IBM

RISCWatch Debugger Installation Guide

13H6984

Eighth edition (September 1996)

This edition of *IBM RISCWatch Debugger Installation Guide* applies to IBM RISCWatch Debugger Version 3.3 and to all subsequent versions of the debugger until otherwise indicated in new versions or technical newsletters.

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.

Address comments about this publication to:

IBM Corporation
Department 0H83A
P.O. Box 12195
Research Triangle Park, NC 27709

Printed in the United States of America.

4 3 2 1

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
Micro Channel
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

Installing the RISCWatch Debugger

Installation of RISCWatch requires both hardware and software to be installed on the host platform. System requirements and installation procedures vary, depending on the host platform and whether or not the target application supports the JTAG port. The following sections will guide you through the steps necessary to install these items.

Additionally, for non-JTAG targets, you MUST also complete the target platform's installation instructions for debugging before continuing. They can be found in the PowerPC 400Series evaluation board kit user documentation (ROM Monitor target) or the OS Open user documentation (OS Open target).

These instructions describe specific host configuration steps and other setup (editing /etc/services files) required by RISCWatch for host/target communications. Refer to the Configuration chapter of the *PowerPC 403 Evaluation Board Kit User's Manual* or the Installation chapter of the *OS Open User's Guide*. Both documents are listed in "Related IBM Publications" on page xxvi of the *RISCWatch Debugger User's Guide*.

BEFORE BEGINNING THE HARDWARE INSTALLATION, YOU MUST REFER TO THE ENCLOSED IBM SAFETY BOOKLET (SD21-0030-02).

Hardware Installation for JTAG Targets

A JTAG target is defined as a board using a PowerPC processor, for example, a PowerPC 400Series evaluation board, connected via the JTAG port of the controller to the host platform running the RISCWatch Debugger.

RISCWatch Micro Channel Adapter (RS/6000 Only)

The following hardware is required before you can install the RISCWatch Micro Channel™ adapter on the RISC System/6000 platform:

RISC System/6000 PowerStation with a graphics display

One RISCWatch Micro Channel adapter card for RISC System/6000

One RISCWatch Processor Interface Assembly (buffer card) for RISC System/6000

One 34-pin connector cable

What follows is a step by step procedure for installing the RISCWatch Micro Channel adapter card for RISC System/6000. Be sure to follow the exact sequence of steps and follow the directions for each step exactly.

1. Logon as **root** or use the **su** command to assume root user privileges on the RISC System/6000 that is to contain the adapter card.
2. Issue the shutdown command. This will terminate all running processes and force the logoff of all logged-on users.
3. When the "...Halt completed..." message appears, turn the power switch to the off (O) position.
4. Insert the machine's key and turn it to the Service position.
5. Perform the actions necessary to remove the cover.
6. Find an empty Micro Channel slot and remove its protective silver slot tab.
7. Install the RISCWatch adapter card in this slot and tighten the tab fastener.
8. Plug one end of the 34-pin cable into the RISCWatch adapter card.
   Note: The connector is keyed so that it can only be plugged in one way. Do **not** force the cable or the adapter card pins may be damaged.
9. Plug the other end of the 34-pin cable into the RISCWatch buffer card, again noting the orientation of the keyed connector.
10. Plug the 16-pin cable of the RISCWatch buffer card into the JTAG port connector of the JTAG target.
11. Replace the RISC System/6000 cover.
12. Turn the machine's key to the Normal position and turn the power switch to the on (1) position.
13. Wait for the login prompt to appear after the RISC System/6000 has finished its boot up procedure.

RISCWatch Parallel Port Adapter

The following hardware is required before you can install the RISCWatch parallel port adapter on the host platform:

One RISCWatch parallel port adapter

One RISCWatch parallel port adapter cable

- One RISCWatch Parallel Port Adapter power supply for U.S.A and Canada use only.
- One RISCWatch Parallel Port Adapter power supply jack for countries other than the U.S.A or Canada.
  YOU MUST PROVIDE A 240VA (OR LESS) POWER SUPPLY THAT IS AGENCY-APPROVED IN THE COUNTRY YOU ARE IN.

Note: The power requirements for the adapter are 5V, 300mA, regulated, 5V on the inner conductor, GND on the outer conductor.

1. Plug one end of the adapter cable into the adapter and plug the other end into the parallel port of the host platform.
2. Plug the 16-pin cable of the RISCWatch parallel port adapter into the JTAG port connector of the JTAG target.
3. If you are in the U.S.A. or Canada, plug the connector of the enclosed power supply into the parallel port adapter box and plug the power supply into the wall outlet.
4. If you are not in the U.S.A or Canada, attach the enclosed power supply jack to your own agency-approved, 240VA (or less), 5V, 300ma, regulated power supply. The inner conductor of the jack has to be connected to 5V and the outer conductor has to be connected to ground. Plug the jack into the parallel port adapter box and plug the power supply into the wall outlet.

PC Specifics

The following hardware is required before you can install RISCWatch on a PC:

IBM or compatible PC

Minimum required: x486 DX2 50/66 MHz with 8 MB of RAM

VGA/SVGA Display

Minimum required: VGA 640x480
Recommended: SVGA 1024x768
Also supports: SVGA 800x600, SVGA 1280x1024

RISCWatch defaults to the parallel port on the PC motherboard. The port is usually mapped at address 0x03BC. If the parallel port to which the hardware is attached is not mapped at this address, refer to "PC Specifics" on page 18 for information about changing the default setting.

RS/6000 Specifics

The following hardware is required before you can install RISCWatch for the RISC System/6000 :

RISC System/6000 PowerStation with a graphics display

RISCWatch only supports the parallel port on the motherboard of the RISC System/6000. It does not support parallel port Micro Channel adapter cards.

Sun Specifics

The following hardware is required before you can install RISCWatch for Sun:

Sun SPARCstation 5, 10, or 20

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 probe's JTAG port. The probe has a dedicated JTAG controller chip to drive the JTAG signals in hardware as opposed to a slower, emulated approach in software. For additional information, see "JTAG Ethernet Targets and the RISCWatch Processor Probe" on page 3-10 of the *RISCWatch Debugger User's Guide*.

The following hardware is required before you can install the RISCWatch processor probe:

One RISCWatch processor probe

One RISCWatch processor probe power supply and power cord

One RISCWatch processor probe transfer adapter and JTAG cable. The transfer adapter is the small circuit board with two connectors.

1. Connect the 20-pin connector on the transfer adapter to the front of the RISCWatch processor probe. If the transfer adapter is supplied with a long 20-pin ribbon cable, attach the 20-pin connector on the long ribbon cable to the front of the RISCWatch processor probe instead.
2. Connect the 16-pin cable attached to the transfer adapter to the JTAG port connector of the JTAG target. This cable may be long or short based on the transfer adapter type. The connector is keyed. Failure to align the key properly may damage the JTAG target.
3. Connect the power cord to the power supply and to a socket outlet. USE ONLY THE SUPPLIED POWER CORD.
4. Connect the 5v power cord to the back of the RISCWatch processor probe. The power light on the front of the RISCWatch processor probe will be illuminated. The RISCWatch processor probe does not have an on/off switch.

   Note: The combination of a RISCWatch processor probe connected to a JTAG target is referred to as a JTAG Ethernet target.

Connecting the RISCWatch Processor Probe to an Existing Ethernet Network

1. Obtain a TCP/IP address and gateway address for the RISCWatch processor probe from your system administrator.
2. Follow the instructions under "Changing the TCP/IP Address of the RISCWatch Processor Probe" on page 11 and use the "lan" command to set the TCP/IP address and gateway address on the processor probe.

3. Request a BNC type connection to your lan. Connect the BNC connection on your lan (IEEE 802.3 Type 10Base2 ThinLAN) to the round BNC receptacle marked LAN on the RISCWatch processor probe.

Note: This installation guide assumes that a 10Base2 type connection will be made to the RISCWatch processor probe. However, if a 10BaseT connection is required, connect your 10BaseT lan drop to the rectangular 10BaseT receptacle marked LAN on the RISCWatch processor probe. Set switch number 5 on the RISCWatch processor probe to the open position and cycle power on the RISCWatch processor probe.
The default settings for the RISCWatch processor probe configuration switches are switch 1 open and the other seven closed. Configuration switch functions are indicated on the underside of the processor probe frame.

Establishing an Ethernet Network for the RISCWatch Processor Probe

Establishing an Ethernet connection between a host and the RISCWatch processor probe can be done in several ways, depending on the type of connection supported by the Ethernet adapter in your host. A 10Base2 connection between a host machine and the RISCWatch processor probe requires at a minimum two pairs of BNC T-connectors and 50-ohm terminators, plus a short length of 10Base2 ThinLAN cable, as shown in Figure 1 below:

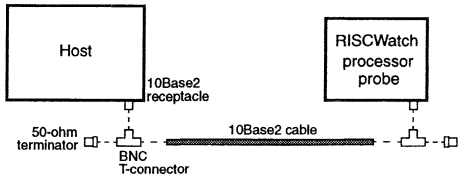


Figure 1. 10Base2 Ethernet Connection

For 10BaseT Ethernet, the connection can be made in two ways. If the connection is to be used exclusively between the host and the EVB, a crossover cable can be used to connect the two nodes. Otherwise, a 10BaseT hub must be used to connect the nodes together.

Figure 2 shows the connections and signal assignments in a crossover cable:

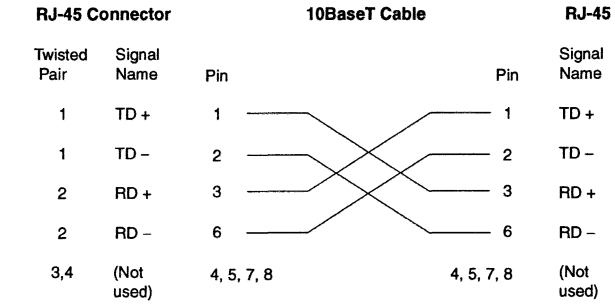


Figure 2. Wiring in a 10BaseT Crossover Cable

Figure 1 shows a point-to-point Ethernet connection using a 10BaseT crossover cable:

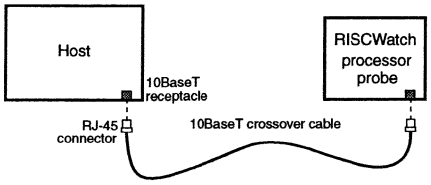


Figure 3. 10BaseT Crossover Connection

Figure 1 shows a 10BaseT Ethernet connection using a hub:

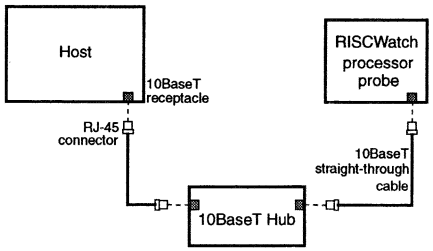


Figure 4. 10BaseT Hub Connection

PC Specifics:

Establishing an Ethernet network requires additional hardware. Because most PC models do not come with an Ethernet connection, an ISA bus Ethernet adapter with the required BNC 10Base2 or RJ45 10BaseT connection has been provided. To install the adapter hardware, refer to the installation instructions found with the adapter.

To correctly install the supplied ISA bus Ethernet adapter under IBM's TCP/IP for DOS, the following steps must be executed before running the TCP/IP for DOS's "custom" program. These instructions assume TCP/IP for DOS has already been installed.

1. Place the ISA Ethernet Adapter Feature Diskette into the A: drive.
2. Copy the device driver, KTC2000.DOS, to the appropriate TCP/IP directory:

   copy A:\NDIS\KTC2000.DOS C:\TCPDOS\BIN\KTC2000.DOS

   If this fails, the driver file may be located in a different directory. Try the following command:

   copy A:\NDIS\DOS\KTC2000.DOS C:\TCPDOS\BIN\KTC2000.DOS

   If this command fails, locate the KTC2000.DOS file on the diskette and copy it to the \TCPDOS\BIN directory.
3. Create the file C:\TCPDOS\ETC\PROTOCOL.INI, containing the lines:

   [PROTMAN]
   DriverName=PROTMAN$

   [KTC2000]
   ; Kingston EtheRx LC Adapter
   ; KTC2000.DOS
   DriverName=KTC2000$
4. Edit the C:\CONFIG.SYS file and add the following line after PROTMAN.DOS:

   DEVICE = C:\TCPDOS\BIN\KTC2000.DOS

The RISCWatch processor probe supports connection via Standard Ethernet, either 10Base2 or 10BaseT. See "Establishing an Ethernet Network for the RISCWatch Processor Probe" on page 5 for further details.

Because TCP/IP packages for PCs vary, users should consult their TCP/IP documentation for information regarding the management and configuration of an Ethernet network interface.

Establishment of an Ethernet interface will require a host TCP/IP address. To maintain consistency with this document, a TCP/IP address of 7.1.1.4 is suggested.

Once the required hardware has been installed on the host PC, IBM TCP/IP for DOS users can establish an Ethernet interface by executing the following steps:

1. Type "custom" from the DOS prompt
2. Select Ok or press Enter
3. Go to Configure at the top of the menu and press Enter
4. Select NDIS Interfaces
5. Select ND0 (or any available NDx) Interface
6. Set the IP address field to the IP address of the PC host: 7.1.1.4 is suggested to maintain consistency with this document
7. Set the Subnet mask to 255.255.240.0
8. Select the appropriate type of Ethernet adapter installed for the Bound adapter field (select the down arrow for a list of adapter types). If using the supplied ISA bus Ethernet adapter, select KTC2000.
9. Select Enable under Options
10. Select Advanced Functions from the bottom of the screen and ensure only arp is selected under Options
11. Select OK from the NDIS Interfaces menu
12. Select Exit - Save Changes from the Configure menu

The interface is activated by starting TCP/IP via the tcpstart command from the DOS prompt. If problems occur, verify the custom settings for the Ethernet interface, re-boot the system, and re-try the tcpstart command.

RS/6000 Specifics:

Establishing an Ethernet network requires additional hardware.

The RISCWatch processor probe supports connection via Standard Ethernet, either 10Base2 or 10BaseT. See "Establishing an Ethernet Network for the RISCWatch Processor Probe" on page 5 for further details.

Other hardware required will depend on the type of Ethernet adapter you have on your RS/6000.

AIX Communications Concepts and Procedures (GC23-2203, two volumes) has additional information about the management and configuration of a TCP/IP network, including specifics as to how to configure an Ethernet network interface.

Some of the basic steps are outlined below.

1. The host must be equipped to participate in a 10Base2 or 10BaseT Ethernet network.

   This may involve the installation of any or all of the following hardware: an Ethernet adapter card for the specific RS/6000 model, a 10Base2 network transceiver, a BNC "T" type connector, and a terminating resistor. Consult the documentation included with the hardware for installation instructions. Most RS/6000 models come with Ethernet adapters already installed. They are labeled ET in the back of the RS/6000 system unit.

2. Assuming the host system is equipped with the appropriate Ethernet adapter, the Ethernet interface must be configured properly. To do this:

   a. Log in as **root** or superuser (**su**)

   b. Enter **smit**

   c. Select Communication Applications and Services

   d. Select TCP/IP

   e. Select Further Configuration

   f. Select Network Interfaces

   g. Select Network Interface Selection

   h. Select Add a Network Interface

   i. Select Add a Standard Ethernet Network Interface.
   Choose "Standard Ethernet" as opposed to "IEEE 802.3 Ethernet".

   Note: If you receive an error message stating that there is "No available adapter", go directly to step 3. Skip the remaining items in step 2.

   j. Select en0

   k. Set the INTERNET ADDRESS field to the host TCP/IP address. An acceptable value would be 7.1.1.4

   l. Set the Network MASK field to 255.255.240.0

   m. Insure that ACTIVATE is yes

   n. Insure that the Use Address Resolution Protocol is yes

   o. Leave the BROADCAST ADDRESS blank

   p. Select Do or press Enter

   Upon successful completion, a properly configured Ethernet interface has been added. The Ethernet set-up is complete and step 3 need not be performed.

3. Perform this step only if you received the "No available adapter" error message when trying to Add a Standard Ethernet Network Interface in step 2.

   This message indicates that either the Ethernet adapter is missing (or possibly misplugged) or the Ethernet Network Interface already exists. To determine whether the interface already exists:

   a. Return to the Network Interface Selection screen in **smit**

   b. Select Change/Show Characteristics of a Network Interface

   - If en0 is not listed, insure that the RS/6000 host does have an Ethernet adapter and, if possible, verify that it is plugged correctly. If the adapter was misplugged, repeat step 2 to add the Ethernet Network Interface.
   - If en0 is listed, then the Ethernet Network Interface already exists. Select en0 and note the TCP/IP address listed for the INTERNET ADDRESS field. This value is the host's Ethernet TCP/IP address and will be needed later. If no TCP/IP address is listed, choose one. The TCP/IP address 7.1.1.4 is recommended to maintain consistency with the menus and examples in this document. The Ethernet set-up is complete.

Sun Specifics

Establishing an Ethernet network requires additional hardware.

The RISCWatch processor probe supports connection via Standard Ethernet, either 10Base2 or 10BaseT. See "Establishing an Ethernet Network for the RISCWatch Processor Probe" on page 5 for further details.

Other hardware that may be required is an AUI (or thick Ethernet) adapter cable (or an AUI/Audio Adapter cable depending on your SPARCstation model and options - both are available from Sun) and an Ethernet/IEEE 802.3 10Base2 network transceiver. Consult the documentation included with the hardware for installation instructions.

The **ifconfig** command can be used to establish the network. Users should consult their network administrator and Sun documentation for additional information. A host TCP/IP address of 7.1.1.4 is suggested to maintain consistency with this document.

Changing the TCP/IP Address of the RISCWatch Processor Probe

The RISCWatch processor probe ships with a TCP/IP address of 7.1.1.100 and a gateway address of 0.0.0.0. To change these addresses to be valid addresses on your network, a serial port connection must be made from the host to the processor probe.

Once the addresses are made valid for your network, the "telnet" utility on your host and the "lan" command on the RISCWatch processor probe can be used to change the addresses from then on. The "lan" command is described in items 10-14 under the subsection PC Specifics, items 13-17 under the subsection RS/6000 Specifics and items 5-9 under the subsection Sun Specifics.

Please follow the instructions that apply to the host that you are using when changing the TCP/IP and gateway addresses for the first time.

PC Specifics

Most PCs include two serial ports to support communications via asynchronous data transfer. These ports are sometimes referred to as communication or COM ports. These ports are usually accessed from the back of the system unit.

This document refers to them as serial ports S1 and S2. Consult your PC literature to determine how many serial ports are available on your unit and where they are located.

1. Connect the 9-pin female connector of the supplied serial cable to S1 or S2 on the PC.
2. Connect the 9-pin male connector of the supplied cable to the connector labeled RS232 on the RISCWatch processor probe.
3. Start Microsoft Windows if it is not active.
4. Select Accessories from the Windows Program Manager.
5. Double-click on the Terminal icon to start the terminal emulator program.
6. Select Settings->Communications.
7. Select COM1 if using S1 or COM2 if using S2.
8. Select Baud Rate 9600, Data Bits 8, Stop Bits 1, Parity None and Flow Control Xon/Xoff
9. Select OK.
10. Press enter. The RISCWatch processor probe will respond with a status prompt consisting of a letter followed by the ">" sign.
11. Enter "lan" to display the current lan settings.
12. To change the TCP/IP address, enter "lan -i 'dotted tcp/ip address'". For example, enter "lan -i 7.1.1.101".

13. To change the gateway, enter "lan -g 'dotted gateway address'". For example, enter "lan -g 0.0.0.0".
14. To change the Processor Probe's port number, enter "lan -p 'port number'". For example, enter "lan -p 6470".
15. Select File->Exit to exit the terminal session.
16. If asked to save changes to terminal settings, select No.
17. Cycle power on the RISCWatch processor probe for the changes to take effect.

RS/6000 Specifics

The RS/6000 includes two serial ports to support communications via asynchronous data transfer. These ports are labeled S1 and S2 on the back of the RS/6000's system unit.

1. Connect the supplied 25-pin-to-9-pin adapter to the 9-pin female connector of the supplied serial cable.
2. Connect the 25-pin connector to port S1 or S2 on the RS/6000.
3. Connect the 9-pin male connector of the supplied serial cable to the connector labeled RS232 on the RISCWatch processor probe.
4. Log in as **root** or superuser (**su**)

Proper set-up involves the configuration of tty devices for either S1 or S2. The following steps should be taken to insure proper S1 or S2 configuration:

5. Determine whether the tty0 or tty1 devices already exist. tty0 must exist if using port S1 and tty1 must exist if using S2.
   a. Enter **smit**
   b. Select Devices
   c. Select TTY
   d. Select List All Defined TTYs
   e. Perform step 6 if tty0 or tty1 is not listed. To properly configure a defined tty device, perform step 7 for systems running AIX 3, or perform step 8 for systems running AIX 4 or higher.
6. To add a tty device:
   a. Select Done or PF3 to exit the List All Defined TTYs screen
   b. Return to the TTY screen
   c. Select Add a TTY
   d. Select tty RS232 Asynchronous Terminal
   e. Select sa0 - Serial Port 1 when adding tty0
      sa1 - Serial Port 2 when adding tty1

f. Select s1 for the port number when adding tty0
   s2 for the port number when adding tty1
g. Insure that the BAUD rate is 9600
h. Insure that the PARITY is none
i. Insure that the BITS per character is 8
j. Insure that the Number of STOP BITS is 1
k. Insure that Enable LOGIN is disabled
l. The default settings for all the other fields are satisfactory.
m. Select Do or press Enter

Upon successful completion, a properly configured tty device is created and thus, steps 7 and 8 can be skipped for the particular tty (tty0 or tty1) added. Go directly to step 9.

7. To properly configure a previously defined tty device:
   - For systems running AIX 3 :

   a. Select Done or PF3 to exit the List All Defined TTYs screen
   b. Return to the TTY screen
   c. Select Change / Show Characteristics of a TTY
   d. Select tty# (where # = 0 or 1)
   e. Select Change / Show TTY Program
   f. Insure that the following fields are set to the indicated values:

TTY	tty# (#=0 for tty0, 1 for tty1)
TTY type	tty
TTY interface	RS232
Description	Asynchronous Terminal
Status	Available
Location	00-00-S*-00 (*=1 for tty0, 2 for tty1)
Parent Adapter	sa# (#=0 for tty0, 1 for tty1)
Port Number	s* (*=1 for tty0, 2 for tty1)
Terminal Type	dumb
Enable LOGIN	disable

   The other fields can remain at their default values.

   g. Select Do or press Enter
   h. Upon successful completion, select Done or press PF3 to return to the TTY screen
   i. Select Change / Show Characteristics of a TTY

   j. Select tty# (where # = 0 or 1)
   k. Select Change/Show HARDWARE TTY Characteristics
   l. Insure that the BAUD rate is 9600
   m. Insure that the PARITY is none
   n. Insure that the BITS per character is 8
   o. Insure that the Number of STOP BITS is 1
   p. Select Do or press Enter

Upon successful completion, the tty device is properly configured. Go directly to step 9.

8. To properly configure a previously defined tty device:
   - For systems running AIX 4 or later :

   a. Select Done or PF3 to exit the List All Defined TTYs screen
   b. Return to the TTY screen
   c. Select Change / Show Characteristics of a TTY
   d. Select tty# (where # = 0 or 1)
   e. Insure that the following fields are set to the indicated values:

TTY	tty# (#=0 for tty0, 1 for tty1)
TTY type	tty
TTY interface	RS232
Description	Asynchronous Terminal
Status	Available
Location	00-00-S*-00 (*=1 for tty0, 2 for tty1)
Parent Adapter	sa# (#=0 for tty0, 1 for tty1)
Port Number	s* (*=1 for tty0, 2 for tty1)
Terminal Type	dumb
Enable LOGIN	disable

   f. Insure that the BAUD rate is 9600 for tty0
   g. Insure that the PARITY is none
   h. Insure that the BITS per character is 8
   i. Insure that the Number of STOP BITS is 1
      The other fields can remain at their default values.
   j. Select Do or press Enter

Upon successful completion, the tty device is properly configured.

9. Edit the file /etc/uucp/Devices and add one of the following lines:

"Direct tty0 - 9600 direct" if using S1
"Direct tty1 - 9600 direct" if using S2

10. File the changes.
11. Exit from **root**.
12. From the AIX command line, enter:
    "/usr/bin/cu -m -l tty0" if using S1
    "/usr/bin/cu -m -l tty1" if using S2
13. Press Enter one more time. The RISCWatch processor probe will respond with a status prompt consisting of a letter followed by the ">" sign.
14. Enter "lan" to display the current lan settings.
15. To change the TCP/IP address, enter "lan -i 'dotted tcp/ip address'". For example, enter "lan -i 7.1.1.101".
16. To change the gateway, enter "lan -g 'dotted gateway address'". For example, enter "lan -g 0.0.0.0".
17. To change the Processor Probe's port number, enter "lan -p 'port number'". For example, enter "lan -p 6470".
18. Enter "~." to return to the host.
19. Press Enter to quit the session.
20. Cycle power on the RISCWatch processor probe for the changes to take effect.

Sun Specifics

The Sun SPARCstation includes two serial ports to support communications via asynchronous data transfer. These ports are labeled Serial A and Serial B on the back of the Sun's system unit. Some SPARCstation models multiplex these two ports into one physical port labeled A/B. Use A if it is available because use of the B port requires a special demultiplexing cable from Sun.

This section refers to these ports as S1 and S2, respectively.

1. Connect the supplied 25-pin-to-9-pin adapter to the 9-pin female connector of the supplied serial cable.
2. Connect the 25-pin connector to port S1 or S2 on the Sun.
3. Connect the 9-pin male connector of the supplied serial cable to the connector labeled RS232 on the RISCWatch processor probe.
4. From the command line, enter:
    "/usr/bin/tip -9600 /dev/ttya" if using S1
    "/usr/bin/tip -9600 /dev/ttyb" if using S2

5. Press Enter one more time. The RISCWatch processor probe will respond with a status prompt consisting of a letter followed by the ">" sign.
6. Enter "lan" to display the current lan settings.
7. To change the TCP/IP address, enter "lan -i 'dotted tcp/ip address'". For example, enter "lan -i 7.1.1.101".
8. To change the gateway, enter "lan -g 'dotted gateway address'". For example, enter "lan -g 0.0.0.0".
9. To change the Processor Probe's port number, enter "lan -p 'port number'". For example, enter "lan -p 6470".
10. Enter "~." to quit the session.
11. Cycle power on the RISCWatch processor probe for the changes to take effect.

Verifying Your Network

From your host, enter "ping 'dotted tcp/ip address'". For example, enter "ping 7.1.1.101".

Hardware Installation For Non-JTAG Targets

A non-JTAG target is defined as a board using a PowerPC processor, for example, a PowerPC 400Series evaluation board, connected via an Ethernet link to the host platform running the RISCWatch Debugger. A non-JTAG target must be running OS Open or the IBM ROM Monitor for PowerPC debug software to communicate with RISCWatch.

Ethernet Link

The following hardware is required before you can install the RISCWatch Ethernet link to the target:

If the host platform is already on an Ethernet network, request a new Ethernet drop and TCP/IP address for the target from your system administrator. If the host platform is not on an Ethernet network, the following is recommended to establish a 10Base2 Ethernet link:

Two BNC T-section (male to female-female)

Two BNC male 50 ohm terminating resistors

One BNC coaxial cable

One female LAN connector on the host platform

One female LAN connector on the target

1. Attach a BNC terminator to each BNC T-section.
2. Connect the two BNC T-sections with the BNC coaxial cable.
3. Attach one BNC T-section to the host platform BNC LAN connector and the other to the target BNC LAN connector.

PC Specifics

The following hardware is required before you can install RISCWatch on a PC:

IBM or compatible PC

Minimum required: x486 DX2 50/66 MHz with 8 MB of RAM

VGA/SVGA Display

Minimum required: VGA 640x480
Recommended: SVGA 1024x768
Also supports: SVGA 800x600, SVGA 1280x1024
Ethernet adapter
Ethernet/IEEE 802.3 transceiver unit (MAU) if a BNC connector is not available on the Ethernet adapter

RS/6000 Specifics

The following hardware is required before you can install RISCWatch for the RISC System/6000 :

RISC System/6000 PowerStation with a graphics display

Ethernet adapter, if not already available

Ethernet/IEEE 802.3 transceiver unit (MAU) if a BNC connector is not available

Sun Specifics

The following hardware is required before you can install RISCWatch for Sun:

Sun SPARCstation 5, 10, or 20

Attachment Unit Interface Adapter (AUI) Cable, or Attachment Unit Interface (AUI)/Audio Adapter Cable, if not already available

Ethernet/IEEE 802.3 transceiver unit (MAU) if a BNC connector is not available

Software Installation

PC Specifics

The following items are required before you can install RISCWatch software on a PC:

Microsoft Windows 3.1

RISCWatch Installation Diskette(s)

One 3.5" diskette drive

Three megabytes of hard disk space

For JTAG Ethernet and non-JTAG targets, a TCP/IP for Windows package compliant with the Microsoft Windows Socket API definition

To install RISCWatch, perform the following :

1. Place the RISCWatch Installation diskette in the proper drive
2. Start Microsoft Windows if it is not active
3. Select Run... from the File pull-down of Program Manager
4. Type "a:install"(or "b:install" if applicable) then press Enter
5. When the Welcome window appears, click on Continue
6. When the Custom Installation window appears, use the Set Location button if you wish to change the directory where the program files will be installed (C:\RW)
7. Click on Install
8. Follow any instructions that may prompt you to insert additional installation diskettes.

   Note: You will be prompted to insert a "Processor Probe Driver" diskette. For JTAG parallel port targets, this diskette is not necessary, so simply "Cancel" this prompt.
9. For JTAG Ethernet and non-JTAG targets, the following additional steps are required to establish communications between the host and target. Named communications ports must be established for TCP/IP socket communications. Most often, this involves an update to the **services** file.

   a. Most TCP/IP packages place the **services** file under one of the package's subdirectories. Consult your TCP/IP documentation or contact your system administrator if this file cannot be found. The following lines must be added to the file:

```
osopen-dbg   20044/tcp    # For ROM Monitor targets
osopen-dbg   20044/udp    # For OS Open targets
jtag_eth     6470/tcp     # For JTAG Ethernet targets
             Note: use underscore, not hyphen
```

b. For the update to take effect, TCP/IP may need to be restarted. This may require a reboot of the system and/or a restart of the TCP/IP package.

Once the installation is completed, a RISCWatch group will be created along with some program items. The RISCWatch "README" file will then be displayed. Please view the entire file for the latest changes to the program and its operation.

If you are using the RISCWatch parallel port adapter, the default RISCWatch program item assumes that the parallel port to which the hardware is attached is mapped at address 0x03BC. If the parallel port is mapped at address 0x0378, use Windows Program Manager to modify the properties of the RISCWatch program item by adding the switch '-par2' on the command line. If the parallel port is mapped at address 0x0278, use '-par3' instead.

If you are using the RISCWatch processor probe, you must modify the properties of the RISCWatch program item to use the "-proc" switch to select the processor. The "-proc" switch is only needed when you are using the RISCWatch processor probe for the first time or when you want to switch from one processor to another. For the current list of valid processor names, see "Invoking the Debugger" on page 3-7 in the *RISCWatch Debugger User's Guide*.

Modify the file **rwppc.env** to make the appropriate changes to any of the RISCWatch environment variables. Select the proper target type, as described in "Environment Resources" on page 3-5 in the *RISCWatch Debugger User's Guide*. Users that have a working directory other than the install directory have their own copy of the **rwppc.env** file. Such users should backup their copy, make a fresh one from the install directory, and then merge their changes from the old one to the new one.

Note that the RISCWatch environment variable TARGET_TYPE, when set to "jtag", informs RISCWatch that you are using the RISCWatch parallel port adapter. When TARGET_TYPE is set to "jtag_eth", it informs RISCWatch that you are using the RISCWatch processor probe. For the current list of TARGET_TYPE settings, refer to "Environment Resources" on page 3-5 in the *RISCWatch Debugger User's Guide*.

This completes the software installation for RISCWatch.

RS/6000 Specifics

The following items are required before you can install RISCWatch software on a RISC System/6000 :

AIX Version 3.2.5 or later

AIX/Windows with X11R5 and Motif 1.2

RISCWatch Installation Diskette(s)

One 3.5" diskette drive

Three megabytes of hard disk space

Note: During program installation, an additional three megabytes of hard disk space are needed temporarily to hold both the RISCWatch program files and the installation file.

What follows is a step by step procedure for installing the RISCWatch software. Be sure to follow the exact sequence of steps and follow the directions for each step exactly.

1. Logon as **root** to the RISC System/6000 or use the **su** command to gain root user privileges.
2. Choose an already existing directory or create a new directory. For example:

   mkdir /usr/rwppc

   which will contain the RISCWatch program and its associated files.
3. Issue the cd command to make it the present working directory. For example:

   cd /usr/rwppc
4. Insert the RISCWatch Installation Diskette into the diskette drive
5. Issue the following command to extract the files from the RISC System/6000 RISCWatch Installation Diskette(s) and place them in the chosen directory:

   tar -xvf /dev/rfd0
6. For this next step, you will need to run the installation program. The installation program is highly automated and will automatically update any RISCWatch drivers that are already on your system. The installation program will detect currently installed MCA cards, MCA drivers and parallel port drivers and update them accordingly. The one exception that the installation program cannot detect is when the parallel port is to be configured for use by RISCWatch. Since the parallel port is generally used by a printer, you must instruct the installation program to use it for RISCWatch. Once it is done the first time, a driver will then exist for the installation program to detect in future installations. So if you are installing a parallel port version of RISCWatch for the first time you must run the installation program as 'rw_inst -p'; otherwise run it as 'rw_inst'.
7. The install program will prompt you for a driver diskette. Insert the Processor Probe Driver diskette for JTAG Ethernet targets, or the Device Driver diskette for parallel port or MCA JTAG targets. No diskette is required for OS Open or ROM Monitor targets.

8. Modify the file **rwppc.env** to make the appropriate changes to any of the RISCWatch environment variables. Select the proper target type, as described in "Environment Resources" in the *RISCWatch Debugger User's Guide*. Users that have a working directory other than the install directory have their own copy of the **rwppc.env** file. Such users should backup their copy, make a fresh one from the install directory, and then merge their changes from the old one to the new one.

   Note that the RISCWatch environment variable TARGET_TYPE, when set to "jtag", informs RISCWatch that you are using the RISCWatch Micro Channel or parallel port adapter. When TARGET_TYPE is set to "jtag_eth", it informs RISCWatch that you are using the RISCWatch processor probe. For the current list of TARGET_TYPE settings, refer to "Environment Resources" on page 3-5 in the *RISCWatch Debugger User's Guide*.

9. Add the following line to every user's .profile that will be running RISCWatch:

   export UIDPATH=./%U:/usr/rwppc/%U (for Korn shell)
   setenv UIDPATH ./%U:/usr/rwppc/%U (for C shell)

   You must specify the *%U* at the end of each path in the UIDPATH line.

   Be sure to change the directory in the above lines if you did not install RISCWatch in the /usr/rwppc directory. If you wish to use RISCWatch without logging off and logging on your machine again, type in the above line at the AIX prompt to set this environment variable immediately.

10. For proper device configuration reporting, ensure that one of the following export lines exists in each user's **.profile** :

    For AIX 3.2 :

    export LANG=En_US (for Korn shell)
    setenv LANG En_US (for C shell)

    For AIX 4.1 :

    export LANG=en_US (for Korn shell)
    setenv LANG en_US (for C shell)

11. For JTAG Ethernet and non-JTAG targets, these additional steps are required to establish communications between the host and target:

    a. To modify the **/etc/services** file, the user must be logged in as **root** or superuser (**su**). The following lines must be added to the file:

    osopen-dbg 20044/tcp
    osopen-dbg 20044/udp
    jtag_eth 6470/tcp # Note: Underscore used, not hyphen

    b. The AIX **refresh -s inetd** command must then be run to synchronize the object data manager (ODM) database and to update the **inetd** daemon.

This completes the software installation for RISCWatch.

Notes:
It may be necessary to add the chosen directory to the PATH environment variable if it has not already been added. Furthermore, it may be necessary to change ownership of this directory as well as all of its files if many people will need access to the RISCWatch program.
If you are using the RISCWatch processor probe, refer to "Invoking the Debugger" on page 3-7 in the *RISCWatch Debugger User's Guide* before you start RISCWatch.

Sun Specifics

The following items are required before you can install RISCWatch software on a Sun SPARCstation 5, 10, or 20.

A Sun SPARCstation 5, 10 or 20 workstation

One 3.5" diskette drive

SunOS 4.1.3 (or higher) or Solaris (higher)

OpenWindows 3.0 (SunOS 4.1.3) or 3.3 (Solaris)

RISCWatch Installation Diskette(s)

Three megabytes of hard disk space

Note: During program installation, an additional three megabytes of hard disk space are needed temporarily to hold both the RISCWatch program files and the installation file.

SunOS Device Driver Installation

For SunOS, the operating system kernel must be recompiled without a device attached to the parallel port for the RISCWatch parallel port device driver to dynamically install correctly. The kernel must be recompiled BEFORE the RISCWatch installation program, "rw_inst" is run. The following explains how to reconfigure the SunOS kernel.

Configuring the SunOS Kernel for RISCWatch

1. Log in as **root**.
2. Change directory to /usr/kvm/sys/sun4m/conf.
3. Copy your existing kernel configuration file, eg. GENERIC, to a new name, eg. SUN4RW.

   cp GENERIC SUN4RW; chmod +w SUN4RW

4. Edit SUN4RW to comment out the bidirectional device driver that comes with your kernel. # in the first column is used to indicate a comment. Any devices that use this device driver, such as a printer, will not be useable.

   #device-driver bpp # bpp support commented out

5. Verify that the following line exists in SUN4RW to dynamically load device drivers:

   options VDDRV # loadable modules

6. Verify that enough space exists to recompile the kernel. Approximately 2.5MB of disk space is needed in /usr to recompile the GENERIC configuration kernel.

7. Run config:

   /etc/config SUN4RW

   (The directory ../SUN4RW will be made if it doesn't exist and "make depend" will be done unless you specify a "-n" flag)

8. Make the new system:

   cd ../SUN4RW
   make

9. Typically the running kernel should be "/vmunix" because programs like 'ps' and 'w' expect "/vmunix" to be the running kernel. Save the original kernel, install the new one in /vmunix, and try it out:

   mv /vmunix /vmunix.old
   cp vmunix /vmunix
   /etc/halt
   boot vmunix

10. If the system does not appear to work, boot and restore the original kernel, then fix the new kernel:

    /etc/halt
    b vmunix.old -s
    mv /vmunix.old /vmunix
    boot vmunix

Software Installation Instructions for SunOS and Solaris

1. Logon as **root** or use the **su** command to gain root access.
2. Open at least two windows for this process.
3. Choose an already existing directory or create a new directory which will contain the RISCWatch program and its associated files. For example, to create a new directory rwppc within /usr:

   mkdir /usr/rwppc

4. Issue the **cd** command in both windows to make /usr/rwppc the current working directory. For example:

   cd /usr/rwppc

5. Insert RISCWatch Installation Diskette 1 into the diskette drive.

Instructions for SunOS

6. From the second window run the command:

   cpio -ivB rwppc.SunOS4.tar.Z rw_inst </dev/rfd0

   where '/dev/rfd0' is the name of your diskette device.

7. When the system prompts you for a new volume, move to the first window and type eject to eject the diskette. Insert the next diskette.
8. Move to the second window and type the name of the diskette device (/dev/rfd0) to continue the process.
9. If prompted for more diskettes, repeat steps 7 and 8 above.
10. Insert the driver diskette (Processor Probe Driver or Device Driver).
11. From the second window run the command:

    cpio -ivB < /dev/rfd0

12. Skip to instruction 20 below:

Instructions for Solaris

13. From the first window run the command 'volcheck'. This creates a file called /vol/dev/rdiskette0/unlabeled (the diskette device name).

    If the system pops up a message box saying the diskette format is unrecognized, ignore the message and cancel the message box. The name of the file created may be different on your system; use the command eject -q to see the actual name.

14. From the second window run the command:

    cpio -ivB rwppc.SunOS5.tar.Z rw_inst </vol/dev/rdiskette0/unlabeled

15. When the system prompts you for a new volume, move to the first window and type eject to eject the diskette. Insert the next diskette and type volcheck.
16. Move to the second window and type the name of the diskette device (/vol/dev/rdiskette0/unlabeled) to continue the process.
17. If prompted for more diskettes, repeat steps 15 and 16 above.
18. Insert the driver diskette (Processor Probe Driver or Device Driver) and type "volcheck" from the first window.
19. From the second window run the command:

    cpio -ivB < /vol/dev/rdiskette0/unlabeled

Instructions for both SunOS and Solaris

20. From the directory where RISCWatch was installed, type "./rw_inst" to untar the RISCWatch files and install the parallel port device driver, if required.

21. Modify the **rwppc.env** file to make the appropriate changes to any of the RISCWatch environment variables. Select the proper target type, as described in "Environment Resources" in the *RISCWatch Debugger User's Guide*. Users that have a working directory other than the install directory have their own copy of the **rwppc.env** file. Such users should backup their copy, make a fresh one from the install directory, and then merge their changes from the old one to the new one.

    Note that the RISCWatch environment variable TARGET_TYPE, when set to "jtag", informs RISCWatch that you are using the RISCWatch parallel port adapter. When TARGET_TYPE is set to "jtag_eth", it informs RISCWatch that you are using the RISCWatch processor probe. For the current list of TARGET_TYPE settings, refer to "Environment Resources" on page 3-5 in the *RISCWatch Debugger User's Guide*.

22. Add the XVTPATH environment variable to specify the directory for RISCWatch to find the OpenWindows resource file (.frl). Here are examples for setting the environment variable for different user shells, assuming RISCWatch is installed under /usr/rwppc:

    export XVTPATH=/usr/rwppc (for Korn shell)

    setenv XVTPATH /usr/rwppc (for C shell)

    This environment variable should be added to any user's startup shell (.profile for Korn shell, .cshrc for C shell).

23. Add the LD_LIBRARY_PATH environment variable to include the OpenWindows libraries (in /usr/openwin/lib) if it does not already do so. The environment variable should be added to any user's startup shell (.profile for Korn shell, .cshrc for C shell).

    For Korn shell:
    export LD_LIBRARY_PATH=/usr/openwin/lib:$LD_LIBRARY_PATH
    For C shell:
    setenv LD_LIBRARY_PATH /usr/openwin/lib:$LD_LIBRARY_PATH

24. For JTAG Ethernet and non-JTAG targets, an additional step is required to establish communications between the host and target. To modify the **/etc/services** file, the user must be logged in as **root** or superuser (**su**).

    The following lines must be added to the file:

    ```
    osopen-dbg   20044/tcp
    osopen-dbg   20044/udp
    jtag_eth     6470/tcp  # Note: Underscore used, not hyphen
    ```

25. Exit from **root** or **su**.

This completes the software installation for RISCWatch.

Notes:

- It may be necessary to add the chosen directory to the PATH environment variable if it has not already been added. Furthermore, it may be necessary to change ownership of this directory as well as all of its files if many people will need access to the RISCWatch program.
- If you are using the RISCWatch processor probe, refer to "Invoking the Debugger" on page 3-7 in the *RISCWatch Debugger User's Guide* before you start RISCWatch.
- OpenWindows must be running before starting RISCWatch.

Notes for SunOS

1. rw_inst will update /etc/rc.local, if it exists, to automatically load the RISCWatch parallel device driver upon machine reboot. If /etc/rc.local does not exist or the modifications made by rw_inst are removed, type "rw_inst -a" to manually load the device driver into the kernel after machine reboot.
2. libC.so.5.0 and libC.sa.5.0, two dynamic libraries which RISCWatch needs to run, will be installed in /usr/lang/lib.

Index

IBM

RISCWatch Debugger User's Guide

13H6964

Eighth edition (September 1996)

This edition of *IBM RISCWatch Debugger User's Guide* applies to IBM RISCWatch Debugger Version 3.3 and to all subsequent versions of the debugger until otherwise indicated in new versions or technical newsletters.

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 xix and page xxi, respectively. You may also address written comments about this publication to:

IBM Corporation
Department 0H83A
P.O. Box 12195
Research Triangle Park, NC 27709

Printed in the United States of America.

4 3 2 1

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
Micro Channel
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

Figures

Tables

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: ppc400pubs@vnet.ibm.com

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

- ☐ OS Open Real-Time Operating System
- ☐ PowerPC 403 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:

Host Platform	RS/6000 ☐	Sun (SunOS) ☐	Sun (Solaris) ☐
	PC (Win 3.1) ☐	PC (Win 95) ☐	
Target Processor	403GA ☐	403GB ☐	403GC ☐
	602 ☐	603 ☐	603e ☐
	604 ☐	Other:_____	
Target Platform	IBM Evaluation Board ☐	Other Evaluation Board (please specify):	
	Other Platform:_____		
Target Application	IBM ROM Monitor ☐	OS Open ☐	Own ROM Monitor ☐
		Other:_____	
Interface Used	JTAG (via Parallel Port) ☐	JTAG (via Microchannel) ☐	JTAG (via Ethernet) ☐
	Ethernet ☐	SLIP ☐	Token Ring ☐

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: ppc400pubs@vnet.ibm.com

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

☐	High C/C++ Language Reference
☐	High C/C++ Compiler, ELF Linker and Assembler
☐	OS Open User's Guide
☐	OS Open Programmer's Reference Volume 1
☐	OS Open Programmer's Reference Volume 2
☐	PowerPC 403 Evaluation Board Kit User's Guide
☐	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.

This release of the RISCWatch Debugger supports the following PowerPC processors and versions:

- PowerPC 401GF
- PowerPC 403GA
- PowerPC 403GB
- PowerPC 403GC
- PowerPC 602 Rev2
- PowerPC 603 Rev3
- PowerPC 603e Rev1
- PowerPC 603e Rev3
- PowerPC 603ev Rev2
- PowerPC 604 Rev3
- PowerPC 604ev Rev2

For PowerPC 6xx processors, this version of RISCWatch does not support Micro Channel or parallel port adapters for JTAG targets.

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.

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:

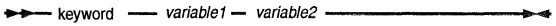
- The names of invariant objects known to RISCWatch appear in bold type. In some text, however, such as in lists, no special typographic treatment is used. Examples of such objects include:
  - File and command names
  - Data types and structures
  - Constants and flags
- Variable names that are supplied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

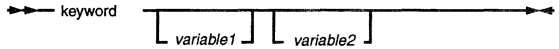
Syntax Diagram Conventions

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.

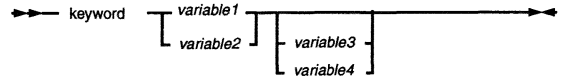


- Keywords or variables shown on branches below the main path are optional.

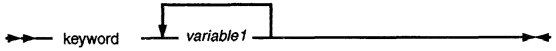


- Keywords or variables can appear in a stack, indicating that only one item in a stack can be chosen. If an item in a stack is on the main path, you must choose an item from the stack. If all items in a stack are below the main path, you may choose an item from the stack.

  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 3.2 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, SC23-2197

Commands Reference, Volume 1, SC23-2376

Commands Reference, Volume 2, SC23-2366

Commands Reference, Volume 3, SC23-2367

Commands Reference, Volume 4, SC23-2393

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

To receive a copy of the EABI specification, send an email to eabi@goth.sps.mot.com, and include the word "eabi" or "EABI" in the subject line. The EABI PostScript file will be sent to you.

- **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**

PPC403GA Embedded Controller User's Manual, 13H6960

403GA Evaluation Board Kit User's Manual, 13H6987

PPC403GB Embedded Controller User's Manual, 13H6985

PPC403GC Embedded Controller User's Manual, 13H6986

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

PowerPC 602 RISC Microprocessor User's Manual, in progress

PowerPC 603 RISC Microprocessor User's Manual, MPR603UMU-01

PowerPC 603e RISC Microprocessor User's Manual, MPR603EUM-01

PowerPC 604 RISC Microprocessor User's Manual, MPR604UMU-01

- **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 (SVR4 ABI and PowerPC Embedded ABI) and XCOFF/STABS.

Hardware Level Debugger (HLD) Tutorial

Included in the installed RISCWatch software is a text file named **rwppc.tut**. This file contains a tutorial on how to use the various facilities of RISCWatch to perform hardware level debugging.

The tutorial is broken up into several lessons progressing from the skills of the beginner up to the intermediate and advanced user levels. Followed in the given sequence, these lessons will familiarize the user with an understanding of the workings of hardware level debug.

This tutorial has been distributed as a text file so that you can view it online with your favorite editor while you are using the debugger, or so that you can print it if you prefer.

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 level debug capability
- Low-level program debug (assembly level)
- Read, modify and write of all processor registers
- Read, modify and write of processor register fields
- Read, modify and write of all processor memory (1, 2 & 4 byte) with memory fill and write verification testing
- Memory loading of many types of file formats (ELF, XCOFF, HP UNIX, Extended Tektronix Hexadecimal, Motorola 32-bit, Verilog, and straight binary)
- Save/load processor memory image to/from file
- Save/load processor register values to/from file
- Command file execution
- Command file execution with user-created variables, programming constructs, expressions and **printf**-like function
- Command file single-step execution
- Command file calling input parameters
- Batch mode command file execution
- Command sequence recorder with file save and playback
- Program disassembler allowing memory modify/write capability
- Program assembler allowing memory write capability
- Single-step execution (assembly or source) of loaded program
- Set/clear of multiple-event breakpoints
- Saving and loading of customized window layout
- User-defined windows consisting of register, register field, memory and disassembly interfaces

- User-defined buttons window
- Processor reset functions
- Logging of all commands and messages
- File browsing
- Shell command capability
- On-line help for all screens including extensive processor register definitions and assembly instructions user's guide

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.

Compiling the Example Program

For ROM Monitor and JTAG targets, no compilation is necessary. There is an executable already included called "demo", compiled with debug information to run on PowerPC processors. The demo3.c file was compiled without debug information to demonstrate the assembly debug capabilities of RISCWatch.

For OS Open targets, refer to the section "Developing OS Open Applications" in the OS Open User's Guide for information on how to include these example files in an OS Open image or dynamically loadable object that can be loaded onto the target processor.

Starting the Debugger

Before you start the RISCWatch debugger , alter the 'rwppc.env' file to designate the correct target type, target name, and RISCWatch directory, as described in "Environment Resources" on page 3-5 and "Invoking the Debugger" on page 3-7. Alter any additional environment resources required for your specific setup.

From a RISC System/6000 workstation running Motif, type "rwppc"to run RISCWatch, or type "rwppc -par" if the parallel port is being used to communicate with a JTAG target.

From a Sun workstation running OpenWindows, type "rwppc" to run RISCWatch.

When running under Windows on a PC, simply double-click on the RISCWatch icon created during program installation. The following window will be displayed:

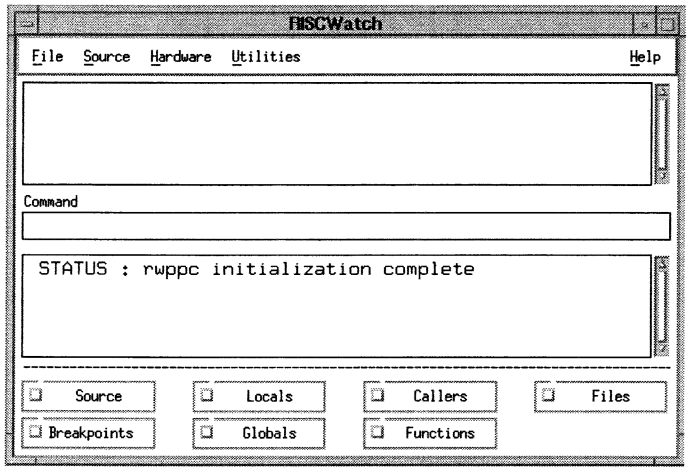


Figure 2-1. Sample Main Window

Note: An OS Open button will only be displayed if the target specified is OS Open. Also, if the target is JTAG, an additional Chip pulldown will be present.

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 5-2 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 examples 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

For JTAG and ROM Monitor targets, enter from the command line:

load file demo t=0xa000 d=0xc000

For OS Open targets (if compiled into an OS Open image), enter from the command line :

start_thread main

or :

load file filename

where *filename* is the fully qualified name of the dynamically loaded object module.

Note: If the target board under test does not have this address range configured, use other valid values.

Scrolling Through Source Code

Now that the program has been loaded, the next step is to bring up the source files. Move the cursor to the "Files" button on the Main window, and single-click the left mouse button. The following window will be displayed:

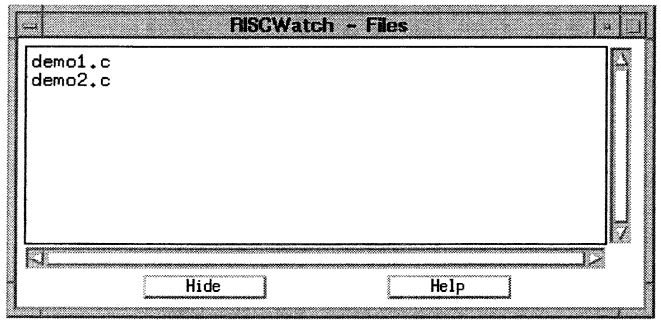


Figure 2-2. Sample Files Window

Single-click the left mouse button on the "demo1.c" entry in the Files window. It will become highlighted.

Single-click the left mouse button on the "Source" button on the Main window. The following window will be displayed:

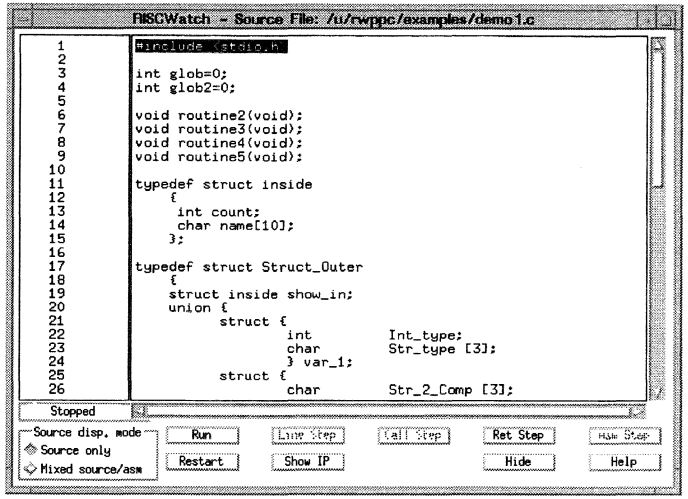


Figure 2-3. Sample Source Window

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.

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.

Move the cursor into the source file area next to line 39 over the statement "i = 111;". Single-click the left mouse button. A "BP" indicator will appear next to the line number 39. This means a breakpoint has been set at line 39.

Single-click the left mouse button on the "Breakpoints" button on the Main window. The following window will be displayed:

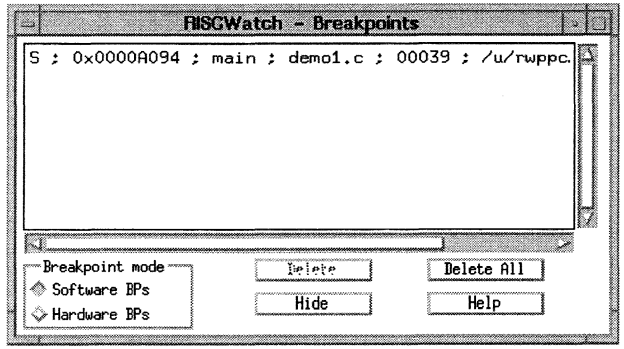


Figure 2-4. Sample Breakpoints Window

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.

Single-click the left mouse button on the "Functions" button on the Main window. The following window will be displayed:

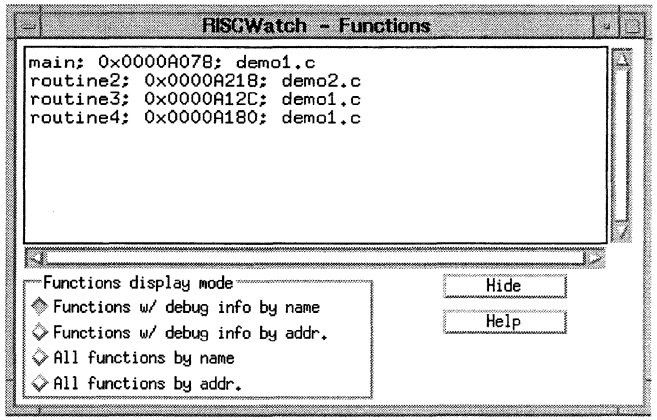


Figure 2-5. Sample Functions Window

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 on the same "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, and 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 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 back to the Main window, and press the "Callers" button. The following window will be displayed:

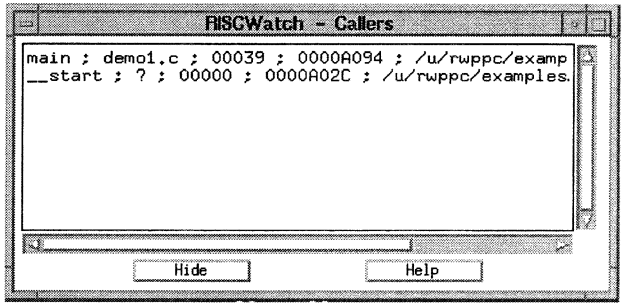


Figure 2-6. Sample Callers Window

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 actually enter 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.

For the JTAG and ROM Monitor targets, 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

Go back to the Main window and press the "Locals" button. The following window will be displayed:

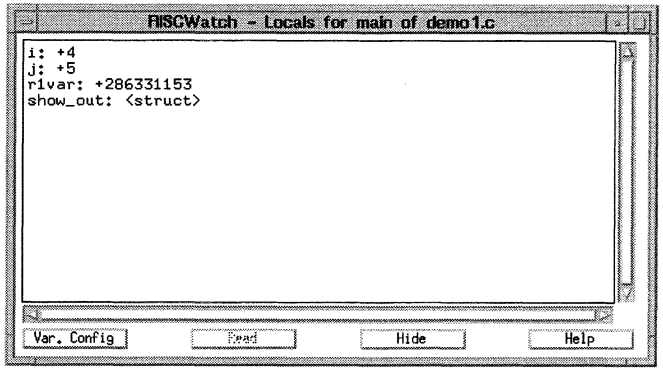


Figure 2-7. Sample Locals Window

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" on page 3-64 for a complete description of the available options. Only a few will be shown in this example.

Press the "Variable Configuration" button on the Locals window. Figure 2-11 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.

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.

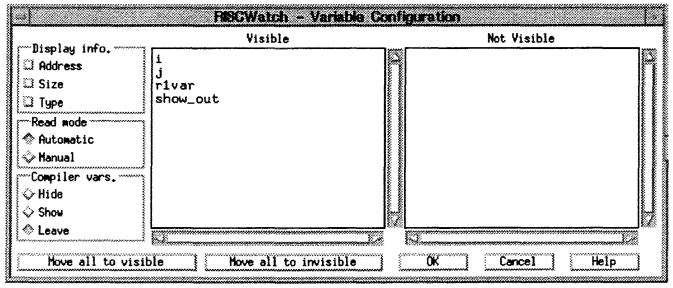


Figure 2-8. Sample Variable Configuration Window

Individual variables may also be custom tailored. Single-click on the "show_out" variable in the Locals window. The following window is displayed:

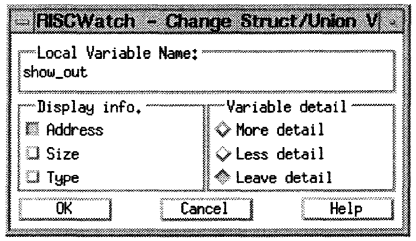


Figure 2-9. Sample Change Struct/Union Window

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.

Single-click on the +2 next to the ".count:" variable. The following window is displayed:

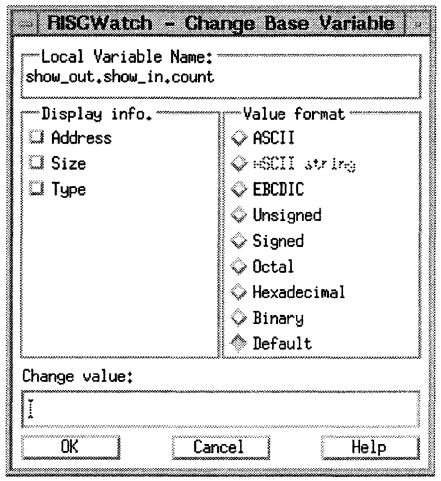


Figure 2-10. Sample Change Base Window

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 carried out 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 in the Source window. However, 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" choice. The following window is displayed:

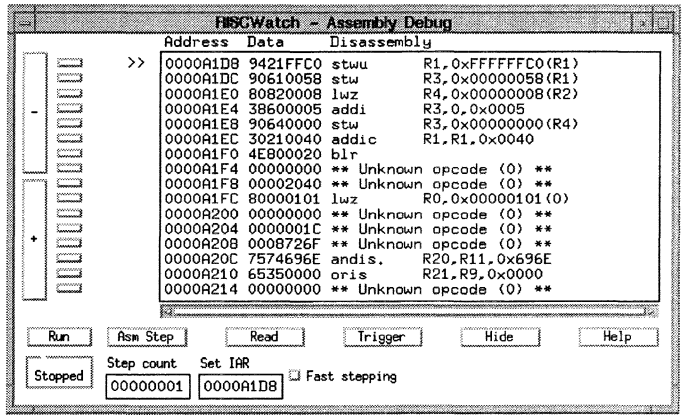


Figure 2-11. Sample Assembly Debug Window

This contains a memory disassembly of a number of instructions, beginning with the one corresponding to the current instruction pointer. Press the "Asm Step" 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 "Asm Step" 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 "Asm Step" button until the return is made to the calling function. The Source window is updated to show the 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 "Mixed 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 "cror 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 the "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 "Command File Programming" on page 3-94 for more details on how to run RISCWatch in this mode.

- Remote/network debug mode (non-PC host only)

  A TCP/IP communications link is used to debug a PowerPC processor connected to a remote host. The graphical user interface is present on the local host while the remote processor is being accessed via RISCWatch running on the remote host. This mode may be likened to running RISCWatch in normal mode over a remote login session.

- TTY mode (non-PC host only)

  This mode allows RISCWatch to be run on a RISC System/6000 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.

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.

Table 3-1. Quick Reference for the RISCWatch Debugger

Task or Resource	Applicable Sections
Setting the Environment How to initialize the **rwppc.env** file	"Environment Resources" on page 3-5
Invoking the Debugger How to bring up the RISCWatch Main Window	"Invoking the Debugger" on page 3-7 "JTAG Ethernet Targets and the RISCWatch Processor Probe" on page 3-10
Main Window Resources Overview of menus and windows	"Main Window Resources" on page 3-11 "Menus" on page 3-12 "Command Line Usage" on page 3-15 "Command History Usage" on page 3-16 "Message Window" on page 3-16
Running Your Programs How to compile, load, and execute programs	"Preparing the Program for Debug" on page 3-16 "Loading Files" on page 3-17 "Loading Boot and Boot Image Files" on page 3-18 "Executing the Program" on page 3-20 "Following Program Execution Flow" on page 3-20 "Input Line Usage" on page 3-20
Source Level Debugging How to use the interface to debug your C source code	"Source Window" on page 3-23 "Assembly Debug Window" on page 3-26 "Programs Window" on page 3-30 "Callers Window" on page 3-32 "Files Window" on page 3-33 "Functions Window" on page 3-33
OS Open Debugging How to use the interface to display operating system information and to control debug attachment	"OS Open Debugging" on page 3-35
Managing Breakpoints How to use the interface and command set to set hardware and software breakpoints	"Managing Breakpoints" on page 3-39 "Using Software Breakpoints" on page 3-39 "Using Hardware Breakpoints" on page 3-40 "Breakpoints Window" on page 3-41 "Breakpoint Select Window" on page 3-43 "Trigger/Trace Window (400Series Only)" on page 4-6 "Compound Trigger/Trace Window (400Series Only)" on page 4-9

Table 3-1. Quick Reference for the RISCWatch Debugger

Task or Resource	Applicable Sections
Reading and Writing Program Data How to use the interface to read, modify, and write program variables	"Reading and Writing Program Data" on page 3-44 "Program Variables" on page 3-44 "Variable Windows" on page 3-60 "Local Variables Window" on page 3-60 "Global Variables Window" on page 3-62 "Formatting Variables Overview" on page 3-45 "Changing Variable Information via Change Variable Windows" on page 3-45 "Configuring Variable Information via the Variable Configuration Window" on page 3-45 "Configuring Variable Information via the Variable Configuration Window" on page 3-45 "Expanding/Contracting Variable Detail" on page 3-46 "Variable Configuration" on page 3-64 "Change Variable Windows" on page 3-66 "Change Array Variable" on page 3-67 "Change Base Variable" on page 3-68 "Change Enum Variable" on page 3-70 "Change Pointer Variable" on page 3-71 "Change Struct/Union Variable" on page 3-74
Reading and Writing Memory 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 3-75 "Assembly Debug Window" on page 3-26 "Memory Access Window (JTAG Target Only)" on page 3-75 "ASCII Memory Window" on page 3-78 "Custom Memory Window" on page 3-80 "Cache Windows (JTAG Target Only)" on page 3-82 "Translation Lookaside Buffer Window (PPC403GC Only)" on page 4-12
Reading and Writing Registers How to use the interface and command set to read, modify, and write processor registers and register fields	"Reading and Writing Registers" on page 3-84 "Register Windows" on page 3-84 "Register Field Windows" on page 3-86
User-Defined Resources	"User-Defined Windows" on page 3-87 "User-Defined Buttons" on page 3-90

Table 3-1. Quick Reference for the RISCWatch Debugger

Task or Resource	Applicable Sections
Command Files How to create and run command files which are used to perform repetitious tasks and help to automate testing	"Command Files" on page 3-92 "Command File Programming" on page 3-94 "Command File Special Expressions" on page 3-95 "Command File Parameters" on page 3-96 "Command File Pseudo-Variables" on page 3-97 "Running a Command File" on page 3-98 "Command File Programming Example" on page 3-97 "Running a Command File" on page 3-98 "Command File Single-Step Window" on page 3-98
Processor Resources How to use the interface to perform processor resets and to read processor status	"Processor Resources" on page 3-100 "Processor Reset Window (JTAG Target Only)" on page 3-100 "Processor Status Window (400Series JTAG Only)" on page 4-14
General Resources How to use various program resources	"Window Layout" on page 3-102 "Window List" on page 3-102 "Log Files" on page 3-102 "Logging Control" on page 3-103 "Logging User Comments" on page 3-104 "Viewing Log Files" on page 3-104 "Shell Command Window (Non-PC Host Only)" on page 3-105 "Screen Capture" on page 3-105 "Calculator Window" on page 3-106 "Profiler Window" on page 3-107
RISCTrace Describes using RISCTrace and the trace capabilities of 400Series processors	"Using RISCTrace (400Series JTAG Processor Probe Only)" on page 4-2
Help How to use the interface to display the extensive on-line information available while debugging	"Online Help" on page 3-108

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 to 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:

Resource name	Description
TARGET_TYPE	jtag, jtag_eth, rom_mon, osopen (one required) If the target type is osopen, rom_mon, or jtag_eth, refer to the README file which came with RISCWatch for information on which version levels of these targets are required for proper RISCWatch operation. Each target type is described below.
jtag	JTAG target. RISCWatch is connected through a parallel or Microchannel interface to the JTAG port on the PowerPC 400Series target system.
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 target system.
rom_mon	IBM ROM Monitor target. RISCWatch is connected via Ethernet or SLIP to a PowerPC target system running the IBM ROM Monitor for PowerPC 400Series 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 (required for JTAG Ethernet, OS Open and ROM Monitor targets) TCP/IP dotted address may also be used.
RW_DIR	A fully qualified path name to the directory in which the RISCWatch executable and support files reside.
SEARCH_PATH	Path names used for source/object search, delimited by colons (:) (optional, default = current directory); for a PC host, the delimiter is a semicolon instead of a colon

CMD_FILE_DIR	A fully qualified path name to the directory of where to find RISCWatch command files. Using command files is described in "Command File Programming" on page 3-94.
LOG_FILE_DIR	A fully qualified path name to the directory of where RISCWatch is to maintain all log files.
CMD_FILE_LOG	Whether or not the log file will be updated while running a command file.
EDITOR	The fully qualified name of the program to be called when a file is edited using either the **edit** command or the Edit selection from the File Menu.
ANNOUNCE	Whether or not the application notes file (**rwppc.anf**) will be displayed on program start.
STACK_FRAMES	Indicates the number of stack frames to show on the Callers Window. If not designated, the default setting is twelve.
STACK_SIZE	Indicates the number of bytes to reserve for the stack if it is not supplied on the **load** file command. The stack address is calculated by adding this value to the last byte loaded on the target. The stack address is forced word aligned. If not specified, the default stack size will be set to 16K, provided a stack address is not designated (via STACK_ADDR or 's=' option on the **load** command).
STACK_ADDR	Indicates the value to be used for the stack address if it is not supplied on the LOAD FILE command. This value is ignored if a stack size is designated. USE OF THIS ENVIRONMENT RESOURCE IS NOT RECOMMENDED. Designating a stack size is the preferred method of setting the stack address since certain applications may define a heap space starting beyond the stack size.
ADAP_FWARE_FILE	The name of the file which contains the firmware used to program the RISCWatch MCA card.
BUFF_FWARE_FILE	The name of the file which contains the firmware used to program the RISCWatch buffer card.
SAVE_LAYOUT	Save/restore window layout when ending/beginning session (yes/no) (optional, default = yes)
APPLPROG_NAME	Allows renaming of applprog executable (OS Open target only - optional)

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.

603_DRTRYMODE — For 603 processors which are run in Data Retry ($\overline{\text{DRTRY}}$) mode, 603_DRTRYMODE must be set to yes for RISCWatch to operate properly (optional: default = no).
The PowerPC 603 can be put in DRTRY mode by having the $\overline{\text{DRTRY}}$ line on the PowerPC 603 negated during the negation of $\overline{\text{HRESET}}$. Please see the *PowerPC 603 User's Manual* for more information.

Note: RISCWatch cannot detect that the 603 has $\overline{\text{DRTRY}}$ mode enabled, nor can it change the $\overline{\text{DRTRY}}$ mode of the 603 processor.

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:

RW_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 run, 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
2. Check to see if it is in a directory specified by the environment variable PATH; if so, use it
3. Check to see if it is in the same directory as the executable specified on start-tup; if so, use it, else
4. Print an error message and terminate RISCWatch.

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 Target (MicroChannel or Parallel Port Connection):

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

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

- 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 7.1.1.4').

- ROM Monitor Target:

  Verify that the host is configured correctly for serial port or 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 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 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 **rsId_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 3-18) or by using ROM monitor **bootp** support. See the evaluation board kit user's documentation and the *OS Open User's Guide*, listed in "Related IBM Publications" on page xxvi 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 (ex '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-PC platform.
-help or ?	used to display the help information for RISCWatch which lists all of the available command line options
-par	specifies that the parallel port adapter should be used for JTAG communications to the processor. See the Software Installation section in the RISCWatch Installation Guide for variations of this parameter when running under Windows.
-proc*NAME*	tells RISCWatch what processor it is debugging. Valid processor names are: 403GA, 403GB, 403GC, 602, 603, 603e, 604. It is recommended that the -proc flag only be used when attaching via a JTAG Ethernet processor probe for the first time, or when switching to a processor which would require a processor probe driver change. Currently, all PowerPC 6xx processors have separate driver files and the 400Series processors are contained in a single driver file. Therefore, the -proc flag would be needed when switching from any PowerPC 6xx processor to another PowerPC 6xx processor or 400Series processor, or when switching from any 400Series processor to a PowerPC 6xx processor.
-prog	forces RISCWatch to reprogram the Micro-Channel and buffer card firmware. This will be needed if the buffer card loses power, such as when the adapter cable is disconnected. (JTAG targets only)
-rev	Distinguishes between different 6xx processor revision levels when connected via the RISCWatch Processor Probe. The -rev flag must be used when debugging a 6xx 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.
-slot	specifies the Micro-Channel adapter slot holding the RISCWatch adapter card that RISCWatch is to communicate with; the option is followed immediately by the number of the slot (for example, -slot3). (JTAG targets only)
-tty	specfies 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-PC 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) against the versions of the files located in the directory specified by the RW_DIR environment variable. If the Generics or JTAG drivers do not match, the file(s) from the RW_DIR will be loaded into the processor probe. Because loading the processor probe will corrupt the processor's JTAG controller, RISCWatch will reset the processor if new drivers are loaded.

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.

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, the debugger brings up only the main command window. It is this window, shown in Figure 3-1, that will be used to access all of the debugger features.

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.

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 and then the 'Command syntax' menu option.

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.

Checkboxes located at the bottom of the Main window control and provide access to several other source level debug windows.

Clicking on a checkbox toggles the state of the corresponding window. If the window is closed, clicking its checkbox opens the window. If the window is open, clicking its checkbox closes the window. When a window is open, its checkbox will be selected.

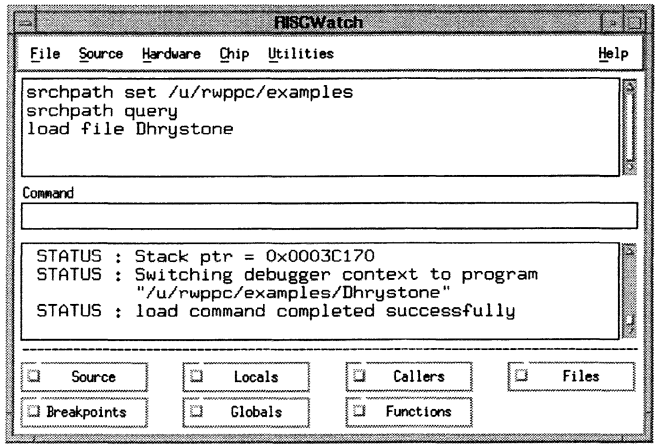


Figure 3-1. Sample Main Window

Note: An OS Open checkbox will only be displayed if the target specified is OS Open. Also, if the target is JTAG, an additional Chip pulldown will be present.

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 3-13.

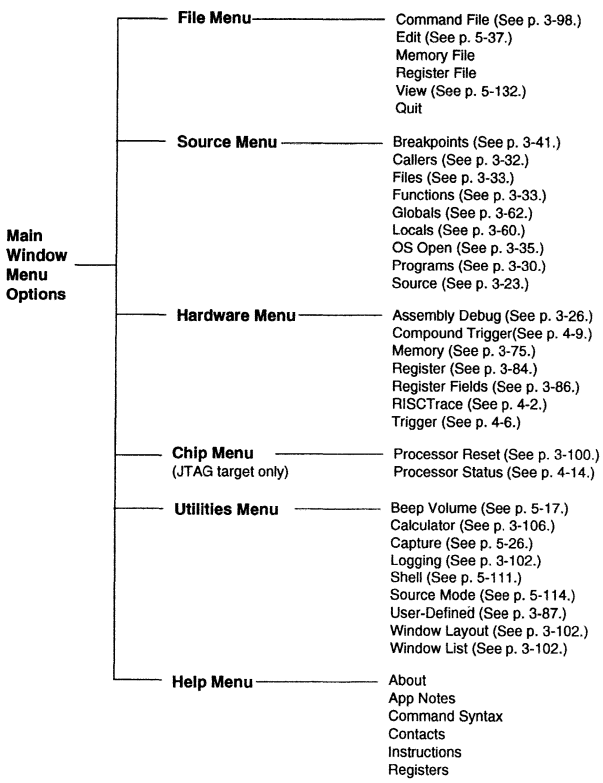


Figure 3-2. Main Window Menu Options

The menu bar contains the following menus:

- File
- Source
- Hardware
- Chip (JTAG target only)
- Utilities
- Help

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

File Menu

Command File	Run a command file
Edit	Edit a selected file (non-PC host only)
Memory File	Load/Save a memory file
Register File	Load/Save a register 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	Display OS Open threads and status (OS Open target only)
Programs window	Displays programs in current context
Source window	Displays source file in current context

Hardware Menu

Asm. Debug	Displays the Assembly Debug window
Com Trig	Displays the Compound Trigger window
Memory	Displays memory window pull-down
Register	Displays a register access window
Reg Fields	Displays a register field access window

RISCTrace	Displays the RISCTrace window (JTAG target, RISC System/6000 host only)
Trigger	Displays the Hardware Trigger window

Chip Menu (JTAG Target Only)

Reset	Displays the Processor Reset window
Status	Displays the Processor Status window

Utilities Menu

Beep Volume	Adjust the program error beep volume
Calculator	Displays the desktop Calculator window
Capture	Captures screen contents to a file
Logging	Log comments or enable/disable logging
Shell	Pass a command to AIX to execute (non-PC host only)
Source Mode	Sets source mode on or off
User-Defined	Loads a user-defined window or buttons definition
Window Layout	Loads or saves a window layout
Window List	Display window list

Help Menu

About	Display RISCWatch version information
App Notes	Display the RISCWatch application notes file (**rwppc.anf**)
Command Syntax	Display RISCWatch command syntax
Contacts	Display RISCWatch technical support contacts
Instructions	Display PowerPC assembly instruction information
Registers	Display detailed RISCWatch register information

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 5-2.

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 3-20.

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.

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.

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 3-102.

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 Files|MemoryFile|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.

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).

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.

   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 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. Hide all windows except the Main window.
3. Issue the command **attach 42** to inform the ROM Monitor that a process will be running.
4. Issue the **load image** command with the file name of the OS Open boot image file.
5. 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.
6. Select 'file' on the Main window and then select 'quit' to exit RISCWatch.
7. 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.
8. Bring up RISCWatch using the OS Open target.
9. Issue a **start_thread** or **attach** command to the thread you want to debug.
10. 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 startup 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 used to provide a shortcut method of performing some user interface 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 function which is supported by that window. Table 3-2 below describes each of the available functions:

Table 3-2. Input Line Functions

Key	Function	Parameter	Supported Windows
F12	command line	any command line command	all
/	find forward (**find** command)	search string	specified in **find** command description
****	find backward (**findb** command)	search string	specified in **findb** command description
?	find exact (**finde** command)	search string	specified in **finde** command description
:	scroll to line (**line** command)	line number	specified in **line** command description
;	scroll to source line (**srcline** 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 to be used to specify any parameters for the function, if necessary. 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 hide 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

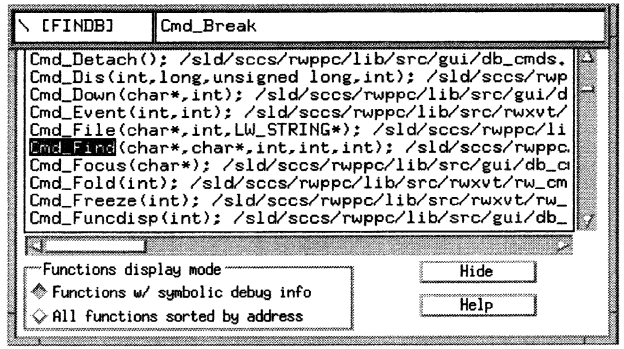


Figure 3-3. Sample Input Line Displayed

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...' pushbuttons 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.

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

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 3-4 shows the Source window in Mixed 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:

- 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 or performing an execution command.

- 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.

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.

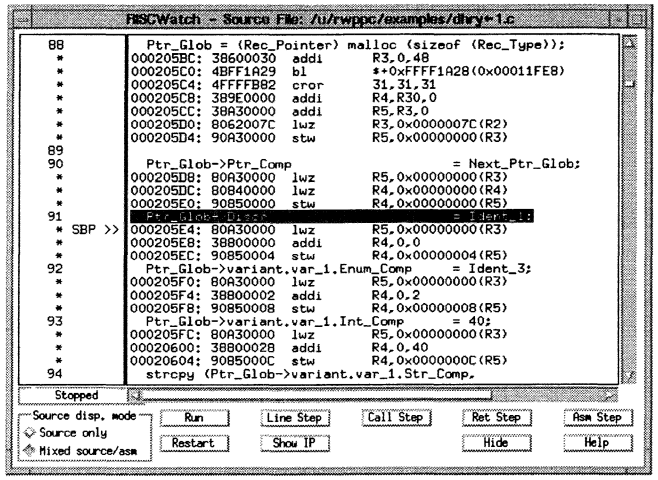


Figure 3-4. Sample Source Window

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 Mixed 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 Mixed 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 Mixed 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 Mixed 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 Mixed 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, breakpoints can only be set on assembler instructions (requiring Mixed 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. 3-28). 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 clicking on a valid line. If no break is currently set at the line, a breakpoint is set by clicking 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 clicking 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.

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.

Refer to "Debugging at the Assembly Level" on page 2-14 for an example of how assembly level debug can be performed.

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

- **Page Up/Down buttons**

The page up and page down buttons are located along the left hand edge of the Assembly Debug window. These buttons are used to page through memory. Clicking on a page button alters the display address by one line or opcode. Double-clicking on a page button 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.

- **Breakpoint buttons**

The breakpoint buttons are located right next to the page buttons and are used to set and clear hardware or software breakpoints. There is one breakpoint button for every line in the data area containing the addresses, data words and disassembled memory.

To set a hardware or software breakpoint for a particular memory address, simply use the mouse to click on the appropriate breakpoint button. 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

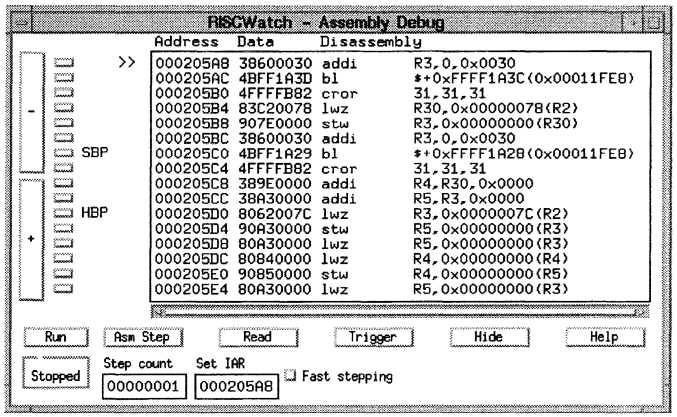


Figure 3-5. Sample Assembly Debug Window

address an 'HBP' or 'SBP' marker appears next to the breakpoint button indicating that a hardware or software breakpoint has been set for that address.

To clear a breakpoint, simply click on the breakpoint button which has the 'HBP' or 'SBP' marker next to it. This will clear the breakpoint and remove the marker for that breakpoint button.

- **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 and will point to the IAR memory address if it appears in the data area display text.

- **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 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.

- **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 run or stopped. In the case of a ROM Monitor or OS Open target, a process is run 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. 3-25). 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, while the data area will display the code at the IAR address and the IAR cursor will point to the appropriate memory data.

- **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.

- **Trigger button (400Series Only)**

The Trigger button is used to display the Trigger window which is used to enable and disable hardware breakpoints. See "Trigger/Trace Window (400Series Only)" on page 4-6 for information on how to use this window. Hardware breakpoints are not available for OS Open targets.

- **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.

- **Fast stepping (400Series Only)**

Fast stepping is used to speed up the execution of single instruction stepping. This function is controlled by the check box located in the lower right-hand corner of the window. When the check box is selected, fast stepping is on. By default, fast stepping is off when the Assembly Debug window is displayed.

When fast stepping is on, there are two limitations to be aware of:

1. Stepping over an RFI, RFCI or SC instruction will not work properly.
2. Stepping over an instruction which has a hardware debug event active or pending will not be successful.

If these types of instructions must be stepped over, make sure fast stepping is off.

Programs Window

The Programs window consists of a Programs subwindow with horizontal and vertical scrollbars, and pushbuttons.

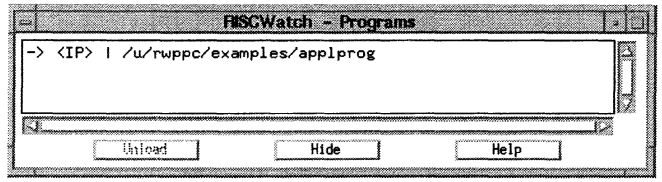


Figure 3-6. Sample Programs Window

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 will appear in this field if the program entry matches the program in which the current Instruction Pointer is located, otherwise it will be 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 will still be updated to show the programs loaded on the host. In this case the source level debug

screens will not be functional, so single-clicking the mouse on an entry will not affect any source debug screens. The window can still be used, however, to unload programs.

If the mouse is single-clicked on a program entry for the program which is currently active (ie., has the '->' symbol next to it), the selection is highlighted and the Unload pushbutton will become enabled. The Unload pushbutton 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 would be 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).

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 3-7.

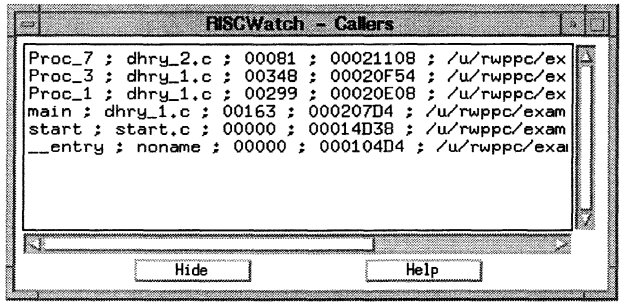


Figure 3-7. Sample Callers Window

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. 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.

Files Window

The Files window displays source file names in the current context. This window consists of a menu bar and a scrolling text window, as illustrated in Figure 3-8.

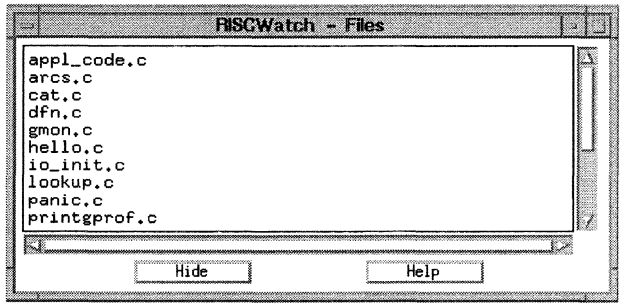


Figure 3-8. Sample Files Window

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 FileIView pulldown found on the Main window or by using the **view** command.

Functions Window

The Functions window consists of a Functions subwindow with horizontal and vertical scrollbars, a Function 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 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 mode is changed by selecting the appropriate button. The button which is selected indicates the current mode.

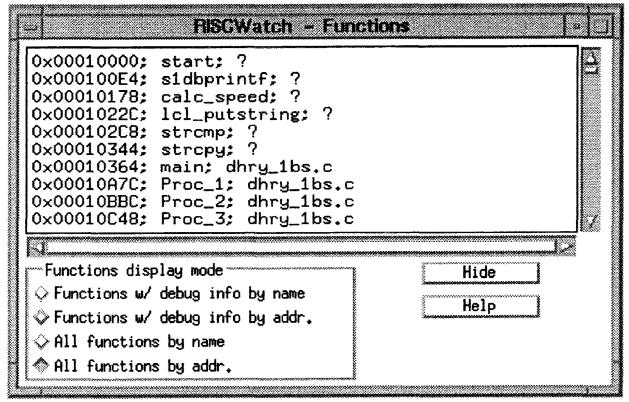


Figure 3-9. Sample Functions Window

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 functions 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.

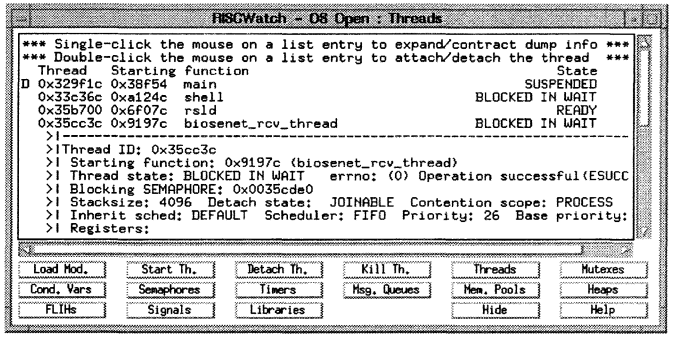


Figure 3-10. Sample OS Open Window

A breakpoint can be toggled by double-clicking on a function entry. If the function associated with the function entry has symbolic debug information, the necessary breakpoints corresponding to the first executable line of the function will be toggled. If the function does not have symbolic debug information, 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.

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.

The OS Open window consists of a subwindow with horizontal and vertical scrollbars and a number of pushbuttons 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 pushbutton 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.

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.

Notes:
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 pushbutton 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.

- **Hide button**

This pushbutton hides the window.

- **Help button**

This pushbutton accesses the help information for the OS Open window.

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

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.

Note: For PowerPC 6xx 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

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 by moving the cursor to the targeted source line and then single-clicking the left mouse button. 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. 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 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 by double-clicking the cursor on a function entry. A breakpoint will be toggled at the first executable line of the function if the function has symbolic debug information. A breakpoint will be toggled at the address of the start of the function if the function does not have any symbolic debug information.

- **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 0xFFFC0004, issue the following command:

```
bp set 0xFFFC0004
```

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 0xFFFC0004
```

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

Using Hardware Breakpoints

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 targeted assembly instruction 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.

"Trigger/Trace Window (400Series Only)" on page 4-6 and "Compound Trigger/Trace Window (400Series Only)" on page 4-9 provide descriptions of other (processor-specific) windows for handling hardware breakpoints.

- **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.

Breakpoints Window

The Breakpoints window consists of a Breakpoint subwindow with horizontal and vertical scrollbars, a Breakpoint Mode selection groupbox, and pushbuttons. 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 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.

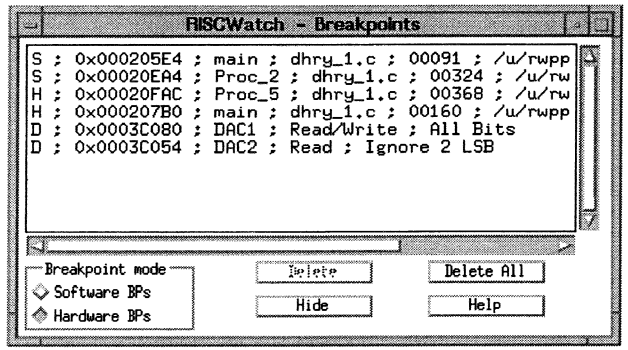


Figure 3-11. Sample Breakpoints Window

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 the 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 two buttons, one for Software BPs and one for Hardware 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, breakpoints are set via the hardware debug registers of the target processor. An example of the use of Hardware breakpoints would be if you were debugging code resident in read only memory, where software traps could 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 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 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 "Trigger/Trace Window (400Series Only)" on page 4-6 and "Compound Trigger/Trace Window (400Series Only)" on page 4-9 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 pushbuttons.

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.

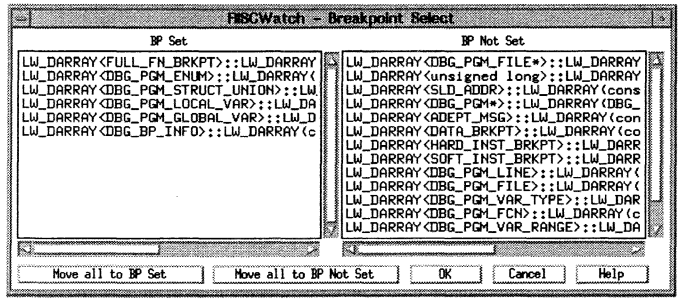


Figure 3-12. Sample Breakpoint Select Window

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 pushbutton will move all the functions to the BP Set subwindow. The Move All to BP Not Set pushbutton 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 Data

Many methods of updating and viewing 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 the data is presented.

Program Variables

Program variables can be viewed and updated using the Locals and/or Global windows. See "Local Variables Window" on page 3-60 and "Global Variables Window" on page 3-62 for detailed descriptions of these windows.

One option is to present the address of the variable. This address can then be used in conjunction with other commands and screens to provide additional ways to view and update the 'variable' contents.

Once the address is known, the program variables can also be viewed and altered using the **read** and **write** commands from the command line on the Main window.

Formatting Variables Overview

Objects in the Locals and Globals windows may be reformatted in a variety of ways, depending on their type.

Changing Variable Information via Change Variable Windows

Single-clicking the left mouse button on a variable entry in the Locals or Globals window will select the variable and open the Change Variable window appropriate for the type of the selected variable (integer, structure etc.). The Change Variable windows are used to configure variable information for an individual variable. There are Change Variable windows for each major variable type:

"Change Array Variable" on page 3-67
"Change Base Variable" on page 3-68
"Change Enum Variable" on page 3-70
"Change Pointer Variable" on page 3-71
"Change Struct/Union Variable" on page 3-74

The following information may be customized for variables via the Change Variable windows (note: the information which may be customized for a particular variable is dependent on that variable's type):

- Display Information (Address, Size and Type)
- Variable Detail (Expand/Contract)
- Value Format (Hexadecimal, Binary, Octal etc.)
- Change Value
- Change Subrange
- Change multiple instances of a variable within an array

Configuring Variable Information via the Variable Configuration Window

Selecting the Var.Config pushbutton on the Locals or Globals window will open the Variable Configuration window for those variables. The Variable Configuration window is used to change variable information for all local or global variables. Refer to "Variable Configuration" on page 3-64 for detailed information on this window.

The following information may be configured for local or global variables via the Variable Configuration window:

- Which variables are visible
- Display Information (Address, Size and Type)
- Read Mode (Automatic/Manual)
- Compiler variables (Hide/Show)

Expanding/Contracting Variable Detail

The level of detail for an individual variable on the Locals or Globals screen can be expanded or contracted. One way to change the variable detail is from the Variable Detail groupbox available on some Change Variable windows. Another shortcut method of changing the variable detail for a variable is to double-click the left and right mouse buttons on a variable entry line within the Locals or Globals variable window itself.

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.

Formatting Examples

Following are some examples of 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 3-13. 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 3-14. 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 3-15. 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 3-16. 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 Array Variable window which is opened by single-clicking the left mouse button on the array variable entry. (See p. 3-67.)

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 3-17. 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 3-18. Sample Pointer Variable

Single-clicking the left mouse button on this variable line will open the Change Pointer Variable window. (See p. 3-71.) 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 3-19. 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 3-20. Sample Character Array

Single-clicking the left mouse button on any of the character variable entries will open the Change Base Variable window. (See p. 3-68.) One of the options under Value Format is ASCII string. Selecting this format and applying the change will result in the character variable entry being updated to show the ASCII string being pointed to, 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 3-21. 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 (in which case the debugger will append a NULL character as the last character).

Handling Multiple Data Elements Referenced by a Single Pointer

Suppose we initialize a data pointer to point to a memory buffer which has been allocated for the purpose of holding a number of identical data structures. Typically, then, individual buffer elements can be manipulated by the program by using pointer arithmetic with the pointer value. It could normally then be cumbersome to view and change any of the various individual data elements in the buffer. RISCWatch provides a way to simplify this task.

Consider the following variable which is a pointer to type **struct record** on a Locals or Globals variable window. It is used to reference 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 3-22. 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 3-23. 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 Pointer 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]:

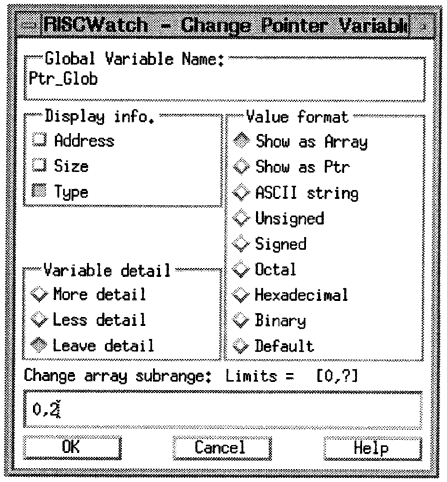


Figure 3-24. Changing Pointer Variables

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 3-25. 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 3-26. 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 to open the Change Pointer Variable window, and then using the Show as Ptr option under Value Format.

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 checkbox on any of the Change Variable windows titled 'Apply to each var. instance at this level' when changes are applied. This checkbox used to apply changes to multiple elements 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 checkbox is selected on a window which contains a Variable Detail groupbox, it will be disabled as long as the checkbox is selected (and any detail selections will be ignored if the checkbox 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 3-27. 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 Base Variable window shown in Figure 3-28. Notice the checkbox 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:

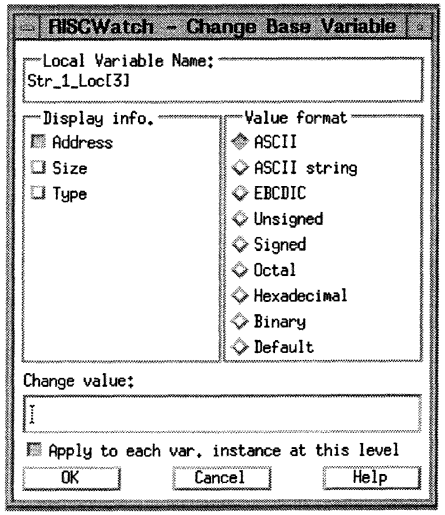


Figure 3-28. Changing Multiple Elements of a Variable Array

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 3-29. 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 3-30. Sample Multi-Element, Multilevel Variable Display

Selecting the Str_Comp array variable of the first Rec_Ptr element brings up the Change Array Variable window. The checkbox 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 checkbox 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 3-31. 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 Base 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 checkbox to apply to multiple instances.

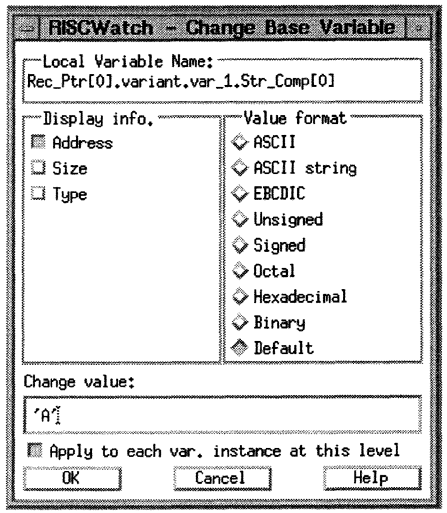


Figure 3-32. Sample Change Value Display

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 3-33. 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.

Variable Windows

A total of eight windows are used to display and manipulate program variable information.

The Locals and Globals windows display selected local and global variables, respectively, for the program currently being debugged.

The Variable Configuration window is selectable from the Locals or Globals window, and is used to configure variable information for all Local or Global variables. The five Change Variable windows are accessed from the Locals or Globals window, and are used to configure variable information displayed for a single selected variable of a particular type.

Local Variables Window

The Locals window displays local variables in the current source file. Figure 3-34 shows an example of a Locals window.

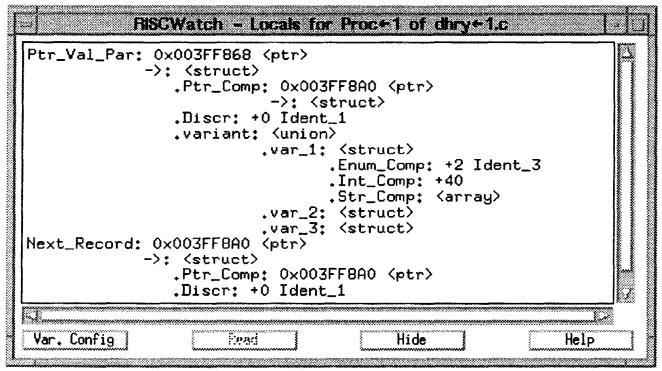


Figure 3-34. Sample Locals Window

The Locals window consists of a Locals subwindow with horizontal and vertical scrollbars and pushbuttons. 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.

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 type of the variable 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 uninitialized 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 windows are used to configure variable information for an individual variable. See "Change Variable Windows," p. 3-66.

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 pushbutton 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," p. 3-64.

The Read pushbutton is used to manually read the values of the variables which are displayed on the Locals window from the target. It is only enabled if the Read mode for the Locals is set to Manual. If the Read mode is set to Automatic, the button is disabled. The Read mode is set via an option on the Variable Configuration window.

One practical use of Manual read mode is to minimize the overhead of normally having the debugger read the visible variable values from the target after each execution command, as happens in Automatic read mode. For local variables the debugger automatically updates the Locals window when the locals context changes to show the variables and their current values for that context, even if the read mode is Manual. This is done so that the local variables shown match the current context when it changes, and those variables can then be manipulated using the Variable Configuration and Change Variable windows.

There are other exceptions where visible variable values are read even if the Read mode is Manual. The variable values are read for all the visible local variables if a variable detail change is made by double-clicking on a variable in the window, if any changes are made via the Variable Configuration window for the Locals window, or if any changes are made for a visible local variable via a Change Variable window.

The rationale for applying the read from the target in these instances is because it is likely that without Automatic read mode on the values shown may not be correct, and the fact that an action is performed on a variable implies that the present state of the variable is desired. Also, in the case of these one-shot updates, the overhead for doing the read will not be noticeable since the read processing will only be a portion of the total processing required for the requested action.

Global Variables Window

The Globals window consists of a Globals subwindow with horizontal and vertical scrollbars and pushbuttons.

The Globals subwindow displays the visible global variables for the program currently being debugged. 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 type of the variable 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.

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.

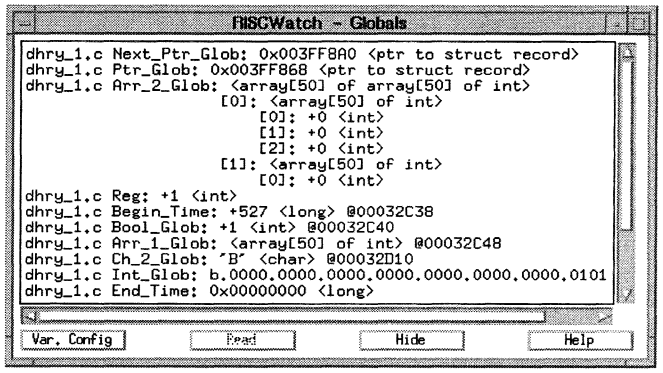


Figure 3-35. Sample Globals Window

For example, if an uninitialized 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 windows are used to configure variable information for an individual variable. Refer to the Change Variable window descriptions.

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 pushbutton 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. Refer to the Variable Configuration window description. The Variable Config pushbutton will be disabled if there is no source debug information for the current program.

The Read pushbutton is used to manually read the values of the variables which are displayed on the Globals window from the target. It is only enabled if the Read mode for the Globals is set to Manual. If the Read mode is set to Auto, the button will be disabled. The Read mode is set via an option on the Variable Configuration window.

One practical use of Manual read mode is to minimize the overhead of normally having the debugger read the visible variable values from the target after each execution command, as happens in Automatic read mode.

There are exceptions where visible variable values will be read even if the Read mode is Manual. The variable values will be read for all the visible global variables if a variable detail change is made by double-clicking on a variable in the window, if any changes are made via the Variable Configuration window for the Globals window, or if any changes are made for a visible global variable via a Change Variable window.

The rationale for applying the read from the target in these instances is because it is likely that without Automatic read mode on the values shown may not be correct, and the fact that an action is performed on a variable implies that the present state of the variable is desired. Also, in the case of these one-shot updates, the overhead for doing the read will not be noticeable since the read processing will only be a portion of the total processing required for the requested action.

Variable Configuration

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 Read Mode 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 pushbuttons.

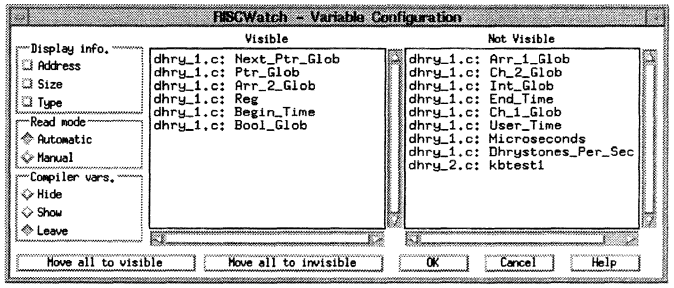


Figure 3-36. Sample Variable Configuration Window

The Variable Configuration window is opened via the Variable Configuration pushbutton on the Locals or Globals window. The OK pushbutton 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 pushbutton is used to close the window without applying any changes.

The Variable Configuration window is intended to be used for the purpose of 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 three checkboxes, one each to display the Address, Size and Type information for the visible variables on the associated variable window. The initial state of the checkboxes shows the currently enabled display information for the associated variable window. If the information on the Variable Configuration window is applied, each 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 variables on the variable window which have been previously 'revealed' or expanded, whether they are currently visible or not.

The Read Mode groupbox consists of two buttons, one for Automatic read mode and one for Manual read mode. The Read mode is changed by selecting the

appropriate button. The button which is on initially indicates the current mode for the visible variables on the associated variable window. If the information on the Variable Configuration window is applied, the selected read mode will be applied to the associated variable window, and the Read pushbutton on the variable window will be enabled/disabled accordingly. Refer to the variable window descriptions for a discussion on variable Read mode.

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 pushbutton will move all the variables to the Visible subwindow. The Move All to Invis pushbutton 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.

Change Variable Windows

The Change Variable windows are used to change variable information for an individual selected local or global variable.

A Change Variable window is opened by single-clicking the mouse on a variable entry in the Locals or Globals window. The type of the variable determines which Change Variable window is opened. There are five Change Variable windows: Change Array Variable, Change Base Variable, Change Enum Variable, Change Pointer Variable, and Change Stuct/Union Variable.

The OK pushbutton is used to apply the selected information to the variable entry on the associated variable window (the variable window from which the Change

Variable window was opened). The Cancel pushbutton is used to close the window without applying any changes.

A Change Variable window is intended to be used for the purpose of applying configuration changes to a variable once it is opened. The Change Variable 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 Change Variable window will also be brought down with no changes applied if another Change Variable window or a Variable Configuration window is opened while the Change Variable window is up.

Change Array Variable

The Change Array Variable window is used to change variable information for an array variable. It consists of Variable Name field, a Display Information selection groupbox, a Variable Detail selection groupbox, a Change Subrange field and pushbuttons. The name of the selected variable appears in the name field, with the title indicating whether it is a local or global variable.

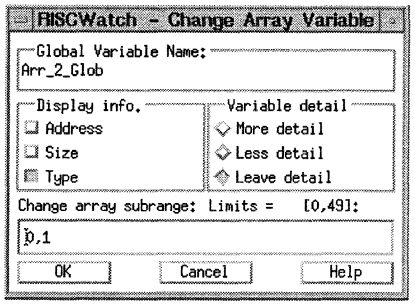


Figure 3-37. Sample Change Array Window

The Display Information groupbox consists of three checkboxes, one each to display the Address, Size and Type information for the selected variable on the associated variable window. The initial state of the checkboxes shows the currently enabled 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 have been previously 'revealed' or expanded, whether they are currently visible or not.

The Variable Detail groupbox consists of three checkboxes: 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 "Expanding/Contracting Variable Detail" on page 3-46 for more discussion on changing the level of detail for a variable.

The Change Subrange field is used to change the subrange to be shown for an array. It will be 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 entered 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 Array 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 the limit 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.

A checkbox 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 somewhere 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 3-52 for a detailed description of this support.

Change Base Variable

The Change Base Variable window is used to change variable information for a variable which is a fundamental type (integer, char, etc.). It consists of a Variable Name field, a Display Information selection groupbox, a Value Format selection groupbox, a Change Value field and pushbuttons. The name of the selected variable appears in the name field, with the title indicating whether it is a local or global variable.

The Display Information groupbox consists of three checkboxes, one each to display the Address, Size and Type information for the selected variable on the associated variable window. The initial state of the checkboxes shows the currently enabled 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 Value Format groupbox consists of a number of buttons used to change the format of the variable value in the variable entry. For example, if the value of the

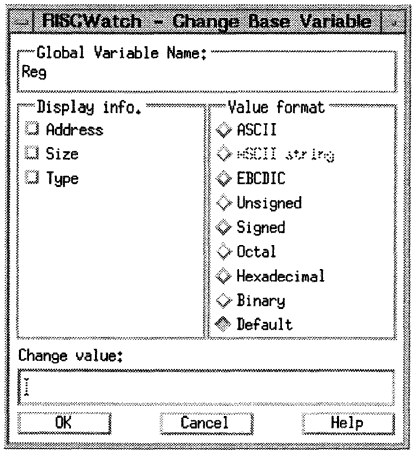


Figure 3-38. Sample Change Base Window

number is decimal 12, it will be displayed in the variable entry as '0x0000000C' if the Hexadecimal format is applied. The following formats are supported: ASCII, ASCII string, EBCDIC, Unsigned, Signed, Octal, Hexadecimal, Binary, and Default. ASCII string is enabled only for types of 'char'. If selected, 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. Refer to "Displaying ASCII Strings" on page 3-48 for a detailed description of this support. Default is the format which corresponds to the type which the variable is defined as in the program.

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 Base Variable window will remain up 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.

A checkbox 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 somewhere 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 3-52 for a detailed description of this support.

Change Enum Variable

The Change Enum Variable window is used to change variable information for a variable which is an enumeration type. It consists of Variable Name field, a Display Information selection groupbox, a Value Format selection groupbox, a Change Value field and pushbuttons. The name of the selected variable appears in the name field, with the title indicating whether it is a local or global variable.

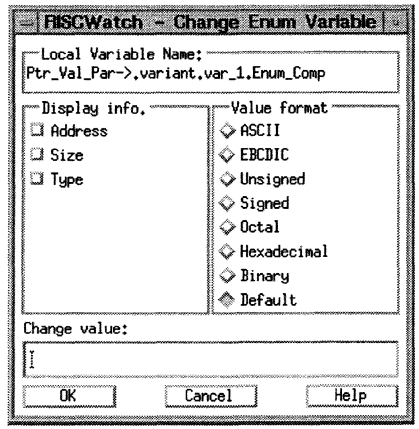


Figure 3-39. Sample Change Enum Window

The Display Information groupbox consists of three checkboxes, one each to display the Address, Size and Type information for the selected variable on the associated variable window. The initial state of the checkboxes shows the currently enabled 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 Value Format groupbox consists of a number of buttons used to change the format of the variable value in the variable entry. For example, if the value of the number is decimal 12, it will be displayed in the variable entry as '0x0000000C' if the Hexadecimal format is applied. The following formats are supported: ASCII, EBCDIC, Unsigned, Signed, Octal, Hexadecimal, Binary, and Default. Default is the format which the corresponds to the type which the variable is defined as in the program.

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 Enum window will remain up 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.

A checkbox 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 somewhere 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 3-52 for a detailed description of this support.

Change Pointer Variable

The Change Pointer Variable window is used to change variable information for a variable which is a pointer type. It consists of Variable Name field, a Display Information selection groupbox, a Variable Detail selection groupbox, a Value Format selection groupbox, a Change Value/Subrange field and pushbuttons. The name of the selected variable appears in the name field, with the title indicating whether it is a local or global variable.

The Display Information groupbox consists of three checkboxes, one each to display the Address, Size and Type information for the selected variable on the associated variable window. The initial state of the checkboxes shows the currently enabled 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 have been previously 'revealed' or expanded, whether they are currently visible or not.

The Variable Detail groupbox consists of three checkboxes: 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

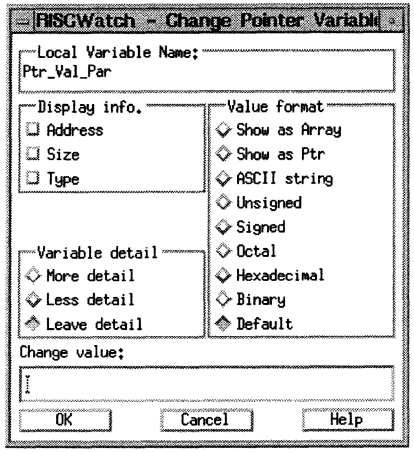


Figure 3-40. Sample Change Pointer Window

variable. The detail changes will only take effect if the changes for the window are applied. Refer to "Expanding/Contracting Variable Detail" on page 3-46 for more discussion on changing the level of detail for a variable.

The Value Format groupbox consists of a number of buttons used to change the format of the variable value in the variable entry. The following formats choices are available: Show as Array, Show as Ptr, ASCII string, EBCDIC, Unsigned, Signed, Octal, Hexadecimal, Binary, and Default. If ASCII string is selected, 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. Default is the format which corresponds to the type which the variable is defined as in the program.

Show as Array and Show as Ptr are two special format choices used to support displaying data elements pointed to by pointers. In normal operation, the pointer variable will be processed as a normal pointer type, and the format and value of the pointer can be changed as for a base type. If Show as Array is selected, the entry field will be used to enter a subrange value. The title of the entry field will be 'Change array subrange: Limits = [0,?]'. The subrange entry field will be

initialized to the current subrange value, or 0,0 if the pointer is being changed to be displayed as an array. When the Show as Array format is applied, the pointer variable entry will be expanded as an array, with the elements displayed corresponding to the entered subrange value. Now, each array element will be of the type which the pointer is pointing to, and each individual array element can be processed (expanded/contracted, value change etc.) according to the treatment for that variable type.

An example of this is if I have defined a pointer to a structure of type STRUCT_X. If this pointer is initialized to point to a region of memory containing multiple instances of STRUCT_X, and the format of this pointer is changed using Show as Array, each individual STRUCT_X instance appears as an element of an array, and can be processed using normal structure manipulation. Refer to "Handling Multiple Data Elements Referenced by a Single Pointer" on page 3-49 for a detailed description of this support.

If a pointer has been changed using Show as Array and it is selected, the Value format selected will be Show as Array, and the entry field will accept a subrange value. In this mode, however, the format and the value of the pointer itself can still be changed by selecting one of the other normal format options besides Show as Ptr. This will change the entry field to accept a pointer value, and the format selected will be applied to the variable entry. Note that the pointer will remain in Show as Array mode even after this is done. To change the pointer back to normal pointer mode, select the Show as Ptr format option. This will return the pointer back to a normal pointer type.

A checkbox 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 somewhere 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 3-52 for a detailed description of this support.

Change Struct/Union Variable

The Change Struct/Union Variable window is used to change variable information for a structure or union variable.

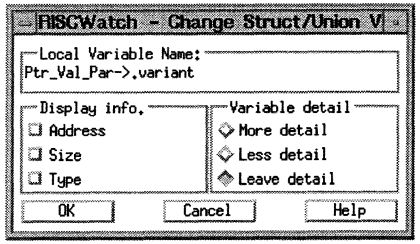


Figure 3-41. Sample Change Struct/Union Window

It consists of Variable Name field, a Display Information selection groupbox, a Variable Detail selection groupbox, and pushbuttons. The name of the selected variable appears in the name field, with the title indicating whether it is a local or global variable.

The Display Information groupbox consists of three checkboxes, one each to display the Address, Size and Type information for the selected variable on the associated variable window. The initial state of the checkboxes shows the currently enabled 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 have been previously 'revealed' or expanded, whether they are currently visible or not.

The Variable Detail groupbox consists of three checkboxes: 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 "Expanding/Contracting Variable Detail" on page 3-46 for more discussion on changing the level of detail for a variable.

A checkbox 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 somewhere 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 3-52 for a detailed description of this support.

Reading and Writing Memory

The Hardware I 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 Custom Memory screen also allows the user to customize the presentation of data even further.

See "ASCII Memory Window" on page 3-78 and "Custom Memory Window" on page 3-80 for detailed descriptions of the memory windows. "Memory Access Window (JTAG Target Only)" on page 3-75, "Cache Windows (JTAG Target Only)" on page 3-82, and "Translation Lookaside Buffer Window (PPC403GC Only)" on page 4-12 may also be applicable, depending on the target processor.

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 Access Window (JTAG Target Only)

The Memory Access window is used to control data and instruction cache updating during memory reads and writes. This window is displayed by selecting the Memory I Access option of the menubar's Hardware pulldown choice.

If caching is disabled via the appropriate hardware registers (DCCR/ICCR for PowerPC 400Series, HID0 for PowerPC 6xx), reads and writes from/to memory will directly reflect the contents of physical memory.

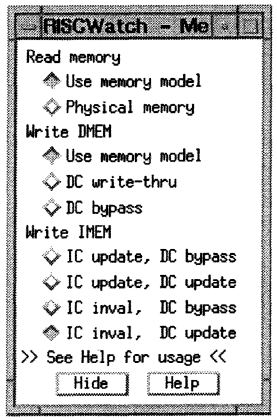


Figure 3-42. Sample Memory Access Window

If the processor is configured to control data and instruction caching, a memory model is said to have been established for how this data and code are being accessed. Once a memory model has been established, reads and writes to/from memory will provide data and/or code that is 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 Access 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 Access 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 udpate 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 Access window options and exactly how they function:

1. Write DMEM	Coherency	D-Cache	I-Cache	Physical Memory
Use memory model	Yes	Note 1	No	Note 2
DC write-thru	Yes	Note 1	No	Yes
DC bypass	No	No	No	Yes
2. Write IMEM	**Coherency**	**D-Cache**	**I-Cache**	**Physical Memory**
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)
2. Physical memory written if D-Cache disabled (via DCCR for PowerPC 400Series, HID0 for PowerPC 6xx)
3. Coherent if D-Cache disabled (via DCCR for PowerPC 400Series, HID0 for PowerPC 6xx)

4. I-Cache updated if enabled (via ICCR for PowerPC 400Series, HID0 for PowerPC 6xx)
5. I-Cache line invalidated

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 I ASCII option of the menubar's Hardware pull-down choice. What follows is a description of this window's functionality.

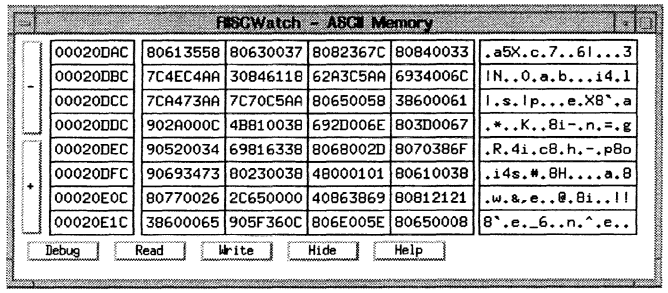


Figure 3-43. Sample ASCII Memory Window

- **Page Up/Down buttons**

The page up and page down buttons are located along the left hand edge of the ASCII Memory window. These buttons are used to page through memory. Clicking on a page button alters the display address by one screen's worth of data. To display a given address, use the address entry scheme described in the Address fields section.

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

- **Address fields**

The address fields of the ASCII Memory window are used to display data anywhere within the configured range of the processor. The address fields are located in a column adjacent to the page up/down buttons. To display any part of memory, simply use the mouse to place the cursor in any one of the address fields, type in the desired address and press the Enter key.

- **Data fields**

The data fields of the ASCII Memory window are 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 fields per display line with each field displaying four bytes of data.

To alter any of these data values, simply use the mouse to place the cursor in any one of the data fields, type in the desired data and press the Enter key to write the data field to the processor memory. Changed data will not be written to the processor unless the cursor is in the data field that was changed when the Enter key is pressed. To change multiple data fields and have all displayed data written back to processor memory, use the Write button.

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 fields**

The ASCII fields of the ASCII Memory window are used to display data read from the processor as well as alter this data so that it may be written back. The ASCII fields are located in a column along the right hand side of the window. Each ASCII field contains sixteen (16) ASCII characters that represent the data bytes in the data fields.

To alter any of these data values, simply use the mouse to place the cursor in any one of the ASCII fields, type in the desired data and press the Enter key to write the ASCII field to the processor memory. Changed data will not be written to the processor unless the edit cursor is in the data field that was changed when the Enter key is pressed.

- **Debug button**

The Debug button is used to bring up the Assembly Debug window. This window is used to read, alter and write processor memory as assembly opcodes and text.

- **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.

- **Write button**

The Write button is used to write the contents of all the data fields to processor memory. This allows multiple data or address fields to be edited and then all written to the processor using one memory write.

Custom Memory Window

The Custom Memory window allows memory to be read, altered and written in a number of radices and word sizes. This window is displayed by selecting the Memory I Custom option of the menubar's Hardware pull-down choice. What follows is a description of this window's functionality.

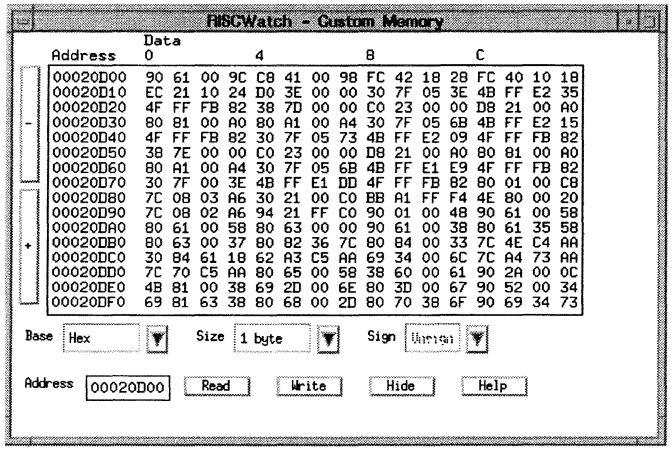


Figure 3-44. Sample Custom Memory Window

- **Page Up/Down buttons**

The page up and page down buttons are located along the left hand edge of the Custom Memory window. These buttons are used to page through memory. Clicking on a page button alters the display address by one screen's worth of data. To display a given address, use the address entry scheme described in the Address field section.

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

- **Data area**

The data fields of the Custom Memory window are used to display data read from the processor as well as alter this data so that it may be written back. The amount of data displayed is dependent on the data base and size.

The data area is composed of a column of addresses located along the left hand edge of the area and one or more columns of data. Both addresses and data may be edited. However, changing an address value will not have any effect.

To change data, simply use the mouse or arrow cursor keys to place the cursor. Edit one or more data values and then click on the Write button which will write all the values in data area to the processor memory.

- **Base selection button**

The Base selection button is used to select the radix in which data will be displayed and edited in the data area. To select a different base or radix, click on the base selection button. A list of available bases will be displayed. Place the mouse over the desired base and click the left mouse button. The data area will be redrawn to display data in the newly selected base.

The currently available bases are ASCII, binary, decimal and hexadecimal.

WARNING: Make sure all edited data has been written back to processor memory before changing the data base otherwise edited data may be lost!

- **Size selection button**

The Size selection button is used to select the size of data words that are displayed in the data area. To select a different size, click on the size selection button. A list of available sizes will be displayed. Place the mouse over the desired size and click the left mouse button. The data area will be redrawn to display the data in the newly selected size.

The currently available sizes are 1, 2 and 4 bytes.

WARNING: Make sure all edited data has been written back to processor memory before changing the data size otherwise edited data may be lost!

- **Sign selection button**

The Sign selection button is used to select whether data displayed in decimal form are shown as signed or unsigned quantities. To select a different sign, click on the sign selection button. From the displayed list, use the mouse to select either Signed or Unsigned. The data area will be redrawn to display the data in the newly selected sign.

WARNING: Make sure all edited data has been written back to processor memory before changing the data sign otherwise edited data may be lost!

- **Address field button**

The Address field is used to alter the base address at which data is displayed in the data area. This allows any section of memory to be viewed instantly without having to use the page up/down buttons repeatedly.

Use the mouse to place the edit cursor in the address field, type in the new base address and press the Enter key. The processor memory will be read starting at the new base address and the data area will be redrawn to display the memory contents just read.

- **Read button**

The Read button is used to refresh the data being displayed in the data area. When this button is activated, the processor memory at the displayed address is read and then displayed. Use this button to refresh the data or to cancel a partially edited entry in the data area.

- **Write button**

The Write button is used to write the entire contents of the data area to the processor memory. This is the only means of writing changed data area values back to processor memory.

Cache Windows (JTAG Target Only)

The Data and Instruction 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. The pulldown in the lower left corner is used to change the currently displayed way. The buttons located on the left side of the windows are used to page up and down the available cache lines for the displayed way.

For the Data Cache window, the following fields are shown:

Set	Set number (congruence class)
Address	Address tag
Word N	32-bit data cache word N
V	Valid bit
L	LRU (Least Recently Used) line in set
LK	Lock bit (401 Core and ASIC processors only)
D	Dirty bit

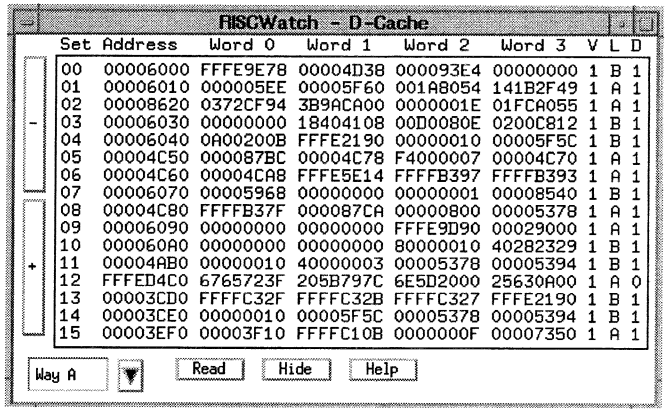


Figure 3-45. Sample Data Cache Window

For the Instruction Cache window, the following fields are shown:

Set	Set number (congruence class)
Address	Address tag
Word N	32-bit instruction cache word N
V	Valid bit
L	LRU (Least Recently Used) line in set
LK	Lock bit (401 Core and ASIC processors only)

Notes: For these cache displays, the address tag is always displayed normalized to bit 0 (MSB).

The Read button is used to force a read of the processor cache and display the latest contents.

The Hide button is used to remove this window from the screen.

Reading and Writing Registers

The Hardware I 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 only
- Floating Point Registers (FPRs): processors with FPUs

There is another class of registers, Scratch, which are kept internally by RISCWatch and can be used to hold temporary data or for calculating results.

See "Register Windows" on page 3-84 and "Register Field Windows" on page 3-86 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 only), Floating Point Registers (FPR: processors with FPUs), and Scratch registers (program defined registers used for temporary results). To bring up a particular register window, use the HardwareIRegister pulldown of the Main window menubar.

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.

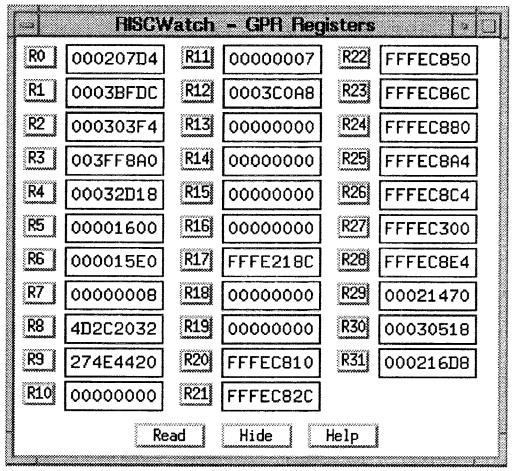


Figure 3-46. Sample Registers Window

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.

Scratch registers can be manipulated using the **expr**, **read**, **set**, and **write** commands from the command line of the Main window just like any other register. There are a total of ten Scratch registers labelled S0 - S9.

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.

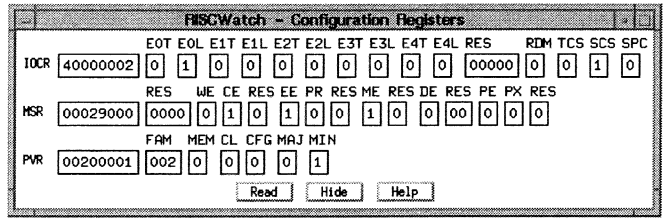


Figure 3-47. Sample Register Field Window

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. It is also possible to edit multiple field values for a single register and when the Enter key is pressed, all the field values will be used to construct the new register value which is then written to the processor.

For register fields which are only one bit in size, the mouse may be used 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 Resources

User-Defined Windows

User-Defined windows allow a RISCWatch user to create windows containing customizable register, register field, memory and disassembly entries. Using a simple syntax, ASCII files are created to define the contents of a user-defined window.

- **File Syntax**

The file used to describe a user-defined window is a simple ASCII file that is created with a text editor. The file names for such files usually, but do not have to, end in .wdf (window descriptor file).

The file can be broken up into five optional sections:

1. Title
2. Registers
3. Register fields
4. Memory
5. Disassembly

The file is composed of simple keywords and may contain comments. The keywords used to define the contents of user-defined entries are TITLE, REG,

FLD, MEM, and DIS. These keywords and their usage are explained in the sections that follow.

Even though these sections are optional, they must appear in the same sequence as listed above ("TITLE" must be first if it is used, "MEM" must be before "DIS but after any "TITLE", "REG" or "FLD", and so on).

Comments are allowed in the file and they must start with the # character as the first character on the line.

- **Window Title**

The user-defined window is given a title by using the TITLE keyword followed by the desired title window. If the TITLE keyword is used in the file, it must be the first keyword used. If no window title is assigned using the TITLE keyword, one will be assigned for it.

- **Register Entries**

Register entires are used to place registers in the user-defined window. Up to three (3) registers may be placed on a single line using the REG keyword. Simply follow the REG keyword with the names of up to any three (3) valid processor register names. The names of registers that can be used in a user-defined window correspond to the buttons on an associated register window. There may be up to ten (10) register entries for a single user-defined window.

- **Register Field Entries**

Register field entries are used to place register fields in the user-defined window. One register field is allowed per FLD keyword. In other words, there can only be one register field entry per line in a user-defined window. Simply follow the FLD keyword with the name of a valid processor register name. The names of registers that can be used in a user-defined window correspond to the buttons on an associated register window. There may be up to ten (10) register field entries for a single user-defined window.

- **Memory Entries**

Memory entries are used to place memory data in the user-defined window. Memory can be displayed as words, half-words, or bytes by using the MEM, MEMH, or MEMB keywords, respectively. A maximum of four words (MEM), eight half-words (MEMH), or sixteen bytes (MEMB) can be placed on a single line.

A memory entry consists of the memory keyword (MEM, MEMH, or MEMB) followed by the address of memory to be displayed, followed by the number of elements (words, half-words or bytes based on the memory keyword), followed by an optional label (maximum eight characters) to give the line of memory. There may be up to ten memory entries for a single user-defined window.

The leftmost field of each memory line is the label field. If a label is specified as part of a memory entry it will initially appear in the label field on the window,

otherwise the label field will be blank. Once the window is up, the label field for any memory entry can be updated by selecting the field with the mouse and typing in the new label name.

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 (ie., word, half-word, or byte).

- **Disassembly Entries**

Disassembly entries are used to place disassembly text in the user-defined window. Up to ten (10) disassembly entries per DIS keyword may be placed in a user-defined window. The DIS keyword is followed by the address of memory to be disassembled which is followed by the number of words (1-10) to be displayed. There may be up to ten (10) disassembly entries for a single user-defined window.

Creating the Window

A user-defined window is created by using the User-Def Win entry of the User-Defined menu of the Utilities 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.

Example

The following example illustrates the use of the user-defined window file syntax:

```
TITLE My Window

# Let's add some registers we use a lot
REG R0 R1 R2
REG IAR SRR0 SRR1

# Add a register field
FLD MSR

# Add some memory definitions
MEM  0x000305AC 4 INSTRS
MEM  0x00050000 2 GLOBDATA
MEMB 0x000C0000 2 IODEV1,2
MEMH 0x000C0002 1 IODEV3
MEMB 0x000C0008 6 IODEV4-9

# Disassemble some memory
DIS  0x000305AC 4
```

When coded as above, the window file will produce the window shown in Figure 3-48 below.

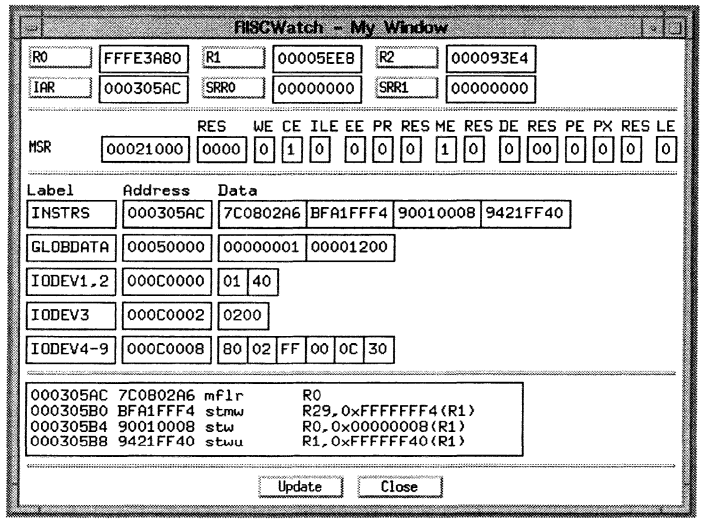


Figure 3-48. Sample User-Defined Window

A sample window descriptor file is included with the software installation of RISCWatch and is titled **rwppc.wdf**.

User-Defined Buttons

User-defined buttons allow a RISCWatch user to create a window containing buttons which will execute one or more specified commands. Using a simple

syntax, ASCII files are created to define the buttons and the commands they will execute when activated.

- **File Syntax**

The file used to describe user-defined buttons is a simple ASCII file that is created with a text editor. The file names for such files usually, but do not have to, end in .btn. The file is composed of a single keyword followed by a button title and the button commands.

Comments are allowed in the file and they must start with the # character. Comments may be on a line by themselves or follow a BUTTON title or a button command.

- **Button Entry**

A button entry is used to define a button, its title and the commands it will execute when activated. A button entry is started with the BUTTON keyword which is immediately followed by the button's title. This title will be displayed within the button when the button definition is loaded into the user interface.

The lines that follow consist of the commands that will be executed when the button is activated by the user. Each button entry may have up to 50 commands. These commands may be any command that would normally be entered on the command line of the RISCWatch Main window.

The button entry definition ends with the start of another button entry, or end of file.

Example

The following example is used to illustrate the use of the user-defined button file syntax:

```
# Let's define some buttons for things we use a lot
# One to read the IAR
BUTTON Read IAR
read iar
# One to reset the processor core
BUTTON Reset core
reset core
# One to load and run our favorite program
BUTTON load 'n run
load bin program.bin
run
```


When coded as above, the window file will produce the following screen:

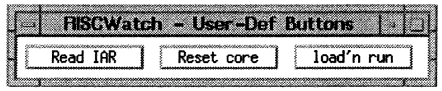


Figure 3-49. Sample User-Defined Buttons Window

A sample button definition file is included with the software installation of RISCWatch and is titled **rwppc.btn**.

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, **rwppc.cmd**, may be used to perform a series of commands which would normally be entered on the command line whenever RISCWatch is started, to help set up the debugging environment and/or specific processor facilities.

Every time RISCWatch is started in graphical user interface mode, it attempts to locate **rwppc.cmd** in the current directory. If it is found, it will execute it accordingly. If the **rwppc.cmd** file is not found in this directory, RISCWatch looks

for it in the directory specified by the RW_DIR variable in the environment file. If it is found there, it will be executed.

This scheme allows individuals to create their own startup command files by placing it in their own directories. This also allows one startup command file to be placed in the install directory (specified by RW_DIR) so that everyone will execute it whenever RISCWatch is started.

A third option allows for the individual users to use their own **rwppc.cmd** files in their directories and still execute the **rwppc.cmd** in the install directory by placing a command similar to the following in their **rwppc.cmd** files:

```
exec /usr/rwppc/rwppc.cmd
```

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:

delay	Delays command file execution for the specified number of seconds.
end	Forces the immediate termination of the command file.
parms	Specifies a parameter variable list for the command file. See "Command File Parameters" on page 3-96 for details.
print/fprint	Takes the contents of the command after the print keyword and prints them in the host window. See the **fprint** 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.

- **if-then**

  ```
  if (expression)
    block
  endif
  ```

- **if-then-else**

  ```
  if (expression)
    block
  elseif
    block
  endif
  ```

- **while**

  ```
  while (expression)
    block
  endwhile
  ```

  Note: The **while()** construct cannot be nested. A while loop **cannot** contain another while or do-while.

- **do-while**

  ```
  do
    block
  while (expression)
  ```

  Note: The do-while() construct cannot be nested. A do-while cannot contain another do-while or while.

Where:

block	Represents one or more RISCWatch commands.
expression	Composed of either a mathematical or logical expression. See the **set** command for a detailed description of RISCWatch expression syntax. Most expressions take the form (argument operator argument) Arguments can be references to reigsters, 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 >.

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 mathematcial value and this value will indicate FALSE if equal to zero and TRUE all other times.

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
proc_stopped	Returns 1 if the processor (JTAG) or process (non-JTAG) is in the stopped state, else returns 0
run_timeout	Returns 1 if the processor/process was stopped due to a run timeout since the **run** 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.
rw_cmd_error	Returns 1 if the last executed RISCWatch command caused an error to be generated, else returns 0. This value is cleared on program start and is reset every time a command issued. After the command is completed, this value will remain valid until the next command is issued.
rw_prog_error	Returns 1 if any executed RISCWatch command has caused an error to be generated, else returns 0. This value is cleared on program start and its value is never cleared once it is set.
stop_timeout	Returns 1 if the processor/process was stopped due to a stop timeout since the **stop** 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 file name
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 one a value of 0, 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 file name 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 **print/fprint** command only.
$ERRORS	Contains the number of program errors generated since RISCWatch was started. This may be used in a **set** expression or a **print/fprint** command.
$FILESIZE	Contains the number of bytes loaded from the last successful **load** command. This may be used in a **set** expression or a **print/fprint** command.
$TIME	Contains the current clock time. The format of this pseudo-variable is hour:minute:second. This may be used in a **print/fprint** command only.
$TIMER	Contains the current timer value. See the timer command for details. This may be used in a **set** expression or a **print/fprint** command.

Command File Programming Example

The following is an example that uses command file programming logic to set a scratch register based on the value of the IAR. In the example the value at memory address location 0xFFFF8000 is added to the contents of GPR0 and compared to the IAR. If the IAR is greater then this value, scratch register S1 is set to indicate this fact; otherwise it is cleared.

```
if (IAR > R0 + (0xFFFF8000))
    set S1 = 1
elseif
    set S1 = 0
endif
```

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 is specified at program startup, RISCWatch does not bring up the graphical user interface but it does execute each of the commands in the file just as if it were being executed from within RISCWatch.

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

To run a command file at program startup type in the following at the shell prompt:

rwppc command_file

Where:

command_file	The name of the command file to be executed. For example: test.cmd
step	Runs the command file in single-step mode. This option is only valid when executing a command file from the user interface. See "Command File Single-Step Window" on page 3-98 for more information on running a command file using single-step mode.

Command File Single-Step Window

The Command File Single-Step 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.

- **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.

- **Cursor window**

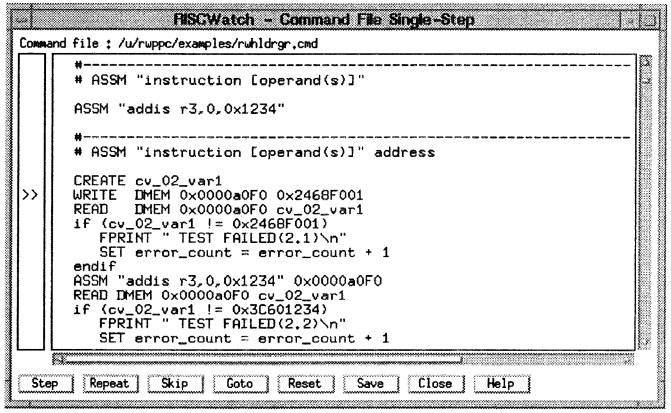


Figure 3-50. Sample Command File Single-Step Window

The Cursor window is used to display a cursor that indicates the next line of the command file that will be executed if the Step button were to be pressed. As commands are executed, the cursor will move to the next executable line, skipping blank and comment lines.

- **Text window**

The Text window is used to display the contents of the command file. When the Single-Step window is first brought up, the contents of the command file will be read and placed in this window. Since the Single-Step window is a constant size, vertical and horizontal scroll bars are provided to help with viewing the command file.

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.

- **Repeat button**

The Repeat button is used to execute the last executed command.

- **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.

- **Goto button**

The Goto button is used to skip execution of one or more instructions. To select the line to "goto", simply use the mouse to highlight a word on that line and click on the Goto button. This function may be used to skip forwards of backwards in the command file. All commands between the current line and the goto line will be skipped; they will not be executed.

- **Reset button**

The Reset button is used to reset the execution of the command file to the first command in the command file. The Text window will be scrolled to the top and the command cursor will be placed next to the first executable command of the file.

- **Save button**

If the contents of the Text window are changed and these changes are to be saved, click on the Save button. This will bring up a file selection dialog box from which an existing or new command file may be specified to save the Text window contents to.

- **Close button**

The Close button is used to remove the Command File Single-Step window from the interface. Be advised that any changes made to the Text window that have not been saved will be lost!

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.

For PowerPC 400Series devices, see also "Processor Status Window (400Series JTAG Only)" on page 4-14.

Processor Reset Window (JTAG Target 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 processors the Core and Chip resets are equivalent. They will reset the processor and soft stop at address 0xFFF00100. Also, the System reset will reset the processor and run from address 0xFFF00100.

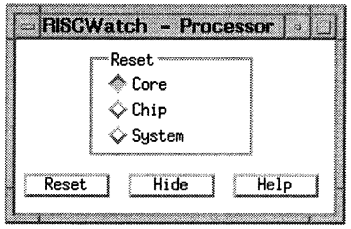


Figure 3-51. Sample Processor Reset Window

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.

WARNING: To ensure that RISCWatch maintains an accurate status of chip conditions, the processor should be reset using this window. Avoid using the contact switch on the evaluation board to reset the processor unless a reset via RISCWatch is not possible. If this contact switch is used to reset the processor, RISCWatch will not be able to detect a change in the processor running/stopped status. While this should not prove to be a dangerous condition, confusing information may be displayed if the reset started the processor running, but RISCWatch still thinks the processor is stopped. Status indicators in one or more windows may indicate that the processor is stopped, when in fact it is running due to the asynchronous nature of the run operation via the contact switch reset.

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 SAVE_LAYOUT variable in the environment resources file, **rwppc.env**, is set to YES, 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 Utilities|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 Utilities|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.

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 Utilities|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.

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 methodolgy 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 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 5-75 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 5-74 in the Command Reference for a detailed description.

The Log Comment window, shown in Figure 3-37 below, is displayed by using the Utilities|Logging|Comment pulldown of the Main window menubar.

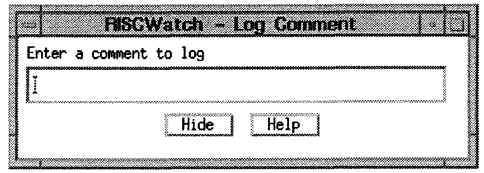


Figure 3-52. Sample Log Comment Window

Type the text to be entered in the log file in the edit field and then press the Enter key. Select the Hide button to remove this window from the screen. Select the Help button to bring up help information for this window.

Viewing Log Files

The contents of a log file can be viewed using any ASCII file editor or the RISCWatch file viewer. To view a log file while running RISCWatch, select the Utilities|Logging|View option of the Main window menubar. This will display a file selection dialog of all available log files.

Log files are named according to the day on which they were created and for which they therefore contain entries. For example, a log file created on August 19 will be called RW0819.LOG.

Simply select the desired log file from the files listed and a viewer window will be opened to display the contents of the selected log file.

Shell Command Window (Non-PC Host Only)

The Shell Command window is available for a non-PC host only. This window is used to pass command strings to the native operating system for execution. Figure 3-37 shows the window that is displayed by using the Utilities|Shell pulldown of the Main window menubar.

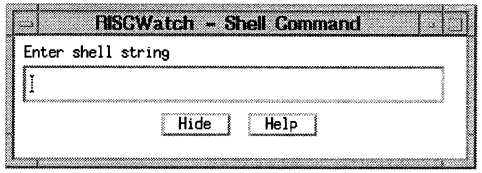


Figure 3-53. Sample Shell Command Window

Type the command to be executed in the edit field and then press the Enter key. Select the Hide 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 file name 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.

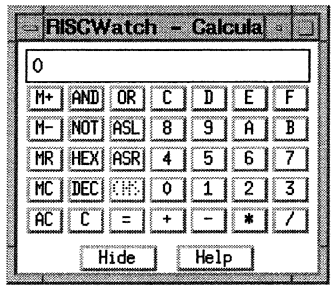


Figure 3-54. Sample Calculator Window

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

Profiler Window

The Profiler window is used to monitor the progress of a running **profile** command and to review the data collected after the run is stopped.

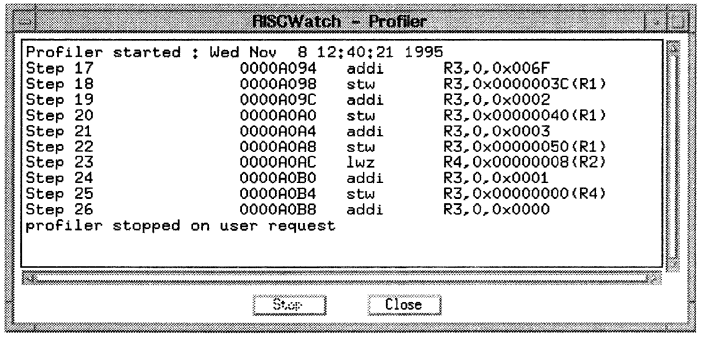


Figure 3-55. Sample Profiler Window

The Profiler window consists of a large text area and two action buttons. The text area is used to display the results collected from a profiling session. A profiling session is configured and run using the **profile** command.

Profiling of code is accomplished by single-stepping the processor, one assembly instruction at a time. After each step is taken, the requested profiling conditions set up by the user will be checked. If the conditions specified are met, the requested profile information will be gathered for display in the window.

The entry that is displayed will consist of the step number, the current IAR value, disassembled opcode at the IAR memory address and a listing of requested profile data.

Once a profile run command is given, this window will appear and display the requested profile data as it is being collected. This data will usually consist of register and/or memory data.

To stop the profiling session, the **profile** stop command is used or the Stop button located along the bottom of the window is selected. Once stopped, the scroll bar attached to the text area can be used to view the collected data.

The Close button is used to remove the window from the screen after the collected data has been viewed.

Online Help

RISCWatch provides extensive online help. Most windows contain a Help button which is used to bring up context-sensitive help. Help is also available by using the Help pulldown of the Main window menubar. Once a help window is displayed, the Search option can be used to browse a list of all available help topics.

Using the Help pulldown of the Main window, it is possible to display help information for the following topics :

- The RISCWatch program version number
- User modifiable Application Notes file
- RISCWatch Command Syntax
- RISCWatch Technical Support phone numbers
- Processor Instruction Sets
- Processor Register and Field Definitions

Since the help viewer invoked varies depending on the host platform, 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 and select the How to Use Help option for a description of the resources available.

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 4-6 "Compound Trigger/Trace Window (400Series Only)" on page 4-9
Memory Resources	"Translation Lookaside Buffer Window (PPC403GC Only)" on page 4-12
Processor Resources	"Processor Status Window (400Series JTAG Only)" on page 4-14

PPC403GC Implementation Notes

RISCWatch support for the Memory Management Unit (MMU) of the PPC403GC 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 (PPC403GC Only)" on page 4-12, or within the program itself that cause nonconformance to these conditions will produce unpredictable results.

Refer to the *PPC403GC Embedded Controller User's Guide* in "Related IBM Publications" on page xxvi for more information regarding the operating characteristics of the MMU.

Managing Hardware Breakpoints and Trace Events

See "Using Hardware Breakpoints" on page 3-40 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 400. 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.

RISCTrace requires a JTAG Ethernet processor probe target which has trace capabilities. The RISCWatch controls for RISCTrace appears 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 trigger event occurs, RISCTrace automatically collects the trace information and reconstruct and format it. The formatted trace is saved in the file **rwppc.trc** and displayed in a view 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 is appended to the beginning of the formatted trace information in the saved file.

Selecting the Abort 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.

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 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.


```
# RISCTrace : Trace Output File
# DATE      : Tue Aug 13 17:53:05 1996


# TRACE TRIGGER SETTINGS   : IAC1 occurring 01 times
> TRACE TRIGGER EVENT CYCLE: 00000

#       Total Cycle/          (optional )
# Line  Cycle Instr Address   (+F_Offset)  Disassembly
# ----- ----- ----- --------------------- -----------

$ FUNCTION: main  START_ADDR: 0x0000A078  FILE: demo1.c  PROGRAM: ./demo
  00001 00000 00001 0x0000A0A0(+0x000028) stw       R3,0x00000040(R1)
  00002 00000
  00003 00001 00001 0x0000A0A4(+0x00002C) addi      R3,0,0x0003
  00004 00002 00011 0x0000A0A8(+0x000030) stw       R3,0x00000050(R1)
  00005 00013 00001 0x0000A0AC(+0x000034) lwz       R4,0x00000008(R2)
                                          ...
          *** Entries removed for figure display purposes ***
                                          ...
  00051 00087 00002 0x0000A0E4(+0x00006C) cmpwi     CR1,R3,0x0005
  00052 00089 00002 0x0000A0E8(+0x000070) blt       CR1,$+0xFFFFFFE4
  00053 00089 00002 0x0000A0EC(+0x000074) bl        $+0x000000EC

$ FUNCTION: routine5  START_ADDR: 0x0000A1D8  FILE: demo3.c  PROGRAM: ./
  00054 00091 00001 0x0000A1D8(+0x000000) stwu      R1,0xFFFFFFC0(R1)
  00055 00092 00013 0x0000A1DC(+0x000004) stw       R3,0x00000058(R1)
  00056 00105 00001 0x0000A1E0(+0x000008) lwz       R4,0x00000008(R2)
  00057 00106 00001 0x0000A1E4(+0x00000C) addi      R3,0,0x0005
  00058 00107 00001 0x0000A1E8(+0x000010) stw       R3,0x00000000(R4)
  00059 00108 00001 0x0000A1EC(+0x000014) addic     R1,R1,0x0040
  00060 00108 00001 0x0000A1F0(+0x000018) blr

$ FUNCTION: main  START_ADDR: 0x0000A078  FILE: demo1.c  PROGRAM: ./demo
  00061 00109 00001 0x0000A0F0(+0x000078) cror      31,31,31
  00062 00109 00001 0x0000A0F4(+0x00007C) bl        $+0x0000008C

$ FUNCTION: routine4  START_ADDR: 0x0000A180  FILE: demo1.c  PROGRAM: ./
  00063 00110 00001 0x0000A180(+0x000000) stwu      R1,0xFFFFFFC0(R1)
  00064 00111 00015 0x0000A184(+0x000004) stw       R3,0x00000058(R1)
  00065 00126 00001 0x0000A188(+0x000008) addis     R3,0,0x4444
                                          ...
          *** Entries removed for figure display purposes ***
                                          ...
$ FUNCTION: ?  START_ADDR: ?  FILE: ?  PROGRAM: ?
  00126 00261 00001 0xFFFE0700            sync
  00127 00262 00001 0xFFFE0704            stw       R1,0x00000034(0)
  00128 00263 00012 0xFFFE0708            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. A line preceded by the special character '>' follows, containing the information 'TRACE TRIGGER EVENT CYCLE : cycle', where *cycle* is a decimal number indicating the cycle number at which the trace trigger occurred. Also, the trace output entry immediately following the entry for the instruction at which the trace was triggered contains the comment:
   '** STATUS: Trigger event **' at the far right of the entry.

   This entry also has the same Cycle value as the instruction entry preceding it at which the trace was triggered.

7. The trace header (preceded by the comment character '#') follows:

```
#        Total  Cycle/
# Line   Cycle  Instr    Address  (+F_Offset)  Disassembly
# ------ ------- -------- ------------------------- ----------------
```

8. The trace entries follow next. Each field of the entry is aligned below the field name in the header, as described below:

Line	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.
+F_Offset	The optional offset from the beginning of the function. This only appears if there is program symbol information loaded for the function containing this executed instruction. Otherwise it is blank.

   All hex numbers are preceded by the characters '0x'. Otherwise, numbers are decimal.

9. 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

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 '?'.

10. 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.

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.

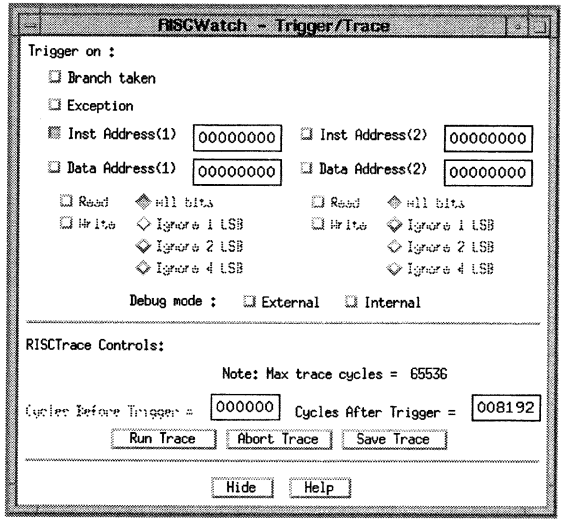


Figure 4-3. Sample Trigger/Trace Window with Trace Supported

- **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**

There are two Instruction Address Compare events that can be set. 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.

- **Data Address Compare events**

There are two Data Address Compare events that can be set. A Data Address Compare event trigger is enabled and disabled according to the state of its Read and Write check boxes. 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 enabling the respective check box. The Data Address Compare events also allow for byte, half-word and word masking of the data address on compares through the use of the All bits/Ignore 1 LSB/Ignore 2 LSB buttons. Use the mouse to select the appropriate button for the specified data address.

- **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.

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 maximum number of cycles which can be traced is shown above the controls used to specify the cycle count(s) for the trace.

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, a 'cycles before trigger' count (the count of cycles before the trigger event occurs) is always zero and cannot be altered. A '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, a 'cycles before trigger' count and a 'cycles after trigger' count can be both 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.

- **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.

- **Abort Trace button**

Selecting the Abort 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.

- **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.

Compound Trigger/Trace Window (400Series 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 4-6 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.

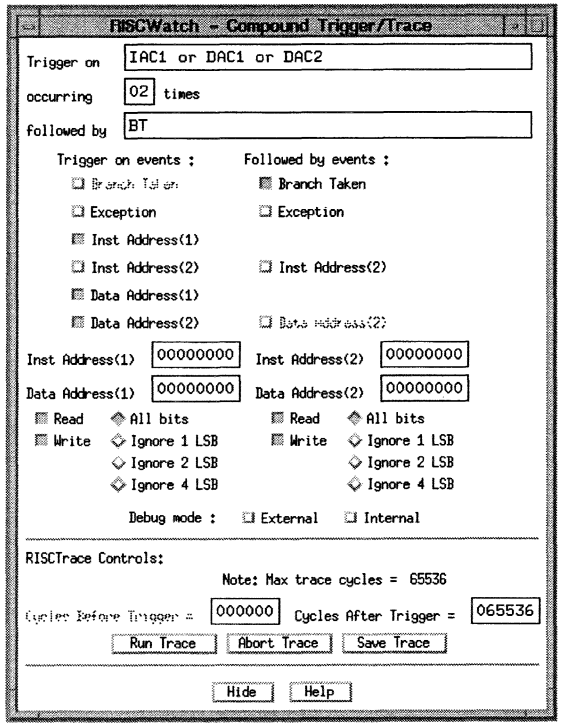


Figure 4-4. Sample Compound Trigger/Trace Window with Trace Supported

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 checkboxes under the "Trigger on events" heading. These checkboxes 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 checkboxes 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 4-8 for further information.

Memory Resources

See "Reading and Writing Memory" on page 3-75 for a general description of RISCWatch features and windows for memory access.

Translation Lookaside Buffer Window (PPC403GC 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).

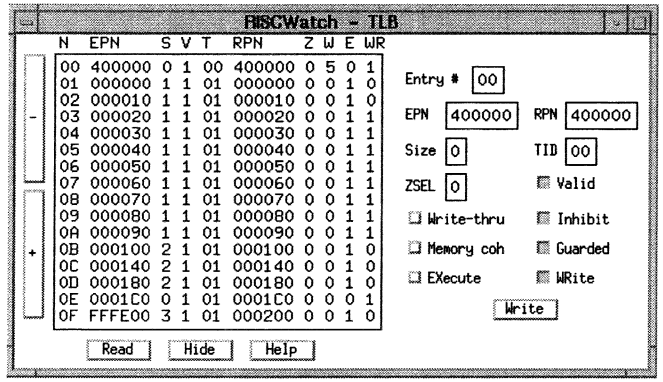


Figure 4-5. Sample TLB Window

"PPC403GC Implementation Notes" on page 4-1 provides details affecting RISCWatch support for PPC403GC TLB operations.

This window is displayed by selecting the Memory I TLB option of the menubar's Hardware pulldown choice. The left half of the TLB window displays the contents read from the TLB. The right half is used to edit a TLB entry and write it back to the TLB.

The buttons located along the far left side of the window are used to page up and down through the available TLB entries when they are clicked on.

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 :

N	entry number
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 Hide button is used to remove this window from the screen.

The right half of the window is composed of text fields and check boxes which are used to set the attributes of a TLB entry so that it can be written back. All the text fields display hexadecimal quantities.

Note: Page numbers (EPN & RPN) are always displayed normalized to bit 0 (MSB). These values should also be entered as such.

To view the contents of a particular entry, enter the TLB entry number in the text field and press Enter. The attributes for the specified entry are displayed.

Edit the text fields and set the check box states accordingly. To write the new data to the TLB, simply click on the Write button.

Processor Resources

See "Processor Reset Window (JTAG Target Only)" on page 3-100 for a description of RISCWatch options for resetting a PowerPC processor.

Processor Status Window (400Series JTAG Only)

This window is used to convey the status of important processor facilities. The first

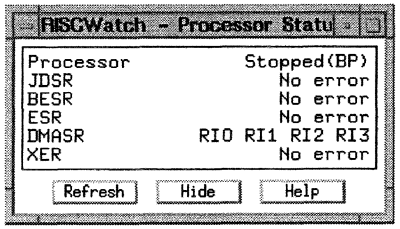


Figure 4-6. Sample Processor Status Window

line in the window is used to indicate if the processor is running or stopped. If the processor is stopped, an attempt will be made to give an explanation as to why the processor is in the stopped state.

Stopped indicators (DBSR):

BP = software breakpoint

IAC = instruction address compare

DAC = data address compare

BT = branch taken

EXC = exception

IC = instruction complete

TRAP = trap

UDE = unconditional debug event

IDE = imprecise debug event

The remaining lines in the window are used to provide the status of important processor facitlities which are provided by bits in various registers. Each of these lines is composed of a register name followed by the status for that register. If the processor is running, the lines will not be updated.

If there are no error indications for that register, the string 'No error' will be displayed. If the register contents indicate that there have been one or more errors, the register field name for the indicated error(s) are displayed.

JDSR JTAG Debug Status Register

HAS HoldAck status

IMC	Instruction machine check
IO	Illegal instruction
FP	Protection error
AE	Alignment exception
PWS	Processor in wait state
ISO	Instruction stuff overrun
BESR	Bus Error Syndrome Register
DSE	data-side error
DME	DMA error
RD	read error
WR	write error
PV	protection violation
CFG	non-configured address
BE	bus error
BTO	bus time-out
ESR	Exception Syndrome Register
IMCP	instruction machine check (protection)
IMCN	instruction machine check (non-configured)
IMCB	instruction machine check (bus error)
IMCT	instruction machine check (time-out)
PEI	program exception (illegal)
PEP	program exception (privileged)
PET	program exception (trap)
DMASR	DMA Status Register
RI0	DMA channel 0 error
RI1	DMA channel 1 error
RI2	DMA channel 2 error
RI3	DMA channel 3 error
XER	Fixed-Point Exception Register
SO	summary overflow
OV	overflow
CA	carry

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
- Examples
- Related Information

Processors Currently Supported

This release of the RISCWatch Debugger supports the following PowerPC processors and versions:

- PowerPC 401GF
- PowerPC 403GA
- PowerPC 403GB
- PowerPC 403GC
- PowerPC 602 Rev2
- PowerPC 603 Rev3
- PowerPC 603e Rev1
- PowerPC 603e Rev3
- PowerPC 603ev Rev2
- PowerPC 604 Rev3
- PowerPC 604ev Rev2

For PowerPC 6xx processors, this version of RISCWatch does not support Micro Channel or parallel port adapters for JTAG targets.

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

Reading the Syntax Diagrams

See "Syntax Diagram Conventions" on page xxv 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.

Each command description provides a table to summarize the processors, modes, hosts, and targets with which that command can be used. The combination of processors, targets (JTAG, OS Open, or ROM Monitor), and usage modes applicable to each command are indicated by bullets (•) in the appropriate table cells. Notes below the tables provide additional details of command applicability.

A sample environment table is shown below:

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•		
ROM Mon	•	•	•		

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	•

Note: TTY mode is available only on RS/6000 and Sun workstations.

All the RISCWatch Debugger commands can be used when source mode is on, except for commands restricted to command files usage or not applicable to a specific host or target.

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

address any valid memory address value (usually specified as a 32 bit hex number)

create_var any variable created with the **create** command

field_name an appropriate register field name as it appears in a Register Field window

imm_var any immediate variable created with the **assign** command

mem_var any memory variable created with the **assign** command

reg_name any valid processor register name

reg_var any register variable created with the **assign** command

value any decimal, octal or hexadecimal value

window window name, specified by one of the following keywords:

bpset	Breakpoint Select window, window showing functions with bp set
bpnotset	Breakpoint Select window, window showing functions with bp not set
break	Breakpoints window
callers	Callers window
files	Files window
functions	Functions window
globals	Globals window
locals	Locals window
osopen	OS Open window
programs	Programs window
source	Source window
varinvis	Variable Config window, window showing invisible vars
varvis	Variable Config window, window showing visible vars

Table 5-1 summarizes the syntax of the RISCWatch Debugger commands:

Table 5-1. Syntax Summary for Debugger Commands

Command	Parameters
asmstep	[value]
assign	reg_var = reg_name[.field_name]
	imm_var = value
	mem_var = (address)
assm	"assembly" [address|create_var|reg_name|reg_var]
attach	threadid|processid

Table 5-1. Syntax Summary for Debugger Commands

Command	Parameters
beep	[off|on]
bot	[window]
bp	set address
	clear address|all
	set|clear brnt|exc|icmp|trap
	set dac1r|dac1w|dac1rw|dac2r|dac2w|dac2rw address [byte|half|word|double]
	set iac1|iac2|iabr|ihw address
	clear dac1r|dac1w|dac1rw|dac2r|dac2w|dac2rw|iabr|iac1|iac2
	set [ihw]|clear at file:line|at line|in "function"
bpmode	[hw|hardware]|[sw|software]
callstep	
capture	all|ascii|dcr|debug|fpr|gpr|spr|sr|window [total] [filename]
create	create_var [= initial_value]
delay	value|create_var|imm_var
detach	
dis	value|(address)|create_var|mem_var|reg_name|reg_var
down	[lines [window]]
edit	[filename]
end	
event	enable|clear event_name
exec	command_file[{variable_list}] [step]
exit	[-f]
expr	expression

Table 5-1. Syntax Summary for Debugger Commands

Command	Parameters
	append|new filename
fctrl	close
	errors|log|status on|off
file	filename
find	[string [window]] | [$last$ window]
findb	[string [window]] | [$last$ window]
finde	[string [window]] | [$last$ window]
focus	[window]
fold	on|off
fprint	print_string
freeze	never|stop|always
funcdisp	[all_addr|all_name|dbg_addr|dbg_name]
goto	line
halt	[on|off]
hidewins	
ip	
jtagclk	[value]
kill_thread	
line	[line[window]]
linestep	

Table 5-1. Syntax Summary for Debugger Commands

Command	Parameters
	binary|bin filename address|create_var|imm_var
	dmem|imem filename [address|create_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]
load	hp filename
	image filename
	layout filename
	motorola|mot filename
	reg filename
	tektronix|tek filename
log	message
logging	[on|off]
logoff	
memchk	address|create_var|imm_var [length|create_var|imm_var]
memcopy	source|create_var|imm_var dest|create_var|imm_var length|create_var|imm_var
memfill	address|create_var|imm_var length|create_var|imm_var value
memfind	address|create_var|imm_var length|create_var|imm_var string|value
memrwait	[value]
memwwait	[value]
mode	enable|clear bdm|dedm|ftd|eidm|tdm
pagedn	[window]
pageup	[window]
parms	{var1 [, var2, ..., varN]}
print	print_string

Table 5-1. Syntax Summary for Debugger Commands

Command	Parameters
	reg_name
	address
	start iar = value
	start mem address = value
profile	start reg reg_name = value
	count value
	run [output_file]
	stop
	reset
	filename
quit	[-f]
read	address|mem_var|create_var|imm_var [create_var|reg_name|reg_var]
readb	address|mem_var|create_var|imm_var [create_var|reg_name|reg_var]
readh	address|mem_var|create_var|imm_var [create_var|reg_name|reg_var]
read	[reg] reg_name|reg_var [create_var|reg_name|reg_var]
read	regs
	off|on
record	play [filename]
	save filename
reset	core|chip|sys
restart	
retstep	
run	[timeout]

Table 5-1. Syntax Summary for Debugger Commands

Command	Parameters
	reg filename
save	layout filename
	mem filename address|create_var|imm_var bytes|create_var|imm_var
	argument [=] expression
	argument= (address)|mem_var|create_var|reg_name[.field_name|.#]|reg_var
	expression= logical|mathematical
	logical= expr_arg|expr_arg log_op expr_arg
set	mathematical= [math_op1] expr_arg [math_op2 mathematical]
	expr_arg= reg_name[.field_name|.#]|(address)|mem_var|create_var|imm_var|reg_var|value
	log_op= == != > >= < <=
	math_op1= + - ~
	math_op2= + - * / mod % & | ^ ~ << >>
shell	command
showip	
socket	retry|timeout [value]
sourcemode	on|off
srcdisp	source|mixed
	q[uery]
srchpath	set dir1 (dir2 . . . dirN)
	add dir
	c[lear]
srcline	[line]
start_thread	funcname
stop	[timeout]
stuff	opcode|*assembly*|reg_name|variable

Table 5-1. Syntax Summary for Debugger Commands

Command	Parameters
timer	start|stop
top	[window]
unload	all|filename
up	[lines[window]]
varinfo	locals|globals all|none|[addr][size][type]
varvis	locals|globals vis|invis
view	[filename]
write	dmem address|mem_var|create_var|imm_var valuelcreate_var|imm_var|reg_name|reg_var
	imem address|mem_var|create_var|imm_var valuelcreate_var|imm_var|reg_name|reg_var
writeb	dmem address|mem_var|create_var|imm_var valuelcreate_var|imm_var|reg_name|reg_var
writeh	dmem address|mem_var|create_var|imm_var valuelcreate_var|imm_var|reg_name|reg_var
write	[reg] reg_name|reg_var valuelcreate_var|imm_var|reg_name|reg_var

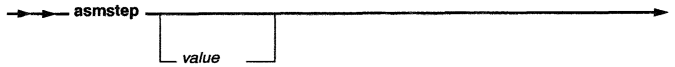
asmstep

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Note: TTY mode is available only on RS/6000 and Sun workstations.

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 dicatate 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

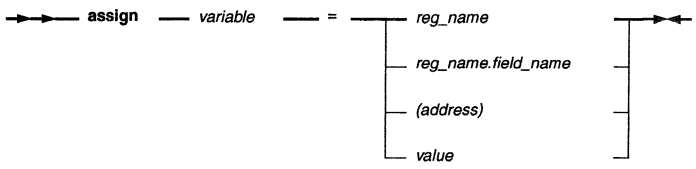
- **callstep** on page 5-25

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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. 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 octal, decimal or hexadecimal form. To assign the value of a register or field, the register or register field name 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 variables and immediate values.

Flags

value	An initial data value
(address)	The memory address to which the value of an assembled instruction is written. Note that the () characters are used to distinguish a memory address from an immediate value.

reg_name	The name of the register to which the value of an assembled instruction is written. The register must not be larger than 32 bits.
reg_name.field_name	The register name concatenated with the field name to which the value of an assembled instruction is written. The register must not be larger than 32 bits.
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

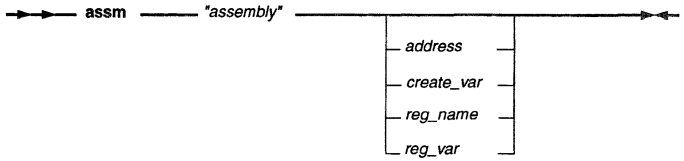
- **create** on page 5-29
- **set** on page 5-107

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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

"assembly"	A string containing a valid assembly instruction
address	The memory address to write the assembled instruction value to
create_var	Any variable created with the **create** command
reg_name	The name of a register to write the assembled instruction value to
reg_var	Any register variable created with the **assign** 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 are 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 assm_value
assm "mfspr r13,LR" assm_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.

```
assm "mfspr r13,LR" R8
```

See Also

- **dis** on page 5-33

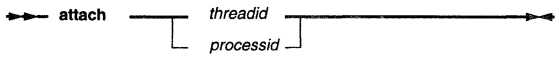
attach

	401x	403x	602	603x	604x
JTAG					
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Note: TTY mode is available only on RS/6000 and Sun workstations.

Syntax



Description

For an OS Open target:

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.

For a ROM Monitor target:

attach initializes a source mode debug session with *processid* under the ROM Monitor on a 403GA evaluation board. The *processid* must be 42 to connect with the ROM debugger. The process to be debugged must already have been loaded and be either running or stopped on a breakpoint. See the **nimgbld** utility described in the 403GA evaluation board kit user documentation for how to cause the ROM monitor to begin a debug session after an image is loaded.

Flags

threadid	The number of an existing thread
processid	The number 42, if the process is to connect with the ROM Monitor

Examples

- Attach to an existing OS Open thread.

```
attach 0x31568
```

- Attach to a process loaded on a 403GA evaluation board

```
attach 42
```

See Also

- **detach** on page 5-32
- **kill_thread** on page 5-67
- **start_thread** on page 5-119

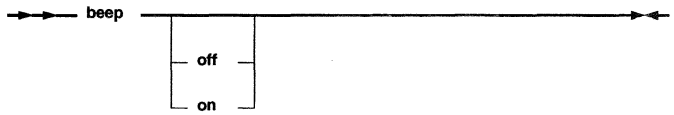
beep

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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

off Turn the program beeper off

on 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

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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 See list of window keywords in "Command Quick Reference" on page 5-2.

Examples

- Scroll to the last line of the window previously specified by this command.

```
bot
```

- Scroll to the last line of the Breakpoint window.

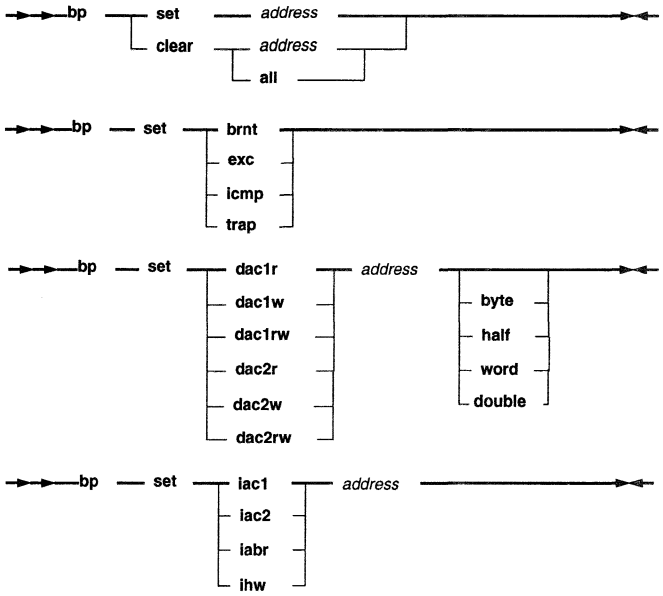
```
bot break
```

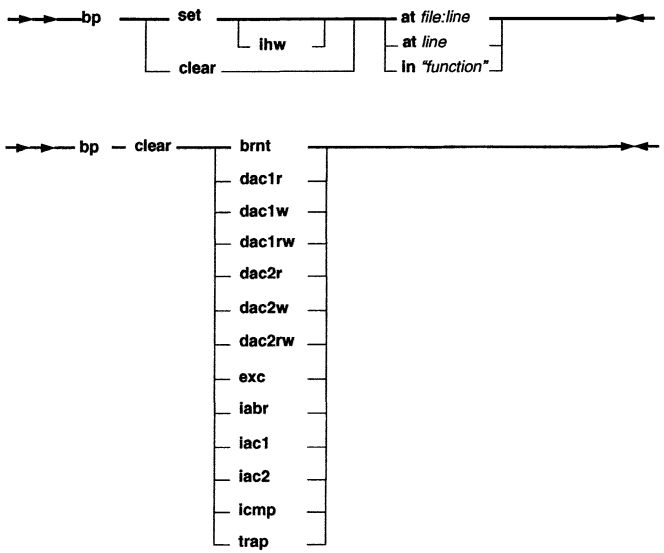
	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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 either pre-defined event names, or the ihw keyword (which uses the first available instruction breakpoint register to set an instruction breakpoint). The 'bp clear address' syntax applies to both hardware and software instruction breakpoints.

WARNING: For non-JTAG targets, only iac and dac (400Series) and iabr (PowerPC 6xx) breakpoints are recommended. For example, when connected to a ROM Monitor target, a brnt (branch taken) breakpoint will most likely stop execution while running the ROM Monitor debug code and will not stop execution in the program RISCWatch is debugging, thus hanging the debug session.

Flags

clear	Clear one or all breakpoints
set	Set a breakpoint
address	Address of the data or instruction where the breakpoint should be set or cleared
all	Remove all breakpoints(hardware and software)
brnt	400Series: Break on branch taken
dac1r	400Series: Break on Data Address Compare #1 Read
dac1w	400Series: Break on Data Address Compare #1 Write
dac1rw	400Series: Break on Data Address Compare #1 Read or Write
dac2r	400Series: Break on Data Address Compare #2 Read
dac2w	400Series: Break on Data Address Compare #2 Write
dac2rw	400Series: Break on Data Address Compare #2 Read or Write
exc	400Series: Break on exception
iabr	PowerPC 6xx: Break on Instruction Address Breakpoint Register
iac1	400Series: Break on Instruction Address Compare #1
iac2	400Series: Break on Instruction Address Compare #2
icmp	400Series: Break on instruction completion
trap	400Series: Break on trap
byte	400Series: An optional parameter that is used to match the exact DAC1 or DAC2 address compare value. This parameter is used by default if none other is specified.
half	400Series: An optional parameter that is used to mask off the LSB of the compare value during data address compares if the specified register is DAC1 or DAC2. Use of this parameter allows a breakpoint on any access within an aligned halfword.
word	400Series: An optional parameter that is used to mask off the two LSBs of the compare value during data address compares if the specified register is DAC1 or DAC2. Use of this parameter allows a breakpoint on any access within an aligned word.
double	400Series: An optional parameter that is used to mask off the four LSBs of the compare value during data address compares if the specified register is DAC1 or DAC2. Use of this parameter allows a breakpoint on any access within an aligned quad word (16 bytes).

ihw	An optional parameter that is used to set a hardware instruction breakpoint using the first available instruction breakpoint register for the target processor.
at	Indicates a source file line number is to follow. Used when the environment is set to 'Source Mode On'.
file:line	A source file name followed by a decimal number indicating a specific source line.
line	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 **file** command).
in	Indicates a function name is to follow. Used when the environment is set to 'Source Mode On'.
"function"	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). 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. 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.

Examples

- Set a software breakpoint at address 0xFFFFFF0.

```
bp set 0xFFFFFFF0
```

- 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.

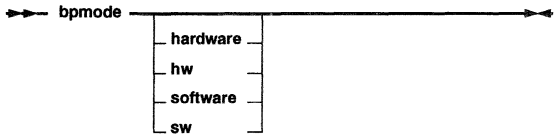
```
bp set ihw 0xFFFF00D0
```

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

Syntax



Description

bpmode is used to set or query the Breakpoint Mode used during source level 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 on the Source window and Assembly Debug window will result in a hardware breakpoint being set (if hardware facilites 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.

sw | software Set the Breakpoint Mode to software.

See Also

- "Assembly Debug Window" on page 3-26
- "Breakpoints Window" on page 3-41

- "Functions Window" on page 3-33
- "Managing Breakpoints" on page 3-39
- "Source Window" on page 3-23

callstep

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax

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 5-102

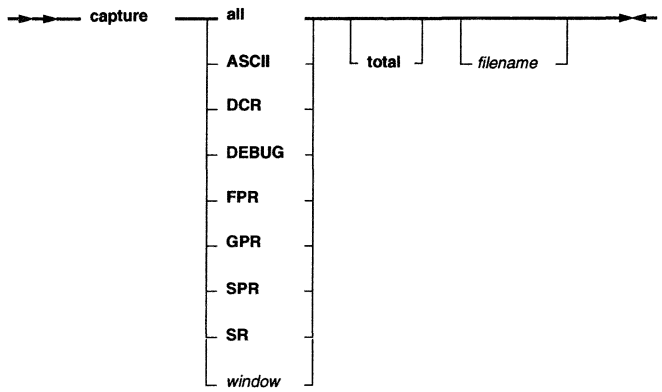
capture

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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: ASCII Memory, DCRs, Assembly Debug, FPRs, GPRs, SPRs, SRs, source level debug windows, 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 capturable windows are to be captured.

ASCII Specifies that the contents of the ASCII Memory window are to be captured.

DCR 400Series only: Specifies that the contents of the DCR Registers window are to be captured.

DEBUG Specifies that the contents of the Assembly Debug window are to be captured.

filename Specifies the name of the file to which the window capture is written.

FPR Specifies that the contents of the FPR Registers window are to be captured (processors with floating point units only)

GPR Specifies that the contents of the GPR Registers window are to be captured.

SPR Specifies that the contents of the SPR Registers window are to be captured.

SR PowerPC 6xx only: Specifies that the contents of the SR Registers window are to be captured.

total If this flag is specified, generally the entire window contents will be captured for all screens included in the window flag (the exception may be the Source window as described below). The default is to capture only the visible lines for a window.

Note: If the **all** option is specified, only the visible line 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.

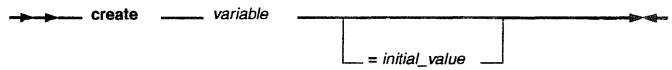
window Any of the list of window keywords in "Command Quick Reference" on page 5-2 between 'Break' and 'Source' inclusive.

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax



Description

create is used to create a variable. The variable value is stored as a signed four byte quantity. 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 the **set**, **read**, **write**, **stuff**, **dis**, **create**, **assm**, and **assign** commands.

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

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 no initial value.

```
create  cr_var2
```


- 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

- **ass ign** on page 5-11
- **set** on page 5-107

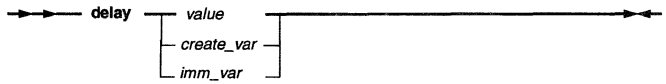
delay

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
		•			

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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
create_var	Any variable created with the **create** command
imm_var	Any immediate variable created with the **assign** command

detach

	401x	403x	602	603x	604x
JTAG					
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Note: TTY mode is available only on RS/6000 and Sun workstations.

Syntax

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.

Examples

- Detach from the thread or process being debugged.

```
detach
```

See Also

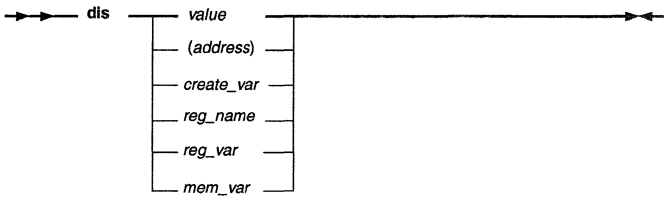
- **attach** on page 5-15
- **kill_thread** on page 5-67
- **start_thread** on page 5-119

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax



Description

dis is used to disassemble a 4-byte instruction value and print its opcode and operands in assembly code. The options for this command allow disassembly of an immediate value or of the contents of a specified processor memory location, register or user-variable.

Flags

value	Specifies an immediate numeric value
(address)	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.
create_var	Any variable created with the **create** command
mem_var	Any memory variable created with the **assign** command
reg_name	Specifies any valid register name whose value will be disassembled

reg_var	Any register variable created with the **assign** command

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
```

See Also

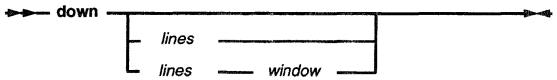
- **assm** on page 5-13

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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

lines	Specifies the number of lines to be scrolled down
window	See list of window keywords in "Command Quick Reference" on page 5-2.

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 5-127

edit

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: **edit** is available on only RS/6000 and Sun workstations.
For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax

Description

edit allows a specified file to be edited by using the editor specified by the EDITOR resource in the environment resources file.

To be able to call your favorite editor from within the program, a line must be placed in the environment resources file (**rwppc.env**) to indicate the name of the program that should be invoked to edit the file. The line in the resources file could look like this :

EDITOR = /usr/bin/vx

In this example, the editor that is to be called when the edit command is given is vx. If the editor resides in a directory that is included in the PATH environment variable, then only the name of the editor itself must be supplied. However, to call an editor that is not in the PATH, the full path name to the editor must be specified.

Flags

filename Specifies the name of the file to be edited

end

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
		•			

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

Syntax

Description

end is used to end the execution of a command file.

Examples

- End execution of the command file.

```
if (R0 != 0xFC001234)
   end
endif
```

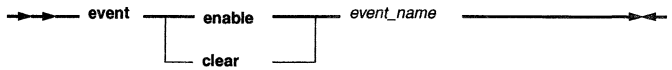
event

	401x	403x	602	603x	604x
JTAG	•	•			
OS Open					
ROM Mon					

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Note: TTY mode is available only on RS/6000 and Sun workstations.

Syntax



Description

event is used to set and clear conditions used by the processor to determine when a running program should be interrupted. It is also used to set up debug event conditions.

Flags

clear Clear the debug event

enable Enable the debug event

event_name The name of the debug event

- **brnt** Branch taken
- **dac1r** Data address compare 1 read
- **dac1w** Data address compare 1 write
- **dac1rw** Data address compare 1 read or write
- **dac2r** Data address compare 2 read
- **dac2w** Data address compare 2 write
- **dac2rw** Data address compare 2 read or write
- **exc** Exception
- **iac1** Instruction compare 1
- **iac2** Instruction compare 2
- **icmp** Instruction completion
- **trap** Trap
- **all** Used to set or clear all of the above events

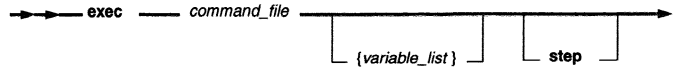
exec

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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.

Note: RISCWatch does not support nested command files while single-stepping a command file from the user interface.

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 3-94.

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 3-96 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. See "Command File Single-Step Window" on page 3-98 for more details.

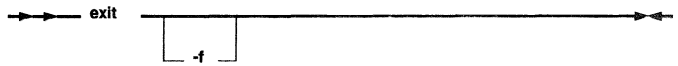
exit

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax



Description

exit terminates the program. If the processor is running when this command is given and the user interface is active, a prompt will be displayed to provide notification of the processor state and confirm the intent to terminate.

Avoid using the **exit** command in a command file. If the command file is executed while the user interface is active, execution of the **exit** 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.

The **exit** command is equivalent to the **quit** command.

Flags

-f Using this flag forces termination regardless of the processor state.

See Also

- **quit** on page 5-95

expr

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax

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	= expr_arg|expr_arg log_op expr_arg
mathematical	= [math_op1] expr_arg [math_op2 mathematical]
expr_arg	= reg_name[.field_name|.#]|(address)|immediate|variable|mem_var
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

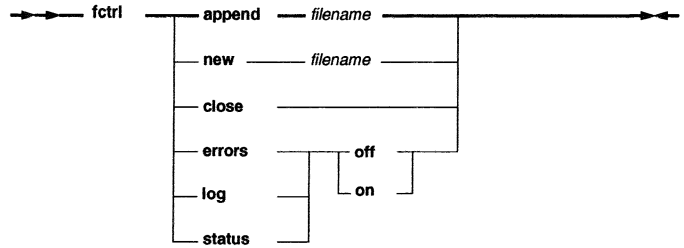
- **set** on page 5-107

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax



Description

fctrl controls access of the print files used by the **fprint** command.

Flags

append	Open a print file. If the file exists, it will be opened and all messages will be appended to the end of the file.
new	Open a print file. If the file exists, it will be erased.
close	Close the print file
errors	This flag controls whether or not program error messages are copied to the print file.
log	This flag controls whether or not log messages are copied to the print file.

status	This flag controls whether or not program status messages are copied to the print file.
off	Disables message copying
on	Enables message copying
filename	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
```

- Close an open print file.

```
fctrl close
```

See Also

- **fprint** on page 5-55
- **print** on page 5-91

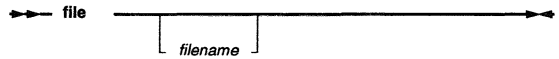
file

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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

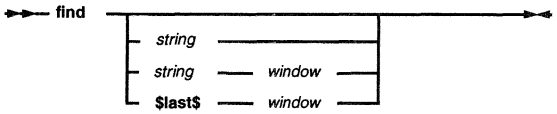
filename	Specifies the name of the source file to make current and display in the Source window

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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 3-20.

Flags

string	Sequence of characters to be found
window	See list of window keywords in "Command Quick Reference" on page 5-2.

See Also

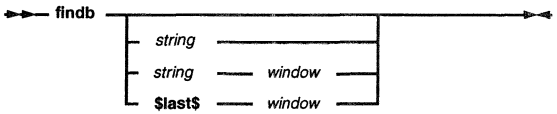
- **findb** on page 5-49
- **finde** on page 5-51
- **focus** on page 5-53

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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 3-20.

Flags

string	Sequence of characters to be found
window	See list of *window* keywords in "Command Quick Reference" on page 5-2.

See Also

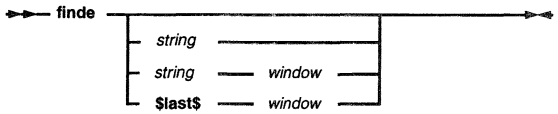
- **find** on page 5-47
- **finde** on page 5-51
- **focus** on page 5-53

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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 3-20.

Flags

string	Sequence of characters to be found
window	See list of *window* keywords in "Command Quick Reference" on page 5-2.

See Also

- **find** on page 5-47
- **findb** on page 5-49
- **focus** on page 5-53

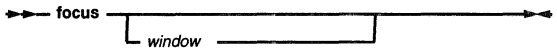
focus

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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

window See list of window keywords in "Command Quick Reference" on page 5-2.

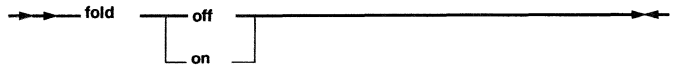
fold

	401x	403x	602	603x	604x
JTAG		•			
OS Open					
ROM Mon					

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Note: TTY mode is available only on RS/6000 and Sun workstations.

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'.

Flags

off Turns instruction folding off

on Turns instruction folding on

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax

Description

fprint prints user defineable strings to a print file that was opened with the **fctrl** 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 :

Constant	Meaning
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	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]#] cliluIxIX]

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.
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.

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 defineable 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 succesive memory locations, reads back what was written and prints the result of the comparison between the two values :

```
fctrl new test.mem                  # open a new print file
fprint "Start : ", $TIME, "\n"      # print test start time
create mem_addr = 0x0000FFFF        # start at this address
while (mem_addr < 0x00010000)       # check until this address
fprint "Addr : ", mem_addr/08X      # print address being checked
fprint "\n"                         # print newline
write dmem mem_addr 0xFFA55AFF      # write canned value to memory
read mem_addr S0                    # read back memory value
if (S0 == 0xFFA55AFF)               # if values match
fprint "Test  : PASSED\n\n"         # print success message
elseif
fprint "Test  : FAILED\n\n"         # else print error message
endif
set mem_addr = mem_addr + 1         # check next address
endwhile
fprint "End   : ", $TIME, "\n"      # print test end time
fctrl close                         # close print file
```

See Also

- **fctrl** on page 5-44
- **print** on page 5-91

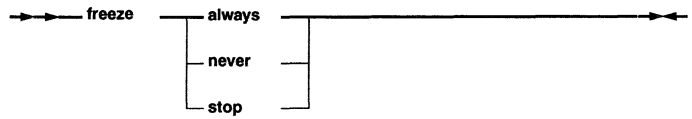
freeze

	401x	403x	602	603x	604x
JTAG	•	•			
OS Open					
ROM Mon					

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Note: TTY mode is available only on RS/6000 and Sun workstations.

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'.

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

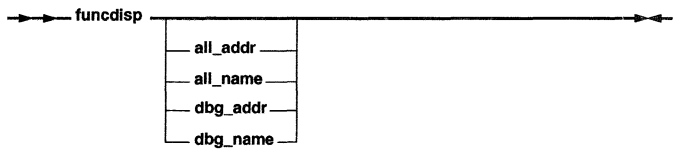
stop Forces timers to be frozen whenever the processor is stopped. Timers will remain stopped until the next run is performed.

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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.

Example

- Set the Functions window display to show all functions in the program, sorted by address

```
funcdisp all_addr
```

See Also

"Functions Window" on page 3-33

goto

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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

line Specifies the next source line to be run in the file which contains the current instruction

Example

- Change the next source line to be executed to line 100

```
goto 100
```

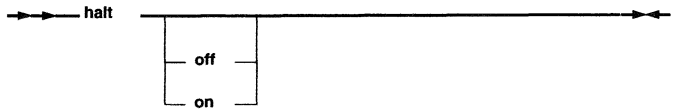

halt

	401x	403x	602	603x	604x
JTAG	•	•			
OS Open					
ROM Mon					

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Note: TTY mode is available only on RS/6000 and Sun workstations.

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

hidewins

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	

Syntax

Description

hidewins hides all the currently visible RISCWatch windows except for the Main window.

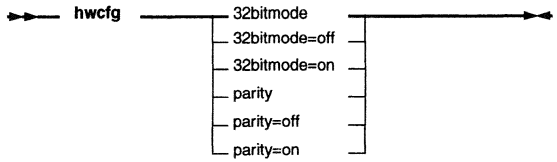
hwcfg

	401x	403x	602	603x	604x
JTAG			•	•	
OS Open					
ROM Mon					

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	•

Notes: Only JTAG Ethernet targets are supported.
32bitmode options are for 602, 603, 603e, and 603ev processors only.
Parity flags are for 603, 603e, and 603ev processors only.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax



Description

hwcfg configures different hardware options for a particular processor. Selecting a hardware option without a value to set will display the current option's setting.

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. **Note:** RISCWatch cannot automatically detect the processor's 32bitmode setting.
parity	Used to display or change RISCWatch's parity generation setting. For performance reasons, RISCWatch does not typically generate parity bits on memory accesses. However, some memory controllers may require parity generation.

ip

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax

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 5-112

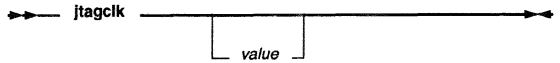
jtagclk

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open					
ROM Mon					

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	•

Notes: Only JTAG Ethernet targets are supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax



Description

jtagclk displays or sets the JTAG TCK clock speed on the RISCWatch Processor Probe. When **jtagclk** is entered without specifying *value*, the current setting is displayed.

Flags

value Specifies the clock speed to set, where:

1 = 10 MHz
2 = 5 MHz
3 = 2.5 MHz
4 = 1.25 MHz
5 = 625 KHz
6 = 312.5 KHz
7 = 156.25 KHz

kill_thread

	401x	403x	602	603x	604x
JTAG					
OS Open	•	•	•	•	•
ROM Mon					

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Note: TTY mode is available only on RS/6000 and Sun workstations.

Syntax

Description

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

Examples

- Kill the current thread

```
kill_thread
```

See Also

- **attach** on page 5-15
- **detach** on page 5-32
- **start_thread** on page 5-119

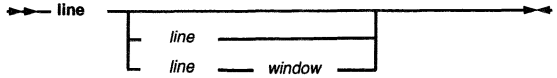
line

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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 3-20.

Flags

line	Specifies the physical line number to be scrolled to
window	See list of *window* keywords in "Command Quick Reference" on page 5-2.

linestep

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax

Description

linestep steps the program to the next source line.

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 5-10
- **callstep** on page 5-25

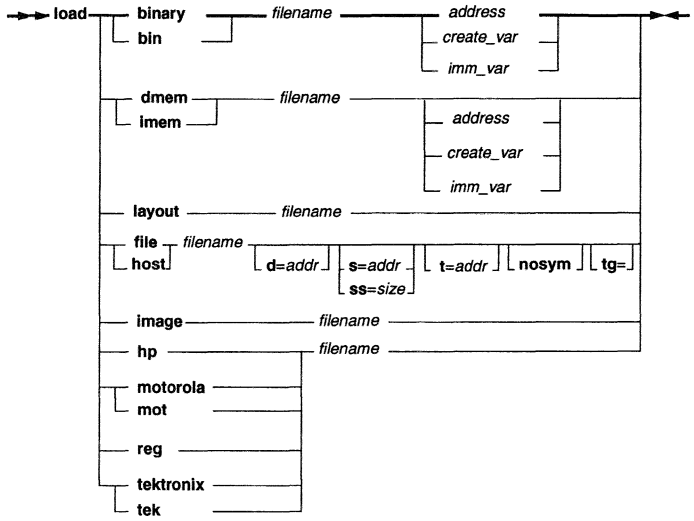
load

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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 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.

file 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

host 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.

image Loads the target system with the contents of a Boot Image file (images created with the 403GA 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 'Loading Boot and Boot Image Files' on page 3-14 for further discussions on the use of this flag.

hp Load the contents of a HP format file into data memory

motorola Load the contents of a Motorola format file into data memory

mot Same as the motorola flag

tektronix Load the contents of a TEXHEX format file into data memory

tek Same as the tektronix flag

reg This command is the complement to the **save** *reg* command and can only process those files which were generated by the **save** *reg* 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.

address Memory address to load file contents at

d= Indicates that the specified address is to be used to locate the data segment (ELF and XCOFF formats only)

s= 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 STACK_ADDR in the environment resources file will be used. THE USE OF THIS FLAG IS NOT RECOMMENDED.

t= Indicates that the specified address is to be used to locate the text segment (ELF and XCOFF formats only)

ss= 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 bss data). If this value is not specified, the STACK_SIZE in the environment resources file will be used. If STACK_SIZE is undefined, the default size of 16K bytes is used.

nosym Indicates that symbol table and string table are NOT to be loaded on the target. This applies to boot files only (images created with 403GA evaluation board support package entry code). See "Loading Boot and Boot Image Files" on page 3-18 for a discussion of boot files.

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 5-119 for more information.

filename Name of the file containing the data, in the appropriate format, to be loaded

create_var Any 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 fully qualified, then the initial search for the file will begin in that directory. If the file is not found there or the file name is not fully qualified, then the directory search will be governed by the order specified via the **srchpath** command.

See Also

- **save** on page 5-105
- **srchpath** on page 5-116
- **start_thread** on page 5-119

log

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax

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

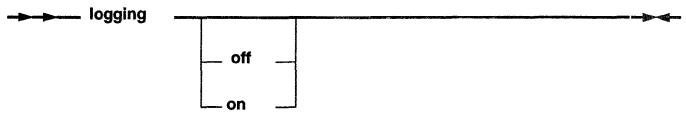
- **logging** on page 5-75

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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* argument. No messages will be written to the log file until a **logging** *on* command is given. If neither the *off* nor the *on* parameter 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

on Logging is turned on (enabled).

off Logging is turned off (disabled).

See Also

- **log** on page 5-74

logoff

	401x	403x	602	603x	604x
JTAG					
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Note: TTY mode is available only on RS/6000 and Sun workstations.

Syntax

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 3-18 for further details.

Example

- Load and execute an OS Open boot image file.

```
attach 42
load image applprog.img
logoff
```

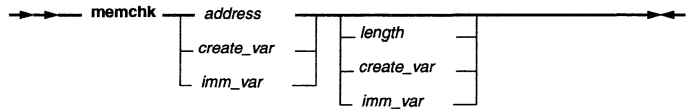

memchk

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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

address	Specifies the memory address to be checked
length	Specifies the number of sequential addresses to check. The default value is 1
create_var	A user-created variable that may be used as the memory address to be written
imm_var	An assigned user-created variable specifying an immediate value that may be used as a data memory address

See Also

- **memcopy** on page 5-79
- **memfill** on page 5-80

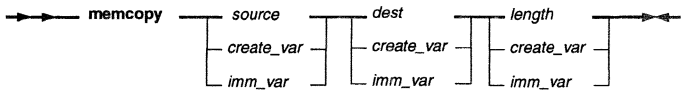
memcopy

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax



Description

memcopy 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

source	Specifies the source memory address
dest	Specifies the destination memory address
length	Specifies the number of bytes to copy
create_var	A user-created variable that may be used as the memory address to be written
imm_var	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 5-80

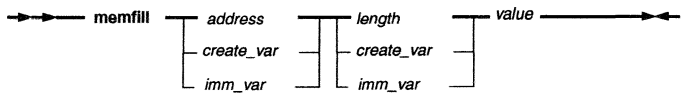
memfill

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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

address	Specifies the memory address to start the fill at
length	Specifies the number of bytes to fill
value	Specifies the value to be written
create_var	A user-created variable that may be used as a memory address or a value to be written
imm_var	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

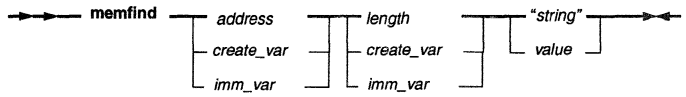
- **memchk** on page 5-78
- **memcopy** on page 5-79

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax



Description

memfind locates the address of a specified string in memory. For every occurence of the string found, a message is printed.

Flags

address	Specifies the memory address to start searching at
length	Specifies the number of bytes to search
"string"	Specifies a string of ASCII characters to be searched for
value	Specifies a string of hexadecimal characters to be searched for
create_var	A user-created variable that may be used as a memory address or a value to be written
imm_var	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
```

See Also

- **memchk** on page 5-78
- **memcopy** on page 5-79

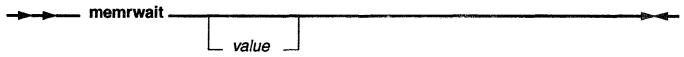
memrwait

	401x	403x	602	603x	604x
JTAG			•	•	•
OS Open					
ROM Mon					

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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 valid delay range is 0 to 10,000,000 µs (10 seconds). The initial delay is zero.

See Also

- **memwwait** on page 5-84

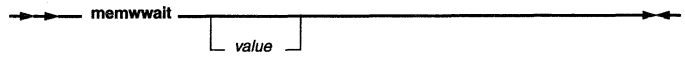
memwwait

	401x	403x	602	603x	604x
JTAG			•	•	•
OS Open					
ROM Mon					

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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 valid delay range is 0 to 10,000,000 µs (10 seconds). The initial delay is zero.

See Also

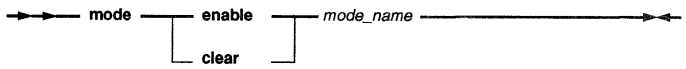
- **memrwait** on page 5-83

	401x	403x	602	603x	604x
JTAG		•			
OS Open					
ROM Mon					

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Note: TTY mode is available only on RS/6000 and Sun workstations.

Syntax



Description

mode enables or clears the debug modes of the processor.

Flags

clear	Clear the debug mode
enable	Enable the debug mode
mode_name	The name of the debug mode:
bdm	Bus status debug mode
de	Debug interrupt enable
edm	External debug mode
ftde	Freeze timers on debug event
idm	Internal debug mode
tdm	Trace status debug mode

Examples

- Enable external debug mode.

```
mode enable edm
```

- Disable debug interrupts.

```
mode clear de
```


See Also

- **event** on page 5-39

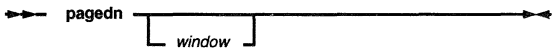
pagedn

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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

window See list of window keywords in "Command Quick Reference" on page 5-2.

See Also

- **pageup** on page 5-88

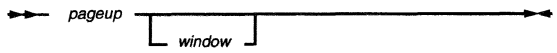
pageup

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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

window See list of window keywords in "Command Quick Reference" on page 5-2.

See Also

- **pagedn** on page 5-87

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
		•			

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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.

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 :

```
rw400 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 3-96

print

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
		•			

Notes: **print** is available only on RS/6000 and Sun workstations.
For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

Syntax

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 defineable string containing string literals, user-created variable names and the same type of expressions used in the **set** command.

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 5-44
- **fprint** on page 5-55

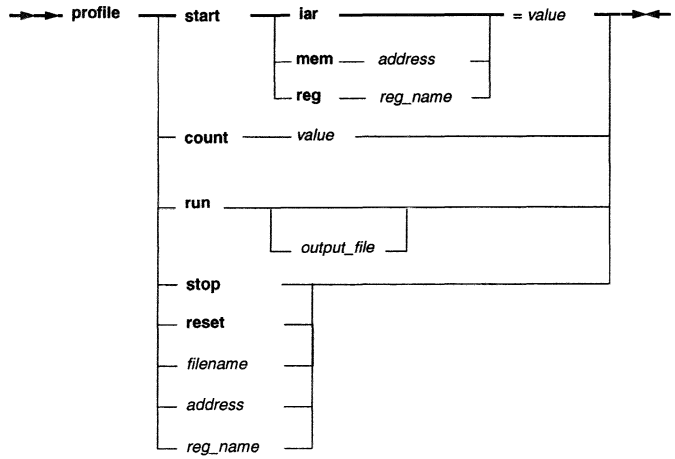
profile

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•		•	•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

Syntax



Description

profile allows for a limited amount of profiling for a program that has been loaded into the processor. The single-step feature of the processor is used to perform an invasive profiling of the loaded program.

Once a profiling session has been started, the processor is single-stepped to execute each instruction of the loaded program. After each step is taken, a step count, iar value and the decoded instruction residing at the iar address are saved to the profile output file. It is possible to tailor the profiling session to save register and memory contents to the output file as well.

Flags

start Specifies one or more conditions that must be met before profile data is saved to a file after a **profile** run command is given

- **iar** Specifies an address that must match the value of the iar before profile data is saved
- **mem** Specifies a memory location address whose contents must match the specified value before profile data is saved
- *address* Specifies the memory address for the **profile** mem command
- **reg** Specifies a register whose contents must match the specified value before profile data is saved
- *reg_name* Specifies the register name for the **profile reg** command. The register must not be larger than 32 bits.

count This specifies the number of steps for which profile data is saved after the start condition is met. After a start condition is met, each step of the program saves profile data and decrements this counter. Once this counter reaches zero (0), the start condition is reset and must be triggered again in order to start saving profile data again.

value Specifies the match value for the memory address, register or IAR, or a step count value

run Instructs the profiler to begin the profiling session. If no filename is specified after **run**, the output is saved to the file **rw400.prf**; otherwise the specified filename is used to capture the profile output.

output_file Specifies a file to save all collected profile information so that it may be viewed at a later time

stop Instructs the profiler to stop the currently running profile session

reset Instructs the profiler to remove all register and address profile commands. It is suggested that this be the first command of a profile file to ensure that the commands for a previous profile session are erased and only the desired registers and addresses are profiled.

reg_name Specifies a register whose value is to be read and saved after every step is taken. The register must not be larger than 32 bits.

address Specifies a 4-byte memory address whose value is to be read and saved after every step is taken

filename Specifies a file containing **profile** commands to be used in a profiling session. This allows the user to set up a profiling session ahead of time and removes the task of manually entering the profile commands for repeated profiler use.

quit

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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.

The **quit** command is equivalent to the **exit** command.

Flags

-f Using this flag forces termination regardless of the processor state.

See Also

- "Profiler Window" on page 3-107
- **exit** on page 5-41

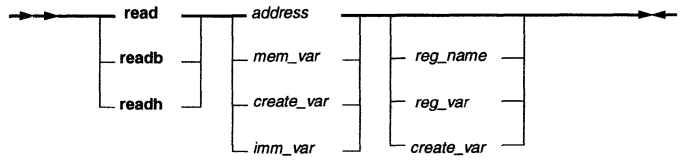
read

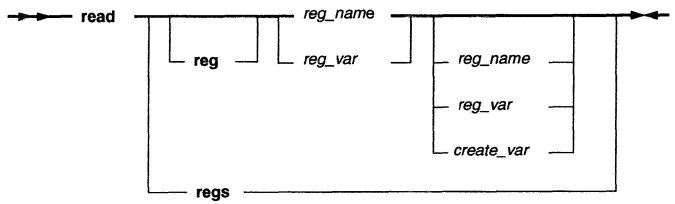
	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax





Description

read is used to read register values or four bytes of data memory. 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.

The first argument is used to indicate the object (either memory or register) to be read. If a second argument is specified, it indicates the object to be written using the value just read. If two objects are specified, the sizes of the objects must match: 32-bit reg→32-bit reg, 32-bit reg→address, 32-bit reg→user-created var, 64-bit reg→64-bit reg.

Flags

reg	An optional flag that indicates to read a processor register. Use of this flag is usually to enhance command file readability.
regs	Indicates that all processor register values are to be read
address	Specifies an immediate address value from which to read data memory
mem_var	Any memory variable created with the assign command
imm_var	An assigned user-created variable specifying an immediate value that may be used as a data memory address
reg_name	A valid processor register name to be read and/or written
reg_var	An assigned user-created variable that may be used to specifiy a processor register to be read and/or written
create_var	A created user-created variable that may be used to hold the value just read

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
```

See Also

- **write** on page 5-133

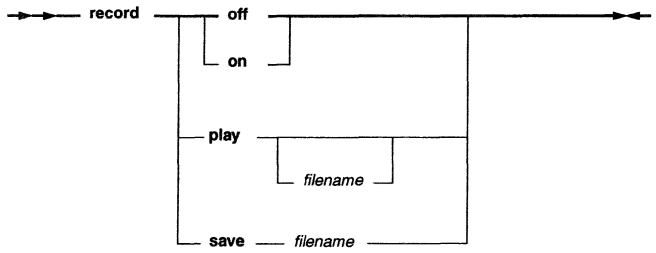
record

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•		•	•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

Syntax



Description

record is used to record commands that are entered into the main window command line interface. Having saved a sequence of commands, it is then possbile to play them back or save them to a file so that they may be played back at a later time. Up to 100 commands can be recorded at a time.

Note: No record commands will be recorded.

Flags

off	Turns off command recording
on	Turns on command recording
play	Plays back recorded commands
save	Saves recorded commands to the specified file

filename Name of the file to save recorded commands to or containing commands to be played back

If the **play** option is used with a *filename*, the commands from the file are read into memory thereby overwriting commands which might have already been recorded earlier. Be sure to save any recorded commands before exercising this option.

Once a set of commands has been loaded using the play with *filename* option, the command sequence may be played back any number of times by simply using the **play** option.

record commands are not valid within a command file.

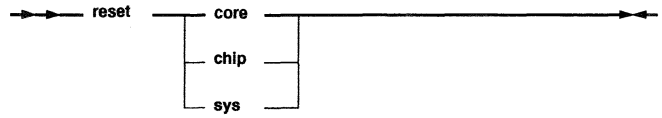
reset

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open					
ROM Mon					

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax



Description

reset resets the processor or system. For further details about processor reset, see the relevant sections in the PowerPC processor manual for the specific device being reset.

Flags

core	Reset the processor core
chip	Reset the processor core and ASIC
sys	Reset the entire system

See Also

"Processor Reset Window (JTAG Target Only)" on page 3-100

restart

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax

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 5-15
- **start_thread** on page 5-119

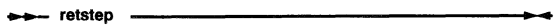
retstep

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax



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

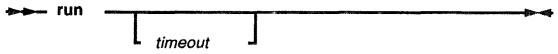
- **asmstep** on page 5-10
- **bp** on page 5-19
- **callstep** on page 5-25
- **run** on page 5-103

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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

timeout — 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
```


See Also

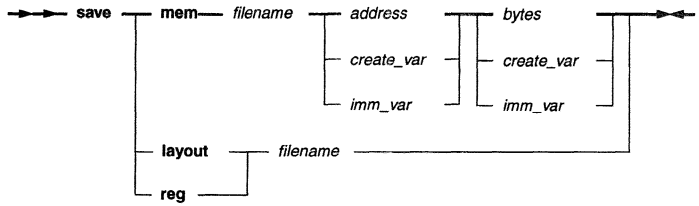
- **stop** on page 5-120

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax



Description

save is used to save a block of memory values, all processor register values, or the current window layout to a file. This command complements the **load** command and generates the files read by the **load** command.

The files generated by **save** are human-readable ASCII files that can be used to capture the state of the processor any time that it is not running. 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

mem	Specifies that a portion of processor memory is to be saved
layout	Specifies that the window layout is to be saved
reg	Specifies that all processor registers are to be saved

filename	The name of the file to save data to
address	The address of memory where to start saving data. This may also be specified using a created variable or an assigned immediate variable.
bytes	The number of memory bytes to save. This may also be specified using a created variable or an assigned immediate variable.
create_var	A created user-created variable that whose value may be used in place of the address or bytes flags.
imm_var	An assigned user-created variable specifying an immediate value that may be used in place of the address or bytes flag.

See Also

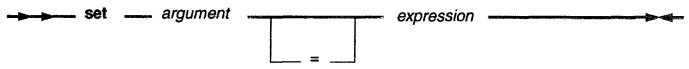
- **load** on page 5-70

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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.

The name of the variable should not conflict with names in the program that is being debugged. 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 and various operators.

The pseudo-variables $ERRORS and $TIMER may also be used in an expression.

Memory address values which appear on 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; there are no parentheses allowed to change this order
2. All operations are performed using integers; there are no floating point results stored or used for intermediate calculations
3. All operators are binary except for
   a. The operators + and - can be either binary or unary
   b. The operator ~ is always unary

The following list shows the supported operators and describes their functionality :

Operator	Function
+	arithmetic addition
-	arithmetic subtraction
*	arithmetic multiplication
/	arithmetic division
mod	arithmetic remainder or modulus
%	arithmetic remainder or modulus
&	bitwise AND
|	bitwise Inclusive OR
^	bitwise Exclusive OR
~	bitwise one's complement
<<	bitwise shift left
>>	bitwise shift right

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)|create_var|reg_name[.field_name|.#]|reg_var
expression	= logical|mathematical
logical	= expr_arg|expr_arg log_op expr_arg
mathematical	= [math_op1] expr_arg [math_op2 mathematical]
expr_arg	= reg_name[.field_name|.#]|(address)|immediate|variable|mem_var
log_op	= == != > >= < <=
math_op1	= + - ~
math_op2	= + - * / mod % & | ^ << >>
#	= ordinal bit number

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 scratch register S4 to indicate if the IAR exceeded some known memory address boundary.

```
assign max = 0xFFFFC14A
set    S4  = IAR > max
```

  In this example, if the IAR was greater than 0xFFFFC14A, scratch register 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
```


See Also

- **Command File Programming** on page 3-94

shell

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: **shell** is only available on RS/6000 and Sun workstations.
For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax

Description

shell passes a string to the environment's shell for execution. The shell, such as the Korn shell (ksh) or C shell (csh), is retrieved from the user's SHELL environment variable. The string should only contain valid host operating system commands or script file/executable program invocations.

A fork operating system call is used to create a new process for the command string to run under. To ensure correct command file processing, the forked process is allowed to finish execution before allowing control to return to the program. Therefore, care must be taken as to the commands passed to the operating system using the **shell** command.

Flags

command Name of the host command, script file, or executable

Examples

- Create a new directory and save a copy of a capture file to it.

```
shell mkdir save
shell copy rwppc.cap save
```

- The procedure above could also have been performed by executing

```
shell mkdir save;copy rwppc.cap save
```


showip

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

Syntax

Description

showip updates the entire debgger context based on the current Instruction Pointer address. All appropriate source debug windows are updated accordingly. Also, the output of the **ip** command will appear in the Main window.

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

- **ip** on page 5-65

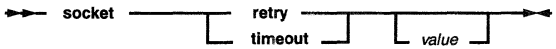
socket

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: JTAG Ethernet is the only supported JTAG target.
The retry parameter is not available for JTAG Ethernet targets.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax



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

retry The number of times that RISCWatch resends a command to a target before timing out. A negative retry amount sets an unlimited number of retries.

timeout The length of time in seconds that RISCWatch waits for information from a target before timing out.

value Number of retries or timeout value in seconds

Examples

- Set the number of retries to 10

```
socket retry 10
```

- Examine current timeout setting

```
socket timeout
```

- Set the timeout to wait for a target to 3 seconds

```
socket timeout 3
```

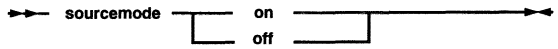

sourcemode

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax



Description

sourcemode enables or disables Source level debug functions.

When Source Mode is turned off, all the Source level debug screens which are visible are brought down, the 'Source' menubar option on the Main window is disabled, the Main window is resized to hide the Source level debug window checkboxes, and all commands relevant only to Source level debug are disabled.

When Source Mode is turned on, the 'Source' menubar option on the Main window is enabled, the Main window is resized to show the Source level debug window checkboxes, all commands relevant only to Source level debug are enabled, and the Source level debugger is reinitialized to the context of the current Instruction Pointer value.

This command is also available from the 'Source Mode' pulldown selectable from the 'Utilities' menubar option on the Main window.

Flags

on Enables source level debug features and menu options

off Disables source level debug features and menu options

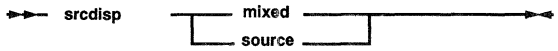
srcdisp

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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.

Flags

mixed Sets the Source window display to show mixed source/assembly lines.

source Sets the Source window display to show source lines only.

Example

- Set the Source window display to show mixed source/asm

```
srcdisp mixed
```

See Also

"Source Window" on page 3-23

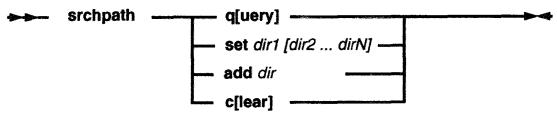
srchpath

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

Syntax



Description

srchpath determines the file search order used by the debugger to reference source files and executables. Initially, the debugger attempts to find the file exactly as it was presented by the action performed. For example, for the **load file** command, it would attempt to find the file exactly as it was typed in the command line. Or in the case of a single click on an entry in the Files window, it would attempt to locate the file as displayed in the window.

If it is not found, 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 search. 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 /home, and typed /usr/rwppc/rwppc to start RISCWatch, the current directory would be /home.

Windows platform — 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

q[uery]	Shows current directory search setting in main I/O command status window
set	Sets the search path to the directories listed, in the order that they are entered. Note this deletes any previous setting.
add	Adds a directory to the search path at the end of the current setting
c[lear]	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/stevewin/sandbox /u/mandzak/lib
            /u/kburke/test
```

The search path order for source and executables is set to

1. As entered by the action
2. /u/stevewin/sandbox
3. /u/mandzak/lib directory, and if still not found,
4. /u/kburke/test.
5. Current directory

- 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

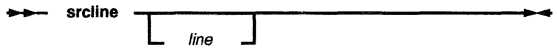
- **Environment Resources** on page 3-5
- **load** on page 5-70

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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 3-20.

Flags

line	Specifies the source line number to scroll to

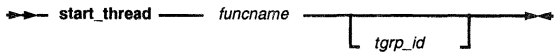
start_thread

	401x	403x	602	603x	604x
JTAG					
OS Open	•	•	•	•	•
ROM Mon					

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Note: TTY mode is available only on RS/6000 and Sun workstations.

Syntax



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

funcname	Name of function to be started
tgrp_id	ID of thread group for *funcname*

Examples

- Schedule a specified thread to be queued:

```
start_thread routine1
```

See Also

- **attach** on page 5-15
- **detach** on page 5-32
- **kill_thread** on page 5-67
- **load** on page 5-70

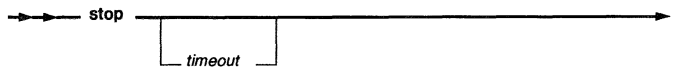
stop

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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

timeout	Specifies the number of seconds to wait before sending the **stop** command to the processor/process

See Also

- **run** on page 5-103

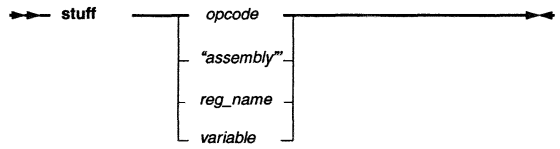
stuff

	401x	403x	602	603x	604x
JTAG					
OS Open	•	•			
ROM Mon					

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Note: TTY mode is available only on RS/6000 and Sun workstations.

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 **assm** 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-create 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

opcode An immediate machine instruction value to be stuffed

assembly A valid assembly instruction string enclosed in quotation marks to be assembled and then stuffed

reg_name The name of a register whose contents are to be read and then stuffed. The register must not be larger than 32 bits.

variable The name of a user-created variable whose contents are to be read and then stuffed

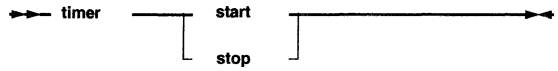
timer

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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/fprint** command. It may also be referenced in a **set** expression.

Flags

start If the timer is stopped, this flag starts it running. If the timer is running, it updates the $TIMER program variable so that it may be printed while leaving the timer running.

stop Stops the timer and saves the time elapsed since the **start** was given into the $TIMER program variable

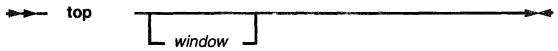
top

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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

window See list of window keywords in "Command Quick Reference" on page 5-2.

See Also

- **bot** on page 5-18

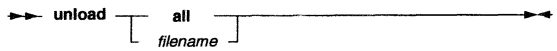
unload

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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

all Unloads all programs currently loaded in the debugger

filename Specifies program to be unloaded. If unqualified, the file unloaded will be determined by the **srchpath** settings currently in effect.

See Also

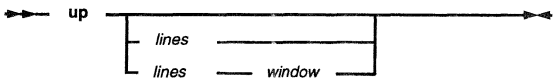
- **load** on page 5-70
- **srchpath** on page 5-116

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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

lines	Specifies the number of lines to be scrolled up in *window*
window	See list of window keywords in "Command Quick Reference" on page 5-2.

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

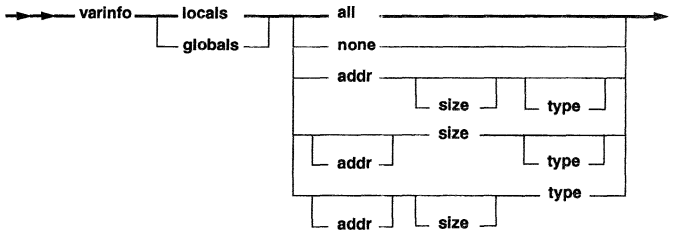
- **down** on page 5-35

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

Syntax



Description

varinfo changes the Local or Global variable window display to show type, address, and size information displayed for each visible variable. Any combination of **addr**, **size** and **type** can be specified. This is the same capability provided by the Display Information groupboxes on the Variable Configuration window and each Change Variable window.

Flags

locals	Specifies Locals variable window
globals	Specifies Globals variable window
all	Shows the address, size and type for each variable

none	Shows no address, size and type information for each variable
addr	Shows the address of each variable
size	Shows the size of each variable
type	Shows the type of each variable

Example

- Set the Locals window display to show address and type information for each visible variable

```
varinfo locals addr type
```

See Also

"Variable Windows" on page 3-60

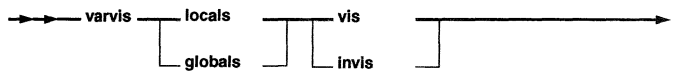
varvis

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•		•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

Syntax



Description

varvis changes the visibility of variables on the Locals or 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

locals	Specifies Locals variable window
globals	Specifies Globals variable window
vis	Make all variables visible
invis	Make all variables invisible

Example

- Set the Globals window display to show all variables

```
varvis globals vis
```

See Also

"Variable Configuration" on page 3-64

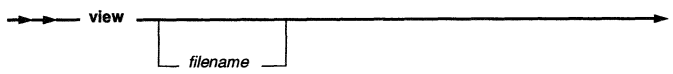
view

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	

Note: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.

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 font and size in the display are adjustable using the menubar at the top of the window.

This command is equivalent to using the View option of the File pull-down menu.

See Also

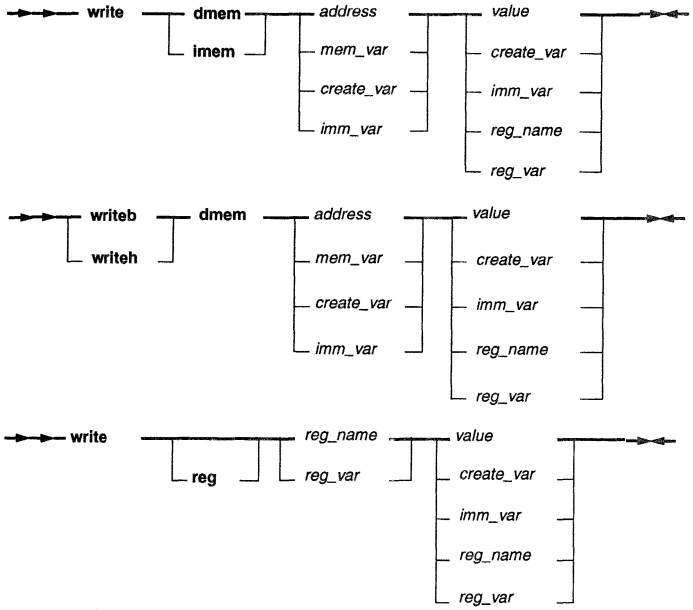
- **edit** on page 5-37

	401x	403x	602	603x	604x
JTAG	•	•	•	•	•
OS Open	•	•	•	•	•
ROM Mon	•	•	•	•	•

Modes	Cmd Line	Cmd File	Source Mode Off	Source Mode On	TTY
	•	•	•	•	•

Notes: For PowerPC 6xx processors, Micro Channel and parallel port adapters for JTAG targets are not supported.
TTY mode is available only on RS/6000 and Sun workstations.

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, 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.

write reg Write a new value to a register.

write dmem Write a new value to a data memory location. Up to four (4) bytes of data can be written to a valid address.

writeb dmem Write a new value to a data memory location. One (1) byte of data can be written to a valid address.

writeh dmem Write a new value to a data memory location. Two (2) bytes of data are written to a valid address.

write imem Write a new value to an instruction memory location This command ensures that the contents of the instruction cache is updated along with memory. All instruction addresses must fall on word boundaries.

When data is written using the imem option a DCBST and ICBI instruction are executed at every memory address that is written. This ensures that the data gets to physical memory and that the instruction cache contents is correct.

Note: The sizes of the specified source and destination arguments must match: 32-bit reg→32-bit reg, 32-bit reg→address, 32-bit reg→user-created var, 64-bit reg→64-bit reg. If a 64-bit register is specified as the destination of the write, immediate values can be used to write the register in two forms. If the value is preceded by '0x' or '0X', the hex data is copied bit for bit into the register (with leading zeroes appended if necessary). If the value is not preceded by '0x' or '0X', the value will be converted to floating point format, with valid scientific notation also accepted: '1.23456e+002)'.

Flags

dmem Write to a data memory address

imem Write to an instruction memory address

reg An optional parameter indicating a write to an architected register in the chip

address Specifies an immediate value which represents the memory location to be written

mem_var Any memory variable created with the **assign** command

create_var A created user-created variable that may be used as the memory address to be written or as the value to be written

imm_var	An assigned user-created variable specifying an immediate value that may be used as as the memory address to be written or as the value to be written
reg_name	A valid processor register name to be read and/or written
reg_var	An assigned user-created variable that may be used to specifiy a processor register to be read and/or written
value	An immediate value to be written to the specified memory address or register

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 0xFFFFFFF0.

```
write dmem 0xFFFFFFF0 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
```

Note: Any of the **write dmem** examples are also valid for **write imem**, just replace the word **dmem** in each example to **imem**.

See Also

- **read** on page 5-96

write

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 that this version of RISCWatch supports, see "About This Book" on page xxiii.

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.

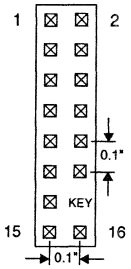


Figure A-1. JTAG Header Connector (top view)

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 (Micro Channel adapter, parallel port adapter, or 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 connections for the PowerPC 400Series processors, and Table A-2 provides information on the PowerPC 6xx connections.

Table A-1. PowerPC 400Series JTAG Interface Connections and Resistors

Header Pin #	I/O	Signal Name	PowerPC Processor Pin # 403GA, 403GC	403GB	Board Resistor[1]
1	Out	TDO	16	12	
2		No Connect			
3	In	TDI	8	13	10KΩ PU
4		No Connect			
5		No Connect			
6		+POWER[2]			1KΩ SR[3]
7	In	TCK	6	18	10KΩ PU
8		No Connect			
9	In	TMS	7	21	10KΩ PU
10		No Connect			
11	In	$\overline{\text{HALT}}$	9	22	10KΩ PU
12		No Connect			
13		No Connect			
14		KEY			
15		No Connect			
16		GND			

[1]PU = pullup, PD = pulldown, SR = series

[2]The +POWER signal is sourced from the target development board and is used as a reference signal. It should be the power signal being supplied to the processor (either +3.3V or +5V). It does not supply power to the RISCWatch hardware.

[3]This 1K ohm series resistor provides short circuit current limiting protection only. If the resistor is present, it should be 1K ohm or less.

Table A-2. PowerPC 6xx JTAG Interface Connections and Resistors

Header Pin #	I/O	Signal Name	PowerPC Processor Pin[1] 602	603, 603e	604	Board Resistor[2]
1	Out	TDO	28	198	248	
2		No Connect				
3	In	TDI			250	10KΩ PU
		TDI	24	199		1KΩ PD
4	In	$\overline{\text{TRST}}$	27	202	253	10KΩ PU
5		No Connect				
6		+POWER[3]				1KΩ SR[4]
7	In	TCK	26	201	252	10KΩ PU
8		No Connect				
9	In	TMS	25	200	251	10KΩ PU
10		No Connect				
11	In	$\overline{\text{SRESET}}$	10	189	236	10KΩ PU
12		No Connect				
13	In	$\overline{\text{HRESET}}$	9	214	265	10KΩ PU
14		KEY				
15	Out	$\overline{\text{CHECKSTOP}}$		216		10KΩ PU
		$\overline{\text{CKSTP\_OUT}}$	3		267	
16		GND				
N/A	In	$\overline{\text{QACK}}$[5]	142	235		1KΩ PD
		L2_TEST_CLK	21	203	254	10KΩ PU
		L1_TEST_CLK	22	204	255	
		$\overline{\text{LSSD\_MODE}}$	23	205	256	
		ARRAY_WR			271	

[1]Pin numbers for PQFP packages

[2]PU = pullup, PD = pulldown, SR = series

[3]The +POWER signal is sourced from the target development board and is used as a reference signal. It should be the power signal being supplied to the processor (either +3.3V or +5V). It does not supply power to the RISCWatch hardware.

[4]This 1K ohm series resistor provides short circuit current limiting protection only. If the resistor is present, it should be 1K ohm or less.

[5]If the target development board does not use this signal, the board must have a 1K ohm pulldown resistor connected to this pin. This signal allows the processor to enter the soft stop state.

The $\overline{\text{HRESET}}$, $\overline{\text{SRESET}}$, and $\overline{\text{TRST}}$ signals from the RISCWatch Processor Interface Assembly connector must be logically ORed with the $\overline{\text{HRESET}}$, $\overline{\text{SRESET}}$, and $\overline{\text{TRST}}$ 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 Target Only)" on page 3-100.

RISCTrace Status Port (400Series JTAG Processor Probe Only)

A 20-pin male 2x10 header connector is recommended for connecting to the RISCTrace Status Port of a PowerPC 400Series processor. The connector outline, shown in Figure A-2, and the signal descriptions in Table A-3 match the

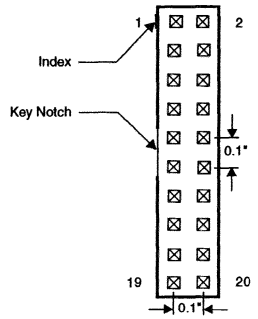


Figure A-2. RISCTrace Header (top view)

requirements of RISCTrace, when used with the RISCWatch processor probe with RISCTrace option. The connector for RISCTrace should be placed as close as possible to the processor to insure signal integrity.

The seven Trace Status signals, TS0:6, are active-high outputs from the PPC403GA and PPC403GC processors. These signals should be sampled on the rising edge of the processor clock.

Table A-3 describes the assignment of signals TS0:6 and the system clock (SysClk) output to the header pins:

Table A-3. RISCTrace Header Pin Description

Pin Number	Signal Name	Pin Number	Signal Name
1	No Connect	11	No Connect
2	No Connect	12	No Connect
3	SysClk	13	TS0
4	No Connect	14	TS1
5	No Connect	15	TS2
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

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 use either a serial (SLIP) or 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 xxvi.

Index

Numerics

A

B

C

D