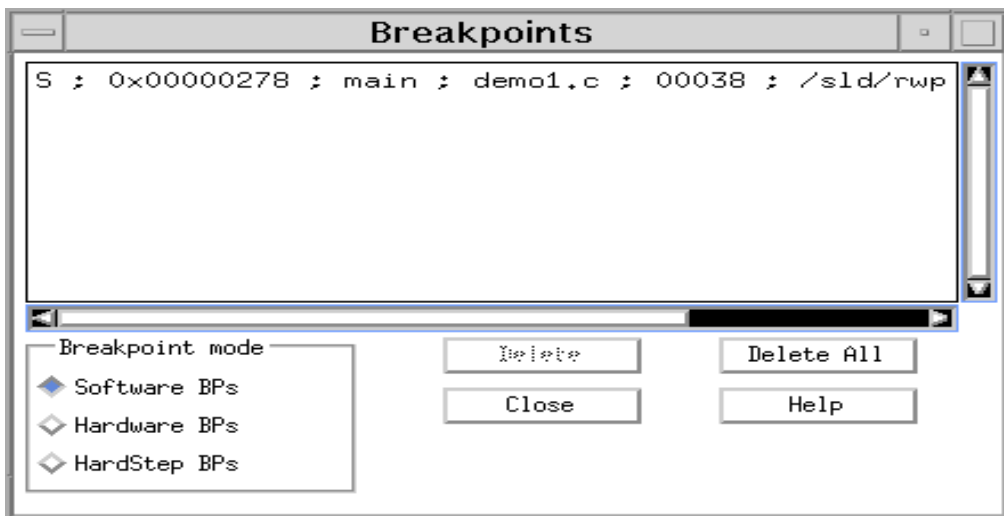


Figure 2-4. Sample Breakpoints Window

Various information about the breakpoint is displayed in the Breakpoints window, including type (hardware or software), address, function name, source file, and line number corresponding to the breakpoint.

Move the cursor button over the entry in the Breakpoints window and single-click the left mouse button. The entry is highlighted, and its corresponding location in the Source window is highlighted. The Delete button is also enabled.

Single-click the left mouse button again on the entry. The highlight is removed, and the Delete button is disabled.

Move the cursor to the to the “Source” menu bar entry on the Main screen and single click the left mouse button. Then, single click the left mouse button on the “Functions” option. Figure 2-5 shows the display.

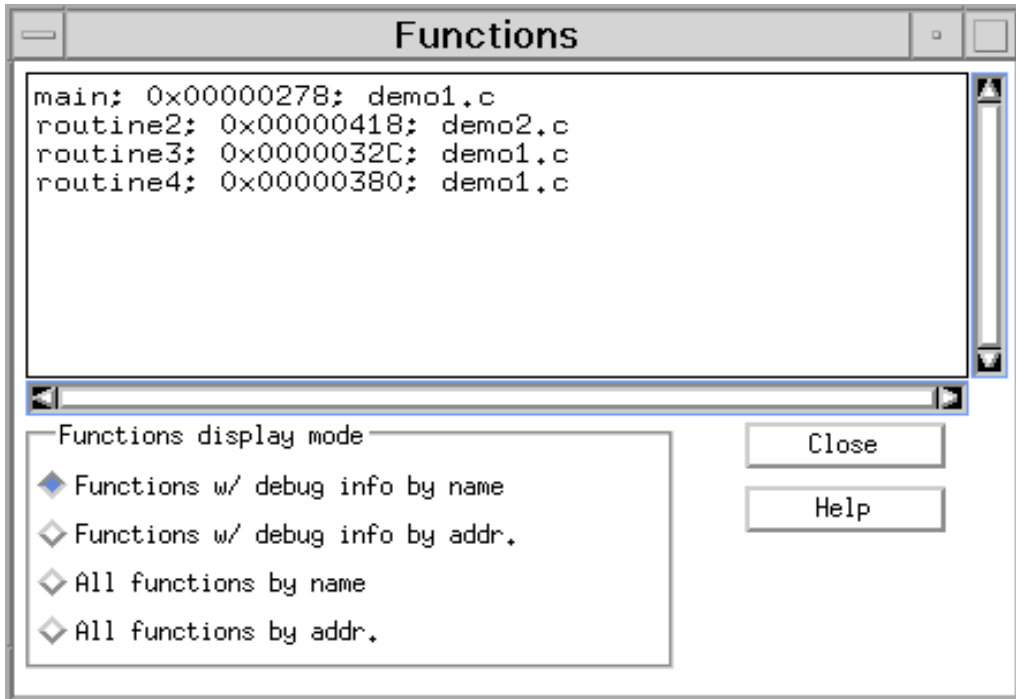


Figure 2-5. Sample Functions Window

Locate the entry “routine2; demo2.c”. Move the cursor to this entry, and single-click the left mouse button. The source file containing routine2 (demo2.c) will now be shown in the Source window, and the entry will be highlighted in the Functions window.

Double-click the left mouse button in the Functions window at the line corresponding to the “routine2; demo2.c” function entry. This will set a breakpoint at the beginning of the routine2 function. The “BP” indicator will appear in the Source window at the first executable line in the function. Information about the breakpoint will also appear in the Breakpoints window.

Move the cursor to the newly added routine2 entry in the Breakpoints window. Double-click the left mouse button on the entry. The breakpoint is removed from the Breakpoints, Functions, and Source windows.

---

## Stepping Through the Code

Move the cursor to the “Run” button in the Source window, and single-click the left mouse button. The program is “run” until it hits the breakpoint set earlier in this example. The source file corresponding to the breakpoint location that stopped the program execution is displayed in the Source window. The source line corresponding to the current Instruction Pointer address is indicated by the “>>” next to the line number where the program has stopped.

Press the “Show IP” button in the Source window. Information relating to the current Instruction Pointer is listed in the Main window status and message area.

Press the “Line step” button in the Source window. The “>>” appears on the next source line, which is now highlighted.

Move the cursor to source line 48, over the source line “routine4();” and single-click the left mouse button. The BP indicator appears next to the line, and the breakpoint entry is entered in the Breakpoints window.

Press the “Run” button once more, and the program runs to the break just set. The “>>” appears next to line number 48, which is now highlighted.

Move the cursor to the “Source” menu bar entry on the Main screen and single click the left mouse button. Then, single click the left mouse button on the “Callers” option. Figure 2-6 shows the display.

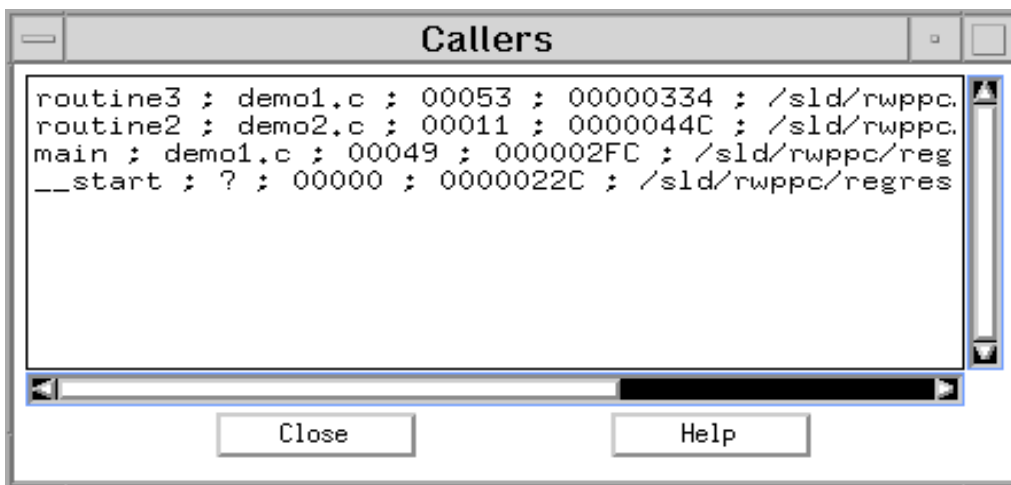


Figure 2-6. Sample Callers Window

The information contained in the Callers window is essentially a “push down” stack that contains information about the current call stack.

Press the “Line step” button in the Source window. The “>>” appears on the next source line, which is now highlighted. Notice the program did not step into the routine4() function. The Line step command essentially steps over function calls.

Now press the “Call step” button in the Source window. This command causes the debugger to enter the context of the called function. The file containing the routine2() function is displayed in the Source window. The first executable source line is highlighted, and the “>>” indicator shows the source line corresponding to the current instruction pointer. The Callers window is also updated to reflect the current debugger context. Press the “Line step” button in the Source window 3 times. The “>>” will be next to the source line “routine3():”, line number 11.

Now press the “Call step” button in the Source window. The file containing the routine3() function is displayed in the Source window. The first executable source line is highlighted, and the “>>” indicator shows the source line corresponding to the current instruction pointer. The Callers window is again updated to reflect the current debugger context, routine3.

Single-click on the “routine2” entry in the Callers window. The context is switched back to the function that made the call, namely routine2(), with the Source window being updated to show the file and line where the function call was made. The Callers window is used in this manner to traverse the call stack.

Press the “Show IP” button on the Source window. The current IP information is again displayed in the message area of the Main window. The Source window is also returned to the current context, which is the function listed at the top of the Callers window.

Press the “Ret step” button on the Source window. This returns the debugger context to the calling function. Notice that the Callers window is also updated as the stack entry is “popped” from the current call stack.

Press the “Ret step” button again, and the debugger traverses the stack again, returning to the original caller in main().

Now press the “Restart” button on the Source window. The program is essentially reloaded, and the instruction pointer is reset to the entry point of the program. Notice the breakpoints that have been saved and the messages that appear in the Main window.

The entry point in this example is in startup code that has no source files associated with it. Thus the debugger displays messages that indicate why it is unable to display code in the Source window.

Press the “Run” button. Since the breaks are still set, the program stops again at the breakpoint on line 39 in demo1.c.

---

## Altering and Displaying Variables

Move the cursor to the “Source” menu bar entry on the Main screen and single click the left mouse button. Then, single click the left mouse button on the “Locals” option. Figure 2-7 shows the display.

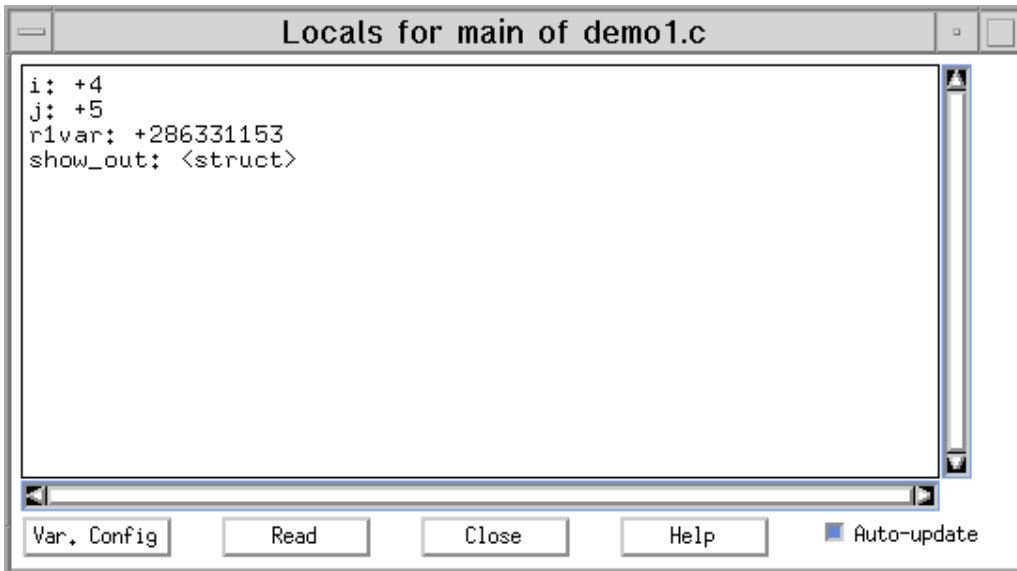


Figure 2-7. Sample Locals Window

This window lists all of the defined local variables in the current debugger context, and their current values. The window contents can be custom tailored in a variety of ways. Refer to “Variable Configuration Window” on page 3-83 for a complete description of the available options. Only a few will be shown in this example.

Press the “Var. Config” (Variable Configuration)” button on the Locals window. Figure 2-8 shows the window that will be displayed.

Press the “Address” button in the Display info. area.

Single-click on the variable “i” shown in the Visible area. This moves the variable to the Not Visible area, meaning the variable will no longer be shown. This is used to reduce clutter of uninteresting variables and also to reduce the number of variable values requiring refresh when the debugger context changes.

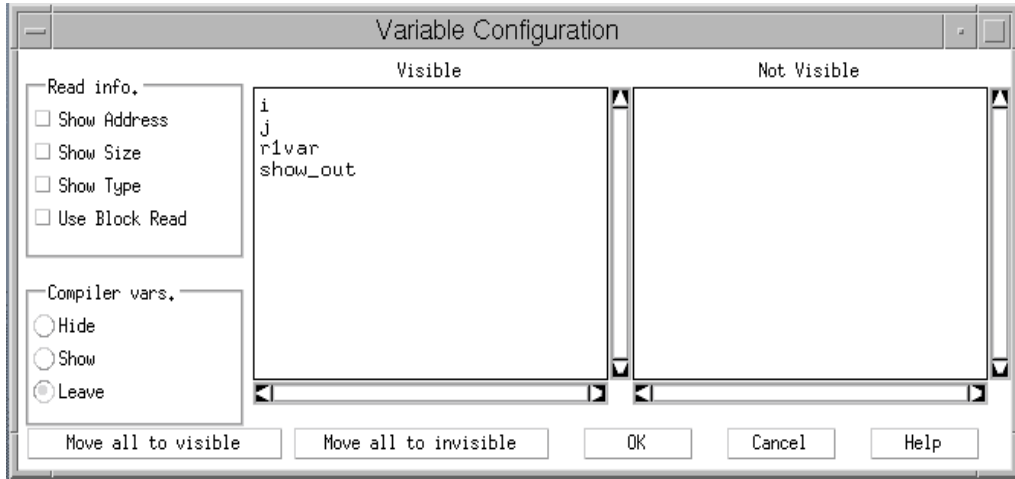


Figure 2-8. Sample Variable Configuration Window

Press the “OK” button in the Variable Configuration window. This applies the changes and removes the window. Notice variable “i” is no longer shown, and that the addresses of all the variables are now displayed.

Individual variables may also be custom tailored. Single-click on the “show\_out” variable in the Locals window. Figure 2-9 shows the display.

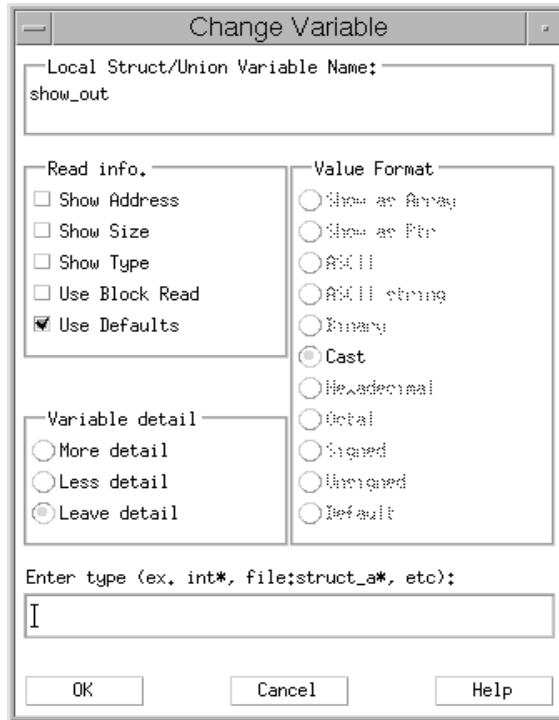


Figure 2-9. Change Display Information

The “Address” button in the Display info. field is selected because of the previous Variable Configuration window update. Press the button again to deselect the “Address” button. Press the “OK” button to apply the change and remove the window. Notice the Locals window display no longer shows the address of the show\_out variable.

Move the cursor again to the show\_out variable and double-click the left mouse button. Notice that the variable is “expanded” to show another level of detail of the structure. Double-click on the show\_out variable again to show even more detail. Move the cursor down three lines to the “.name:” variable name, and double-click on it. Notice that just that variable gets expanded even further.

Single-click on the “.name:” variable. Notice in the Change Array Variable window that the subrange shown can be tailored. Change the “0,2” to “2,6” and then press “OK”. Now only array elements 2-6 are shown in the Locals window for the “.name:” array.

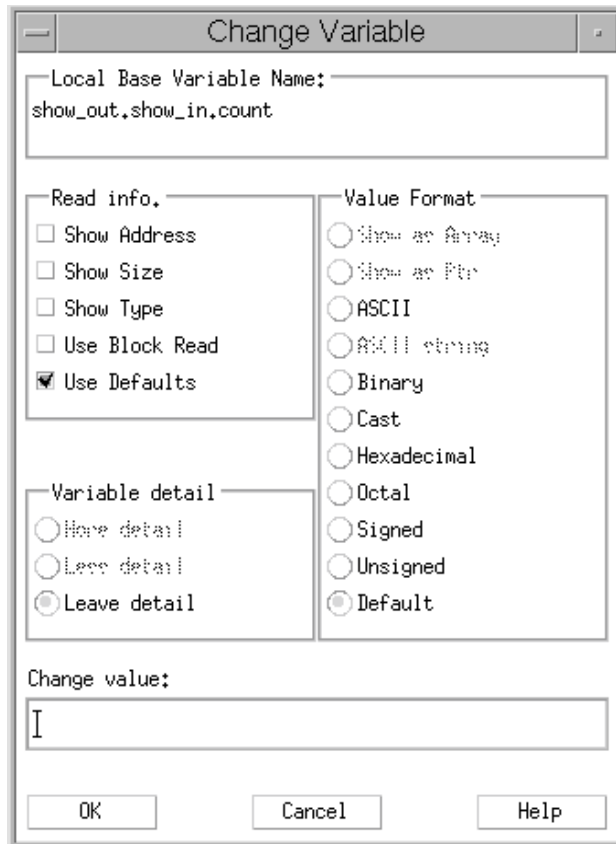


Figure 2-10. Change Base Variable

Single-click on the +2 next to the “.count:” variable. Figure 2-10 shows the display

Press the “Hexadecimal” button in the Value format field. Enter 10 in the Change value field, and press “OK”. Notice that the display for the “.count:” variable is now in hex, and reflects the decimal value 10 just entered. Single-click on the “r1var:” variable, and change the Value format to “Hexadecimal” as well. Press the “OK” button to change the variable.

Press the “Line step” button in the Source window. Notice no variables are updated since “i” was moved to invisible earlier. Press the “Line step” button again. Notice that the variable “show\_out.show\_in.count” got updated in the Locals window as the source line was executed.

The Globals window operates in the same manner as the Locals, but contains variables defined as global in the program.



---

## Debugging at the Assembly Level

Assembly level debug can be accomplished in several ways. One way is via a source disassembly in the Source window. Another is to use an actual memory disassembly found in the Assembly Debug window.

Press the “Delete All” button on the Breakpoints window. Notice that all the breakpoints are cleared in both the Source and Breakpoints windows. Single-click on the source code of line 47 in the Source window to set a breakpoint. Run to that breakpoint by pressing the “Run” button in the Source window.

Press the “Call step” button in the Source window. Notice that the source file associated with the called function, routine5, is shown as “?” in the Source window. In addition, some of the buttons have been disabled, and some warning messages have been posted in the Main window. Also, no local variable information is available.

This is a result of stepping into a function that was compiled with no debug information—a prime example of why it might be desirable to do assembly level debug with a source level debugger. Notice also that the warning message presents the opportunity to return immediately to the calling function in case the Call step issued was inadvertent, or the user decides not to step through the assembly code.

But since you are still reading this, we’ll have to assume you are a hard core user and want to move on! Move the cursor back to the Main window to the “Hardware” menu bar entry and single-click the left mouse button. Then, single-click on the “Asm Debug” option. Figure 2-11 shows the display.

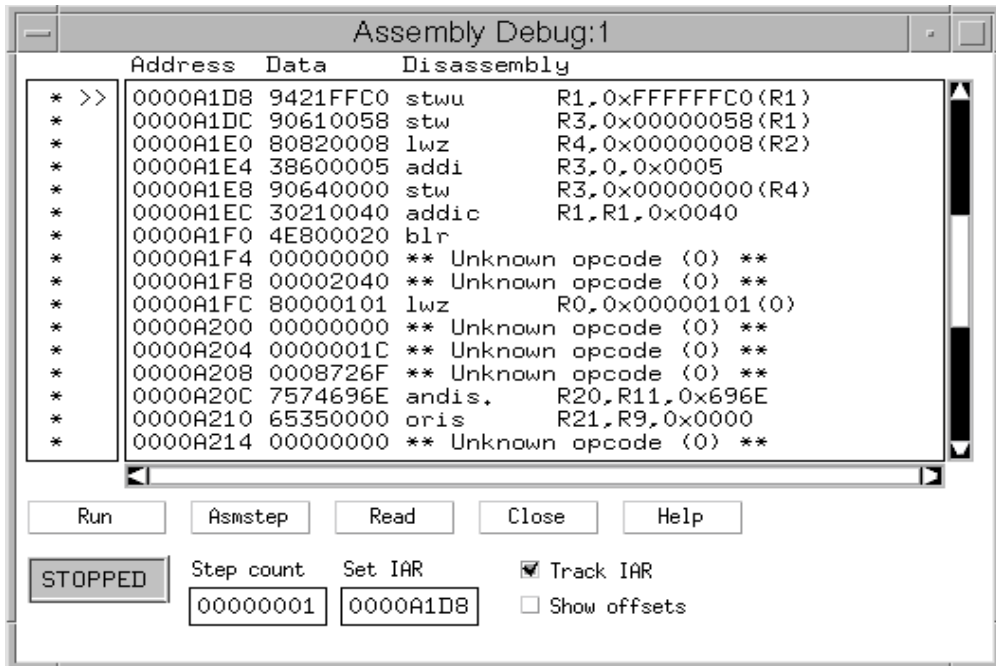


Figure 2-11. Sample Assembly Debug Window

This contains a memory disassembly of a number of instructions, beginning with the one corresponding to the current instruction pointer. Press the “Asmstep” button in the Assembly Debug window. Notice the current instruction indicator has moved to the next assembly instruction. Also notice that the “Return step” button on the Source window has been disabled.

This is the debugger’s way of politely saying that you had your chance to return easily per the previous warning message, and you chose not to, so you’re on your own getting back!

This can be done either by pressing the “Asmstep” button until the return is made, or by going back to the source line calling the function and setting a break after the line and running to it. We’ll do the former since this function has only a few instructions.

Press the “Asmstep” button until the return is made to the calling function. The Source window is updated to show the source file containing the original call. Notice that the current instruction pointer is still pointing to the line number containing the call.

The source disassembly feature can be used to show why this is the case. Press the “Source/Asm” button in the Source Mode area of the Source window. This produces a mixed source and disassembly listing in the window. Notice that there is more than one assembly

instruction associated with each source line. In our example, we returned from the function call, but we're still on the same source line as the call itself.

Breakpoints can also be set while in mixed mode. Move the cursor to the "crr 31,31,31" instruction below the routine2() source line and single-click on it. Notice that the breakpoint is indicated in the Source, Assembly Debug, and Breakpoints windows.

Press the "Run" button in the Source window. Notice that the current instruction pointer is updated at the breakpoint address in both the Source and Assembly Debug windows.

Press the "Source only" button in the Display mode area in the Source window. Notice that the break is still shown on the source line corresponding to the assembly line on which the breakpoint was set.

Numerous other screens are also useful when doing assembly level debug. Please refer to Table 3-1, "Quick Reference for the RISCWatch Debugger" on page 3-2, for a list of the available windows.



---

## Chapter 3. Using the RISCWatch Debugger

RISCWatch is designed to be run in one of several configurations:

- Normal mode

The user interacts with the graphical user interface. This is the mode in which RISCWatch is usually run.

- Command File Batch mode

RISCWatch runs via commands contained in an ASCII file. A shell script can, for example, invoke RISCWatch several times with several command files. The graphical user interface is not available in this mode. See “Running a Command File” on page 3-131 for more details on how to run RISCWatch in this mode.

- TTY mode (not supported for Windows host)

This mode allows RISCWatch to be run on a UNIX workstation which does not have a graphical user interface windowing system available. This mode provides a command line interface where commands are typed in after a TTY prompt and resulting execution messages are printed to the terminal. This mode is invoked by starting RISCWatch with the `-tty` command line option.

- Programming Interface mode<sup>†</sup>

This mode allows RISCWatch to be run as a “server” on one host while a C or PERL program runs on a “client” host and connects to it. This connection allows the “client” program to send RISCWatch commands to the RISCWatch “server” (which may be running on a remote host) for execution. This mode is invoked by starting RISCWatch with the `-pi` command line option.

Target types currently supported by RISCWatch are described in “Environment Resources” on page 3-5.

---

### Debugger Facilities

The RISCWatch Debugger has many facilities that can be used to develop, test, and debug your evaluation board code and programs. As you find it necessary to perform certain tasks, this section can be used as a quick lookup of the facilities that might be used to accomplish those tasks. Table 3-1 below provides a quick reference to RISCWatch resources, both in this chapter on general debug features and in the next chapter on processor-specific debug features.

---

<sup>†</sup>

**Table 3-1. Quick Reference for the RISCWatch Debugger**

Task or Resource	Applicable Sections
<b>Setting the Environment</b> How to initialize the environment resources, register definition files, and multi-processor files	“Environment Resources” on page 3-5 “Core + ASIC Resources” on page 3-8 “Multi-Processor Resources” on page 3-28
<b>Invoking the Debugger</b> How to bring up the RISCWatch Main Window	“Invoking the Debugger” on page 3-35 “JTAG Ethernet Targets and the RISCWatch Processor Probe” on page 3-38
<b>Main Window Resources</b> Overview of menus and windows	“Main Window Resources” on page 3-40 “Menus” on page 3-41 “Command Line Usage” on page 3-44 “Command History Usage” on page 3-44 “Message Window” on page 3-45
<b>Running Your Programs</b> How to compile, load, and execute programs	“Preparing the Program for Debug” on page 3-45 “Loading Files” on page 3-45 “Loading Boot and Boot Image Files” on page 3-47 “Executing the Program” on page 3-48 “Following Program Execution Flow” on page 3-48 “Input Line Usage” on page 3-49 “Scrolling Source Window Contents Using the Keyboard” on page 3-55
<b>Source Level Debugging</b> How to use the interface to debug your C source code	“Source Window” on page 3-52 “Assembly Debug Window” on page 3-55 “Programs Window” on page 3-59 “Callers Window” on page 3-60 “Files Window” on page 3-62 “Functions Window” on page 3-62 “Load Memory Window” on page 3-64
<b>OS Open Debugging</b> How to use the interface to display operating system information and to control debug attachment	“OS Open Debugging” on page 3-67

**Table 3-1. Quick Reference for the RISCWatch Debugger**

<b>Task or Resource</b>	<b>Applicable Sections</b>
<b>Managing Breakpoints</b> How to use the interface and command set to set hardware and software breakpoints	“Managing Breakpoints” on page 3-71 “Using Software Breakpoints” on page 3-72 “Using Hardware Breakpoints” on page 3-73 “Breakpoints Window” on page 3-74 “Breakpoint Select Window” on page 3-76 “Trigger/Trace Window (400Series Only)” on page 4-7 “Compound Trigger/Trace Window (403Series Only)” on page 4-11
<b>Reading and Writing Program Variables</b> How to use the interface to read, modify, and write program variables	“Reading and Writing Program Variables” on page 3-77 “Local Variables Window” on page 3-78 “Global Variables Window” on page 3-79 “Inspect Variable Windows” on page 3-81 “Variable Configuration Window” on page 3-83 “Change Variable Window” on page 3-85 “Formatting Examples” on page 3-88 “Source Variable Command Support” on page 3-102
<b>Reading and Writing Memory</b> How to use the interface and command set to read, modify, and write processor memory in many different formats	“Reading and Writing Memory” on page 3-103 “Assembly Debug Window” on page 3-55 “Memory Coherency Window (JTAG Targets Only)” on page 3-104 “ASCII Memory Window” on page 3-107 “Custom Memory Window” on page 3-109 “Cache Windows (JTAG Targets Only)” on page 3-112 “Translation Lookaside Buffer Window (Applicable Processors Only)” on page 4-14 “Load Memory Window” on page 3-64 “Save Memory Window” on page 3-113
<b>Reading and Writing Registers</b> How to use the interface and command set to read, modify, and write processor registers and register fields	“Reading and Writing Registers” on page 3-115 “Register Windows” on page 3-115 “Register Field Windows” on page 3-117

**Table 3-1. Quick Reference for the RISCWatch Debugger**

<b>Task or Resource</b>	<b>Applicable Sections</b>
<b>User-Defined Windows</b> How to create and run customized windows	“User-Defined windows allow a RISCWatch user to create windows containing customizable register, register field, memory, disassembly, and button entries. Using a simple syntax, ASCII files are created to define the contents of a user-defined window.” on page 3-118
<b>Command Files</b> How to create and run command files which are used to perform repetitious tasks and help to automate testing	“Command Files” on page 3-123 “Command File Programming” on page 3-125 “Command File Special Expressions” on page 3-127 “Command File Parameters” on page 3-128 “Command File Pseudo-Variables” on page 3-129 “Running a Command File” on page 3-131 “Command File Programming Example” on page 3-130 “Running a Command File” on page 3-131 “Command File Window” on page 3-132
<b>Processor Resources</b> How to use the interface to perform processor resets and to read processor status	“Processor Resources” on page 3-134 “Processor Reset Window (JTAG Targets Only)” on page 3-134
<b>General Resources</b> How to use various program resources	“Window Layout” on page 3-135 “Output Window” on page 3-136 “Window List” on page 3-138 “User Created Variables Window” on page 3-138 “Log Files” on page 3-139 “Logging Control” on page 3-140 “Logging User Comments” on page 3-140 “Screen Capture” on page 3-141 “Calculator Window” on page 3-142 “Online Help” on page 3-143
<b>RISCTrace</b> Describes using RISCTrace and the trace capabilities of 400Series processors	“Using RISCTrace (400Series JTAG Processor Probe Only)” on page 4-2

It may prove helpful to glance through each of the sections listed in Table 3-1 to gain an overall picture of the available facilities that RISCWatch offers. Such an understanding can help you avoid doing something “the hard way.”



---

## Environment Resources

RISCWatch employs an environment resources file to specify or configure various resources. This file, **rwppc.env**, is designed to allow the RISCWatch user to tailor program operation to meet specific operating preferences. This file should be examined and changed where necessary, before RISCWatch is run to ensure that the environment will conform to your debugging needs.

What follows is a list of the environment resources that can be used in the **rwppc.env** file and their functionality:

<b>Environment variable</b>	<b>Description</b>
PROC	Specifies the target processor name for non-MPS RISCWatch debug sessions ( <b>required</b> ). See the README file provided with RISCWatch for a list of valid processor names.
REV	Specifies the revision number of the target processor. This field is required when debugging a 6xx/7xx processor in which RISCWatch supports more than one revision number. For example, if debugging a 603e REV 3 processor, "REV = 3" must be designated.
TARGET_TYPE	jtag_eth, rom_mon, osopen (one <b>required</b> ) Refer to the README file which came with RISCWatch for information concerning host and target requirements for proper RISCWatch operation. Each target type is described below.
jtag_eth	JTAG Ethernet target. RISCWatch is connected via Ethernet to a RISCWatch processor probe. The JTAG connector of the processor probe is then connected to the JTAG port on the PowerPC 400Series or PowerPC 6xx/7xx target system.
rom_mon	ROM monitor target. RISCWatch is connected via Ethernet or SLIP to a PowerPC target system running the IBM ROM Monitor for PowerPC in debug mode.
os_open	OS Open target. RISCWatch is connected via Ethernet or SLIP to a PowerPC target system running IBM's OS Open real-time operating system.
TARGET_NAME	Name of target found in TCP/IP services file ( <b>required</b> for JTAG Ethernet, OS Open and ROM Monitor targets) TCP/IP dotted address may also be used.
TARGET_PORT	Port number of TCP/IP target. If this field is not specified, the default value of 6470 or the value read from the TCP/IP services file will be used.

RWPPC_DIR	A fully qualified path name to the directory in which the RISCWatch executable and support files reside. This is <b>required</b> for all targets.
SEARCH_PATH	Path names used for source/object/command file search, delimited by colons (:); for a Windows host, the delimiter is a semicolon instead of a colon. (if not specified, default = current directory)
LOG_FILE_DIR	A fully qualified path name to the directory of where RISCWatch is to maintain all log files.
STACK_FRAMES	Indicates the number of stack frames to show on the Callers Window. If not designated, the default setting is twelve.
LAYOUT	Save/load window layout when ending/beginning session. "SAVE" will save the layout on exit. "LOAD" will load the layout when starting. "LOADSAVE" (or omitting the variable altogether) will do both. "NONE" will do neither.
APPLPROG_NAME	Allows renaming of applprog executable (OS Open target only)
FONT_SIZE	Specifies the font size to use in the main window for the text in the command history and message windows. This size should be one of 8, 10, 12 or 14.
COLOR_CTRL_BG	Specifies color for background control areas (non-MPS)
COLOR_CTRL_FG	Specifies color for foreground control areas (non-MPS)
COLOR_TEXT_BG	Specifies color for background text areas (non-MPS)
COLOR_TEXT_FG	Specifies color for foreground text areas (non-MPS)
COLOR_WIN_BG	Specifies color for background window areas (non-MPS)
COLOR_WIN_FG	Specifies color for foreground window areas (non-MPS)
MPS_FILE	Specifies file containing multiprocessor support configuration and options. See "Multi-Processor Resources" on page 3-28
PRD_FILE	Specifies a single file containing user defined registers, register fields and chips. See "Core + ASIC Resources" on page 3-8 for additional information
STARTUP_FILE	Specifies a command file which will be run each time RISCWatch is started. See "Command Files" on page 3-123.
PROBE_FLASH	Specifies the firmware loading sequence for JTAG ethernet targets.
AUTO	When RISCWatch is first invoked, a time stamp comparison is made between the preloaded Processor Probe firmware and the firmware files (drivers) provided with RISCWatch. If the time stamps do not match, the RISCWatch firmware files

	are loaded. This is the default operation of RISCWatch if PROBE_FLASH is not designated in the environment file.
NO	The Processor Probe will not be loaded with the RISCWatch firmware files. Any firmware file time stamp checks are ignored. <b>Warning:</b> The user assumes responsibility for guaranteeing that the firmware loaded on the Processor Probe is compatible with the target processor and the version of RISCWatch being used.
YES	The Processor Probe will be loaded with the firmware files provided with RISCWatch. Any firmware file time stamp checks are ignored. RISCWatch initialization time will be extended to load both the generic and processor specific driver files.
DEFAULT_TCK	Defines the JTAG clock speed used by RISCWatch at startup. It can be set to any value which is accepted by the “jtag clock” command. For example: DEFAULT_TCK = 10M.
TRACE_ENABLE	Specifies one or more RISCWatch commands to be executed prior to starting trace. If more than one command is needed, each additional command must be preceded by a semicolon (i.e. TRACE_ENABLE = write 0x0 22;set gpio = 15). This optional variable is only applicable to PowerPC 400Series Core+ASIC JTAG targets, where unique register or memory initialization may be required to enable the trace feature. The designated commands will be executed during the processing of the <b>trace run</b> command, which can be called from the RISCWatch Trigger or Compound Trigger windows. Please note that the <b>exec</b> command is not a valid TRACE_ENABLE designation.
TRACE_DISABLE	Specifies one or more RISCWatch commands to be executed immediately following the first stop request after a trace run. If more than one command is needed, each additional command must be preceded by a semicolon (i.e. TRACE_DISABLE = write 0x0 22;set gpio = 15;memacc clear). This optional variable is only applicable to PowerPC 400Series Core+ASIC JTAG targets, where unique register or memory initialization may be required to disable the trace feature. The designated commands will be executed during the processing of the <b>trace run</b> command, which can be called from the RISCWatch Trigger or Compound Trigger windows. Please note that the <b>exec</b> command is not a valid TRACE_DISABLE designation.
IR_SEG0	Specifies a binary sequence to be added to each JTAG SCAN_IR command. This variable is only applicable to PowerPC 400Series Core+ASIC JTAG targets. The binary sequence indicates the number of bits added to the SCAN_IR

chain as well as the value needed to communicate with the core processor. For example, 'IR\_SEG0 = 100' indicates the value of three additional bits that are connected to the end (TDO) of the SCAN\_IR chain. The user assumes responsibility for identifying the correct binary sequence.

File syntax consists of placing the resource name on a new line, and then following it with one or more spaces, an equal sign, one or more spaces and then specifying the resource value.

For example:

```
RWPPC_DIR = /usr/rwppc
```

To enhance readability of this file, comment and blank lines are allowed. A comment can only start in the first column and does so by beginning with the # character.

Every time RISCWatch is started, it attempts to locate the environment resources file using the following rules:

1. Check to see if it is in the current directory; if so, use it, else
2. If a relative or absolute path is given for the executable, see if the environment file is in the same directory as the executable; if so, use it, else
3. Check to see if it is in a directory specified by the environment variable PATH; if so, use it, else
4. Print an error message and terminate RISCWatch.

---

## Core + ASIC Resources

With the introduction of the IBM PowerPC 401 Core and a growing library of on chip peripherals, IBM offers high-performance custom processors. Using a single PowerPC core, hundreds of unique chips can be developed to satisfy specific customer needs.

Many of the basic functions performed by a debugger (line stepping, memory display, etc.) depend on both PowerPC core and peripheral resources. RISCWatch allows users to define both the register organization and the memory configuration of their Core+ASIC environment. The following user interfaces are provided to accomplish this task:

1. Processor Definition File: used to define the processor and ASIC resources that RISCWatch will need to access
2. **memacc** Command: used to define the correct access size and read/write restrictions of any memory access initiated by RISCWatch
3. Window Descriptor File: used to create User-Defined RISCWatch windows which can be used to display specified registers, register fields, or memory regions.

The following sections provide additional details about these Core+ASIC interfaces. Please read all sections to get a complete understanding of the flexibility RISCWatch provides for custom chip designs.

## Processors, Cores and Chip Resources

When RISCWatch is first started, the Processor Resource Definition (PRD) file is read to determine the debug environment. The PROC environment variable, designated in the environment file (**rwppc.env**), is used as an index into the PRD file to completely define the unique resources (processor, cores, registers, TLBs, caches, etc.) of the specified target. The default PRD file is updated with each new version of RISCWatch to contain the latest definitions for all standard PowerPC chips.

With the increasing popularity of Core + ASIC designs, a flexible debug solution is needed to allow debugging on non-standard, customer specific PowerPC parts. RISCWatch makes use of Processor Configuration Files (PCF) to provide this solution. These ASCII text formatted files allow users to define the processor, core, and chip resources (registers, cache, memory, etc.) needed to uniquely define a custom design.

Users working on standard PowerPC parts (403GCX, 603e, etc.) can use the resources defined in the default RISCWatch PRD file (**rwppc.prd**) for their debugging sessions. This file is loaded by default and is totally transparent to the end user. If a custom Core + ASIC solution is being used, or additional register definitions are needed, a new PRD file will need to be created by compiling one or more customer supplied Processor Configuration Files.

## Processor Configuration File (PCF)

Processor Configuration Files (PCF), introduced with RISCWatch version 4.3, are created by the user prior to starting RISCWatch. These files contain the resource definitions required to uniquely define the target debug environment. This new file format, with its enhanced support for Indirectly Mapped Registers, replaces the Register Definition File format of previous releases (see Appendix B, "Register Definition File (Outdated)").

Once created, the PCF file must be compiled into a loadable form. The user must invoke the RISCWatch PCF compiler program (**rwpcf**) with the PCF file name designated as an input argument. If no errors are detected, a new Processor Resource Definition (PRD) file is created. This new file will have the same PCF base file name, with a file extension of ".prd".

The RISCWatch **PRD\_FILE** environment variable is used to identify the name of the customized PRD file. Upon initial program startup, RISCWatch will read the designated PRD file to determine the unique target debug environment. Only one PRD file can be designated and it is located using the following rules:

- If the file name is qualified (directory path indicated), the file search is performed using the specified directory only.
- If the name is not qualified, the file search is performed using the directory paths designated with the RISCWatch **SEARCH\_PATH** environment variable. If not found, the current directory is searched.

## File Management

The Processor Configuration File (PCF) is an ASCII file that can be created with any text editor. Once defined, it must be compiled by the **rwpcfc** compiler program. Typically, a PCF file contains definitions for a unique ASIC macro.

The easiest approach is to simply put all definitions into a single PCF file so all the necessary information can be easily located and edited as needed. Other times it might make more sense to create multiple files, each containing the necessary details for a separate macro. It will be up to each user to determine which method works best for them.

If just one PCF file is created, it can be simply compiled on its own. If the multiple PCF file approach is used, there needs to be one “master” PCF which is used to gather the other low-level PCF files into one, coherent entity. When the low-level PCF files are compiled, they need to use the **-refer** flag which tells the compiler that they will later be included, or referred to (using **REFER**), by another file.

Regardless of how deeply these low-level files are nested, if they will be included by another PCF file, they need to be compiled with the **-refer** flag. Once it is time to compile the highest level “master” PCF file, the **-refer** flag is NOT used since it represents the highest level of user compilation.

Successful compilation of the highest level “master” PCF file will result in a valid PRD file whose name can then be specified in the RISCWatch environment file with the **PRD\_FILE** variable.

RISCWatch comes with a default PRD file (**rwppc.prd**) which is automatically loaded if no **PRD\_FILE** environment variable is specified.

## File Syntax

The general syntax rules for the PCF file are as follows:

1. The “#” character denotes the start of a comment. All text following the “#” character on a given line will be ignored.
2. Blank lines are allowed and will be ignored.
3. Any error detected while compiling the PCF will generate error messages and terminate execution of the compiler.
4. Hex values are preceded by ‘0x’, such as 0x12AB0423.
5. Implied Hex values are not preceded by ‘0x’, such as ABCD1245.

The following sections define the complete list of PCF definitions and their valid line entries.

## REFER Definitions

The **REFER** definition indicates that the current PCF file will use resources defined in another PCF file. This functionality allows a PCF file to refer to resources defined in another. The **rwpcfc** compiler does not allow a reference before a definition. For this reason, **REFER** definitions should appear at the very top of the PCF file so that when resources are referenced later in the current PCF file, the compiler will already be aware of them.

Each **REFER** definition must adhere to the following syntax:

```
REFER filename.prd
```

Where:

- *filename.prd* is the name of the PRD file which is the result of successfully compiling its associated PCF file.

**Note:** If a resource is referenced before it has been defined, an error message will be generated and the compile will be aborted. Place the **REFER** definitions near the top of the file to define all resources before they are referenced further down in the file.

## MACRO Definitions

The **MACRO** definition is the basic building block of the PCF file and allows for a high degree of modularity and reusability. By carefully defining PCF file **MACROs** along logical or functional divisions, references can be made to them by countless other definitions. Such abilities may not seem important if only one chip is being defined but it becomes crucial if the same or similar resources are used in multiple ASIC designs.

Each **MACRO** definition must adhere to the following syntax:

```
DEFINE MACRO macro_name
```

```
macro_def
```

```
END
```

Where:

- *macro\_name* is the unique name being assigned to this **MACRO** and is the name by which other definitions will reference it. This name can only contain alpha-numeric and underscore (\_) characters.

**Note:** the prefix "PPC\_" has been reserved for debugger use. Therefore no *macro\_name* may begin with "PPC\_"

- *macro\_def* represents one or more **MACRO** definitions, each of which defines one resource. The resources which can be defined in a **MACRO** are fields (**FIELD** definition), registers (**REG** definition) and register fields (**REGFLD** definition). See below for information about these definitions.

**Note:** If a resource is referenced before it has been defined, an error message will be generated and the compile will be aborted.

## CHIP Definitions

There are two forms of **CHIP** definitions. One is used to define a chip based on an existing RISCWatch-defined core (**DEFINE CHIP**) while the other is used to add resources to an existing RISCWatch-defined chip (**APPEND CHIP**).

A **DEFINE CHIP** definition must adhere to the following syntax:

```
DEFINE CHIP chip_name  
    dc_def  
END
```

Where:

- *chip\_name* is the unique name being assigned to this **CHIP** and is the name used to set the **PROC** environment variable.
- *dc\_def* represents one or more **DEFINE CHIP** entries, each of which defines or includes one resource. The resources which can be defined are names (**NAME** definition), PVRs (**PVR** definition) and firmware revisions (**REV** definition). The resources which can be included are macros and a single core (**INCLUDE** definition). See below for information about these definitions.

The following restrictions apply to a **DEFINE CHIP** definition

1. There must be one and only one **INCLUDE CORE** definition
2. There may be zero or more **INCLUDE MACRO** definitions
3. There may be zero or more **DEFINE NAME** definitions
4. There must be at least one **DEFINE PVR** definition

**Note:** If a resource is referenced before it has been defined, an error message will be generated and the compile will be aborted.

An **APPEND CHIP** definition must adhere to the following syntax:

```
APPEND CHIP chip_name  
    ac_def  
END
```

Where:

- *chip\_name* is the unique name being assigned to this **CHIP** and is the name used to set the **PROC** environment variable.
- *ac\_def* represents one or more **APPEND CHIP** entries, each of which defines or includes one resource. The resources which can be defined are exec (**EXEC** definition), field (**FIELD** definition), name (**NAME** definition), register (**REG** definition), register field (**REGFLD** definition), register alias (**REGALIAS** definition), PVRs (**PVR** definition) and firmware revisions (**REV** definition). The resources which



can be included are macros and a single chip (**INCLUDE CHIP** definition). See below for information about these definitions.

The following restrictions apply to an **APPEND CHIP** definition

1. Any included **MACRO**s may only contain **EXEC**, **FIELD**, **REG**, **REGFLD** and **REGALIAS** entries
2. There may be zero or more **EXEC** definitions.
3. There may be zero or more **FIELD** definitions.
4. There may be zero or more **REG** definitions.
5. There may be zero or more **REGFLD** definitions.
6. There may be zero or more **REGALIAS** definitions.
7. There must be one **INCLUDE CHIP** definition.

**Note:** If a resource is referenced before it has been defined, an error message will be generated and the compile will be aborted.

For the list of RISCWatch defined **CHIP**s and **CORE**s which are available for use, see the on-line FAQ available via the program Help pulldown, or visit the RISCWatch Support Center web page.

The following sections define the complete list of valid line entries for the **DEFINE MACRO**, **DEFINE CHIP** and **APPEND CHIP** definitions.

## **INCLUDE Definitions**

There are three forms of **INCLUDE** definitions which allow the inclusion of a previously defined **CHIP**, **CORE**, or **MACRO** definition.

An **INCLUDE CHIP** definition must adhere to the following syntax:

**INCLUDE CHIP** *chip\_name*

Where:

- *chip\_name* is the unique name of the chip being included. These chip names are defined by IBM and are representative of the parts contained in the PowerPC processor library. Examples of valid chip names include PPC\_403GCX, PPC\_603EV, PPC\_740, PPC\_750, etc..

An **INCLUDE CORE** definition must adhere to the following syntax:

**INCLUDE CORE** *core\_name*

Where:

- *core\_name* is the unique name of the core being included. These core names are defined by IBM and are representative of the parts contained in the PowerPC ASIC

library. Examples of valid cores names include PPC\_401M1\_CORE, PPC\_401B2\_CORE, etc.

An **INCLUDE MACRO** definition must adhere to the following syntax:

```
INCLUDE MACRO macro_name [offset] [prefix] [REL|RELOCATE]
```

Where:

- *macro\_name* is the unique name of a previously defined **MACRO** definition.
- *offset* is an optional address offset which is used for relocating register **MACRO**s. The offset is added or subtracted (depending on the specified sign) for each **DCR** or **MMIO** register definition contained in the referenced **MACRO**. This allows a "base" definition to be defined once and then relocated anywhere in the DCR/memory address space.
- *prefix* is an optional alpha-numeric prefix which, if specified, is used to replace an existing prefix for each register definition contained in the reference **MACRO**. This allows a base definition to be defined once (typically in a Core+ASIC environment) and then included multiple times with each instance defining a different prefix to be used. See the Note under **REG Definitions** for details regarding the proper use of register prefix naming conventions.
- **REL|RELOCATE** is an optional keyword for **MMIO** register definitions contained in the referenced **MACRO**. When specified, a base address is calculated for the **MMIO** registers in the **MACRO** and displayed on the register window in a field that can be later modified by the user. This keyword, in effect, allows the user to relocate the defined registers to any location in memory.

**Note:** If a resource is referenced before it has been defined, an error message will be generated and the compile will be aborted.

For the list of RISCWatch defined **CHIP**, **CORE** and **MACRO** names available for use on the **INCLUDE** directive, see the online FAQ available via the program Help pulldown, or visit the RISCWatch Support Center web page.

## EXEC Definitions

An **EXEC** definition is used to associate a name with an ordered sequence of RISCWatch commands which will be executed whenever the register it is referenced from is read or written (See **REXEC** and **WEXEC** keywords listed under REG Definitions). Each **EXEC** definition must adhere to the following syntax:

```
DEFINE EXEC exec_name
```

```
    command
```

```
ENDEXEC
```

Where:

- *exec\_name* is the unique name being assigned to this **EXEC** and is the name used by **REG** definitions that reference it. This name can only contain alpha-numeric and underscore (\_) characters.
- *command* is a list of one or more RISCWatch commands which will be executed in the order in which they are listed. There can only be one command specified per line. Parameters may be passed into the **EXEC** by using the **PARMS** command (see “Command File Parameters” listed under “Command Files”).

The **EXEC** definition is basically a mini-command file (see “Command Files”) which can be executed whenever customized registers are read from or written to. Such functionality provides for a great deal of power but it must be used responsibly. Care must be taken to avoid “dangerous” operations. If any other processor resources are manipulated while the **EXEC** runs, it is usually wise to back up data values before the **EXEC** starts executing and then restore these values just before leaving the **EXEC**.

To facilitate the transfer of data values into and out of these customized register **EXEC** definitions, RISCWatch provides the **\$INPUT** and **\$RETURN** pseudo-variables. Whenever the **REG** command is used to write a value to a register which uses an **EXEC** definition, the commands inside the **EXEC** can reference **\$INPUT** to obtain the data value to be written. Likewise, while using the **REG** command to read an **EXEC** definition register, **\$RETURN** can be set to the value read so that it may be referenced once the **EXEC** returns.

Once an **EXEC** has been defined, it may be referenced using the **REXEC** and **WEXEC** keywords inside a **REG** definition. See the “PCF Example” section which follows for more details.

**Note:** A **DEFINE EXEC** may only be defined within a **DEFINE MACRO** definition.

## FIELD Definitions

A **FIELD** definition is used to associate a name with defined logical bit groupings so that it may be referenced later by another resource. Each **FIELD** definition must adhere to the following syntax:

```
DEFINE FIELD field_name {field_def}
```

Where:

- *field\_name* is the unique name being assigned to this **FIELD** and is the name used by other definitions to reference it. This name can only contain alpha-numeric and underscore (\_) characters.
- *field\_def* defines one or more bit fields. Each bit field must adhere to the following syntax:

```
name start length
```

Where:

- *name* is the unique name being assigned to this bit field and is the name used to reference it. This name can only contain alpha-numeric and underscore (\_) characters.
- *start* is the decimal physical starting bit (MSB=0) of this bit field
- *length* is the decimal length of this bit field in bits

## NAME Definitions

The **NAME** definition is used to define a name for a given chip. Many chips are known by both a marketing name (e.g. 740), and a "pet" name (e.g. Arthur).

This definition allows any number of "pet" names to be defined and is intended to give a unique code name for a given chip definition. Once defined, the "pet" name can be used when specifying the value of the **PROC** variable in the RISCWatch environment file.

A **NAME** definition must adhere to the following syntax:

```
DEFINE NAME user_name
```

Where:

- *user\_name* is the unique name being assigned. This name can only contain alpha-numeric and underscore (\_) characters.

## PVR Definitions

The **PVR** definition is used to define the value of the **PVR** for the given chip. When RISCWatch is started, the **PROC** environment variable, along with the PRD file, are used to define the resources of the target system. Once communication with the chip has been established, the PVR register will be read and compared to the **PVR** definition value. A mismatch of these values will result in a warning or error message to the user. Multiple **PVR** definitions are allowed to help account for changes in the PVR due to multiple revisions of the same ASIC.

A **PVR** definition must adhere to the following syntax:

```
DEFINE PVR pvr_value [and_mask] [rev_mask]
```

Where:

- *pvr\_value* is the processor's unique PVR value and is an implied hexadecimal number (e.g. **DEFINE PVR** 40ABCD04).
- *and\_mask* is an optional implied hexadecimal number which is used to mask certain bits of the specified PVR and the PVR read from the target before comparing them for a match (e.g. **DEFINE PVR** 00083311 FFFFF000).
- *rev\_mask* is an optional implied hexadecimal number which is used to mask certain bits of the PVR read from the target before comparing it to the revision(s) specified with the **REV** definition(s) (e.g. **DEFINE PVR** 00083311 FFFFF000 0000F00).

## REG Definitions

A **REG** definition is used to define an instance of a physical register. Each REG definition must adhere to the following syntax:

```
DEFINE REG reg_class reg_name  
#address|addr_reg addr_val data_reg  
bit_width R|W|RW [access] [VOLATILE] [BR] [display]  
[REXEC exec_name{parm_list}]  
[WEXEC exec_name{parm_list}]
```

Where:

- *reg\_class* is the class of register being defined; **DCR**, **IMR**, **SPR** or **MMIO**
- *reg\_name* is the unique name being assigned to this **REG** and is the name by which other definitions will reference it. This name can only contain alpha-numeric and underscore (\_) characters.

**Note:** registers being defined because they are part of an ASIC macro should use a common naming prefix so that they can be grouped together by RISCWatch.

The naming convention RISCWatch supports for ASIC register prefixes is to use a few letters as an acronym for the macro, followed by a number (usually starting with 0) and ending with an underscore (\_). This prefix is then added to the beginning of each register contained in that macro.

By following this convention, RISCWatch is able to detect the prefix and group the registers accordingly. This allows RISCWatch to create an ASICs entry in the Hardware | Register pulldown listing all the prefixes it detected during initialization. Selecting one of these prefixes then creates a Register window containing all registers with this prefix thereby grouping them together to represent the associated macro.

For example, if a PLB bus macro were being added, all registers defined for use with the PLB could be prefixed with 'PLB0\_'. RISCWatch could then detect this common prefix and the Hardware | Register | ASICs pulldown would list 'PLB0'. Selecting this entry would bring up a Register window with all the PLB0 prefixed registers in it.

- # is the hex number for a **DCR** or **SPR** register.

- *address* is the hex address for an **MMIO** register. Only physical addresses should be used. The address can be more than 32 bits long. Effective addresses should NOT be used.
- *addr\_reg* is the name of the address register for an Indirectly Mapped Register (IMR).
- *addr\_val* is the value to be written to *addr\_reg* to access an IMR register's contents.
- *data\_reg* is the name of the data register for an IMR register.
- *bit\_width* is the decimal width of this register in bits.
- **R|W|RW** defines the register's access; read-only (**R**), write-only (**W**) or read-write (**RW**).
- *access* is an optional parameter for **MMIO** registers which is used on JTAG ethernet RISCWatch targets. It is a decimal number which indicates the access size, in bits, RISCWatch must use when reading or writing this memory location. The access size should be 8, 16 or 32, with each multiple identifying a unique PowerPC load/store instruction to use. For example, an access size of "16" instructs RISCWatch to read the register by executing the "load halfword" PowerPC instruction. Specifying an access size will override any access size settings made with the **memacc** command. If no access size is specified, RISCWatch will use the access size defined for the memory region. See **memacc** on page 5-75 for information about how to set up a unique memory region access size.
- **VOLATILE** is an optional keyword which indicates this register will change its value after a read operation is performed. It must be entered in uppercase. RISCWatch users must issue an explicit read to display the contents of a volatile register. This keeps RISCWatch from asynchronously reading the register thus avoiding changes to it. Having the auto-update mode enabled on a window containing these registers will not cause them to be read during the update.
- **BR** is an optional keyword which indicates that values read from and written to this register will be byte-reversed. On a read, after the register is read it is byte-reversed before being displayed. On a write, the specified value will be byte-reversed before being written to the target.
- *display* is an optional keyword which indicates the default display format for floating point registers. It may be either **HEX** or **SCI**.
- **REXEC** is an optional keyword which indicates that a previously defined **EXEC** entry is to be run to read the value of this register.
- **WEXEC** is an optional keyword which indicates that a previously defined **EXEC** entry is to be run to write a value to this register.
- *exec\_name* is the name of a previously defined **EXEC** entry which is to be run.
- *parm\_list* is an optional list of one or more parameters whose values are to be passed to the specified **EXEC** entry.

**Note:** If a resource is referenced before it has been defined, an error message will be generated and the compile will be aborted.

## REGALIAS Definitions

A **REGALIAS** definition is used to refer to a previously defined register by another name.

A **REGALIAS** definition must adhere to the following syntax:

```
DEFINE REGALIAS new_reg = exist_reg
```

Where:

- *new\_reg* is the new register alias name.
- *exist\_reg* is the name of the previously defined register.

**Note:** If a resource is referenced before it has been defined, an error message will be generated and the compile will be aborted.

## REGFLD Definitions

A **REGFLD** definition is used to define a register field. Each **REGFLD** definition must adhere to the following syntax:

```
DEFINE REGFLD register field_name{field_def}
```

Where:

- *register* is the name of a previously defined register.
- *field\_name* is the name of a previously defined **FIELD** entry.
- *field\_def* defines one or more bit fields. Each bit field must adhere to the following syntax:

```
name start length
```

Where:

- *name* is the unique name being assigned to this bit field and is the name by which it will be referenced. This name can only contain alpha-numeric and underscore ( `_` ) characters and should not exceed 6 characters.
- *start* is the decimal physical starting bit (MSB=0) of this bit field.
- *length* is the decimal length of this bit field in bits.

**Note:** If a resource is referenced before it has been defined, an error message will be generated and the compile will be aborted.

## REV Definitions

The **REV** definition is used to indicate to RISCWatch which Processor Probe firmware files are to be used with a specific definition of a chip/core. For chips/cores with multiple revisions, multiple **REV** definitions may be used to define a unique set of files for each, if necessary.

A **REV** definition must adhere to the following syntax:

```
DEFINE REV n driver generics
```

Where:

- *n* is the processor revision number. 0 should be used for all PowerPC 400 family cores/chips.
- *driver* is the name of the Processor Probe driver filename to be used for this revision. Such names usually start with "E34" and have a ".X" extension.
- *generics* is the name of the Processor Probe generics filename to be used for this revision. Such names usually start with "E34" and have a ".X" extension.

## PCF Compiling

Once a PCF file has been created, it must be compiled by **rwpcfc** to verify that the file syntax is correct and that all definitions are complete. If so, the file is turned into a format which can be loaded by RISCWatch at run time.

The compiler program, **rwpcfc**, was installed in the same directory as the main RISCWatch executable. This is also where the default RISCWatch PRD file is located. This PRD file contains a number of predefined processors, cores and other resources that may need to be referenced.

To run the compiler, use the following syntax:

```
rwpcfc [-log] [-refer] filename.pcf
```

Where:

- **-log** is an optional flag. If present, all messages which are usually printed to the screen are directed to the rwpcfc log file (rwpcfc.log).
- **-refer** is an optional flag. It must be specified when compiling a file which will be REFERred by another file (see the **REFER** definition above). In other words, if file A.pcf has a "REFER B.prd" line in it, file B.pcf must have been compiled with the -refer flag, since it is being REFERred to. If only one PCF file is being defined and compiled, this option will not be used.
- *filename.pcf* is the name of the PCF file that is to be compiled. At this time, the compiler is a DOS-based tool so Windows 95/98 long filenames are not supported.

If any errors are detected, an appropriate error message will be printed to the screen or log file. These should be used to correct the source of the error message(s) prior to attempting a recompile.

Successful compilation will result in a valid PRD file whose name can then be specified in the RISCWatch environment file with the **PRD\_FILE** variable. When RISCWatch is started, this variable will be used to load the appropriate resource information uniquely defined by the processor/core designated by the **PROC** environment variable value.

## PCF Example

The following examples are provided to acquaint users with some of the more common coding of the PCF file.



In this example, a few simple memory mapped registers are added to an existing chip (a function previously accomplished with the Register Definition File). In addition, register fields have been defined for MMIO\_2 and MMIO\_3:

```

APPEND CHIP MY_403GB
    INCLUDE CHIP PPC_403GB

    DEFINE REG MMIO MMIO_1 0xA000 32 RW
    DEFINE REG MMIO MMIO_2 0xA004 32 RW 8
    DEFINE REG MMIO MMIO_3 0xA008 32 RW VOLATILE
    DEFINE REG MMIO MMIO_4 0xA00C 32 RW 8 VOLATILE

    DEFINE REGFLD MMIO_2 { RES    0 16
                            AVR_L 16 8
                            AVR_R 24 8 }
    DEFINE REGFLD MMIO_3 { RES    0 16
                            AVR_L 16 8
                            AVR_R 24 8 }

END

```

Once compiled, this would result in a custom PRD file which would be set using the **PRD\_FILE** environment variable while the defined chip, MY\_403GB, would be set using the **PROC** environment variable.

In the next example, a **MACRO** will be used to accomplish the same results as the previous example. In addition, macro IMR\_REGS is defined to demonstrate how to define IMR registers. Note the use of **DEFINE FIELD** for registers that share the same register field designations:

```

DEFINE MACRO MMIO_REGS
    DEFINE REG MMIO MMIO_1 0xA000 32 RW
    DEFINE REG MMIO MMIO_2 0xA004 32 RW 8
    DEFINE REG MMIO MMIO_3 0xA008 32 RW VOLATILE
    DEFINE REG MMIO MMIO_4 0xA00C 32 RW 8 VOLATILE

    DEFINE FIELD MMIO_FIELD1 { RES    0 16
                                AVR_L 16 8
                                AVR_R 24 8 }
    DEFINE REGFLD MMIO_2 MMIO_FIELD1
    DEFINE REGFLD MMIO_3 MMIO_FIELD1

END

DEFINE MACRO IMR_REGS
    DEFINE REG DCR T0_ADDR 0x0180 32 RW
    DEFINE REG DCR T0_DATA 0x0181 32 RW
    DEFINE REG IMR T0_REG1 T0_ADDR 0x00000001 T0_DATA 32 RW

```

```

        DEFINE REG IMR T0_REG2 T0_ADDR 0x00000002 T0_DATA 32 RW
        DEFINE REG IMR T0_REG3 T0_ADDR 0x00000003 T0_DATA 32 RW
    END

```

```

APPEND CHIP MY_403GB
    INCLUDE CHIP PPC_403GB
    INCLUDE MACRO MMIO_REGS
    INCLUDE MACRO IMR_REGS
END

```

In the next example, a **MACRO** will be used to define a base set of registers which can be located anywhere within the chip's address space. Using the offset feature of the **INCLUDE MACRO** statement, this set of registers can be used in two different processors in two different memory locations:

```

DEFINE MACRO BASE_REGS
    DEFINE REG MMIO BASE_1 0x0 32 RW
    DEFINE REG MMIO BASE_2 0x4 32 RW
    DEFINE REG MMIO BASE_3 0x8 32 RW
    DEFINE REG MMIO BASE_4 0xC 32 RW
END

```

```

APPEND CHIP MY_A1
    INCLUDE CHIP PPC_401A1
    INCLUDE MACRO BASE_REGS 0xA0000000
END

```

```

APPEND CHIP MY_B2
    INCLUDE CHIP PPC_401B2
    INCLUDE MACRO BASE_REGS 0xFFFF0000
END

```

In the next example, a **MACRO** will be used to define a customized register which uses two **DEFINE EXEC** definitions to create read and write "routines" which can then be used to manipulate the contents of this specialized register:

```

DEFINE MACRO CUST_REG
    DEFINE EXEC cust_read
        PARMS {stuff_instr}
        STUFF stuff_instr
        SET $RETURN = 0x00E80024
    ENDEXEC

    DEFINE EXEC cust_write
        PARMS {stuff_instr}
        WRITE 0x00E80024 $INPUT
    ENDEXEC
END

```

```

        STUFF stuff_instr
    ENDEXEC

    DEFINE REG DCR APU1 0xFFFF 64 RW
        REXEC cust_read{0x90A10024}
        WEXEC cust_write{0x80A10024}
    END

```

## MEMACC Command

When a memory read or write operation is requested, RISCWatch must first determine if the request is valid and then determine the proper way to proceed with the request. Performing a read to an invalid memory address, or issuing a store word instruction to a memory region configured for half word access, could result in unwanted machine checks, data corruption, or system hangs.

When RISCWatch is first started, the target processor name (designated with the **PROC** environment variable) is used to define the type of memory address validation to perform. RISCWatch internal address validation can be summarized as follows:

- On 403GA and GB processor targets, RISCWatch will read the bank registers to determine valid address regions and read/write access restrictions. Four byte word access is assumed valid for any read or write operation. Access type defaults to instruction and data.
- On 403GC and GCX processor targets, RISCWatch will default to the operations defined for the 403GA if translation is off. If translation is on, the TLB is read to determine valid address regions and access restrictions. Four byte word access is assumed valid for all read/write operations. Access type is determined by the instruction and data translation bits defined in the machine state register (MSR). Since OS Open performs its own address validation when translation is on, RISCWatch assumes all addresses are valid for OS Open targets.
- On 405GP processor targets, RISCWatch will read EBC and SDRAM registers to determine address regions and read/write access restrictions when address translation is off. If translation is on, the TLB is read to determine valid address regions and access restrictions. Four byte word access is assumed valid for all read/write operations. Access type is determined by the instruction and data translation bits defined in the machine state register (MSR). Since OS Open performs its own address validation when translation is on, RISCWatch assumes all addresses are valid for OS Open targets.
- On 6xx and 7xx targets, all addresses are assumed valid for both read and write access. Access size defaults to 8 bytes. Access type defaults to instruction and data.
- On Core+ASIC processors, all addresses are assumed valid for both read and write accesses when address translation is off. Access size defaults to 4 bytes. Access

type defaults to instruction and data. If the MMU is enabled, addresses are checked against the current TLB entries.

**Note:** For ROM Monitor and OS Open targets, access size is governed by the monitor code running on the target processor.

Obviously, the internal address validation may not be adequate for all users. In an effort to provide additional memory access protection, RISCWatch provides the **memacc** command which allows a user to define the unique memory configuration associated with a processor target.

## Use of MEMACC ADD

Users can override any RISCWatch internal address validation checks by executing the **memacc add** command. The command syntax is defined as follows:

```
memacc add beg_addr end_addr [access [size [type]]]
```

Where:

- **add** is a keyword on the **memacc** command indicating that a new entry is to be added to the list of user defined address regions.
- *beg\_addr* indicates the beginning address of target memory being defined with this command. The address can be designated in hex (leading "0x" or "0X"), octal (leading 0), or decimal.
- *end\_addr* indicates the last address of target memory being defined with the command. The address can be designated in hex (leading "0x" or "0X"), octal (leading 0), or decimal
- *access* is an optional parameter which indicates the access restrictions of the specified region. Access can be "**RO**" (read only), "**WO**" (write only), "**NA**" (no access), or "**RW**" (read/write). If not specified, access defaults to "**RW**".
- *size* is an optional parameter which is used on JTAG ethernet RISCWatch targets. It is a decimal number which indicates the maximum access byte size RISCWatch can use when reading or writing the specified memory region. *Size* can be 0, 1, 2, 4, or 8, with each multiple identifying a unique PowerPC load/store instruction to use. For example, an access size of "4" instructs RISCWatch to read memory by executing the "load word" PowerPC instruction. If no access size is specified, the default size defined for the target processor will be used. A size of "0" also indicates that the default size, used for RISCWatch internal address checking, should be used.
- *type* is an optional parameter indicating the valid type of access for the specified memory region. Valid types are **IMEM** (instruction memory), **DMEM** (data memory), or **MEM** (instruction and data memory). If not specified on the command, the type defaults to **MEM**. Since users are not aware of the internal access types used for the various RISCWatch screens, the default setting of **MEM** should normally be used.

**Note:** Additional variations of the **memacc** command are possible but not pertinent to this discussion. See **memacc** on page 5-75 for additional information.

Examples:

```
memacc add 0x40000000 0x40000009 RW 1
memacc add 0xFFFF0000 0xFFFFFFFF RO 4
memacc add 0x4000000a 0x4FFFFFFF NA
```

Each “**memacc add**” command adds an entry to a list of user defined address definitions. When RISCWatch performs a memory operation, address validation proceeds as follows:

1. Check the user defined address regions first to determine if the address can be read/written. Entries are searched LIFO, meaning the last “**memacc add**” command entered is checked before any previous entries.
2. Perform the internal address checking defined for the target processor for any portion of the address range not included in the user defined entries.

### Practical Application Example

The following example is provided to demonstrate how the **memacc** command can be used to customize RISCWatch memory access.

Example:

1. RISCWatch is running on a customized chip that is built around the PowerPC 401 Core.
2. There is a two byte region of memory, starting at address 0x50004444, which can be accessed with the load/store halfword PowerPC instructions.
3. All other addresses, starting at 0x50000000 and ending at 0x5FFFFFFF, are considered invalid. A store or load to any one of these addresses will result in a machine check.
4. Memory mapped IO addresses 0x40000000 to 0x40000009 are one byte read/write access locations used for serial port operations. Addresses 0x4000000A to 0x4FFFFFFF are invalid
5. All other addresses are valid read/write regions which can be accessed via the PowerPC load/store word instructions.

Based on the target processor, RISCWatch is set up to perform internal address checking:

- Every address, from 0x00000000 to 0xFFFFFFFF, is valid for both read and write operations.
- Access size defaults to 4. This means PowerPC load/store word instructions can be used to access memory.

**Note:** The user always has the option of not using any of the RISCWatch internal address checking. This is accomplished by completely defining the entire address space with “**memacc add**” commands. For example, “**memacc add 0 0xFFFFFFFF**” defines the entire address space as a read/write region of 4 byte access size. With all possible addresses defined, there is no need for RISCWatch to perform any internal address checking.

Using the default internal checking as a base, “**memacc add**” commands must be issued to indicate all address regions that do not allow 4 byte read/write access. These would be all the addresses from 0x40000000 to 0x5FFFFFFF.

The following commands should be added to the RISCWatch start-up command file (designated with the **STARTUP\_FILE** environment variable):

- 1.memacc add 0x50000000 0x5FFFFFFF NA
- 2.memacc add 0x50004444 0x50004445 RW 2
- 3.memacc add 0x40000000 0x40000009 RW 1
- 4.memacc add 0x4000000a 0x4FFFFFFF NA

**Note:** Notice the addresses overlap between the first and second commands. Since the second command is issued after the first, RISCWatch will use the restrictions of the second command, since LIFO search order is used.

RISCWatch is now customized to the unique memory constraints presented in this example. Any attempt to read/write memory addresses 0x0400000a to 0x50004444, or 0x50004446 to 0x5FFFFFFF, will be flagged as an error and the memory access will not be attempted. Any attempt to read/write memory addresses 0x40000000 to 0x40000009 will be performed using PowerPC load/store byte instructions. PowerPC load/store halfword instructions will be used to access the halfword that exists at address 0x50004444. All other address regions will be considered valid read/write requests that can be performed using PowerPC load/store word instructions.

## Window Descriptor File

Once the unique memory access restrictions and register definitions are complete (by using the **memacc** command and creating a Processor Configuration File), RISCWatch commands can be issued to read or alter resources which are accessible from the core processor. In addition, users can create their own customized windows which display Core+ASIC resources. Please see “User-Defined Windows” on page 3-118 for details about customized RISCWatch windows.

---

## RISCWatch Programming Interface (RWPI)

The RISCWatch Programming Interface (RWPI) is designed to allow users to interact with RISCWatch using their own C or PERL programs. This level of functionality affords users who are comfortable with their native programming language to use the features of that language instead of being forced to use RISCWatch command files.

Users can write their own C/C++ or PERL programs which are then linked into an appropriate library for their platform. This library provides a simple yet effective means of communicating with the RISCWatch program via TCP/IP sockets.

Since the RWPI uses TCP/IP to communicate with RISCWatch, it goes much further than simply allowing control via a user program. The ability to use TCP/IP allows the user program and RISCWatch programs to exist on two different computers separated by any distance as long as they are networked in such a way that a communications path exists between them. It is the user's responsibility to see that the proper network software and interface cards are properly installed and running to support TCP/IP on the desired computers.

RISCWatch can now be started in a special RWPI mode in which it waits for a connection from a "user client" program. Once connected, the user client sends commands to RISCWatch which are executed, and results and status returned to the user client for any further processing.

The RWPI package consists of basic API documentation (see **readme.txt**), C/C++ and PERL libraries, and a sample test program, all of which were installed in the **rwpi** subdirectory created off of the installation directory.

The sample user client program, **uc** or **uc.exe**, has already been built for you from the included user code sample, **uc.c**, and RWPI client object module, **rwpic.obj** (compiled on Windows using Microsoft Visual C++) or **rwpic.o** (on AIX using xLC, Sun using SunSoft SPARCCompiler C++ or Linux using gcc). To test this sample program, start RISCWatch on one computer using the **-pi** command line flag. This will start up RISCWatch in RWPI mode where it waits for a connection from a client program. The client program can then be started with the IP address or hostname (if you have a DNS) of the computer running RISCWatch specified on the command line.

If all goes well, the user client program should connect to the RISCWatch session. The sample user client then tests the interface connection and presents a command line interface to the user. Almost any RISCWatch command can be entered to be sent to the remote RISCWatch session where it will be executed and the results sent back to the user client.

While RISCWatch is running in RWPI mode, it will not accept input from its command line or normal menu system to prevent local users from interfering with command data being sent across the RWPI connection. RISCWatch simply waits for a connection from the user client and echoes all commands sent to it for execution. For obvious reasons, RISCWatch only accepts a connection from one client program at a time. To terminate RISCWatch, simply select File then Quit from the menu bar.

To begin writing your own client program, study the supplied user client C source code along with the API documentation provided in **readme.txt**. C syntax and comments were used so that both C and C++ compilers could be used. Make copies of **uc.c** and the provided **makefile** to begin writing and building your own client program. If you are into PERL, you can use the **uc.pl** and **rwpic.pl** files to do your client development work.

---

## Multi-Processor Resources

In an effort to support multi-core and multi-processor PowerPC systems, RISCWatch allows a user to create Multi-Core PCF and Multi-Processor Support (MPS) files. These files, created prior to starting RISCWatch, contain information which allows a single RISCWatch session to communicate with each core/processor.

Currently, certain restrictions apply when running RISCWatch in a multi-core/processor environment. See the RISCWatch README file for details.

The following sections provide additional details needed to run RISCWatch in a multi-core/processor environment.

### PCF File Syntax For Multi-Core Processors

Processor Configuration Files (PCFs) are used to define processors. They define whether a processor has one or more embedded CPU cores. They also define the ASIC content of the processor. The term Core Select (CS) Bits is introduced. The CS bits can have a length of N bits and the patterns that select each core can change from one design to another. The designer of the processor selects the value of N and the patterns that select the individual cores.

The RISCWatch developers will create PCFs and set the CS bits for all supported application specific processors. The RISCWatch product will include a Processor Resource Definition file (PRD) that reflects these settings. For customer specific processors, it is the responsibility of the RISCWatch user to determine the CS bits and to create the correct PCF/PRD for the processor being debugged.

Here is an example of a PCF that defines a processor called 440405. The processor includes both a 405 and a 440 core. The CS bits are 4 bits long and the binary pattern 1010 selects the 440 core while the binary pattern 1001 selects the 405 core.

```
# Define a new chip with both a 440 and 405 core
# The 440 core is called CORE1
# The 405 core is called CORE2
DEFINE CHIP 440405
    # Define the two cores, give them names and define
    # the Core Select (CS) bits in binary
    INCLUDE CORE PPC_440A4_CORE CORE1 b'1010
    INCLUDE CORE PPC_405B3_CORE CORE2 b'1001

    # Define what else is in this chip
    INCLUDE MACRO PPC_ASIC_UART_1 0xEF600300
END
```



## MPS File Syntax

Multi Processor Support Files (MPS) are used to define boards. By definition, a board is a unit that can execute PowerPC instructions. By definition, a JTAG chain is a list of devices on a board that have common TCK and TMS signals and have TDO of device number N connected to TDI of device number N+1. MPS files list, in order, all devices that exist on the JTAG chain of a board. They define whether a board has one or more PowerPC processors on the JTAG chain. They also define if other devices other than PowerPC processors exist on the same JTAG chain. MPS files also support the definition of more than one board.

The MPS file is an ASCII file that can be created with any text editor. The file is identified to RISCWatch via the **MPS\_FILE** environment variable, and must have a file extension of “.mps”.

The general file syntax rules are as follows:

1. The “#” character denotes the start of a comment. All text following the “#” character on a given line will be ignored.
2. Blank lines are allowed and will be ignored.
3. Any error detected during the processing of the MPS file will surface an error message in the RISCWatch log file and execution will terminate.

## Board Definitions

Board definitions span multiple lines of the file and are used to identify the type of PowerPC chip on a board and the communication protocol RISCWatch should use. Each board definition must adhere to the following syntax:

```
BOARD brd_name target_name target_type [target_port] [BYPASS]  
    CHIP proc_id chip_name ir [BYPASS]  
    DEVICE dev_name ir  
    .....
```

```
ENDBOARD
```

Where:

- **BOARD** indicates the start of a new board definition and must appear in uppercase.
- *brd\_name* indicates a user defined name for the board. The name must be enclosed in double quotes. Names exceeding 24 characters will be truncated.
- *target\_name* indicates a valid target name found in the TCP/IP services file or a TCP/IP dotted address (e.g. 7.1.1.100). This overrides any **TARGET\_NAME** designation made in the **rwppc.env** file.
- *target\_type* indicates the type of RISCWatch target to use. Valid target types are those defined for the **TARGET\_TYPE** environment variable. See “Environment Resources” on page 3-5 for valid target types to use.
- *target\_port* is an optional parameter which specifies a unique port for the target being connected to.

- **BYPASS** is an optional keyword which indicates that the **BOARD** or **CHIP** are to be considered as bypassed entities on the JTAG scan chain. RISCWatch will not allow debug of a **BYPASS**ed **BOARD** or **CHIP**.
- **CHIP** is a keyword indicating chip information will follow. It must be designated in uppercase. At least one chip entry must be designated for each board defined or a syntax error will occur.
- **DEVICE** is a keyword indicating the existence of a non PowerPC device on the JTAG chain. These **DEVICES** are **BYPASS**ed by RISCWatch.
- *proc\_id* indicates a valid processor target name. Valid processor names are those defined for the **PROC** environment variable. See "Environment Resources" on page 3-5 for valid processor names to use.
- *chip\_name* indicates a user defined name for the chip. Names exceeding 24 characters will be truncated.
- *dev\_name* indicates a user defined name for the device. Names exceeding 24 characters will be truncated.
- *ir* is a decimal number indicating the bit size of the JTAG instruction register.

Here is an example of a Multi Processor Support (MPS) file. It shows how RISCWatch can connect to multiple RISCWatch Processor Probes at the same time and how each probe can be used to scan a number of chips/devices connected in series on the same JTAG chain.

```
# Define the board connected to the probe with
# IP address 9.44.18.46
BOARD Board1 9.44.18.46 jtag_eth
  # Define all the devices on the JTAG chain
  # The first processor is called CHIP1
  # The second processor is called CHIP2
  CHIP 440405 CHIP1 8
  CHIP 440405 CHIP2 8
  DEVICE Bypass1 7
ENDBOARD

# Define the board connected to the probe with
# IP address 9.44.18.47
BOARD Board2 9.44.18.47 jtag_eth
  # Define all the devices on the JTAG chain
  # The first processor is called CHIP3
  # The second processor is called CHIP4
  CHIP 405GP CHIP3 7
  CHIP 440GP CHIP4 7
ENDBOARD
```

When an MPS file is designated, the **TARGET\_NAME** and **TARGET\_TYPE** environment variable designations (specified in the **rwppc.env** file) will be ignored.

## MPS Debugging

RISCWatch v4.x customers are familiar with the typical uni processor/core support where only one board with one processor on the JTAG chain with one embedded core is supported. Typically, the customer would select the processor being debugged by setting the PROC environment variable or by using the -proc command line parameter.

The same applies for RISCWatch v5.0 and later. However, if RISCWatch v5.0 detects that the selected processor has more than one embedded core, it will automatically start **MPS Mode**. By definition, **MPS Mode** is a mode where the customer is given a window, called the MPS window, which lists all the PowerPC processors/cores that the customer can debug. The customer can select the context by using the MPS window or by using the **mpsset** 'context' command.

Here is an example of the MPS window if the PROC is set to the 440405 processor defined in the example PCF. Note that the context names for this example are CORE1 and CORE2.

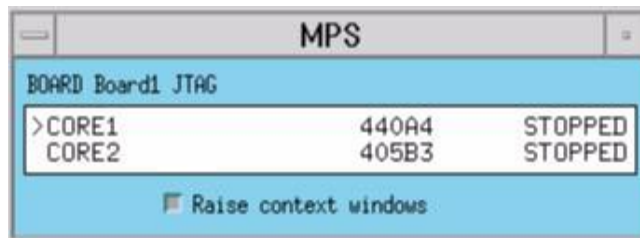


Figure 3-1. Sample MPS Window

RISCWatch v5.0 also introduces **MPS Unimode**. By definition, **MPS Unimode** is a mode where RISCWatch detects that the selected processor has more than one embedded core but the customer has requested that only one of these cores be debugged with the current RISCWatch session. Referring to the example PCF, the customer can select MPS unimode by setting the PROC to 440405:CORE1 to debug the 440 core or by setting the PROC to 440405:CORE2 to debug the 405 core. In **MPS Unimode**, the customer does not have access to an MPS window and cannot change the context using the **mpsset** command. This means that RISCWatch, in this mode, will look, feel and behave just like the case where the processor only has one embedded core.

Multiple RISCWatch sessions in **MPS Unimode** can be connected to the same probe at the same time. Referring to the example PCF, the customer can start one RISCWatch session in **MPS Unimode** by setting the PROC to 440405:Core1 then the customer can start another RISCWatch session in **MPS Unimode** by setting the PROC to 440405:Core2. The multiple RISCWatch session in **MPS Unimode** approach may be useful when running long automated tests using command files.

If an MPS file is specified, either with the **MPS\_FILE** environment variable or the **-mps** command line option and the specified file is found, it is read in and used to put RISCWatch in **MPS Mode**. The file is located using the following rules:

- If the file name is qualified (directory path indicated), the file search is performed using the specified directory only.
- If the name is not qualified, the file search is performed using the directory paths designated with the RISCWatch **SEARCH\_PATH** environment variable. If not found, the current directory is searched.

Once in **MPS Mode**, RISCWatch has the ability to switch between the chips that were specified in the MPS file. This switching ability allows for the resources of a particular chip to be specified and debugged as though it were a single chip system. The following sections contain more information on how individual chips are identified and debugged using the RISCWatch interface.

Here is an example of the MPS window if the example MPS file is specified. Note how RISCWatch concatenates the chip and core names to form the context names. For this example, the context names are CHIP1CORE1, CHIP1CORE2, CHIP2CORE1, CHIP2CORE2, CHIP3 and CHIP4.

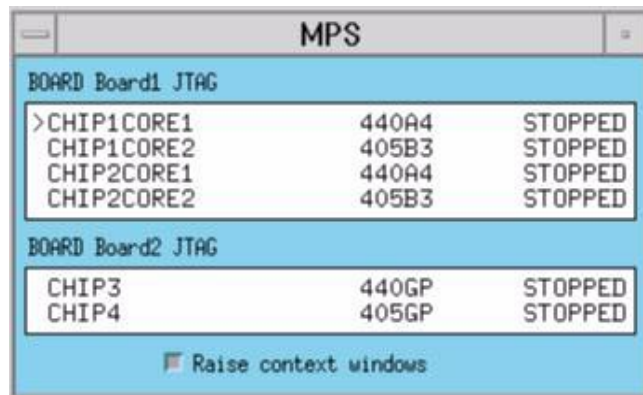


Figure 3-2. Sample MPS Window

If an MPS file is specified, the **BYPASS** keyword can be used to bypass entire boards or to bypass certain processors/chips on the JTAG chain of a board. If all boards and processors in an MPS file are bypassed except for one processor on one board and the processor has one core, RISCWatch will start in **MPS Unimode**. The multiple RISCWatch session in **MPS Unimode** approach can be used even with an MPS file. The **BYPASS** keyword can be moved around in the MPS file to select a certain processor on a certain board before starting RISCWatch.

In this example, all processors/boards are bypassed except CHIP3 on Board2. RISCWatch will start in **MPS Unimode** and debug CHIP3.

```
# Define the board connected to the probe with
# IP address 9.44.18.46
BOARD Board1 9.44.18.46 jtag_eth BYPASS
# Define all the devices on the JTAG chain
# The first processor is called CHIP1
# The second processor is called CHIP2
CHIP 440405 CHIP1 8
CHIP 440405 CHIP2 8
DEVICE Bypass1 7
ENDBOARD

# Define the board connected to the probe with
# IP address 9.44.18.47
BOARD Board2 9.44.18.47 jtag_eth
# Define all the devices on the JTAG chain
# The first processor is called CHIP3
# The second processor is called CHIP4
CHIP 405GP CHIP3 7
CHIP 440GP CHIP4 7 BYPASS
ENDBOARD
```

In this example, all processors/boards are bypassed except CHIP4 on Board2. RISCWatch will start in **MPS Unimode** and debug CHIP4.

```
# Define the board connected to the probe with
# IP address 9.44.18.46
BOARD Board1 9.44.18.46 jtag_eth BYPASS
# Define all the devices on the JTAG chain
# The first processor is called CHIP1
# The second processor is called CHIP2
CHIP 440405 CHIP1 8
CHIP 440405 CHIP2 8
DEVICE Bypass1 7
ENDBOARD

# Define the board connected to the probe with
# IP address 9.44.18.47
BOARD Board2 9.44.18.47 jtag_eth
# Define all the devices on the JTAG chain
# The first processor is called CHIP3
# The second processor is called CHIP4
CHIP 405GP CHIP3 7 BYPASS
CHIP 440GP CHIP4 7
ENDBOARD
```

## MPS Context

At any given moment, RISCWatch can only communicate with a single chip. In an MPS environment, it is necessary to debug the resources of several chips which may reside on physically separate boards. To communicate with each individual chip, there must be a way for RISCWatch to switch its communications path to “talk” to a particular chip.

The resources specified in the MPS file define the communications paths RISCWatch will use to communicate with all the chips in the MPS system. The target names and types that were specified are used to select the proper physical communications path. These resources are managed internally by RISCWatch and are transparent to the end user.

The chip names specified in the MPS file are used to uniquely identify a particular chip on a particular board. These names serve as a way for the user to communicate to RISCWatch which chip’s resources are to be debugged. To switch the communications path to talk to a particular chip, the **mpsset** command is used.

The argument supplied to the **mpsset** command is simply the chip name specified in the MPS file. RISCWatch is then able to use this name to look up the communications path to the specified chip. RISCWatch configures its communications so that it is able to debug the resources of the specified chip. Using the **mpsset** command is referred to as setting the MPS context. It is within this context that a particular chip’s resources will be accessed.

The RISCWatch Main window will be the primary means of identifying what MPS context is currently set. The status bar, located at the bottom of the Main window, will display the name of the chip which is currently being debugged.

The MPS context is said to have been set to this chip that is displayed in the status bar. Any command issued from the command line on the Main window will execute in this context. If a **read register** command is executed, the specified register will be read from the current MPS context (the chip displayed in the status bar). If a register from a different chip is to be read, the **mpsset** command must be issued to switch the MPS context to that chip and then the **read register** command can be used.

When running a command file, the commands are executed under the current MPS context. To switch the context during execution of the command file, simply issue **mpsset** as necessary.

Some examples are:

```
mpsset CHIP1CORE1
mpsset CHIP1CORE2
mpsset CHIP2CORE1
mpsset CHIP2CORE2
mpsset CHIP3
mpsset CHIP4
```

## MPS Windows

In **MPS Mode**, windows are classified as being one of three types:

- MPS dynamic
- MPS specific
- MPS neutral

The RISCWatch Main window is the only instance of an MPS dynamic window. This window can have its MPS context switched by using the **mpsset** command. Its current MPS context is displayed in the status bar.

MPS specific windows are assigned the current MPS context upon creation and its MPS context can not be changed thereafter. Any processor accesses or commands issued from such a window will only pertain to its MPS context and no other. The MPS context for each of these windows will be displayed in the window's title bar. What will be displayed is the chip name assigned to that MPS context in the MPS file. Most windows in RISCWatch are MPS specific.

MPS neutral windows are assigned no MPS context because they do not access processor resources or they simply use the current context (as displayed in the Main window status bar). Examples of MPS neutral windows include Calculator, Command File, Log, Memory Load, Memory Save, User-Created Variable, MPS, Output and Window List.

The MPS window is only available in **MPS Mode** and is used as a shortcut to set the MPS context as well as providing status for each chip. Displayed in this window are all the boards and chips defined in the MPS file. The mouse is used to select a chip which in turn issues the appropriate **mpsset** command to switch the MPS context to this chip.

---

## Invoking the Debugger

Before RISCWatch is started for the first time, a few items need to be taken care of. First, make sure that the RISCWatch executable is in a directory that can be located by the PATH environment variable. Prior to starting RISCWatch, change the environment resource file **rwppc.env** to match the specific target configuration you plan to use. Below is the complete list of the different target types available and a brief description of some of the key steps that need to be taken. See "Environment Resources" on page 3-5 for additional resource setup information.

- JTAG Ethernet Target (RISCWatch Processor Probe Connection):

Verify that the Processor Probe hardware was installed as defined in the *RISCWatch Debugger Installation Guide*.

Verify that the **rwppc.env** file designates 'TARGET\_TYPE = jtag\_eth', as discussed in "Environment Resources" on page 3-5.

Verify that the **rwppc.env** file designates 'TARGET\_NAME = x...x', where 'x...x' is replaced by the TCP/IP name or address chosen for the processor probe during installation.

Verify proper installation and network recognition of the RISCWatch Processor Probe. This can be accomplished by 'pinging' the TARGET\_NAME from the host system (ex. 'ping 7.1.1.100').

- ROM Monitor Target:

Verify that the host is configured correctly for Ethernet setup, as discussed in the configuration section of the evaluation board kit user's documentation. These instructions describe specific host configuration steps and other setup (editing /etc/services files) required by RISCWatch for successful host/target communication.

Verify that the target ROM monitor is set up in debug mode, as discussed in the IBM PowerPC evaluation board kit user's documentation. This typically involves starting a terminal emulation screen, resetting the board, enabling an ethernet or serial port boot source, and selecting an option to enable ROM monitor debug.

Verify that the **rwppc.env** file designates 'TARGET\_TYPE = rom\_mon' as discussed in "Environment Resources" on page 3-5.

Verify that the **rwppc.env** file designates 'TARGET\_NAME = x...x', where 'x...x' is replaced by the TCP/IP name or address chosen for the ROM monitor. See the IBM PowerPC evaluation board kit user's documentation for more information about setting up a local address for the ROM monitor.

From the host system, ping the TARGET\_NAME to verify proper network and ROM monitor initialization (ex 'ping 7.1.1.4'). Note that the ROM monitor must be in debug mode when the ping command is issued.

- OS Open Target

Verify that OS Open is running on the target system. RISCWatch cannot communicate with OS Open programs that have not called **rsld\_start()**. Loading an OS Open image can be performed using one of the other RISCWatch targets (see "Loading Boot and Boot Image Files" on page 3-47) or by using ROM monitor **bootp** support. See the IBM PowerPC evaluation board kit user's documentation and the *OS Open User's Guide*, listed in "Related IBM Publications" on page xxiv of this user's guide.

Verify that the **rwppc.env** file designates 'TARGET\_TYPE = osopen' as discussed in "Environment Resources" on page 3-5.

Verify that the **rwppc.env** file designates 'TARGET\_NAME = x...x', where 'x...x' is replaced by the TCP/IP address chosen for the OS Open image.

From the host system, ping the TARGET\_NAME to verify proper network and OS Open initialization (e.g. 'ping 7.1.1.4').



Under normal circumstances, RISCWatch will be started as described in “Starting the Debugger” on page 2-1. RISCWatch does have a few command line parameters which may or may not have to be specified depending on how you run RISCWatch. Here is a list of the command line parameters that RISCWatch understands:

- echo                   used to echo each command file line as it is executed; use this to debug command file execution. This option is only available on a non-Windows platform.
- help or ?             used to display the help information for RISCWatch which lists all of the available command line options
- proc*NAME*           overrides PROC setting specified in the environment resources file (**rwppc.env**). This allows multiple icons on PC hosts to be defined for different processors while using only one environment file.  
See the README file for a list of currently supported processor names.
- rev*REV*             Overrides REV setting specified in the environment resources file. This distinguishes between different 6xx/7xx processor revision levels when connected via the RISCWatch Processor Probe. The -rev flag must be used when debugging a 6xx/7xx processor in which RISCWatch supports more than one revision level. For example, if debugging a 603e Rev3 processor, one would use -rev3 to distinguish Revision 3 from other supported revision levels. Once the proper JTAG driver is loaded into the Processor Probe memory, the -rev flag is not required.  
  
If RISCWatch only supports one revision level of a given processor, the -rev flag is not required.
- tname*NAME*          Overrides the TARGET\_NAME setting specified in the environment resources file. *NAME* can be the name of the target or the target's IP address.
- ttype*TYPE*          Overrides the TARGET\_TYPE setting specified in the environment resources file..
- tport*PORT*          Overrides the TARGET\_PORT setting specified in the environment resources file.
- mps*MPS*             Overrides the MPS\_FILE setting specified in the environment resources file.
- tty                   specifies that RISCWatch is to be run in TTY mode. TTY mode is a command line driven mode of RISCWatch that does not rely on the user interface for input and output. This option is only available on a non-Windows host.

## JTAG Ethernet Targets and the RISCWatch Processor Probe

The RISCWatch processor probe is an Ethernet-to-JTAG convertor, converting commands sent from RISCWatch to the appropriate series of processor accesses through the JTAG port of the probe. The probe has a dedicated JTAG controller chip to drive the JTAG signals in hardware as opposed to a slower, emulated approach in software.

To talk to RISCWatch, the processor probe contains two programs in its flash memory: the interface that RISCWatch communicates with (called the “Generics”), and the underlying specific JTAG device driver. When a RISCWatch JTAG Ethernet target is initially invoked, RISCWatch will check the version of the Generics and the specific JTAG driver loaded in the processor probe (or requested with the `-proc` flag or `PROC` environment variable) against the versions of the files located in the directory specified by the `RWPPC_DIR` environment variable. If the Generics or JTAG drivers do not match, the file(s) from the `RWPPC_DIR` will be loaded into the processor probe. Because loading the processor probe may corrupt the processor’s JTAG controller, it is recommended that the processor be reset after the loading is complete.

**Note:** If you wish to maintain the current processor state, the processor probe must be disconnected from the target until the correct Generics and JTAG driver are loaded.

Generics and JTAG driver filenames supported for currently available processors are included in the README file provided for this version of RISCWatch.

The following are some considerations to note when using the Processor Probe:

- For JTAG connections, the target processor clock speed must be at least twice the JTAG clock speed. For Processor Probe targets, the JTAG clock speed defaults to the maximum supported value by the processor probe and the processor being used. The RISCWatch command `'jtag'` (see p. 5-65) can be used to query the default setting and set it if need be. It is possible that noise on the interface to the target may adversely affect data passed between RISCWatch and the target. If memory or register reads appear to be unstable when using a processor probe connection, see if using the `'jtag'` command to lower the JTAG clock speed fixes the problem.
- RISCWatch will attempt to update the Processor Probe flash memory if it detects that the processor type desired for the RISCWatch session does not match the processor type which the Processor Probe is currently initialized for (this behavior may be overridden by using the `PROBE_FLASH` environment variable). Updating the Processor Probe flash memory with the JTAG connector connected to the target typically puts the processor into an unrecoverable state. Therefore, RISCWatch will always attempt to reset the processor after the Processor Probe flash memory is updated.
- The `PROBE_FLASH` environment variable can be used to disable updates to the Processor Probe flash memory. See “Environment Resources,” p. 3-5 for details on how to force, or completely bypass, a reflash of the Processor Probe.

- The suggested procedure when updating the Processor Probe flash memory is as follows:
  1. Start with the Processor Probe connected to the target.
  2. Following the update of the flash, if you got a warning message saying RISCWatch was unable to soft stop the processor while RISCWatch was coming up, attempt to reset the target from RISCWatch via the 'reset' command or Reset window.
  3. If the reset from RISCWatch fails, reset the target via its reset switch. If that doesn't satisfactorily reset the entire board, a power on reset will be required on the target.
  4. Following the reset, enter the 'stop' command from RISCWatch to start debugging.

#### Operational Notes:

- Disconnecting/connecting the Processor Probe from/to the target while power is applied may affect the state of the target and a reset of the target may be required.
- Cycling power on the Processor Probe typically puts the target in an unrecoverable state and a reset of the target will be required. Always wait for the LEDs on the front of the Processor Probe to stop blinking before resetting the target.
- RISCWatch will not allow more than one copy of RISCWatch to communicate with the Processor Probe at one time. If the Processor Probe is in use, the error message "communications port already in use" will appear. In some conditions the communications link may not close correctly, thereby locking out RISCWatch from coming up again. A couple of the more common situations where this may occur are:
  1. Rebooting a PC while RISCWatch is running
  2. Disconnecting the Processor Probe from the host while RISCWatch is running, and subsequently terminating that RISCWatch session with the Processor Probe still disconnected.
  3. If this condition occurs, cycling power on the Processor probe will clear the communications link.

---

## Main Window Resources

RISCWatch employs a graphical user interface (GUI) that needs to have the host platform window system running.

When RISCWatch is started, it will bring up the windows specified in the **rwppc.lay** file. The first time RISCWatch is run, or at any other time when no **rwppc.lay** file is available, or if `LOAD_LAYOUT = NO` is specified in the environment file, the debugger brings up only the main command window. It is this window, shown in Figure 3-3, that will be used to access all of the debugger features.

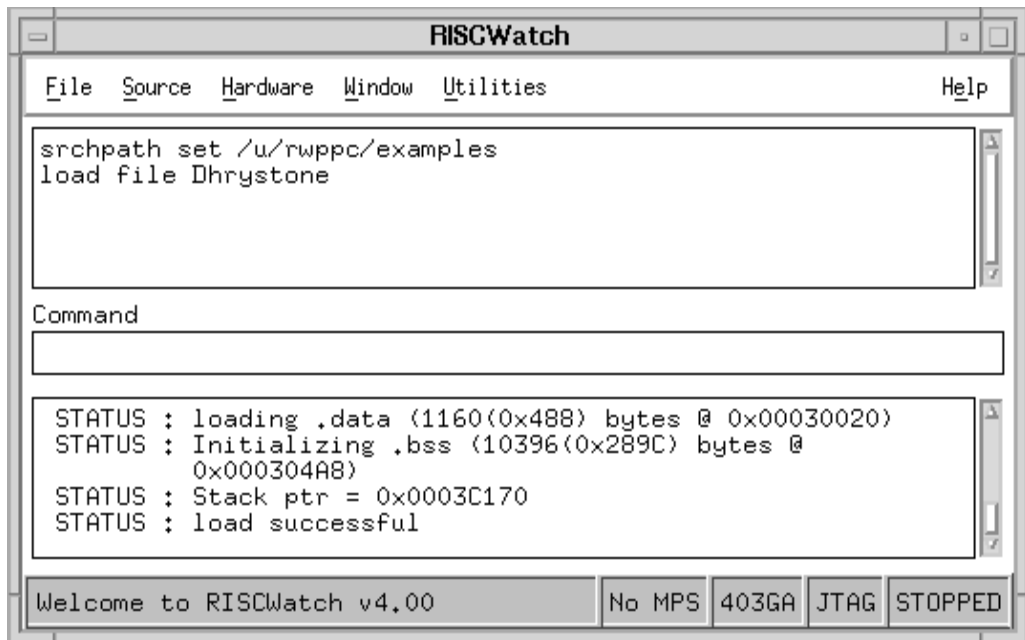


Figure 3-3. Sample Main Window

**Note:** The list of items found at the bottom of the screen may be different depending on the level of RISCWatch, target type, etc.

At the top of the window resides the menu bar which contains the names of the major program access points. Directly below the menu bar is a scrolling window which maintains a history of all the commands entered through the command line interface. Commands in this window can be re-executed or edited and then executed as described in "Command History Usage" on page 3-44.

Directly below the command history window is the command line interface that is used to send commands to RISCWatch to be processed. The commands entered here are the same as the ones which may be used in a command file to help automate development and testing of products using supported PowerPC processors. For a list of the commands and their syntax, select the Help option from the menu bar.

Directly beneath the command line interface, is the scrolling message window which maintains a history of all entered commands and their resultant status, help and error messages. As each command is entered, it is echoed to this window and will be followed by status or error messages. This format allows all commands and their resultant actions to be viewed at any time.

At the bottom of the Main Window resides a status bar, which displays updated information about current debug activity. A message area shows progress messages. An MPS area indicates whether or not multiprocessor support is enabled. A chip area identifies the chip name corresponding to the current debug context. A target type area indicates the method of communication being used for the current debug session. A processor status field also indicates whether the target processor is either running, stopped, halted, powered off, or if the status is unknown.

## Menus

The RISCWatch menus are used to access those parts of the program which require interaction with the user. Menu items can be commands or sub-menus. Selecting an item runs its corresponding command or displays its corresponding sub-menu.

Menu items can be selected by clicking on a menu option to pull down the corresponding menu. Moving the mouse to a menu item highlights the item. Clicking on a highlighted item selects the item. Unavailable selections are grayed-out. Clicking outside the menu closes the menu without making a selection.

Clicking on a menu displays a pull-down containing the selections for that particular menu, as shown in “Main Window Menu Options” on page 3-42.

The menu bar contains the following menus:

- File
- Source
- Hardware
- Window
- Utilities
- Help

What follows is a list of the menus and their selections. Next to each selection is a brief description of its function.

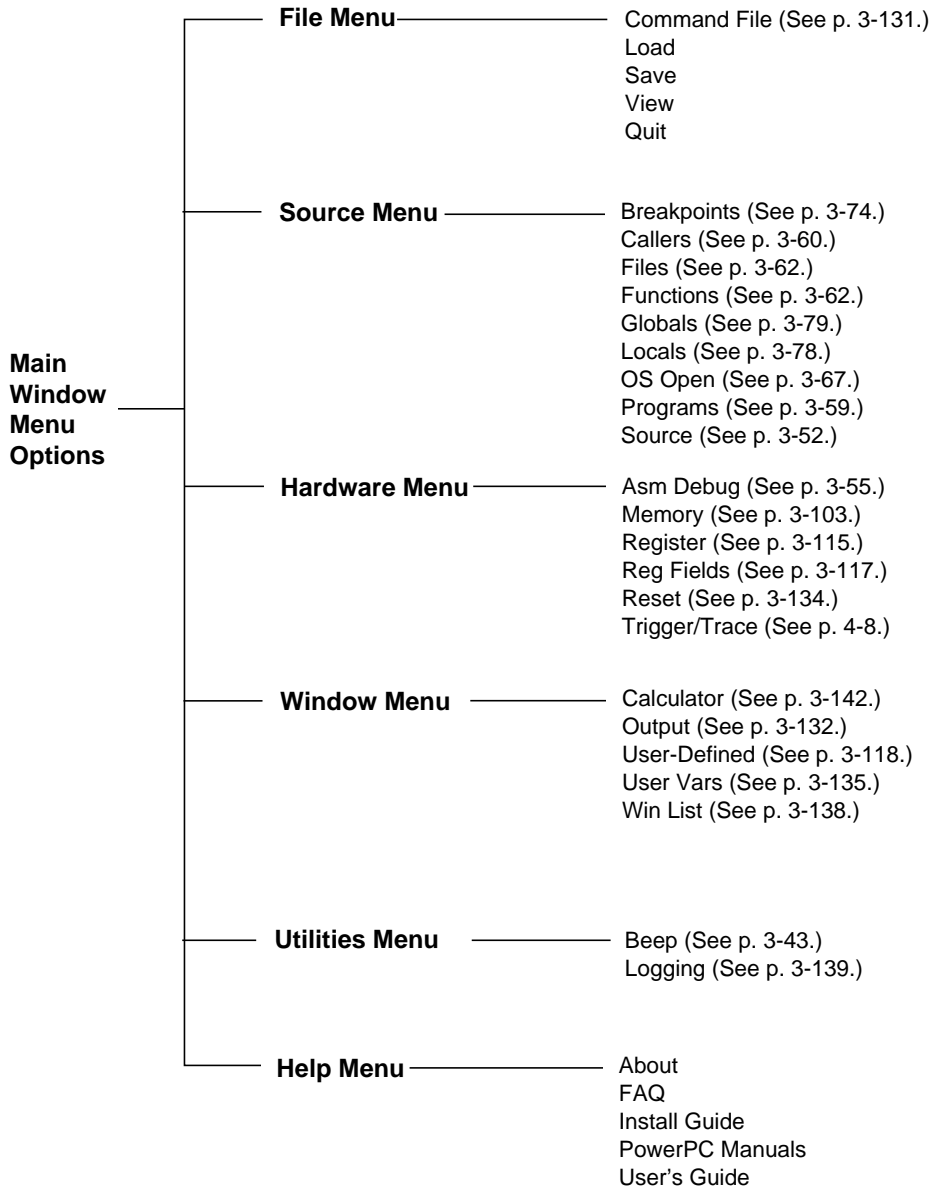


Figure 3-4. Main Window Menu Options

## File Menu

Command File	Run a command file
Load	Load a memory/register/layout file
Save	Save a memory/register/layout file
View	View a selected file
Quit	Terminate the program

## Source Menu

Breakpoints window	Displays breakpoints
Callers window	Displays called functions
Files window	Displays files in current context
Functions window	Displays functions in current context
Globals window	Displays global variables
Locals window	Displays local variables
OS Open window (OS Open target only)	Display OS Open threads and status
Programs window	Displays programs in current context
Source window	Displays source file in current context

## Hardware Menu

Asm. Debug	Displays the Assembly Debug window
Memory	Displays memory window pull-down
Register	Displays a register access window
Reg Fields	Displays a register field access window
Reset	Reset the processor, or display a reset window (JTAG target only)
Trigger/Trace	Displays the Hardware Trigger/Trace window

## Window Menu

Calculator	Displays the desktop Calculator window
Output	Command Message Output information window
User-Defined	Loads a user-defined window
User Vars	Displays user-created variables window
Win List	Display window list

## Utilities Menu

Beep	Turns the program error beep on or off
Logging	Enable/disable logging and give current logging status

## Help Menu

About	Display RISCWatch version information
FAQ	Frequently asked questions
Install Guide	Display RISCWatch User's Guide
PowerPC Manuals	Display links to various PowerPC documentation
User's Guide	Display RISCWatch User's Guide

## Command Line Usage

RISCWatch supports a rich set of commands which are used to access processor resources, thereby facilitating debug of software and hardware. A list of RISCWatch commands and their syntax is given in the section "Command Quick Reference" on page 5-3.

These commands may be typed into a command file to be executed by RISCWatch or used in the user interface via the command line. The command line is the interface between RISCWatch and the user. It is simply a single-line text editor that is used to compose commands and their arguments.

Commands that are valid from the command line may also be entered on the input line, as described in "Input Line Usage" on page 3-49.

The command line understands all alphanumeric keys as well as the Enter, Backspace, Delete, Insert, Home, End and arrow keys.

## Command History Usage

The RISCWatch Main window maintains a list of all commands the program has executed since it was started. This list consists of a scrollable window located between the menu bar and command line interface.

After more than a few commands have been entered, the scroll bar attached to the window will need to be used to view the commands which have scrolled off.

By using the scroll bar attached to the window, it is possible to view all the commands entered since RISCWatch was started. This proves helpful at times to see the precise order in which the commands were issued.

The command history list is also useful for editing or executing previously entered commands. To edit a previous command, simply place the mouse over the command and click the left mouse button. RISCWatch will place the command on the command line where it may be edited and executed if desired. The up arrow and down arrow keys can also be used to retrieve previously issued commands from the command history list. The up arrow key navigates sequentially back through the list while the down arrow moves forward through the list.



To execute a previously entered command, simply place the mouse over the command and double-click the left mouse button. RISCWatch will execute the command as though it had been typed in by the user.

The up and down arrow keys may also be used to sequentially scroll through the history of entered commands. As each command is recalled in turn, it will appear on the command line. From there it may be edited as desired and executed.

## Message Window

The message window is located at the bottom of the RISCWatch Main window. Every time a command is entered into the command line interface, it is echoed in this window. It will then be followed by status or error messages indicating the result of the execution of the command. After a few commands have been entered, it will be necessary to use the scroll bar attached to the window to view earlier commands because they have been scrolled off to show the latest ones.

The message window is not editable and is used as feedback to the user as well as maintaining a history of command usage and status. The contents of the message window will be very similar to that of a RISCWatch log file, which is described in “Log Files” on page 3-139.

---

## Running Your Programs

### Preparing the Program for Debug

Generally, for source level debug, a program must be compiled with a debug option selected. Additionally, no optimization option can be used. Also, the target processor architecture must be specified as PowerPC. All libraries used must also be statically linked into the program unless they already reside on the target.

For specifics about compiling and linking programs for debugging, refer to the documentation included with the compiler and linker being used.

For compiling and linking programs intended for use with the PowerPC 400Series Evaluation Board Kits, refer to the documentation for the kit being used.

### Loading Files

Files can be loaded either from the command line in the Main window, or by using the File|Load pulldown. Refer to the command reference for the complete list of options available for the **load** command. Enter the command and desired options on the command line and hit enter.

To load a file using the load pulldown, select the file to be loaded. Additional prompts will be presented to allow the user to specify the file format and any other applicable options. See “Load Memory Window” on page 3-64 for more information.

For source level debug, loading a file includes both target and host initialization. The target embedded system is typically initialized with the text and data sections of the file. The host system is initialized with the symbolic debug sections of the file (symbol table, line table, etc.). If the debugger has not been initialized to debug a program via **load**, **start\_thread**, **attach**, or **restart**, all source level debug capabilities are disabled.

To facilitate source level debug on applications which are resident on the target prior to RISCWatch invocation, the **load** command provides the ‘host’ keyword which will load the symbolic debug information on the host without changing the state of the target system. This method of loading is quite useful when debugging ROM resident code.

The actions performed during the load are summarized below.

For ROM Monitor and JTAG targets:

1. A **load file** command will unload ALL previously loaded files.
2. A **load host filename** command will unload only the *filename* being loaded, if it is already loaded.
3. A **load host filename** must either be statically linked at the desired text/data locations or the text/data parameters must be supplied with the **load** command (that is, load information is not retrieved from the target).

**Note:** If a specified filename and/or directory path contains a space character, you’ll need to put the entire string in quotation (”) marks whenever you type these in by hand (usually on the command line and in command files)

For an OS Open target:

1. A **load file filename** will be assumed to be a dynamic load. A load info will be issued after the target load. All programs included in the return block will be loaded on the host. If the target program, that is, the program specified in the **load** command, is already on the host, it will be unloaded and then reloaded. If other programs are already on the host, they will remain loaded, that is, they will not be reloaded.

**Note:** Any other programs loaded on the host but not included in the load info return block will be left alone.

2. A **load host filename** must either be statically linked at the desired text/data locations or the text/data parameters must be supplied with the **load** command (that is, load information is not retrieved from the target).
3. A **start\_thread** or **attach** will behave as the **load** file except the target will not be loaded.

## Loading Boot and Boot Image Files

A boot file is defined to be an XCOFF or ELF file which was created with entry code consistent with an OS Open executable or a PowerPC 400Series evaluation board support package executable. This type of executable was never designed to run successfully on the target system.

The PowerPC 400Series evaluation board support package provides a boot image program which takes a boot file and creates a boot image file. The boot image file contains a 32 byte header, followed by a binary image of the loadable portions of the ELF or XCOFF file. This file may also contain additional binary data (controlled by options on the 'boot image' program) which is required for OS Open use (symbol table, string table, etc.).

To facilitate the user in debugging boot files, the **load file** command attempts to recognize a boot executable. This is done by looking for the hex number '004d5054' four bytes beyond the designated entry point. If this special sequence is found, RISCWatch will edit the text section of the executable in an attempt to make the code execute without the need of loading the boot image file. In addition, the symbol and string table is loaded on the target system if the 'nosym' flag is not designated. This method of loading has proven to be effective on non-OS Open boot files.

It is important to note that the entry code in a boot file load executes differently from the entry code provided in a boot image file. For this reason, the **load image** command has been added to allow the user to load a boot image file. RISCWatch will strip off the 32-byte header of the boot image file and load the remaining bytes of the file on the target. The start address of the load is designated in bytes 3-7 of the header. Once loaded, the IAR register is set to the value designated in bytes 16-19 of the header.

The following actions and descriptions define three typical debug scenarios using boot and boot image files

- Load and Debug of a Boot File
  1. Issue the **load file** command to load the host and target.
  2. This provides full-function support with restart capability.
  3. Entry code is modified by RISCWatch to allow execution.
- Load and Debug of a Boot Image File
  1. Issue the **load image** command to load the target.
  2. Issue the **load host** command to load the debug information on the host system.
  3. Entry code runs exactly as intended without modification.
  4. Program restart is accomplished by reissuing the **load image** command.
- Load and Debug of OS Open Threads

1. Bring up RISCWatch using the ROM Monitor target.
2. Close all windows except the Main window.
3. Issue the **load image** command with the filename of the OS Open boot image file.
4. Issue the command **logoff**. The ROM Monitor will exit debug mode and start the execution of OS Open. If a terminal emulation screen is up, you should see the OS Open shell prompt.
5. Select 'file' on the Main window and then select 'quit' to exit RISCWatch.
6. Edit the environment file (**rwppc.env**). Change the TARGET\_TYPE to 'osopen'. Make sure the TARGET\_NAME matches the name or address used by your OS Open image.
7. Bring up RISCWatch using the OS Open target.
8. Issue a **start\_thread** or **attach** command to the thread you want to debug.
9. Note that steps 1-6 are required to load OS Open. These steps are not required if some other method is used to load OS Open.

## Executing the Program

Once a file has been loaded successfully, it can be started by issuing the **run** command from the Main window, or by pressing the Run button on the Source or Assembly Debug window. Note that the debugger may not automatically stop when it gets to the end of the program. Breakpoints or other mechanisms should be used to prevent the program from running into non-program memory locations upon execution completion.

When a program is initially loaded, the Instruction Pointer will often be pointing to start-up code which has no corresponding source files for the debugger to use. A message will be displayed when this situation occurs. In these cases, a breakpoint can first be set in the application code and, when it is hit, the debugger context will be updated for the current Instruction Pointer. The source code will then appear in the Source window.

## Following Program Execution Flow

Program flow is usually followed with a series of actions that cause the program to start and stop at various locations of interest throughout the code. Some of the actions that control program execution include:

1. Setting breakpoints and running to them (**run**)
2. Stepping one source line (**linestep**)
3. Stepping into a function (**callstep**)
4. Returning from a function (**retstep**)

5. Stepping one assembler instruction (**asmstep**)

6. Restarting a program (**restart**)

These commands can be executed from the command line, as specified in the command reference section, or via buttons on the Source and/or Assembly Debug windows.

Tracing back through execution contexts can be performed using the Callers window. Refer to the Callers window description and the Quick Start sections for more details on how these windows and commands can be used to follow program execution flow.

## Input Line Usage

The RISCWatch input line can be used to provide a shortcut method of performing window search and scroll actions. The input line will appear at the top of a RISCWatch window if the window has focus and a keyboard character is typed which corresponds to a supported function for that window. Table 3-2 describes each of the available functions:

**Table 3-2. Input Line Functions**

<b>Key</b>	<b>Function</b>	<b>Parameter</b>	<b>Supported Windows</b>
<b>F12</b>	command line	Any command line command	all
<b>/</b>	find forward ( <b>find</b> command)	search string	specified in <b>find</b> command description
<b>\</b>	find backward ( <b>findb</b> command)	search string	specified in <b>findb</b> command description
<b>?</b>	find exact ( <b>finde</b> command)	search string	specified in <b>finde</b> command description
<b>:</b>	scroll to line ( <b>line</b> command)	line number	specified in <b>line</b> command description
<b>;</b>	scroll to source line ( <b>srcline</b> command)	source line number	Source window

The first field of the input line will indicate the function being performed. That will be followed by an entry field which can be used to specify any parameters for the function, if necessary.

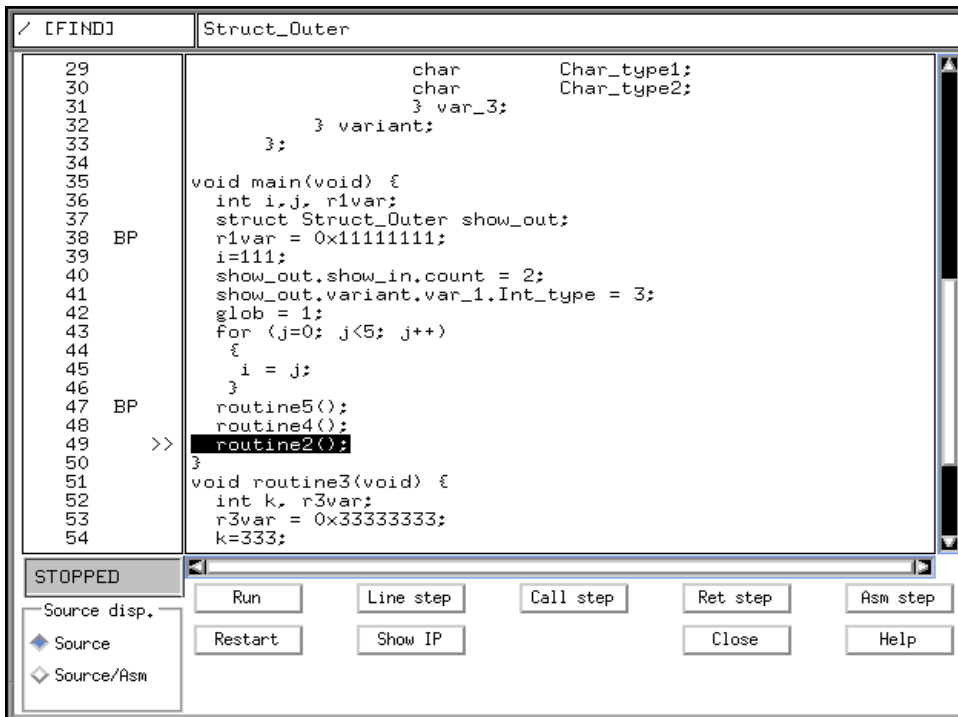


Figure 3-5. Sample Input Line Displayed

For example, entering a command valid from a command line (not all commands can be used from a command line) or searching for a string in a window can be done in the input line.

For example, typing a '/' character in a window which supports the **find** command will display the input line at the top of the window with the first field specifying '/' [FIND]'. In this case the parameter to be entered in the entry field would be the string to search for.

Typing the enter key will perform the requested function. Typing the ESC key, or performing any mouse action on another window, will close the input line with no action taken.

Refer to Chapter 5, "Debugger Command Reference," for detailed information concerning any of the commands mentioned above.

The input line automatically uses the associated window (the window which had focus when the input line was brought up) as the window parameter for those functions which require it. In the case of the Variable Configuration window and the Breakpoint Select window, which have more than one subwindow, the subwindow to use for an input line function can be selected by clicking the mouse in the subwindow (either on an entry or on a blank line) or by selecting a scrollbar with the mouse if it will result in a scrolling event.

Also for these two windows, selecting one of the 'Move all to...' push-button will select the subwindow to which the move was done as the subwindow to be used for subsequent input line functions.

If the entry field is left blank for any of the find functions, the last string which was specified for a find function will be used as the search string to perform a 'next' type search for the associated window.

**Note:** On some host platforms, if a control in a window has focus, it may be necessary to give the window itself focus by clicking the mouse on the window background or titlebar before it will recognize keyboard characters.

## Source Level Debugging

### Source Window

The Source window consists of a Source File subwindow with a Status subwindow, a Source Mode selection groupbox, and pushbuttons. For example, Figure 3-6 shows the Source window in Source/Assembly mode.

The title bar indicates the source file currently being displayed. The file which is displayed in the Source window can be changed by performing one of the following actions:

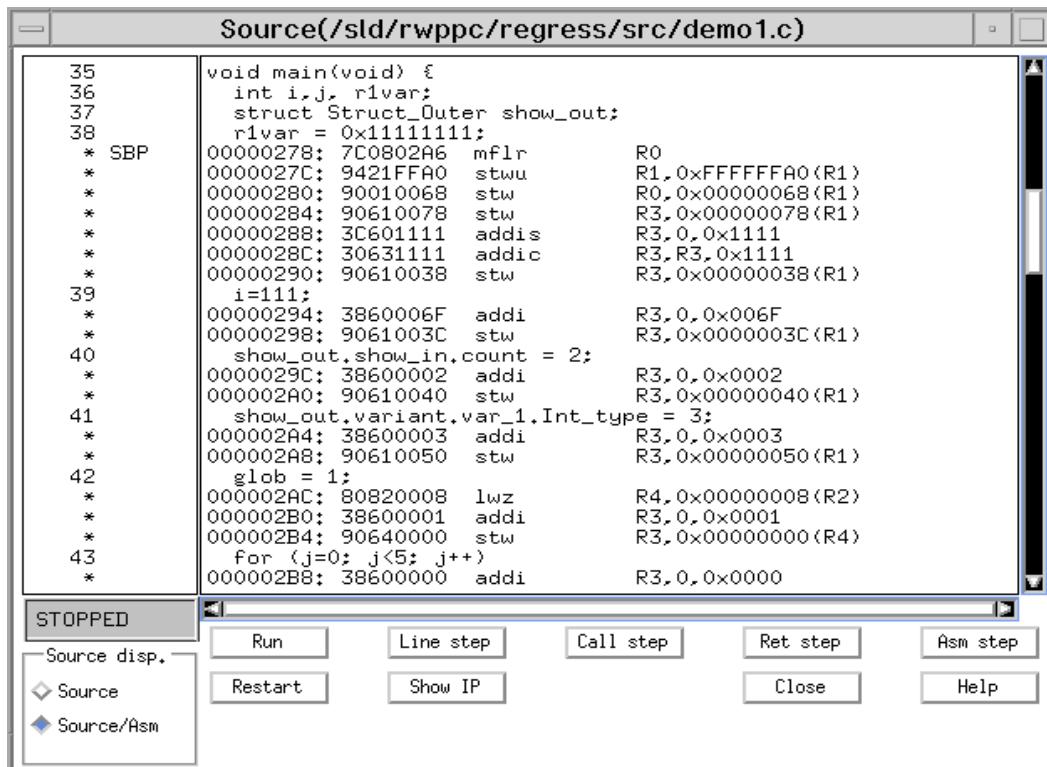


Figure 3-6. Sample Source Window



- Initiate debugging via a command like **load**, **start\_thread**, **attach**, or **restart**.
- If the debugger has not been initialized to debug a program via one of the above commands, all source level debug capabilities are disabled.
- Change the current context as in the case of a breakpoint being hit in another file, performing an execution command, or selecting an entry from the Callers window.
- Select an entry from the Files, Functions, or Breakpoints windows using the **file** command

The title bar will also include the name of the function containing the current Instruction Pointer if the following is true:

- The Source window was updated as a result of an execution action completing (stepping, hitting a breakpoint, etc.), and the file in the Source window contains the function associated with the current Instruction Pointer.
- The file in the Source window has no debug information.

In regular Source Mode, a source file which is part of the current program is displayed in the Source File subwindow, with the corresponding source line numbers displayed in the Status subwindow. In Source/Asm Mode, a source file which is part of the current program is displayed in the Source File subwindow, with both source lines and assembly instructions displayed. Assembly instructions appear for each source line which has instructions associated with it, directly below the corresponding source line. In this mode, the Status subwindow shows the line number for corresponding source lines, and an asterisk for assembler lines. The displayed assembly instructions come from the file image of the loaded program. This differs from the instructions displayed on the Assembly Debug window, which are determined by reading the target system memory.

The Source Mode groupbox consists of two buttons, one for Source only and one for Source/Asm. The display mode is changed by selecting the appropriate button. The button which is on indicates the current mode. If a file is currently displayed when the display mode is changed, the window will be updated to show the source file in the new mode. Regardless of whether a file is currently displayed, any subsequent files which are displayed in the window will be displayed in the mode reflected by the button which is on in the Source Mode checkbox.

The Status subwindow shows source line numbers, denotes assembly instructions with an asterisk, indicates the current Instruction Pointer, and indicates any instruction breakpoints which are set. A double arrow (>>) is displayed on the line corresponding to the current Instruction Pointer address. In Source/Asm mode, this indicator will appear next to the assembly instruction associated with the Instruction Pointer address.

The letters 'BP' will appear on the line corresponding to an instruction breakpoint if the Source window is in Source Only mode. In Source/Asm mode, the letters 'SBP' or 'HBP' will appear next to each assembly instruction for which a software or hardware breakpoint has been set. Breakpoints can be set or deleted by clicking the mouse in the Source subwindow on a valid line. If in Source/Asm mode, breakpoints can only be set by clicking on lines corresponding to assembler instructions. If a breakpoint cannot be set on a selected line, an error message will be generated.

If the Breakpoint Mode (selectable via the **bpmode** command or from the Breakpoints window) is set to Hardware or Hardware Step, breakpoints can only be set on assembler instructions (requiring Source/Asm mode). This is because setting a break on some source lines may require setting breakpoints on multiple assembly lines associated with the source line (the 'for' statement is an example), and only a finite number of hardware breakpoint registers are available at any one time.

Directly below the Status subwindow is the processor/process running indicator. This field indicates whether the processor (in the case of a JTAG target) or process (in the case of a ROM Monitor or OS Open target) is currently running or stopped. If the processor/process is running, the Run/Stop button will be titled "Stop", and the status indicator will be "Running". Pressing the button in this state will cause the processor/process to be stopped. If the processor/process is stopped, the Run/Stop button will be titled "Run", and the status indicator will be "Stopped". Pressing the button in this state will cause the processor/process to run. This is the same functionality which exists on the Assembly Debug window (see p. 3-57). The status and button state will be updated automatically during the course of the debug session to reflect any changes in the processor/process state. If the debugger is currently not attached to and debugging a target, the status indicator on this window will be a string of periods ("....."). If a processor/process is running, all controls or actions are disabled for all source level debug windows except for the processor/process status indicator and the Run/Stop button on the Source window.

Breakpoints are toggled by single clicking the mouse on a line in the Status subwindow corresponding to a valid source line. If no break is currently set at the line, a breakpoint is set by single clicking the mouse on the line, and the bp indicator appears in the Status subwindow on that line. Conversely, if a break is currently set at the line, a breakpoint is deleted by single clicking the mouse on the line, and the bp indicator is removed in the Status subwindow on that line. If a breakpoint is set or deleted from the source window, the Breakpoints window is updated accordingly.

For details on how to perform character string search operations, or how to quickly scroll to a specific source line number, see "Input Line Usage" on page 3-49.

- **Additional Functions available in Popup Menu**

Holding down the right mouse button in the Source File subwindow will produce a popup menu containing a list of additional functions relating to the source window. It allows the user to Go To or Run To a particular source line, to toggle the display mode of the source window, and to inspect variables within the file being displayed. The Go To option will set the IAR to the location of the address corresponding to the first assembly instruction of the source line selected. The Run To option will actually set a break at that address and run the processor to the target location. The toggle mode option will change the display format of the Source File subwindow between Source only and Mixed Source/Assembly modes. Choosing the Inspect option will bring up an Inspect Window containing the variable selected by the initial right mouse button click.

## Scrolling Source Window Contents Using the Keyboard

The data contained in a source level debug window with focus can be scrolled different ways using the keyboard. Following are the keys which can be used to scroll data:

**Table 3-3. Keyboard Options for Scrolling**

Key	Function
Up Arrow	Scroll up one line
Down Arrow	Scroll down one line
Left Arrow	Scroll left one section
Right Arrow	Scroll right one section
Page Up	Scroll up one page
Page Down	Scroll down one page
Home	Scroll to top of contents of window
End	Scroll to bottom of contents of window

## Assembly Debug Window

Assembly level debug can be carried out in several ways. One way is via a source disassembly in the Source window. This can be used only when the source file has been compiled with debug information.

Another way to perform assembly level debug is via the Assembly Debug window. The Assembly Debug window allows memory to be read, altered and written as assembly opcodes and disassembled text. This window uses an actual memory disassembly, so it can be used independent of whether the source exists or was compiled with debug information. Multiple instances of the Assembly Debug window are permitted to show a variety of address ranges simultaneously. The screens are distinguishable by an instance number appearing in the window title.

Refer to “Debugging at the Assembly Level” on page 2-13 for an example of how assembly level debug can be performed.

This window is displayed by selecting the Asm Debug option of the menu bar’s Hardware pull-down choice. What follows is a description of this window’s functionality.

- **Data area**

The data area for the Assembly Debug window is a large text editing area which consists of three parts: memory addresses, data words and disassembled text. The memory addresses are listed sequentially in a column along the left hand side of the data area. The data words are located in a column adjacent to their respective memory addresses. The

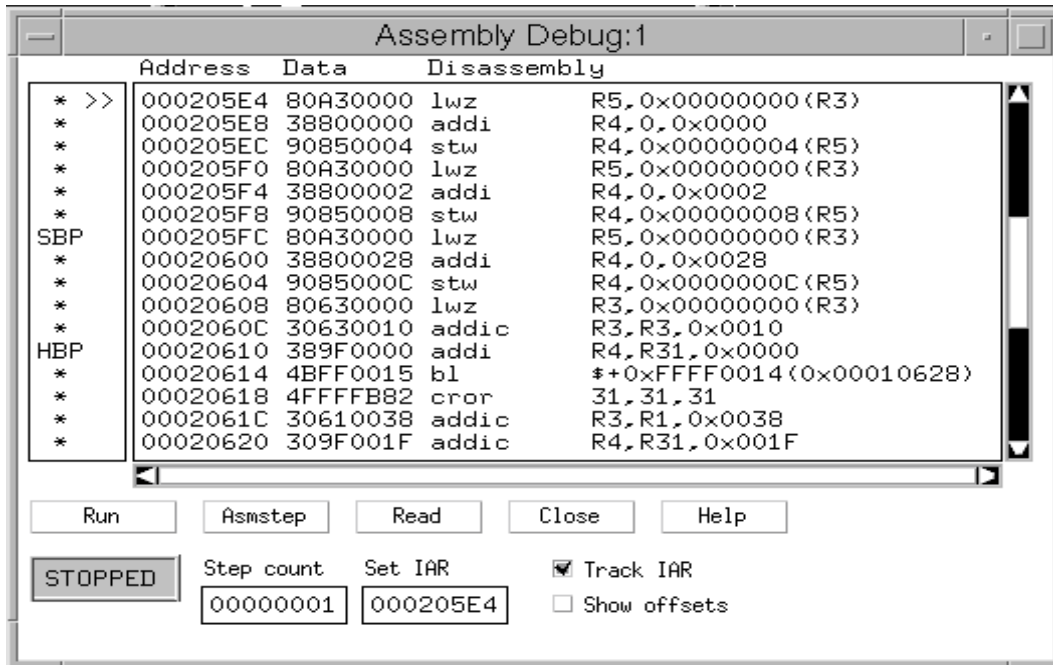


Figure 3-7. Sample Assembly Debug Window

disassembled text consists of each data word being disassembled and then displayed in the adjacent column.

Each of these areas can be edited thereby allowing addresses or data to be altered and then written back to memory. Editing one of the memory address values allows for the disassembled display of any piece of memory. Simply use the mouse to place the cursor next to one of the addresses. Then type in the new address and press the Enter key. The appropriate memory addresses will be read from memory, disassembled, and then displayed in the data area.

It is also possible to change the memory words or disassembled text. To change a particular memory word, simply use the mouse to place the edit cursor next to the desired word. Type in the new word and press the Enter key. The newly entered value will be written and then the display will be updated with the disassembly text for the new word.

Similarly, the disassembled text may be edited by using the mouse to place the edit cursor next to the desired text. Type in the new assembly text and press Enter. The assembler will then be called to create a new memory word which will be written to the appropriate address. The display will then be updated with the newly created memory word.

Data values entered for new addresses and memory words are expected to be input in hexadecimal format.

- **Scroll Bars**

Clicking on a vertical scroll arrow alters the display address by one line or opcode. Clicking on the area between a vertical arrow and the current scroll position alters the display address by one screen's worth of data. To display a given address, use the address entry schemes described in the Data area section.

The page up and page down feature may also be accessed via the keyboard Page Up and Page Down buttons.

- **Breakpoint subwindow**

The breakpoint subwindow is located to the left of the data area and is used to set, clear, and display hardware or software breakpoints. An asterisk appears next to each disassembly line shown in the data area when no breakpoints are set on the corresponding address. It also is used to perform Run To or Go To actions corresponding to the selected memory location.

To set a hardware or software breakpoint for a particular memory address, simply use the mouse to click on the corresponding asterisk. This will set a hardware or software breakpoint, depending on the current Breakpoint Mode (selectable via the **bpmode** command or from the Breakpoints window). For that address an 'HBP' or 'SBP' marker replaces the asterisk, indicating that a hardware or software breakpoint has been set for that address.

To clear a breakpoint, simply click on the 'HBP' or 'SBP' marker. This will clear the breakpoint and restore the asterisk marker for that memory location.

To execute the Run To and Go To actions, place the cursor on the asterisk next to the desired target address of the action. Then hold down the right mouse button. A selection list will appear, allowing the user to choose the desired function. The GoTo selection will cause the IAR to be changed to the target address, while the RunTo selection will set a break at the target address and run the processor to that location.

- **IAR cursor**

The IAR cursor is used to indicate which memory word is being pointed to by the IAR register. The IAR cursor appears as the >> characters in the breakpoint subwindow and will point to the IAR memory address if it appears in the data area display text.

- **Run/Stop button**

The Run/Stop button is used to start the processor/process if it is currently stopped, or to stop it if it is currently running. In the case of a JTAG target, a processor is running or stopped. In the case of a ROM Monitor or OS Open target, a process is running or stopped.

Run is used to start or stop a processor/process; Stop is used to stop it. When a processor/process is stopped, debugger context is updated based on the current Instruction Pointer value for the target. If a processor/process is running, all controls or

actions are disabled for all source level debug windows except for the processor/process status indicator and the Run/Stop button on the Source window.

The current run/stop state of the processor/process is seen directly below this button in the processor/process running indicator. This is the same functionality which exists on the Source window (see p. 3-54). Once memory has been loaded with code and any applicable hardware and/or software breakpoints set, the Run button would be pressed to start the processor/process running.

If the processor/process successfully starts running, the Run button will change to a Stop button and the processor/process running indicator will be updated to indicate running. The processor/process may be stopped asynchronously by pressing the Stop button. Doing so will change the Stop button to the Run button and change the processor/process running indicator.

If, while the processor/process is running, a breakpoint is activated, or the processor/process stops for any reason, the Stop button will change to the Run button and the processor/process running indicator will be updated to indicate that the processor/process is stopped. The IAR field will reflect the current IAR value.

Depending on the setting of the Track IAR check box, the data area will either remain unchanged (check box not selected) or will display the code at the IAR address (check box selected). The IAR cursor will point to the appropriate memory location if the check box is set or if the check box is not selected and the IAR is still within the address range of the displayed data. Otherwise, it will be removed.

- **Asm Step button**

The Asm Step button is used to single-step the processor/process to execute one or more 4-byte instruction values. Instruction stepping single-steps the processor/process starting with the instruction at the memory address referenced by the IAR. Every press of the Asm Step button will execute the number of instructions indicated by the value in the Step count field located directly beneath the Asm Step button.

- **Step count**

The Step count field is used to register a new step count value. This value is used to determine how many instructions will be single-stepped for every press of the Asm Step button. To change this step count value, use the mouse to place the edit cursor in the step count, type in the new count value and then press Enter. The step count value must be entered in hexadecimal format.

- **Modifying the IAR**

The current IAR value may be modified to change the execution sequence of code that is being debugged using the Assembly Debug window. Use the mouse to place the cursor in the Set IAR field. Then type in the new IAR value and press ENTER. This will write the new value to the IAR and update the contents of the data area to reflect this new code execution point. The IAR value must be entered in hexadecimal format. When the IAR

value is changed, the entire source level debugger context will be updated for the new IAR value.

- **Track IAR Check Box**

This check box is used to select the update policy used when the processor is stepped or is stopped after a run. When the check box is selected, the window contents will track the IAR setting; the data area will display the code at the IAR address and the IAR cursor will point to the IAR address location. If the check box is not selected, the data area contents will remain unchanged regardless of the IAR setting. The IAR cursor will move to the new address location if it is within the currently displayed address range; otherwise it will be removed.

## Programs Window

The Programs window consists of a Programs subwindow with horizontal and vertical scrollbars, and push-buttons.

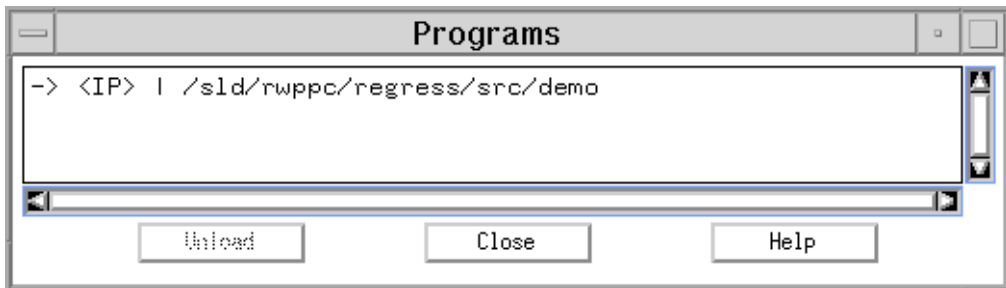


Figure 3-8. Sample Programs Window

The Programs subwindow shows a list of all the programs which the debugger session knows about. The **load** command is the mechanism by which the debugger generates program information on the host for a particular program, and thus becomes 'aware' of the program.

The first field for a program entry is used to indicate which program is currently active. A '->' symbol will appear in this field if the program entry matches the program which is currently active, otherwise it will be blank. The next field for a program entry is used to indicate which program contains the current Instruction Pointer. A '<IP>' symbol appears in this field if the program entry matches the program in which the current Instruction Pointer is located, otherwise it is blank. The last field shows the fully qualified name of the program which was loaded.

If the mouse is single-clicked on a program entry for a program which is not currently active, the debugger context will be switched to the new program, making it the active program. If the new program contains the Instruction Pointer, and the debugger is attached to the target, all appropriate source debug screens will be updated to reflect the context at the current Instruction Pointer. If the new program does not contain the Instruction Pointer, and the

debugger is attached to the target, the Source, Locals, and Caller windows will be blanked out, and the Files, Functions, and Globals windows will be updated for the new program. In these cases, the Programs window itself will be updated to indicate the new active program and execution commands will still be valid.

If the debugger is not currently attached to the target (for example, after detaching from a thread for an OS Open target), the Programs window is still updated to show the programs loaded on the host. In this case the source level debug screens is not functional, so single-clicking the mouse on an entry will not affect any source debug screens. The window can still be used to unload programs.

If the mouse is single-clicked on a program entry for the program which is currently active (i.e., has the '->' symbol next to it), the selection is highlighted and the Unload push-button will become enabled. The Unload push-button will unload the program from the host debugger, effectively making the debugger unaware of the programs existence, and preventing the use of any normal source level debug capabilities for that program. The target will not be affected by the unload. Any program on the Programs window can also be unloaded by double-clicking on the program entry. If a program has been unloaded and you wish to debug it once again, the **load** command can be used to make the debugger aware of any program which is still resident on the target. Refer to the **load** and **unload** commands in Chapter 5, "Debugger Command Reference."

One example of the usefulness of this function is for dynamically loaded programs on an OS Open target. If the OS Open image and the loaded programs have any function calls to the other, it is possible to use the Programs window to switch active programs so that code and variables may be viewed at any time for each program.

It is also possible to set breakpoints in either program, if you wanted to stop in another program at a certain instruction, or if you inadvertently stepped into another program (say, at a place with no debug information) and you wanted to view the code in the program from which you came (and possibly set a break and do a run to get back to where you were previously).

For details on how to locate specific character strings in this window, see "Input Line Usage" on page 3-49.

## Callers Window



The Callers window lists the names of calling programs and functions in the current context. This window consists of a scrolling text window and a menu bar, as shown in Figure 3-9.

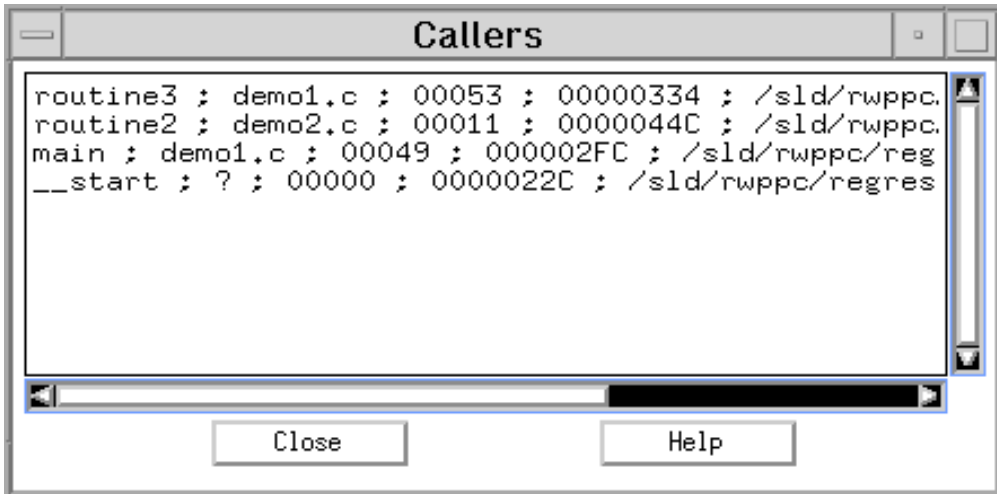


Figure 3-9. Sample Callers Window

The information is presented essentially as a pushdown stack, with the current (called) function appearing as the top entry. As subsequent function calls are made, they then appear at the top, and the other functions are listed below. Similarly, as function returns are carried out, the top entry is removed, and the others moved up on the screen.

Single-clicking the left mouse button over any given entry causes the debugger to change context to the selected (caller) function entry. The Source window shows the source file associated with the given function, and the source line where the function call was made is highlighted. Similarly, the Locals window variables are switched back to the variables and values valid at the time of the function call. See "Local Variables Window," p. 3-78 for additional information on assuring correct Local variable display. This method can be repeated on all of the entries to traverse the entire call chain at any point in the program execution.

Each Callers entry lists, in order, fields that indicate the function name, the source file containing the function, the line number of the calling instruction, the return address of the calling function, the program name, and the stack pointer address.

For details on how to locate specific character strings in this window, see "Input Line Usage" on page 3-49.

## Files Window

The Files window displays source filenames in the current context. This window consists of a menu bar and a scrolling text window, as illustrated in Figure 3-10.

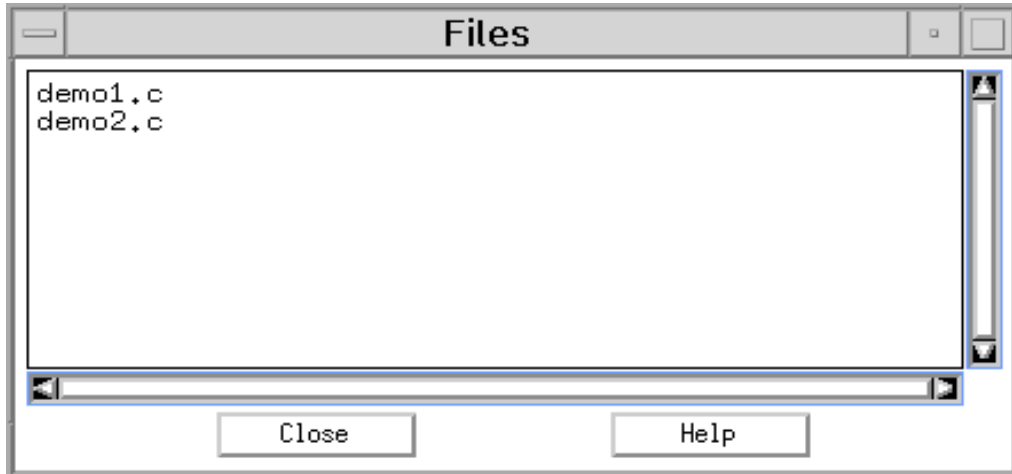


Figure 3-10. Sample Files Window

The Files window lists all the source files contained in the executable currently loaded in the debugger. Single-clicking on any given entry causes that source file to appear in the Source window. The path the debugger uses to search for the file is dictated by the settings made using the **srchpath** command.

The debugger first looks for the source file according to the path specified in the window. If it is not found there, the search proceeds according to any paths that were specified via the **srchpath** command. Source files can also be viewed as ASCII files using the File|View pulldown found on the Main window or by using the **view** command.

For details on how to quickly locate a specific file name in this window, see "Input Line Usage" on page 3-49.

## Functions Window

The Functions window consists of a Functions subwindow with horizontal and vertical scrollbars, a Functions display mode selection groupbox and pushbuttons. The Functions subwindow displays functions for the current program. The format of the function entries, and which functions are displayed, depends on the Functions display mode setting.

The Functions display mode groupbox consists of four radio buttons. Each radio button can be used to change which functions are displayed in the window (only those functions with

symbolic debug information, or all functions in the program) and how they are sorted (alphabetically by name, or by ascending address). The Functions display mode is changed by selecting the appropriate button. The button which is selected indicates the current mode.

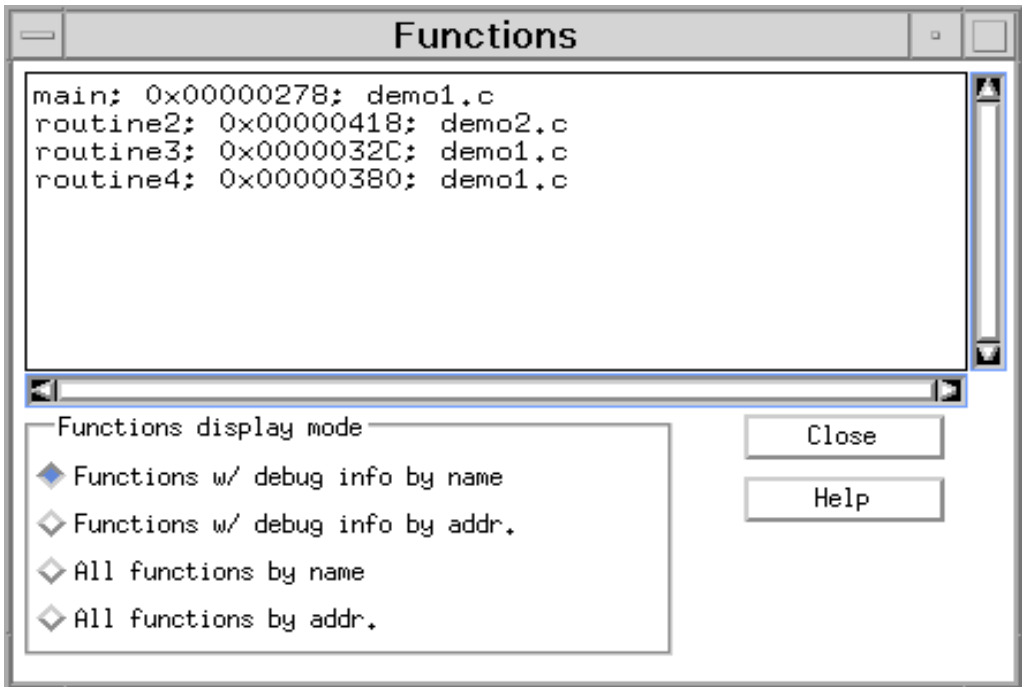


Figure 3-11. Sample Functions Window

When a mode is selected which sorts the function entries by name, each entry will consist of the function name, followed by an address value, followed by the name of the source file which contains the function. The entries will be displayed in alphabetical order by name. When a mode is selected which sorts the function entries by address, each entry will consist of an address value, followed by the function name, followed by the name of the source file which contains the function. The entries will be displayed in order by ascending address.

In all cases, the address value in a function entry will be the address of the start of the function.

When a mode is selected which displays functions with symbolic debug information, only those functions for which there is symbolic debug information in the program will appear. Otherwise, all functions in the program will be displayed.

A function's entry can be selected by single-clicking the mouse on a line containing a functions entry within the window. If the debugger has sufficient information from the functions entry, the Source window will be updated to show the file which the function is in, with the

source line corresponding to the start of the function appearing highlighted in the middle of the view.

A breakpoint can be toggled by double-clicking the left mouse button on a function entry. A breakpoint will be toggled at the address of the start of the function (which is the address value in the entry). Regardless of the function mode setting, the Breakpoint Mode setting (selectable via the **bpmode** command or from the Breakpoints window) determines whether hardware or software breakpoint processing will be used.

For details on how to quickly locate a specific function name in this window, see “Input Line Usage” on page 3-49.

## Load Memory Window

The Load Memory window provides target memory write capabilities using most of the file formats defined for the **Load** command. The window is displayed by choosing the 'File' pulldown on the RISCWatch Main window, then selecting Load|Memory.

Prior to display of the Load Memory window, a standard dialog window is presented to enable the user to supply the name of a file to be opened for reading. The dialog supplies a list of files and directories to choose from. A single left mouse click on a specific file, followed by the selection of the 'OK' button, will result in the display of the Load Memory window.

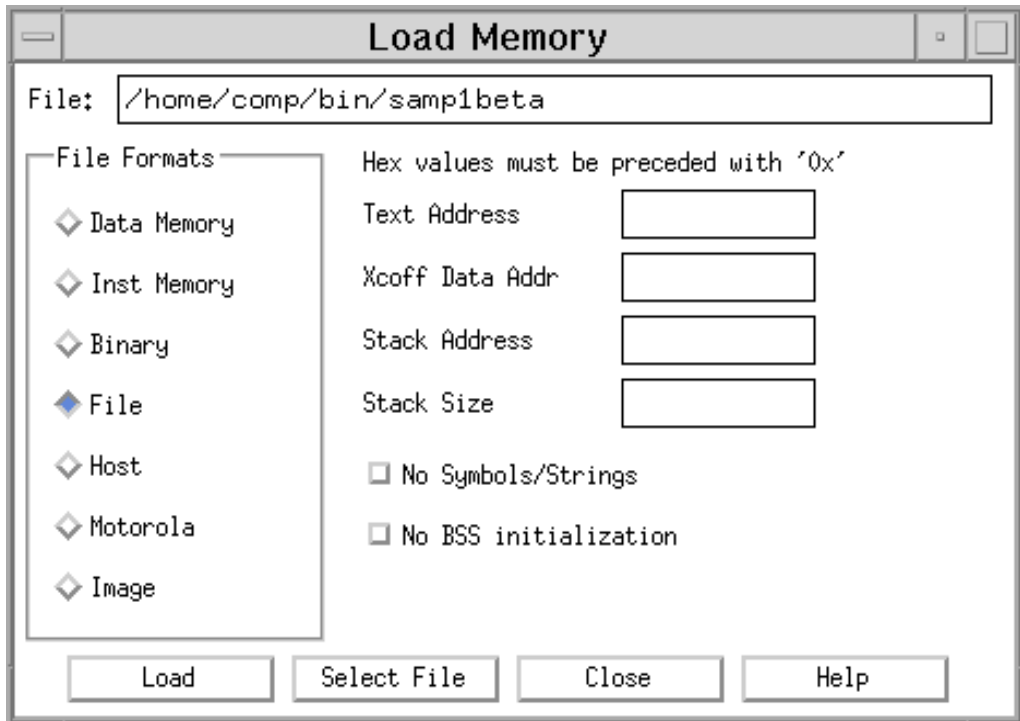


Figure 3-12. Load Memory

The Load Memory window consists of the following fields:

- **File**

This field indicates the file name to be loaded. The file name can be altered by choosing the 'Select File' button at the bottom of the window or by directly typing in the field provided.

- **File Formats**

This group box provides a list of supported file formats to choose from. See "load," p. 5-69 for a description of the supported file formats. Choosing a particular file format will result in the enabling, or disabling, of the remaining input areas of the window.

- **Start Address**

This field is enabled for 'Data Memory', 'Instr Memory', or 'Binary' format selections and indicates the initial target address to use when loading the file. This field must be coded if the 'Binary' format is selected.

- **Text Address**

This field is enabled for 'File' and 'Host' format selections and indicates the initial target address of the instruction section of the file. This field is required when loading an XCOFF file and is optional when loading an ELF executable. If the ELF file was not compiled with relocation enabled, this field is ignored.

- **XCOFF Data Addr**

This field is enabled for 'File' and 'Host' format selections and indicates the initial target address of the data section of the file. This field is required when loading an XCOFF file and is ignored when loading an ELF executable.

- **Stack Address**

This field is enabled for 'File' and 'Host' format selections and indicates the initial target address of the local stack area. This is an optional field which is equivalent to the 's=' option provided on the Load command. Use of this field is not recommended since RISCWatch creates a default 16K stack area during a file load. In addition, most embedded applications establish their own stack area during initial start up code execution.

- **Stack Size**

This field is enabled for 'File' and 'Host' format selections and indicates the maximum size, in decimal, of the local stack area. This is an optional field which is equivalent to the 'ss=' option provided on the Load command. Use of this field is not recommended since RISCWatch creates a default 16K stack area during a file load. In addition, most embedded applications establish their own stack area during initial start up code execution.

- **No Symbols/Strings**

This Check box is enabled for 'File' and 'Host' format selections and directs RISCWatch not to load the additional symbol table information associated with boot files. See "Loading Boot and Boot Image Files" on page 3-47 for a detailed discussion of boot files.

- **No BSS Initialization**

This check box is enabled for 'File' and 'Host' format selections and directs RISCWatch not to zero out the un-initialized data section (BSS) of the file. This option can significantly improve the file load performance but requires the loaded application to zero out the un-initialized area during program start up. Selection of this option is equivalent to the 'NOZERO' option of the Load command.

- **Load**

This pushbutton is used to execute the appropriate Load command, based on the selections made on the window. The RISCWatch main window will indicate the success or failure of the operation.

- **Select File**

This pushbutton presents a standard dialog window that allows the user to supply an alternate file name for loading.

- **Close**

This pushbutton removes the Load Memory window from view. A subsequent display of the Load Memory window will present the field settings that were in existence when the 'Close' button was pressed.

- **Help**

This pushbutton will present any available help topic for this window.

## OS Open Debugging

The OS Open window is used to display operating system construct information and control debug attachment for an IBM OS Open Real-time Operating system program image. The OS Open window is available only if OS Open is specified as the target in the RISCWatch environment file.

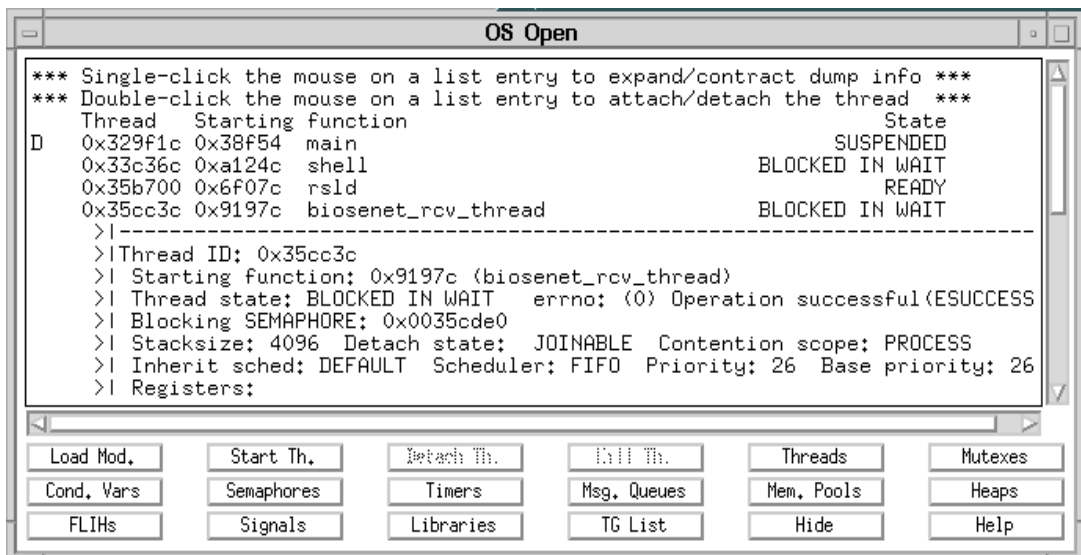


Figure 3-13. Sample OS Open Window

The OS Open window consists of a subwindow with horizontal and vertical scrollbars and a number of push-buttons used to dynamically load a file, start/kill/detach an OS Open thread, and display OS Open construct information.

The subwindow displays information relevant to the construct display push-button which was last selected. For some constructs, single-clicking the mouse on a list entry will display more specific information immediately under the entry, or will contract this information if it is already displayed. There will be a message at the top of the display window if the expansion/contraction function is available for the current display.

**Note:** In general, the contents of the subwindow will not be automatically updated as the application runs on the target. In each case, when a display pushbutton is selected, or a single-click is performed for a construct which supports it, the latest information for the entire window will be retrieved from the target and displayed.

For details on how to locate specific character strings in this window, see “Input Line Usage” on page 3-49.

Following are descriptions of the pushbuttons in the OS Open window:

- **Load Module button**

This pushbutton brings up the Load Module window. Entering the name of a file which is located on a file system mounted on the target OS Open system causes that file to be dynamically loaded by OS Open into the target. Also, the file to be loaded must be located in the current RISCWatch search path. A thread corresponding to the entry point for the program loaded will be queued. A breakpoint will be put at this entry point and the debugger will be initialized to debug this thread.

**Note:** for OS Open systems with Virtual Memory support: Unless otherwise specified, newly loaded modules will be loaded into a new thread group. To specify an existing thread group, use the load file command's tg parameter.

For example, to load module /fat/cat.ld into thread group 0x5435770, type:

```
/fat/cat.ld tg=0x5435770
```

- **Start Thread button**

This pushbutton brings up the Start Thread window. Entering a function name which is part of the target program image will initialize a source mode debug session with OS Open.

A thread corresponding to the specified function will be queued, with a breakpoint set at the entry of the function.

**Note:** RISCWatch cannot be used to debug the OS Open shell.

For OS Open systems with Virtual Memory support: Unless otherwise specified, newly started threads will be started in a new thread group. To specify an existing thread group, specify the thread group id after the function name. For example, to start the thread my\_hello\_world in thread group 0x5435701, type:



my\_hello\_world 0x5435701

- **Detach Thread button**

This pushbutton ends the source mode debug session with OS Open by disconnecting from the thread which is currently being debugged. The thread will continue to run normally on the target.

- **Kill Thread button**

This pushbutton ends the source mode debug session with OS Open by destroying the thread which is currently being debugged.

- **Threads button**

This pushbutton lists each thread in the OS Open system in the display subwindow. If a thread is currently being debugged, a 'D' will appear in the first column of the list entry. If the mouse is double-clicked on a thread list entry, the thread will be attached if it is not already being debugged, or detached if it is currently being debugged.

**Note:** RISCWatch cannot be used to debug the OS Open shell.

If the mouse is single-clicked on a list entry which is not already expanded, the window display will be expanded to show detailed information about that specific thread directly below the thread list entry. If the mouse is single-clicked on a list entry which is already expanded, the detail for that list entry will be contracted.

- **Mutexes button**

This push-button lists each mutex in the OS Open system in the display subwindow. If the mouse is single-clicked on a list entry which is not already expanded, the window display will be expanded to show detailed information about that specific mutex directly below the mutex list entry. If the mouse is single-clicked on a list entry which is already expanded, the detail for that list entry will be contracted.

- **Condition Variables button**

This pushbutton lists each condition variable in the OS Open system in the display subwindow. If the mouse is single-clicked on a list entry which is not already expanded, the window display will be expanded to show detailed information about that specific condition variable directly below the condition variable list entry. If the mouse is single-clicked on a list entry which is already expanded, the detail for that list entry will be contracted.

- **Semaphores button**

This pushbutton lists each semaphore in the OS Open system in the display subwindow. If the mouse is single-clicked on a list entry which is not already expanded, the window display will be expanded to show detailed information about that specific semaphore directly below the semaphore list entry. If the mouse is single-clicked on a list entry which is already expanded, the detail for that list entry will be contracted.

- **Timers button**

This pushbutton lists each timer in the OS Open system in the display subwindow. If the mouse is single-clicked on a list entry which is not already expanded, the window display will be expanded to show detailed information about that specific timer directly below the timer list entry. If the mouse is single-clicked on a list entry which is already expanded, the detail for that list entry will be contracted.

- **Message Queues button**

This pushbutton lists each message queue in the OS Open system in the display subwindow. If the mouse is single-clicked on a list entry which is not already expanded, the window display will be expanded to show detailed information about that specific message queue directly below the message queue list entry. If the mouse is single-clicked on a list entry which is already expanded, the detail for that list entry will be contracted.

- **Memory Pools button**

This pushbutton lists each memory pool in the OS Open system in the display subwindow.

- **Heaps button**

This pushbutton lists each heap in the OS Open system in the display subwindow.

- **FLIHs button**

This pushbutton lists each first level interrupt handler in the OS Open system in the display subwindow.

- **Signals button**

This pushbutton lists each signal in the OS Open system in the display subwindow.

- **Libraries button**

This pushbutton lists each registered library in OS Open system in the display subwindow.

- **Thread Group List button**

This pushbutton is only available if the target is an OS Open system with Virtual Memory support. It will list each thread group in the OS Open system in the display subwindow.

If the mouse is single-clicked on a list entry which is not already expanded, the window display will be expanded to show detailed information about that specific thread group directly below the thread group list entry. If the mouse is single-clicked on a list entry which is already expanded, the detail for that list entry will be contracted.

- **Close button**

This pushbutton closes the window.

- **Help button**

This push-button accesses the on-line RISCWatch User's Guide.

For more information on the OS Open Real-Time Operating System, refer to “Related IBM Publications” on page xxiv.

---

## Managing Breakpoints

Breakpoints within RISCWatch fall into two categories:

- Software breakpoints
- Hardware breakpoints

Software breakpoints are implemented by replacing the instruction at the breakpoint address with a trap instruction. Hardware breakpoints make use of the debugging features designed into specific PowerPC processors. When the processor/process stops, all the trap instructions are replaced with the original instructions residing at the breakpoint addresses.

**Notes:** For PowerPC 6xx/7xx processors connected via a JTAG target, hardware breakpoints cannot be used if software breakpoints are set and, conversely, software breakpoints cannot be used if hardware breakpoints are set.

Hardware breakpoints are not available on OS Open targets.

## Using Software Breakpoints

- **Setting Software Breakpoints from the RISCWatch Debugger Windows**

Software breakpoints can be set or cleared in a number of ways using the RISCWatch Debugger windows. Note that the Breakpoint Mode must be set to Software mode (see **bpmode** on page 5-23).

1. Source window

Software breakpoints can be set and cleared in the Source window (Figure 3-6) by moving the cursor to the targeted source line and then single-clicking the left mouse button on the line corresponding to the targeted source line in the Status window, left of the source lines. An indicator will appear next to the line number of the target source line. Similarly, an existing breakpoint can be cleared by single-clicking on the line. The single-clicking toggles the breakpoint setting for a target source line.

If in mixed/source and assembly mode, the breakpoints can be set and cleared the same way, with the target line in this case being an assembly instruction instead.

2. Breakpoints window

Software breakpoints can be viewed and cleared from the Breakpoints window (Figure 3-14). Double-clicking on an entry will clear the breakpoint. Single-clicking on an entry will highlight the entry and enable clearing by then pressing the Delete button. The Delete All button can be used to delete all current breakpoints.

3. Assembly Debug window

Software breakpoints can be set and cleared from the Assembly Debug window (Figure 3-49) by single-clicking on the buttons along the left side of the disassembly entries. This action also toggles the breakpoint each time it is performed.

4. Functions window

Software breakpoints can be set and cleared from the Functions window (Figure 3-11) by double-clicking the left mouse button on a function entry. A breakpoint will be toggled at the address of the start of the function.

- **Setting Software Breakpoints with the bp Command**

To set a software breakpoint, you can use a **bp** command along with the address of the instruction to stop at and RISCWatch takes care of the rest. For example, to stop just prior to the execution of the instruction at address 0xFFFFC0004, issue the following command:

```
bp set 0xFFFFC0004
```

The processor/process could then be started using the **run** command. If the processor/process were to try and execute the instruction at this address, the processor/process would stop and an event would be generated which RISCWatch would detect. It would then be possible to examine the state of the processor.

To clear this software breakpoint, simply issue the command

```
bp clear 0xFFFFC004
```

See **bp** on page 5-19 in the Command Reference for a detailed description of available functionality.

## Using Hardware Breakpoints

- **Setting Hardware Breakpoints from the RISCWatch Debugger Windows**

Hardware breakpoints can be set or cleared in a number of ways using the RISCWatch Debugger windows. Note that the Breakpoint Mode must be set to Hardware mode (see **bpmode** on page 5-23).

1. Source window

Hardware breakpoints can be set and cleared in the Source window only when the source screen is in mixed source/assembly mode. Single-clicking the left mouse button on the line corresponding to the targeted assembly instruction in the Status window, left of the assembly instructions, will alternately set and clear the breakpoint. An indicator will appear next to the target line in the line number field when the breakpoint is set.

2. Breakpoints window

Hardware breakpoints can be viewed and cleared from the Breakpoints window. Double-clicking on an entry will clear the breakpoint. Single-clicking on an entry will highlight the entry and enable clearing by then pressing the Delete button. The Delete All button can be used to delete all current breakpoints.

3. Assembly Debug window

Hardware breakpoints can be set and cleared from the Assembly Debug window by single-clicking on the buttons along the left side of the disassembly entries. This action also toggles the breakpoint each time it is performed.

4. Functions window

Hardware breakpoints can be set and cleared from the Functions window by double-clicking the left mouse button on a function entry. A breakpoint will be toggled at the address of the start of the function.

- **Setting Hardware Breakpoints with the bp Command**

RISCWatch allows access to the available hardware registers used to control breakpoints through the use of the **bp** command. This type of access allows for the usage of native processor debugging facilities to control when a running processor will be stopped. This access is dependent on the processor being used and the available functionality may vary.

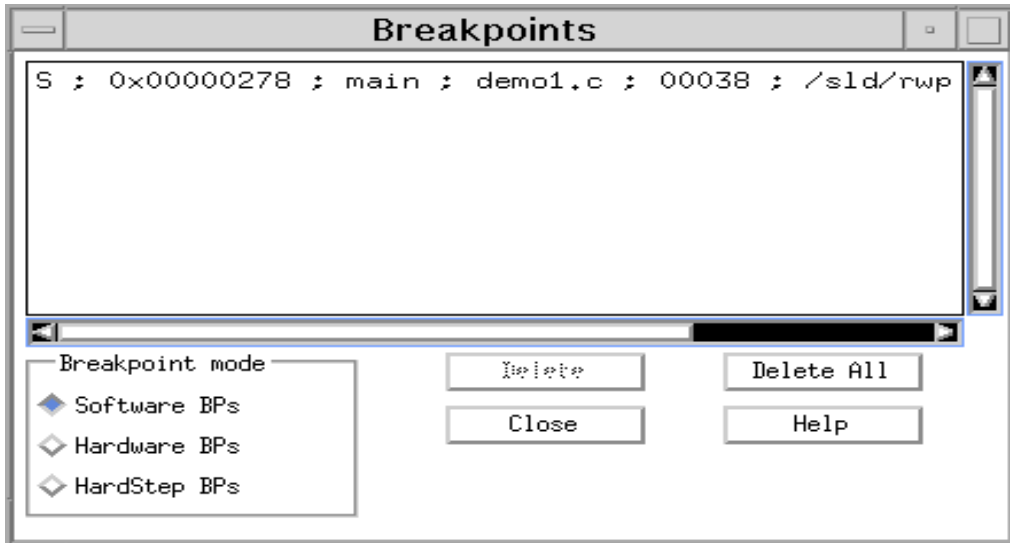


Figure 3-14. Sample Breakpoints Window

“Trigger/Trace Window (400Series Only)” on page 4-7 and “Compound Trigger/Trace Window (403Series Only)” on page 4-11 provide descriptions of other (processor-specific) windows for handling hardware breakpoints.

## Breakpoints Window

The Breakpoints window consists of a Breakpoint subwindow with horizontal and vertical scrollbars, a Breakpoint Mode selection groupbox, and push-buttons. The Breakpoint subwindow displays any breakpoints that are currently set.

The Breakpoint entry contains information about the breakpoint, with each field separated by a semicolon. If the entry is for an Instruction breakpoint, the first field contains the letter ‘H’ or ‘S’ to indicate a Hardware or Software breakpoint, respectively. The next fields in order show the address of the breakpoint, the function containing the breakpoint, the file containing the breakpoint, the line number in the file which the breakpoint is set at, and the program which the breakpoint is set in. If the values of any of the fields cannot be determined by the debugger they will be designated by values of zero in the case of numbers and ‘?’ in the case of strings.

If the entry is for a Data breakpoint, the first field contains the letter ‘D’. The next fields in order show the Data Address Compare value, the Data Address Compare register used, the Data Address Compare Write/Read enable, and the Data Address Compare size.

Breakpoints may be set or deleted in several ways during a debug session. In each case, the Breakpoints window will be automatically updated to reflect the currently set breakpoints.

A breakpoint can be selected by single-clicking the mouse on a line containing a breakpoint entry within the window. This will cause the breakpoint entry to become highlighted. For an Instruction breakpoint, if the debugger has sufficient information from the breakpoint entry, the Source window will be updated to show the source file in which the breakpoint is set, with the source line which the breakpoint is set at appearing highlighted in the middle of the view. No attempt will be made to update the Source window for a breakpoint with an unknown program (program field is "?"). The Assembly Debug window will also be updated when an Instruction breakpoint entry is selected to display memory starting at the address of the breakpoint. Single-clicking on an already selected breakpoint entry will deselect it.

The Delete pushbutton is disabled unless a breakpoint entry is selected, at which time it is enabled. Pressing the Delete pushbutton will cause the selected breakpoint to be deleted. A breakpoint can also be deleted by double-clicking on the breakpoint entry. When an Instruction breakpoint is deleted, the Breakpoints window and the Status subwindow in the Source window will reflect the current status.

The Delete All pushbutton will delete all current breakpoints.

The Breakpoint Mode groupbox consists of three buttons: Software BPs, Hardware BPs, and HardStep BPs. The Breakpoint mode is changed by selecting the appropriate button. The button which is on indicates the current mode.

When in Software mode, breakpoints are set by writing trap instructions in place of program instructions.

When in Hardware mode, user designated breakpoints are set via the hardware debug registers of the target processor. RISCWatch breakpoints used for line stepping and call stepping are applied as software breakpoints.

In HardStep mode, all breakpoints are applied using the hardware debug registers of the target processor. When performing line steps or call steps, a single hardware debug register (highest number IAC/IABR register) is used to run to the next source line or function. In addition, if a breakpoint is applied when all processor resources are in use, a previously applied breakpoint (contained in the highest numbered IAC/IABR) is automatically removed and the new breakpoint applied. HardStep mode is useful when debugging code resident in read only memory, where software traps can not be written.

There are a finite number of hardware breakpoints available. The number is based on the target processor and is dependent on how many hardware debug registers it has. Error messages will be generated if attempts are made to set Hardware breakpoints in Hardware mode and none are available.

If the mouse is single-clicked on an Instruction breakpoint entry which corresponds to a program which is currently not active, the debugger context will be switched for the new program, making it the active program. If the new program contains the Instruction Pointer, all appropriate source debug screens will be updated to reflect the context at the current Instruction Pointer. If the new program does not contain the Instruction Pointer, the Source, Locals, and Caller windows will be blanked out, and the Files, Functions, and Globals

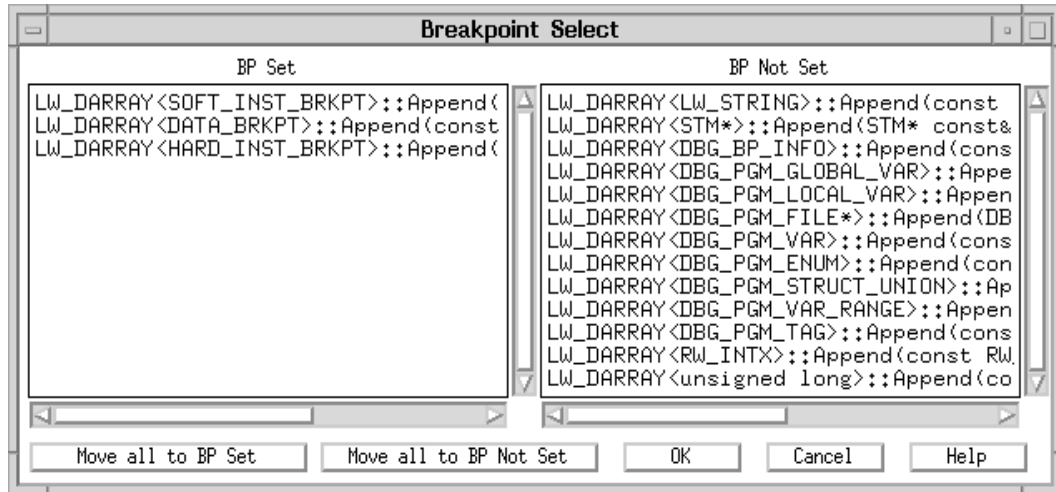


Figure 3-15. Sample Breakpoint Select Window

windows will be updated for the new program. Refer to the Programs window description for more information on debugging with multiple programs simultaneously.

The RISCWatch Debugger also uses the **bp** command to manage both types of breakpoints. See **bp** on page 5-19 for further details.

See “Compound Trigger/Trace Window (403Series Only)” on page 4-11 and “Compound Trigger/Trace Window (403Series Only)” on page 4-11 for additional RISCWatch debugging windows that manage PowerPC 400Series hardware breakpoints.

## Breakpoint Select Window

The Breakpoint Select window appears when an attempt is made to set or delete a breakpoint with the mouse on a source line in the Source window, and that source line corresponds to multiple functions in the program. An example of when this situation could exist is when debugging source code containing C++ templates. The Breakpoint Select window can then be used to set or remove breakpoints for particular functions associated with the selected source line.

The window consists of a BP Set subwindow with horizontal and vertical scrollbars, a BP Not Set subwindow with horizontal and vertical scrollbars, and push-buttons.

The BP Set and BP Not Set subwindows are used to select the functions for which breakpoints related to the chosen source line will be set. If breakpoints are currently set for an associated function, its name will initially appear in the BP Set window. If breakpoints are not currently set for an associated function, its name will initially appear in the BP Not Set window.



Single clicking the mouse on a function in one of the subwindows will move it to the other subwindow. The Move All to BP Set push-button will move all the functions to the BP Set subwindow. The Move All to BP Not Set push-button will move all the variables to the BP Not Set subwindow.

If the information on the Breakpoint Select is applied via the OK pushbutton, the appropriate breakpoints for the selected source line will be set for each function currently listed on the BP Set subwindow. Also, associated breakpoints will be removed if a function is in the BP Not Set subwindow at the time the changes are applied and it initially had breakpoints set. The Cancel pushbutton is used to close the window without applying any changes.

---

## Reading and Writing Program Variables

Many methods of updating and viewing program data are provided by the RISCWatch Debugger. They can be used by themselves or in concert with others to provide a wide range of options on how data is presented.

The Locals and Globals windows display selected local and global variables, respectively, for the program currently being debugged. The Variable Configuration window can be selected from the Locals or Globals window to configure variable information for all Local or Global variables. In addition, the Change Variable window can be used to alter an individual variable's value, type, or display information. The following sections describe the capabilities of each of these windows.

## Local Variables Window

The Locals window displays local variables in the current source file. Figure 3-16 shows an example of a Locals window.

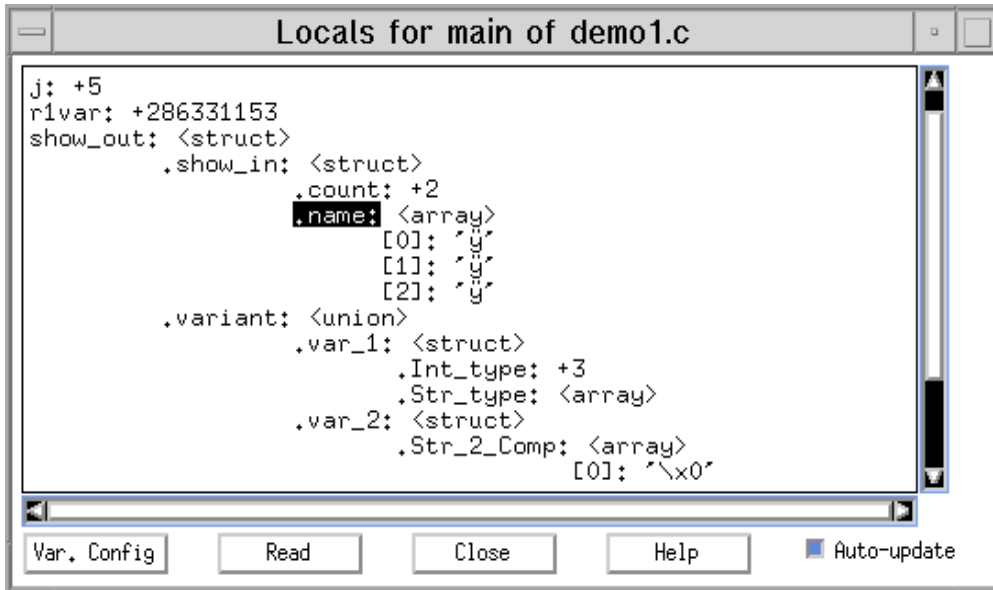


Figure 3-16. Sample Locals Window

The Locals window consists of a Locals subwindow with horizontal and vertical scrollbars and push-buttons. The Locals subwindow displays the visible local variables for a function. The variables which can be displayed are dependent on the current local variable context for the debugger. Variables can be shown which correspond to the current instruction context, that is, variables for the function associated with the current Instruction Pointer address. These are automatically shown after performing an execution command like **run** or **linestep**.

Variables can also be shown which correspond to a previous function in the call chain. The Callers window is used to select the context of a function on the callers stack, and the Locals window will be updated appropriately. Variables displayed in the Locals Screen may have an address indicating a processor register. Proper display of a calling function's register variable (selecting a back level entry on the Callers Screen) requires the existence of a tag word section in the executable. In the absence of a ".tags" section, the Caller's register variable value is assumed to be in the register save area of the called function, which will not always be correct. By using the `-Hoff=debugger_handles_reg_vars` High C/C++ compile option, you can disable local register assignments. All locals will be assigned to memory locations and proper display of all the caller's variables will be guaranteed.

A local variable entry consists of the variable name followed by configurable variable information. Configurable variable information includes the value of the variable (if it is a

fundamental type) expressed in a format selectable by the user, the variable type enclosed in a left/right arrow pair (<>), the address of the variable preceded by an 'at' sign (@), and the size of the variable enclosed in parentheses. The Variable Configuration and Change Variable windows are used to configure the variable information for the local variables.

If the address for a variable is not a valid memory address for the target being debugged, the words 'INVALID VALUE' will appear in place of a numeric value as long as the address is invalid. The address field will show the current address associated with the variable. Variable detail and format changes can still be applied while the variable is in this state, and will be applied if during the course of debugging the program the variable address becomes valid.

For example, if an un-initialized pointer is defined, the contents of this pointer may initially be outside the range of valid memory for the target, in which case any data element pointed to by the pointer would have an invalid value. As soon as the pointer is assigned a valid value for the program, say, by a call to malloc(), the data elements pointed to should then contain valid data.

Single-clicking the left mouse button on a variable entry selects the variable and opens the Change Variable window appropriate for the type of the selected variable (integer, structure, and so on). The Change Variable window is used to configure variable information for an individual variable. See "Change Variable Window," p. 3-85.

Double-clicking the left mouse button on a structure, pointer, or union variable entry expands the variable detail one level if it is expandable and it has not already been fully expanded. You can continue to expand the variable detail another level by continuing to double-click on the variable entry.

Double-clicking the right mouse button on a structure, pointer, or union variable entry contracts the variable detail to the point which was clicked on. Subsequent expansion of the variable at this point will result in the variable being expanded to the level of detail which it was at when it was contracted.

The Variable Config push-button is used to open the Variable Configuration window. The Variable Configuration window, when opened from the Locals window, is used to configure variable information for all the local variables in the current locals context. See "Variable Configuration Window," p. 3-83. The Read push-button is used to manually force a read of the values of the variables which are displayed on the Locals window from the target.

**Note:** "Input Line Usage" on page 3-49 describes shortcut key operations for performing character string searches on this window.

## Global Variables Window

The Globals window consists of a Globals subwindow with horizontal and vertical scrollbars and push-buttons.

The Globals subwindow displays the visible global variables for the program currently being debugged. For performance reasons, when a program is initially loaded, all global variables are set up to be invisible. The Var. Config button must be used to make them visible. A global

variable entry consists of the file which the variable is in, followed by the variable name and configurable variable information. Configurable variable information includes the value of the variable (if it is a fundamental type) expressed in a format selectable by the user, the variable type enclosed in a left/right arrow pair (<>), the address of the variable preceded by an 'at' sign (@), and the size of the variable enclosed in parentheses. The Variable Configuration and Change Variable windows are used to configure the variable information for the global variables.

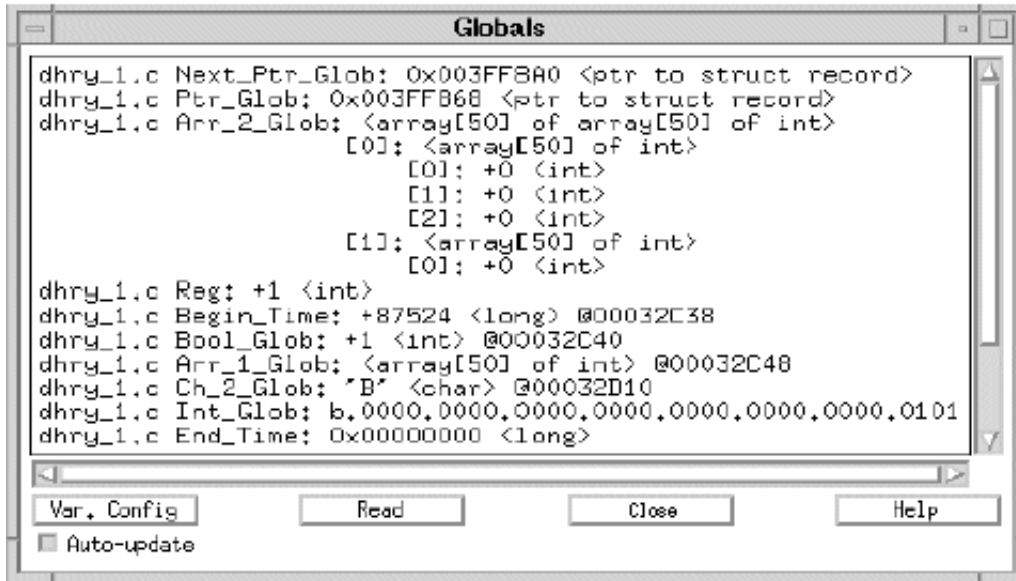


Figure 3-17. Sample Globals Window

If the address for a variable is not a valid memory address for the target being debugged, the words 'INVALID VALUE' will appear in place of a numeric value as long as the address is invalid. The address field will show the current address associated with the variable. Variable detail and format changes can still be applied while the variable is in this state, and will be applied if during the course of debugging the program the variable address becomes valid.

For example, if an un-initialized pointer is defined, the contents of this pointer may initially be outside the range of valid memory for the target, in which case any data element pointed to by the pointer would have an invalid value. As soon as the pointer is assigned a valid value for the program, say, by a call to malloc(), the data elements pointed to should then contain valid data.

Single-clicking the left mouse button on a variable entry will select the variable and open the Change Variable window appropriate for the type of the selected variable (integer, structure etc.). The Change Variable window is used to configure variable information for an individual variable. Refer to the Change Variable window description.

Double-clicking the left mouse button on a structure, pointer, or union variable entry will expand the variable detail one level if it is expandable and it has not already been fully expanded. You can continue to expand the variable detail another level by continuing to double-click on the variable entry.

Double-clicking the right mouse button on a structure, pointer, or union variable entry contracts the variable detail to the point which was clicked on. Subsequent expansion of the variable at this point will result in the variable being expanded to the level of detail which it was at when it was contracted.

The Variable Config push-button is used to open the Variable Configuration window. The Variable Configuration window, when opened from the Globals window, is used to configure variable information for all the global variables in the program. See "Variable Configuration Window," p. 3-83. The Variable Config push-button will be disabled if there is no source debug information for the current program.

The Read push-button is used to manually read the values of the variables which are displayed on the Globals window from the target.

**Note:** "Input Line Usage" on page 3-49 describes shortcut key operations for performing character string searches on this window.

## Inspect Variable Windows

An Inspect window consists of a variable subwindow with horizontal and vertical scrollbars and push-buttons.

Inspect variable windows are used to display and change variable contents and display information in much the same manner as the Local and Global Variable windows. However, the Inspect window contains only one variable per window, and it can be either a local or a global variable. Multiple windows are allowed, and are identified by a colon and instance number, along with the variable name.

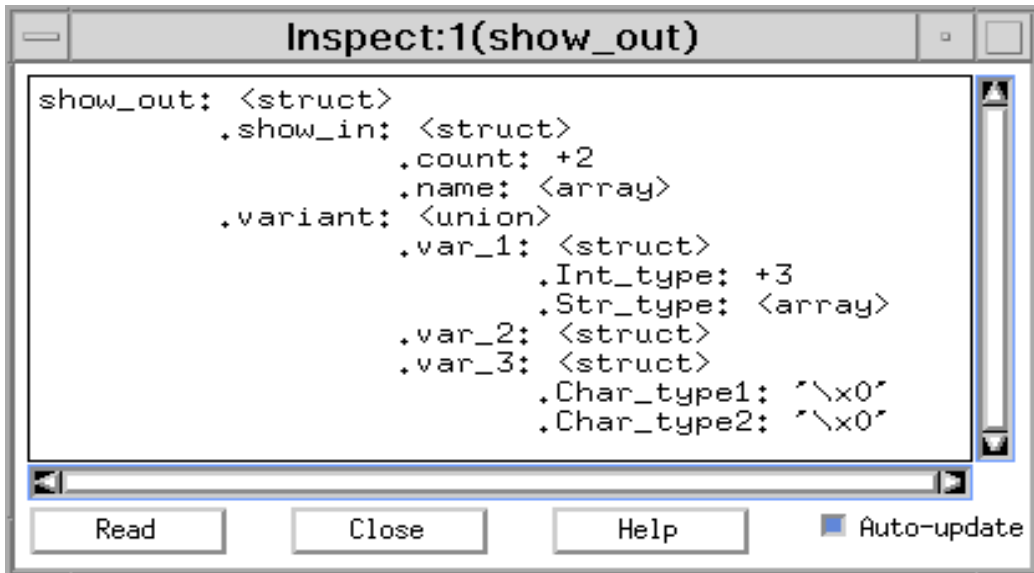


Figure 3-18. Sample Inspect Window

Inspect windows can be created via the GUI interface using the Source window or by command line using the window command. To invoke an Inspect window via the Source screen, the user simply clicks and holds the right mouse button over the variable to be inspected. A menu list will appear that includes an Inspect selection. When selected, a new window is created for that particular variable only. The ability to have multiple copies of the same variable is also supported.

Single-clicking the left mouse button on the variable entry will open the Change Variable window appropriate for the type of the selected variable (integer, structure etc.). The Change Variable window is used to configure variable information for an individual variable. Refer to the Change Variable window description.

Double-clicking the left mouse button on a structure, pointer, or union variable entry will expand the variable detail one level if it is expandable and it has not already been fully expanded. You can continue to expand the variable detail another level by continuing to double-click on the variable entry.

Double-clicking the right mouse button on a structure, pointer, or union variable entry contracts the variable detail to the point which was clicked on. Subsequent expansion of the variable at this point will result in the variable being expanded to the level of detail which it was at when it was contracted.

## Variable Configuration Window

The Variable Configuration window is used to change variable information for all local or global variables. It consists of a Display Information selection groupbox, a Compiler-created Variable selection groupbox, a Visible subwindow with horizontal and vertical scrollbars, a Not Visible subwindow with horizontal and vertical scrollbars, and push-buttons.

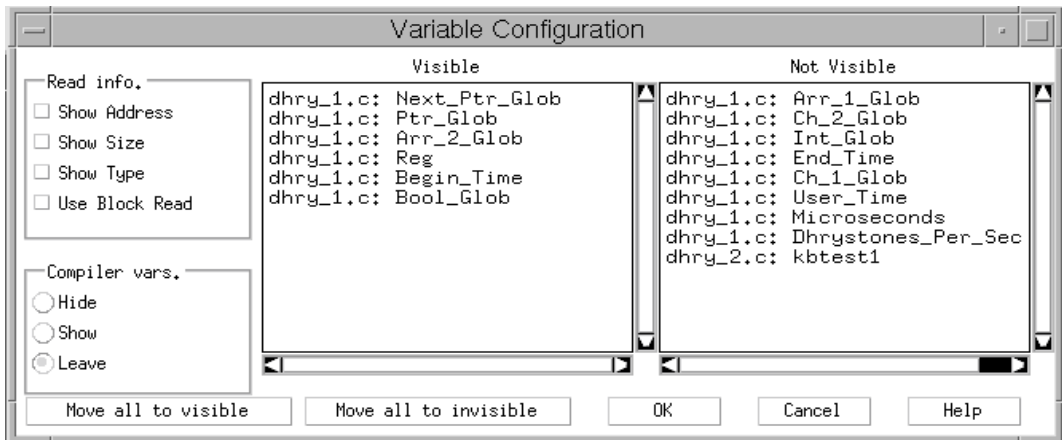


Figure 3-19. Sample Variable Configuration Window

The Variable Configuration window is opened via the Variable Configuration push-button on the Locals or Globals window. The OK push-button is used to apply the selected information to the associated variable window (the variable window from which the Variable Configuration window was opened). The Cancel push-button is used to close the window without applying any changes.

The Variable Configuration window is intended to be used for applying configuration changes to a variable window once it is opened. The Variable Configuration window will be brought down without any changes being applied if it is open and the associated variable window is brought down or updated. An existing Variable Configuration window will also be brought down with no changes applied if another Variable Configuration window or a Change Variable window is opened while the Variable Configuration window is up.

The Display Information groupbox consists of four check boxes. The first 3 check boxes enable display of the Address, Size and Type information. The 'Use Block Read' check box is available to improve variable read performance by performing block reads of structures and array elements. The initial state of the check boxes shows the currently enabled default display information for the associated local or global variable window. If the information on the Variable Configuration window is applied, each variable entry on the variable window will be updated to reflect the selected display information. The display changes will be applied to any variable actively using the 'Use Defaults' read information setting. This setting can be displayed or altered on the Change Variable window.

The Compiler-created variable groupbox consists of three buttons, one to hide variables which are created by the compiler, one to show variables which are created by the compiler, and one to leave the current setting. The debugger keys off variables beginning with two underscores ('\_\_') to determine variables created by the compiler. They are typically present in C++ programs. The initial state is to have the compiler-created variables hidden. Selecting the Hide button will move all variables beginning with two underscores to the Not Visible subwindow. Conversely, selecting the Show button will move all variables beginning with two underscores to the Visible subwindow.

The Visible and Not Visible subwindows are used to select which variables will be visible on the associated variable window. No processing is done for a variable while it is not visible. All local variables are initially visible. All global variables are initially not visible.

Single-clicking the mouse on a variable in one of the subwindows will move it to the other subwindow. The Move All to Vis push-button will move all the variables to the Visible subwindow. The Move All to Invis push-button will move all the variables to the Not Visible subwindow. If the information on the Variable Configuration window is applied, a variable entry will appear on the associated variable window for each variable in the Visible subwindow.

**Note:** For local variables, all variables defined for the function will be shown, regardless of whether they are currently in scope. If multiple instances of variables with the same name are defined with different scope within a function, the variable name will appear repeated times in the window. Each variable instance on the window will correspond to a variable definition within the function.

**Note:** "Input Line Usage" on page 3-49 describes shortcut key operations for performing character string searches on this window.



## Change Variable Window

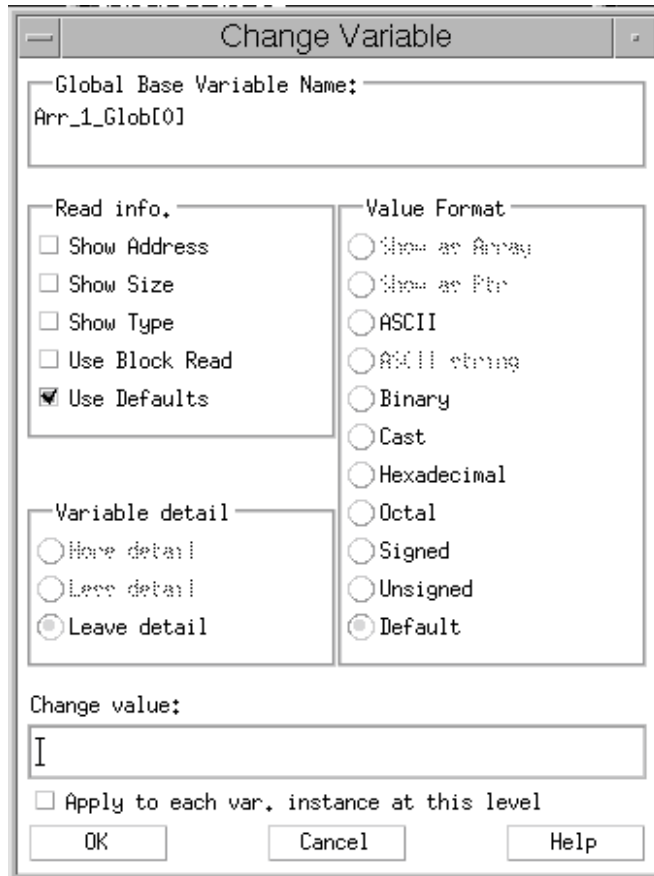


Figure 3-20. Change Variable Window

The Change Variable window is used to change variable information for an individual local or global variable. This window is opened by single-clicking the mouse on an individual line entry in the Locals or Globals window. The type of variable selected determines which format options are enabled when the window is displayed. Variable types are classified as Base (int, char, enum, etc), Array, Pointer, Structure or Union. The following information describes the fields displayed in the Change Variable window.

- **Variable Name**

The Variable Name field contains the name of the variable chosen. In addition, the field title area indicates the associated variable window (local or global) and the current type (Array, Pointer, Base, or Struct/Union) assigned to the variable.

- **Display Information**

The Display Information groupbox consists of three check boxes to enable display of Address, Size and Type information for the selected variable on the associated variable window. The 'Use Block Read' check box is used to improve performance by executing memory block reads on structures and array elements. The 'Use Defaults' check box informs RISCWatch to use the default settings made on the Variable Configuration window.

The initial state of the check boxes shows the currently active display information for the associated variable. If the information on the Change Variable window is applied, the variable entry on the associated variable window will be updated to reflect the selected display information. The display changes will be applied to any portions of the variable which were 'revealed' or expanded, whether they are currently visible or not.

- **Variable Detail**

The Variable Detail groupbox consists of three check boxes: 'More detail', 'Less detail', and 'Leave detail'. 'Leave detail' will always be the default when the window comes up. Selecting 'More detail' will expand the variable to the next level of expansion, if it can be expanded further. If the variable was previously expanded multiple levels from that point, those levels of expansion will be shown as well. Selecting 'Less detail' will contract the variable detail to the level of the selected variable. The detail changes will only take effect if the changes for the window are applied.

Refer to "Expansion/Contraction from Locals or Globals Window" on page 3-88 for more discussion on changing the level of detail for a variable. Note that base variables, such as 'longs' or 'ints', have no additional detail to display, so the 'More detail' and 'Less detail' selections are disabled.

- **Value Format**

The Value Format groupbox consists of a number of buttons used to change the format of the selected variable. For example, if the value of the selected variable is displayed as decimal 12 on the Locals window, it will be displayed as '0x0000000C' if the Hexadecimal format is applied. The following formats are supported: "Show as Array" and 'Show as Ptr' (valid for Pointer types only), ASCII, 'ASCII string' (for base char types), Binary, Cast, Hexadecimal, Octal, Signed, Unsigned, and Default. Default is the format which RISCWatch has defined for each fundamental type. See "Formatting Examples," p. 3-88, for specific details on Type Casting, ASCII string display, and other formatting options.

- **Change Value**

The Change Value field is used to change the value of the variable. Values can be entered in decimal or hexadecimal notation. If an invalid value is entered, an error message will be displayed in the Main window and the Change Variable window will remain visible to accept another entry. When applied, the variable value will be written to the target and the variable entry on the associated variable will be updated to reflect the new value. The Change Value Field does not exist if the Cast or 'Show as Array' Value Format selections are chosen. In addition, 'Array' and 'Struct/Union' variable types do not contain the Change

Value field (a value is only meaningful when referring to a specific member or element of these types).

- **Change Array Subrange**

The Change Subrange field is used to change the number of elements displayed for either an array variable or a pointer variable (when 'Show as Array' Value Format is used). It is initialized with the current subrange value. The limits of the array will be shown in the title above the change field. The low and high subrange values should be separated by a comma with no spaces. If an invalid subrange is entered, an error message will be displayed in the Main window and the Change Variable window will remain up to accept another entry. If a subrange value is entered which is outside the limits for the array, a warning message is displayed and the entered value is used. When applied, the array variable will be expanded on the associated variable window to show the array elements for the entered subrange.

- **Enter Type**

The Enter Type field is used to change the type (Type Cast) of the selected variable. This field is activated when the Cast Value Format is selected. A valid 'type name', followed by any number of '\*'s to indicate pointer indirection, must be entered. Valid 'type names' include fundamental 'C' types such as 'int', 'long', 'signed char', etc... In addition, any user defined 'type name' (i.e. structures defined in your 'C' code) can be used. Using the name '#default' will restore the variable's type to the original compiler setting.

The 'type name' may also be preceded by an optional file name (with or without single quotes), followed by a colon (i.e. file.c:structa\* or 'file.c':structa\*), to direct RISCWatch to search the debug information of a specific source file. In the absence of a file identifier, RISCWatch will only search for a name match in the source file where the variable was originally defined.

Refer to "Type Casting a Variable" on page 3-101 for additional information.

- **Apply To Each Var. Instance**

A check box titled 'Apply to each var. instance at this level' will appear above the buttons at the bottom of the window if the selected variable is part of an array element (and more than one element exists for the array from the perspective of the debugger). If it is selected when changes are applied for the window, they will be applied to each instance of the variable within multiple elements of the array. Refer to "Changing Multiple Instances of a Variable Within an Array" on page 3-93 for a detailed description of this support.

- **OK**

The OK push-button is used to apply the selected information to the associated window (Locals or Globals) for the variable selected. If an invalid value is detected, the Change Variable window will remain visible and an error message will be displayed in the RISCWatch Main status window.

- **Cancel**

The Cancel push-button is used to close the window without applying any changes.

- **Help**

The Help push-button is used to display any available help topic for this window.

## Formatting Examples

This section contains examples on how to manipulate the variable information which is displayed on the Locals or Globals variable window:

### Expansion/Contraction from Locals or Globals Window

Consider the following (unexpanded) structure variable entry on a Locals or Globals variable window:

---

```
show_out: <struct Struct_Outer>
```

---

Figure 3-21. Sample Unexpanded Structure Variable

Double-clicking the left mouse button on this variable line will result in expanding the structure to show the individual elements:

---

```
show_out: <struct Struct_Outer>
          .show_in: <struct inside>
          .variant: <union>
```

---

Figure 3-22. Sample Expanded Structure Variable

Double-clicking the left mouse button again on the same line will continue to expand by one level each data element of the structure:

---

```
show_out: <struct Struct_Outer>
          .show_in: <struct inside>
                  .count: +928 <int>
                  .name: <array[10] of char>
          .variant: <union>
                  .var_1: <struct>
                  .var_2: <struct>
                  .var_3: <struct>
```

---

Figure 3-23. Further Structure Variable Expansion

Note that we could have chosen above to only expand one of the data elements of the structure by moving the mouse to that specific element (.show\_in, say) and double-clicking the left mouse button on it. We can demonstrate this ability to expand an individual element by now double-clicking the left mouse button on the (now visible) name array element of the nested .show\_in structure:

---

```
show_out: <struct Struct_Outer>
    .show_in: <struct inside>
        .count: +928 <int>
        .name: <array[10] of char>
            [0]: "\x0" <char>
            [1]: "\x2" <char>
            [2]: "%" <char>
        .variant: <union>
            .var_1: <struct>
            .var_2: <struct>
            .var_3: <struct>
```

---

Figure 3-24. Single-Element Structure Variable Expansion

Note that in this case the expansion took place from the line which was double-clicked on. Also, because this was an array and not a structure, the elements are listed by array index. In this case, only the first three elements of the array were shown when it was expanded, which is the default setting for arrays with three or more elements. The subrange to view for an array can be changed via the Change Variable window which is opened by single-clicking the left mouse button on the array variable entry. (See p. 3-85.)

Now, we can demonstrate the ability to contract variable elements by double-clicking the right mouse button on the .show\_in element. This will contract the variable information displayed up to this element.

---

```
show_out: <struct Struct_Outer>
    .show_in: <struct inside>
    .variant: <union>
        .var_1: <struct>
        .var_2: <struct>
        .var_3: <struct>
```

---

Figure 3-25. Structure Variable Contraction

The next time the .show\_in element is expanded, it will be expanded to the level of detail to which it was previously expanded above.

Using these techniques, variables consisting of complex data elements can be customized to show various levels of detail for each data element comprising the variable.

## Displaying ASCII Strings

Consider the following variable which is a pointer to type **char** on a Locals or Globals variable window:

```
Str_1_Par_Ref: 0x0002E248 <ptr to char>
```

Figure 3-26. Sample Pointer Variable

Single-clicking the left mouse button on this variable line will open the Change Variable window (See p. 3-85.). One of the options under Value Format is 'ASCII String'. Selecting this format and applying the change will result in the variable entry being updated to show the ASCII string being pointed to:

```
Str_1_Par_Ref: 0x0002E248->"DHRYSTONE PROGRAM, 1"ST STRING"
```

Figure 3-27. Sample ASCII String Display

Variables of type **char** can also be used as the initial point for an ASCII string display. Consider the same string being displayed as an array of characters (expanded to show the first few elements):

```
Str_1_Par_Ref: 0x0002E248 <ptr>
                [0]: "D"
                [1]: "H"
                [2]: "R"
                [3]: "Y"
```

Figure 3-28. Sample Character Array

Single-clicking the left mouse button on any of the character variable entries will open the Change Variable window (See p. 3-85.). Selecting the 'ASCII String' Value Format and applying the change will result in the character variable entry being updated. An ASCII string is displayed, starting from the address of the variable. In this case, it would probably make most sense to choose the first element of the array, resulting in the following format change:

```
Str_1_Par_Ref: 0x0002E248 <ptr>
                [0]: "DHRYSTONE PROGRAM, 1"ST STRING"
                [1]: "H"
                [2]: "R"
                [3]: "Y"
```

Figure 3-29. Sample Array Element Display

Note that in either case of using a pointer or a char as the basis for displaying the string, the debugger will display characters starting from the address of the variable until a NULL character is reached in memory or an internally defined maximum length is reached.

## Handling Multiple Data Elements Referenced by a Single Pointer

Suppose we initialize a data pointer to point to a memory buffer allocated to hold several identical data structures. Typically, individual buffer elements can be manipulated by the program by using pointer arithmetic with the pointer value. It would be cumbersome to view and change the full range of data being pointed to if the pointer variable is restricted to displaying a single element. Fortunately, RISCWatch provides a format to aid in this task.

Consider the following variable, a pointer to type **struct record** on a Locals or Globals variable window. It references individual elements of a buffer containing multiple **struct record** instances, and points to the beginning of the buffer:

---

```
Ptr_Glob: 0x00335DEF <ptr to struct record>
```

---

Figure 3-30. Sample **struct record** Pointer Display

Normally, if we were to expand this pointer, it would only expand one instance of the structure at the address which it is currently pointing to:

---

```
Ptr_Glob: 0x00335DEF <ptr to struct record>
->: <struct record>
    .Ptr_Comp: NULL <ptr to struct record>
    .Discr: +0 STRUCT_0 <enum>
    .variant: <union>
```

---

Figure 3-31. Sample Initial **struct record** Pointer Expansion

What we want to do is to be able to manipulate individual records. RISCWatch supports this ability by allowing a pointer variable entry to be expanded as an array (with a specified number of elements), with each element of the array subsequently being of the type which the original pointer is pointing to.

Single-clicking the left mouse button on this variable line for the original pointer will open the Change Variable window. One of the options under Value Format is 'Show As Array'. Selecting this format option changes the entry field at the bottom of the window so that an array subrange (with the first element having the address of the pointer value) may be specified.

In this case we'll specify the first three elements [0,2]:

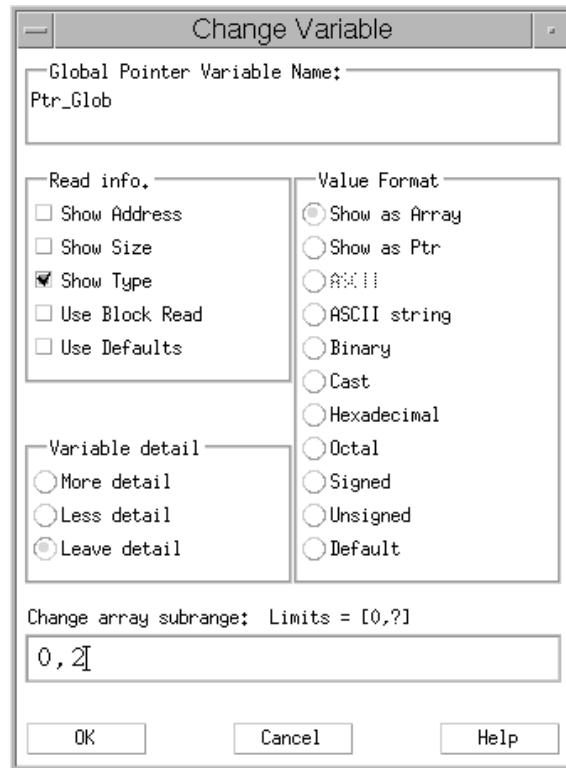


Figure 3-32. Changing Pointer Variables

Applying the changes will result in the variable entry being updated to show an array of three data structures, each representing one of the individual data elements in the buffer.

---

```
Ptr_Glob: 0x00335DEF <ptr to struct record>
          [0]: <struct record>
          [1]: <struct record>
          [2]: <struct record>
```

---

Figure 3-33. Sample Pointer Variable Shown as an Array



Now each individual array element can be manipulated according to the treatment for that type.

---

```
Ptr_Glob: 0x00335DEF <ptr to struct record>
  [0]: <struct record> @00335DEF
    .Ptr_Comp: NULL <ptr to struct record>
    .Discr: +0 STRUCT_0 <enum> @00335DF3
    .variant: <union> @00335DF7
      .var_1: <struct> @00335DF7
      .var_2: <struct> @00335DF7
      .var_3: <struct> @00335DF7
  [1]: <struct record> @00335E1F (48 bytes)
    .Ptr_Comp: 0x00335DEF <ptr> @00335E1F
    .Discr: +1 STRUCT_1 <enum> @00335E23 (4 bytes)
    .variant: <union> @00335E27 (40 bytes)
  [2]: <struct record>
    .Ptr_Comp: 0x00335E1F <ptr to struct record>
    .Discr: +2 STRUCT_2 <enum>
    .variant: <union>
```

---

Figure 3-34. Sample Expanded Pointer Variable Shown as an Array

At any time, the original pointer can be returned to its normal pointer designation by single-clicking the left mouse button on the pointer variable. This action will open the Change Variable window. Selecting the 'Show as Ptr' Value Format will restore the pointer back to its original display.

## Changing Multiple Instances of a Variable Within an Array

If a local or global variable is part of an array element, RISCWatch provides the ability to simultaneously change the format, display, or value of each instance of the variable within multiple elements of the array. This is accomplished by selecting a check box on the Change Variable window titled 'Apply to each var. instance at this level' when changes are applied. This check box is used to apply changes to multiple elements and will only appear on the Change Variable window if the selected variable is somewhere part of an array element (and more than one element exists for the array from the perspective of the debugger).

If the check box is selected on a window which contains a Variable Detail groupbox, it will be disabled as long as the check box is selected (and any detail selections will be ignored if the check box is selected when changes are applied).

If display information changes are applied, they will only apply to portions of the variable which have previously been 'revealed' or expanded, whether they are currently visible or not. If a value change is applied, it will only apply to the associated variables which are currently visible on the variable window. Also, when applying a change to multiple instances, a pop-up dialog will appear to verify the action. This underscores the fact that care should be taken when this option is used.

Consider the following variable which is an array of **chars**, with each element value currently displayed as hexadecimal:

---

```
Str_1_Loc: <array[31] of char>
          [0]: 0x49
          [1]: 0x42
          [2]: 0x4D
          [3]: 0x20
          [4]: 0x52
          [5]: 0x49
          [6]: 0x53
          [7]: 0x43
          [8]: 0x57
          [9]: 0x61
         [10]: 0x74
         [11]: 0x63
         [12]: 0x68
```

---

Figure 3-35. Sample **char** Array Display

As a simple example of applying a change to multiple elements at once, we'll first select an element of the array (it doesn't have to be the first). This will bring up the Change Variable window shown in Figure 3-36. Notice the check box above the buttons at the bottom of the window. It appears because the variable we selected was part of an array element. We'll update the display so that the address of each element will be shown, and the value be formatted as ASCII instead of hex. We do this by selecting the appropriate Display Info. and Value Format options just as we would for any variable, along with selecting the checkbox to indicate we wish to apply these changes to each element.

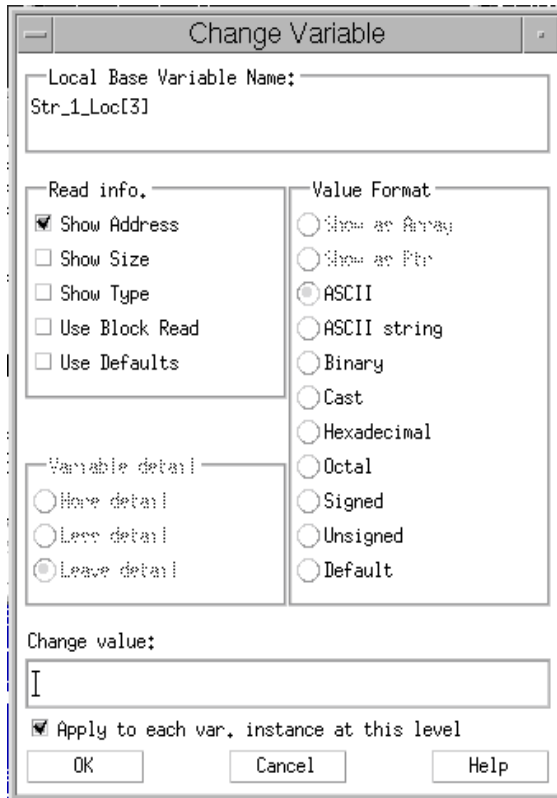


Figure 3-36. Changing Multiple Elements of a Variable Array

Applying these changes results in each element being updated accordingly on the variable screen:

---

```
Str_1_Loc: <array[31] of char>
           [0]: ^I^ (0x49) @0002E248
           [1]: ^B^ (0x42) @0002E249
           [2]: ^M^ (0x4D) @0002E24A
           [3]: ^ ^ (0x20) @0002E24B
           [4]: ^R^ (0x52) @0002E24C
           [5]: ^I^ (0x49) @0002E24D
           [6]: ^S^ (0x53) @0002E24E
           [7]: ^C^ (0x43) @0002E24F
           [8]: ^W^ (0x57) @0002E250
           [9]: ^a^ (0x61) @0002E251
           [10]: ^t^ (0x74) @0002E252
           [11]: ^c^ (0x63) @0002E253
           [12]: ^h^ (0x68) @0002E254
```

---

Figure 3-37. Updated Display of Variable Array

Note that in the example above, we could also have initialized each element of the array by entering a value in the Change Value field. With a value change being applied to multiple instances, a pop-up dialog would first appear to verify the change request. Applying the value change would result in the value of each element of the array being changed.

The robustness of this capability can be fully realized by understanding that it applies to all data types at any level of detail expansion within an array element.

Consider the following pointer formatted to show as array, with the first two elements expanded to multiple levels of detail:

---

```
Rec_Ptr: 0x003FF6F0 <ptr>
  [0]: <struct>
    .Ptr_Comp: NULL <ptr>
    .Discr: +0 STRUCT_0
    .variant: <union>
      .var_1: <struct>
        .Enum_Comp: +0 STRUCT_0
        .Int_Comp: +0
        .Str_Comp: <array> @003FF700
      .var_2: <struct>
        .E_Comp_2: +0 STRUCT_0
        .Str_2_Comp: <array> @003FF6FC
          [0]: "\x0" @003FF6FC
          [1]: "\x0" @003FF6FD
          [2]: "\x0" @003FF6FE
      .var_3: <struct>
    [1]: <struct>
    .Ptr_Comp: NULL <ptr>
    .Discr: +1 STRUCT_1
    .variant: <union>
      .var_1: <struct>
        .Enum_Comp: +1 STRUCT_1
        .Int_Comp: +1
        .Str_Comp: <array> @003FF730
      .var_2: <struct>
      .var_3: <struct>
```

---

Figure 3-38. Sample Multi-Element, Multilevel Variable Display

Selecting the Str\_Comp array variable of the first element brings up the Change Variable window. The check box to apply to multiple instances appears since ultimately this variable is contained within an array element. This time we'll change the array subrange to '0,2', select to show address information, and select the check box to apply the change to each element. Notice that the variable window is updated for each instance of the variable at that level in both Rec\_Ptr array elements.

---

```

Rec_Ptr: 0x003FF6F0 <ptr>
  [0]: <struct>
    .Ptr_Comp: NULL <ptr>
    .Discr: +0 STRUCT_0
    .variant: <union>
      .var_1: <struct>
        .Enum_Comp: +0 STRUCT_0
        .Int_Comp: +0
        .Str_Comp: <array> @003FF700
          [0]: "I" @003FF700
          [1]: "B" @003FF701
          [2]: "M" @003FF702
      .var_2: <struct>
        .E_Comp_2: +0 STRUCT_0
        .Str_2_Comp: <array> @003FF6FC
          [0]: "\x0" @003FF6FC
          [1]: "\x0" @003FF6FD
          [2]: "\x0" @003FF6FE
      .var_3: <struct>
  [1]: <struct>
    .Ptr_Comp: NULL <ptr>
    .Discr: +1 STRUCT_1
    .variant: <union>
      .var_1: <struct>
        .Enum_Comp: +1 STRUCT_1
        .Int_Comp: +1
        .Str_Comp: <array> @003FF730
          [0]: "H" @003FF730
          [1]: "A" @003FF731
          [2]: "L" @003FF732
      .var_2: <struct>
      .var_3: <struct>

```

---

Figure 3-39. Updated Multi-Element, Multilevel Variable Display

This last example will further explain the processing used to determine where changes will be applied if the option is used to change multiple instances of a variable within a complex structure. Selecting the first element of the Str\_Comp variable in the first Rec\_Ptr element brings up the Change Variable Window. We'll initialize each (visible) element of the Str\_Comp array in this and every other (visible) Rec\_Ptr element by putting the value in the Change Value field and selecting the check box to apply to multiple instances.

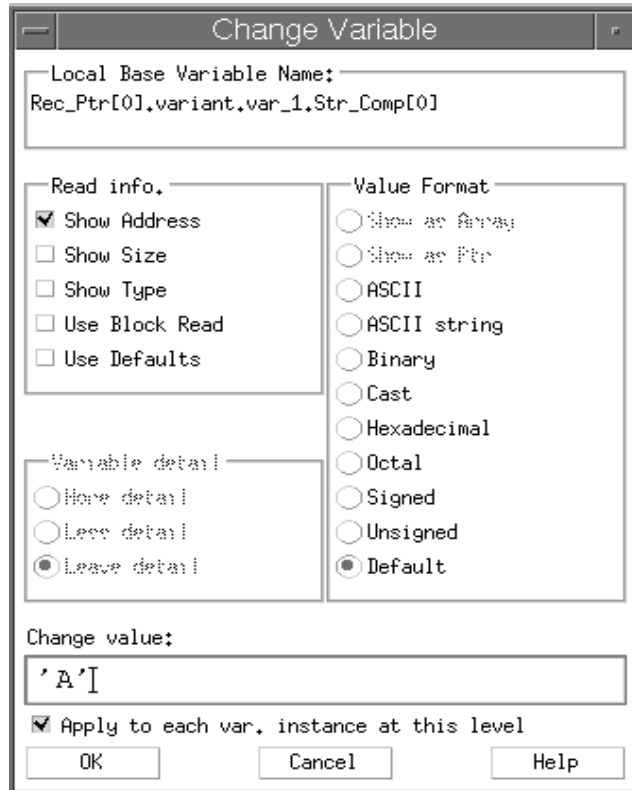


Figure 3-40. Sample Change Value Display

Now, notice the variable's name in the window above: `Rec_Ptr[0].variant.var_1.Str_Comp[0]`. First, all elements of this instance of `Str_Comp` will be changed. Next, going back through the name, the changes will also be applied to all the elements of any other instance of the `Str_Comp` variable. We can see in this example that there is another instance of the `Str_Comp` variable, in the second `Rec_Ptr` element having the name `Rec_Ptr[1].variant.var_1.Str_Comp`.

Applying the change results in the following update:

---

```
Rec_Ptr: 0x003FF6F0 <ptr>
  [0]: <struct>
    .Ptr_Comp: NULL <ptr>
    .Discr: +0 STRUCT_0
    .variant: <union>
      .var_1: <struct>
        .Enum_Comp: +0 STRUCT_0
        .Int_Comp: +0
        .Str_Comp: <array> @003FF700
          [0]: "A" @003FF700
          [1]: "A" @003FF701
          [2]: "A" @003FF702
      .var_2: <struct>
        .E_Comp_2: +0 STRUCT_0
        .Str_2_Comp: <array> @003FF6FC
          [0]: "\x0" @003FF6FC
          [1]: "\x0" @003FF6FD
          [2]: "\x0" @003FF6FE
      .var_3: <struct>
    [1]: <struct>
      .Ptr_Comp: NULL <ptr>
      .Discr: +1 STRUCT_1
      .variant: <union>
        .var_1: <struct>
          .Enum_Comp: +1 STRUCT_1
          .Int_Comp: +1
          .Str_Comp: <array> @003FF730
            [0]: "A" @003FF730
            [1]: "A" @003FF731
            [2]: "A" @003FF732
        .var_2: <struct>
        .var_3: <struct>
```

---

Figure 3-41. Sample Result of Change Value Update

All elements of the each Str\_Comp array are now initialized to the character 'A'. Notice that the elements of the Str\_2\_Comp array are not affected, even though the Str\_2\_Comp array is an array of characters nested the same number of 'levels' from Rec\_Ptr[0]. This is because it is a different variable and the changes were only applied to Str\_Comp variable instances.

It should be apparent that care should be taken when applying value changes to multiple variable instances within complex data structures. Format and Display changes are not destructive, but once the values are changed they cannot be recovered.



## Type Casting a Variable

The Cast format option is available on the Change Variable window to enable dynamic Type Casting of a variable. This feature is particularly useful when debugging code that contains void pointers.

Consider a local variable called 'voidptr' which has been defined in 'C' code as 'void \*voidptr'. A single left mouse click on this variable will bring up the Change Variable window.

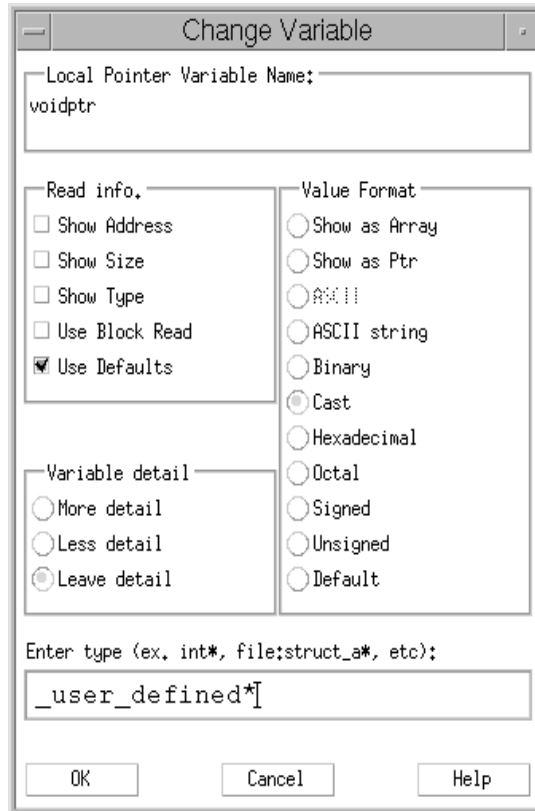


Figure 3-42. Sample Variable Type Cast

Figure 3-42 illustrates the actions needed to cast 'voidptr' to a new type:

- Select the Cast Value Format. Notice the 'Change Value:' field near the bottom of the window is now prompting for a new 'type name'.
- Enter a valid type name followed by any number of '\*'s to indicate pointer indirection. Valid type names include fundamental 'C' types such as 'int', 'long', 'signed char', 'short', etc. In addition, any user defined type name (i.e. structures defined in your 'C' code) can

be used. Figure 3-42 indicates 'voidptr' will be changed to be a pointer to structure '\_user\_defined'.

- Press the 'enter' key or select 'OK'. RISCWatch will search for a type name match using the debug information defined for the source file where the variable was originally defined. If no match is found, an error message is reported in the RISCWatch main window.
- Preceding the type name with a source file name, followed by ':', will direct RISCWatch to search for a name match in the designated file. For example, 'file1.c:\_user\_defined' will force RISCWatch to search the debug information defined for source file 'file1.c'. Valid file names are those names found in the Files window. Note that the directory path of the source file does not need to be specified. The file name may also be enclosed in double quotes.
- The original type definition can be restored by entering '#default' as the new type name.

The Cast format option is available for all variables except arrays. Since an array variable has no value, type casting should be performed on the individual elements of the array. Note that multiple elements of an array can be type caste simultaneously by making use of the 'Apply to each var' check box of the Change Variable window.

## Source Variable Command Support

In addition to the graphical user interface support described above, local and global source variables can be used with memory access commands such as **read**, **write**, **set** and **expr**. Source variables are distinguished from other RISCWatch variables by a leading colon ":", and adhere to the following syntax rules:

```
:[ "filename "]:[&]var_name
```

Where:

- ':' indicates a source variable, as opposed to other RISCWatch variables, such as register names or address locations.
- *filename* is a valid file name, as displayed in the Files Window. This optional field is used to address a specific global variable whose name may conflict with an active local, or another static global variable. If the *filename* option is not specified, RISCWatch will search for *var\_name* in the list of locals defined for the current instruction address (IAR). If not found in the current function, the list of global debug names will be searched.
- Note: the file's directory path does not need to be specified when designating a *filename*.
- *var\_name* is a valid variable name, as seen in the Locals or Globals Window. Normal 'C' language naming conventions are used to identify a specific data member. For example ":ptr->member1", ":ptr[0].member1", ":structa.structb.member2" are all valid

name constructs. The variable name must be defined within the active function (defined by the IAR) or within the active list of globals.

- **'&'** is a request for the address of a variable, similar to the 'C' language definition. If **'&'** precedes a *var\_name* that is assigned to a register, the address can not be determined and an error is reported.
- **'\*\*'** is a request for the value of a *var\_name* that resolves to a 'C' pointer. If **'\*\*'** precedes a non pointer variable, or the variable is a pointer to a structure, an error is reported.

Typical examples:

```
read :i# reads src variable 'i'  
write :ptr[100] 14# write decimal 14 at the 101st element of 'ptr'  
set : "file.c":glob1.a = 0x13# write hex 13 to variable 'glob1.a'  
read :&test[2].a R1# write R1 with address of 'test[2].a'  
expr : **ptr1# display value of '**ptr'
```

---

## Reading and Writing Memory

The Hardware | Memory pulldown on the Main window provides a number of different ways to view memory. They allow the user to view specified memory contents in hex, ASCII, or disassembled instruction formats. The following page references are good sources of information:

- "ASCII Memory Window" on page 3-107
- "Custom Memory Window" on page 3-109
- "Memory Coherency Window (JTAG Targets Only)" on page 3-104
- "Cache Windows (JTAG Targets Only)" on page 3-112
- "Translation Lookaside Buffer Window (Applicable Processors Only)" on page 4-14
- "Load Memory Window" on page 3-64
- "Save Memory Window" on page 3-113

Some windows also provide the ability to alter memory contents.

Memory can also be viewed and altered using the **read** and **write** commands from the command line on the Main window.

**Note:** Be aware that there are situations where changing the content of an individual memory location may result in sections of adjacent memory being read. If data is written to an address, and that address corresponds to an address which is contained in a Memory or Asm Debug window which is currently up, a memory region the size of the memory displayed in these windows will be read from the target. Similarly, if the address of changed memory corresponds to a portion of an individual memory element existing on any user-defined window, an amount of memory equal to the size of the memory

element will be read (for example, if a byte-sized memory element at address 0x00000001 is written, and another user-defined memory region is defined with four word size elements starting at address 0x00000000, one word of data will be read from address 0x00000000 in this case).

## Memory Coherency Window (JTAG Targets Only)

The Memory Coherency window is used to control data and instruction cache updating during memory reads and writes. This window is displayed by selecting the Memory | Coherency option of the menubar's Hardware pulldown choice.

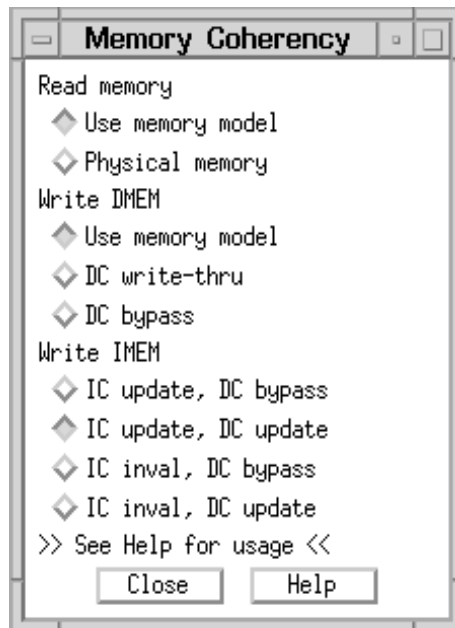


Figure 3-43. Sample Memory Access Window

If caching is disabled via the appropriate hardware registers (DCCR/ICCR for PowerPC 400Series, HID0 for PowerPC 6xx/7xx), reads and writes from/to memory will directly reflect the contents of physical memory.

If the processor is configured to control data and instruction caching, a memory model is said to have been established for how the data and instructions will be accessed. Once a memory model has been established, reads and writes to/from memory will provide data and/or instructions that are a combination of information from the caches and memory.

Using the read memory options, it is possible to force reads to use your memory model (a combination of cache and memory information) or to read directly from physical memory (by bypassing the data cache).

When a memory model is used to control data caching, the Memory Coherency window allows control over how the data is written to the data cache and memory. To allow the processor to manage data coherency between the data cache and memory, select the memory model option. To force memory writes to immediately update the data cache and memory contents, select the write-thru option. To force memory writes to update physical memory only, and not the data cache, select the bypass option.

Similarly, an instruction cache (IC) memory model can be controlled with the options in the Memory Coherency window. The update options should be selected to force instruction memory writes to update both physical memory and the instruction cache. The invalidate options are used to force instruction memory writes to update physical memory while marking the associated addresses as invalid in the instruction cache.

For instruction memory writes, the data cache (DC) options are used to indicate whether instruction memory writes are to update the data cache or not. Select the bypass option to indicate that instruction memory writes are NOT to be written to the data cache. Selecting the update option forces instruction memory writes to update the data cache as well.

**WARNING:** The DC bypass option should be used with caution when data caching is enabled. This option is used to force the data memory writes to update physical memory without updating the data in the data cache. This mechanism essentially overrides the memory model that would be set up using the registers which control caching. Data written to physical memory using this option could be overwritten by “dirty” data in the cache that had not yet been written out to memory.

Following is a description of the Memory Coherency window options and exactly how they function:

<b>1. Write DMEM</b>	<b>Coherency</b>	<b>D-Cache</b>	<b>I-Cache</b>	<b>Physical Memory</b>
Use memory model	Yes	Note 1	No	Note 2
DC write-thru	Yes	Note 1	No	Yes
DC bypass	No	No	No	Yes
<b>2. Write IMEM</b>	<b>Coherency</b>	<b>D-Cache</b>	<b>I-Cache</b>	<b>Physical Memory</b>
IC update DC bypass	Note 3	No	Note 4	Yes
IC update DC update	Yes	Note 1	Note 4	Yes
IC inval DC bypass	Note 3	No	No (Note 5)	Yes
IC inval DC update	Yes	Note 1	No (Note 5)	Yes

**Notes:**

1. D-Cache updated if enabled (via DCCR for PowerPC 400Series, HID0 for PowerPC 6xx/7xx)
2. Physical memory written if D-Cache disabled (via DCCR for PowerPC 400Series, HID0 for PowerPC 6xx/7xx)
3. Coherent if D-Cache disabled (via DCCR for PowerPC 400Series, HID0 for PowerPC 6xx/7xx)
4. I-Cache updated if enabled (via ICCR for PowerPC 400Series, HID0 for PowerPC 6xx/7xx)
5. I-Cache line invalidated

**Note:** RISCWatch will ignore the settings in the Memory Coherency Window during a memory download (load command) and default to an I\_Cache invalidate, D\_Cache flush model. This model achieves the best download rate while maintaining coherency of the system under test.

## ASCII Memory Window

The ASCII Memory window allows memory to be read, altered and written as four-byte data words or as ASCII text. This window is displayed by selecting the Memory | ASCII option of the menubar's Hardware pull-down choice. What follows is a description of this window's functionality.

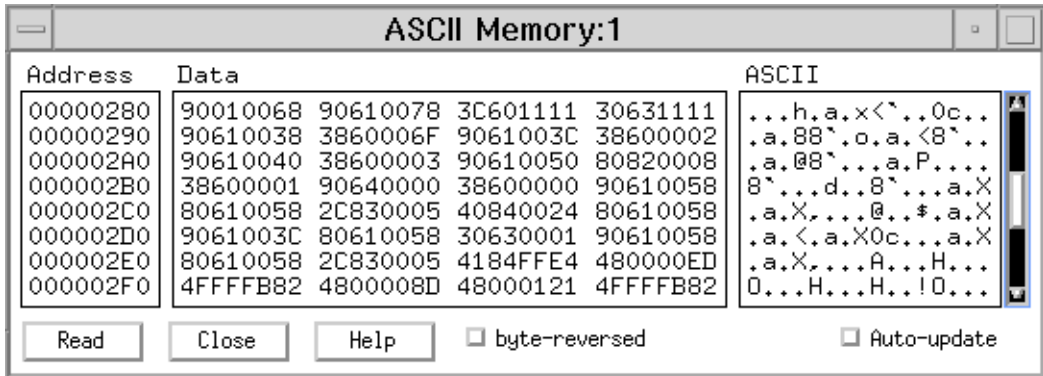


Figure 3-44. Sample ASCII Memory Window

- **Scroll Bar**

Clicking on a vertical scroll arrow alters the display address by one line or opcode. Clicking on the area between a vertical arrow and the current scroll position alters the display address by one screen's worth of data. To display a given address, use the address entry schemes described in the Data area and Address entry sections.

The page up and page down feature may also be accessed via the keyboard Page Up and Page Down buttons.

- **Address area**

The address area of the ASCII Memory window is used to display data anywhere within the configured range of the processor. The address area is located at the far left under the Address: heading. To display any part of memory, simply use the mouse to place the cursor anywhere within one of the address values, type in the desired address and press the Enter key.

- **Data Area**

The data area of the ASCII Memory window is used to display data read from the processor as well as alter this data so that it may be written back. There are four data values per display line with each value displaying four bytes of data.

To alter any of these data values, simply use the mouse to place the cursor anywhere within one of the data values, type in the desired data, and press the Enter key to write the

data value to the processor memory. Changed data will not be written to the processor unless the cursor is in the data value that was changed when the Enter key is pressed. If data is mistakenly entered into a data field that is not to be written to memory, simply click on the Read button to refresh the displayed data.

- **ASCII area**

The ASCII area of the ASCII Memory window is used to display data read from the processor as well as alter this data so that it may be written back. The ASCII area is located in a column along the right hand side of the window. Each ASCII line contains sixteen (16) ASCII characters that represent the data bytes in the data area.

To alter any of these data values, simply use the mouse to place the cursor in any one of the ASCII character areas, type in the desired data and press the Enter key to write the ASCII data to the processor memory. Changed data will not be written to the processor unless the edit cursor is in the data line that was changed when the Enter key is pressed.

- **Read button**

The Read button is used to read the processor memory to refresh the contents of all currently displayed data and address fields. Use this button to force a refresh of displayed data or to remove the contents of a partially edited data or address field which has not been written back to the processor.

RISCWatch allows multiple instances of the ASCII Memory screen to be used simultaneously. The instance number is included after the ':' in the title bar. Each time the ASCII Memory screen is selected via the Memory pulldown or the "**window ascii**" command is issued, a new instance of the window will be created.

- **Byte-reversed checkbox**

The byte-reversed checkbox is used to enable the byte-reversal of all values displayed in the Data and ASCII areas. A memory value displayed normally as 0x12345678 would appear as 0x78563412 when byte-reversed. If byte-reversed is enabled, any value which is entered by the user will also be byte-reversed in the same manner before it is written to memory.



## Custom Memory Window

The Custom Memory window allows memory to be written or read in several different formats and word sizes. This window is displayed by selecting the Memory | Custom option of the menu bar's Hardware pull-down choice. What follows is a description of this window's functionality.

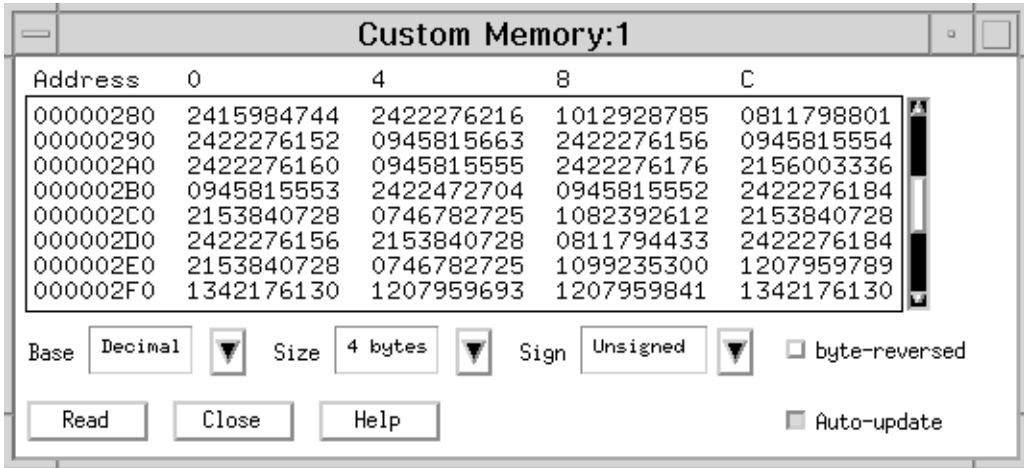


Figure 3-45. Custom Memory Window

- **Address Area**

The address area of the Custom Memory window is used to display data anywhere within the configured range of the processor. The address area starts at the leftmost column under the Address heading. To display any part of memory, simply use the mouse to place the cursor anywhere within one of the address values, type in the desired address and press the Enter key.

- **Data Area**

The data area of the Custom Memory window is used to display data read from the processor as well as alter this data so that it may be written back. The data area consists of all the values displayed to the right of the address area and underneath the numeric headings which are used to help in determining on-screen addresses.

To alter any of these data values, simply use the mouse to place the cursor anywhere within one of the data values. The Input Area will appear at the bottom of the window to allow new values to be entered. When the input area is displayed, the Custom Memory window will be locked until the enter key is pressed.

- **Scroll Bar**

Clicking on a vertical scroll arrow alters the display address by one line's worth of data. Clicking on the area between a vertical arrow and the current scroll position alters the

display address by one screen's worth of data. To display a given address, use the address entry scheme described above in the Address Area section.

The page up and page down feature may also be accessed via the keyboard Page Up and Page Down buttons.

- **Base Selection**

The base selection button is used to select the radix that the data values will be displayed with in the data area. To change the currently selected base, click on the arrow button with the mouse and select the desired base. Once done, the data will be refreshed and displayed in the newly selected base.

- **Size Selection**

The size selection button is used to select the word size of the data values to be displayed in the data area. To change the currently selected size, click on the arrow button with the mouse and select the desired size. Once done, the data will be refreshed and displayed in the newly selected size.

- **Sign Selection**

The Sign selection button is used to select the sign used to display data values in the data area. To change, the currently selected size, click on the arrow button and select the desired sign. Once done, the data will be refreshed and displayed in the newly selected sign.

The sign selection button is only enabled when the currently selected base is decimal.

- **Read button**

The Read button is used to read the processor memory to refresh the contents of all currently displayed data and address fields. Use this button to force a refresh of displayed data or to remove the contents of a partially edited address value.

- **Input Area**

The input area of the Custom Memory window is used to input data values that are to be written to processor memory. This area appears when a value has been selected from the data area by clicking on it with the mouse. Once done, the input area will appear between the Help and Auto-update buttons. The value to be written to memory is typed in and followed by the Enter key. To cancel this input operation, press the Esc key.

**Note:** Once the input area is displayed, the window controls will be locked to prevent activation until the input operation is completed or cancelled.

Values typed into the input area can be specified in ASCII, binary, decimal, hexadecimal and octal radices. The following rules are used in the specified order to determine the radix being specified:

- If the currently selected base is ASCII, an ASCII value is assumed.
- If the value starts with '0x', '0X', 'x' or 'X', a hexadecimal value is assumed.
- If the value starts with 'd', '+' or '-', a decimal value is assumed.
- If the value starts with '0', an octal value is assumed.
- If the value starts with 'b', a binary value is assumed.
- If the value starts with '"', an ASCII value is assumed.
- If the value starts with a decimal number, a decimal value is assumed.
- All other values are assumed to be ASCII.

When the selected base is not ASCII, an ASCII string can be specified by enclosing it in quotation(") marks so that it will not be confused with data in another radix. When the selected base is ASCII, an ASCII string is assumed so no quotation marks are necessary.

When the selected base is not ASCII, input values are written to the processor as a single value in the currently selected word size. In other words, if an input value is specified which is numerically larger than that which fits in the currently selected word size, the "extra" data is disregarded. The exception to this rule occurs when an ASCII string is specified. In this case, the entire string is written to the processor as a series of one byte values.

RISCWatch allows multiple instances of the Custom Memory window to be used simultaneously. The instance number is included after the ':' in the title bar. The Custom Memory window can be resized vertically but will ignore horizontal size changes.

- **Byte-reversed checkbox**

The byte-reversed checkbox is used to enable the byte-reversal of all values displayed in the Data and ASCII areas. A memory value displayed normally as 0x12345678 would appear as 0x78563412 when byte-reversed. If byte-reversed is enabled, any value which is entered by the user will also be byte-reversed in the same manner before it is written to memory.

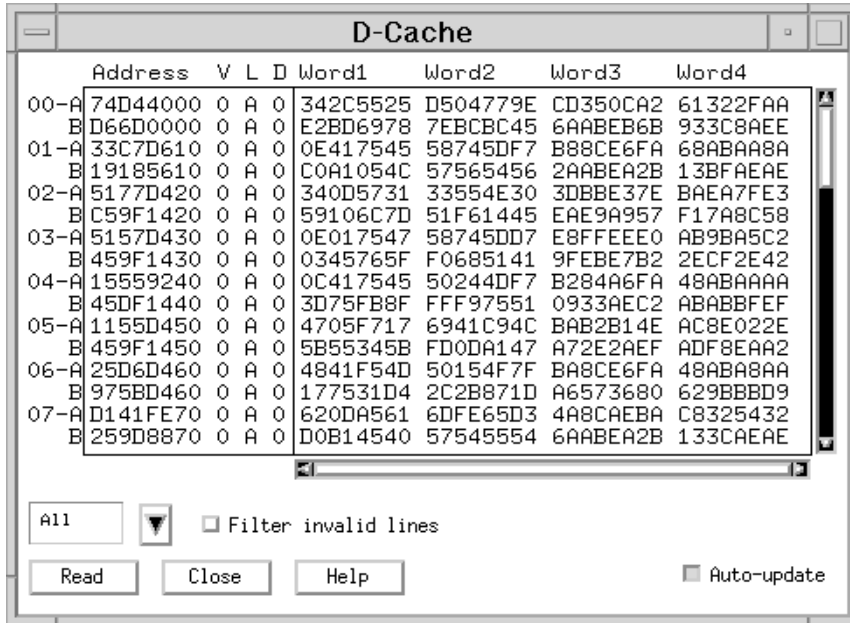


Figure 3-46. Sample Data Cache Window

## Cache Windows (JTAG Targets Only)

The Data, Instruction and L2 Cache windows are used to read and display the contents of the processor caches.

The processor caches are displayed one way (or side) at a time, or all together. The pulldown in the lower left corner is used to change the currently displayed way(s). The vertical scroll bar on the right is used to page up and down the available cache lines for the displayed way(s).

For the Data Cache window, the following fields are shown:

Set	Set number (congruence class) down the left hand side
Way	A, B, C, (or 0, 1, 2) etc. next to the Set number
Address	Address tag
V	Valid bit
L	LRU (Least Recently Used) line in set
K	Lock bit (401 Core+ASIC processors only)
D	Dirty bit
Word N	32-bit data cache word N

For the Instruction Cache window, the following fields are shown:

Set	Set number (congruence class) down the left hand side
Way	A, B, C, (or 0, 1, 2) etc. next to the Set number
Address	Address tag
V	Valid bit
L	LRU (Least Recently Used) line in set
K	Lock bit (401 Core+ASIC processors only)
Word N	32-bit instruction cache word N

**Note:** For these cache displays, the address tag is always displayed normalized to bit 0 (MSB).

The Way Control at the bottom left corner of the screen allows the visible way(s) to be changed.

The Read button is used to force a read of the processor cache and display the latest contents.

The Close button is used to remove this window from the screen.

The Filter check box is used to filter the cache lines so that only lines which are presently marked as Valid are displayed. If this option is selected, a pop-up window will appear informing the user that enabling this feature may result in a performance degradation due to the large amount of data which must be processed. On some processors it may be desirable to disable the Auto-update feature when Filter is enabled to reduce update overhead. In this case, the Read button may be used to force the window contents to be updated.

---

## Save Memory Window

The Save Memory window provides target memory read capabilities using the file format options defined for the Save command. The window is displayed by choosing the 'File' pulldown on the RISCWatch Main window, then selecting Save | Memory.

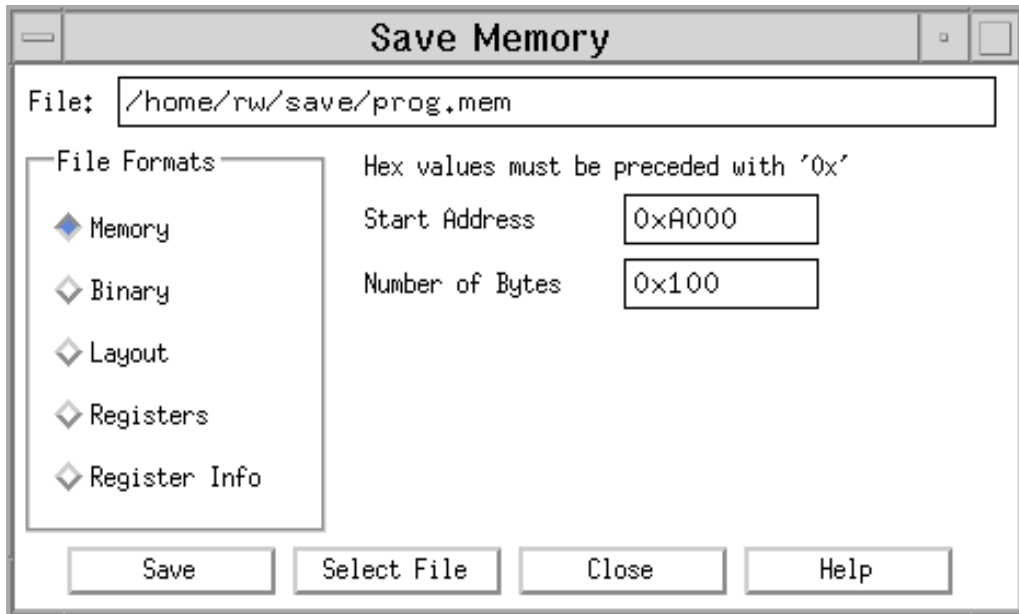


Figure 3-47. Save Memory

The Save Memory window consists of the following fields:

- **File**

This field indicates the file name to be created. The file name can be altered by choosing the “Select File” button at the bottom of the window or by directly typing in the field provided. When the Save pushbutton is pressed, RISCWatch will prompt the user for permission to erase the file if the file currently exists.

- **File Formats**

This group box provides a list of supported file formats. See “save,” p. 5-108 for a description of the supported file formats. Choosing a particular file format will result in the enabling, or disabling, of the remaining input areas of the window.

- **Start Address**

This field is enabled for Memory and Binary format selections and indicates the initial target address to use when reading target memory.

- **Number of Bytes**

This field is enabled for Memory and Binary formats and indicates the number of memory bytes to read.

- **Save**

This pushbutton is used to execute the selected **Save** command. The RISCWatch main window will indicate the success or failure of the operation.

- **Select File**

This pushbutton presents a standard dialog window that allows the user to supply a file name. The file chosen will be used to save the requested information.

- **Close**

This pushbutton removes the Save Memory window from view. A subsequent display of the Save Memory window will present the field settings that were in existence when the 'Close' button was pressed.

- **Help**

This pushbutton will present any available help topic for this window.

---

## Reading and Writing Registers

The Hardware | Register pulldown on the Main window provides the ability to view and update the architected registers of the target chip. They are divided into classes:

- General Purpose Registers (GPRs)
- Special Purpose Registers (SPRs)
- Device Control Registers (DCRs): 400Series only
- Segment Registers (SRs): PowerPC 6xx/7xx only
- Floating Point Registers (FPRs): processors with FPUs
- ASIC Registers (ASICs): defined in user-created PRD files

See “Register Windows” on page 3-115 and “Register Field Windows” on page 3-117 for detailed descriptions of the register windows. Register Field windows are used to manipulate individual fields of selected registers. These provide a bit breakdown of the selected register divided into logical field groupings applicable to the register.

Registers can also be viewed and altered using the **expr**, **read**, **set**, and **write** commands from the command line on the Main window.

## Register Windows

Register windows are used to read, display, modify and write-back processor registers. Register windows are broken up into classes based on the types of registers they contain. Current register windows include General Purpose Registers (GPR), Special Purpose

Registers (SPR), Device Control Registers (DCR: PowerPC 400Series only), Segment Registers (SR: PowerPC 6xx/7xx only) Floating Point Registers (FPR: processors with FPUs) and ASIC Registers (ASIC: defined in user-created PRD files). To bring up a particular register window, use the Hardware|Register pulldown of the Main window menubar.

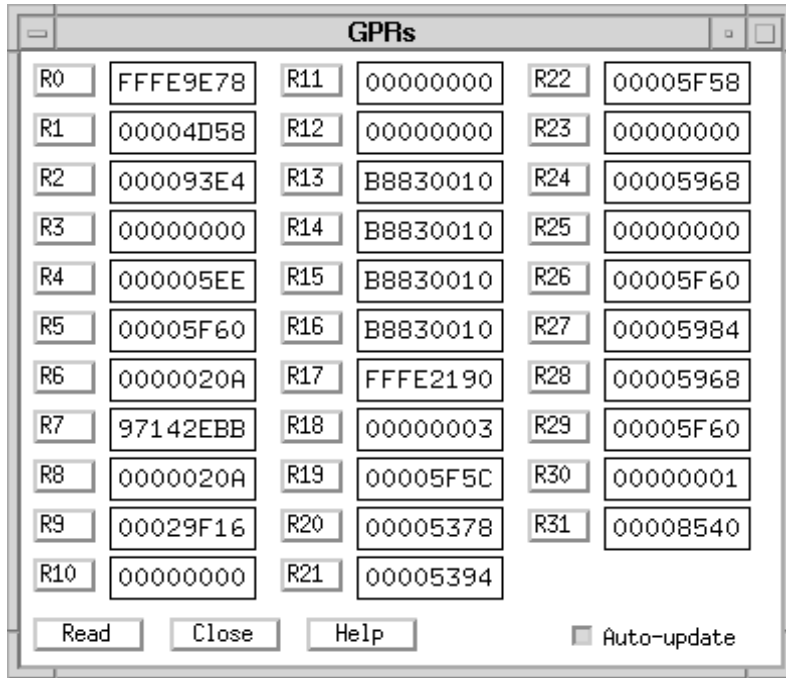


Figure 3-48. Sample Register Window

**Note:** RISCWatch will only display an ASIC selection for the Register pulldown if a custom PRD file is being used and the ASIC registers defined in the PRD file followed the prescribed naming convention for such registers. Refer to the **REG Definitions** section of “Processor Configuration File (PCF)” for register naming details.

**Note:** If an ASIC Register window is selected, more than one physical window may appear if RISCWatch determines that there are too many registers for a single window to hold (as dictated by the physical constraints imposed by your monitor’s resolution). If this is the case, the multiple windows created can be thought of as one logical entity. Therefore, pressing the Read or Close buttons on one window applies to all of them (with the same ASIC register prefix). If a specific subset of ASIC registers is desired for read or write operations, simply create a User-Defined window to access them.

A register window is split into two or more columns with each column containing a push button and register edit field. The push button contains a register name while the edit field contains its value. The push button is used to bring up a register field window for that



particular register (if it has a field definition). Use the mouse to press the push button and bring up its register field window. If it has no field definition, an error message will be displayed.

To edit a register value, use the mouse to place the edit cursor in the appropriate field and enter a new hexadecimal value for the register. This new value will not be written to the processor unless the edit cursor is in the field and the Enter key is pressed.

To refresh the contents of all register fields at any time, use the mouse button to click on the Read button located at the bottom of the window.

## Register Field Windows

Register field windows are used to read, display, modify and write-back processor registers. To bring up a particular register field window, use the Hardware|Reg Fields pulldown of the Main window menubar.

A register field window is composed of one or more registers. Each register definition in the window takes up one display line. This line is composed of the register name, a register value field and register field value fields.

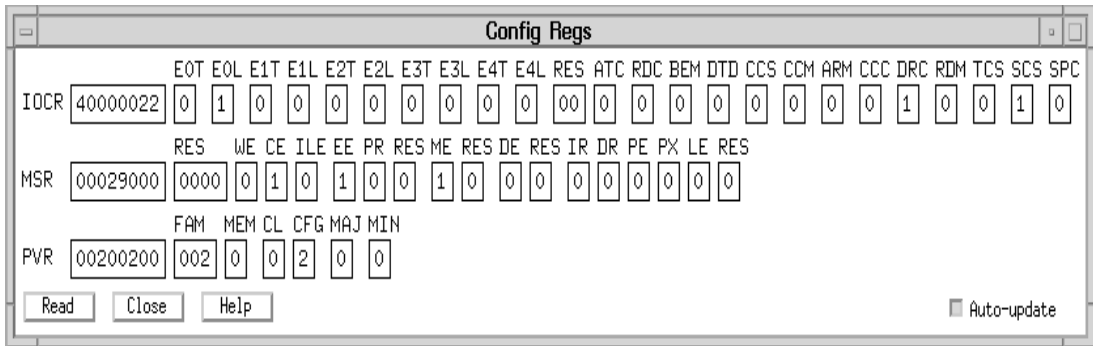


Figure 3-49. Sample Register Field Window

The register value field contains the full data value for the register and should track to the value of the register in its Register window. This field may be edited and written to the processor just like its counterpart in the Register window.

The register field value fields are a series of fields that represent the individual logical bit groupings for that register. Each field value contains a heading which matches the register bit definitions in the PowerPC User's Manual for that specific processor. The heading is a two or three character mnemonic derived from the field's name.

For each register field, the appropriate bits are extracted from the register value, shift to bit zero to normalize them, and then displayed in their appropriate field. Such a display allows these field values to be compared directly with the values in the User's Manual for that register, edited and written back to the processor.

Register or register field values may be modified by using the mouse to place the edit cursor in the appropriate input field and then typing new hexadecimal values. This new data will not be written to the processor unless the Enter key is pressed.

For register fields which are only one bit in size, the mouse may be used to toggle the current bit value and write it back to the processor. To do so, simply use the mouse to double-click over the single-bit field.

Whenever data is changed and written back to the processor, the appropriate data fields in the window will be updated to reflect this latest value. If the register value is changed and written, the field values will be updated accordingly. Likewise, if one or more register field values are changed and written, the register value will be updated.

To refresh the entire window's contents with the latest processor data, simply use the mouse to click on the Read button. This will read the latest data value for all the registers in the window and update the display accordingly.

**WARNING:** Any data that has been changed in the window and not written back to the processor will be lost!

---

## User-Defined Windows

User-Defined windows allow a RISCWatch user to create windows containing customizable register, register field, memory, disassembly, and button entries. Using a simple syntax, ASCII files are created to define the contents of a user-defined window.

**Note:** Look for the file RWPPC.WDF in the install directory for an example of a User-Defined window file. We recommend that you make a copy of this file and then work off of this copy as you experiment with defining your own windows.

### File Syntax

The file used to describe a user-defined window is a simple ASCII file that is created with a text editor. The filenames for such files usually, but do not have to, end in .wdf (window descriptor file).

The file is composed of simple keywords and may contain comments. The keywords used to define the contents of user-defined entries are BUTTON, BUTTONDEF, CMDLINE, DIS, FORMAT, HEADING, LABEL, MEM, REG, REGFLD, SEPARATOR, STATUSBAR, and TITLE. These keywords and their usage are explained in the sections that follow.

The general syntax rules are as follows:

1. The following keyword definitions must appear on a line by themselves  
BUTTONDEF, CMDLINE, FORMAT, HEADING, SEPARATOR, STATUSBAR, TITLE
2. The following keyword definitions may only appear once in the file  
CMDLINE, STATUSBAR, TITLE

3. Except for those listed above, multiple keywords may be listed on a single line.
4. Comments in this file are defined by starting a line with a '#'. Comments may also appear at the end of a line.
5. Blank lines are ignored except where they are used to mark the end of a BUTTON-DEF definition.

## Keyword Definition/Syntax

- **BUTTON - User Defined Button Placement Entries**

Buttons corresponding to user defined functions can be placed in the user defined window. The BUTTON keyword is followed by a button identifier that was previously defined using the BUTTONDEF keyword

- **BUTTONDEF - User Defined Button Function Definition**

Users can define their own buttons and corresponding function for inclusion on the window. The BUTTONDEF keyword is followed by a button id (to be used to place the button using the BUTTON keyword), followed by the button name (to appear on the screen) enclosed in quotation marks. On subsequent consecutive lines (no blank lines), valid RISCWatch commands can be entered, one per line. A blank line ends the button definition. Whenever the button is pressed on the user defined window, those actions listed in the definition will be executed.

- **CMDLINE - Command Line Entry**

A user defined window may also contain a command line. This command line allows RISCWatch commands to be entered from the user defined window in the same manner they are normally entered from the main window. Only one entry is allowed per window definition, and it can be the only command on that line. Regardless of where the definition keyword is located, the command line will appear at the bottom (or next to the bottom if a status bar is defined) of the user defined window. If the window locks its controls, the only commands which are valid to be entered are STOP and QUIT.

- **DIS - Disassembly Entries**

Disassembly entries are used to place disassembly text in the user-defined window. The DIS keyword is followed by the address of memory to be disassembled, which is followed by the number of words to be displayed.

- **FORMAT- Format Entries**

Format entries are used to define the width of user interface controls so that effected controls conform to a common size which ensures that these controls are laid out in "pretty" columns.

By default RISCWatch will select a default FORMAT size which doesn't always result in nice, orderly columns. Using FORMAT allows the user to override these default rules and select values which results in user interface components which are laid out in an orderly fashion and therefore are easier to interact with.

The **FORMAT** keyword is followed by either **MEM**, **REG** or **REGFLD**, the equal (=) sign, and a number which represents the request size of the effected control. For **REG** and **REGFLD** entries, this number sets the width of the push button (in characters) containing the register name. For **MEM** entries, it sets the number of columns in the displayed memory pane. Once set, a **FORMAT** selection will remain in effect until overridden by another. Setting the number to 0 will cancel the user selection and return to default sizing of controls.

- **HEADING - Window/Section Headings**

The user-defined window is given a title by using the **HEADING** keyword followed by the desired window title. The **HEADING** keyword can also be used to add titles to different sections within the window as well.

- **LABEL - Label Item Entries**

Labels can be placed throughout the window. The **LABEL** keyword is followed by the desired label text enclosed in quotation marks.

- **MEM - Memory Entries**

Memory entries are used to place memory data in the user-defined window.

A memory entry consists of the memory keyword **MEM**, followed by the address of memory to be displayed, followed by the number of bytes in each word, followed by the number of words to display.

The leftmost field of each memory line is the address field. Placing the cursor in an address field and pressing **Enter** will result in the amount of memory displayed in the line being read starting at the specified address. The address can also be changed by typing over the current address and pressing **Enter**. This will also result in a memory read of an entire line's worth of data.

The contents of an individual memory element can be written by typing in the new value and pressing **Enter**. This will only write an amount of memory equal to the size of the individual memory element (i.e., word, half-word, or byte).

- **REG - Register Entries**

Register entries are used to place registers in the user defined window. Each **REG** keyword is followed by any valid processor register name. Multiple **REG** name pairs are allowed on a single line.

- **REGFLD - Register Field Entries**

Register field entries are used to place register field values in the user defined window. The **REGFLD** keyword is followed by the name of a valid processor register that has a valid field defined.

- **SEPARATOR - Section Separator Entries**

The user defined window can be separated into various sections to improve readability and clarity. The SEPARATOR keyword provides a graphical horizontal separator between window sections. This keyword must be the only keyword on the line.

- **STATUSBAR - Status Bar Entry**

The user defined window may also include a status bar if desired. Specifying the STATUSBAR keyword will include a status bar with similar content to that of the main window status bar. This keyword must be the only keyword on the line. Regardless of where the keyword appears within the file, the status bar will be located on the bottom of the user defined screen.

- **STOP - Processor Stop Button**

In order to stop the processor, a special button can be placed in the user defined window. Even though the window may lock its controls while the processor is running, this button will remain available for use ensuring that the processor can be stopped.

- **TITLE - Window Title**

A title definition allows you to provide a custom name for a user-defined window. By default, RISCWatch assigns the name "User-Defined" to each user-defined window which is loaded. While this may be fine for a single user-defined window, if you decide to load several at one time, it is easier to manage them if they are each given unique titles.

To define a customer name for a window, simply follow the TITLE keyword by the name of the window between quotation marks (").

## Creating the Window

A user-defined window is created by using the User-Def Win entry of the User-Defined menu of the Window pull-down. This will display a file selection dialog allowing the window descriptor file to be chosen. Once a file has been selected, it will be read by RISCWatch. If no errors were detected, the user-defined window will be created for use. Alternatively, the window command can be used to bring up the window. The syntax is "WINDOW UDW *filename*", where *filename* indicates the fully qualified name of the user defined window definition file.

## Example

The following example illustrates the use of the user-defined window file syntax:

```
# Provide a custom window title
TITLE "Custom #1"

# Miscellaneous heading
LABEL "The window they forgot to design for me!"
SEPARATOR
```

```

# Button Definition Section
BUTTONDEF Button1 "Read R1-R3"
read R1
read R2
read R3

BUTTONDEF Button2 "Load & Run"
load file demo.elf
bp set in main
run

# Window Layout section
HEADING "The Window they forgot to design for me!"
SEPARATOR
HEADING "ASIC Registers"
REG ASIC01 REG ASIC02
SEPARATOR
LABEL "Stack Regs : " REG R0 REG R1
REG R14 LABEL "<- Key Reg for my application"
SEPARATOR
REGFLD MSR
SEPARATOR
MEM 0xC000 4 4
SEPARATOR
DIS 0x0000A000 8
SEPARATOR
BUTTON Button1 BUTTON Button2
SEPARATOR
CMDLINE
STATUSBAR

```

When coded as above, the window file will produce the window, Figure 3-50.

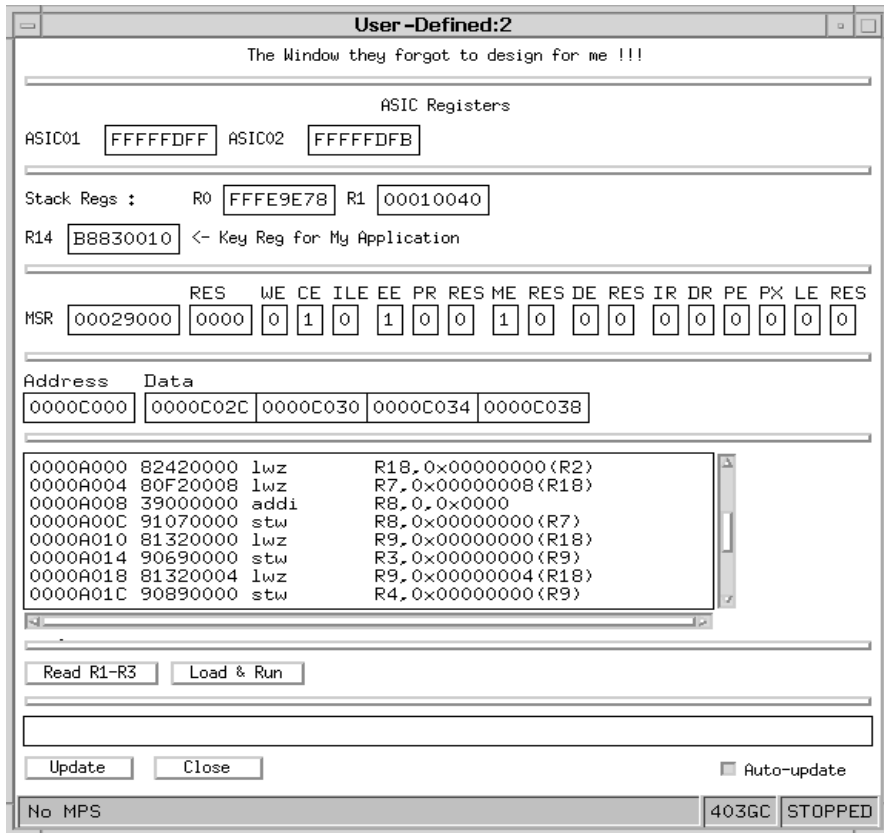


Figure 3-50. Sample User-Defined Window

A sample window descriptor file is included with the software installation of RISCWatch and is titled **rwppc.wdf**.

## Command Files

RISCWatch command files are ASCII text files which contain commands that are understood by RISCWatch. Various commands allow for access to almost all of RISCWatch's processor functionality. These command files are designed to be human-readable and therefore can contain comment and blank lines.

The commands contained in a command file are the same as those commands that can be typed into the command line of RISCWatch's Main window. See the following sections for a list of available commands and their usage.

## Using Shell Scripts to Execute Command Files

By using a shell script, several command files could be generated, one for each piece of logic or function to be tested, and then the entire suite could be called from within a single script file and allowed to run overnight. At some later time when the test suite was completed, the output files from the test suite would be checked to verify the status of each test file run.

## Startup Command File

RISCWatch allows a pre-defined command file to be executed every time the program is brought up in graphical user interface mode.

This command file, identified with the **STARTUP\_FILE** environment variable in **rwppc.env**, may be used to perform a series of commands which would normally be entered on the command line whenever RISCWatch is started. This enables a user to set up the debugging environment and/or specific processor facilities.

The startup command file is searched for using the following rules:

- If the file name is qualified (directory path indicated), the file search is performed using the specified directory only.
- If the name is not qualified, the file search is performed using the directory paths designated with the RISCWatch **SEARCH\_PATH** environment variable. If not found, the current directory is searched.

This search scheme allows individuals to create their own startup command file by placing it in their own directories. This also allows one startup command file to be placed in a common directory so that everyone will execute it whenever RISCWatch is started.

**Note:** Commands in the startup command file are executed after the environment file is read. Therefore, search paths set with the **SEARCH\_PATH** environment variable will be overridden by **srchpath** commands in the startup command file.

## Special Command File Commands

The following commands can only be used from within a command file:

<b>delay</b>	Delays command file execution for the specified number of seconds.
<b>end</b>	Forces the immediate termination of the command file.
<b>parms</b>	Specifies a parameter variable list for the command file. See “Command File Parameters” on page 3-128 for details.
<b>print/fprint</b>	Takes the contents of the command after the print/fprint keyword and prints them in the host window/print file. See the <b>fprint</b> command for details and available formatting options.



## Blank Lines and Comments in Command Files

To make the command files more readable, blank lines can be placed anywhere in a command file. Comments can also be added to help document the command file.

The # character indicates the beginning of a comment on a line. The # character can be placed anywhere on a line. Everything after the # character on a line is taken as a comment. Comments do not carry over onto the lines that follow them. An example command file that uses comments is shown below:

```
# This is a sample command file
# In this command file are examples of comments that start
# in column 1 and comments that start after a command on a line.
stop # This command stops the processor
run # This command starts the processor running
```

## Command File Programming

The following programming logic and flow commands are available for use in RISCWatch command files. These logic and flow commands are not understood by RISCWatch's command line interface and are therefore only valid in command files.

- **if-then**

```
if (expression)
    block
endif
```

- **if-then-else**

```
if (expression)
    block
else
    block
endif
```

```
if (expression)
    block
elseif (expression)
    block
endif
```

```

if (expression)
    block
elseif (expression)
    block
else
    block
endif
• while
while (expression)
    block
endwhile
• do-while
do
    block
dowhile (expression)
• goto
LABEL label_name
    block
GOTO label_name

```

Where:

block	Represents one or more RISCWatch commands.
expression	Composed of either a mathematical or logical expression. See the <b>set</b> command for a detailed description of RISCWatch expression syntax. Most expressions take the form (argument operator argument) Arguments can be references to registers, register fields, memory values, immediate values or created/assigned variables. The operator(s) used in an expression are dependent upon the arguments used. Examples of operators in a mathematical expression are + and - while examples of

operators in a logical expression are == and >. Arguments can also be predefined special expressions as described below.

Regardless of whether a mathematical or logical expression is specified, RISCWatch will evaluate the expression accordingly. A logical expression will always evaluate to either a 1 (TRUE) or 0 (FALSE). A mathematical expression will evaluate to a resultant mathematical value and this value will indicate FALSE if equal to zero and TRUE all other times.

label\_name                      The label can consist of any characters but MUST begin with a letter.

## Command File Special Expressions

Several special expressions can be used by themselves in an if, while, or do expression. For each expression, RISCWatch determines its state and returns a Boolean value used to evaluate the expression. These special expressions include:

proc_running	Returns 1 if the processor (JTAG) or process (non-JTAG) is in the run state, else returns 0. This expression is useful in detecting a processor stop request failure which may occur on a <b>run</b> timeout command. For all other situations, this expression will return a value of 0 (since command file expressions are evaluated only when the processor is stopped).
proc_stopped	Returns 1 if the processor (JTAG) or process (non-JTAG) is in the stopped state, else returns 0. This expression is useful in detecting a processor stop request failure which may occur on a <b>run</b> timeout command. For all other situations, this expression will return a value of 1 (since command file expressions are evaluated only when the processor is stopped)
run_timeout	Returns 1 if the processor/process was stopped due to a run timeout since the <b>run</b> command was given. This value is cleared on program start and is reset every time a RUN command is issued. After a RUN is completed, this value will remain valid until the next RUN is issued.
stop_timeout	Returns 1 if the processor/process was stopped due to a stop timeout since the <b>stop</b> command was given. This value is cleared on program start and is reset every time a STOP command is issued. After a STOP is completed, this value will remain valid until the next STOP is issued.

To use these special expressions, simply put the desired expression between the () characters of an if, while or do construct.

## Command File Parameters

When starting a command file to be run by RISCWatch, it is possible to pass values into the command file using RISCWatch command file parameters.

To do so, two things must be done:

1. A parameter list must be supplied with the command filename
2. A parameter definition must be specified in the command file

A parameter list is a set of one or more values enclosed by the '{' and '}' characters. If more than one value is specified, they must be separated by commas (,).

A parameter definition takes the form of the keyword **parms** followed by a list of the parameters that will take on the values specified in the parameter list. This list is composed of one or more variable names enclosed by the '{' and '}' characters.

To enhance readability and maintainability of a command file, it is suggested that the **parms** command be the first command of a command file, although RISCWatch does not explicitly require this.

When the **parms** command is read by RISCWatch, it immediately creates the variables and assigns each int or float variable a value of 0 or string variable a null, just as though a **create** command was executed with no initial value. This allows these variables to be used as normally created variables even if no parameter list is specified.

The following command could be used in a command file to create three command file variables to be used as parameters:

```
parms {var1, var2, var3}
```

Notice the space between the **parms** command and the '{' character. This space must be there for RISCWatch to identify the command.

To pass outside values into the command file and have them assigned to these variables simply call the command file like this:

```
rwppc file.cmd{10, 20, 30}
```

Notice that there is no space between the command filename and the '{' character.

For this example, var1 would be assigned a value of 10, var2 a value of 20, and var3 a value of 30. The values passed in the parameter list are assigned in sequence to the variable names in the parameter definition.

It is possible for the caller to specify fewer parameters in the list than are in the parameter definition. Using the previous example, if the command file was executed with the following call:

```
rwppc file.cmd{10, 20}
```

the variable var1 would have a value of 10, var2 a 20 and var3 a 0. Since all parameter variables are assigned a value of zero (0) when they are created, if no value for them is specified in the parameter list, they remain zero (0).

Similarly, if no parameter list was specified, all the variables would have a value of zero (0). A parameter list can also be specified when executing a command file from within RISCWatch using the **exec** command.

## Command File Pseudo-Variables

There are a few special variables that are available for use but they can not be used like normal variables. Hence they are called pseudo-variables. Pseudo-variables are used to determine the values of certain system resources. They can not be read or written in the normal sense. However, they can be used in **set** expressions and/or referenced inside a **print** or **fprint** command.

The RISCWatch pseudo-variables include:

\$DATE	Contains the current calendar date. The format of this pseudo-variable is weekday month day year. This may be used in a <b>print/fprint</b> command only.
\$ERRORS	Contains the number of program errors generated since RISCWatch was started. This may be used in a <b>set</b> expression or a <b>print/fprint</b> command. The program error count can be reset at any time by issuing " <b>set \$ERRORS = 0</b> ".
\$FILESIZE	Contains the number of bytes loaded from the last successful <b>load</b> command. This may be used in a <b>set</b> expression or a <b>print/fprint</b> command.
\$IP_FILE	Indicates the source file name associated with the current IAR setting. The file name is qualified and will match the names listed in the Files window. If the IAR is not associated with a specific file name, this variable will be set to '?'.
\$IP_FUNC	Indicates the function name associated with the current IAR setting. The function name will match the names listed in the Functions window. If the IAR is not associated with a specific function, this variable will be set to '?'.
\$IP_FUNC_END	Indicates the first address beyond the end of the function associated with the current IAR. If the IAR is not associated with a specific function, this variable will be set to zero.

\$IP_FUNC_START	Indicates the first address of the function associated with the current IAR. If the IAR is not associated with a specific function, this variable will be set to zero.
\$IP_LINE	Indicates the file source line number associated with the current IAR. If the IAR is not associated with a specific source line, this variable will be set to zero.
\$IP_PROG	Indicates the program name associated with the current IAR setting. The program name is qualified and will match the names listed in the Programs window. If the IAR is not associated with a specific program name, this variable will be set to '?'.
\$STOP_ON_ERROR	This variable is always set to zero when command file processing is started. Setting this variable to one will halt command file execution if an error message is generated.
\$TIME	Contains the current clock time. The format of this pseudo-variable is hour:minute:second. This may be used in a <b>print/fprint</b> command only.
\$TIMER	Contains the current timer value. See the timer command for details. This may be used in a <b>set</b> expression or a <b>print/fprint</b> command.

## Command File Programming Example

The following is an example that uses command file programming logic to set a register variable based on the value of the IAR. In the example, a while loop is executed a maximum of 20 (count) times. The value at memory address location 0xFFFF8000 is added to the contents of GPR0 and compared to the IAR. If the IAR is greater than this value, register variable S1 (and hence R2) is set to the loop count value; otherwise the value of GPR0 is increased by 0x1000 for the next loop iteration.

```

assign S1 = R2
create count = 20
while (count != 0)
    if (IAR > R0 + (0xFFFF8000))
        set S1 = count
        set count = 0
    elseif
        set R0 = R0 + 0x1000
        set count = count - 1
    endif
endwhile

```

## Running a Command File

Command files can be run from within RISCWatch using the **exec** command or they can be run by passing their filename to RISCWatch on the command line when RISCWatch is started.

If a command file parameter is passed to RISCWatch at program startup, RISCWatch is put in command file batch mode. In this mode, each of the commands in the file are executed without enabling the graphical user interface. Once the last command in a command file executes, RISCWatch terminates itself and returns control to its parent process. This allows RISCWatch to be run from either a host command prompt or called from within a host shell script.

To run a command file from within RISCWatch, type in the following on the command line of the user interface:

```
exec command_file step or window cfss command_file
```

To run a command file at program startup in command file batch mode (no graphical user interface provided), type in the following at the shell prompt:

```
rwppc command_file
```

To run a command file at program startup in normal mode (with the graphical user interface enabled), add the following line to the **rwppc.env** file:

```
STARTUP_FILE = command_file
```

Where:

command_file	The name of the command file to be executed. For example: test.cmd
step	Runs the command file in step mode. This option is only valid when executing a command file from the user interface. See “Command File Window” on page 3-132 for more information on running a command file using step mode.

## Command File Window

The Command File window allows a command file to be run in an interactive session for development and debugging. It also allows the command file to be edited and saved. The following section describes the functionality of this window.

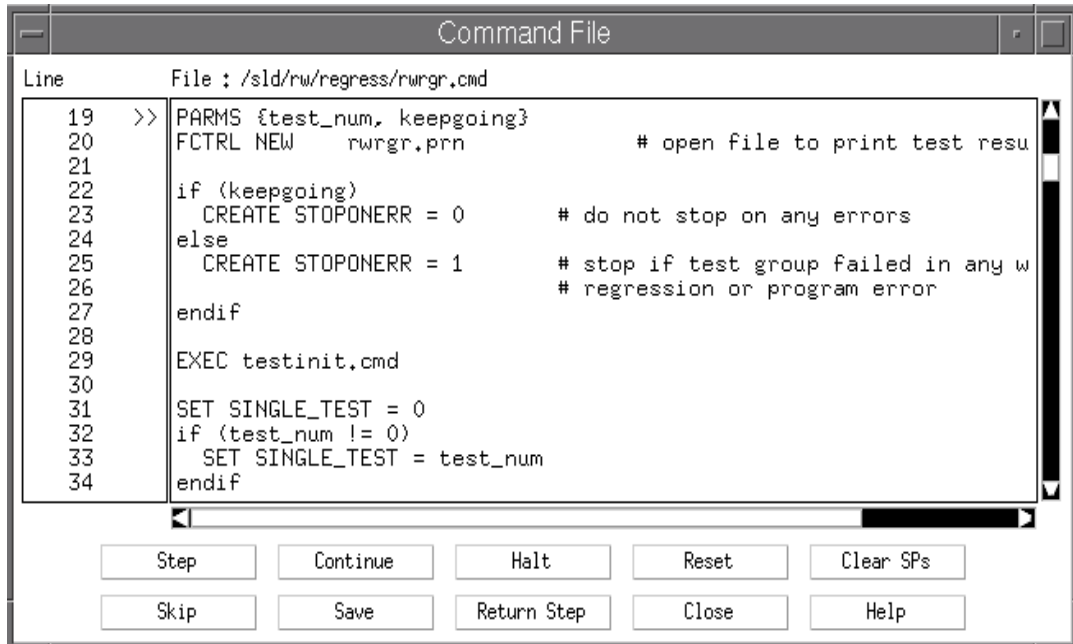


Figure 3-51. Sample Command File Window

- **Filename**

At the top of the window, the current command file being run is displayed. If the save option is used to save an edited command file and a different name is chosen, this filename will be changed to reflect the new command filename.

- **Line subwindow**

The Line subwindow is used to display the line numbers of the command file, active stop points, and the next line of the command file that will be executed (designated by the command cursor symbol '>>'). As commands are executed, the cursor will move to the next executable line, skipping blank and comment lines.

A single left mouse click in this subwindow will toggle a stop point (designated by the characters 'SP'). A stop point is used to halt command file execution when the selected line is about to be executed. Stop points are used in conjunction with the Continue button to quickly run to a specific location in a command file.



To execute the Run To and Go To actions, place the cursor on a specific line number and hold down the right mouse button. A selection list will appear, allowing the user to choose the desired function. The GoTo selection will cause the command cursor symbol to be changed to the target line, while the RunTo selection will set a stop point at the target line and run the command file to that location.

- **Text subwindow**

The Text subwindow is used to display the contents of the command file. When the Command File window is initially invoked, the contents of the command file will be read and placed in this window.

To change the contents of the window, simply use the mouse to place the edit cursor in the desired location, and then enter new text or delete existing text. To save your changes, use the Save button (see description below).

- **Step button**

The Step button is used to execute the command which appears next to the command cursor.

- **Continue button**

The Continue button is used to run the command file until a stop point is reached, the Halt button is selected, or the command file terminates.

- **Halt button**

The Halt button is used to halt command file execution.

- **Reset button**

The Reset button is used to reset the execution pointer to the first command in the initial command file. The Text subwindow will be scrolled to the top and the command cursor will be placed next to the first executable command of the file. The `$STOP_ON_ERROR` pseudo variable is reset to zero. For details of pseudo variable usage, see "Command File Pseudo-Variables" on page 3-129.

- **Clear SPs button**

The Clear SPs button is used to remove all stop points.

- **Skip button**

The Skip button is used to skip execution of the command appearing next to the command cursor. The command cursor will be placed beside the next executable command in the file.

- **Save button**

The Save button is used to save the contents of the Text subwindow. A file selection dialog box will allow any changes made to be saved in either the existing file or a new command file.

- **Return Step button**

The Return Step button is used to run through nested command file executions. When pressed, the current command file is run to completion and execution will stop at the line containing the nested exec command.

- **Close button**

The Close button is used to remove the Command File. Be advised that any changes made to the Text window that have not been saved will be lost!

- **Help button**

The Help button will display help text for this window.

- **Additional Features**

The Page Up/Down keys enable page scrolling of the command file. The up/down arrow keys are used to move the displayed text by a single line.

For details on how to perform character string search operations, or how to quickly scroll to a specific source line number, see “Input Line Usage” on page 3-49.

---

## Processor Resources

For PowerPC processors, RISCWatch can reset a target processor through its JTAG test port. Exact debug functions are specific to individual PowerPC processors.

### Processor Reset Window (JTAG Targets Only)

This window is used to access the reset functions of the processor. The three different kinds of resets available are Core, Chip (Core + ASIC) and System. Each reset performs a slightly different function.

For PowerPC 400Series processors, please refer to the appropriate processor User's Manual for a description of each reset.

For PowerPC 6xx/7xx processors the Core and Chip resets are equivalent. They will reset the processor and soft stop at address 0xFFFF00100. Also, the System reset will reset the processor and run from address 0xFFFF00100.

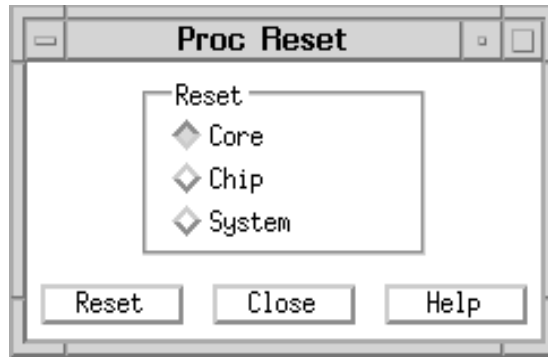


Figure 3-52. Sample Processor Reset Window

This window consists of three buttons which are used to select the type of reset that is desired. Use the mouse to select the appropriate reset then click on the Reset button located near the bottom of the window. To monitor the status of the reset, watch the contents of the message window. This status will indicate, among other things, whether the processor is running or stopped after the reset was performed.

---

## General Resources

### Window Layout

The window layout feature of RISCWatch is used to save the position and size of each visible window so that the exact screen layout can be loaded thereafter. If the LAYOUT variable in the environment resources file, **rwppc.env**, is set to LOADSAVE, RISCWatch automatically saves a window layout when the program is exited. This allows RISCWatch to load the same window layout the next time it is started.

To save the current window layout, access the Window|Layout|Save option of the Main window menubar. This will display a file selection dialog that can be used to specify an existing layout file or to create a new layout file of your choosing. Select an existing filename or type in a new filename and click on OK. This will save the window layout to the specified file. By allowing users to select their own files, RISCWatch allows multiple screen layouts to be saved to facilitate the needs of multiple users or resource dependent debugging needs.

To load a window layout, access the Window|Layout|Load option of the Main window menubar. Select the layout filename using the file selection dialog. The specified layout file will be accessed to configure the window layout just as it was saved.

## Output Window

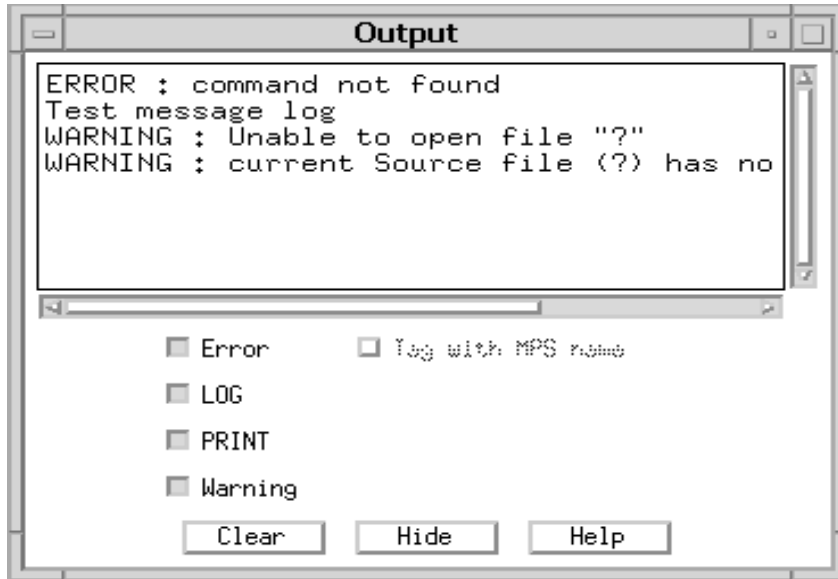


Figure 3-53. Sample Output Window

The Output window is used to display user-selectable messages generated when RISCWatch commands are executed. It is particularly useful when used to monitor the progression of long running command files, but can also be used whenever desired. The following section describes the functionality of this window.

- **Message Window**

The message window is used to display the type of messages that have been selected. Attached to this window are vertical and horizontal scroll bars used to view text that lies beyond the boundaries of the visible window.

- **Message Check Boxes**

The message check boxes are used to select the types of messages that will be displayed. Use the check boxes to configure the types of messages you would like to display.

- **MPS Tag Check Box**

The MPS tag check box is used, when in MPS mode, to tag each message with the MPS id from which it was generated. To turn off this tagging, simply unselect this check box.

This check box is only available for use when in MPS mode.

- **Clear Button**

The Clear button is used to clear the contents of the message window.

- **Close Button**

The Close button is used to remove the Output Window from the interface.

- **Help Button**

The Help button will display help text for this window.

## Window List

The window list is used to display any active window. An active window is a window that has been created by RISCWatch or by a user and may or may not be visible on the screen. This feature is particularly useful when a large number of windows are on the screen which may hide one or more windows from view.

By accessing the Window|List option of the Main window menubar, a window will be displayed that lists all of the active windows. Use the mouse to select the desired window and this window will be made visible and placed on top of all other RISCWatch windows.

## User Created Variables Window

A powerful part of using RISCWatch is the ability to create user variables which can then be used to manipulate numbers and strings. Such variables are created using the **assign** and **create** commands and destroyed using **unassign** and **uncreate**. These variables can be altered using the **set** command and their values displayed using **expr**

To provide an efficient means of creating, managing and destroying user created variables, the User Created Variables (UCV) window was created. Using this one window, it is possible to create, edit and destroy such variables as well as obtain instant feedback as to the current value for each one.

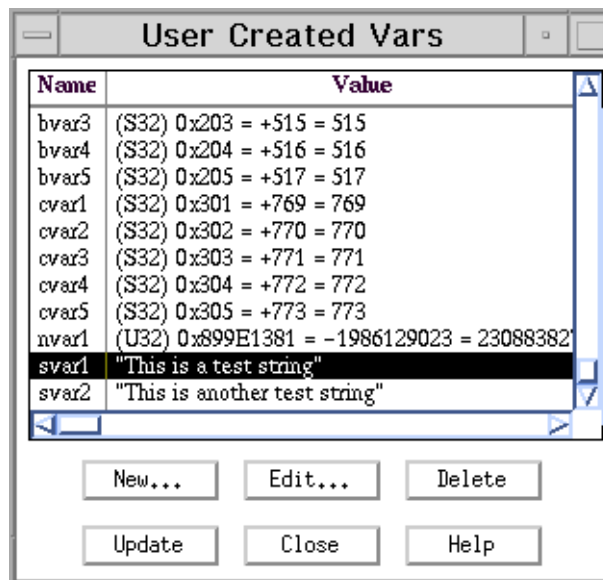


Figure 3-54. Sample User Created Variables Window

- **Data Area**

This area is divided into two columns displaying the name and value of each user create variable. The display order is sorted alphabetically by variable name. The window can be resized in both dimensions while the two scroll bars can be used to access data which is not contained within the visible data area.

The mouse is used to select a variable to manage and such a selection must be made before the Edit... or Delete buttons can be accessed.

- **New... button**

The New... button is used to create a new variable using the **assign** or **create** commands. Once selected, a new dialog will appear prompting you to enter the variable name, initial value and create type.

- **Edit... button**

The Edit... button is used to edit the value of an existing variable. Simply use the mouse to select the desired variable from the list and then click on this button. A new dialog will appear prompting for the new value to be entered.

- **Delete button**

The Delete button is used to destroy an existing variable. Simply use the mouse to select the desired variable from the list and then click on this button. The appropriate **unassign** or **uncreate** command will then be issued to destroy this variable.

- **Update button**

The Update button is used to immediately update the display with the latest value for each variable. This window will automatically update whenever new variables are created or changed via the **set** command but assigned variables which are linked to memory or register contents might not always reflect their latest values. Use this button to make sure you are seeing the current value if there is any doubt.

## Log Files

Every time that RISCWatch is started, a log file is opened. Log files are used by RISCWatch to log all commands entered by the user, actions accessed via the graphical user interface, the results of actions, and all status and error messages. Each entry put in a log file is time stamped so that the exact times of actions can be recalled if they will be needed at some later date.

Log files also allow for the sequence of actions to be recorded so they may either be repeated, performed again in the exact same sequence, or for a system operator to figure who's been doing what with RISCWatch and the processor it is connected to.

RISCWatch creates a new log file for each day that it is started. When RISCWatch is started, it notes the month and day and looks to see if a log file already exists for this date. If a file does not exist, RISCWatch opens a new file for logging. If a file does exist for this date,

RISCWatch simply opens the existing file and appends all new log entries to the end of the file.

RISCWatch log files are given names to reflect the month and day they contain log entries for. For example, if you were to run RISCWatch on August 19, after leaving RISCWatch, there would be a file in the current directory called RW0819.LOG. This naming convention allows for several months, or even years, of development time, effort and methodology to be tracked and/or used to generate status and activity logs.

When RISCWatch is started, logging of all entries is automatically enabled. By using the Logging option of the Utilities pull-down menu in the main program window, or the **logging** command, it is possible to disable logging if need be. It is also possible for any user to place their own comments in the log file by using the Utilities|Logging|Comment pull-down or the **log** command.

By using a resource defined in the RISCWatch environment file (**rwppc.env**), it is possible to specify the directory where all log files are kept by RISCWatch. The name of the resource is LOG\_FILE\_DIR. The following is an example of how to use this resource in the **rwppc.env** file:

```
LOG_FILE_DIR = /u/rwppc/log_files
```

RISCWatch will detect this resource and maintain all log files in the specified directory.

## Logging Control

By default, RISCWatch saves all commands and messages to the current log file. At certain times, it may be deemed necessary to disable this functionality. To control the state of logging, the **logging** command or the Logging State window is used.

To determine the current logging state, enter the **logging** command on the command line in the Main window and note the message displayed in the message window. To turn off logging, type 'logging off' on the command line. To turn logging back on, type 'logging on'.

The same actions can be accomplished using the user interface. Select the Utilities|Logging|State option of the Main window menubar. This will display a small popup window indicating the current logging state. To switch logging states, select the Yes button. To leave the logging state as is, select the No button.

See **logging** on page 5-73 in the Command Reference for a detailed description of this command.

## Logging User Comments

It is possible for RISCWatch users to enter their own comments into the current log file. To do so, either the **log** command or Log Comment window is used. The **log** command keyword is entered on the command line of the Main window followed by the text to be entered in the log file. See **log** on page 5-72 in the Command Reference for a detailed description.



The Log Comment window, shown in Figure 3-49 below, is displayed by using the Utilities|Logging|Comment pulldown of the Main window menubar.



Figure 3-55. Sample Log Comment Window

Type the text to be entered in the log file in the edit field and then press the Enter key. Select the Close button to remove this window from the screen. Select the Help button to bring up help information for this window.

## Screen Capture

The contents of certain data intensive windows may be saved to an ASCII file using the **capture** command. This command allows significant amounts of information to be saved so that it may be viewed later or for several samples to be taken to be used for comparison purposes.

When the **capture** command is used, the desired window is specified and the contents are captured to a file. If no file is specified, the contents will be saved to a file named **rwppc.cap**. To override this name, a filename is specified with the capture options.

The contents of the capture file will contain a time and date stamp for each capture that is requested along with a description of the window captured followed by the appropriate window data.

See **capture** on page 5-26 in the Command Reference for a detailed description and a list of available options.

## Calculator Window

The Calculator window is used to mimic the operations of a basic arithmetic calculator.

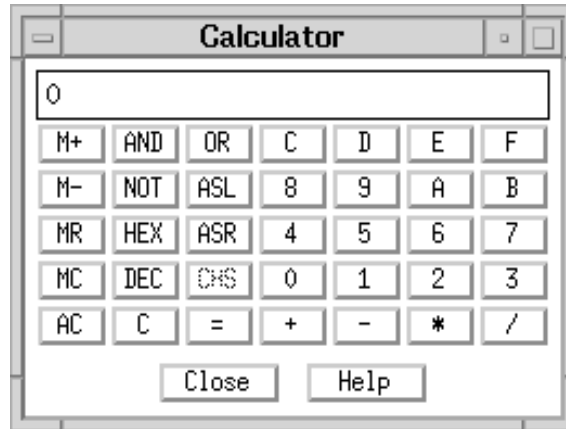


Figure 3-56. Sample Calculator Window

The calculator will run in either decimal or hexadecimal modes. Use the DEC and HEX buttons to switch the current mode.

When in DEC mode, the AND, OR, NOT, A, B, C, D, E, and F buttons will not function. When in HEX mode, the CHS button will not function.

To convert a number between the two modes, simply enter the mode that the number is to be entered in, enter the number and then click on the alternate mode button which will convert the number and then display its value.

- The mathematical operations available are:

+ = addition

- = subtraction

\* = multiplication

/ = division

CHS = change sign

- The bitwise operations available are:

AND = bitwise AND

OR = bitwise OR

NOT = one's complement

ASL = arithmetic shift left

ASR = arithmetic shift right

- Memory buttons:

M+ = add value in display to memory value

M- = subtract value in display from memory value

MR = recall the memory value to the display

MC = clear the memory value to 0

- Other buttons:

AC = all-clear - clears the value in the display and current calculation

C = clear - clears the value in the display

= = computes the value of the previously entered number with the value in the display using the previously specified operator

---

## Online Help

RISCWatch provides extensive online help. Most windows contain a Help button which provides a detailed description of the window's features.

Using the Help pulldown of the Main window, it is possible to display help information for the following topics:

- The RISCWatch program version number
- Frequently Asked Questions (FAQ)

The Help pull down from the Main window also provides selections for links into online documentation:

- RISCWatch Install Guide
- RISCWatch User's Manual
- PowerPC Manuals (web URL is provided)

The RISCWatch online documents are contained on the CD-ROM provided with RISCWatch and are in PDF format. For successful display of these documents, the following setup must be performed:

- The host machine must have Adobe Acrobat Reader installed (visit [www.adobe.com](http://www.adobe.com) to download it for free).
- The system PATH environment variable or RISCWatch search path must contain the path for the “acroread” program provided with AcrobatReader.
- The RISCWatch search path must contain the path for the “rw\_ig.pdf” file. This is the name of the install guide PDF file and is provided on the CD-ROM.
- The RISCWatch search path must contain the path for the “rw\_um.pdf” file. This is the name of the user’s manual PDF file and is provided on the CD-ROM.

**Note:** RISCWatch will display an error message if any of these files cannot be found.

Since the help viewer invoked varies depending on the host platform and option selected, the instructions for using that particular viewer must be viewed online. Once a help window is displayed, access the Help selection of the window’s menubar to obtain additional information about the viewer being used.

---

## Chapter 4. Using Processor-Specific Debug Features

This chapter provides detailed information about RISCWatch features applicable to specific PowerPC processors or families of processors. Individual processor implementations within the PowerPC architecture may vary in terms of internal register types, cache size and organization, availability of a memory management unit, and other hardware functions. The RISCWatch windows in this chapter support these implementation-specific functions.

Table 4-1 summarizes the features of the RISCWatch Debugger presented in this chapter, along with the applicability of each feature or window to specific PowerPC processors or processor families:

**Table 4-1. Quick Reference for Processor-Specific Debug Features**

Task or Resource	Applicable Sections
Managing Hardware Breakpoints	“Using RISCTrace (400Series JTAG Processor Probe Only)” on page 4-2 “Trigger/Trace Window (400Series Only)” on page 4-7 “Compound Trigger/Trace Window (403Series Only)” on page 4-11
Memory Resources	“Translation Lookaside Buffer Window (Applicable Processors Only)” on page 4-14

---

### PowerPC 400Series MMU Implementation Notes

RISCWatch support for the Memory Management Unit (MMU) of the 400Series processors is subject to adherence to the following conditions:

1. The translation mode for Data and Instruction access must be the same. They can both be enabled or disabled; having only one enabled is not supported.
2. If program execution is stopped at a point where the translation mode has changed from the state existing upon the initial file load, then the mapping must be real = virtual. If this is not the case, the source level debug information for the stopped context will not be displayed correctly.
3. The real addresses in the TLB entries are assumed to be correct and valid addresses.

Actions performed via the TLB window, described in “Translation Lookaside Buffer Window (Applicable Processors Only)” on page 4-14, or within the program itself that cause nonconformance to these conditions will produce unpredictable results.

---

## Managing Hardware Breakpoints and Trace Events

See “Using Hardware Breakpoints” on page 3-73 for a general discussion of hardware breakpoints in RISCWatch.

### Using RISCTrace (400Series JTAG Processor Probe Only)

Certain PowerPC 400Series processors provide a real-time trace debug mode which supports tracing the instruction stream being executed out of the instruction cache in real time. This mode does not affect the performance of the processor.

RISCWatch provides a mechanism to utilize the hardware trace capabilities of the chip and gather a nonintrusive reconstruction of the flow of executing processor instructions. This feature of RISCWatch is known as RISCTrace. RISCTrace collects trace information from the trace status port in real-time and then reconstructs the flow of the code using the collected information and the contents of processor memory or program files.

**Note:** This is an instruction trace only; RISCTrace does not capture the contents of registers or memory.

RISCTrace requires a JTAG Ethernet processor probe target which has trace capabilities. The RISCWatch controls for RISCTrace appear only if RISCWatch detects that it is connected to a processor probe which supports trace and a PowerPC 400Series chip which supports trace.

When trace is supported, the Trigger/Trace and Compound Trigger/Trace windows provide the RISCTrace controls necessary to define and manage trace collection. From these windows the user can define the events which initiate the trace collection, and other trace parameters such as the number of cycles to trace. Refer to the Trigger/Trace window descriptions which follow in this section for a detailed description of the controls on these windows.

After the trace parameters are specified, the Run Trace button can be used to start the processor running and initiate trace collection. When a specified trace is complete, RISCTrace automatically stops the processor, collects the trace information and reconstructs and formats it. This is true if the Autostop checkbox is enabled. If the Autostop checkbox is disabled, RISCTrace will only indicate that the trace is complete. When the user stops the processor, RISCTrace then collects the trace information and reconstructs and formats it. The formatted trace is saved in the file **rwppc.trc** and displayed in a view window (see the “view” on page -137 for details on using this window). The Save Trace button can be used to save the formatted trace in a file of your choice, as well as allowing you to enter optional comment lines which are appended to the beginning of the formatted trace information in the saved file.

Selecting the Stop Trace button while a trace is running causes the trace which is currently running to be aborted. The abort results in the processor being stopped with no trace reconstruction occurring for the trace which was running. However, if the trace is complete, the trace is collected, reconstructed and formatted. The Stop Trace button is useful when Autostop mode is disabled.

If it is not desired to have any program symbol information included in the trace output, the **unload all** command can be used to unload all the program information from RISCWatch prior to initiating the trace. This also speeds up the trace reconstruction. A detailed description of the trace output follows in the 'RISCTrace Output' section below.

For additional information on processor-supported trace, consult the appropriate chip user's manual.

## RISCTrace Operational Notes

1. RISCTrace uses the IOCR[RDM] bits (bits 26-27 of the IOCR register) to collect a trace. RISCTrace cannot properly trace code that modifies these bits. Also, if bits other than the RDM bits are to be changed by the application code, a read/modify/write operation is recommended. Note that the IOCR[RDM] field is 403 specific and does not necessarily apply to other 400Series processors.
2. If the IOCR[RDM] bits are set to bus status mode and a logic analyzer disassembler (aka inverse assembler) is hooked up to the processor, using RISCTrace to collect a trace will change the IOCR[RDM] bits from bus status mode to trace mode, collect the trace, then restore the bits to bus status mode. This operation may affect the operation of the logic analyzer disassembler. Note that the IOCR[RDM] field is 403 specific and does not necessarily apply to other 400Series processors.
3. RISCTrace uses debug events to collect a trace. Thus, RISCTrace cannot trace code that clears the DBSR because clearing this register also clears all debug events.
4. On the 403GCX, the RISCTrace pins are multiplexed with the new parity pins. If the IOCR[RDM] bits are set to parity mode, RISCTrace will not collect a trace. Pressing the run trace button will result in an error message. Also, RISCTrace cannot collect a trace from a clock doubled 403GCX unless the enhanced JTAG adapter assembly with RISCTrace is used (see the install guide for descriptions of the different adapters).
5. Known causes for a run trace failure are:
  - Invalid cycle count specified on the trace window (see "RISCTrace Controls" on page 4-10).
  - Trace port used for parity generation (403GCX only).
  - CPU is clock doubled (403GCX only) and an older version RISCTrace Processor Probe (maximum 64K cycle trace buffer) is being used. The new Processor Probe with RISCTrace should be used.
6. RISCTrace writes the reconstructed code to a file in the directory from which RISCWatch was started. These files may be in excess of 100Mbytes.

## RISCTrace Output

The output file resulting from a successful trace contains various elements of information which are presented in a consistent manner for each trace. Guaranteeing that key information is presented in a consistent manner allows users the flexibility to write their own post-processing routines which can operate on the trace output file.

The trace output file format is currently at version 2.1. Please run a trace with the latest version of RISCWatch and view the output stored in the file `rwppc.trc`.



```
# RISCTrace : Trace Output File Version 2.1
# DATE : Sun Aug 20 06:03:04 2000
```

```
# TRACE TRIGGER SETTINGS : IAC1 occurring 1 time
```

```
#Instr      Total      Cycle/ (function
#Count      Cycle      Instr  Address offsets)  Opcode  Disassembly
#-----      -----      -----      -----      -----      -----
```

```
$ FUNCTION: main START_ADDR: 0x0000A078 FILE: demo1.c PROGRAM: ./demo
00000001 00000000      0000A0A0(+000028) 90610040 stw      R3,0x00000040(R1)
#          00000000      ** STATUS: Trigger event **
00000002 00000002 0000002 0000A0A4(+00002C) 38600003 addi     R3,0,0x0003
00000003 00000004 0000002 0000A0A8(+000030) 90610050 stw      R3,0x00000050(R1)
```

```
*** Entries removed for figure display purposes ***
```

```
$ FUNCTION: routine4 START_ADDR: 0x0000A180 FILE: demo1.c PROGRAM: ./demo
00000062 00000214 0000011 0000A180(+000000) 9421FFC0 stwu     R1,0xFFFFF0(R1)
00000063 00000216 0000002 0000A184(+000004) 90610058 stw      R3,0x00000058(R1)
```

```
*** Entries removed for figure display purposes ***
```

```
00000073 00000253 0000002 0000A1AC(+00002C) 30210040 addc     R1,R1,0x0040
00000074 00000267 0000014 0000A1B0(+000030) 4E800020 blr
```

```
$ FUNCTION: main START_ADDR: 0x0000A078 FILE: demo1.c PROGRAM: /sld/rwppc/regress/src/demo
00000075 00000269 0000002 0000A0F8(+000080) 48000121 bl       $+0x00000120
```

```
$ FUNCTION: routine2 START_ADDR: 0x0000A218 FILE: demo2.c PROGRAM: /sld/rwppc/regress/src/demo
00000076 00000280 0000011 0000A218(+000000) 7C0802A6 mflr    R0
```

```
*** Entries removed for figure display purposes ***
```

```
00000121 00000505 0000011 0000A290(+000008) 800C0000 lwz     R0,0x00000000(R12)
00000122 00000507 0000002 0000A294(+00000C) 804C0004 lwz     R2,0x00000004(R12)
```

```
#          00000000      ** Interrupt detected **
```

```
$ FUNCTION: ? START_ADDR: ? FILE: ? PROGRAM: ./demo
00000125 00000543 0000032 FFFE0700      7C0004AC sync
00000126 00000559 0000016 FFFE0704      90200034 stw     R1,0x00000034(0)
00000127 00000575 0000016 FFFE0708      90400038 stw     R2,0x00000038(0)
```

Figure 4-1. Sample Trace Output File

The following general rules hold true for any trace output file, such as the sample in Figure 4-1:

1. All comments are preceded by the comment character '#'  
These may be separate comment lines, or comments at the end of trace entries.
2. If comment lines are added to the trace via the Save Trace window, they are the first lines in the file and preceded by the comment character '#'
3. A comment line containing the words 'RISCTrace : Trace Output File' either follows the optional comment lines (if they exist) or is the first line in the file.
4. A comment line containing the information 'DATE : time\_info' follows next, where *time\_info* is the time/date information in the format defined by the ANSI **ctime()** function.
5. A comment line containing the information 'TRACE TRIGGER SETTINGS trigger\_settings' follows, where *trigger\_settings* describes the trigger settings at the time the trace was collected and in the format shown at the top of the Compound Trigger/Trace window.
6. The trace header (preceded by the comment character '#') follows:

```
#Instr Total Cycle/ (function  
#Count Cycle Instr Address offsets) Opcode Disassembly
```

7. The trace entries follow next. Each field of the entry is aligned below the field name in the header, as described below:

Instr Count	The sequential entry number within the trace output.
Total Cycle	The running count of cycles for the trace.
Cycle/Instr	The number of cycles for this executed instruction. This field provides a quick way to determine which instructions in the trace are taking the most cycles to execute.
Address	The address of this executed instruction. This may include an offset in () from the beginning of the function if program symbol information is available
Opcode	The hex opcode for this instruction executed.
Disassembly	The disassembled Opcode value.

The first three columns of data are displayed in decimal format while the last three are always hexadecimal (whether preceded by '0x' or not).

8. If program information is loaded corresponding to a trace instruction address, a program information entry preceded by the special character '\$' appears before the first instruction of each new function entry point as it is encountered in the trace.

The format of the program information entry is as follows:

```
FUNCTION: func START_ADDR: start_addr FILE: file PROGRAM: prog
```

Where:

func	function name, '?' if unknown
start_addr	start address for the function, '?' if unknown
file	file containing the function, '?' if unknown
prog	fully qualified program name, '?' if unknown

If the trace execution flow goes from an instruction which has program information associated with it, to one with no program information, all the fields above are '?'.

9. A blank line appears between trace entries where a break in sequentially executed instruction addresses (for example, a branch to another area of the program) occurs.

## Trigger/Trace Window (400Series Only)

The Trigger/Trace window is used to manage hardware breakpoints and trace events. Breakpoints managed by this window are accessible by using the built-in debug functions of the processor. Hardware breakpoints are not available for OS Open targets. An explanation of trace capabilities is explained in "Using RISCTrace (400Series JTAG Processor Probe Only)" on page 4-2.

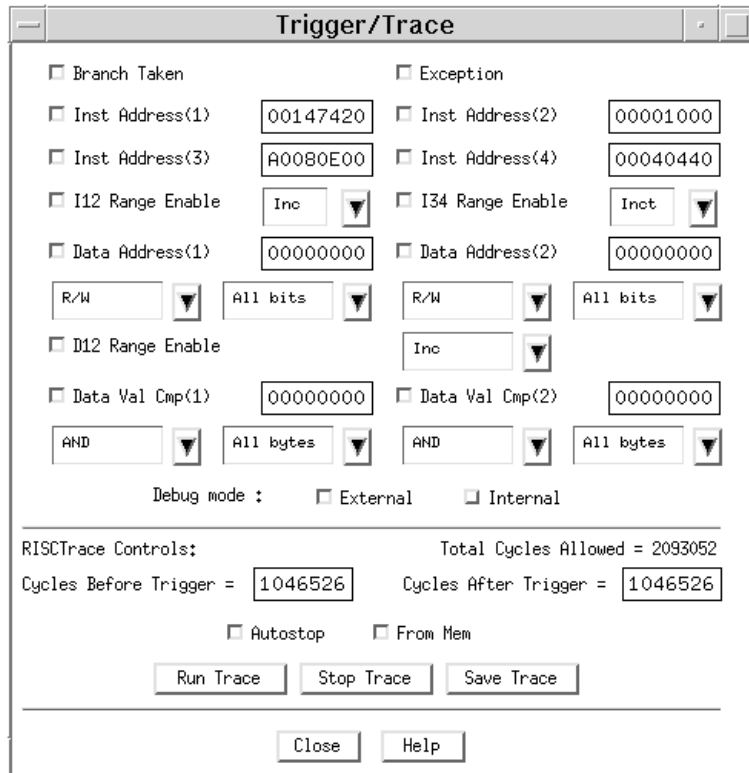


Figure 4-2. Sample Trigger/Trace Window with Trace Supported

- **Branch Taken event**

The Branch Taken event trigger is enabled and disabled according to the state of its check box. If the check box is enabled, the trigger is enabled too.

- **Exception event**

The Exception event trigger is enabled and disabled according to the state of its check box. If the check box is enabled, the trigger is enabled too.

- **Instruction Address Compare events**

An Instruction Address Compare event trigger is enabled and disabled according to the state of its check box. If the check box is enabled, the trigger is enabled too.

If an Instruction Address Compare is enabled, the appropriate address to trigger on should be entered in the address field. Use the mouse to place the edit cursor in the appropriate address field, enter a new hexadecimal value and then press the Enter key.

- **Instruction Address Range events**

Instruction address range events are enabled and disabled according to the state of the Instruction Range Enable check boxes. These selections are enabled only when the corresponding Instruction Address check boxes have been selected. For example, the I12 Range Enable checkbox would be enabled only if the Inst Address(1) and Inst Address(2) check boxes were already selected. The range specified can either be inclusive (Inc), inclusive toggle (Inct), exclusive (Exc) or exclusive toggle (Exct) to the values specified in the Inst Address data fields.

The toggle ranges are used to automatically toggle between ranges each time the processor stops for the specified range. For example, if inclusive toggle is enabled and the processor stops in the specified range, the toggle will switch to exclusive mode. If the processor is run and then stops in the exclusive range, the toggle will again occur and return to inclusive mode.

The toggle ranges are only available on some 400Series processors.

- **Data Address Compare events**

A Data Address Compare event trigger is enabled and disabled according to the state of its Data Address check box. If a check box is enabled, the trigger is enabled for that event.

If a Data Address Compare is enabled, the appropriate address to trigger on should be entered in the address field. Use the mouse to place the edit cursor in the appropriate address field, enter a new hexadecimal value and then press the Enter key.

For the Data Address Compare events, a trigger may be generated for a read and/or write to the specified address. Enable the desired event(s) by selecting the desired mode in the corresponding list box. The list box entries are displayed by using the mouse to place the edit cursor on the triangle next to the selection and pressing the left mouse button. The option is then selected by clicking on the desired selection. The Data Address Compare events also allow for masking of the data address on compares through the use of the corresponding list button to the right of the mode selection. The masking size is once again selected by using the triangle to display the possible options and clicking on the desired entry.

- **Data Address Range events**

Data address range breakpoints are enabled and disabled according to the state of the Data Range Enable check box. These selections are enabled only when both of the Data Address check boxes have been selected. The range specified can either be inclusive (Inc) or exclusive (Exc) to the values specified in the Data Address Compare data fields.

- **Data Value Compare events**

A Data Value Compare event trigger is enabled and disabled according to the state of its Data Val Cmp check box. If a check box is enabled, the trigger is enabled for that event.

If a Data Value Compare is enabled, the appropriate data to be used for the compare should be entered in the data field to the right of the check box. Use the mouse to place the edit

cursor in the appropriate data field, enter a new hexadecimal value and then press the Enter key.

Next, the compare conditions must be specified by indicating which bytes are to be compared, and how they are to be compared. This is accomplished by selecting the desired options in the list boxes shown directly under the corresponding Data Val Cmp check box.

Note that a Data Value Compare can only be enabled when the corresponding Data Address compare event has previously been enabled.

- **Debug mode**

The Debug mode check boxes are used to select the debug mode under which the processor will be running which in turn dictates the action to be taken when an event is triggered. Select the External check box to run in External Debug mode. Select the Internal check box to run in Internal Debug mode. In External Debug mode, when a debug event is detected the processor will be stopped. In Internal Debug mode, when a debug event is detected, the processor will vector to the appropriate exception handler for processing.

**Note:** For normal exception-driven processing of Data or Instruction Address breakpoints by a ROM Monitor or OS Open target, Internal debug mode should be selected.

For additional information on these and other processor debug features, consult the Debugging chapter of the User's Manual for the specific PowerPC 400Series processor being used. Not all features shown above are supported across all 400 Series processors.

## **RISCTrace Controls**

RISCTrace controls appear on the window only if RISCWatch determines that trace is supported. Refer to "Using RISCTrace (400Series JTAG Processor Probe Only)" on page 4-2 for an explanation of RISCTrace. When a trace is running, the trigger events described above define when the trace is triggered. The following controls are specific to RISCTrace:

- **Cycle count specification**

The 400Series processor which RISCWatch is attached to may support either a 'forward only' trace (where tracing begins only after the specified trigger event occurs) or a 'backtrace' capability (where a 'window' of cycles around the trigger event may be specified).

If the processor supports a 'forward only' trace, the 'Cycles Before Trigger' count (the count of cycles before the trigger event occurs) is always zero and cannot be altered. The 'Cycles After Trigger' count (the count of cycles following the trigger event) can be adjusted with a value not exceeding the maximum size of the trace.

If the processor supports a 'backtrace' capability, the 'Cycles Before Trigger' count and the 'Cycles After Trigger' count can both be adjusted to define a 'window' of cycles around the trigger event, with the total of the two not exceeding the maximum size of the trace.

Use the 'Total Cycles Allowed' field value to help you select suitable values for the before and/or after trigger counts.

In both cases, if the total of the two is below a certain minimum, the 'Cycles After Trigger' will be rounded up to this minimum.

- **Run Trace button**

After the trigger event(s) and cycle count(s) are specified, the Run Trace button starts the processor running and initiates trace collection. When a specified trigger event occurs, RISCTrace automatically collects the trace information and reconstructs and formats it. The formatted trace is saved in the file **rwppc.trc** and displayed in a view window (see the "view" on page -137 for details on using this window).

- **Stop Trace button**

Selecting the Stop Trace button while a trace is running causes the trace which is currently running to be aborted. The abort results in the processor being stopped with no trace reconstruction occurring for the trace which was running. However, if the trace is complete, the trace is collected, reconstructed and formatted. The Stop Trace button is useful when Autostop mode is disabled.

- **Save Trace button**

The Save Trace button can be used to save the formatted trace in a file of your choice, as well as allowing you to enter optional comment lines appended to the beginning of the formatted trace information in the saved file.

- **Autostop checkbox**

Enabling the autostop checkbox makes RISCTrace automatically stop the processor, collect the trace and then reconstruct and format it when the specified trace is complete.

Disabling the autostop checkbox makes RISCTrace only indicate that the specified trace is complete. Using the Stop Trace button will then stop the processor, collect the trace and then reconstruct and format it.

- **From Mem checkbox**

Enabling the From Mem checkbox informs the RISCTrace reconstruction software to use the contents of memory on the user's target during the reconstruction process.

Disabling the From Mem checkbox informs the RISCTrace reconstruction software to use the program files that were previously loaded with the RISCWatch load command during the reconstruction process. Any address required by the reconstruction process and not found in the program files is read from memory on the user's target.

## **Compound Trigger/Trace Window (403Series Only)**

The Compound Trigger/Trace window is available on those processors which support compound debug events. This window is very similar to the Trigger window with some additional features to make use of compound debug event functionality. Refer to "Trigger/Trace Window (400Series Only)" on page 4-7 for an understanding of the basic

features this window provides and to “Using RISCTrace (400Series JTAG Processor Probe Only)” on page 4-2 for the control information provided with RISCTrace

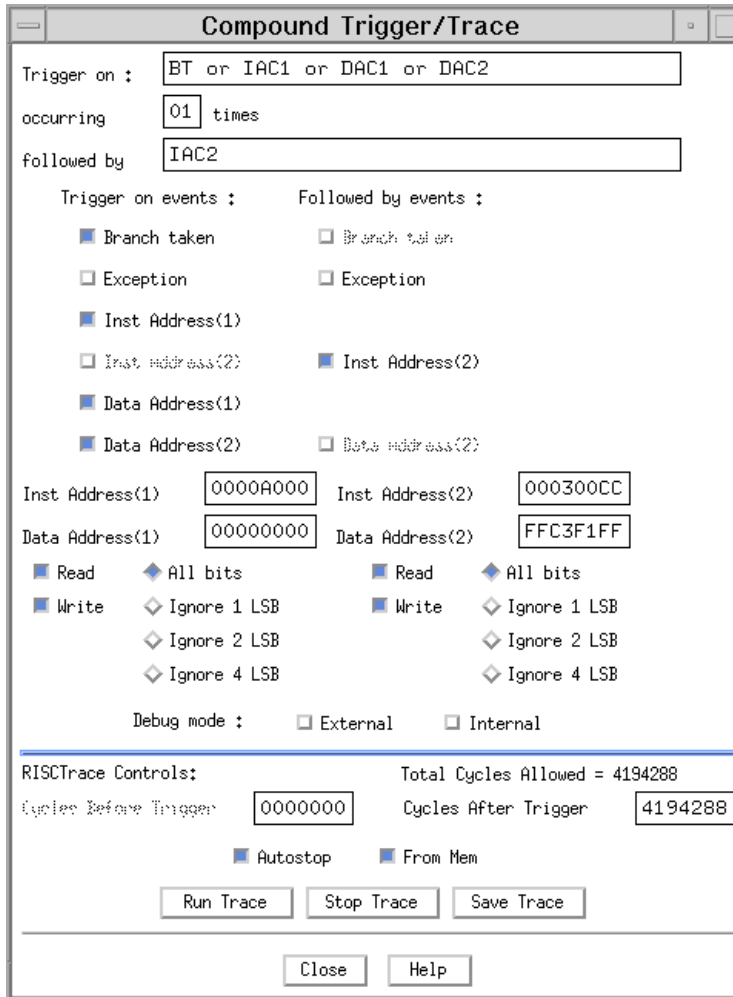


Figure 4-3. Sample Compound Trigger/Trace Window with Trace Supported



Using the Compound Trigger/Trace window, three classes of triggers may be set up:

1. Trigger on one or more events
2. Trigger after one or more events occurs a specified number of times
3. Trigger after one or more events occurs a specified number of times which is followed by a single occurrence of one or more events.

Available debug events include:

1. Branch taken
2. Exception
3. Instruction address compare
4. Data address compare

The initial trigger events are selected using the check boxes under the “Trigger on events” heading. These check boxes are the same as those found in the Trigger window. One or more of these events may be specified. As events are selected, notice the text appearing in the “Trigger on” field at the top of the window.

If it is desired, an event occurrence counter may be set using the text field at the top of the window. Enter the desired count into the box and press Enter.

Once a Trigger on event is specified, several “followed by” events are available for use as check boxes under the “Followed by events” heading. If an event is selected as a Trigger-on event, it is not available for use as a Followed by event and vice versa. As Followed by events are selected, notice the text appearing in the “followed by” field at the top of the window.

The Instruction and Data address controls at the bottom of the window can only be accessed if the appropriate event has been selected as a “Trigger on” or “followed by” event.

The Debug mode check boxes are used to select the debug mode under which the processor is running which in turn dictates the action to be taken when an event is triggered. Select the External check box to run in External Debug mode. Select the Internal check box to run in Internal Debug mode. In External Debug mode, when a debug event is detected the processor is stopped. In Internal Debug mode, when a debug event is detected, the processor vectors to the appropriate exception handler for processing.

**Note:** For normal exception-driven processing of Data or Instruction Address breakpoints by a ROM Monitor or OS Open target, Internal debug mode should be selected. Hardware breakpoints are not available for OS Open targets.

RISCTrace controls appear on the window only if RISCWatch determines that trace is supported. See “RISCTrace Controls” on page 4-10 for further information.

---

## Memory Resources

See “Reading and Writing Memory” on page 3-103 for a general description of RISCWatch features and windows for memory access.

### Translation Lookaside Buffer Window (Applicable Processors Only)

The TLB window is used to read and write entries in the Translation Lookaside Buffer (TLB) of a processor which contains a Memory Management Unit (MMU). “PowerPC 400Series MMU

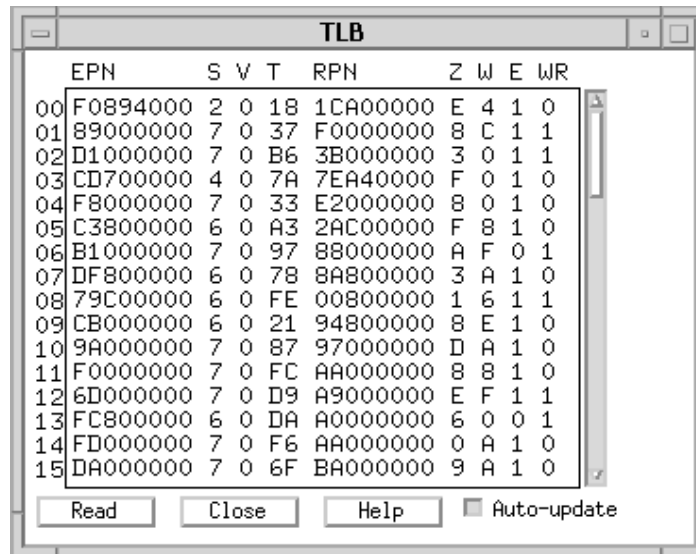


Figure 4-4. Sample TLB Window

Implementation Notes” on page 4-1 provides details affecting RISCWatch support for TLB operations. Additionally, for OS Open targets, the TLB window is only functional with OS Open version 1.6 or later and is not available for OS Open with Virtual Memory targets.

This window is displayed by selecting the Memory|TLB option of the menu bar’s Hardware pulldown choice. Along the left hand side of the window are the TLB entry numbers. To the right is the data area, where the contents of the TLB are shown .

The scroll bar located on the right side of the data area can be used to show entries that do not fit in the window display. Alternatively, the window can be resized to show the desired number of entries.

The labels across the top of the data window are used to help identify the quantities being displayed for the TLB entries. The labels are:

EPN	effective page number
S	page size
V	valid bit
T	TID
RPN	real page number
Z	ZSEL field value
W	WIMG bits (Write-through, Inhibit, Memory coherence, Guarded)
E	EXecute bit
WR	WRite bit

**Note:** Page numbers (EPN & RPN) are always displayed normalized to bit 0 (MSB). WIMG bits are displayed as a hexadecimal value with bit positions, from left to right, being W, I, M, and G.

The Read button is used to force a read of the processor TLB data to display the latest contents.

The Close button is used to remove this window from the screen.

---

## Processor Resources

See “Processor Reset Window (JTAG Targets Only)” on page 3-134 for a description of RISCWatch options for resetting a PowerPC processor.



---

## Chapter 5. Debugger Command Reference

This chapter describes the RISCWatch Debugger commands. These commands can be entered on the command line of the Main window of the graphical user Interface.

The commands are listed in alphabetical order. Each command description contains the following sections:

- Name
- Syntax
- Description

Some command descriptions contain one or more of the following sections:

- Flags
- Restrictions
- Examples
- Notes
- See Also

---

### Processors Currently Supported

The RISCWatch Debugger supports numerous PowerPC processors and versions. For more information on current processors supported and other up to date information, please refer to the README file included with the product, or visit our web site at <http://www-3.chips.ibm.com/chips/products/powerpc/tools/riscwatc.html>

Support for additional PowerPC processors and targets is planned for future RISCWatch releases.

---

### Reading the Syntax Diagrams

See “Syntax Diagram Conventions” on page xxiii for detailed information about the conventions used in the RISCWatch Debugger command syntax diagrams.

---

### Using RISCWatch Debugger Commands

Commands and keywords are not case sensitive. You may enter commands using either uppercase or lowercase characters. File names and variable names are typically case sensitive and should be entered in lower case or as shown in the individual command descriptions.

## Window Quick Reference

*window* window name, specified by one of the following keywords:

ascii	ASCII Memory window
asic	ASIC register window
* breakpoint	Breakpoints window
* cache	Cache (unified) window
calculator	Calculator window
* callers	Callers window
cfss	Command File Single Step window
coherency	Memory Coherency window
ctrigger	Compound Trigger Trace window
* dcache	Data Cache window
dcr	Device Control Registers window
debug	Assembly Debug window
dtlb	Data Translation Lookaside Buffer window
* files	Files window
fpr	Floating Point Registers window
* functions	Functions window
* globals	Globals window
gpr	General Purpose Registers window
* icalche	Instruction Cache window
* inspect	Variable Inspect window
itlb	Instruction Translation Lookaside Buffer window
* l2	L2 Cache window
* locals	Locals window
log	Log Comment window
* osopen	OS Open window
output	Output window
* programs	Programs window
reset	Processor Reset window
regfld	Single register field
rfcache	Fields for Cache Registers
rfconfig	Fields for Configuration Registers
rfdbat	Fields for 6xx DBAT Reg Fields
rfdbg	Fields for Debug Registers
rfdma0	Fields for DMA0 Registers
rfdma1	Fields for DMA1 Registers
rfdma2	Fields for DMA 2 Registers
rfdma3	Fields for DMA 3 Registers
rfdram	Fields for DRAM Registers
rfpr0	Fields for FPRs 0-7

rffpr1	Fields for FPRs 8-15
rffpr2	Fields for FPRs 16-23
rffpr3	Fields for FPRs 24-31
rffibat	Fields for 6xx IBAT Reg Fields
rffmem	Fields for Memory Protection Registers
rffmmu	Fields for 6xx MMU Regs
rffpm	Fields for Performance Monitor Regs
rffsr	Fields for 6xx Segment Reg Fields
rffsram	Fields for SRAM Registers
rfftimr	Fields for Timer Registers
* source	Source window
spr	Special Purpose Registers window
tlb	Translation Lookaside Buffer window
trigger	Trigger Trace window
ucv	User Created Variables window
udw	User Defined window
* varcfg	Variable Configuration window
winlist	Window list

\* Applies to commands using both the window and pane keyword (except for **window** command).

**Note:** Not all windows are applicable to all target processors and hosts.

## Command Quick Reference

The following is a list of commands and the syntax of each command. For further details, see the syntax and description sections in the individual command reference pages which follow this quick reference.

The following identifiers are used to improve readability :

[ ]	an optional item
	a selection between two or more items
<i>address</i>	any valid memory address value (usually specified as a 32 bit hex number)
<i>board_id</i>	valid MPS board name from MPS file
<i>int_var</i>	any integer variable created with the <b>create</b> command
<i>field_name</i>	an appropriate register field name as it appears in a Register Field window, or bit number
<i>filename</i>	a filename which may have an absolute or relative directory path. <b>NOTE:</b> If a specified filename and/or directory path contains a space character, you'll need to put the entire string in quotation ("" ) marks

whenever you type these in by hand (usually on the command line and in command files)

<i>fld_var</i>	any register field variable created with the <b>assign</b> command
<i>float</i>	any valid floating point number
<i>flt_var</i>	any float variable created with the <b>assign</b> or <b>create</b> command
<i>imm_var</i>	any immediate variable created with the <b>assign</b> command
<i>instance</i>	a register or window instance number
<i>mem_var</i>	any memory variable created with the <b>assign</b> command
<i>mips_id</i>	valid MPS chip or device name from MPS file
<i>pane</i>	window pane name, specified by one of the following keywords:
<b>bpset</b>	Breakpoint Select window, window showing functions with bp set
<b>bpunset</b>	Breakpoint Select window, window showing functions with bp not set
<b>cachedata</b>	Cache window, window showing data/tag entries
<b>cacheword</b>	Cache window, window showing word entries
<b>varinvis</b>	Variable Config window, window showing invisible vars
<b>varvis</b>	Variable Config window, window showing visible vars
<i>reg_class</i>	any valid processor register class (DCR, GPR, etc.) The list of valid classes may be found by accessing the program menu bar under Hardware   Register. The keyword ALL is always a valid class.
<i>reg_name</i>	any valid processor register name
<i>reg_pre</i>	any valid ASIC register prefix. If the chip you are debugging has any ASIC registers defined, the list of valid prefixes may be found by accessing the program menu bar under Hardware   Register   ASICs.
<i>reg_var</i>	any register variable created with the <b>assign</b> command
<i>src_var</i>	any valid local or global source variable name that is currently in scope. The name must be preceded by a colon ":". See "Source Variable Command Support" on page 3-102 for further information.
<i>str_var</i>	any string variable created with the <b>assign</b> or <b>create</b> command
<i>value</i>	any decimal, octal or hexadecimal value



Table 5-1 summarizes the syntax of the RISCWatch Debugger commands:

**Table 5-1. Syntax Summary for Debugger Commands**

<b>Command</b>	<b>Parameters</b>
<b>asmstep</b>	[value]
	fld_var = reg_name.field_name
	imm_var = value
<b>assign</b>	mem_var = (address)
	flt_var = float
	reg_var = reg_name
	str_var = "string"
<b>assm</b>	"assembly" [address int_var reg_name reg_var]
<b>attach</b>	threadid
<b>beep</b>	[off on]
<b>bot</b>	[window [pane]]
	set [dacr dacw dacrw] address [dac_reg] [cmp_size]
	set dvc dvc_reg [comp_bytes] [comp_mode]
	set ihw address [iac_reg]
	set [ihw] at src_file:line_num
	set [ihw] in ["function"]
<b>bp</b>	set range [inc inct exc ext] range_reg1 range_reg2
	clear address all iac_reg
	clear [dacr dacw] address
	clear at file_name:line_num
	clear in ["function"]
	clear range [inc exc] range_reg1 range_reg2
<b>bpmode</b>	[hw [step]] hardware [step]] [sw software]

**Table 5-1. Syntax Summary for Debugger Commands**

<b>Command</b>	<b>Parameters</b>
<b>callstep</b>	
<b>capture</b>	window[reg_pre] pane all [total] [filename]
<b>cfss</b>	sp set clear at file_name:line_num sp set clear at line_num sp clear all
<b>color</b>	[reset   [cbak cfore tback tfore wback wfore color]]
<b>config</b>	parity 32bitmode [on off]
<b>create</b>	flt_var = float int_var [= value] str_var = "string"
<b>delay</b>	value imm_var int_var
<b>detach</b>	
<b>dis</b>	value (address) int_var mem_var reg_name reg_var
<b>down</b>	[lines [window [pane]]]
<b>end</b>	[all]
<b>exec</b>	command_file[{variable_list}] [step]
<b>expr</b>	expression
<b>fctrl</b>	append new filename close errors log status on off
<b>file</b>	[filename]
<b>find</b>	[[string   "expression" [window [pane]]]   [\$last\$ window [pane]]]
<b>findb</b>	[[string   "expression" [window [pane]]]   [\$last\$ window [pane]]]
<b>finde</b>	[[string   "expressions" [window [pane]]]   [\$last\$ window [pane]]]
<b>focus</b>	[window [pane]]

**Table 5-1. Syntax Summary for Debugger Commands**

<b>Command</b>	<b>Parameters</b>
<b>fold</b>	on off
<b>fprdisp</b>	[hex sci]
<b>fprint</b>	print_string
<b>freeze</b>	never stop always
<b>funcdisp</b>	[all_addr all_name dbg_addr dbg_name]
<b>goto</b>	value label [addr address][[line file_name:line_num]
<b>halt</b>	[on off]
<b>hidewins</b>	
<b>ip</b>	
<b>jtag</b>	clock [value]   reset
<b>kill_thread</b>	
<b>line</b>	[value [window [pane]]]
<b>linestep</b>	[value]
<b>load</b>	binary bin filename address int_var imm_var dmem imem filename [address int_var imm_var] file filename [d=address] [s=address ss=size] [t=address] [nosym] host filename [d=address] [s=address ss=size] [t=address] [nosym] hp filename image filename layout filename motorola mot filename reg filename tektronix tek filename
<b>log</b>	message
<b>logging</b>	[on off]

**Table 5-1. Syntax Summary for Debugger Commands**

<b>Command</b>	<b>Parameters</b>
<b>logoff</b>	
<b>memacc</b>	
<b>memchk</b>	address [length]
<b>memcoh</b>	read mm phys reset write dmem bypass cache mm reset thru write imem cahce iidb iidu iudb iudu iidf reset
<b>memcopy</b>	source dest mm_var int_var length
<b>memfill</b>	address imm_var int_var value imm_var int_var value
<b>memfind</b>	address length string value [int_var]
<b>memrwait</b>	[value]
<b>memwwait</b>	[value]
<b>mpsset</b>	mps_id
<b>pagedn</b>	[window [pane]]
<b>pageup</b>	[window [pane]]
<b>parms</b>	{var1 [, var2, ..., varN]}
<b>poll</b>	run status value imm_var int_var query enable disable board_id mps_id
<b>post</b>	string
<b>prefer</b>	window command option [= value]
<b>print</b>	print_string
<b>quit</b>	[-f]
<b>read</b>	address mem_var src_var int_var imm_var [int_var reg_name reg_var] cache dcache icache set way [word field]
<b>readb</b>	address mem_var int_var imm_var [int_var reg_name reg_var]

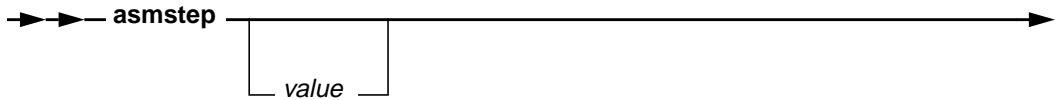
**Table 5-1. Syntax Summary for Debugger Commands**

<b>Command</b>	<b>Parameters</b>
<b>readh</b>	address mem_var int_var imm_var [int_var reg_name reg_var]
<b>read</b>	reg_name reg_var [int_var reg_name reg_var]
<b>reg</b>	reg_class reg_pre
<b>reset</b>	core chip sys
<b>restart</b>	
<b>retstep</b>	
<b>run</b>	[timeout][[to [address   file_name:line_num]]]
<b>save</b>	reg reginfo regfldinfo layout filename [reg_class reg_pre] mem filename address int_var imm_var bytes int_var imm_var bin binary filename address int_var imm_var [append]
<b>set</b>	(address) mem_var src_var int_var reg_name[.field_name .#] reg_var fld_var flt_var str_val = expression
<b>shell</b>	expression str_var
<b>showip</b>	
<b>socket</b>	timeout [value]
<b>srcdisp</b>	source mixed
<b>srchpath</b>	[q[query]] set dir1 (dir2 . . . dirN) add dir c[lear]
<b>srcline</b>	[imm_var int_var line]
<b>start_thread</b>	funcname [group_id]
<b>stop</b>	[timeout]
<b>stuff</b>	opcode "assembly" reg_name variable
<b>timer</b>	start stop

**Table 5-1. Syntax Summary for Debugger Commands**

<b>Command</b>	<b>Parameters</b>
<b>top</b>	[window [pane]]
<b>trace</b>	
<b>unassign</b>	all fld_var imm_var mem_var flt_var reg_var str_var
<b>uncreate</b>	all flt_var int_var str_var
<b>unload</b>	all filename
<b>up</b>	[int_var imm_var lines[window [pane]]]
<b>varinfo</b>	locals globals all none [addr][size][type]
<b>varvis</b>	locals globals vis invis
<b>view</b>	[filename]
<b>window</b>	[window [reg_pre] [mps_id]]   [cfss [filename]]   [udw [mps_id] filename]   [regfld [mps_id] regname [instance]]
<b>write</b>	dmem imem address mem_var int_var imm_var value int_var imm_var reg_name
<b>write</b>	src_var value int_var imm_var
<b>writeb</b>	dmem address mem_var int_var imm_var value int_var imm_var reg_name
<b>writew</b>	dmem address mem_var int_var imm_var value int_var imm_var reg_name

## Syntax



## Description

**asmstep** runs the processor for the execution of one or more 4-byte machine instructions.

If the *value* parameter is omitted, it defaults to 1.

## Flags

- value* Specifies the number of machine instructions the processor is to step.
- Note for 400Series JTAG targets: If the IAR is pointing to an RFI or RFCI instruction, processor requirements dictated that two instruction steps be taken to execute these instructions. This special case is handled automatically by the program.
- If the debugger is in source mode and the IAR is pointing to a branch instruction that will be taken, the debugger context will be switched to the target of the branch. This has the same effect as issuing a **callstep** instruction.

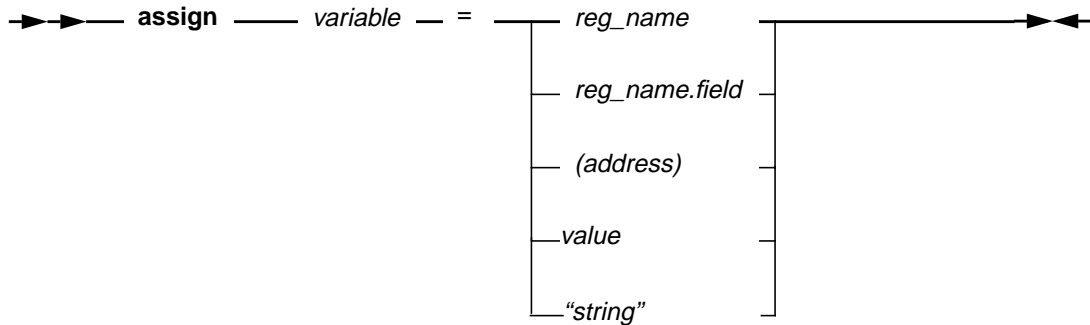
## See Also

- **callstep** on page 5-25

# assign

---

## Syntax



## Description

**assign** is used to assign a value to a variable name. The value can be an immediate value, a memory address value, a value in a register, or the value of a register field or character string. The name given to the variable must not start with a number or match any processor register name. Variable names are also case sensitive.

An immediate value can be any number given in floating point, octal, decimal or hexadecimal form. To assign the value of a register or field, the register or register field name or number is specified. A memory address is specified as an immediate value enclosed by the '(' and ')' characters to differentiate it from an immediate value.

Having assigned a value to a variable name, the variable name can be used in commands that accept *fld\_var*, *flt\_var*, *imm\_var*, *mem\_var*, or *str\_var* as valid input parameters. See Table 5-1 for a command syntax summary that shows which commands accept **assign** variables as parameters.

## Flags

- value* An initial data value
- (address)* The memory address assigned to the variable. Note that the () characters are used to distinguish a memory address from an immediate value.
- reg\_name* The name of the register assigned to the variable.
- reg\_name.field* The register name concatenated with the field name or bit number assigned to the variable.
- "string"* A character string, enclosed in double quotes, that is placed into the variable
- variable* The name given to the assigned variable so that it may be referenced in future commands



## Example

- Assign a register to a variable and then uses the variable to initialize and read the register's value.

```
assign count_reg = SPRG1      # make count_reg = SPRG1
set    count_reg = 0         # init count register
read   count_reg            # i.e. read SPRG1
```

- Assign an immediate value to a variable which is then used to initialize the value of a register.

```
assign reg_val = 0x11223344
set    SPRG0 = reg_val
```

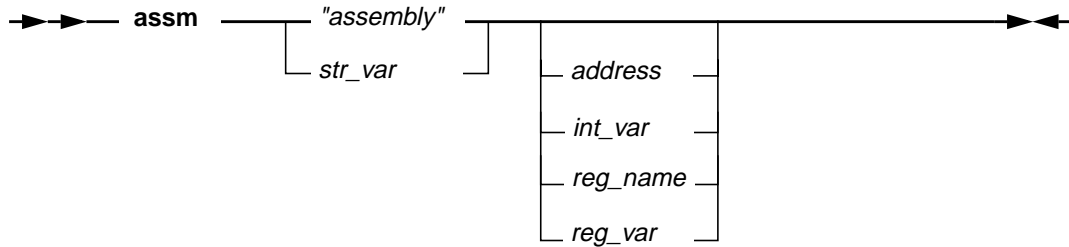
## See Also

- **create** on page 5-32
- **set** on page 5-110
- **unassign**

# assm

---

## Syntax



## Description

**assm** converts a valid assembly instruction into a 4-byte instruction value and then optionally writes this value to the specified register, user-created variable, or processor instruction memory at the specified address.

## Flags

- |                   |  |
|-------------------|--|
| <i>"assembly"</i> | A string containing a valid assembly instruction                   |
| <i>str_var</i>    | A string variable containing a valid assembly instruction          |
| <i>address</i>    | The memory address to write the assembled instruction value to     |
| <i>int_var</i>    | Any variable created with the <b>create</b> command                |
| <i>reg_name</i>   | The name of a register to write the assembled instruction value to |
| <i>reg_var</i>    | Any register variable created with the <b>assign</b> command       |
- Any operands that accompany an assembly instruction must consist of one contiguous string of characters. There can be no spaces between the operands if there are more than one.
- If no memory address, register name or user-created variable is specified, the string will simply be assembled and the subsequent machine instruction that is generated will be printed out in a status message.

## Examples

- Generate the instruction necessary to move the contents of a special purpose register to a general purpose register and then write the generated instruction at memory address 0xE0B15.  

```
assm "mfspr r13,LR" 0xE0B15
```
- Generate the instruction necessary to move the contents of a special purpose register to a general purpose register and then store the generated instruction in a user-created variable.

```
create asm_value  
asm "mfspr r13,LR" asm_value
```

- Generate the instruction necessary to move the contents of a special purpose register to a general purpose register and then write the generated instruction to register GPR8.

```
asm "mfspr r13,LR" R8
```

## See Also

- **dis** on page 5-36

# attach

---

## Syntax

➔➔ **attach** *threadid* ➔➔

## Description

**attach** initializes a source mode debug session with *threadid* under OS Open. *threadid* must be the number of an existing thread. A list of current threads can be found by clicking on the "List Threads" buttons of the OS Open window.

**Note:** RISCWatch cannot be used to debug the OS Open shell.

## Flags

*threadid*      The number of an existing thread

## Restrictions

This command is only supported for OS Open targets.

## Examples

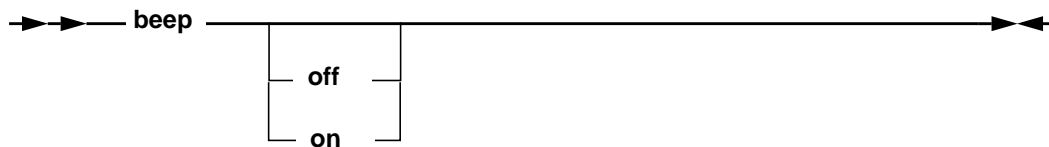
- Attach to an existing OS Open thread.

```
attach 0x31568
```

## See Also

- **detach** on page 5-35
- **kill\_thread** on page 5-66
- **start\_thread** on page 5-121

## Syntax



## Description

**beep** controls the program beeper. It may be used to turn the program beeps on or off or to sound the program beeper. If the **on** and **off** parameters are omitted, it sounds the program beeper.

## Flags

<b>off</b>	Turn the program beeper off
<b>on</b>	Turn the program beeper on

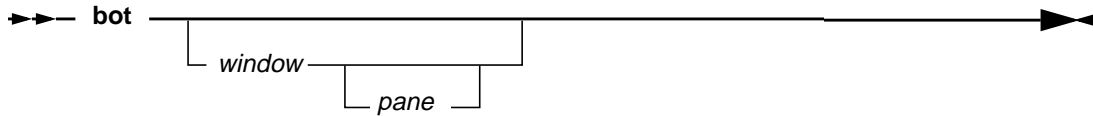
## Examples

- Turn the program beeper off  
`beep off`
- Turn the program beeper on  
`beep on`
- Sound the program beeper  
`beep`

# bot

---

## Syntax



## Description

**bot** scrolls to the last line of a window, highlighting the line if it contains any text.

If the *window* keyword is not specified, the last window specified for this command is used. It initially defaults to the Source window.

## Flags

*window* The window keyword applies to a subset of the windows listed in “Window Quick Reference” on page 5-2. The items marked with an asterisk (\*) refer to commands using both window and pane keywords.

*pane* See list of pane keywords in “Command Quick Reference” on page 5-3.

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

## Examples

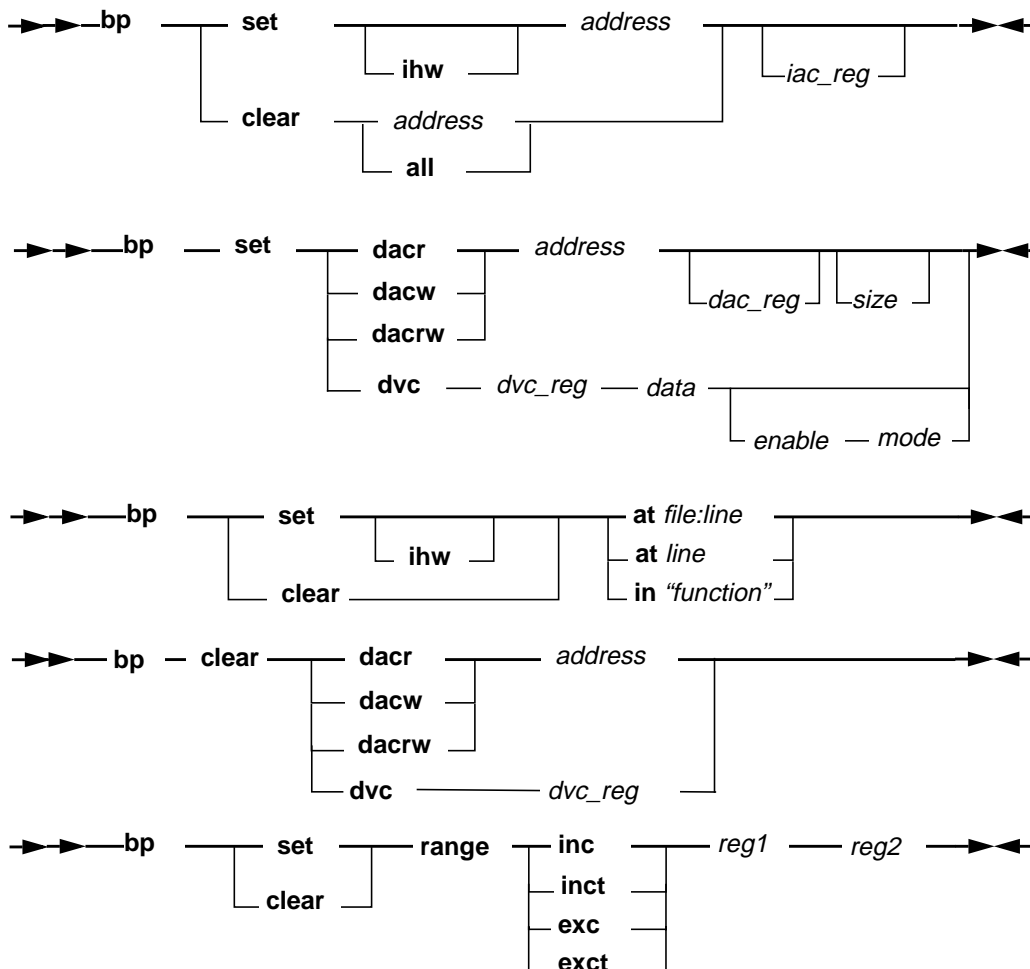
- Scroll to the last line of the window previously specified by this command.

```
bot
```

- Scroll to the last line of the Breakpoint window.

```
bot break
```

## Syntax



## Description

The **bp** command is used to set or clear hardware and software breakpoints.

Software instruction breakpoints are set using the 'bp set address' syntax. Hardware breakpoints are set by using the 'bp set' syntax with the desired pre-defined keyword (which uses the first available instruction/data breakpoint register to set an instruction/data breakpoint). The 'bp clear address' syntax applies to both hardware and software instruction breakpoints.

## Flags

<b>clear</b>	Clear one or all breakpoints
<b>set</b>	Set a breakpoint
<i>address</i>	Address of the data or instruction where the breakpoint should be set or cleared
<i>iacc_reg</i>	400Series: Instruction Address Compare Register Name (IAC1, IAC2, ...)
<b>all</b>	Remove all breakpoints(hardware and software)
<b>dacr</b>	400Series: Break on Data Address Compare Read
<b>dacw</b>	400Series: Break on Data Address Compare Write
<b>dacrw</b>	400Series: Break on Data Address Compare Read or Write
<i>dacc_reg</i>	400Series: Data Address Compare Register Name (DAC1, DAC2)
<i>size</i>	400Series(excluding 440): Data Address Compare Size bit settings (0,1,2, or 3)
<b>dvc</b>	405/440: Break on Data Value Compare
<i>dvc_reg</i>	405/440: Data Value Compare Register Name (DVC1, DVC2)
<i>enable</i>	405/440: Data Value Compare byte enable. Indicates which data bytes are to be compared. (Any combination of 0, 1, 2, and 3, or "ALL")
<i>mode</i>	405/440: Data Value Compare mode. Indicates how data bytes are to be compared. (AND, OR, AND-OR, or UNDEF to reset)
<b>ihw</b>	An optional parameter that is used to set a hardware instruction breakpoint using the first available instruction breakpoint register for the target processor.
<b>range</b>	405/440: Break on Address Range
<b>exc</b>	405/440: Address Range is exclusive
<b>exct</b>	405/440: Address Range is exclusive with toggle (toggle between exclusive and inclusive each time bp is hit). This option is only supported when <i>reg1</i> and <i>reg2</i> specify IAC registers.
<b>inc</b>	405/440: Address Range is inclusive
<b>inct</b>	405/440: Address Range is inclusive (toggle between inclusive and exclusive each time bp is hit). This option is only supported when <i>reg1</i> and <i>reg2</i> specify IAC registers.
<i>reg1</i>	405/440: Address Range lower bound register name. (IAC1, IAC3 for instruction range, DAC1 for data range)
<i>reg2</i>	405/440: Address Range upper bound register name. (IAC2 if <i>reg1</i> is IAC1, IAC4 if <i>reg1</i> is IAC3. DAC2 for data range)
<b>at</b>	Indicates a source file line number is to follow. Used when the environment is set to 'Source Mode On'.



---

<i>file:line</i>	A source file name followed by a decimal number indicating a specific source line.
<i>line</i>	A decimal number indicating a specific source line in the currently active file (the file displayed in the Source window, or last file specified with the <b>file</b> command).
<b>in</b>	Indicates a function name is to follow. Used when the environment is set to 'Source Mode On'.
<i>"function"</i>	<p>A case sensitive function name, as it would appear in the Functions window. If the surrounding quotes are omitted, the function name must be a non-blank character string. If the specified function is not found in the currently active file, the search continues in all remaining files defined by the currently active program (program containing the current instruction address).</p> <p>When searching outside the currently active file, global functions take precedence over functions defined as static and the first static function is used if no global definition is found.</p> <p>The break point will be set/cleared at the first line of the function (if line table information exists) or at the function start address if no line table information exists.</p>

## Notes

Data Value Compare breakpoints can only be set if the corresponding Data Address Compare breakpoint has already been set. Similarly, Range breakpoints can only be set if the corresponding Address Compare breakpoints have both been set for the specified range registers

For additional information on these and other processor debug features, consult the Debugging chapter of the User's Manual for the specific PowerPC 400Series processor being used. Not all features shown above are supported across all 400 Series processors.

## Examples

- Set a software breakpoint at address 0xFFFFF0.

```
bp set 0xFFFFF0
```

- Clear a breakpoint at address 0xFFFF00C0.

```
bp clear 0xFFFF00C0
```

- Clear all breakpoints.

```
bp clear all
```

- Set a hardware instruction breakpoint at address 0xFFFF00D0 using the first available instruction breakpoint register for the target processor.

```
bp set ihw 0xFFFF00D0
```

- Set a hardware instruction breakpoint at address 0xFFFF0C00 using the IAC2 hardware register.  

```
bp set ihw 0xFFFF00D0 IAC2
```
- Set a hardware instruction breakpoint at any address greater than or equal to 0xFFFF8000 but less than 0xFFFFFFFF0.  

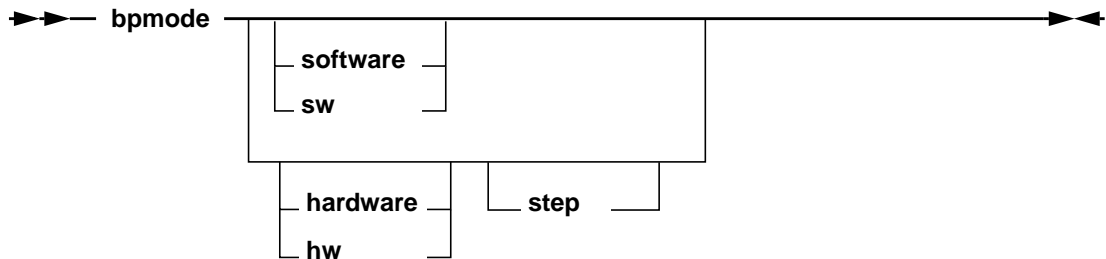
```
bp set ihw 0xFFFF8000 IAC1  
bp set ihw 0xFFFFFFFF00 IAC2  
bp set range inc IAC1 IAC2
```
- Set a Data Address Compare breakpoint at address 0x0000FFF0, using the DAC1 hardware register, and ignoring the LSB of the word address.  

```
bp set DACRW 0xFFF0 DAC1 1
```
- Set a Data Value Compare breakpoint at 0x00000000, if the first halfword of the data location equals 0x12 when read.  

```
bp set DACR 0x00000000 DAC2  
bp set DVC DVC2 0x1234 01 AND
```
- Set a Data Value Compare breakpoint at 0xFFFF1000, if the second or fourth byte of the word is written with 0xA5.  

```
bp set DACW 0xFFFF1000 DAC1  
bp set DVC DVC1 0x00A500A5 13 OR
```

## Syntax



## Description

**bpmode** is used to set or query the Breakpoint Mode used during debug. When the Breakpoint Mode is set to software (the default), operations to set breakpoints on the Source window, Assembly Debug window, and Functions window will result in a software breakpoint being set. When the Breakpoint Mode is set to hardware, operations to set breakpoints in these windows will result in a hardware breakpoint being set (if hardware facilities are available).

Entering the **bpmode** command with no parameters will echo the current Breakpoint Mode setting.

Note that the Breakpoint Mode can also be set via the Breakpoint Mode groupbox on the Breakpoints window.

## Flags

- hw | hardware** Set the Breakpoint Mode to hardware. All user generated breakpoints will be applied using hardware breakpoint registers. Breakpoints used during line step and call step operations are applied using software trap instructions if the step option is not used.
- step** Set the Breakpoint Mode to hardstep mode. All user specified breakpoints and internally generated breakpoints (those applied during line step and call step) will be applied using hardware breakpoint registers.
- sw | software** Set the Breakpoint Mode to software.

## Restrictions

This command is not supported in TTY mode.

# bpmode

---

## See Also

- “Assembly Debug Window” on page 3-55
- “Breakpoints Window” on page 3-74
- “Functions Window” on page 3-62
- “Managing Breakpoints” on page 3-71
- “Source Window” on page 3-52

---

## Syntax

▶▶ callstep ◀◀

## Description

**callstep** steps into the called routine.

**callstep** causes program control and debugger context to switch to the function call specified by the current source line. If the current line does not contain a function call, the command simply performs a line step.

If the current line contains a function call with functions in the parameter list (func1(func2(),func3());), then a **callstep** will first enter the function(s) found in the parameter list. A subsequent return step would return to the original function call source line. When all of the parameter list functions have been entered and returned from using **callstep/retstep** commands, the next **callstep** will transfer the debugger context to the function contained in the original call. In the above example, to enter func1, the first **callstep** would enter func2(). A **retstep** would return to the source line containing the func1 call. The next **callstep/retstep** would enter and then return from func3(). Finally, the next **callstep** would enter func1.

**Note:** If a **callstep** is issued into a function that has no associated debug information, a **retstep** command should be issued to return immediately to the calling function. Alternatively, a breakpoint should be set on the source line immediately following the function call to assure that the return can be made.

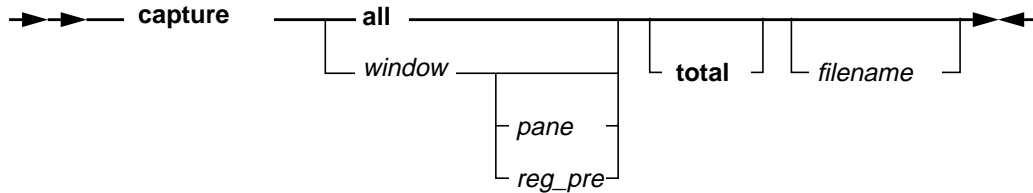
## See Also

- **bp** on page 5-19
- **retstep** on page 5-105

# capture

---

## Syntax



## Description

**capture** copies the contents of a user interface window and writes it to a file. The command options select which window's contents will be captured or all of the preceding choices (depending on the set of flags associated with a particular PowerPC processor).

To capture the contents to a specific file, simply put the filename as the last option on the command line. If no filename is supplied, a default name of **rwppc.cap** will be used. To best understand how this command works simply type **capture all** on the command line and then view the file **rwppc.cap**.

Source level debug windows (those included under the window parameter) will only be captured if the window is visible. The default for source level debug windows is to capture only the visible lines for a window. The **total** keyword can be used to capture the entire contents of any source level debug window except for the Source window. Only the visible lines will ever be captured for the Source window.

Be advised that the information saved into captured files cannot be loaded back into the window from which it was captured or to the processor. To store and restore a particular processor state of memory and/or registers, use the **save** and **load** commands.

## Flags

Some flags listed below are only applicable to particular target processors, as indicated in the description of those flags. The set of windows selected by the **all** flag is also processor-dependent.

**all** Specifies that the contents of all visible capturable windows are to be captured.

*filename* Specifies the name of the file to which the window capture is written.

**total** If this flag is specified, the entire contents of the window will be captured for those windows which support this flag (see table below). If this flag is NOT specified, only the visible contents will be captured.

For some windows which display processor data, using this flag may require a read of target resources since the window may not contain all necessary data due to performance restrictions.

**Note:** If the **all** option is specified, only the visible lines will ever be captured for the Source window. If the **total** option is used when specifying the Source window individually, the entire window will be captured without the status subwindow information. This option may be useful for capturing the contents of a file in mixed mode. When using the **total** option, care should be taken to ensure there is sufficient disk space to hold the desired screen information.

- pane* Some windows contain multiple panes of data. If a specific pane is specified, only its data will be captured. If a pane is NOT specified for a window with multiple panes, all panes for that window will be captured.
- The supported pane keywords are listed under “Command Quick Reference” on page 5-3.
- reg\_pre* When ASIC is specified for *window*, this specifies a unique ASIC window which contains the registers with the specified prefix. See the description for this flag in “Command Quick Reference” located at the beginning of this chapter.
- window* Any of the list of window keywords in “Window Quick Reference” on page 5-2.
- If a window has multiple instances and/or an associated MPS id, the appropriate instance number and/or MPS id must be specified as it appears in the window title bar.

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

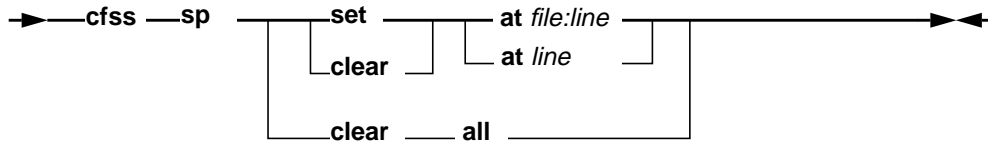
**Table 5-2. Windows that support capture and total**

capture supported	total supported	capture supported	total supported	capture supported	total supported
Cache	Yes	Register Field	No	Locals	Yes
Memory Access	Yes	Breakpoints	Yes	OS Open	Yes
Memory ASCII	No	Callers	Yes	Programs	Yes
Memory Custom	No	Files	Yes	Source	Yes
Debug	No	Functions	Yes	TLB	Yes
Output	Yes	Globals	Yes	User-Defined	No
Register	No	Inspect	Yes		

# cfss

---

## Syntax



## Description

**cfss** is used to set and clear stop points in the Command File Window. A stop point is used to halt command file execution at a specific line number. Stop points remain active for the entire RISCWatch debug session and can only be removed using the 'cfss clear' command.

## Flags

<b>sp</b>	Specifies that a stop point command is being issued.
<b>set</b>	Specifies that a stop point is to be set.
<b>clear</b>	Specifies that a stop point is to be removed.
<b>all</b>	Specifies that all stop points are to be removed.
<b>at file:line</b>	Specifies the file name and line number for the stop point. If the file name cannot be located using the RISCWatch search path, an error message is generated.
<b>at line</b>	Specifies the line number for the stop point. The command file name is assumed to be the currently active file shown in the Command File Window.

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

## Examples

- Set a stop point at line 5 of command file 'startup.cmd'.  

```
cfss sp set startup.cmd:5
```
- Remove the stop point located at line 5 of the file shown in the Command window.  

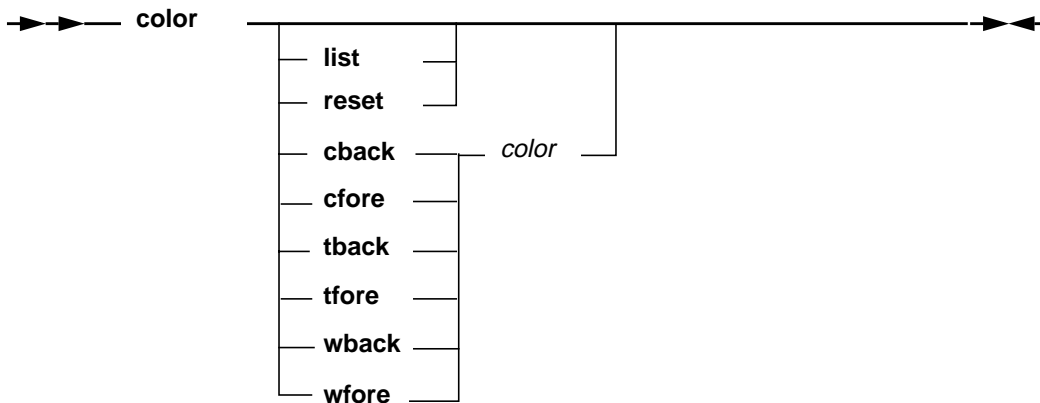
```
cfss sp clear 5
```

## See Also

- See "Command File Window" on page 3-132



## Syntax



## Description

**color** is used to change window color settings. The settings specified by this command are applied to any subsequent window creations.

The **color** command can be used to override the default settings, as well as any settings previously defined in the environment file, **rwppc.env**. If no keyword is specified, the current settings will be displayed.

## Flags

<b>list</b>	Lists all supported color names (e.g. red, blue, peachpuff, goldenrod, etc.).
<b>reset</b>	Specifies that the color settings are to be reset to the original session values.
<b>cback</b>	Specifies the color setting for the background control areas.
<b>cfore</b>	Specifies the color setting for the foreground control areas.
<b>tback</b>	Specifies the color setting for the background text areas.
<b>tfore</b>	Specifies the color setting for the foreground text areas.
<b>wback</b>	Specifies the color setting for the background window areas.
<b>wfore</b>	Specifies the color setting for the foreground window areas.
<i>color</i>	Name or hex representation of the desired color. For a list of color names, use the <b>list</b> option. The hex representation is of the form "0xrrggbb" which defines the red, green, and blue components of the color, respectively.

# color

---

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

## Examples

- Change the window background color to blue.

```
color wback blue
```

- Change the control foreground color to blue using the hex representation.

```
color cfore 0xFF
```

## Notes

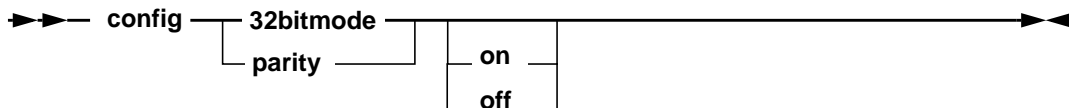
When in MPS mode, use the **mpsset** command followed by a **color** command to select a unique set of color attributes to be used for each, individual MPS context. Doing so will allow windows to be easily identified with regard to their context when many are displayed at the same time. Color attributes are stored for each MPS context during a debug session.

Color for MPS neutral windows can only be changed using settings defined in the environment file.

## See Also

- See "Environment Resources" on page 3-5

## Syntax



## Description

**config** configures RISCWatch to match different hardware options for a particular processor. Selecting an option without a value setting will display the current setting.

**Note:** RISCWatch cannot automatically detect the processor's settings, nor can it change the mode of the processor itself.

## Flags

- 32bitmode** Used to display or set RISCWatch's **32bitmode** setting. This setting must match the 32bitmode setting of the processor's hardware for correct RISCWatch operation.
- parity** Used to display or change RISCWatch's data **parity** generation setting. For performance reasons, RISCWatch does not typically generate data parity bits on memory accesses. However, some systems may require parity generation.

## Restictions

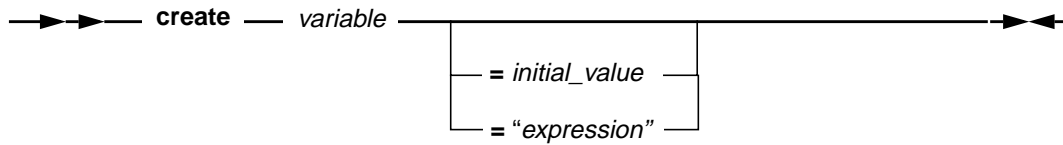
This command is only supported for JTAG Ethernet targets.

The **32bitmode** and **parity** flags are only supported on the 603e, 603ev, 740 and 750 processors.

# create

---

## Syntax



## Description

**create** is used to create a variable. The variable value is stored as a signed quantity (allocated in multiples of 4 bytes), a float, or an expression. The name given to the variable may not start with a number and must not match any processor register name. Variable names are also case sensitive.

The variable can be used in any command that allows *int\_var*, *flt\_var*, or *str\_val* as valid input parameters. See Table 5-1 for a command syntax summary that shows which commands accept **create** variables as parameters.

It is possible to assign an initial value to the variable. If no initial value is specified when creating a variable, a value of 0 will be assigned.

## Flags

*“expression”* An expression, enclosed in double quotes, that is placed into the created variable. This creates a string variable.

*variable* Name of the immediate variable to be created.

*initial\_value* The value assigned to the variable after it is created. If an initial value is not specified, a value of 0 will be assigned.

## Examples

- Create a variable named `cr_var1` and assign it an initial value of 0x1234.

```
create cr_var1 = 0x1234
```

- Create a variable named `cr_var2` and assign it an initial value of 0.

```
create cr_var2
```

- Create a variable named `my_string` and assign it the string “My string contents”.

```
create my_string = "My string contents"
```

- Create two variables, i and j, and use them to calculate a value to write to GPR0.

```
create i           # create variable i
create j          # create variable j
set i = (0x12345678) # read memory into i
set j = i - IAR    # subtract IAR from i
write R0 j        # write value of j to GPR 0
```

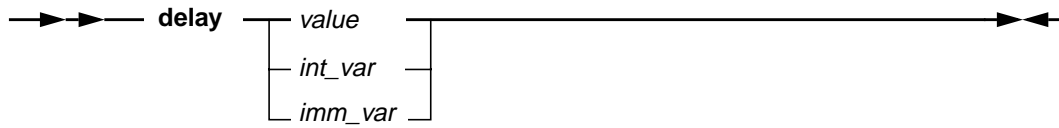
## See Also

- **assign** on page 5-12
- **set** on page 5-110

# delay

---

## Syntax



## Description

**delay** is used to delay the execution of a command file for the specified number of seconds. During this delay period, no program or command file processing is performed.

## Flags

- value* Specifies the number of seconds to delay execution
- int\_var* Any int variable created with the **create** command
- imm\_var* Any immediate variable created with the **assign** command

## Restrictions

This command is only supported in command file mode.

## Syntax

▶▶— detach —▶▶

## Description

**detach** ends a source mode debug session by disconnecting from the thread or process being debugged. The thread or process then continues to run normally.

## Restrictions

This command is only supported in OS Open and ROM Monitor modes.

## Examples

- Detach from the thread or process being debugged.

```
detach
```

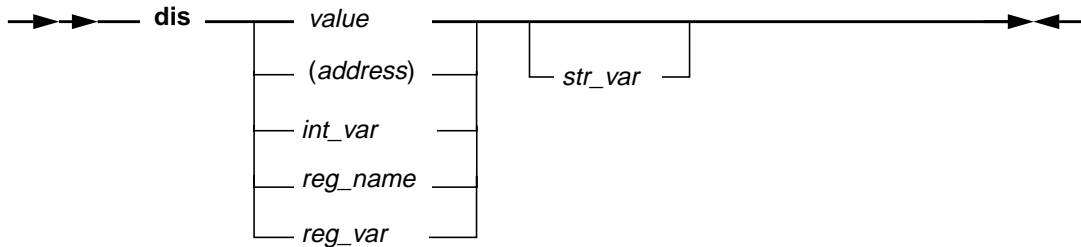
## See Also

- **attach** on page 5-16
- **kill\_thread** on page 5-66
- **start\_thread** on page 5-121

# dis

---

## Syntax



## Description

**dis** is used to disassemble a 4-byte instruction value. The result, by default, is printed as a mnemonic and its operands in assembly code but can also be stored into a string variable. The options for this command allow disassembly of an immediate value or the contents of a specified processor memory location, register or user-variable.

## Flags

<i>value</i>	Specifies an immediate numeric value.
<i>(address)</i>	Specifies a memory location which will be read and its contents then disassembled. Note that the () characters are used to distinguish a memory address from an immediate value.
<i>int_var</i>	Any int variable created with the <b>create</b> command.
<i>reg_name</i>	Specifies any valid register name whose value will be disassembled.
<i>reg_var</i>	Any register variable created with the <b>assign</b> command.
<i>str_var</i>	Any string variable created with the <b>assign</b> or <b>create</b> commands.

## Examples

- Disassemble an immediate value.

```
dis 0x38000000
```
- Disassemble the instruction that resides at a given memory address.

```
dis (0x1D3F0004)
```
- Disassemble the value contained in a user-created variable.

```
create dis_val = 0x38000000
dis dis_val
```



- Disassemble the value contained in a register and store the result in a user-created string variable.

```
write R14 0x38000000
create my_str = ""
dis R14 my_str
```

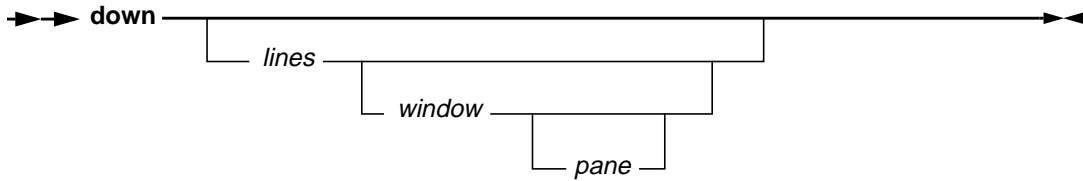
## See Also

- **asm** on page 5-14

# down

---

## Syntax



## Description

**down** scrolls the contents of a window down one or more lines from the top line visible in the window.

The *lines* variable initially defaults to 1. If the value specified for *lines* is larger than the number of lines left in the window, the last line is shown at the bottom of the window.

If the *window* keyword is not specified, the last window specified for this command is used. It initially defaults to the Source window.

If neither the *lines* variable nor the *window* keyword is specified, the last *lines* value and *window* keyword specified for the command are used.

## Flags

- |               |  |
|---------------|--|
| <i>lines</i>  | Specifies the number of lines to be scrolled down.   |
| <i>window</i> | The window keyword applies to a subset of the windows listed in “Window Quick Reference” on page 5-2. The items marked with an asterisk (*) indicate the subset of valid window keywords for this command. |
| <i>pane</i>   | See list of pane keywords in “Command Quick Reference” on page 5-3.  |

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

## Examples

- Scroll down one line in a window previously specified, or the Source window if none has been specified previously.

```
down
```

- Scroll down 10 lines in a window previously specified, or the Source window if none has been specified previously.

```
down 10
```

- Scroll down 12 lines in the global variables window.

```
down 12 globals
```

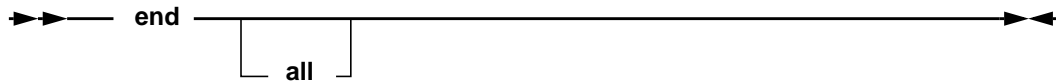
## See Also

- **up** on page 5-131

# end

---

## Syntax



## Description

`end` is used to end the execution of the current command file. `end all` is used to end the execution of all command files, regardless of nesting.

## Restrictions

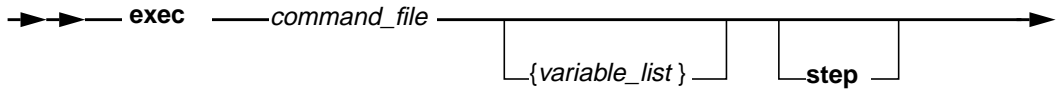
This command is only supported in command file mode.

## Examples

- End execution of the current command file.

```
if (R0 != 0xFC001234)
    end
endif
```

## Syntax



## Description

**exec** is used to execute the instructions contained in a command file. See the Command Files section for more details on command file creation and usage.

## Flags

*command\_file* The name of the command file to be executed. For example, `test.cmd`. For further information, see “Command File Programming” on page 3-125.

*variable\_list* A list of variable values to be passed into the command file and assigned to the variables in the **parms** parameter definition. See “Command File Parameters” on page 3-128 for more details.

**step** Runs the command file in single-step mode. This option is only valid when a command file is executed from the user interface. It causes a Command File window to be created, which is equivalent to issuing the “`window cfss command_file`” command. See “Command File Window” on page 3-132 for more details.

## See Also

- **window** on page 5-138

# expr

---

## Syntax

→ → — **expr** — *expression* ————— → ←

## Description

**expr** is used to evaluate an *expression* and print the results in a status message. For a complete description of the *expression* syntax see the **set** command.

The **expr** command outputs the result of the *expression* in hexadecimal, signed decimal and unsigned decimal forms. Having such a capability allows users to test out expressions before they are used on the command line or in a command file. It also allows numbers to be displayed in multiple radices (hexadecimal, decimal, and unsigned decimal). To display a number in its alternate base, simply type it in after the **expr** command keyword.

## Flags

*expression* = [(*logical*|*mathematical* *logical*)]  
*logical* = *expression*|*expression* *log\_op* *expression*  
*mathematical* = [*math\_op1*] *expression* [*math\_op2* *mathematical*]  
*expression* = *reg\_name*[.*fld\_name*].#](*address*)|*immed*|*variable*|*mem\_var*|*func*  
*func* = supported functions : random()  
*log\_op* = == != > >= < <=  
*math\_op1* = + - ~  
*math\_op2* = + - \* / mod % & | ^ << >>  
# = ordinal bit number

## Examples

- Display the result of adding 10 to GPR0.  

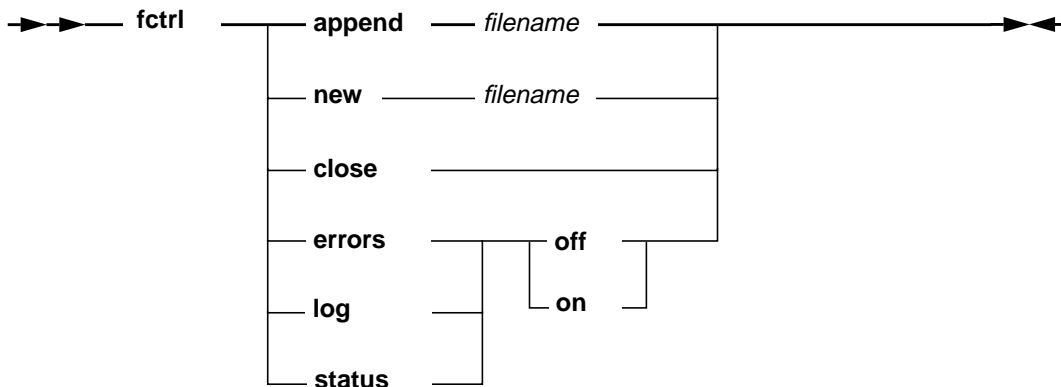
```
expr R0 + 10
```
- Display the value 10 in hexadecimal, decimal, and unsigned decimal.  

```
expr 10
```

## See Also

- **set** on page 5-110

## Syntax



## Description

**fctrl** controls access of the print files used by the **fprint** command.

## Flags

<b>append</b>	Open a print file. If the file exists, it will be opened and all messages will be appended to the end of the file.
<b>new</b>	Open a print file. If the file exists, it will be erased.
<b>close</b>	Close the print file.
<b>errors</b>	This flag controls whether or not program error messages are copied to the print file.
<b>log</b>	This flag controls whether or not log messages are copied to the print file.
<b>status</b>	This flag controls whether or not program status messages are copied to the print file.
<b>off</b>	Disables message copying.
<b>on</b>	Enables message copying.
<i>filename</i>	The name of the print file to open.

## Examples

- Open a new file for printing.
 

```
fctrl new print.dat
```
- Enable copying of error messages to the print file.
 

```
fctrl errors on
```

# fctrl

---

- Close an open print file.

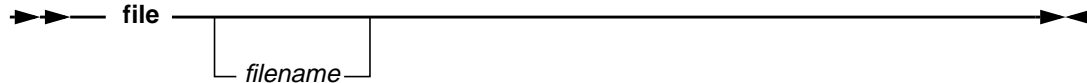
```
fctrl close
```

## See Also

- **fprint** on page 5-55
- **print** on page 5-97



## Syntax



## Description

**file** sets the current source file to *filename* (if specified) and displays it in the Source window if the Source window is active. Entering **file** without specifying a *filename* displays the name of the current file, if available.

**file** can be used in conjunction with the 'at' and 'in' options of the **bp** command to set the current file used by those options.

Only files which belong to the program currently being debugged, and which were compiled to contain debug information, can be displayed using this command. The valid file names are those which are shown in the Files window.

## Flags

*filename* Specifies the name of the source file to make current and display in the Source window.

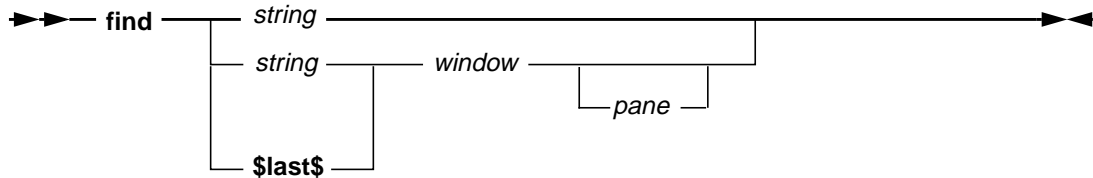
## Restrictions

This command is not supported in TTY mode.

# find

---

## Syntax



## Description

**find** searches for a string in a window, scrolling to the line containing the string, and highlighting the string if found.

The search is case-insensitive ('non-exact'). If no text is currently highlighted, the search will begin from the beginning of the top line visible in the window. If there is text highlighted, the search will begin from either the first character of the selected text (an 'initial' search), or from the character immediately following the first character of the highlighted text (a 'next' search). The **focus** command can be used to locate highlighted text.

If no parameters are specified, the string last specified for a **find** command (**find**, **findb**, **finde**) is used, and a 'next' search is done. This allows the user to initially specify a string, and find subsequent occurrences of the string in the same window by simply entering a **find** command repeatedly. A 'next' search will also be done if the string and window values match those of the last attempted **find** command. This allows the user to initially specify a string, and find subsequent occurrences of the string in the window by double-clicking on the command in the command history list of the Main window.

If the *string* variable is specified, and the *string* and *window* values do not match those of the last attempted **find**, an 'initial' search is done. If the *window* keyword is not specified, the last window specified for this command is used. It initially defaults to the Source window.

If the keyword **\$last\$** is specified in place of *string* and a *window* is specified, the string specified for the last **find** command is used, and a 'next' search is done for the specified window. This allows a window different from the window specified in the previous search to be searched for the same string specified in the previous search.

This function is also available via the input line, as described in "Input Line Usage" on page 3-49.

## Flags

- |               |  |
|---------------|--|
| <i>string</i> | Sequence of characters to be found. If the sequence of characters is to contain a space, then the entire string must be enclosed in quotation marks.   |
| <i>window</i> | The window keyword applies to a subset of the windows listed in "Window Quick Reference" on page 5-2. The items marked with an asterisk (*) indicate the subset of valid window keywords for this command. |

*pane* See list of pane keywords in “Command Quick Reference” on page 5-3.

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

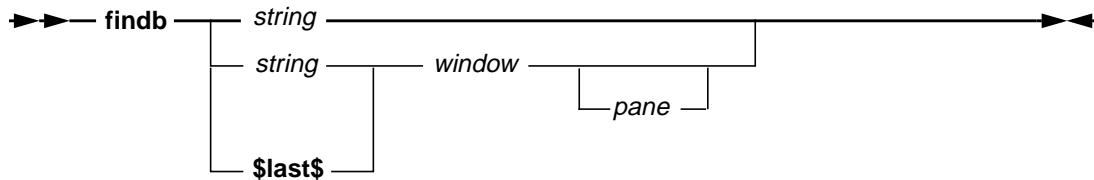
## See Also

- **findb** on page 5-48
- **finde** on page 5-50
- **focus** on page 5-52

# findb

---

## Syntax



## Description

**findb** searches backwards for a string in a window, scrolling to the line containing the string, and highlighting the string if found.

The search is case-insensitive ('non-exact') or case-sensitive ('exact'), depending on the type of forward search (**find** or **finde**) which was done previously. If no forward search was done previously the command defaults to a 'non-exact' search.

If no text is currently highlighted, the search will begin from the end of the bottom line visible in the window. If there is text highlighted, the search will begin from either the last character of the selected text (an 'initial' search), or from the character immediately preceding the last character of the highlighted text (a 'next' search). The **focus** command can be used to locate highlighted text.

If no parameters are specified, the string last specified for a 'find' command (**find**, **findb**, **finde**) is used, and a 'next' search is done. This allows the user to initially specify a string, and find subsequent occurrences of the string in the file by simply entering a 'find' command repeatedly. A 'next' search will also be done if the string and window values match those of the last attempted 'find' command. This allows the user to initially specify a string, and find subsequent occurrences of the string in the window by double-clicking on the command in the command history list of the Main window.

If the *string* variable is specified, and the string and window values do not match those of the last attempted 'find' command, an 'initial' search is done. If the *window* keyword is not specified, the window specified for the last 'find' command is used. It initially defaults to the Source window.

If the keyword **\$last\$** is specified in place of *string* and a window is specified, the string specified for the last **find** command is used, and a 'next' search is done for the specified window. This allows a window different from the window specified in the previous search to be searched for the same string specified in the previous search.

This function is also available via the input line, as described in "Input Line Usage" on page 3-49.

## Flags

<i>string</i>	Sequence of characters to be found. If the sequence of characters is to contain a space, then the entire string must be enclosed in quotation marks.
<i>window</i>	The window keyword applies to a subset of the windows listed in “Window Quick Reference” on page 5-2. The items marked with an asterisk (*) indicate the subset of valid window keywords for this command.
<i>pane</i>	See list of pane keywords in “Command Quick Reference” on page 5-3.

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

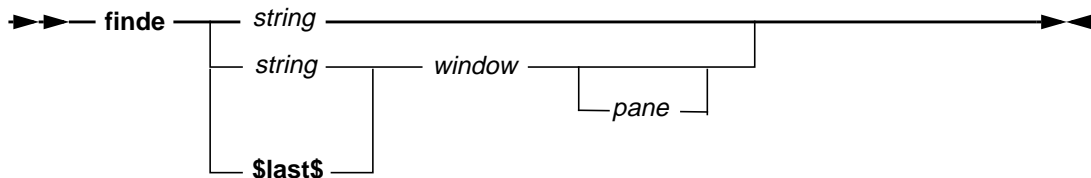
## See Also

- **find** on page 5-46
- **finde** on page 5-50
- **focus** on page 5-52

# finde

---

## Syntax



## Description

**finde** searches for a string in a window, scrolling to the line containing the string, and highlighting the string if found.

Unlike the **find** command, **finde** does an case-sensitive ('exact') search. If no text is currently highlighted, the search will begin from the beginning of the top line visible in the window. If there is text highlighted, the search will begin from either the first character of the selected text (an 'initial' search), or from the character immediately following the first character of the highlighted text (a 'next' search). The **focus** command can be used to locate highlighted text.

If no parameters are specified, the string last specified for a **finde** command (**find**, **findb**, **finde**) is used, and a 'next' search is done. This allows the user to initially specify a string, and find subsequent occurrences of the string in the same window by simply entering a **finde** command repeatedly. A 'next' search will also be done if the string and window values match those of the last attempted **finde** command. This allows the user to initially specify a string, and find subsequent occurrences of the string in the window by double-clicking on the command in the command history list of the Main window.

If the *string* variable is specified, and the *string* and *window* values do not match those of the last attempted **finde**, an 'initial' search is done. If the *window* keyword is not specified, the last window specified for this command is used. It initially defaults to the Source window.

If the keyword **\$last\$** is specified in place of *string* and a *window* is specified, the string specified for the last **finde** command is used, and a 'next' search is done for the specified window. This allows a window different from the window specified in the previous search to be searched for the same string specified in the previous search.

This function is also available via the input line, as described in "Input Line Usage" on page 3-49.

## Flags

- |               |  |
|---------------|--|
| <i>string</i> | Sequence of characters to be found. If the sequence of characters is to contain a space, then the entire string must be enclosed in quotation marks.   |
| <i>window</i> | The window keyword applies to a subset of the windows listed in "Window Quick Reference" on page 5-2. The items marked with an asterisk (*) indicate the subset of valid window keywords for this command. |

*pane* See list of pane keywords in “Command Quick Reference” on page 5-3.

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

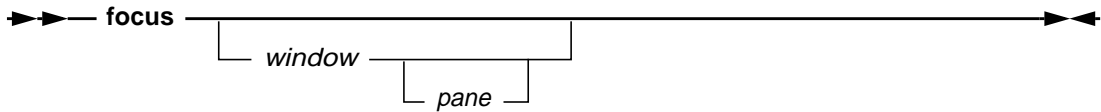
## See Also

- **find** on page 5-46
- **findb** on page 5-48
- **focus** on page 5-52

# focus

---

## Syntax



## Description

**focus** scrolls to the line of a window which has text highlighted, if any.

If no text is currently highlighted in the window, a message is generated stating this fact. If the *window* keyword is not specified, the last window specified for this command is used. It initially defaults to the Source window.

## Flags

- |               |  |
|---------------|--|
| <i>window</i> | The window keyword applies to a subset of the windows listed in “Window Quick Reference” on page 5-2. The items marked with an asterisk (*) indicate the subset of valid window keywords for this command. |
| <i>pane</i>   | See list of pane keywords in “Command Quick Reference” on page 5-3.  |

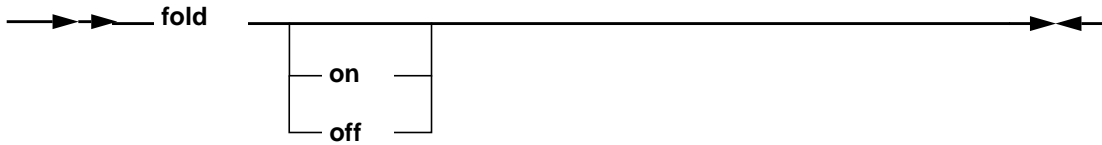
## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.



## Syntax



## Description

**fold** controls instruction folding. Refer to the applicable PowerPC processor documentation for detailed information on instruction folding.

A **fold** setting is effective only until the next processor system reset. After a reset, the **fold** setting defaults to 'on'.

If no parameter is entered, the current **fold** setting is displayed.

## Flags

- |            |                                |
|------------|--------------------------------|
| <b>off</b> | Turns instruction folding off. |
| <b>on</b>  | Turns instruction folding on.  |

## Restrictions

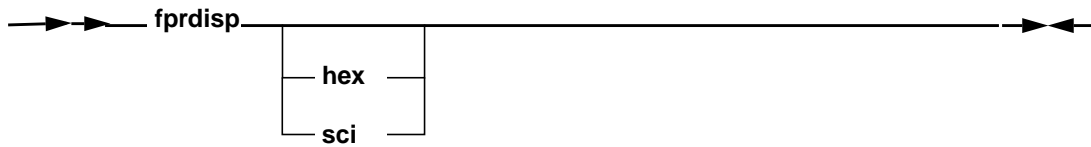
This command is only supported on JTAG targets.

This command is only supported on 403xx processors.

# fprdisp

---

## Syntax



## Description

**fprdisp** controls the display mode of the Floating Point Register window. The default is to display the values on the screen in hexadecimal notation.

If no parameter is entered, the display setting is toggled.

## Flags

- hex** Display FPR window values in hexadecimal notation.
- sci** Display FPR window values in scientific notation.

## Restrictions

This command is only supported on 6xx and 7xx processors.

## Syntax

—▶▶▶ **fprint** — *print\_string* —▶▶▶

## Description

**fprint** prints user definable strings to a print file that was opened with the **fcntl** command.

String literals are ASCII text enclosed by quotation (") marks. The text between the quotation marks is echoed to the print file. A string literal is also used to enclose character constants to help format the printed text :

<u>Constant</u>	<u>Meaning</u>
<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	Newline
<code>\r</code>	Carriage return
<code>\t</code>	Tab

User-created variable values may also be printed to the print file if they appear in the print string. Expressions containing variables and constants may also be used.

Variable values printed to the print file can be written in a variety of forms. Available options include the ability to print integers as signed or unsigned, hexadecimal values and characters.

The syntax for using variable formatting is as follows :

```
variable[/ [+] [[0]#] c|i|u|x|E|X]
```

where

/	Terminates the string to be formatted
+	Prints an integer preceded by a + or - sign. This option is only valid for the i format.
#	Specifies that at least # characters are printed. If the result contains less than # characters, the output will be left-padded with spaces. This option is only valid for the i, u, x, and X formats.
0	This option, if included, must always precede the # option. This specifies that at least # characters are printed. If the result contains less than # characters, the output will be left-padded with 0s. This option is only valid for the i, u, x, and X formats.
c	Prints a value as a series of four ASCII characters. Unprintable characters are output as a period (.).
i	Prints a value as a signed integer.

# fprint

---

- u Prints a value as an unsigned integer.
- x Prints a value as a hexadecimal integer. The letters a, b, c, d, e, and f appear in the output.
- X Prints a value as a hexadecimal integer. The letters A, B, C, D, E, and F appear in the output.
- E Prints a value in scientific notation (i.e. n.nnE-n).
- X Prints a value as a hexadecimal integer. The letters A, B, C, D, E, and F appear in the output.

To use variable formatting, place the / character immediately after the last character of the variable name and then follow it with the formatting options you desire. To format expressions, place the formatting options directly after the last argument in the expression. For example:

```
fprint addr + 0x1234 / 4/08X
```

A single **fprint** statement may contain multiple string literals, variables and expressions in any order. If this is done, each item in the command must be separated with a comma (,).

The following pseudo-variables may be used in the **print** and **fprint** commands for your convenience :

- \$DATE This will be replaced by a string which contains the current date in the format DAY MONTH DATE YEAR.
- \$ERRORS This will be replaced by a string which contains the number of errors generated by executed commands.
- \$TIME This will be replaced by a string which contains the current time in the format HOUR:MINUTE:SECOND.
- \$TIMER This will be replaced by a string which contains the number of seconds in the clock timer. See the **timer** command for more details.

## Flags

- print\_string* This is a user definable string containing string literals, user-created variable names and the same type of expressions used in the **set** command.

## Examples

- The following commands implement a short loop which writes successive memory locations, reads back what was written and prints the result of the comparison between the two values :

```
fctrl new test.mem
fprint "Start : ", $TIME, "\n"
create mem_addr = 0x0000FFFF
while (mem_addr < 0x00010000)
    fprint "Addr : ", mem_addr/08X
    fprint "\n"
    write dmem mem_addr 0xFFA55AFF
    read mem_addr S0
    if (S0 == 0xFFA55AFF)
        fprint "Test : PASSED\n\n"
    elseif
        fprint "Test : FAILED\n\n"
    endif
    set mem_addr = mem_addr + 1
endwhile
fprint "End : ", $TIME, "\n"
fctrl close
```

## See Also

- **fctrl** on page 5-43
- **print** on page 5-97

# freeze

---

## Syntax



## Description

**freeze** controls how and when the processor timers are to be frozen.

A **freeze** setting is effective only until the next processor reset. After any reset, the **freeze** setting defaults to 'never'.

If no parameter is entered, the current **freeze** setting is displayed.

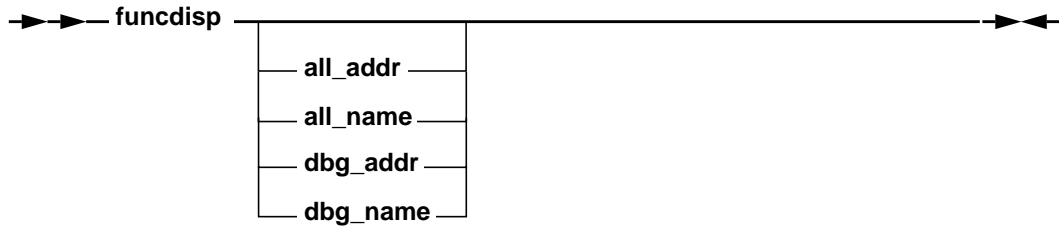
## Flags

- always** Forces timers to be frozen regardless of the processor state.
- never** Forces timers to be free running (not frozen) at all times regardless of the processor state.
- stop** Forces timers to be frozen whenever the processor is stopped. Timers will remain stopped until the next run is performed.

## Restrictions

This command is only supported on 4xx processors.

## Syntax



## Description

**funcdisp** changes the Functions window display to show either all functions in the program sorted by address (**all\_addr**), all functions in the program sorted by name (**all\_name**), functions with symbolic debug information sorted by address (**dbg\_addr**), or functions with symbolic debug information sorted by name (**dbg\_name**). This is the same capability provided by the Functions Mode groupbox on the Functions window.

Entering the **funcdisp** command with no parameters will toggle the current Functions window display (from functions with symbolic debug information to all functions, or the reverse), while keeping the sort algorithm for the display (by name or by address) the same as the current display.

## Flags

- all\_addr** Sets the Functions window display to show all functions in the program, sorted by addr.
- all\_name** Sets the Functions window display to show all functions in the program, sorted by name.
- dbg\_addr** Sets the Functions window display to show only functions with symbolic debug information, sorted by addr.
- dbg\_name** Sets the Functions window display to show only functions with symbolic debug information, sorted by name.

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

## Example

- Set the Functions window display to show all functions in the program, sorted by address

```
funcdisp all_addr
```

# funcdisp

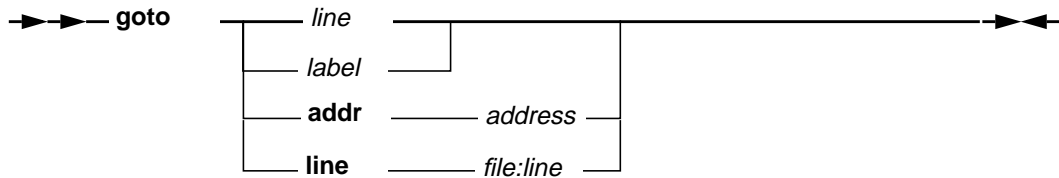
---

## See Also

- “Functions Window” on page 3-62



## Syntax



## Description

**goto** causes the source line designated by *line* to be the next source line run. The specified source line must be in the same function as the current source line.

## Flags

<i>label</i>	Specifies the location within a command file to transfer execution control.
<i>line</i>	Specifies the next source line to be run in the file which contains the current instruction.
<b>addr</b>	Specifies execution is to be resumed at a given address.
<b>line</b>	Specifies that execution is to be resumed at a given filename:line_number.
<i>address</i>	Address at which to resume execution.
<i>file:line</i>	File name and line number at which to resume execution.

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

## Example

- Change the next source line to be executed to line 100 of the current file.

```
goto 100
```

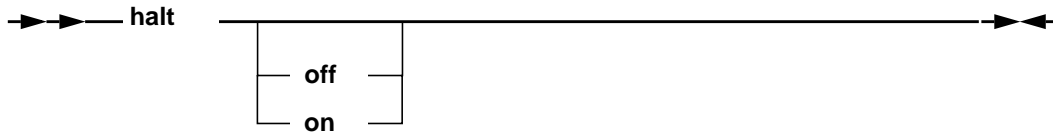
- Change the next source line to be executed to line 10 of a file.

```
goto line hello.c:100
```

# halt

---

## Syntax



## Description

**halt** controls the state of the processor  $\overline{\text{Halt}}$  line. If neither the **on** nor the **off** parameter is specified, it displays the current  $\overline{\text{Halt}}$  line state.

## Flags

- on**            Activate the  $\overline{\text{Halt}}$  line.
- off**           Deactivate the  $\overline{\text{Halt}}$  line.

## Restrictions

This command is only supported on 4xx processors.

## Syntax

→ → hidewins ← ←

## Description

**hidewins** hides all the currently visible RISCWatch windows except for the Main window.

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

# ip

---

## Syntax

▶▶ — ip —————▶▶

## Description

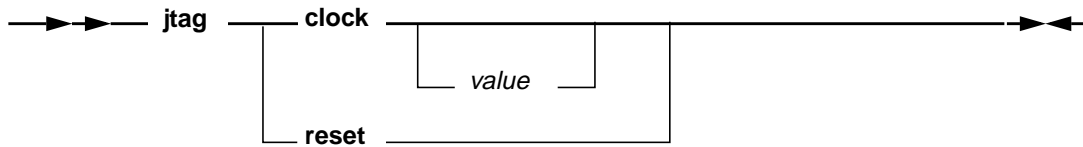
**ip** generates messages in the I/O window giving the current Instruction Pointer address, as well as the Function, File, Line Number, and Current Program associated with the **ip** address if there is debug information available corresponding to it.

For JTAG targets, the Instruction Pointer is actually the current Instruction Address Register (IAR). For non-JTAG targets, it is the process copy of the IAR for the application being debugged.

## See Also

- **showip** on page 5-115

## Syntax



## Description

**jtag** displays or sets the JTAG TCK clock speed on the RISCWatch Processor Probe. It can also be used to reset the JTAG controller to a known state.

## Flags

*value* Specifies the clock speed to set, where:

- 1 = 10MHz
- 2 = 5MHz
- 3 = 2.5MHz
- 4 = 1.25MHz
- 5 = 625KHz
- 6 = 312.5KHz
- 7 = 156.25KHz

for the older probes.

For the newer probes, values 1 to 7 are still supported but the actual value of the JTAG clock will be different based on the processor. The values that should be used are 512K, 1M, 2M, 3M ... 40M. For example, “jtag clock 20M” will set the jtag clock to 20MHz.

**reset** Specifies that the JTAG controller should be reset.

## Restrictions

This command is only supported for JTAG targets.

# kill\_thread

---

## Syntax

▶▶ — kill\_thread —▶▶

## Description

**kill\_thread** ends a source mode debug session with OS Open by destroying the thread which is currently being debugged.

## Restrictions

This command is only supported for OS Open targets.

## Examples

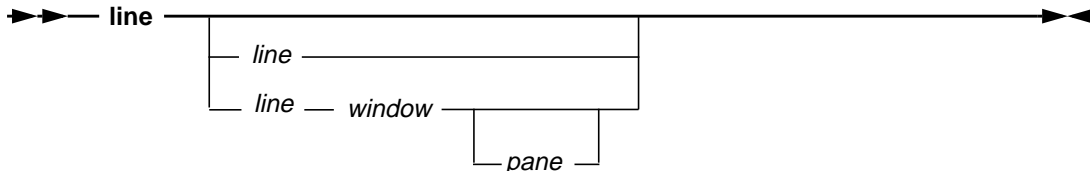
- Kill the current thread

```
kill_thread
```

## See Also

- **attach** on page 5-16
- **detach** on page 5-35
- **start\_thread** on page 5-121

## Syntax



## Description

**line** scrolls the contents of a window to a physical line of text in the window.

If the line number specified is larger than the number of lines in the window, the last line is shown at the bottom of the window. If the *window* keyword is not specified, the last window specified for this command is used. It initially defaults to the Source window. If neither the *line* number nor the *window* keyword is specified, the last *line* number and *window* specified for the command are used. The *line* number initially defaults to 1.

This function is also available via the input line, as described in “Input Line Usage” on page 3-49.

## Flags

- |               |  |
|---------------|--|
| <i>line</i>   | Specifies the physical line number to be scrolled to.  |
| <i>window</i> | The window keyword applies to a subset of the windows listed in “Window Quick Reference” on page 5-2. The items marked with an asterisk (*) indicate the subset of valid window keywords for this command. |
| <i>pane</i>   | See list of pane keywords in “Command Quick Reference” on page 5-3.  |

## Restrictions

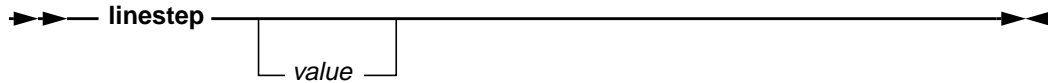
This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

# linestep

---

## Syntax



## Description

**linestep** steps the program to the next source line. If a value is specified, the action is repeated for the number of times specified in the passed value.

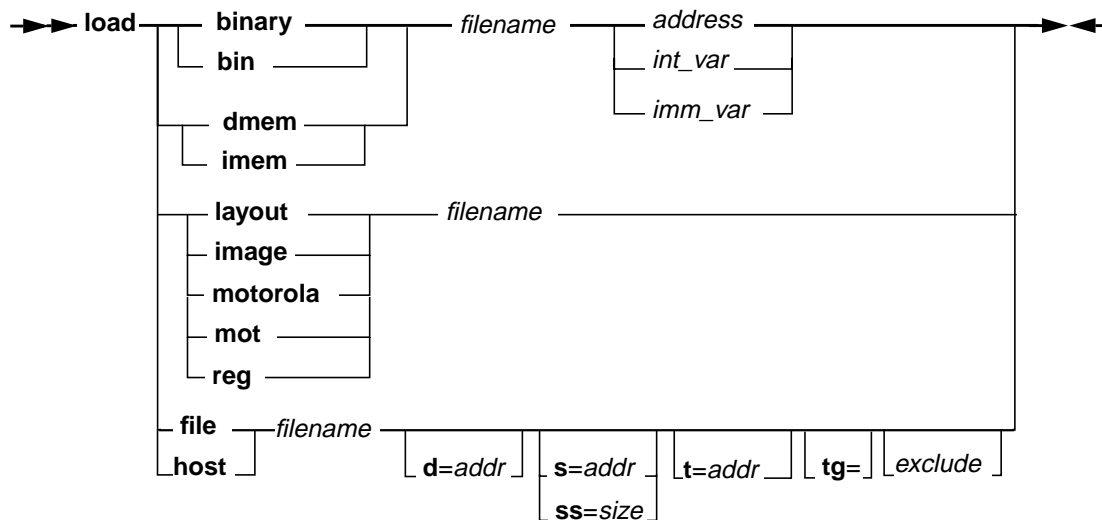
If the current source line contains a call to a function, that function and any subsequent functions will be executed until the program returns to the source line immediately following the current line, or until a breakpoint is hit.

## See Also

- **asmstep** on page 5-11
- **callstep** on page 5-25



## Syntax



## Description

**load** is used to load memory, registers or window layout information using the contents of the specified file. Each of the **load** commands expect files formatted appropriate to the type of data they contain.

## Flags

- binary** Load the contents of a binary file into data memory.
- bin** Same as the **binary** flag.
- dmem** This command is the complement to the **save mem** command and can only process those files which were generated by the **save mem** command. The file contains a block of memory values in a human-readable ASCII format. This allows the saved state of the memory to be loaded back in at any time.  
When loading the saved memory block, the data can be loaded to the same address from which it was saved, or a new address can be specified with the command allowing the data to be placed anywhere in the processor's memory.
- imem** This command is the same as the **load dmem** command except that it ensures that the contents of the instruction cache is updated along with data memory.
- layout** This command is used to load a window layout definition that was filed with a **save** command.

# load

---

<b>file</b>	Loads selective sections (text, data, etc.) of an ELF or XCOFF file into target memory and loads the host system with internal data structures used to perform source level debug.
<b>host</b>	Loads the host system with internal data structures used to perform source level debug on ELF or XCOFF file formats. The target system is not altered. This command is used to enable source level debug on user applications which have been preloaded on the target system. ROM resident code is one example of a preloaded application.
<b>image</b>	Loads the target system with the contents of a Boot Image file (images created with the <code>eimgbld/nimgbld</code> tool of the evaluation board support package). The first 32 bytes of data is assumed to be a 'header' record containing a 'load address' (bytes 4-7) and an 'entry point address' (bytes 16-19). All data following the 32 byte header is loaded on the target system, starting at the 'load address'. The instruction address register (IAR) is loaded with the value designated by the 'entry point address'. See <b>Loading Boot and Boot Image Files</b> on page 5-47 for further discussions on the use of this flag.
<b>motorola</b>	Load the contents of a Motorola format file into data memory.
<b>mot</b>	Same as the <b>motorola</b> flag.
<i>reg</i>	This command is the complement to the <b>save reg</b> command and can only process those files which were generated by the <b>save reg</b> command. The file contains all the processor register values in a human-readable ASCII format. This allows the saved state of the registers to be loaded back in at any time.
<i>address</i>	Memory address to load file contents.
<b>d=</b>	Indicates that the specified address is to be used to locate the data segment (ELF and XCOFF formats only).
<b>s=</b>	Indicates that the specified address is to be used to set the stack address (ELF and XCOFF formats only). If this value is not supplied, the <code>STACK_ADDR</code> in the environment resources file will be used. THE USE OF THIS FLAG IS NOT RECOMMENDED.
<b>t=</b>	Indicates that the specified address is to be used to locate the text segment (ELF and XCOFF formats only).
<b>ss=</b>	Indicates that the specified size is to be used to calculate the stack address. The stack address is set to 'size' bytes beyond the last byte loaded on the target (usually the last byte of <code>bss</code> data). If this value is not specified, the <code>STACK_SIZE</code> in the environment resources file will be used. If <code>STACK_SIZE</code> is undefined, the default size of 16K bytes is used.
<i>exclude</i>	Portions of the load can be excluded to improve performance under certain circumstances. The options listed below can be used individually or together :  <b>nosym</b> Indicates that symbol table and string table are NOT to be loaded on the target. This applies to boot files only (images created with the PowerPC 400Series evaluation board support

package entry code). See “Loading Boot and Boot Image Files” on page 3-47 for a discussion of boot files.

**nozero** This keyword directs RISCWatch to bypass segment initialization. Segment initialization is the term used to describe the act of zeroing out the uninitialized global variables (BSS) of an application. The ‘NOZERO’ keyword should only be used on applications which zero out their own initialization segments at program start up.

*size* **ss=** byte count for stack size.

**tg=** Specifies the thread group for OS Open systems with virtual memory support. See **start\_thread** on page 5-121 for more information.

*filename* Name of the file containing the data, in the appropriate format, to be loaded.

*int\_var* Any integer variable created with the **create** command.

*imm\_var* An assigned user-created variable specifying an immediate value that may be used as a data memory address.

**Note:** If the file name specified in the **load** command is qualified (a directory path is indicated), then the file is search in the designated directory only. If the file name is not qualified, then the directory search will be governed by the order specified via the **srchpath** command; if not found, the current directory will be checked.

## See Also

- **save** on page 5-108
- **srchpath** on page 5-118
- **start\_thread** on page 5-121

# log

---

## Syntax

→ → **log** — *message* ————— → ←

## Description

**log** writes typed message strings to the log file. The entire log message will be echoed to the log file just as if it had been typed on the command line.

Messages will only be written to the log file if the **logging** command has been set to **on** (the default).

## Flags

*message*      The message to be written to the log file.

## Examples

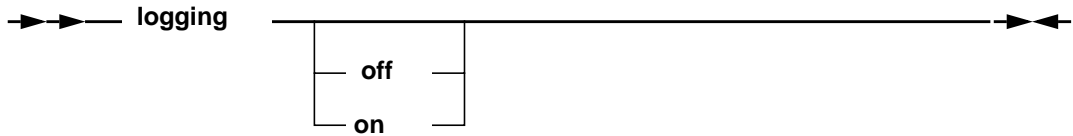
- Write the message 'R3 Test Passed' to the log file.

```
if (r3 != 0x12345678)
    log R3 Test Passed
endif
```

## See Also

- **logging** on page 5-73

## Syntax



## Description

**logging** determines the current logging status and enables or disables the writing of log messages to the log file. On initial program start up, **logging** is set to *on*. This allows all commands and program error and status messages to be written to the log file for that session.

To stop these messages from being written to the log file, use the **off** flag. No messages will be written to the log file until a **logging on** command is given. If neither the **off** nor **on** flag is specified, the command prints the current logging state.

There is also an environment variable that is used to control logging while running a command file. This variable, `CMD_FILE_LOG`, is in the environment resources file (**rwppc.env**) and can be set to YES or NO. Use of this variable in no way affects the current setting of the logging state as set by the logging command. When running command files that are very large or contain loops that will execute many times, use of this variable is suggested to disable logging during the command file run. This will prevent a very large log file from being generated in such cases.

Under normal circumstances, logging will be enabled. But should a case arise where a command file is generating log files that are too large, the `CMD_FILE_LOG` variable can be set to NO. This will keep the commands and messages generated by the command file out of the log file, allowing only commands entered from the command line and their messages to be logged.

## Flags

- |            |                                   |
|------------|-----------------------------------|
| <b>on</b>  | Logging is turned on (enabled).   |
| <b>off</b> | Logging is turned off (disabled). |

## See Also

- **log** on page 5-72

# logoff

---

## Syntax

→ → **logoff** ← ←

## Description

**logoff** allows a user to start an OS Open Boot Image using the ROM Monitor target. When issued, this command informs the ROM Monitor to leave the debug state and continue execution with any previously attached process.

The sole purpose for **logoff** is to load and execute a Boot Image file. No debug is possible once this command is executed. See “Loading Boot and Boot Image Files” on page 3-47 for further details.

## Restrictions

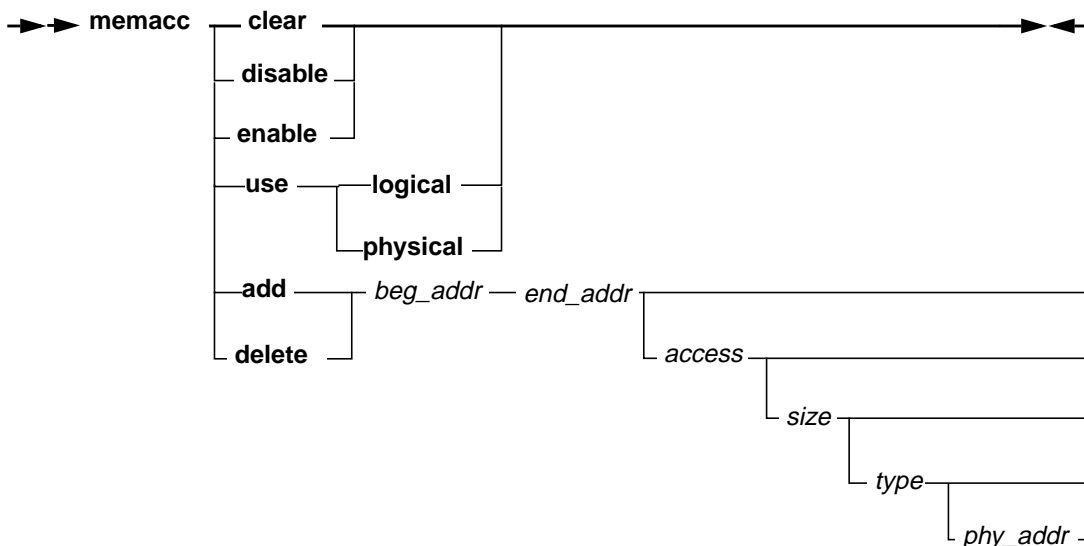
This command is only supported for OS Open and ROM Monitor targets.

## Example

- Load and execute an OS Open boot image file.

```
load image applprog.img
logoff
```

## Syntax



## Description

**memacc** enables the user to define to RISCWatch unique address access restrictions for specified regions of memory.

## Flags

- |                |  |
|----------------|--|
| <b>clear</b>   | Specifies that all user defined memory access entries are to be removed. Address validation proceeds using the default checking provided internally by RISCWatch.                                  |
| <b>disable</b> | Specifies that RISCWatch is to ignore all user defined entries for address validation. Performs the same function as the <b>clear</b> option except the entries are not removed.                   |
| <b>enable</b>  | Specifies that RISCWatch is to enable user defined access checking previously disabled by a <b>memacc disable</b> command.   |
| <b>add</b>     | Specifies a new user defined address access entry is to be added. This entry will override any previously defined entry or internal default RISCWatch checking within the specified address range. |
| <b>delete</b>  | Specifies that an access check entry is to be deleted. If the <b>memacc add</b> command was used to define two identical access entries, the last one added will be removed.                       |
| <b>use</b>     | Specifies a new RISCWatch Memory Access Mode.  |

# memacc

---

<i>beg_addr</i>	Specifies the beginning address for a region of memory. It can be in the form of an integer, <i>imm_var</i> , <i>int_var</i> , or <i>mem_var</i> .
<i>end_addr</i>	Specifies the end address for a region of memory. It can be in the form of an integer, <i>imm_var</i> , <i>int_var</i> , or <i>mem_var</i> .
<i>access</i>	Specifies the allowable accesses for a region of memory. Valid types are the keywords <b>NA</b> (no access), <b>RO</b> (read only), <b>WO</b> (write only), or <b>RW</b> (read/write). Alternatively, a corresponding integer value can be specified explicitly, or in the form of an <i>imm_var</i> or <i>int_var</i> . Valid integers are 0 ( <b>NA</b> ), 1 ( <b>RO</b> ), 2 ( <b>WO</b> ), or 3 ( <b>RW</b> ). The default if no <i>access</i> is specified is <b>RW</b> .
<i>size</i>	Specifies the byte size of memory accesses within the region of memory. It can be in the form of an integer, <i>imm_var</i> , <i>int_var</i> , or <i>mem_var</i> . Valid values are 0, 1, 2, 4, or 8 bytes. If no <i>size</i> is specified, or a <i>size</i> of zero is indicated, it will be set to the default size as determined by the target processor (4 bytes for 4xx processors, 8 bytes for all other processors).
<i>type</i>	Specifies the type of access to be checked for a region of memory. Valid types are the keywords <b>IMEM</b> (instruction only), <b>DMEM</b> (data only), or <b>MEM</b> (instruction and data). Alternatively, a corresponding integer value can be specified explicitly, or in the form of an <i>imm_var</i> or <i>int_var</i> . Valid integers are 1 <b>DMEM</b> , 2 <b>IMEM</b> or 3 <b>MEM</b> . The default if no <i>type</i> is specified is <b>MEM</b> . Since users are not aware of how RISCWatch internally accesses memory, the default value of <b>MEM</b> should be used.
<i>phy_addr</i>	Specifies a physical address associated with <i>beg_addr</i> . It can be in the form of an integer, <i>imm_var</i> , <i>int_var</i> , or <i>mem_var</i> . If not specified, <i>phy_addr</i> will be set to <i>beg_addr</i> .
<b>logical</b>	Specifies RISCWatch Logical Memory Access Mode. Addresses presented to RISCWatch will also map directly to some physical address (no address redirection will take place). The <i>phy_addr</i> field will be ignored. This is the default mode of operation for all RISCWatch memory reads or writes.
<b>physical</b>	Specifies RISCWatch Physical Memory Access Mode. Addresses presented to RISCWatch will map to the physical addresses designated by the <i>phy_addr</i> field. This provides a level of user defined address translation. This address redirection will occur regardless of address translation state of the target processor.

## Examples

- Disable all internal memory access checking done by RISCWatch by creating a user defined entry which defines the entire address space.

```
memacc add 0x00000000 0xFFFFFFFF RW
```

- Make the region of memory from 0 to 0xFFFF read only with an access size of 4.

```
memacc add 0 0xFFFF RO 4 MEM
```



- Make memory address 0x4000 write only with an access size of 1 byte.

```
create serial_addr = 0x4000
assign serial_size = 1
memacc add serial_addr serial_addr WO serial_size
```

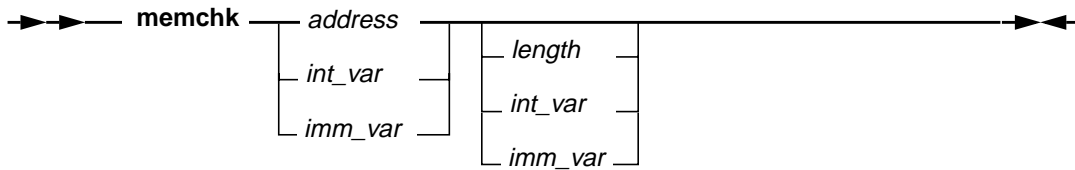
### See Also

- “Core + ASIC Resources” on page 3-8
- “Reading and Writing Memory” on page 3-103

# memchk

---

## Syntax



## Description

**memchk** tests the integrity of the processor's memory. The values 0x00, 0xA5, 0xFF and 0x5A are written to the specified address one at a time and then read back to verify that they were indeed written correctly. An error message is displayed for any read, write or compare failure detected.

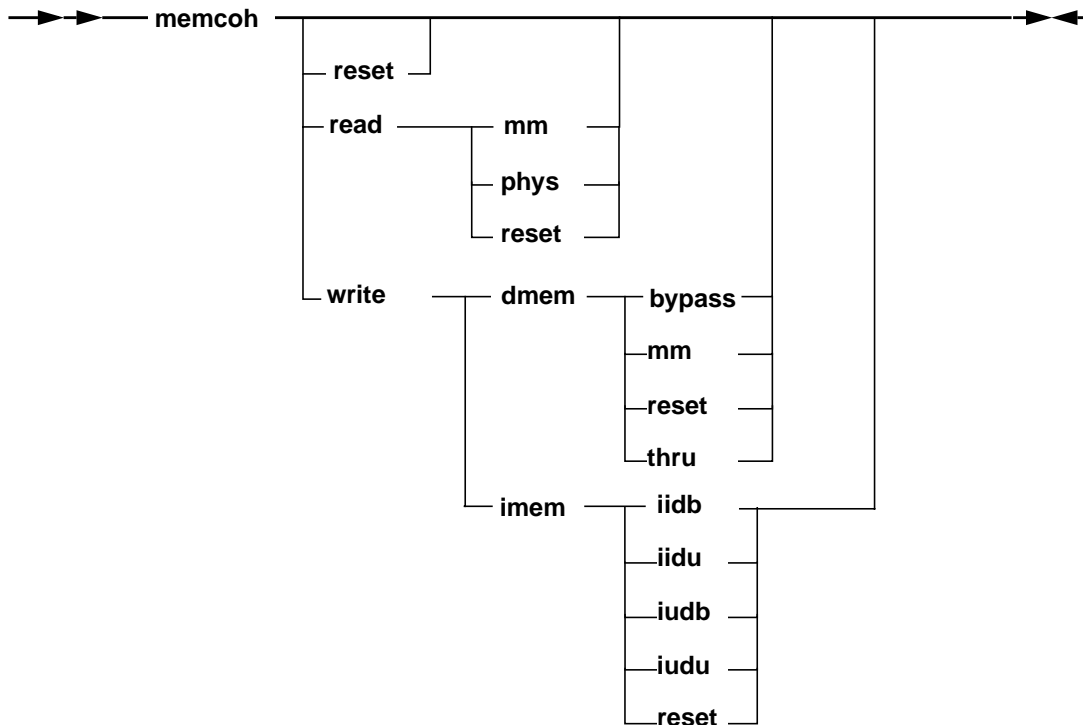
## Flags

<i>address</i>	Specifies the memory address to be checked.
<i>length</i>	Specifies the number of sequential addresses to check. The default value is 1.
<i>int_var</i>	A user-created integer variable that may be used as the memory address to be written or length to be checked.
<i>imm_var</i>	An assigned user-created variable specifying an immediate value that may be used as a data memory address or length to be checked.

## See Also

- **memcpy** on page 5-81
- **memfill** on page 5-82

## Syntax



## Description

**memcoh** is used to control data and instruction cache updating during reads and writes. The command performs the same actions as the selections on the Memory Coherency Window. See "Memory Coherency Window (JTAG Targets Only)" on page 3-104 for more information about the coherency model terms used here.

## Flags

<b>reset</b>	Reset the coherency model or one of its attributes to the default value.
<b>read</b>	Set the read memory attribute of the coherency model.
<b>write</b>	Set the write memory attribute of the coherency model.
<b>imem</b>	Specifies the instruction memory as the write attribute being set.
<b>dmem</b>	Specifies the data memory as the write attribute being set.
<b>bypass</b>	Specifies the cache is to be bypassed on data memory writes.

# memcoh

---

<b>iidb</b>	Specifies icache invalidate, dcache bypass on instruction memory writes.
<b>iidu</b>	Specifies icache invalidate, dcache update on instruction memory writes.
<b>iudb</b>	Specifies icache update, dcache bypass on instruction memory writes.
<b>iudu</b>	Specifies icache update, dcache update on instruction memory writes.
<b>mm</b>	Specifies the memory model is to be used for memory reads and data memory writes.
<b>phys</b>	Specifies the physical model is to be used for memory reads.
<b>thru</b>	Specifies the dcache is treated as write thru for data memory writes.

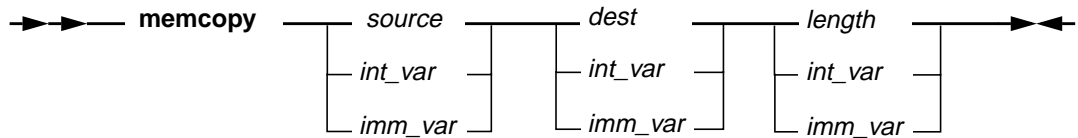
## Restrictions

This command is only supported on JTAG targets.

## See Also

- “Memory Coherency Window (JTAG Targets Only)” on page 3 - 104

## Syntax



## Description

**memcpy** copies a block of memory from one address to another. The memory block is copied from the source address to the destination address. The number of bytes to copy is specified.

## Flags

<i>source</i>	Specifies the source memory address
<i>dest</i>	Specifies the destination memory address
<i>length</i>	Specifies the number of bytes to copy
<i>int_var</i>	A user-created integer variable that may be used as the memory address to be written
<i>imm_var</i>	An assigned user-created variable specifying an immediate value that may be used as a data memory address

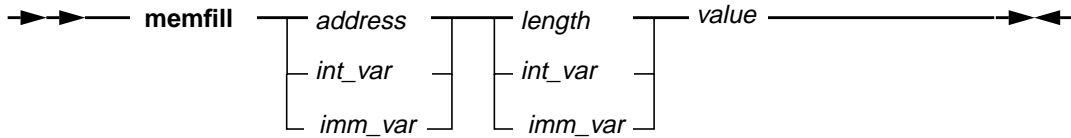
## See Also

- **memchk** on page 5-78
- **memfill** on page 5-82

# memfill

---

## Syntax



## Description

`memfill` fills a region of the processor's memory. The value specified is written starting at the specified address for the specified number of bytes.

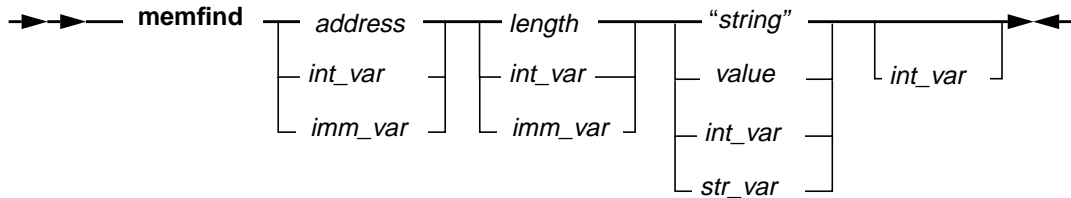
## Flags

<i>address</i>	Specifies the memory address to start the fill at
<i>length</i>	Specifies the number of bytes to fill
<i>value</i>	Specifies the value to be written
<i>int_var</i>	A user-created variable that may be used as a memory address or a value to be written
<i>imm_var</i>	An assigned user-created variable specifying an immediate value that may be used as a data memory address or a value to be written

## See Also

- `memchk` on page 5-78
- `memcpy` on page 5-81

## Syntax



## Description

**memfind** locates the address of the first occurrence of a specified string in memory. A message is printed indicating successful completion of the command. The fourth parameter is used to hold the address where the string or value was found. If the string is not found, the fourth parameter will be set to the first address outside the specified range (address + length).

## Flags

<i>address</i>	Specifies the memory address to start searching.
<i>length</i>	Specifies the number of bytes to search.
<i>"string"</i>	Specifies a string of ASCII characters to be searched.
<i>"str_var"</i>	A user-created variable that holds a string of ASCII characters to be searched.
<i>value</i>	Specifies a string of hexadecimal characters to be searched.
<i>int_var</i>	A user-created variable that may be used as a memory address, length, or a value to be searched for.
<i>imm_var</i>	An assigned user-created variable specifying an immediate value that may be used as a data memory address or a value to be written.

## Examples

- Search for the string "TEST" starting at address 0xFFC0 for the next 0x200 bytes.
 

```
memfind 0xFFC0 0x200 "TEST"
```
- Search for the same string in the previous example but by specifying hex characters.
 

```
memfind 0xFFC0 0x200 54455354
```
- Search for second occurrence of string 'HELLO' within the address range 0 to 0x200.
 

```
create find_addr = 0
create loop_count = 0
create find_value = "HELLO"
create length = 0x200
```

# memfind

---

```
while (find_addr < 0x200 && loop_count != 2)
    set loop_count = loop_count + 1
    if (find_addr != 0)
        set length = 0x200 - find_addr - 1
        set find_addr = find_addr + 1
    endif
    memfind find_addr length find_val find_addr
endwhile
if (find_addr < 0x200)
    log second occurrence found
elseif
    log second occurrence not found
endif
```

## See Also

- **memchk** on page 5-78
- **memcpy** on page 5-81



## Syntax



## Description

**memrwait** displays or sets the delay used in memory read operations. This command is typically used to slow reads down when reading from a memory mapped I/O device.

## Flags

*value* Specifies the delay time to set in microseconds. The initial delay is zero.

## Restrictions

This command is only supported on 6xx and 7xx processors.

For 604 processors, the valid delay range is 0 to 10,000,000  $\mu$ s (10 seconds).

For 603x/7xx processors, the valid delay range is 0 to 6,000  $\mu$ s (6 milliseconds)

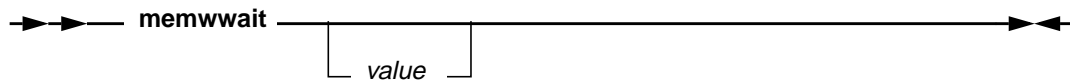
## See Also

- **memwwait** on page 5-86

# memwwait

---

## Syntax



## Description

**memwwait** displays or sets the delay used in memory write operations. This command is typically used to slow writes down when writing to a memory mapped I/O device.

## Flags

*value* Specifies the delay time to set in microseconds. The initial delay is zero.

## Restrictions

This command is only supported on 6xx and 7xx processors.

For 604 processors, the valid delay range is 0 to 10,000,000  $\mu$ s (10 seconds).

For 603x/7xx processors, the valid delay range is 0 to 6,000  $\mu$ s (6 milliseconds)

## See Also

- **memrwait** on page 5-85

## Syntax

→→ **mpsset** — *mps\_id* ←←

## Description

**mpsset** changes the current debugger context to target the specified chip. For more information about multiprocessor support, see “Multi-Processor Resources” on page 3-28

## Flags

*mps\_id* Specifies which chip RISCWatch will target as the current context. The string must match one of the chip names initially defined in the MPS file.

## Restrictions

This command is only supported on JTAG targets.

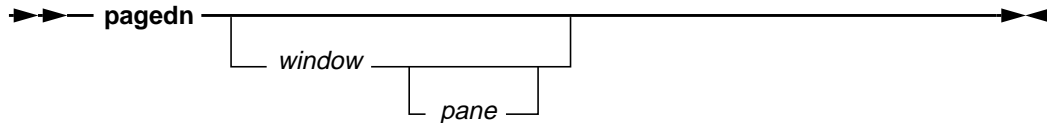
## See Also

- “Multi-Processor Resources” on page 3-28

# pagedn

---

## Syntax



## Description

**pagedn** scrolls the contents of a window down one page.

If the *window* keyword is not specified, the last window specified for this command is used. It initially defaults to the Source window.

## Flags

- |               |  |
|---------------|--|
| <i>window</i> | The window keyword applies to a subset of the windows listed in “Window Quick Reference” on page 5-2. The items marked with an asterisk (*) indicate the subset of valid window keywords for this command. |
| <i>pane</i>   | See list of pane keywords in “Command Quick Reference” on page 5-3.  |

## Restrictions

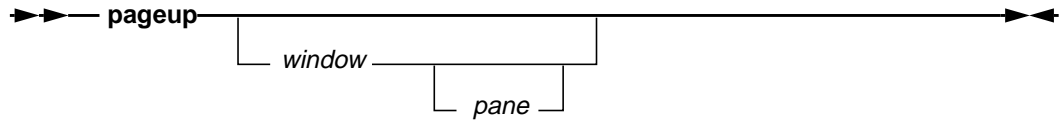
This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

## See Also

- **pageup** on page 5-89

## Syntax



## Description

**pageup** scrolls the contents of a window up one page.

If the *window* keyword is not specified, the last window specified for this command is used. It initially defaults to the Source window.

## Flags

<i>window</i>	The window keyword applies to a subset of the windows listed in “Window Quick Reference” on page 5-2. The items marked with an asterisk (*) indicate the subset of valid window keywords for this command.
<i>pane</i>	See list of pane keywords in “Command Quick Reference” on page 5-3.

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

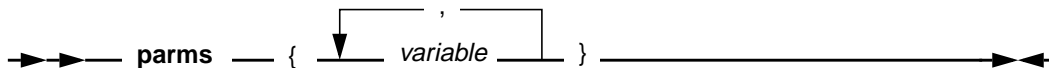
## See Also

- **pagedn** on page 5-88

# parms

---

## Syntax



## Description

`parms` allows one or more parameters to be passed into a command file when it is executed.

## Flags

*variable*      The names of variables to be created. At least one variable name must be specified. The variables are initialized to the values specified in the parameter list. If there are more variables specified in the `parms` list than there are values in the parameter list, the left-over variables are initialized to 0. If there are more values in the parameter list than there are variables in the `parms` list, the extra values are discarded.

## Restrictions

This command is not supported in TTY mode.

## Examples

- Within a command file, use the **parms** command to pass a memory address value:

```
parms {mem_addr}  
read dmem mem_addr
```

The variable *mem\_addr* can now be used like any other user-created variable inside the command file. When RISCWatch is invoked to run this command file, it is now possible to pass the desired memory address into the command file for execution:

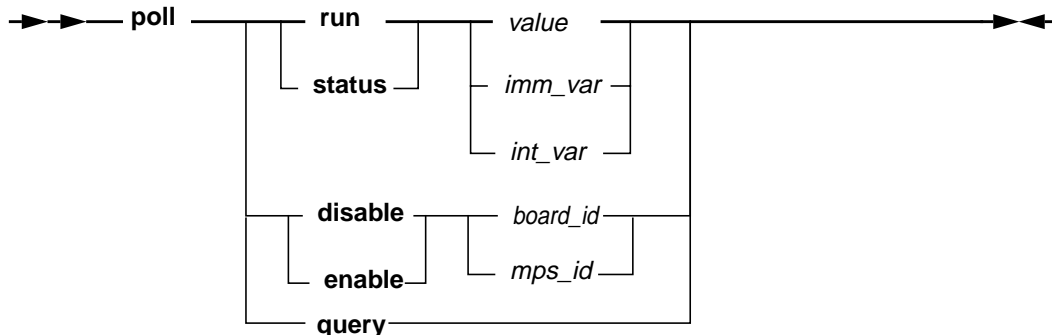
```
rwppc mem_test{0xFFFF0000}
```

**Note:** Be sure that there is NO space between the command file name and the opening '{' character. Also make sure that there IS a space between the **parms** command and the opening '{' character.

## See Also

- “Command File Parameters” on page 3-128

## Syntax



## Description

**poll** enables the user to control the various polling requests RISCWatch uses during debug operations.

## Flags

<b>run</b>	Alter the frequency at which RISCWatch polls the target for a stop when running.
<b>status</b>	Alter the frequency at which RISCWatch polls the target for a change in status while stopped.
<i>value</i>	Polling interval requested, in milliseconds (200 minimum; 0 = disable).
<i>imm_var</i>	An assigned user-created variable specifying the requested polling interval, in milliseconds (200 minimum; 0 = disable).
<i>int_var</i>	A user-created variable specifying the requested polling interval, in milliseconds (200 minimum; 0 = disable).
<b>disable</b>	Disable board/MPS context polling.
<b>enable</b>	Enable board/MPS context polling.
<i>board_id</i>	Valid MPS board name from MPS file.
<i>mps_id</i>	Valid MPS chip name from MPS file.
<b>query</b>	Show the current poll settings.

## Examples

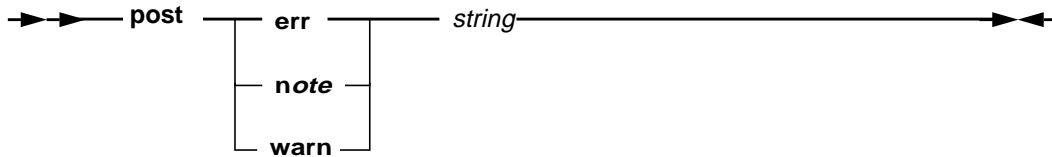
- Shut the polling off on a board in an MPS debugging session:

```
poll disable Board1
```

# post

---

## Syntax



## Description

**post** enables the user to open dialog boxes on the interface that contain specified information and format. This feature can be useful when used in command files to provide pass/fail information at the end of a test, or for providing progress indication that the user must acknowledge.

## Flags

<b>err</b>	Information specified is to be posted with an error indicator
<b>note</b>	Information specified is to be posted with an note indicator
<b>warn</b>	Information specified is to be posted with an warning indicator
<i>string</i>	string of text to display in dialog box

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

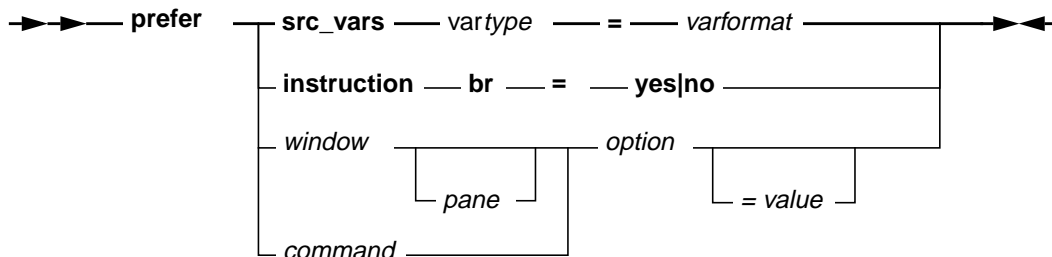
## Examples

- Here is a command file excerpt that uses the **post note** command to indicate that the command file execution has proceeded to the point that it now requires the user action and acknowledgment before continuing execution:

```
# Board 1 init is done at this point of the file...
POST NOTE Board 1 set up complete. Power on board 2.
# Code below here will not execute until user confirms
# note in dialog box
```



## Syntax



## Description

**prefer** allows the user to customize various program settings to their liking. Some settings apply to windows, some to commands and others were created to allow for control over program wide features.

**Prefer** commands are typically contained in the startup command file so that the designated preferences apply to the entire debug session. Prefer commands executed during program execution can be used to override previously set options.

In particular, the **src\_vars** setting enables the user to designate the default display preference for source variables. The **instruction** setting enables limited support of little-endian memory address spaces. The **window** setting allows various attributes of specified windows to be altered.

## Flags

**src\_vars** This specifier is used to change the default display preference for source variables of different types.

**vartype** Indicates a fundamental variable type of the 'C' programming language. Valid types are:

<b>addr</b>	integer and signed integer
<b>uint</b>	unsigned interger
<b>short</b>	short and signed short
<b>ushort</b>	unsigned short
<b>long</b>	long and signed long
<b>ulong</b>	unsigned long
<b>char</b>	character and signed character
<b>uchar</b>	unsigned character
<b>longlong</b>	longlong and signed longlong
<b>ulonglong</b>	unsigned longlong
<b>float</b>	float point

	<b>cfloat</b>	complex float
	<b>double</b>	double float
	<b>ldouble</b>	long double
	<b>cdouble</b>	complex double
<i>varformat</i>	Indicates the default variable display format to be used for the fundamental type specified. Valid formats are:	
	<b>bin</b>	binary
	<b>default</b>	restore to default RISCWatch format
	<b>hex</b>	hexadecimal
	<b>oct</b>	octal
	<b>signed</b>	signed decimal
	<b>unsigned</b>	unsigned decimal
<b>instruction</b>	This specifier is used to change instruction handling.	
<b>br</b>	This specifier enables or disables byte-reversed instruction handling thereby providing limited support for little-endian memory. Enabling this option will effect the assembler, disassembler, breakpoints and I/O of the Assembly Debug window.	
<i>window</i>	The window keyword applies to a subset of the windows listed in “Window Quick Reference” on page 5-2. Here is a list of supported windows and their options:	
	<b>ascii</b>	See the <b>addr</b> and <b>lines</b> options.
	<b>asic</b>	See the <b>prefix</b> option
	<b>custom</b>	See the <b>addr</b> , <b>base</b> , <b>lines</b> , <b>sign</b> and <b>size</b> options.
	<b>debug</b>	See the <b>addr</b> and <b>lines</b> options.
	<b>main</b>	See the <b>fixed</b> and <b>ratio</b> options.
	<b>regfld</b>	See the <b>res</b> option.
<i>pane</i>	Some windows contain multiple panes of data. If a specific pane is specified, only its <i>option</i> setting will be modified. Here is a list of the supported panes and their options:	
	<b>mainmsg</b>	The message pane of the Main window. See the <b>fixed</b> option.
	<b>maincmd</b>	The command history pane of the Main window. See the <b>fixed</b> option.
<i>command</i>	A valid RISCWatch command . Here is a list of supported commands and their options:	
	<b>dis</b>	See the <b>mnemonics</b> option.
	<b>exec</b>	See the <b>error</b> option.
	<b>trace</b>	See the <b>output</b> option.
<i>option</i>	The name of the option being set. Here is a list of supported options:	
	<b>addr</b>	This is the default memory address for displaying data on any of the memory windows.

<b>base</b>	Used by the Custom Memory window to indicate whether data is to be displayed in <b>ascii</b> , <b>binary</b> , <b>decimal</b> or <b>hexadecimal</b> .
<b>error</b>	Used by the exec command. Set this option to <b>stop</b> to stop execution of a command file when an error is encountered.
<b>fixed</b>	Used by the two panes of the main window to force the specified pane to remain fixed in size when the window is resized.
<b>lines</b>	The number of lines used for display text when a window is first created. This option is used by the memory and source windows.
<b>output</b>	Used by the trace command to select a <b>asm</b> , <b>src</b> or <b>mixed</b> listing.
<b>prefix</b>	Used by the ASIC Register windows to indicate whether or not the register macro prefixes are to be displayed as part of the register name or not by selecting <b>yes</b> or <b>no</b> .
<b>ratio</b>	Used by the Main window to select the ratio of lines for the command and message history panes.
<b>res</b>	Used by the Register Field windows to indicate whether or not Reserved fields should be displayed by selecting either <b>yes</b> or <b>no</b> .
<b>sign</b>	Used by the Custom Memory window to indicate whether memory data is to be displayed in <b>signed</b> or <b>unsigned</b> format.
<b>size</b>	Used by the Custom Memory window to indicate whether data is to be read and displayed as <b>1</b> , <b>2</b> or <b>4</b> byte "words."
<i>value</i>	The specific value to set a window or command <i>option</i> to. This can be left blank to erase the current option value thereby setting it to its default value.
<b>yes no</b>	Select either <b>yes</b> or <b>no</b> for this option's value.

## Examples

- Change the default source variable display of all unsigned integers to hexadecimal.

```
prefer srv_vars uint = hex
```
- The following commands can be used to select whether the **dis** command displays extended or non-extended opcodes:

```
prefer dis mnemonics = ext
prefer dis mnemonics = non-ext
```
- The following command is used to indicate that execution of a command file should stop if an error is encountered:

```
prefer exec error = stop
```

# prefer

---

- The following commands can be used to indicate if the register prefix is to appear for each register listed in an ASIC window:

```
prefer window asic prefix = yes
prefer window asic prefix = no
```

- The following commands can be used to indicate that the Custom Memory window is to start display data at address 0xA0C4, read/display data as 2 byte words while displaying them in unsigned decimal notation:

```
prefer custom addr = 0xa0c4
prefer custom size = 2
prefer custom base = decimal
prefer custom sign = unsigned
```

- The following command is used to force the Assembly Debug window to always display 20 lines of text:

```
prefer debug lines = 20
```

- The following command is used to fix the size of the main window command pane regardless of how the window is resized:

```
prefer main pane maincmd fixed
```

**Note:** If the window were resized, the command pane would remain the same size while the message pane would grow or shrink accordingly.

- The following does the same for the message pane:

```
prefer main pane mainmsg fixed
```

**Note:** If the window were resized, the message pane would remain the same size while the command pane would grow or shrink accordingly.

- The following command is used to fix the ratio of the main window command pane lines to its message pane lines:

```
prefer main ratio = 3
```

**Note:** If the window were resized, the two panes would grow or shrink accordingly but always retain the same ratio of lines to each other.

See the sample startup command file (startup.cmd) provided with RISCWatch for additional examples.

## Syntax

→ → **print** *print\_string* ← ←

## Description

**print** takes *print\_string* and prints it in the host window. See the **fprint** command for more details and a list of formatting options.

## Flags

*print\_string* This is a user definable string containing string literals, user-created variable names and the same type of expressions used in the **set** command.

## Restrictions

This command is only supported in command file mode.

## Examples

- Write the print message 'R3 Test completed'.

```
if (r3 != 0x12345678)
    print "R3 Test completed"
endif
```

See also the Examples section of the **fprint** command

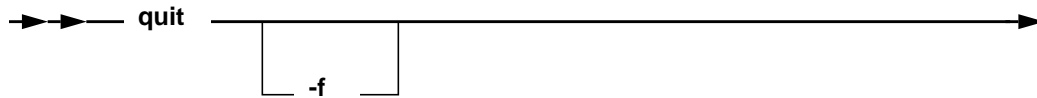
## See Also

- **fctrl** on page 5-43
- **fprint** on page 5-55

# quit

---

## Syntax



## Description

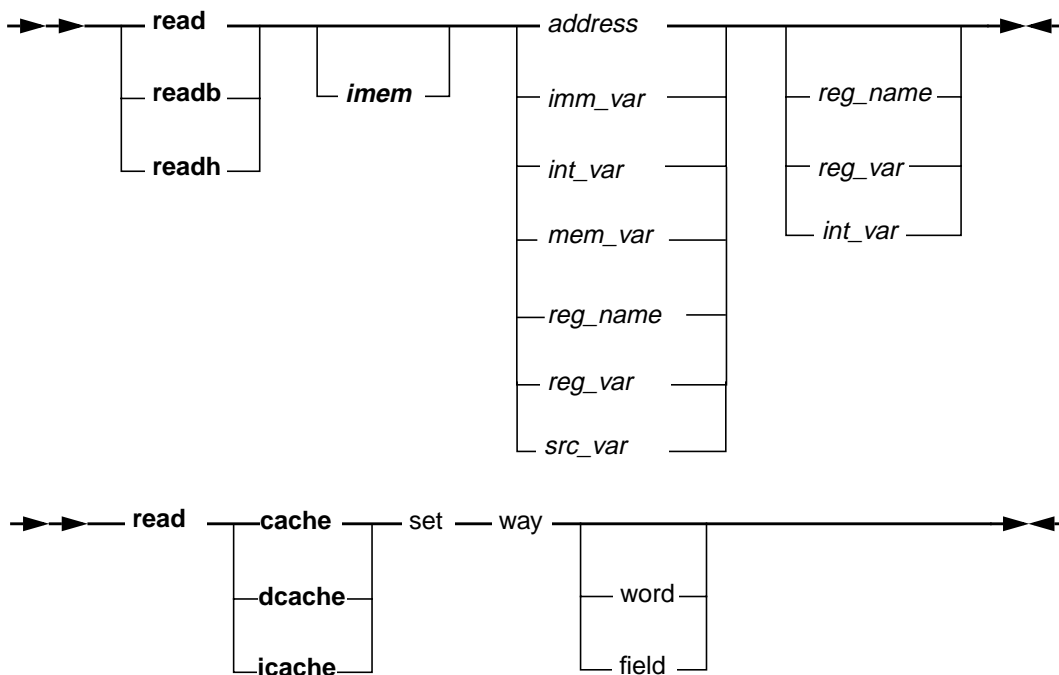
**quit** terminates the program. If the processor is running when this command is given and the user interface is active, a prompt is displayed to provide notification of the processor state and confirm the intent to terminate.

Avoid using the **quit** command in a command file. If the command file is executed while the user interface is active, execution of the **quit** command will not only stop the command file but will also terminate RISCWatch. Use the **end** command within a command file to stop execution of the command file.

## Flags

**-f** Using this flag forces termination regardless of the processor state.

## Syntax



## Description

**read** is used to read register values, four bytes of data memory, cache data, or currently scoped global and local source variables. The **readb** command is used to read one byte of data memory while the **readh** command is used to read two bytes of data memory.

**readb** is used to read a 1-byte data memory location, while **readh** is used to read a 2-byte data memory location.

**Note:** RISCWatch by default will read and display memory exactly as it is in the target's memory and not how it is used by the processor. For limited support of little-endian targets, refer to the byte-reversed feature of the memory windows as well as the **prefer instruction br** command.

The first argument in a register, memory or source variable read is used to indicate the object to be read. If a second argument is specified, it indicates the object to be written using the value just read.

The first argument in a cache read is used to denote the cache to be read (unified, data or instruction). The second argument indicates the set to be read while the third argument indicates the way. If no further arguments are specified, this command reads the address tag

# read

---

for the specified set and way. If a fourth argument is specified, this can indicate the number of the word or a string denoting the data field to be read.

## Flags

<b>imem</b>	Specifies that instruction (not data) memory is to be read
<i>address</i>	Specifies an immediate address value from which to read data memory
<i>field</i>	A string of one or more characters indicating the cache data field to be read. The list of acceptable field strings can be found by referring to the Cache Windows section of this guide.
<i>int_var</i>	A created user-created variable that may be used to hold the value just read
<i>mem_var</i>	Any memory variable created with the assign command
<i>imm_var</i>	An assigned user-created variable specifying an immediate value that may be used as a data memory address
<i>reg_name</i>	A valid processor register name to be read and/or written
<i>reg_var</i>	An assigned user-created variable that may be used to specify a processor register to be read and/or written
<b>cache</b>	Specifies that the unified cache is to be read
<b>dcache</b>	Specifies that the data cache is to be read
<b>icache</b>	Specifies that the instruction cache is to be read
<i>set</i>	The numeric value specifying the cache set whose tag, field or word value is to be read.
<i>src_var</i>	Any valid local or global source variable name that is currently in scope. The name must be preceded by a colon ":". See "Source Variable Command Support" on page 3-102 for further information.
<i>way</i>	A numeric value specifying the cache way whose tag, field or word value is to be read. The first way in any line is 0. For caches which use letters to denote the ways, simply convert A to 0, B to 1, C to 2, etc.
<i>word</i>	A numeric value indicating the word of a line to be read. The first word of a line is 1.

## Examples

- Read the value of the IAR.  
`read IAR`
- Read the value at memory address 0x1FB470.  
`read 0x1FB470`
- Create a user variable to represent a memory location and then use it to read memory.



```
assign mem_addr = 0x000F701A
read mem_addr
```

- Read the contents of source 'array[7]' and store it in GPR R1.  

```
read :array[7] R1
```
- Read the address tag for set 4, way B of the Instruction Cache.  

```
read icache 4 1
```
- The address tag for set 4, way B of the Instruction Cache could also be read using  

```
read icache 4 1 TAG
```
- Read the second word for set 7, way 3 of the Data Cache.  

```
read dcache 7 3 2
```
- Read the value of the Least Recently Used field of set 2, way 0 of the Cache.  

```
read cache 2 0 L
```

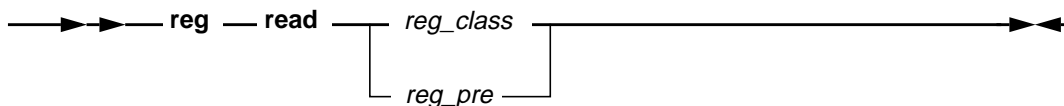
## See Also

- [write on page 5-140](#)

# reg

---

## Syntax



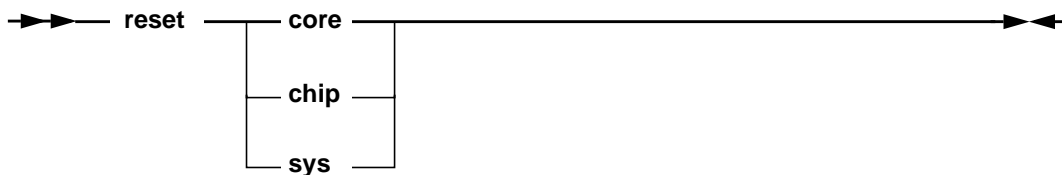
## Description

**reg** is used to force a read of a specified class of registers or a subset of ASIC registers which share a common prefix. This is the equivalent functionality of pressing the Read button of the appropriate register window.

## Flags

- |                  |  |
|------------------|--|
| <i>reg_class</i> | Specifies the class of registers to be read. See the description for this flag in “Command Quick Reference” on page 5-3.       |
| <i>reg_pre</i>   | Specifies the subset of ASIC registers to be read. See the description for this flag in “Command Quick Reference” on page 5-3. |

## Syntax



## Description

**reset** resets the processor or system. For further details about the processor reset, see the chapter concerning Reset and Initialization in the PowerPC 400Series User's Manual for the specific controller being reset.

**Note:** For 6xx/7xx processors, **core** and **chip** reset are equivalent. The processor will be reset and stopped at instruction address 0xFFF00100. Execution of **sys** reset on 6xx/7xx processors will reset and run the processor.

## Flags

<b>core</b>	Reset the processor core.
<b>chip</b>	Reset the processor core and ASIC.
<b>sys</b>	Reset the entire system.

## Restrictions

This command is only supported on JTAG targets.

# restart

---

## Syntax

▶▶—restart————▶▶

## Description

**restart** restarts the debug session.

The debug session is restarted essentially by reloading the program onto the target. However, the debug environment remains intact. This means that any breakpoints that were set will still be set, and all currently selected windows and customizations will be preserved and their context updated as appropriate.

**Note:** If the program was dynamically loaded, the breakpoint addresses will be recalculated based on the new location of the reloaded program.

**OS Open Note:** If the program being debugged was started via a **start\_thread** or an **attach** command, then the program will not be reloaded. The thread will be restarted or reattached only. This means that the data area and bss sections will not get reinitialized.

## See Also

- **attach** on page 5-16
- **start\_thread** on page 5-121

## Syntax

▶▶—retstep—◀◀

## Description

**retstep** returns the debugger to the previous function caller.

This location to which the IAR is returned is effectively the contents of the current link register.

**Note:** When stepping through code that contains no debug information, the link register contents could be altered by subsequent branch and link instructions. In these instances, **retstep** does not produce the desired results. Instead, a breakpoint should be set at the desired return location, and a **run** command executed to carry out the intended action.

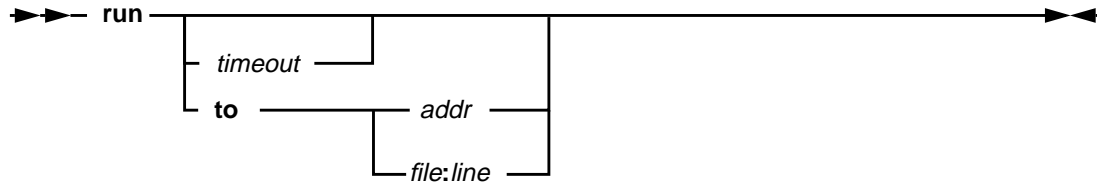
## See Also

- **asmstep** on page 5-11
- **bp** on page 5-19
- **callstep** on page 5-25
- **run** on page 5-106

# run

---

## Syntax



## Description

**run** starts the processor (JTAG target) or process (non-JTAG) running. If the *timeout* parameter is omitted, the processor/process runs until a breakpoint is reached or a **stop** command is issued.

## Flags

- |                  |  |
|------------------|--|
| <b>to</b>        | Run to a specified address or line number within a file. The program will run until the specified location or a breakpoint is reached.   |
| <i>addr</i>      | Address to run to.   |
| <i>file:line</i> | Filename and line number to run to. The program containing the specified file must already have been loaded.   |
| <i>timeout</i>   | The time, in seconds, that the processor/process is allowed to run. If the processor/process is still running after the specified time, the processor/process is stopped. This timeout value may also be specified using a created variable or an assigned immediate variable. |

If a **run** command is issued with a timeout value and then a **stop** command is issued with a timeout value, when either command has timed out the processor/process is stopped.

When a **run** command is executed from within a command file, execution of the command file does not proceed until the processor/process has stopped.

## Examples

- Run the processor/process for a maximum of 10 seconds

```
run 10
```
- Run until the program reaches address 0xFFFF0700.

```
run to 0xFFFF0700
```
- Run until the program reaches line 24 of demo1.c.

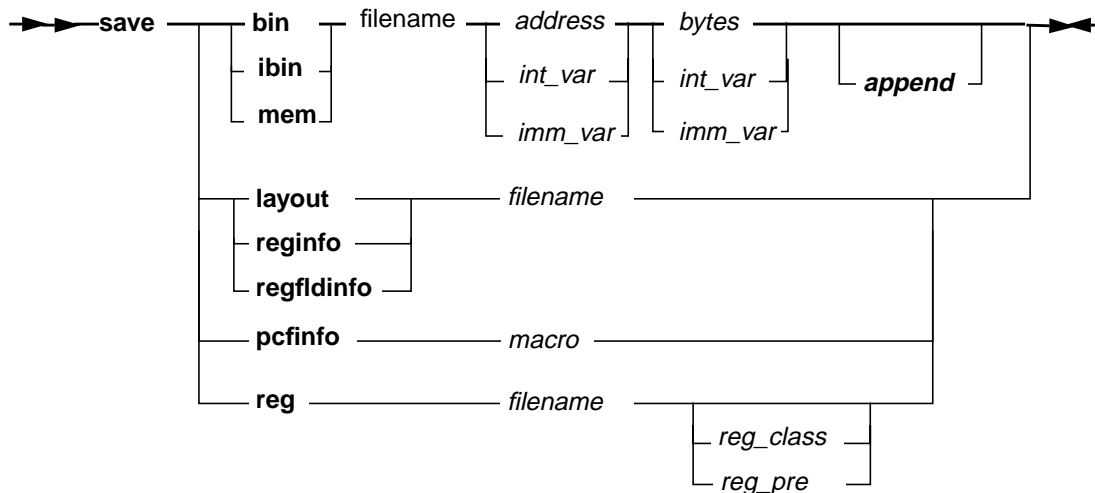
```
run to demo1.c:24
```

**See Also**

- **stop** on page 5-122

# save

## Syntax



## Description

**save** is used to save memory, register values, register address information, register field information, or the current window layout to a file. This command complements the **load** command and generates the files used by the **load** command.

With the exception of the “bin” option, the files generated by **save** are human readable ASCII files that can be used to capture the state of processor facilities. Since these files are human readable, they make excellent reference material when debugging a problem or for providing hard-copy output.

Once saved, the values in these files may be loaded back into the processor, thereby restoring the processor’s state at a later time.

## Flags

- bin** Specifies that a portion of processor memory is to be saved in binary format.
- ibin** Specifies that a portion of instruction memory is to be saved in binary format
- mem** Specifies that a portion of processor memory is to be saved in an ASCII readable format.
- layout** Specifies that the window layout is to be saved.
- reg** Specifies that processor register values are to be saved. If no register class or ASIC prefix are specified then all registers will be saved.
- reg\_class* Specifies which class of registers are to be saved.



<i>reg_pre</i>	Specifies which subset of ASIC registers are to be saved.
<b>regfldinfo</b>	Specifies that all register field information is to be saved.. Registers containing more than one set of field definitions will use an instance number to differentiate each occurrence.
<b>reginfo</b>	Specifies that all register address information is to be saved. All register names known to RISCWatch will be saved along with their respective addresses.
<i>filename</i>	The name of the file to save data.
<i>address</i>	The address of memory where to start saving data. This may also be specified using a created variable or an assigned immediate variable.
<i>bytes</i>	The number of memory bytes to save. This may also be specified using a created variable or an assigned immediate variable.
<i>int_var</i>	A <b>create</b> variable whose value may be used in place of the address or bytes flags.
<i>imm_var</i>	An <b>assign</b> variable specifying an immediate value that may be used in place of the address or bytes flag.
<b>pcfinfo</b>	Specifies that PCF macro info is to be saved. This information can then be used to help generate your own PCF/PRD file.
<i>macro</i>	The name of the processor macro which is to be saved. This information will be stored in a file titled <i>macro.txt</i> .
<b>append</b>	Append the saved memory to the designated file.

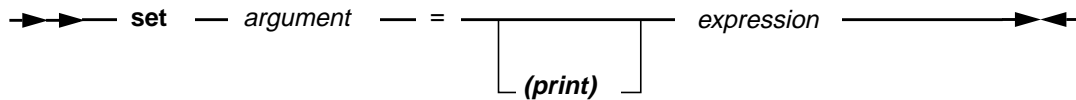
## See Also

- **load** on page 5-69
- **create** on page 5-32
- **assign** on page 5-12

# set

---

## Syntax



## Description

**set** is used to set a processor resource (memory or register) or RISCWatch variable's value to the value represented by the specified expression.

Source variable names (program local or global variables) are preceded by a colon (`:`) to distinguish them from RISCWatch variable names. A variable is expanded to the corresponding expression within other commands.

The **set** command is used to store computed values in memory address locations, registers or user-created variables. The first argument specifies where the result of the expression is to be stored (memory, register or variable).

Following the first argument (or optional = sign), is the expression to be evaluated. This expression may be composed of registers, registers fields (logically related sequences of bits within a register), memory addresses, immediate values, user-created variables, program source variables, and various operators. See command **expr** on page 5-42 for details.

If the very first argument of *expression* is the keyword **(print)** then this designates that *expression* is to be evaluated as a print string. This option should only be used when *argument* references a string variable. See **fprint** for a description of the syntax of print strings.

The pseudo-variables `$ERRORS` and `$TIMER` may also be used in an expression.

Memory address values which appear on the right hand side of the = sign must be enclosed in `()` so that they may be differentiated from immediate values. A memory address value on the left hand side of the = sign can be written as is since it is not possible to assign the value of an expression to an immediate value.

In its simplest form, the **set** command works exactly like a **write** command; writing a value to an object (memory, register or variable).

However, the **set** command allows for complex expressions to be assigned whereas the write instruction does not. For example, the following command adds two (2) registers, divides the result by another and then shifts the result:

```
set R4 = LR + R0 / R17 >> 4
```

The result could have just as easily been assigned to a memory address location as opposed to the register GPR4. When using these expressions there are a few rules which must be kept in mind:

1. Expressions are always evaluated from left to right; no right associative operators are supported (+=, -=, etc.).
2. Registers, register fields, and address locations are treated as unsigned values.
3. When setting a variable that was created with the **create** or **assign** command, the variable will increase in size, if required, to contain the full value determined for the right side of the equation. Variable size expansion is done on multiples of 4 bytes.
4. Operations are performed based on the type of arguments being evaluated. The cast operator can be used to override the default size and sign.

The following list shows the supported operators and describes their functionality:

<u>Operator</u>	<u>Function</u>
~	bitwise negation(one's complement)
!	logical negation
-	arithmetic negation, subtraction
*	integer multiplication
/	integer division
%	integer modulus
mod	integer modulus
+	arithmetic addition
>>	bit shift right
<<	bit shift left
<	logical less-than
<=	logical less-than-or-equal-to
>	logical greater-than
>=	logical greater-than-or-equal-to
==	equality
!=	inequality
&	bitwise AND
^	bitwise XOR
	bitwise OR
&&	logical AND
	logical OR

The evaluation precedence is as follows, but can be overridden using parenthesis:

1. func(), literals, variables and pseudo-variables

# set

---

2. ()
3. ~ ! + -
4. \* / % mod
5. + -
6. >> <<
7. < <= > >=
8. == !=
9. &
10. ^
11. |
12. &&
13. ||

The **set** command also supports limited logical operations should this sort of processing power be desired. The logical operations are used mainly for the programming constructs of command files but have been also included for the **set** command for completeness.

One thing that must be kept in mind when using logical expressions is that their result is only one of two values; 0 or 1. They NEVER return any other value. The form of a logical expression is restricted to one basic form when it appears in a **set** command:

arg1 op arg2

In this expression, arg1 and arg2 may be simple references to registers, register fields, memory address, immediate values or user-created variables. Each argument may also consist of the type of mathematical expressions described above.

## Flags

argument = (address) | int\_var | src\_var | reg\_name[.field\_name|.#] |  
reg\_var  
expression = ([ logical|mathematical ])  
logical = expression|expression log\_op expression  
mathematical = [math\_op1] expression [math\_op2 mathematical]  
expression = reg\_name[.fld\_name|.#] | (address) | immed | variable |  
mem\_var | src\_var | func  
func = supported functions : random()  
log\_op = == != > >= < <=

---

```

math_op1      = + - ~
math_op2      = + - * / mod % & | ^ << >>
#             = ordinal bit number

```

**Note:** Registers specified must not be larger than 32 bits.

## Examples

- Write a value of 0x1234 to GPR0.

```
write R0 0x1234
```

- Use the **set** command to do the same thing.

```
set R0 = 0x1234
```

- Set the integer variable S4 to indicate if the IAR exceeded some known memory address boundary.

```
create S4
assign max = 0xFFFFC14A
set      S4 = IAR > max
```

In this example, if the IAR was greater than 0xFFFFC14A, variable S4 would get set to a 1. If not, S4 would have been set to 0.

- Set the IA1 field of register DBCR.

```
set DBCR.IA1 = 1
```

- Set bit 4 of GPR17 and clear bit 12 of GPR5.

```
set R17.4 = 1
set R5.12 = 0
```

- Set local variable 'array[7].member\_i' to the decimal value 17.

```
set :array[7].member_i = 17
```

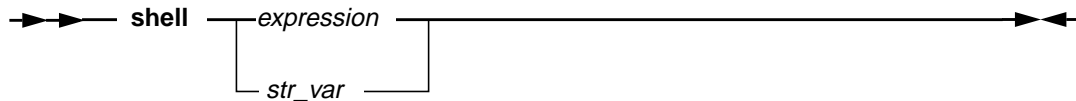
## See Also

- “Command File Programming” on page 3-125

# shell

---

## Syntax



## Description

**shell** is used to pass a user-defined command string to the native operating system for execution. The command string should only contain valid host operating system commands.

A special operating system call is used to create a new process for the command string to run under. To ensure correct command file processing, this new process is allowed to finish execution before control is returned to RISCWatch. Therefore, care must be taken as to the commands passed to the operating system using this command

## Flags

- `expression` = a string expression containing the command to be executed
- `str_var` = string variable created with the **assign** or **create** commands and containing the command to be executed.

## Syntax

▶▶— showip —————▶◀

## Description

**showip** updates the entire Debugger context based on the current Instruction Pointer address. All appropriate source debug windows are updated accordingly. For JTAG targets, the Instruction Pointer is actually the current Instruction Address Register (IAR). For non-JTAG targets, it is the process copy of the IAR for the application being debugged.

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

# socket

---

## Syntax

▶▶ **socket** — **timeout** value ▶▶

## Description

**socket** displays and alters parameters associated with socket communication to a target. If **socket** is issued without *value* to set, the current setting is displayed, otherwise the setting is changed to *value*.

## Flags

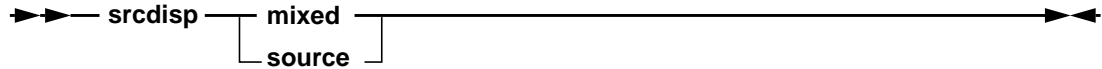
<b>timeout</b>	The length of time in seconds that RISCWatch waits for information from a target before timing out.
<i>value</i>	Number of retries or timeout value in seconds

## Examples

- Examine current timeout setting  
`socket timeout`
- Set the timeout to wait for a target to 3 seconds  
`socket timeout 3`



## Syntax



## Description

**srcdisp** changes the Source window display to show either source lines only (`source`), or mixed source/assembly lines (`mixed`). This is the same capability provided by the Source Mode groupbox on the Source window. If no parameters are entered, the mode is toggled.

## Flags

- |               |   |
|---------------|---|
| <b>mixed</b>  | Sets the Source window display to show mixed source/assembly lines. |
| <b>source</b> | Sets the Source window display to show source lines only.           |

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

## Example

- Set the Source window display to show mixed source/asm  
`srcdisp mixed`

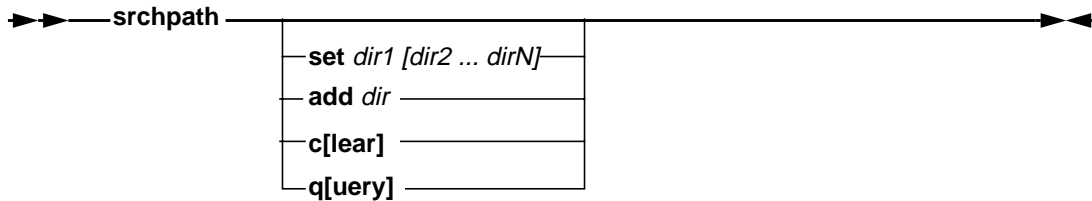
## See Also

- “Source Window” on page 3-52

# srchpath

---

## Syntax



## Description

**srchpath** determines the file search order used by the debugger to reference source files and executables. It is typically used when an unqualified file name is designated on a command. For example, if no directory path is indicated in the file name portion of the **load file** command, then the path(s) specified via the `srchpath` command are searched, in order, until the file is found. If the file is still not found, the current directory is also searched. Note that the current directory can be included anywhere in the search path by explicitly ordering it via the **srchpath** command.

Current directory is defined as the following:

- |               |   |
|---------------|---|
| UNIX platform | The directory which began the debug session. For example, if you were in <code>/home</code> , and typed <code>/usr/rwppc/rwppc</code> to start RISCWatch, the current directory would be <code>/home</code> . |
| Windows       | The Working Directory specified under the Program Manager's File-> Properties pulldown for the RISCWatch icon. It is originally set to the same directory as the installed executable.                        |

## Flags

- |                |   |
|----------------|---|
| <b>q[uey]</b>  | Shows current directory search setting in main I/O command status window  |
| <b>set</b>     | Sets the search path to the directories listed, in the order that they are entered. Note this deletes any previous setting. |
| <b>add</b>     | Adds a directory to the search path at the end of the current setting.  |
| <b>c[lear]</b> | Clears the search path setting, which will default the search to the current setting.                                       |

## Examples

- Set the search path for source and executables.

```
srchpath set /u/kjaget/sandbox /u/mandzak/lib /u/kburke/test
```

If no directory path is indicated on a file name, the search path order for source and executables is set to

1. /u/kjaget/sandbox
2. /u/mandzak/lib
3. /u/kburke/test
4. Current directory

**Note:** Qualified source file names (those shown in the Source and Files Windows), are first checked in the designated directory. If not found, the directory path is removed from the name and the search continues as defined here.

- Add a directory to the current search path.

```
srchpath add /u/marsala/lib
```

The search order would proceed as in the above example, except that /u/marsala/lib would be searched before the current directory.

## See Also

- “Environment Resources” on page 3-5
- **load on page 5-69**

# srcline

---

## Syntax



## Description

**srcline** scrolls the contents of the Source window to a source line in the current file, highlighting the line if it contains text.

This command is equivalent to the **line** command for the Source window if it is in 'Source Only' display mode. It is useful if the Source window is in 'Mixed Source/Asm' mode, where assembly instructions are interspersed with source instructions.

If the line number specified is larger than the number of source lines in the file, the last source line is shown at the bottom of the window. If the line number is not specified, the last line number specified for the command is used. The line variable initially defaults to 1.

This function is also available via the input line, as described in "Input Line Usage" on page 3-49.

## Flags

<i>int_var</i>	A created user-created variable that whose value may be used in place of the line number.
<i>imm_var</i>	An assigned user-created variable specifying an immediate value that may be used in place of the line number.
<i>line</i>	Specifies the source line number to scroll to

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

## Syntax

► **start\_thread** — *funcname* — \_ *tgrp\_id* \_ ►

## Description

**start\_thread** initializes a source mode debug session with OS Open by scheduling a thread to be queued, beginning with the function designated by *funcname*. The function must have been previously linked with or dynamically loaded on OS Open. Threads are started using OS Open default thread characteristics.

For OS Open systems that support Virtual Memory, if *tgrp\_id* is specified, the function will be started in the existing thread group *tgrp\_id*, otherwise the thread will be in its own newly formed thread group.

## Flags

<i>funcname</i>	Name of function to be started.
<i>tgrp_id</i>	ID of thread group for <i>funcname</i> .

## Restrictions

This command is only supported on OS Open targets.

## Examples

- Schedule a specified thread to be queued:

```
start_thread routine1
```

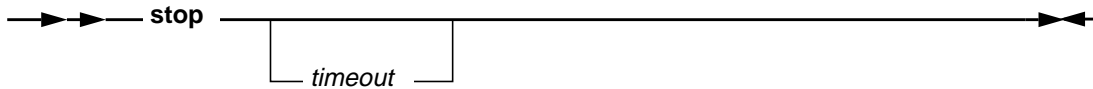
## See Also

- **attach** on page 5-16
- **detach** on page 5-35
- **kill\_thread** on page 5-66
- **load** on page 5-69

# stop

---

## Syntax



## Description

**stop** forces the processor (JTAG target) or process (non-JTAG) to stop running. This command is used whenever the processor/process is running and you want to stop it.

If **run** is issued with no timeout value and no debug events set, the processor/process keeps running until the resident program completes execution or **stop** is issued by the user.

**stop** has an optional timeout value. If a timeout value is specified and the processor/process is stopped, the timeout is ignored and the processor/process stopped normally. If a timeout value is specified and the processor/process is running, a timer is started and the processor/process is left running. If the processor/process is still running when the timer expires, the **stop** command is given to stop the processor/process. If the processor/process stops on its own before the timer expires, the timer is cancelled and the **stop** command is given to insure a stopped processor/process.

If a **run** command is issued with a timeout value and then a **stop** command is issued with a timeout value, when either command has timed out the processor/process is stopped.

## Flags

<i>timeout</i>	Specifies the number of seconds to wait before sending the <b>stop</b> command to the processor/process.
----------------	--

## See Also

- **run** on page 5-106

## Syntax



## Description

**stuff** is used to stuff a 4-byte machine instruction directly into the head of the instruction execution queue where it is immediately executed by the processor. This command must be used with caution since no error checking is done on the machine instruction that is given with the command.

The machine instruction value is sent directly to the processor so an invalid machine instruction could produce disastrous results. It would be wise to use either the **dis** or **asm** command to verify the machine instruction before the **stuff** command is executed.

It is also possible to stuff an assembly instruction into the processor using the built in line assembler. Simply enclose the assembly instruction in quotation marks and pass it to the **stuff** command. If the **stuff** command detects a string in quotes, it passes the string to the line assembler. If the instruction is assembled without error, the equivalent 4-byte machine instruction is stuffed.

Another variation of the **stuff** command allows the contents of a register or user-created variable to be stuffed. Instead of specifying an immediate value or assembly instruction string, place a register or variable name after the **stuff** command. Once entered, the contents of the register or variables are read and then stuffed into the processor.

## Flags

<i>opcode</i>	An immediate machine instruction value to be stuffed.
<i>assembly</i>	A valid assembly instruction string enclosed in quotation marks to be assembled and then stuffed.
<i>reg_name</i>	The name of a register whose contents are to be read and then stuffed. The register must not be larger than 32 bits.
<i>variable</i>	The name of a user-created variable whose contents are to be read and then stuffed.

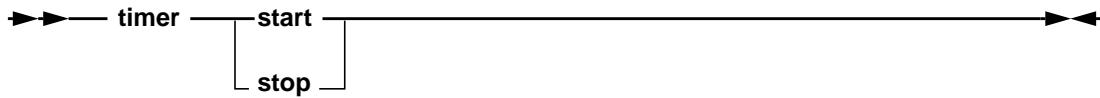
## Restrictions

This command is only supported on 4xx processors.

# timer

---

## Syntax



## Description

**timer** allows for the timing of events from within a command file. The resolution of the timer is one second.

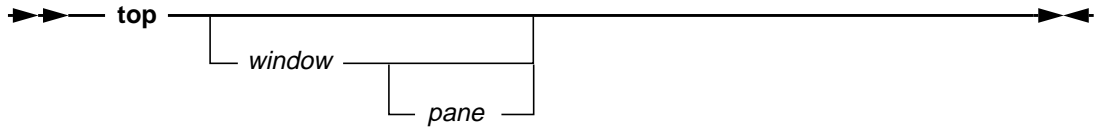
When the timer is stopped, a status message is displayed indicating the time that has elapsed since the timer was started. This elapsed time value is also stored so that it may be printed using the `$TIMER` variable in a **print/print** command. It may also be referenced in a **set** expression.

## Flags

- |              |  |
|--------------|--|
| <b>start</b> | If the timer is stopped, this flag starts it running. If the timer is running, it updates the <code>\$TIMER</code> program variable so that it may be printed while leaving the timer running. |
| <b>stop</b>  | Stops the timer and saves the time elapsed since the <b>start</b> was given into the <code>\$TIMER</code> program variable   |



## Syntax



## Description

**top** scrolls to the first line of a window, highlighting the line if it contains any text.

If the *window* keyword is not specified, the last window specified for this command is used. It initially defaults to the Source window.

## Flags

<i>window</i>	The window keyword applies to a subset of the windows listed in “Window Quick Reference” on page 5-2. The items marked with an asterisk (*) indicate the subset of valid window keywords for this command.
<i>pane</i>	See list of pane keywords in “Command Quick Reference” on page 5-3.

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

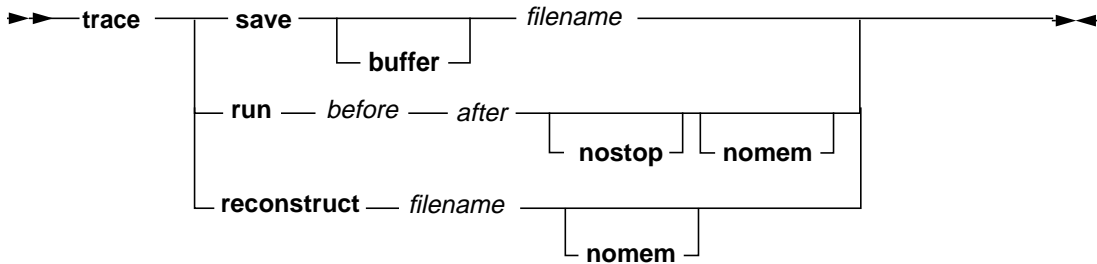
## See Also

- **bot** on page 5-18

# trace

---

## Syntax



## Description

**trace** allows start and stop control of the trace function on 400Series processors. The command assumes that the prerequisite setup has been done prior to issuing the start command. For more information about the trace capabilities of RISCWatch, see “Using RISCTrace (400Series JTAG Processor Probe Only)” on page 4-2

## Flags

- run** Specifies that the trace is to be started.
- reconstruct** Specifies that the trace output file is to be reconstructed. The trace output file is designated by *filename* and was obtained using the ‘trace save buffer filename’ command.
- save** Specifies that the trace is to be saved on disk.
- after* Specifies the number of cycles to collect after the trigger event.
- before* Specifies the number of cycles to collect before the trigger event.
- buffer** Specifies the trace buffer contents are to be saved. The resulting file can be used on the ‘trace reconstruct’ command to create a formatted trace.
- filename* Specifies the fully qualified filename in which to save the trace.
- nomem** An optional parameter that tells the RISCTrace reconstruction software to use the program files that were previously loaded with the RISCWatch load command during the reconstruction process. Any address required by the reconstruction process and not found in the program files is read from memory on the user’s target.
- nostop** An optional parameter that tells RISCWatch not to stop the processor at the completion of the trace. The user must issue the stop command or push the stop button to view the trace.

**Note:** For processors that support forward trace only, the *before* cycle count should be set to 0 and the *after* cycle count should be set to a minimum of 10 cycles and a maximum of the total number of cycles that the JTAG processor probe can handle. For processors that support

backtrace, the *after* cycle count should be set to a minimum of 150 cycles and the sum of *before* and *after* cycles should not exceed the total number of cycles that the JTAG processor probe can handle.

## Restrictions

This command is only supported on JTAG Ethernet targets

This command is only supported on 4xx processors.

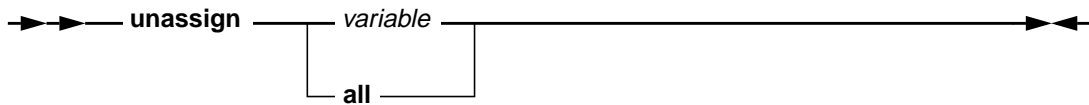
## See Also

- “Using RISCTrace (400Series JTAG Processor Probe Only)” on page 4-2

# unassign

---

## Syntax



## Description

**unassign** is used to remove a variable that was previously defined with the **assign** command.

## Flags

- |                 |  |
|-----------------|--|
| <i>variable</i> | The name given to a previously assigned variable               |
| <b>all</b>      | Indicates all previously assigned variables will be unassigned |

## Example

- Assign a register to a variable and then uses the variable to initialize and read the register's value. Unassign the variable when the read is complete.

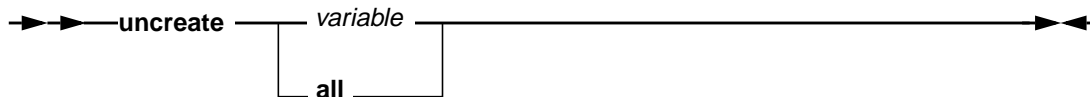
```
assign count_reg = SPRG1 # make count_reg = SPRG1
setcount_reg = 0 # init count register
read count_reg # i.e. read SPRG1
unassign count_reg # remove assignment
```
- Assign an immediate value to a variable which is then used to initialize the value of a register. Unassign this variable and all others that may exist.

```
assign reg_val = 0x11223344
set SPRG0 = reg_val
unassign all
```

## See Also

- assign** on page 5-12
- uncreate** on page 5-129

## Syntax



## Description

**uncreate** is used to remove a variable that was previously defined with the **create** command.

## Flags

<i>variable</i>	Name of the immediate variable that was previously created.
<b>all</b>	Indicates all previously created variables will be uncreated

## Examples

- Uncreate a variable named cr\_var1.
 

```

create cr_var1 = 0x1234
uncreate cr_var1
      
```
- Create two variables, i and j, and use them to calculate a value to write to GPR0. When complete, removes all previously created variables.
 

```

create i           # create variable i
create j           # create variable j
set i = (0x12345678) # read memory into i
set j = i - IAR    # subtract IAR from i
write R0 j        # write value of j to GPR 0
uncreate all
      
```

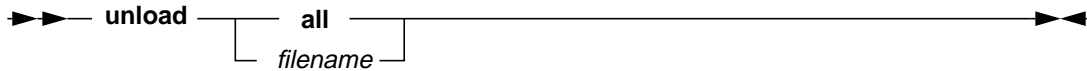
## See Also

- create** on page 5-32
- unassign** on page 5-128

# unload

---

## Syntax



## Description

**unload** removes the program specified by *filename* from the debugger. It also removes any breakpoints set within the specified program context. However, any loaded program will continue to reside in target memory.

Also, this command applies only to files loaded to perform source level debug via the **load** file or **load** host command option.

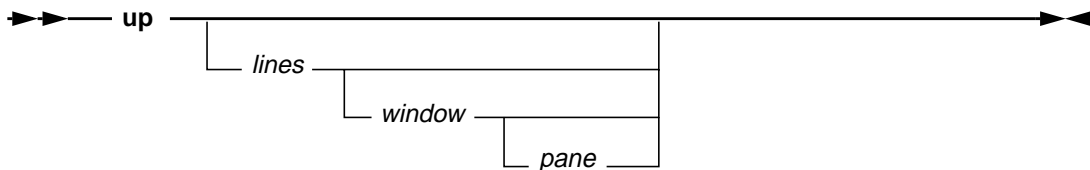
## Flags

<b>all</b>	Unloads all programs currently loaded in the debugger.
<i>filename</i>	Specifies program to be unloaded. If unqualified, the file unloaded will be determined by the <b>srchpath</b> settings currently in effect.

## See Also

- **load** on page 5-69

## Syntax



## Description

**up** scrolls the contents of a window up a number of lines from the top line visible in the window.

If the number of lines specified is larger than the number of lines from the top of the window, the first line is shown at the top of the window. If the *window* keyword is not specified, the last window specified for this command is used. *window* initially defaults to the Source window. If neither the *lines* variable nor the *window* keyword is specified, the last *lines* value and *window* specified for the command are used. The *lines* variable initially defaults to 1.

## Flags

<i>lines</i>	Specifies the number of lines to be scrolled up in <i>window</i>
<i>window</i>	The window keyword applies to a subset of the windows listed in “Window Quick Reference” on page 5-2. The items marked with an asterisk (*) indicate the subset of valid window keywords for this command.
<i>pane</i>	See list of pane keywords in “Command Quick Reference” on page 5-3.

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

# up

---

## Examples

- Scroll up two lines in a window previously specified, or the Source window if none was previously specified.

```
up 2
```

- Scroll up six lines in the Breakpoints window.

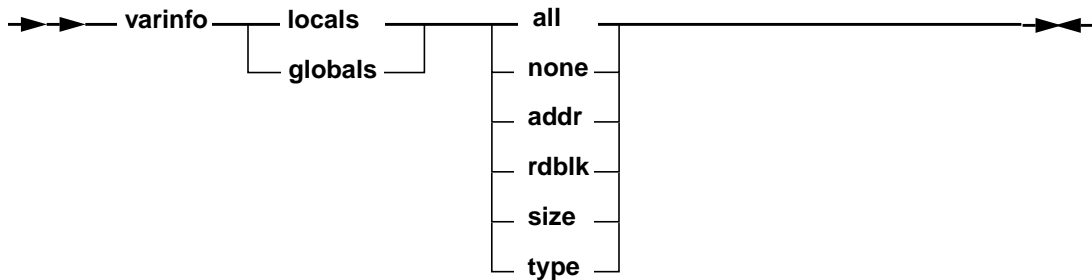
```
up 6 break
```

## See Also

- [down on page 5-38](#)



## Syntax



## Description

**varinfo** changes the Local or Global variable window display to show type, address, and size information for each visible variable. It also allows a user to specify the method used to read variable data. This is the same capability provided by the Display Information groupboxes on the Variable Configuration window.

## Flags

<b>locals</b>	Specifies Locals variable window
<b>globals</b>	Specifies Globals variable window
<b>all</b>	Shows the address, size and type for each variable
<b>none</b>	Shows no address, size and type information for each variable
<b>addr</b>	Shows the address of each variable
<b>rdblkl</b>	Specifies that the variable information is to be read as a block of contiguous data.
<b>size</b>	Shows the size of each variable
<b>type</b>	Shows the type of each variable

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

# varinfo

---

## Example

- Set the Locals window display to show address and type information for each visible variable

```
varinfo locals addr  
varinfo locals type
```

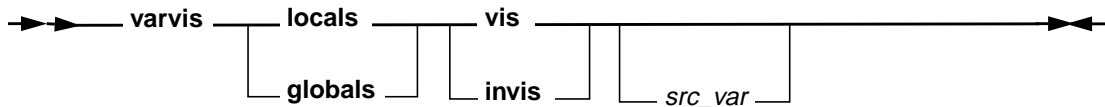
- Change the Globals window display to show address information only, and specify that the variables are each to be read as contiguous data blocks.

```
varinfo globals none # clears current settings  
varinfo globals addr  
varinfo globals rdblkl
```

## See Also

“Reading and Writing Memory” on page 3-103

## Syntax



## Description

**varvis** changes the visibility of variables on the Locals and Globals variable windows. This is the same capability provided by the relevant pushbuttons on the Variable Configuration window.

**Note:** Initially, RISCWatch will default to all local variables being visible, and all global variables being invisible. These defaults could be changed by putting the appropriate **varvis** command entries in a startup command file after a file is loaded.

## Flags

<b>locals</b>	Specifies Locals variable window
<b>globals</b>	Specifies Globals variable window
<i>src_var</i>	Any valid local or global source variable name that is currently in scope. The name must be preceded by a colon ":". If a source variable name is not specified, all variable names will be effected. See "Source Variable Command Support" on page 3-102 for further information.
<b>vis</b>	Make the variable(s) visible
<b>invis</b>	Make the variable(s) invisible

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

## Example

- Set the Globals window display to show all variables

```
varvis globals vis
```
- Show only the global variable show\_me in the Global window display.

```
varvis globals invis #clear previous settings
varvis vis :show_me
```

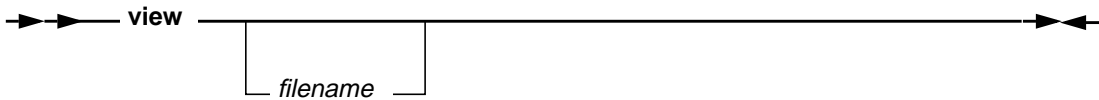
# varvis

---

## See Also

- “Variable Configuration Window” on page 3-83

## Syntax



## Description

**view** allows for a specified file to be viewed. The specification of the filename is optional. If it is not specified, a file dialog box is presented for the user to navigate the directory structure and select a file to view. This functionality is only available when you are using the graphical user interface, not from within a command file.

Once a file has been selected, a window is displayed and the contents of the file are displayed within it. The file may be viewed but not edited. Text size can be adjusted using the menubar Font pulldown. To search for specified text, enter it in the 'Find string' field and press the Enter key or click on the Find button. To perform a backward or case-sensitive search, click on the appropriate check boxes. To go to a specific line in the file, enter the line number in the 'Goto line' field and press the Enter key. To go to the last line, enter "\$", "last" or "bottom".

This command is equivalent to using the View option of the File pull-down menu.

## Restrictions

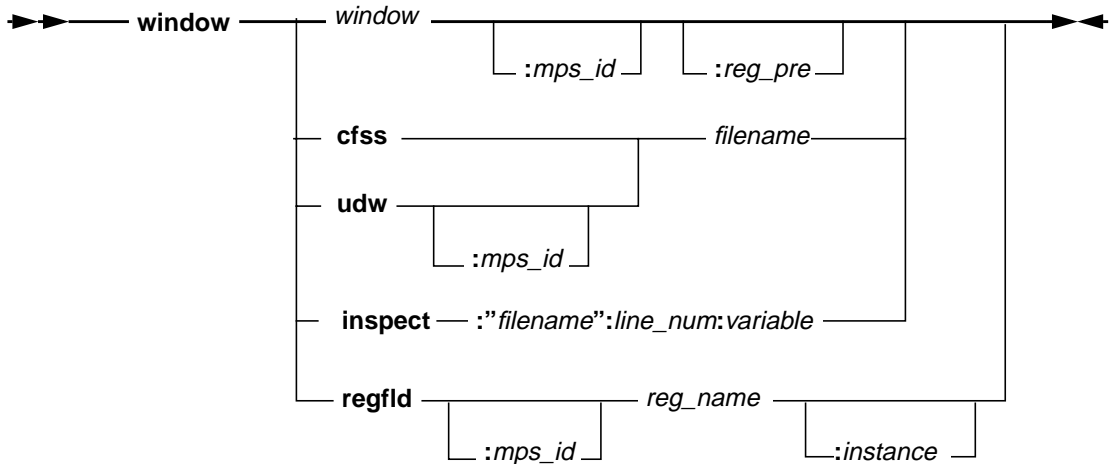
This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

# window

---

## Syntax



## Description

**window** allows the user to either bring up a new window instance or to surface an existing window from the command line interface. For new windows, this is the same capability provided by the main menu pulldowns. For existing windows that can not have multiple instances, it provides the same function as the Window List pulldown. When used for windows that already exist and can have multiple instances, a new instance is created when the command is invoked.

## Flags

- |                 |   |
|-----------------|---|
| <b>cfss</b>     | Specifies the Command File Window.  |
| <b>inspect</b>  | Specifies a Variable Inspect Window.  |
| <b>regfld</b>   | Specifies a Register Field Window.  |
| <b>udw</b>      | Specifies a User Defined Window.  |
| <i>window</i>   | The window keyword applies to a subset of the windows listed in “Window Quick Reference” on page 5-2, including items marked with an asterisk (*). The command will give an error if the window keyword specified is not valid for the command. |
| <i>filename</i> | The name of the file containing the desired variable to inspect. Note it must be enclosed in quotations.  |
| <i>line_num</i> | The specific line number within a file that contains the desired variable to inspect.   |
| <i>variable</i> | The name of the variable to inspect.  |

<i>reg_name</i>	A valid processor register whose field window is to be displayed.
<i>reg_pre</i>	When ASIC is specified for <i>window</i> , this specifies a unique ASIC window which contains the registers with the specified prefix. See the description for this flag in “Command Quick Reference” located at the beginning of this chapter.
<i>instance</i>	A register or window instance number.
<i>mps_id</i>	The mps id of the window.

## Restrictions

This command is not supported in TTY mode.

This command is not supported in Programming Interface mode.

## Examples

- Bring to the foreground the source window on Board 2 while in Board 1 context.  

```
window source:Board_2
```
- Bring up a source window in the current context.  

```
window source
```
- Bring up a second instance of the debug window in the current context, assuming one is already active.  

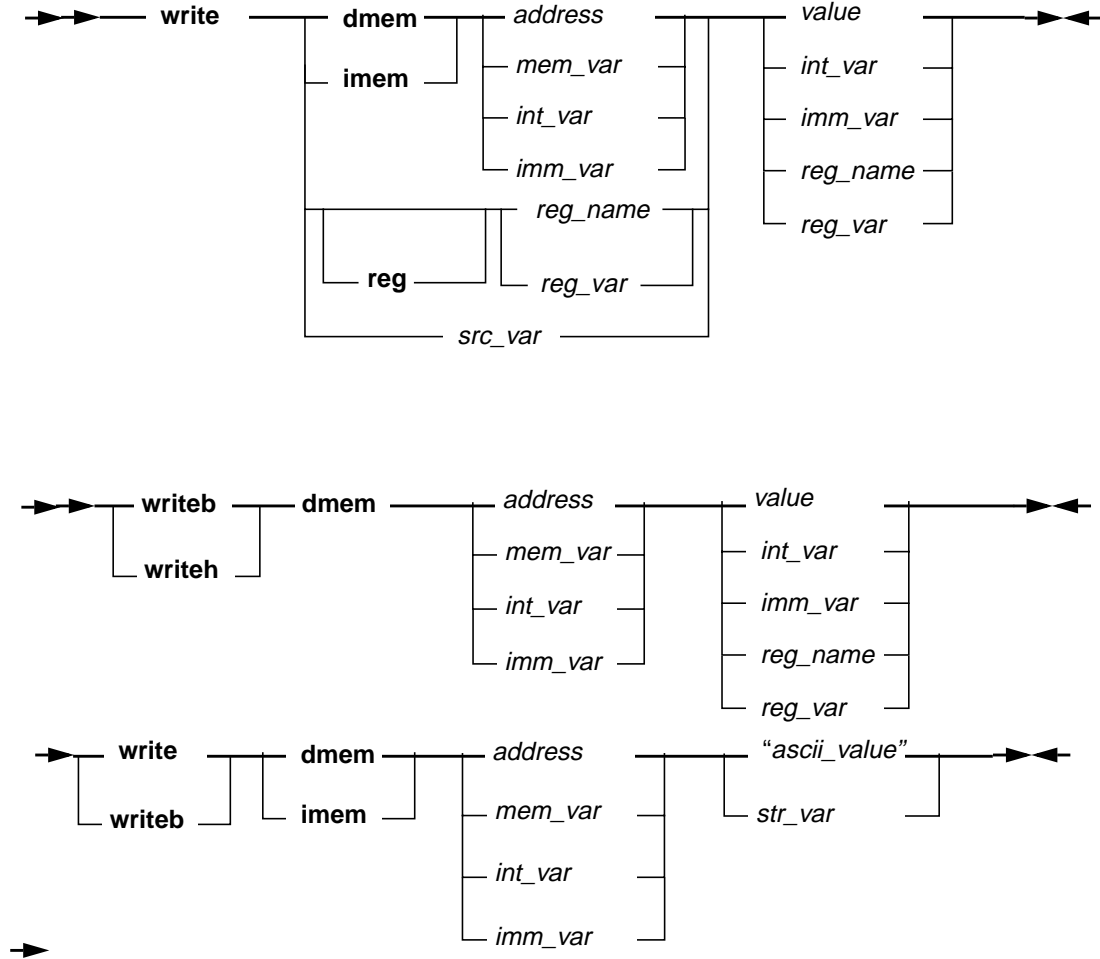
```
window debug
```
- Bring up an inspect window to look at the contents of variable `my_struct.name`, found on line 24 of file `testit.c`:  

```
window inspect : "testit.c":24:my_struct.name
```
- Bring up a user defined register field window in an MPS environment.  

```
window regfld:Board_2 MSR:2
```

# write

## Syntax



## Description

**write** is used to write a value to either a register, a 4-byte data memory location, a 4-byte instruction memory location, a source variable, or to a breakpoint register.

**writeb** is used to write a 1-byte data memory location, while **writeh** is used to write a 2-byte data memory location.

**Note:** RISCWatch by default will write and display memory exactly as it is in the target's memory and not how it is used by the processor. For limited support of little-endian targets, refer to the byte-reversed feature of the memory windows as well as the **prefer instruction br** command.



<b>write reg</b>	Write a new value to a register. Note the reg keyword is optional.
<b>write dmem</b>	Write a new value to a data memory location. Up to four (4) bytes of data can be written to a valid address.
<b>writew dmem</b>	Write a new value to a data memory location. One (1) byte of data can be written to a valid address.
<b>wrieh dmem</b>	Write a new value to a data memory location. Two (2) bytes of data are written to a valid address.
<b>write imem</b>	Write a new value to an instruction memory location.
<b>write tlb</b>	Write a new value to an entry of a unified TLB.

## Flags

<b>dmem</b>	Write to a data memory address.
<b>imem</b>	Write to an instruction memory address.
<b>reg</b>	An optional parameter indicating a write to an architected register in the chip.
<i>address</i>	Specifies an immediate value which represents the memory location to be written.
<i>ascii_value</i>	Specifies an immediate ASCII string value to be written. It must be enclosed in double quotes, The null terminator character is <b>not</b> written.
<i>expression</i>	Additional information required by the specific <b>write</b> command issued.
<i>mem_var</i>	Any memory variable created with the <b>assign</b> command.
<i>int_var</i>	A created user-created variable that may be used as the memory address to be written or as the value to be written.
<i>imm_var</i>	An assigned user-created variable specifying an immediate value that may be used as the memory address to be written or as the value to be written.
<i>reg_name</i>	A valid processor register name to be read and/or written.
<i>reg_var</i>	An assigned user-created variable that may be used to specify a processor register to be read and/or written.
<i>src_var</i>	Any valid local or global source variable name that is currently in scope. The name must be preceded by a colon ":". See "Source Variable Command Support" on page 3-102 for further information.
<i>str_var</i>	Specifies a user <b>created</b> string variable containing the ASCII data to be written. The null terminator character is <b>not</b> written.
<i>value</i>	An immediate value to be written to the specified memory address or register.

# write

---

## Examples

- Write 0xDEADBEEF to the IAR register.

```
write reg IAR 0xDEADBEEF
```

- Write 0x11112222 to GPR0.  
`write R0 0x11112222`
- Write the contents of SRR0 to R14.  
`write R14 SRR0`
- Write 0xDEADBEEF to address 0xFFFFFFFF0.  
`write dmem 0xFFFFFFFF0 0xDEADBEEF`
- Write an immediate hex value bit for bit into a 64-bit register:  
`write FPR0 0x1234567812345678`
- Write an immediate value specified in scientific notation into a 64-bit register in floating point format:  
`write FPR0 1.23456e+002`
- Write the contents of GPR3 to memory at address 0xFFFF0000.  
`write dmem 0xFFFF0000 R3`
- Write the contents of the user-created variable var1 into memory at address 0xFFFF0000.  
`create var1 = 0xDEADBEEF`  
`write dmem 0xFFFF0000 var1`
- Write the contents of the user-assigned variable mem\_val to the address found in the user-assigned memory variable mem\_addr:  
`assign mem_addr = (0xABCD1234)`  
`assign mem_val = 0xDEADBEEF`  
`write mem_addr mem_val`
- Write the contents of the user-assigned variable, mem\_val, to the address found in the user-assigned register variable, mem\_reg, which points to the R0 register.  
`assign mem_val = 0xDEADBEEF`  
`assign mem_reg = R0`  
`set R0 = 0x1234ABCD`  
`write mem_reg mem_val`
- Write the contents of source global variable 'ptr->member\_a', defined in file 'test.c', to the immediate value 0x12345678:  
`write :“test.c”:ptr->member_a 0x12345678`
- Write the ASCII value “KJ The Porcupine” into memory starting at address 0x500:  
`write imem 0x500 “KJ The Porcupine”`

**Note:** Any of the **write dmem** examples are also valid for **write imem**, just replace the word **dmem** in each example to **imem**.

# write

---

## See Also

- [read on page 5-99](#)

# Appendix A. Interfacing RISCWatch to a Target Board

This appendix describes the requirements for connecting RISCWatch to a PowerPC processor on a target development board. For the list of PowerPC processors, consult the README file for this version of RISCWatch.

## IEEE 1149.1 (JTAG) Port

For RISCWatch to interface to the JTAG port on a PowerPC processor, a 16-pin male 2x8 header connector, shown in Figure A-1, must be available on the target development board (except if a mictor connector that combines JTAG and Trace exists on the target development board as described later in this Appendix).

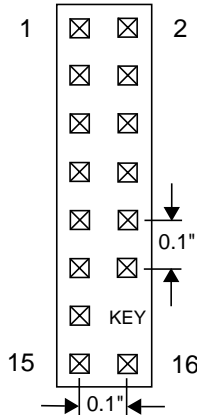


Figure A-1. JTAG Header Connector (top view)

Note that position 14 of the header connector on the target development board should not contain a pin. The mating receptacle supplied with a RISCWatch JTAG adapter cannot be installed if pin 14 has not been removed from the header.

This header connects the RISCWatch JTAG hardware (processor probe) to the JTAG port of the PowerPC processor on the target development board, using the electrical connections described below. The header should be placed as close as possible to the processor to insure signal integrity.

Table A-1 describes the header connections for the PowerPC 400Series processors, and Table A-2 provides the same for the PowerPC 6xx/7xx processors. Consult the specific processor manual for the processor pin number if required.

Table A-1. PowerPC 400Series JTAG Interface Connections and Resistors

Header Pin	I/O	Signal Name	Board Resistor <sup>1</sup>
1	Out	TDO	
2		No Connect	
3	In	TDI	10K $\Omega$ PU
4		No Connect	
5		No Connect	
6		+POWER <sup>2</sup>	1K $\Omega$ SR <sup>3</sup>
7	In	TCK	10K $\Omega$ PU
8		No Connect	
9	In	TMS	10K $\Omega$ PU
10		No Connect	
11	In	$\overline{\text{HALT}}$	10K $\Omega$ PU
12		No Connect	
13		No Connect	
14		KEY	
15		No Connect	
16		GND	

<sup>1</sup>PU = pullup, PD = pulldown, SR = series

<sup>2</sup>The +POWER signal is sourced from the target development board and is used as a reference signal by the RISCWatch Processor Probe. The voltage presented on this pin should indicate the voltage level of the processor I/O. Newer versions of the processor probe will adjust voltage levels on input pins to the processor accordingly.

<sup>3</sup>This 1K ohm series resistor provides short circuit current limiting protection only. If the resistor is present, it should be 1K ohm or less.

**Note:** Pin 4 should be connected to  $\overline{\text{TRST}}$  on the PPC405/PPC440-based processors and should be pulled up with a 10K resistor. PPC401/PPC403-based processors do not have a TRST pin. Also note that TRST, if it exists, must be asserted low in response to a power-on or system reset or else the processor may not boot reliably. It is recommended that  $\overline{\text{TRST}}$  from the JTAG connector be logically ORed with power-on reset of the board before being connected to  $\overline{\text{TRST}}$  on the processor.

Table A-2. PowerPC 6xx/7xx JTAG Interface Connections and Resistors

Header Pin	I/O	Signal Name	Board Resistor <sup>1</sup>
1	Out	TDO	
2		No Connect	
3	In	TDI	10K $\Omega$ PU
4	In	TRST	10K $\Omega$ PU
5		No Connect	
6		+POWER <sup>2</sup>	1K $\Omega$ SR <sup>3</sup>
7	In	TCK	10K $\Omega$ PU
8		No Connect	
9	In	TMS	10K $\Omega$ PU
10		No Connect	
11	In	SRESET	10K $\Omega$ PU
12		No Connect	
13	In	HRESET	10K $\Omega$ PU
14		KEY	
15	Out	CHECKSTOP	10K $\Omega$ PU
		CKSTP_OUT	
16		GND	
N/A	In	QACK <sup>4</sup>	1K $\Omega$ PD
		L2_TEST_CLK	10K $\Omega$ PU
		L1_TEST_CLK	
		LSSD_MODE	
		ARRAY_WR	

<sup>1</sup>PU = pullup, PD = pulldown, SR = series

<sup>2</sup>The +POWER signal is sourced from the target development board and is used as a reference signal by the RISCWatch Processor Probe. The voltage presented on this pin should indicate the voltage level of the processor I/O. Newer versions of the processor probe will adjust voltage levels on input pins to the processor accordingly.

<sup>3</sup>This 1K ohm series resistor provides short circuit current limiting protection only. If the resistor is present, it should be 1K ohm or less.

<sup>4</sup>If the target development board does not use this signal, the board must have a 1K $\Omega$  PD connected to this pin. This signal allows the processor to enter the soft stop state. Otherwise, the target development board must provide the proper logic, so that the QACK goes Low in response to a QREQ. If the proper logic is not provided, the processor will not be able to enter the soft stop state.

The HRESET, SRESET, and TRST signals from the RISCWatch Processor Interface Assembly connector must be logically ORed with the HRESET, SRESET, and TRST signals that connect to the processor on the target development board. They cannot be “dotted” or

“wire-ORed” on the board. In addition, the ORed signals should only reset the processor and no other devices on the target board.

For further information concerning RISCWatch support for processor reset, see “Processor Reset Window (JTAG Targets Only)” on page 3-134.

---

## Trace Status Port (400Series JTAG Processor Probe Only)

A 20-pin male 2x10 header connector (3M 3592-6002 or equivalent) may be used for connecting to the Trace Status Port of a PowerPC 401/403/405 processor but the recommended method is to use a micro connector as described later in this Appendix. The connector outline, shown in Figure A-2, and the signal descriptions in Table A-3 match the

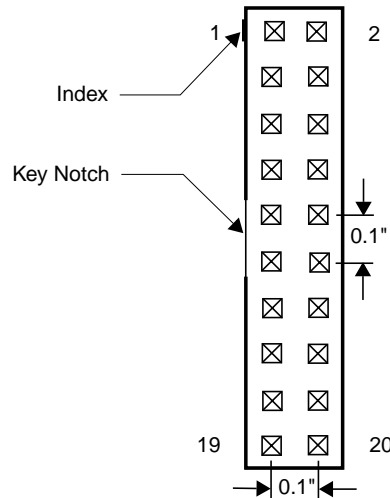


Figure A-2. RISCTrace Header (top view)

requirements of RISCTrace, when used with the RISCWatch processor probe with RISCTrace option or the TracePort Analyzer. The connector for RISCTrace should be placed as close as possible to the processor to insure signal integrity(2 inches or less).

For the 405, it is recommended that the trace signals have 33 ohm source terminators and that the connections from the trace pins to the trace connector be of equal length.

There are seven Trace Status(TS) signals, TS0:6, on the PPC403GA and PPC403GC/GCX processors. There are six Trace Status signals, TS1:6, on the PowerPC 401 Core processors. There are eight Trace Status signals, TS1:2O, TS1:2E and TS3:6, on the PowerPC 405 Core processors. These signals are sampled on the rising edge of the trace clock and must satisfy a minimum of 2 nsec setup and 1 nsec hold.



Table A-3 describes the assignment of the TS signals and the trace clock (TrcClk) output to the header pins:

Table A-3. RISCTrace Header Pin Description

Pin	Signal Name	Pin	Signal Name
1	No Connect	11	No Connect
2	No Connect	12	TS1O
3	TrcClk	13	TS0/TS2O
4	No Connect	14	TS1/TS1E
5	No Connect	15	TS2/TS2E
6	No Connect	16	TS3
7	No Connect	17	TS4
8	No Connect	18	TS5
9	No Connect	19	TS6
10	No Connect	20	GND

**Note:** TrcClk is the system clock for the PPC403GA/GC/GCX processors. For the 401- and 405-based processors, TrcClk is a processor output. The TS signals are sampled on the rising edge of TrcClk and must satisfy a minimum of 2 nsec setup and 1 nsec hold.

**Note:** Pins 12 and 13 are 'No Connect' for 401-based processors.

**Note:** Pin 12 is 'No Connect' for 403-based processors.

**Note:** For the 405, it is recommended that the trace signals have 33 ohm source terminators and that the connections from the trace pins to the trace connector be of equal length and less than 2 inches long.

For additional information, see "Using RISCTrace (400Series JTAG Processor Probe Only)" on page 4-2.

## JTAG and Trace Connector Requirements

Individual JTAG and Trace connectors must be placed as close as possible to the processor and close to each other. It is highly recommended that individual JTAG and Trace connectors be placed parallel to each other with the orientation exactly as shown below

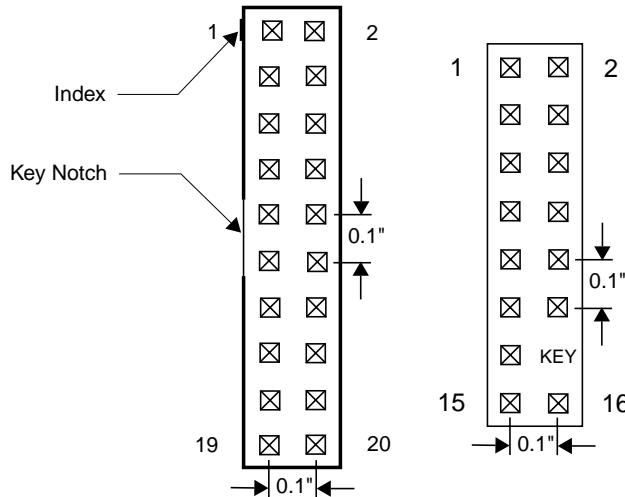


Figure A-3. RISCWatch 2 x 8 Headers and RISCTrace 2 x 10 Headers

Support for individual JTAG and Trace connectors is being replaced by one combined Mictor connector (AMP 2-76704-2 or equivalent, CONN SM MICTOR 2x19 38 PIN W/ SHROUD) for better electrical and mechanical characteristics as shown below. Note that any pin not listed in the tables below is a don't care as far as RISCWatch is concerned. Also note that there is no GND pin. Grounding is achieved through the inner tab of the Mictor connector.

Table A-4. Mictor Connector Signal Assignments

PIN	PPC401	PPC403	PPC405	Notes
6	TrcClk	TrcClk	TrcClk	
7	HALT	HALT	HALT	10K $\Omega$ pullup
11	TDO	TDO	TDO	
12	VREF	VREF	VREF	(POWER SENSE)
15	TCK	TCK	TCK	10K $\Omega$ pullup
17	TMS	TMS	TMS	10K $\Omega$ pullup
19	TDI	TDI	TDI	10K $\Omega$ pullup

Table A-4. Mictor Connector Signal Assignments

PIN	PPC401	PPC403	PPC405	Notes
21	N/A	N/A	TRST	
24			TS1O	
26		TS0	TS2O	
28	TS1	TS1	TS1E	
30	TS2	TS2	TS2E	
32	TS3	TS3	TS3	
34	TS4	TS4	TS4	
36	TS5	TS5	TS5	
38	TS6	TS6	TS6	

**Note:** The VREF/POWER SENSE signal should be sourced from the target development board through a 1K ohm (or less) series resistor. This resistor provides short circuit current limiting protection. The voltage presented on this pin is used as a reference signal by the RISCWatch Processor Probe and should therefore indicate the operating voltage level of the processor I/O. Newer versions of the processor probe will sense this voltage level and use it as a maximum value for all signals driven into the processor.

**Note:** Pin 21 should be connected to  $\overline{\text{TRST}}$  on the PPC405-based processors and should be pulled up with a 10K resistor. PPC401/PPC403-based processors do not have a  $\overline{\text{TRST}}$  pin. Also note that  $\overline{\text{TRST}}$ , if it exists, must be asserted low in response to a power-on or system reset or else the processor may not boot reliably. It is recommended that  $\overline{\text{TRST}}$  from the JTAG connector be logically ORed with power-on reset of the board before being connected to  $\overline{\text{TRST}}$  on the processor.

**Note:** TrcClk is the system clock for the PPC403GA/GC/GCX processors. For the 401- and 405-based processors, TrcClk is a processor output. The TS signals are sampled on the rising edge of TrcClk and must satisfy a minimum of 2 nsec setup and 1 nsec hold relative to the rising edge of TrcClk.

**Note:** Pins 24 and 26 are 'No Connect' for 401-based processors.

**Note:** Pin 24 is 'No Connect' for 403-based processors.

**Note:** For the 405, it is recommended that the trace signals have 33 ohm source terminators and that the connections from the trace pins to the trace connector be of equal length and less than 2 inches long.

Table A-5. Mictor Connector Signal Assignments for 440

PIN	PPC440	Notes
6	TrcClk	
7	HALT	10K $\Omega$ pullup
11	TDO	

Table A-5. Mictor Connector Signal Assignments for 440

PIN	PPC440	Notes
12	VREF	(POWER SENSE)
15	TCK	10K $\Omega$ pullup
17	TMS	10K $\Omega$ pullup
19	TDI	10K $\Omega$ pullup
21	TRST	
25	BS0/BR0	
27	BS1/BR1	
29	BS2/BR2	
31	ES0	
33	ES1	
35	ES2	
37	ES3	
24	ES4	
26	TS0	
28	TS1	
30	TS2	
32	TS3	
34	TS4	
36	TS5	
38	TS6	

**Note:** The VREF/POWER SENSE signal should be sourced from the target development board through a 1K ohm (or less) series resistor. This resistor provides short circuit current limiting protection. The voltage presented on this pin is used as a reference signal by the RISCWatch Processor Probe and should therefore indicate the operating voltage level of the processor I/O. Newer versions of the processor probe will sense this voltage level and use it as a maximum value for all signals driven into the processor.

**Note:** Pin 21 should be connected to  $\overline{\text{TRST}}$  on the PPC440-based processors and should be pulled up with a 10K resistor. Also note that  $\overline{\text{TRST}}$ , if it exists, must be asserted low in response to a power-on or system reset or else the processor may not boot reliably. It is recommended that  $\overline{\text{TRST}}$  from the JTAG connector be logically ORed with power-on reset of the board before being connected to  $\overline{\text{TRST}}$  on the processor.

**Note:** For the 440-based processors, TrcClk is a processor output. The BS/ES/TS signals are sampled on the rising edge of TrcClk and must satisfy a minimum of 2 nsec setup and 1 nsec hold relative to the rising edge of TrcClk.

**Note:** For the 440, it is recommended that the trace signals have 33 ohm source terminators and that the connections from the trace pins to the trace connector be of equal length and less than 2 inches long.

For additional information, see “Using RISCTrace (400Series JTAG Processor Probe Only)” on page 4-2.

---

## **Target Monitor Debugging**

In addition to RISCWatch communicating directly to processor hardware via a JTAG connection, RISCWatch can also communicate with target monitor software included in both the IBM OS Open real-time operating system and the PowerPC evaluation kit ROM monitor. This communication can must use an Ethernet (TCP/IP) connection.

Custom target monitors can also be created using the available Board Support debug libraries supplied in the PowerPC evaluation kits. This provides the ability to port the software debug capabilities of RISCWatch to custom board solutions.

For further information, consult the OS Open and evaluation kit documentation listed in “Related IBM Publications” on page xxiv.



---

## Appendix B. Register Definition File (Outdated)

This appendix describes the file format of the Register Definition File (RDF). Starting with RISCWatch 4.3, the Processor Configuration File should be used in place of the RDF. In addition to supporting the RDF features, the PCF format enables users to define their own chips, macros and IMR registers. Please refer to section “Processor Configuration File (PCF)” on page 3-9 for details on how to convert your Register Definition File to the PCF format.

### Register Definition File

When RISCWatch is first started, the environment file (**rwppc.env**) is read to determine the debug environment. The **PROC** environment variable is used by RISCWatch to enable a unique set of predefined processor registers. For example, if the **PROC** environment variable indicates a 401 core (ie. 401m1), RISCWatch will only enable the SPR registers defined for that core processor.

By using the **REG\_FILE** environment variable, users can identify a customized Register Definition File. This file, created by the user prior to starting RISCWatch, contains additional register definitions that RISCWatch will add to its list of valid processor registers.

The Register Definition File is searched for using the following rules:

- If the file name is qualified (directory path indicated), the file search is performed using the specified directory only.
- If the name is not qualified, the file search is performed using the directory paths designated with the RISCWatch **SEARCH\_PATH** environment variable. If not found, the current directory is searched.

### File Syntax

The Register Definition File is an ASCII file that can be created with any text editor. The file is identified to RISCWatch via the **REG\_FILE** environment variable, and must have a file extension of “.reg”. A sample Register Definition File, called **rwppc.reg**, is provided with RISCWatch and contains comments which detail the required syntax described here.

The general syntax rules are as follows:

- The “#” character denotes the start of a comment. All text following the “#” character on a given line will be ignored.
- Blank lines are allowed and will be ignored.
- Any error detected during the processing of the Register Definition File will surface an error message which will be saved in the RISCWatch log file and execution will terminate.

The following sections define the complete list of valid line entries. Unless specifically stated otherwise, a record is defined to be a single line contained in the Register Definition File.

## DCR Register Definitions

A DCR register definition identifies a unique register that can be accessed via the PowerPC mtdcr and/or mfdcr instructions. Each DCR record must adhere to the following syntax:

**DCR** *name number size type* [**VOLATILE**]

Where:

- **DCR** indicates a new DCR register definition and must appear in uppercase.
- *name* indicates the name of the register being defined and must not exceed 6 characters.
- *number* indicates the DCR register number, as defined by the PowerPC mfdcr or mtdcr instruction. Valid numbers can be expressed in hex (leading "0x" or "0X"), octal (leading "0"), or decimal.
- *size* is a decimal number indicating the number of bits in the register.
- *type* indicates the type of access allowed. Valid types are "R" (read only), "W" (write only), or "RW" (read and write).
- **VOLATILE** is an optional keyword which indicates this register will change its value after a read operation is performed. It must be entered in uppercase. RISCWatch users must issue an explicit read to display the contents of a volatile register. Having the auto-update mode enabled on a window containing these registers will not cause them to be read during the update.

Examples:

DCR BRCRH0 0x70 32 RW

DCR BEAR 0x090 32 R

DCR records are valid for PowerPC 400 Series processors only.



## SPR Register Definitions

An SPR register definition identifies a unique register that can be accessed via the PowerPC `mtspr` and/or `mfspir` instructions. Each SPR record must adhere to the following syntax:

**SPR** *name number size type* [**VOLATILE**]

Where:

- **SPR** indicates a new SPR register definition and must appear in uppercase.
- *name* indicates the name of the register being defined and must not exceed 6 characters.
- *number* indicates the SPR register number, as defined by the PowerPC `mfspir` or `mtspr` instruction. Valid numbers can be expressed in hex (leading "0x" or "0X"), octal (leading "0"), or decimal.
- *size* is a decimal number indicating the number of bits in the register.
- *type* indicates the type of access allowed. Valid types are "R" (read only), "W" (write only), or "RW" (read and write).
- **VOLATILE** is an optional keyword which indicates this register will change its value after a read operation is performed. It must be entered in uppercase. RISCWatch users must issue an explicit read to display the contents of a volatile register. Having the auto-update mode enabled on a window containing these registers will not cause them to be read during the update.

Examples:

```
SPR estat 0x03d4 32 RW
```

```
SPR DEAR02 0x03d5 32 R
```

SPR records are valid for PowerPC 400 Series processors only. SPR records allow users to create their own register names of any core SPR registers. They provide a form of register name aliasing which can be used in conjunction with `FLDDEF` records to customize the display of core registers.

## MMIO Register Definitions

An MMIO register definition identifies a unique ASIC memory-mapped register that can be accessed via the PowerPC load and/or store instructions. Each MMIO record must adhere to the following syntax:

**MMIO** *name address size type* [*access*] [**VOLATILE**]

Where:

- **MMIO** indicates a new memory-mapped register definition and must appear in uppercase.
- *name* indicates the name of the register being defined and must not exceed 6 characters.
- *address* is a hex number indicating the address to use on the appropriate PowerPC load or store instruction. A leading “0x” or “0X” is allowed, but not required.
- *size* is a decimal number indicating the number of bits in the register.
- *type* indicates the type of access allowed. Valid types are “R” (read only), “W” (write only), or “RW” (read and write).
- *access* is an optional parameter which is used on JTAG ethernet RISCWatch targets. It is a decimal number which indicates the access size, in bits, RISCWatch must use when reading or writing this memory location. The access size should be a multiple of eight, with each multiple identifying a unique PowerPC load/store instruction to use. For example, an access size of “16” instructs RISCWatch to read the register by executing the “load halfword” PowerPC instruction. Specifying an access size will override any access size settings made with the **memacc** command. If no access size is specified, RISCWatch will use the access size defined for the memory region. See **memacc** on page 5-75 for information about how to set up a unique memory region access size.
- **VOLATILE** is an optional keyword which indicates this register will change its value after a read operation is performed. It must be entered in uppercase. RISCWatch users must issue an explicit read to display the contents of a volatile register. Having the auto-update mode enabled on a window containing these registers will not cause them to be read during the update.

Examples:

```
MMIO ASIC01 0000A000 32 RW
```

```
MMIO ASIC02 0000A004 32 RW 8
```

MMIO records are valid for all PowerPC processors.

## ALIAS Definitions

An ALIAS definition identifies a new name for one of the predefined processor registers. Each ALIAS record must adhere to the following syntax:

**ALIAS** *new\_name = old\_name*

Where:

- **ALIAS** indicates a new ALIAS register definition and must appear in uppercase.
- *new\_name* indicates the name of the register being defined and must not exceed 6 characters.
- *old\_name* indicates a valid register name for the target processor, which may include any previously processed SPR, DCR, or MMIO records.

Examples:

ALIAS PC = IAR

ALIAS GPR0 = R0

ALIAS records are valid for all PowerPC processors.

### Register Field Definitions

Register field definitions span multiple lines of the file and are used to indicate field names for contiguous groups of bits in a register. Each register field definition must adhere to the following syntax:

```
FLDDEF reg_name
        field_name start_bit size
        ...
ENDFLDDEF
```

Where:

- **FLDDEF** indicates the start of a new register field definition and must appear in uppercase.
- *reg\_name* indicates a valid register name for the target processor, which may include any previously processed SPR, DCR, or MMIO records. Subsequent *field\_name* records will be assigned to this register.
- *field\_name* indicates the name given to a contiguous group of register bits and must not exceed 6 characters. This record is only allowed between enclosing **FLDDEF** and **ENDFLDDEF** records.
- *start\_bit* is a decimal number which indicates the first bit of the register associated with this field name. A value of zero indicates the first bit of the register. This value should not exceed the bit size of the register.
- *size* is a decimal number which indicates the total number of bits assigned to this field name. The sum of *start\_bit* and *size* should not exceed the total bit size of the register.
- ..... indicates one or more *field\_name* records which are used to completely define field names to all bits of the designated register.
- **ENDFLDDEF** indicates the end of a register field definition and must appear in uppercase.

Example:

```
FLDDEF estat
    mcheck 0 4
    progexc 4 3
    resv0a 7 1
    storexc 8 2
    resv0b 10 22
ENDFLDDEF
```

Register field definitions are valid for all PowerPC processors. They are generally used to assign bit field names to user defined registers and core processor registers which do not have any predefined bit fields.

---

# Index

## Numerics

400Series features 3-59

403GC/GCX MMU 4-1

## A

application programs

  demos 2-2, 2-3, 2-6

  file format 1-1

  programming languages 1-1

ASCII Memory window 3-107, 3-109

asmstep command 3-49, 5-11

Assembly Debug window 2-13, 3-48, 3-53, 3-55, 3-75

assembly stepping 3-58, 3-59

assign command 5-12

assm command 5-14

attach command 3-46, 3-48, 3-53, 5-16

## B

beep command 5-17

boot files 3-47

boot image files 3-47

bot command 5-18

bp command 3-72, 5-19

bpmode command 3-54, 3-57, 3-64, 5-23

Breakpoint Mode 3-54, 3-64, 3-72, 3-73, 3-74, 3-75

Breakpoint Select window 3-76

breakpoints

  clearing 3-57, 3-75

  hardware 3-57, 3-71, 3-73, 4-7

  setting 3-57, 3-75

  software 3-57, 3-71, 3-72

Breakpoints window 2-5, 3-74

## C

cache coherency 3-105

Cache windows 3-112

Calculator window 3-142

Callers window 2-7, 3-60

callstep command 3-48, 5-25

capture command 3-141, 5-26

capture file

  rwppc.cap file 3-141

cfss command 5-28

Change Array Variable window 2-11

Change Variable windows 3-85

color command 5-29

command file programming 3-125

command files 3-1, 3-123

  blank lines 3-125

  comments 3-125

  execution 3-37, 3-131

  Output window 3-136

  parameter definition 3-128

  parameter list 3-128

  programming example 3-130

  programming syntax 3-125

  pseudo-variables 3-129

  shell scripts 3-124

  special commands 3-124

  special expressions 3-127

  window 3-132

command history usage 3-44

command history window 2-2, 3-41

command line usage 3-44

commands

  asmstep 3-49, 5-11

  assign 5-12

  assm 5-14

  attach 3-46, 3-48, 3-53, 5-16

  beep 5-17

  bot 5-18

  bp 3-72, 5-19

  bpmode 3-54, 3-57, 3-64, 5-23

  callstep 3-48, 5-25

  capture 3-141, 5-26

  cfss 5-28

  color 5-29

  create 3-128, 5-32, 5-129

  delay 5-34

  detach 5-35

  dis 5-36

  down 5-38

  end 5-40

  exec 3-129, 3-131, 5-41

  expr 5-42

  fctrl 5-43

- file 5-45
- find 5-46
- findb 5-49
- finde 5-50
- focus 5-52
- fold 5-53
- fprdisp 5-54
- fprint 3-129, 5-55
- freeze 5-58
- funcdisp 5-59
- goto 5-61
- halt 5-62
- hidewins 5-63
- ip 5-64
- jtag 5-65
- jtagclk 5-31
- kill\_thread 5-66
- line 5-67
- linestep 3-48, 5-68
- load 3-45, 3-47, 3-53, 5-69
- load image 3-48
- log 5-72
- logging 5-73
- logoff 3-48, 5-74
- memacc 5-75
- memchk 5-78
- memcoh 5-79
- memcpy 5-81
- memfill 5-79, 5-82
- memfind 5-83
- memrwait 5-85, 5-86
- mpisset 5-87
- pagedn 5-88
- pageup 5-89
- parms 3-128, 5-90
- poll 5-91
- post 5-92
- prefer 5-93
- print 3-129, 5-97
- quit 5-98
- read 5-100
- reg 5-102
- reset 5-103
- restart 3-46, 3-49, 3-53, 5-104
- retstep 3-48, 5-105

- run 3-48, 3-72, 5-106
- save 5-108
- set 3-129, 5-110
- showip 5-115
- socket 5-116
- srcdisp 5-117
- srchpath 3-62, 5-118
- srcline 5-120
- start\_thread 3-46, 3-48, 3-53, 5-121
- stop 5-122
- stuff 5-123
- timer 5-124
- top 5-125
- trace 5-126
- unassign 5-128
- unload 5-130
- up 5-131
- varinfo 5-133
- varvis 5-135
- view 3-62, 5-138
- window 5-138
- write 5-140
- Compound Trigger/Trace window 4-11
- conventions
  - highlighting xxii
  - input xxii
  - numeric notation xxii
- create command 3-128, 5-32, 5-129
- cross-development environment 1-1
- current directory
  - definition 5-118

## D

- data coherency 3-105
- DCRs 3-115
- debugger
  - loading files 3-45
- debugger facilities 3-1
- debugger quick reference 3-2, 4-1
- default capture file 3-141
- delay command 3-124, 5-34
- demo programs 2-2, 2-3, 2-6
- detach command 5-35
- Device Control Registers 3-115
- directory

- current 5-118
- dis command 5-36
- down command 5-38

**E**

- end command 3-124, 5-40
- environment resources 3-5
  - rwppc.env file 3-5, 3-48, 3-135, 3-140
  - target name 3-5
  - target type 3-5
- exec command 3-129, 3-131, 5-41
- executing the program 3-48
- expr command 5-42

**F**

- fctrl command 5-43
- file command 5-45
- file formats 1-1
- File menu 3-41, 3-43
- file Syntax 3-118, 3-119
- file syntax 3-8
  - user-defined window 3-121
- Files window 3-62
- find command 5-46
- findb command 5-49
- finde command 5-50
- Flags 5-65
- Floating Point Registers 3-115, 3-116
- focus command 5-52
- fold command 5-53
- following program execution flow 3-48
- forms
  - reader's comments xix
  - user's comments xvii
- fprdisp command 5-54
- fprint command 3-124, 3-129, 5-55
- FPRs 3-115, 3-116
- freeze command 5-58
- funcdisp command 5-59
- Functions mode 3-62
- Functions window 2-6, 3-62, 3-64

**G**

- General Purpose Registers 3-115
- Globals window 3-79
- goto command 5-61

- GPRs 3-115

## H

- halt command 5-62
- Hardware menu 3-41
- Help menu 3-41, 3-44
- Help window 3-143
- hidewins command 5-63
- host systems
  - PC 2-2
  - RS/6000 2-2
- how to use this book xxi

## I

- IEEE 1149.1 port A-1
- input line usage 3-49
- Inspect Variable window 3-81
- Inspect variable window 3-81
- Instruction Address Register 3-58
- instruction pointer 3-48, 3-53, 3-59
- instruction, assembly 3-56
- ip command 5-64

## J

- jtag command 5-65
- JTAG Ethernet target 3-5
- JTAG port A-1
- JTAG target 2-2, 3-41, 3-43, 3-46, 3-54, 3-57, 3-104, 3-112
- jtagclk command 5-31

## K

- kill\_thread command 5-66

## L

- line command 5-67
- linestep command 3-48, 5-68
- load command 2-1, 3-45, 3-47, 3-53, 5-69
- load image command 3-48
- loading files 3-45
- Locals window 2-9, 3-78
- log command 5-72
- Log Comment window 3-140
- log files 3-139
  - creation 3-139
  - directory 3-140

- disabling 3-140
- user commenting 3-140
- logging command 3-140, 5-73
- Logging State window 3-140
- logoff command 3-48, 5-74

## M

- Main window 3-40, 3-48
  - command history 2-2, 3-41
  - command history usage 3-44
  - command line usage 3-44
  - message window 3-41, 3-45
- managing breakpoints 3-71
- memchk command 5-78
- memcpy command 5-81
- memfill command 5-79, 5-82
- memfind command 5-83
- memory
  - reading 3-103
  - writing 3-103
- Memory Coherency window 3-104
- memory management unit
  - PPC403GC/GCX 4-1
- memrwait command 5-85
- memwait command 5-86
- menus 3-41
  - File menu 3-41, 3-43
  - Hardware menu 3-41
  - Help menu 3-41, 3-44
  - Source menu 3-41, 3-43
  - Utilities menu 3-41, 3-43
  - Window 3-135
  - Window menu 3-41, 3-43
- message window 3-41
- Mixed source/assembly mode 2-14, 3-52, 3-53, 3-54
- mpsset command 5-87

## O

- online help 3-41, 3-44, 3-143
- operating modes
  - batch (command file) 3-1
  - normal 3-1
  - TTY 3-1, 3-37
- OS Open
  - ELF version 3-5

- OS Open target 3-46, 3-47, 3-48, 3-54, 3-57, 3-67, 3-71, 4-10, 4-13
- OS Open window 3-67

## P

- pagedn command 5-88
- pageup command 5-89
- parms command 3-124, 3-128, 5-90
- PC host 2-2
- poll command 5-91
- post command 5-92
- prefer command 5-93
- preparing the program for debug 3-45
- print command 3-124, 3-129, 5-97
- Processor Reset window 3-134
- processor/process status indicator 3-58
- programming languages 1-1
- programming, command files 3-125
- Programs window 3-59
- pseudo-variables 3-129

## Q

- quit command 5-98

## R

- read command 5-100
- reader's comments form xix
- reading and writing memory 3-103
- reading and writing registers 3-115
- reading the syntax diagrams xxiii
- reg command 5-102
- Register Definition File B-1
- Register definition file B-1
- registers
  - Device Control 3-115
  - Floating Point 3-115, 3-116
  - General Purpose 3-115
  - reading 3-115
  - Segment 3-115, 3-116
  - Special Purpose 3-115
  - writing 3-115
- related publications xxiv
- reset 5-103
- reset command 5-103
  - Processor Reset window 3-134
- restart command 3-46, 3-49, 3-53, 5-104



- retstep command 3-48, 5-105
- RISCWatch connector A-1
- ROM Monitor 3-5
- ROM Monitor target 2-2, 3-46, 3-48, 3-54, 3-57, 4-10, 4-13
- RS/6000 host 2-2
- run command 3-48, 3-72, 5-106
- running a command file 3-131
- rwppc.cap file 3-141
- rwppc.env file 3-5, 3-48, 3-135, 3-140
- rwppc.wdf file 3-123

## S

- sample user-defined window file 3-121
- save command 5-108
- screen capture 3-141
- Segment Registers 3-115, 3-116
- set command 3-129, 5-110
- shell scripts 3-124
- showip command 5-115
- socket command 5-116
- Source menu 3-41, 3-43
- source mode 3-53
- Source window 3-48, 3-52
  - Mixed source/assembly mode 2-14, 3-52, 3-53, 3-54
- Special Purpose Registers 3-115
- SPRs 3-115
- srcdisp command 5-117
- srchpath command 3-62, 5-118
- srcline command 5-120
- SRs 3-115, 3-116
- start\_thread command 3-46, 3-48, 3-53, 5-121
- stop command 5-122
- stuff command 5-123
- syntax diagrams, how to read xxiii

## T

- target board
  - RISCWatch connector A-1
- target name 3-5
- target type 3-5
  - JTAG 2-2, 3-41, 3-43, 3-46, 3-54, 3-57, 3-104, 3-112
  - OS Open 3-46, 3-47, 3-48, 3-67, 3-71, 4-10, 4-13

- ROM Monitor 2-2, 3-46, 3-48, 4-10, 4-13
- timer command 5-124
- TLB window 4-14
- top command 5-125
- trace command 5-126
- translation lookaside buffer 4-14
- Trigger/Trace window 4-7

## U

- unassign command 5-128
- unload command 5-130
- up command 5-131
- user's comments form xvii
- using hardware breakpoints 3-73
- using software breakpoints 3-72
- Utilities menu 3-41, 3-43

## V

- Variable Configuration window 2-9, 3-79, 3-81, 3-83
- varinfo command 5-133
- varvis command 5-135
- view command 3-62, 5-138

## W

- who should use this book xxi
- window command 5-138
- window descriptor file
  - rwppc.wdf 3-123
- window layout 3-135
- window list 3-138
- Window menu 3-41, 3-43
- windows
  - ASCII Memory 3-107, 3-109
  - Assembly Debug 2-13, 3-48, 3-53, 3-55, 3-75
  - Breakpoint Select 3-76
  - Breakpoints 2-5, 3-74
  - Cache 3-112
  - Calculator 3-142
  - Callers 2-7, 3-60
  - Change Array Variable 2-11
  - Command File 3-132
  - Compound Trigger/Trace 4-11
  - Files 3-62
  - Functions 2-6, 3-62, 3-64

Globals 3-79  
Help 3-143  
Inspect 3-81  
Locals 2-9, 3-78  
Log Comment 3-140  
Logging State 3-140  
Main 3-40, 3-48  
Memory Coherency 3-104  
OS Open 3-67  
Output 3-136  
Processor Reset 3-134  
Programs 3-59  
sample user-defined 3-121  
Source 3-48, 3-52  
TLB 4-14  
Trigger/Trace 4-7  
Variable Configuration 2-9, 3-79, 3-81, 3-83  
window layout 3-135  
Window List 3-138  
write command 5-140



