

WIND RIVER

Wind River[®] Workbench for On-Chip Debugging

BOARD BRING-UP GUIDE FOR ARM AND XSCALE

2.6.1

Copyright © 2007 Wind River Systems, Inc.

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

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

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

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

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

Corporate Headquarters

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

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

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

<http://www.windriver.com>

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

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

Contents

1	Introduction	1
2	On-Chip Debugging	3
3	Board Bring-Up	7
	3.1 Goals and Objectives	7
	3.2 Sequence of Events	7
4	OCD Connections	9
	4.1 Debug Connections	9
	4.2 Creating a Target Connection	9
5	Tool Configuration	15
	5.1 Introduction	15
	5.2 Tool Configuration	16
	5.2.1 Clock Rate	16
	5.2.2 Drive TRESET Line	17
	5.2.3 Monitor Target Reset	17

5.2.4	Emulator HRESET Control	18
5.2.5	CPU Reset Type	18
5.2.6	Saving Changes	19
6	Board Initialization	21
6.1	Introduction	21
6.2	Background Mode	22
6.2.1	The IN Command	22
6.2.2	Set Verbose On	22
6.3	The INN Command	25
6.4	Registers	25
6.4.1	Downloading a Register File	26
6.4.2	Enabling and Disabling Register Groups	27
6.4.3	Modifying Registers Manually	28
7	Verifying Hardware	31
7.1	Introduction	31
7.2	Setting a Workspace	31
7.3	Diagnostic Functions	32
7.3.1	Simple RAM Test	32
7.3.2	Full RAM Tests	34
7.3.3	CRC Calculation	35
7.3.4	Scope Tests	35
	Read From Location	35
	Write To Location	35
	Write and Complement	36
	Write Rotating Value	36
	Write Then Read	36

7.3.5	Bus Tests	36
	Address Bus Test	36
	Data Bus Test	36
8	Testing Memory	37
8.1	Introduction	37
8.2	Testing Memory	37
8.2.1	Stepping an Instruction	38
8.2.2	Running Code	40
8.2.3	Setting Software Breakpoints	42
8.2.4	Setting Hardware Breakpoints	44
9	Debugging in RAM	47
9.1	Overview	47
9.2	Creating a Target Connection	47
9.3	Creating a Project	48
9.4	Downloading Code and Symbol Information	52
9.5	Debugging Code in RAM	55
9.5.1	Monitoring Processes	56
9.5.2	Stepping Through Code	56
9.5.3	Setting a Software Breakpoint	57
9.5.4	Running a Program	58
9.5.5	Stepping Through a Program	59
9.5.6	Setting a Hardware Breakpoint	59
9.5.7	Disconnecting and Terminating Processes	63

10	Programming Flash Memory	65
10.1	Introduction	65
10.2	Testing Flash Workspace	66
	Reading and Writing Memory	66
10.3	Getting Started	67
10.4	Flash Configuration Tab	68
10.4.1	Selecting a Flash Driver	68
10.4.2	Configuring Flash Memory Bounds	69
10.4.3	Configuring RAM Workspace	70
10.4.4	Setting Timeouts	70
10.5	Flash Programming Tab	70
10.5.1	Erasing and Programming Flash	71
10.5.2	Verifying Flash Contents	72
10.5.3	Running a Pre- or Post-Flash Script	72
10.5.4	Selecting Flash Sectors for Erasure	72
10.5.5	Manually Configuring Flash Memory Erasure Bounds	72
10.5.6	Adding Files	73
10.5.7	Removing Files	73
10.5.8	Converting Files To Wind River Flash Binary Format	73
10.5.9	Setting The Download Offset Of A File	75
10.5.10	Enabling A File For Download	76
10.6	Flash Memory/Diagnostics Tab	76
10.6.1	Viewing Memory	77
10.6.2	Running Diagnostic Tests	77
11	Debugging in ROM	79
11.1	Overview	79

11.2	Getting Started	80
11.3	Debugging in ROM	80
11.3.1	Stepping Through Boot Code	83
11.3.2	Setting Hardware Breakpoints	83
A	Pins Mapped to Common Signals	85
A.1	Introduction	85
A.2	ARM Processors -- JTAG	85
B	Internal Breakpoint Capabilities	87
	Line Breakpoints	88
	Expression Breakpoints	88
	Hardware Breakpoints	88
	Importing Breakpoints	90
	Exporting Breakpoints	91
	Refreshing Breakpoints	91
	Disabling Breakpoints	91
	Removing Breakpoints	91
C	Pin Terminations	93
C.1	JTAG Pin Terminations	93
C.1.1	ARM 14-Pin JTAG Connector	93
C.1.2	ARM 20-Pin JTAG Connector	95
C.1.3	ARMX (XScale) 20-Pin JTAG Connector	96
	Index	99

1

Introduction

This document describes procedures for using Wind River Workbench with the Wind River Probe and Wind River ICE emulators to bring up a target board, from the first power-up through running and debugging application code.

This document includes the following chapters:

1. *Introduction* - Introduces the document.
2. *On-Chip Debugging* - Describes of the theory of on-chip debugging.
3. *Board Bring-Up* - Provides an overview of board bring-up procedure.
4. *OCD Connections* - Describes making an OCD connection to a target using a JTAG or BDM port.
5. *Tool Configuration* - Describes hardware-specific configuration options for Wind River emulators.
6. *Board Initialization* - Describes how to use Wind River emulators to initialize the target hardware.
7. *Verifying Hardware* - Describes how to use Wind River Workbench to run hardware diagnostics on your target.
8. *Testing Memory* - Describes how to use Wind River emulators to suspend CPU operations and force the target into background mode.
9. *Debugging in RAM* - Describes how to create a project, download code and symbol information, set software breakpoints, and step through code.
10. *Programming Flash Memory* - Describes working with flash memory on your target.

11. Debugging in ROM - Describes using hardware breakpoints to debug in ROM.

A. Pins Mapped to Common Signals - Provides a mapping reference for Wind River-supported processor families.

B. Internal Breakpoint Capabilities - Provides a detailed reference for line, expression, and hardware breakpoints in Workbench.

C. Pin Terminations - provides a detailed reference of pinouts for Wind River-supported processor families.

2

On-Chip Debugging

Almost all embedded systems have hardware and software elements, which are separate but interdependent. Since embedded systems generally do not have keyboards, or any kind of user interface, debugging of their software elements must be done externally.

An older solution to this problem was the in-circuit emulator, which substituted its own internal processor for the central processing unit (CPU) of the embedded system.

However, in-circuit emulators are expensive; and since they are made by third-party vendors, there is often a long delay between a new target and a new in-circuit emulator that can attach to it. A cheaper, and more easily implemented, solution is *on-chip debugging* (OCD).

Many semiconductor manufacturers now integrate dedicated debug microcircuitry into their chips. This approach adds hardware and software debug capability to the existing JTAG or BDM ports. Since the debug operations occur on a dedicated area of the chip itself, this solution is known as on-chip debugging.

OCD combines many features of software debug monitors and in-circuit emulators. Like an in-circuit emulator, OCD provides low-level hardware access. It does not need to use target memory; it does not need a target communication channel; and, for some processors, it can edit memory and registers without halting the processor. Like a software monitor, OCD lets you set breakpoints, stop and start the CPU, step through code, examine memory, and run diagnostic tests; but unlike a software monitor, OCD does not need good hardware to run.

Software defects that cause the operating system to crash will typically cause an agent-based debug environment to fail. However, since an OCD connection is implemented in the hardware, it is not as sensitive.

An OCD connection remains active even on bad hardware. Using an OCD connection, you can download low-level software even when the target board is not functioning correctly, and the boot loader cannot run.

On-chip debugging capability varies from one processor family to another, but the provided functionality is generally similar. Most processors use the following primitives for Background Debug Mode (BDM):

Table 2-1 **OCD Primitives**

Command	Mnemonic	Description
Read Register	RDREG	Read a data register and return the value.
Write Register	WDREG	Write a value to a data register.
Read Memory	READ	Read from a memory location.
Write Memory	WRITE	Write to a memory location.
Stop Processor	BGND	Assume control of the bus and put processor in background mode.
Single Step	STEP	Step one instruction.
Resume	GO	Resume execution at the program counter's current location.

Wind River tools use these low-level OCD primitives as building blocks to create a higher level of primitives, thus allowing hardware and software verification.

OCD commands invoked while the processor is running “steal” bus cycles from the CPU in the same way a Direct Memory Access (DMA) controller does.

As the debugger reads and writes to memory and registers, it halts the CPU and restarts it. The CPU is not involved in OCD operations. The **BGND** instruction from the OCD hardware causes the CPU to halt, and the OCD hardware assumes control of chip operations. A **GO** (Resume) command flushes OCD operations and restarts the CPU.

The Wind River Probe and Wind River ICE SX tools use the on-chip debug capabilities embedded in the target processor. These tools are not true in-circuit emulators, because they do not replace the target CPU with their own internal processor. However, the functions they perform are similar, and this document will refer to them as “emulators”.

OCD has many advantages over in-circuit emulation. It is cheaper; the debug hardware is included by the silicon manufacturer, not by a third party; and unlike an in-circuit emulator, the OCD hardware does not lag behind chip releases.

When you access the OCD services on the chip, all interaction between the Wind River Probe or Wind River ICE SX and the target runs exclusively through the OCD connection. This means that your system is effective for the entire development process, even before board-level peripherals are stable.

ColdFire processors, and some older PowerPC processors (5xx and 8xx) use a dedicated BDM port for OCD operations. A more recent approach is to attach the OCD functions to the Joint Test Action Group (JTAG) interface to communicate to the target CPU, and share this interface with boundary-scan board-circuit testing. The JTAG interface follows the IEEE 1149.1 boundary-scan (JTAG/Test Interface) specification.

The JTAG interface consists of a set of five signals, three JTAG registers, and a test access port (TAP) controller. The TAP controller is typically embedded in the target microprocessor or device. The information related signals are TDI (Test Data In) and TDO (Test Data Out). The boundary-scan register chain (data) includes registers controlling the direction of the input/output drivers, as well as registers reflecting the signal value received or driven. The expectation and details of particular CPU chains are encoded directly into the emulator firmware.

Each device sharing the JTAG interface employs a serial stream of relative data. The data streams for all devices can be chained together. An associated process can scan the combined chain to extract any particular device's information.

For further information about JTAG operations, refer to the IEEE 1149.1 specification at <http://standards.ieee.org>.

Wind River emulators are non-intrusive; that is, they do not use target resources. An emulator will not affect target memory, stack space, or the flash workspace.

On-chip debug agents reside inside cache and memory management units they share the chip with, so the OCD hardware sees address and data values just like the CPU sees them. Some processor families have dedicated output signals (other than the JTAG pins) that can deliver information on the state of the processor. Combined with external hardware (such as the Wind River ICE SX, in conjunction with the Wind River Trace tool) these signals can log the real-time code execution history to a trace buffer. This data is helpful when you need to debug problems that only occur when the processor is running at full speed.

There is an industry standard, not yet widely adopted, created by the Global Embedded Processor Debug Interface Forum, formally called IEEE-ISTO 5001. For

the standard, and a good deal of further information, see
<http://www.nexus5001.org/standard.html>.

3

Board Bring-Up

[3.1 Goals and Objectives](#) 7

[3.2 Sequence of Events](#) 7

3.1 Goals and Objectives

This chapter provides a general overview of board bring-up procedure.

The goal of a board bring-up procedure is to verify the operation of a target board, all the way from power-on to successfully running and debugging code.

3.2 Sequence of Events

In general, the procedure of bringing up a board uses the following outline:

- Attempt a “smoke test”- that is, see if you can apply power to the board without damaging it.
- Perform a “lamp check” - turn the LEDs on and off
- Establish a JTAG or BDM connection to the emulator.

- Configure the emulator-target interface: set voltage, clock rate, signal logic.
- Enter background mode.
- Read and write core registers.
- Configure the target workspace.
- Run simple RAM tests.
- Run bus tests on the address and data buses.
- Test low-level stepping and breakpoints.
- Execute low-level code.
- Test source-level stepping and breakpoints.
- Execute application.
- Debug application code in RAM.
- Test the target's ability to erase and program flash memory.
- Debug application code in ROM.

4

OCD Connections

[4.1 Debug Connections](#) 9

[4.2 Creating a Target Connection](#) 9

4.1 Debug Connections

To create a target connection, create projects, and download code, you need a Wind River Probe or a Wind River ICE SX. (The Wind River Instruction Set Simulator (ISS) is not supported for ARM and XScale targets.)

The instructions in this document use a Wind River Probe connecting to an ARM MC9328MX1 target. The process for connecting with the ICE is similar; for instructions on connecting with a Wind River ICE SX, see the *Wind River ICE SX for Wind River Workbench Hardware Reference*.

4.2 Creating a Target Connection

To establish a connection with the Wind River Probe, use the following steps.

1. Launch Wind River Workbench according to the method for your host.

Linux/Solaris Hosts

From your installation directory, issue the following command:

```
$ ./startWorkbench.sh
```

Windows Hosts

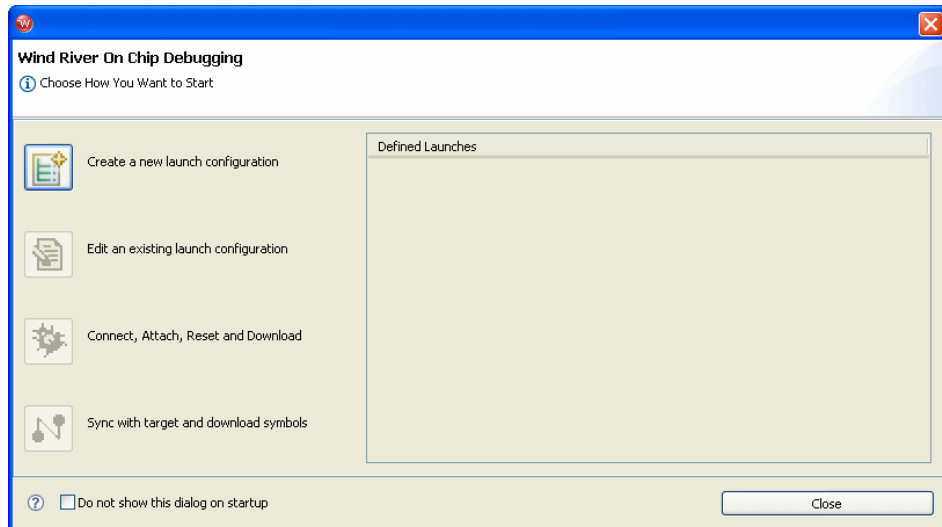
Select **Start > Programs > Wind River > Wind River Workbench 2.6.1 > Wind River Workbench 2.6.**

Wind River Workbench launches.

2. Specify a workspace.

For Windows hosts, Workbench displays a dialog where you can specify a location for your workspace. For Linux hosts, the workspace defaults to *installDir/workspace*.

After you specify your workspace, Workbench opens and the **Quick Target Launch** dialog appears.



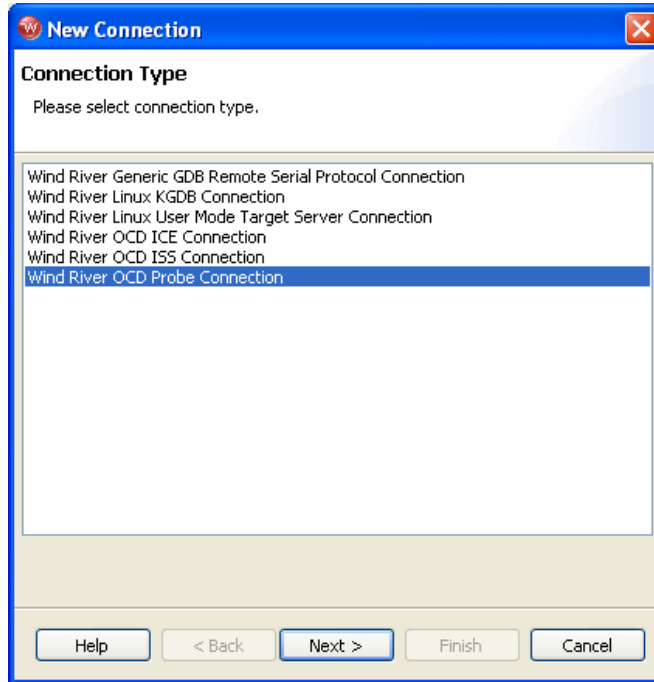
The **Quick Target Launch** dialog allows you to create a launch configuration to initialize your target and download symbols and code. Once created, the launch configuration is persistent, so you can return to it at any time.

If this is the first time you have launched Workbench, the only available option is **Create a new launch configuration**. The next time you open Workbench, the

Quick Target Launch dialog will give you the option of re-launching the same launch configuration or creating a new one.

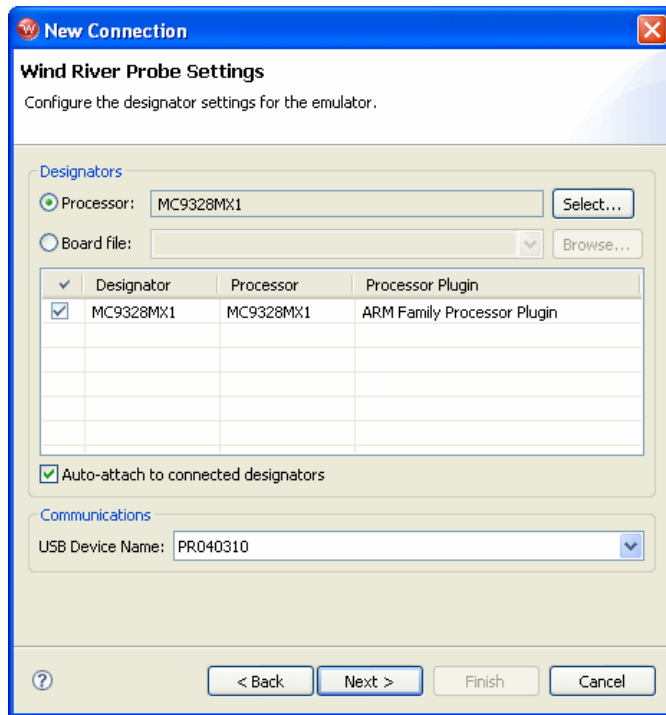
3. Select **Create a new launch configuration**.

The **New Connection Wizard** appears.



4. Select **Wind River OCD Probe Connection** and click **Next**.

The **Processor Selection** dialog appears.

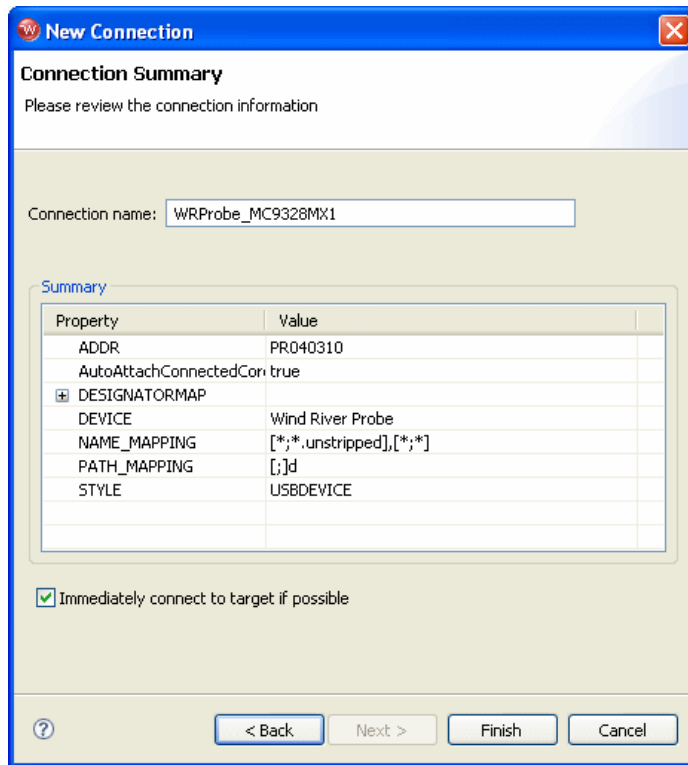


5. Click **Select**. From the list that appears, expand **ARM** and select **MC9328MX1**.
6. Click **OK**.

You are returned to the **Processor Selection** dialog.

7. Click **Next**.

The connection wizard passes through several additional screens. For the purposes of this chapter you do not need to use these screens. Click **Next** until you come to the **Connection Summary**.



8. Make sure that **Immediately connect to target if possible** is selected and click **Finish**.

Workbench creates a target connection called **WRProbe_ARM920T** in the **Target Manager** view.

9. In the **Target Manager** view, click on the “+” sign next to the **WRProbe_ARM920T** target connection name to expand it.

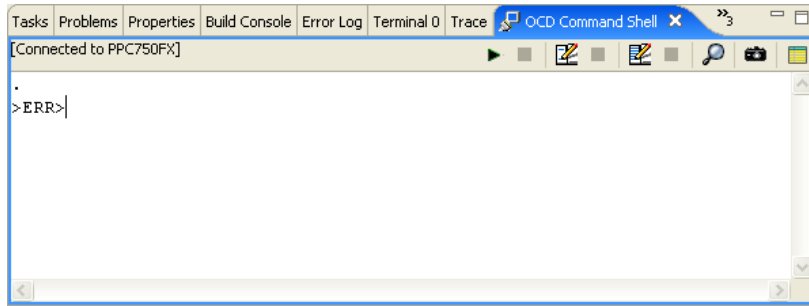
Before Workbench can actually talk to the processor on the target system, Workbench must attach to the core.

10. Right-click on **ARM920T [connected - stopped]** and select **Attach to Core**.

Workbench is now attached to the core, and able to talk to the processor. Workbench switches to displaying the **Device Debug** perspective.

In the Workbench toolbar, select **Window > Show View > OCD Command Shell**.

The **OCD Command Shell** opens.



The prompt in the **OCD Command Shell** will read either **>BKM>** (background mode) or **>ERR>** (error.)

There are several reasons an **>ERR>** prompt might appear; these will be addressed further on.

The next step is to configure the emulator by setting certain configuration options, as described in [5. Tool Configuration](#).

5

Tool Configuration

5.1 Introduction 15

5.2 Tool Configuration 16

5.1 Introduction

Wind River emulators can be configured in several different ways to specify various settings such as electrical properties, connection logic, and clock rate. To configure these settings Workbench uses *configuration options*, or CF options, which you can set in the **OCD Command Shell**.

This document only describes the most important CF options, ones that are common to all Wind River-supported processor families. For a full description of all Wind River CF options sorted by processor family, see the *Wind River Workbench for On-Chip Debugging Configuration Options Reference*.

5.2 Tool Configuration

At the prompt in the **OCD Command Shell** (either **>BKM>** or **>ERR>**) enter the command **CF** with no arguments.

This displays a list of all CF options available for your target processor, along with their current settings.

```
>ERR>cf

Target CPU          TAR[ARM9TDMI, ARM920T, ARM922T, ARM926E, ARM940T, ARM946E]
                    [EPXA, MC9328MX1, MC9328MX21, AT91RM9200, MV88F5181, MV88F5281, KS8695PX] =
MC9328MX1
Vector Table Location      VECTOR[IGNORE,HIGH,LOW] = IGNORE
Set BreakPoint             SB[SB,IHBC] = SB
Monitor Target reset       RST[NO,YES,HALT,RUN] = YES
Drive TRESET line          TRESET[OPENC,ACTIVE,SOFT] = ACTIVE
Emulator HRESET Control    HRESET[ENABLE,DISABLE] = ENABLE
mulator HRESET Command Control  CMDRST[IN,RST,BOTH] = BOTH
HRESET and TRESET Tied Together  TIED[YES,NO] = NO
HRESET Pulse Length N*1ms      RPL[1..600] = 10
Wait to enter Background mode N*100ms  FRZ[1..30] = 1
Issue an IN on coldstart      INCOLD[YES,NO] = NO
Invalidate Instruction Cache on GO      INVC I[YES,NO] = YES
Real time Preservation       RTP[YES,NO] = NO
Trap Exceptions              TRPEXP[YES,NO,value] = NO
Little Endian Mode           LENDIAN[YES,NO] = YES
JTAG clock rate (MHz)        CLK[0.025...100] = 3
Target Console Redirection    TGTCONS[BDM,COM1,COM2] = BDM
Set WRS Work Space           WSPACE[BASE and SIZE] = 00000000 800
Load Boot Table On IN        BL[ENABLE,DISABLE] = DISABLE
Memory Management Unit Mode   MMU[ENABLE,DISABLE] = DISABLE
Cache Size Detection         CSD[ENABLE,DISABLE] = DISABLE
>ERR>
```

5.2.1 Clock Rate

The **CLK** option controls the rate at which the JTAG clock (or BDM clock) clocks debug commands to the target.

Available clock rates, and default settings, vary between processor families. Enter **CF** at the prompt and look for **CLK** in the list of CF options to see the available clock rates for your target.

For an ARM MC9328MX1 target, the available rates (shown above) range from 0.025 to 100. The default is 3. To change the clock rate, say from 3 to 20, use the following command:

```
>ERR>cf clk 20
```


5.2.2 Drive TRESET Line

The TRESET option controls the logic applied to the target reset (TRESET) signal on the target.

The option can be set to **OPENC** or **ACTIVE**. It is set to **ACTIVE** by default.

When set to **ACTIVE**, the emulator uses transistor-transistor logic (TTL.) The emulator drives the TRESET signal to both active and inactive states. On some targets, the conditioning resistors cause excessive rise or fall time on the signal when returning to an inactive state. This excessive time can cause the processor to come out of reset in an incorrect state.

When set to **OPENC**, the emulator uses open-collector logic. The active driver is released by tri-stating the line and allowing conditioning resistors on the target to return the signal to the non-active state.

If you are driving the TRESET signal with an external line, you should set the emulator to use open-collector logic. Otherwise you could have an external line driving the TRESET signal LOW while the emulator is driving it HIGH, thus causing bus contention and possible damage to the target or the emulator.

To set the TRESET option to **OPENC**, use the following command:

```
>ERR>cf treset openc
```

To change it back, use the following command:

```
>ERR>cf treset active
```

5.2.3 Monitor Target Reset

The emulator continuously monitors the TRESET signal. If a target reset occurs, the emulator takes one of the following actions:

- **YES** - If a target reset occurs it is reported to the user, and the target is forced out of background mode.
- **NO** - If a target reset occurs it is ignored. This is normally used if the code contains a reset instruction, which causes a reset to the external hardware, but does not reset the core.
- **HALT** - If a reset occurs, the target is trapped at the restart vector.
- **RUN** - If a reset occurs, the target is restarted and remains in background mode.

By default, this option is set to **YES**. When set to **YES**, the target will start running code after each reset. If you are doing low-level work -- for example, if you are

examining register settings -- you may want the target to halt after a reset so you can get a target snapshot. To set this option to halt the target on a reset, use the following command:

```
>ERR>cf rst halt
```

To change it back, use the following command:

```
>ERR>cf rst yes
```

5.2.4 Emulator HRESET Control

By default, the emulator asserts the hardware reset (HRESET) signal when initializing the hardware.

To configure the emulator not to assert the HRESET signal when it initializes the board, use the following command:

```
>ERR>cf hreset disable
```

To change it back, use the following command:

```
>ERR>cf hreset enable
```

5.2.5 CPU Reset Type

As stated above, the emulator asserts the hardware reset (HRESET) signal when initializing the hardware. You can configure the emulator to assert the software reset (SRESET) signal on an initialization instead.

To configure the emulator to assert the SRESET signal instead of the HRESET signal when it initializes the board, use the following command:

```
>ERR>cf reset sreset
```

To change it back, use the following command:

```
>ERR>cf reset hreset
```

You can also set this option to `HRESET_UNFILTER` or `SRESET_UNFILTER`. With the `_UNFILTER` argument added, The emulator will not sample the reset signal when it initializes the board.

5.2.6 Saving Changes

Most changes to configuration options do not take effect until you initialize the board, as described in [6. Board Initialization](#).

6

Board Initialization

- 6.1 Introduction 21
- 6.2 Background Mode 22
- 6.3 The INN Command 25
- 6.4 Registers 25

6.1 Introduction

In order to establish communications with your target, you must first initialize it. Also, if the code you are running on your target causes the connection to be lost, you must initialize the target to restore that connection. Initialization is also required if you change the register settings in the emulator and want them to be reflected in the target.

The target is initialized whenever you first establish a connection using your emulator. If you need to initialize the target when you are debugging, you can do it using the IN or INN initialization commands, as described in this chapter.

6.2 Background Mode

In order for the emulator to work with the target, it must stop the target CPU and put the target in background mode. When the target is in background mode, a **>BKM>** prompt appears in the **OCD Command Shell**.

If an **>ERR>** prompt appears in the **OCD Command Shell**, the target is not in background mode.

6.2.1 The IN Command

The **IN** command does two different things. First, it places the target board into background mode. Second, it copies all of the register information that is stored in the emulator's NVRAM down to the target.

To initialize the board and enter background mode, enter the following command:

```
>ERR> in
```

The **IN** command may fail for several reasons. For example, if you have not connected power to the target board, the output will resemble the following:

```
>ERR>in
*****
Wind River Probe Initialization Sequence.
Copyright (c) Wind River Systems, Inc. 1999-2005. All rights reserved.
*****
    Support Expires..... FlexLM key in use.
    Target Processor..... MC9328MX1:U1
Wind River Probe      Group ID#= 0
Wind River Probe      Serial#= U1234567      Firmware= pr2.4a_qa7
Type CF For a Menu of Configuration Options
Initializing Background Debug Mode.....Failed
Testing Communications to Hardware Interface...Passed
Driving HRESET to be High.....Passed
Driving HRESET to be Low.....Passed
Waiting HRESET Low Acknowledge.....Failed
>ERR>
```

For a list of the tests the emulator runs during an **IN** sequence, see [6.2.2 Set Verbose On](#), p.22.

6.2.2 Set Verbose On

To see the tests the emulator is running while attempting to enter background mode, put the emulator in verbose mode using the following command:

```
>ERR>set verbose on
```

Then enter the IN command again.

Here is a brief description of the tests with some possible reasons why each test might fail.

Testing Communications to Hardware Interface

This tests the hardware connectivity, and examines the communications path between the host and the emulator. If the test fails, ensure that you have the power properly connected and turned on, that the emulator is correctly connected to the host computer, and that your emulator hardware is properly connected to the target.

Driving HRESET to be High

This function tests the RESET signal to verify that it is HIGH. The emulator is not driving the RESET signal during this test, so the target must drive the RESET signal via a pull-up resistor. If this test fails, check to see if the target board has a pull-up resistor on the RESET signal to the HRESET pin of the connector. Also, check the target board reset logic and verify that it is not continually driving RESET LOW.

Driving HRESET to be Low

The RESET signal is a bi-directional signal for your unit. The emulator drives the RESET signal LOW and clocks it back in to verify that it is LOW. If this test fails, you may have contention on your RESET signal. Check to see if a device on your target board is continually driving RESET HIGH. Verify that the device on your target board that is driving the RESET signal is an open-collector device with a pull-up resistor.

Attempting JTAG Communication

During this test, the emulator stops the processor and attempts to establish JTAG communications. If this fails, check to see that your hardware is connected properly, and that the tests preceding this one passed. It is also possible that there is contention on your board.

Waiting for HRESET to be Released

The emulator only drives RESET low for a specified period of time. After RESET is driven LOW for the allotted time, it tri-states the RESET driver and clocks the RESET signal back in to see if the RESET signal went high. It continues to check for RESET to go high until it sees it go high or until you type **Ctrl+X**. If this test fails,

check to see if your target board reset logic is still driving the RESET signal LOW. Also check that your target board has a pull-up resistor to drive RESET HIGH.

Testing for Target STOP State

This test verifies that the processor stopped during the preceding JTAG Communications test by polling the processor status. If the target is still running, this test fails.

Comparing Target CPU With CF Setting

This test verifies that the emulator is properly configured for the appropriate target processor by comparing the processor type on your target with the processor type specified in your board file. If the test fails, use the `CF TAR` command to properly configure your target. For example, if you are using an ARM MC9238MX1 target, and this test fails, enter the following commands:

```
>ERR>cf tar MC9238MX1  
>ERR>in
```

Loading Internal Registers

Once background communications are established, the emulator downloads register values from the debugger NV-RAM to the target. It will only download register values for those register groups that are enabled. If this test fails, see the information in [6.3 The INN Command](#), p.25.

Testing JTAG Communication

This test examines the JTAG communication between the emulator and the target using the internal clock rate for which the emulator is configured. If this test fails, set the internal clock to a lower rate using the following command:

```
>ERR>cf clk value
```

Attempting to restore CPU context

This test restores the processor scan chains.

6.3 The INN Command

In order to get a processor into background mode, the emulator asserts the RESET line of the processor and then releases it. The processor and its peripherals on the target board are forced into their reset state, and all of the internal registers are forced to their manufacturer's reset value.

The INN command places the target in background mode without overwriting the target's registers, leaving them in their default reset state for the processor.

If the IN command fails to put the target in background mode, enter the following command:

```
>ERR>inn
*****
Wind River Probe Initialization Sequence.
Copyright (c) Wind River Systems, Inc. 1999-2005. All rights reserved.
*****
      Support Expires..... FlexLM key in use.
      Target Processor..... MC9328MX1:U1
Wind River Probe      Group ID#= 0
Wind River Probe      Serial#= U1234567      Firmware= pr2.4a_qa7
Type CF For a Menu of Configuration Options
Initializing Background Debug Mode.....Successful
>BKM>
```

Generally, if an IN command fails but an INN succeeds, it is usually caused by incorrect register values in the emulator's NVRAM.

To configure register values, see [6.4 Registers](#), p.25.

6.4 Registers

Your emulator includes an area of non-volatile memory (NVRAM) where you may store register settings for a target.

Once the register values are present in NVRAM, they are automatically loaded to the target after each cold start, warm start, or IN initialization command. You can select which register values are written to the target by enabling and disabling the appropriate register groups.

Wind River uses low-level SCGA commands to configure registers. Since configuring registers manually would require entering a large number of SCGA

commands, Wind River provides *register files* for many targets. A register file is a Workbench-specific script that you can execute in the **OCD Command Shell**.

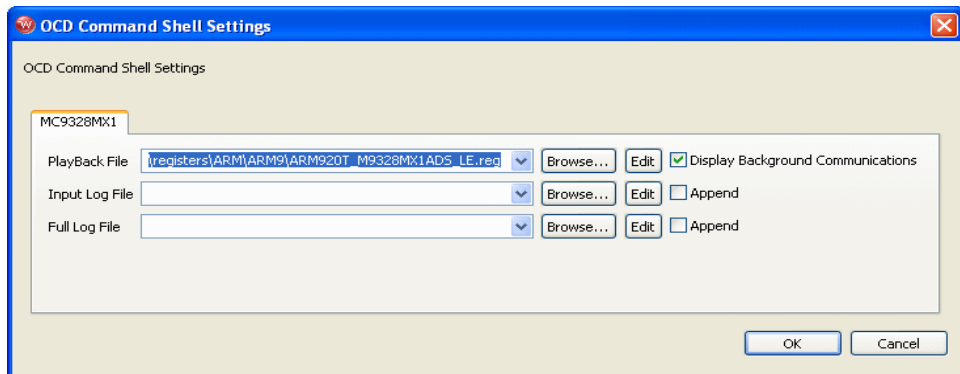
Register files are ASCII files using the extension **.reg**. For example, a register file for the ARM MC9238MX1 920T target is **ARM920T_M9328MX1ADS_LE.reg**, located in *installDir/workbench-2.x/dfw/build/host/registers* in the directory **ARM/ARM9**.

6.4.1 Downloading a Register File

To download a register file to the emulator, use the following steps:

1. In the **OCD Command Shell**, click the **Settings** button.

The **OCD Command Shell Settings** dialog appears.



2. Next to the **PlayBack File** field, click **Browse**.
3. Navigate to the register file you wish to use and click **Open**.
4. Click **OK**.
5. In the **OCD Command Shell**, click the **Playback File** button.

The register file you selected is downloaded to your target. The commands from the file appear in the **OCD Command Shell**.

This procedure only sets the register values in the emulator's NVRAM, not on the target. To copy the register values from the emulator to the target, you must initialize the target with the **IN** command:

```
>BKM>in
```

Only enabled register groups are copied to the target.

6.4.2 Enabling and Disabling Register Groups

If you look at the `ARM920T_M9328MX1ADS_LE.reg` register file, you will see that it ends with several lines that begin `CF GRP`. Registers are stored in logical register groups. When you issue an `IN` command, the emulator only copies down register settings for register groups that are enabled. Register groups that are disabled on your target do not have register data transferred.

Disabling a register group enables you to view the target register value, but prevents it from being overwritten during target initialization.



NOTE: If you change a register value directly on the target of a register group that is disabled, that register does not get overwritten by the emulator during an initialization. Note, however, that the processor may still reset that register value to the processor default during a target initialization.

To enable or disable a register group on your target, use the following steps:

1. At the `>BKM>` prompt, type the command `CF GRP`.

The first register group appears, as shown below:

```
>BKM>cf grp
Group          (CF GRP (M/S)   Name = ENABLED/DISABLED
CUSTOM                (0=Disable 1=Enable)   Enabled >
```

The name of the register group is displayed, along with its current status (either **ENABLED** or **DISABLED**).

2. Type `1` to enable the group or `0` to disable it.
3. To leave the setting as it is and advance to the next register group, press the **ENTER** key without typing `0` or `1`.
4. Continue through the list of register groups enabling and disabling them as required.
5. When the register groups are enabled or disabled, type `CF UPLOAD GROUP` at the `>BKM>` prompt.

This displays a list of all of the register groups on your target with their current settings as shown below:

```
>BKM>cf upload group
CF GRP          GT64260_CPU ENABLED      ; GROUP
CF GRP          GT64260_SDRAM ENABLED    ; GROUP
```

```
CF GRP                GT64260_DEVICE ENABLED    ; GROUP
CF GRP                GT64260_GPP ENABLED      ; GROUP
CF GRP                GT64260_MPP ENABLED     ; GROUP

>BKM>
```

6.4.3 Modifying Registers Manually

Wind River supplies register files for Wind River evaluation boards, as well as for many third-party target boards.

If you are using a target for which Wind River does not supply a register file, you may have to create one. For instructions on creating register files, see the *Wind River Workbench for On-Chip Debugging User Tutorials: Configuring Target Registers*.

Remember that the register file sets the register values in the emulator NVRAM, not on the target. The emulator copies the values you set in its NVRAM down to the target when you initialize the target with an IN command. Without a register file, the NVRAM contains default register values, typically made for a Wind River evaluation board, which most likely are not suitable for your target. So the IN command will not set the target registers properly.

Some target processors come with default register settings. If your target has default register settings, you can modify the registers directly on your target manually, at least to the point where you can download your boot ROM application code.

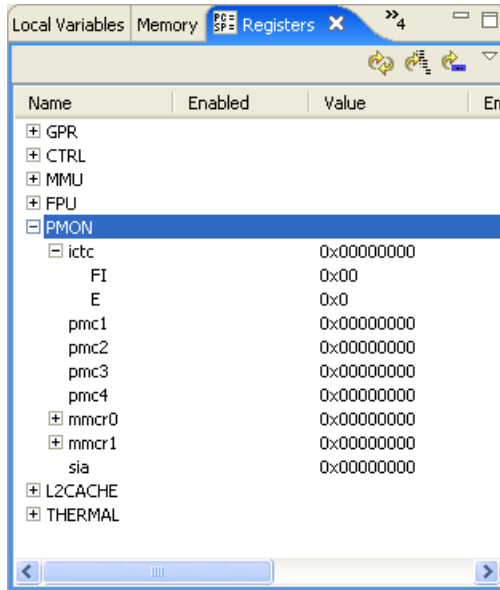
Remember that if you modify your registers manually, any initialization command or target reset will overwrite your changes.

To modify registers manually, use the **Registers** view in Workbench. The **Registers** view lets you examine the bit-level detail for each register. The following sections describe the **Registers** view and the bit-level detail provided.

The Registers View

When the **Registers** view is open in Workbench, all of the register groups for your target are displayed with + signs beside them. Clicking on a + sign expands the register group, showing all of the registers that are included in that register group along with the value that they are currently set to. An example of an expanded register group is shown in [Figure 6-1](#).

Figure 6-1 Expanded Register Group



➔ **NOTE:** Figure 6-1 is only an example of an expanded register group. The groups and the register values vary widely depending on your target architecture.

Bit-Level Detail

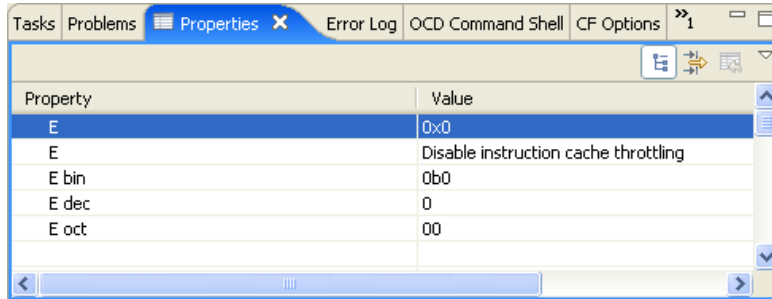
You can view the bit-level detail for any register by clicking on the + sign beside the register in the register group.

➔ **NOTE:** Before you can make any changes to your register settings, you need to enable the register group that contains the register you want to modify, so that the values download to the target when you initialize your system. If you do not enable the register group, you can still modify the settings in the emulator but not on the target. For more information, see [6.4.2 Enabling and Disabling Register Groups](#), p.27.

You can make changes to any of the register settings by modifying each of the bit-level settings for any register.

To modify bit-level values for your target, complete the following steps:

1. In the **Registers** view, double-click on the name of the register you wish to edit.
This opens the **Properties** view, which shows the name of the register you have selected under the **Property** heading and its current setting under the **Value** heading.



2. Select the value under the **Value** heading and edit it as necessary.
3. In the **Registers** view, click **Refresh Values**. The register information reappears with your changes.



NOTE: Some registers are write-protected and cannot be edited.

For more information on registers, including creating custom registers and register groups, see the *Wind River Workbench for On-Chip Debugging User Tutorials: Configuring Target Registers*.

When you have initialized your target and entered background mode, with a **>BKM>** prompt showing in the OCD Command Shell, you can proceed to test your hardware, as described in [7. Verifying Hardware](#).

7

Verifying Hardware

- 7.1 Introduction 31
- 7.2 Setting a Workspace 31
- 7.3 Diagnostic Functions 32

7.1 Introduction

This chapter describes several tests and diagnostics you can use to verify that your hardware is working correctly.

7.2 Setting a Workspace



NOTE: The RAM workspace has no relation to the workspace that Workbench uses to store project information.

The workspace is an area of RAM on the target that the emulator uses to download the hardware diagnostic routines and flash programming algorithms.

You must tell your emulator where writable RAM is located on your target for this purpose.

Depending on the device family and type, this space is limited to under 2 KB. Note that more memory improves the speed of programming.

To configure the workspace, enter the parameters using the syntax

CF WSPACE *base size*

where *base* is the start address, and *size* is the minimum number of bytes of target RAM required.

To find the *base* and *size* values for a Wind River-supported target, consult your target's **target.ref** file, located in *installDir/vxworks-6.x/target/config/yourTarget*. Alternatively, consult your processor documentation.

For example, say the base of the workspace is 00300000 and the size is fde0. To set the workspace, enter the command

```
>BKM>cf wspace 0 fde0
```

This sets the workspace at address 0x00300000 with a size of 0xfde0 bytes.

7.3 Diagnostic Functions

Wind River Workbench provides a set of RAM and bus diagnostics and utilities that can be controlled by the emulator or run on the target.

Some of the following tests can run code directly on the target instead of through the emulator by selecting the **Run on Target** checkbox. This allows the test to run at the execution speed of the target processor.

7.3.1 Simple RAM Test

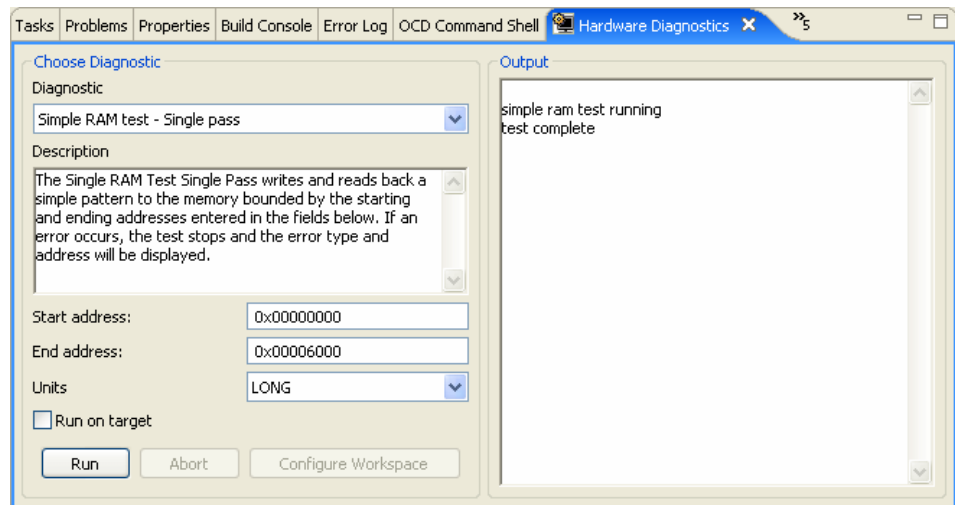
This test writes and reads back a simple pattern to the memory bounded by the starting and ending addresses entered in the **Start Address** and **End Address** fields. If an error occurs, the test stops and the error type and address are displayed in the **Output** field.

The first diagnostic to be run is a Simple Ram Test on the area of memory used by the workspace.

1. In the Workbench toolbar, select **Window > Show View > Hardware Diagnostics**.
2. In the **Diagnostic** field, select **Simple RAM Test – Single Pass**.
3. The workspace cannot be used to test itself, so make sure the **Run on target** checkbox is unchecked.
4. In the **Start Address** field, enter **0**.
5. In the **End Address** field, enter **6000**.
6. In the **Units** field, select **LONG**.
7. Click **Run**.

Workbench displays the test result in the **Output** field. The output of a successful test will resemble that in [Figure 7-1](#).

Figure 7-1 **Successful Simple RAM Test**



If the test fails, the **Address Bus Test** diagnostic and the **Data Bus Test** diagnostic may determine the cause of the failure; see [7.3.5 Bus Tests](#), p.36.

If the RAM test of the memory used by the workspace passed, the rest of the memory in the target system can now be tested at full bus speed.

1. In the **Diagnostic** field, select **Simple RAM Test – Single Pass**.
2. Select the Run on Target checkbox.
3. In the **Start Address** field, enter **14000**.
4. In the **End Address** field, enter **20000000**.
5. In the **Units** field, select **LONG**.
6. Click **Run**.

Workbench displays the test result in the **Output** field.

If the message **Test Complete** appears, then the diagnostic passed.

If the test fails, try re-seating the SDRAM module and repeat the test. If the test still fails, then run the **Address Bus Test** diagnostic and the **Data Bus Test** diagnostic to determine the cause of the failure. See [7.3.5 Bus Tests](#), p.36.

7.3.2 Full RAM Tests

A Full RAM test writes a “walking” **1** on each bit of RAM and reads it back. This is a very lengthy test and can detect bus configuration errors, typically on a new printed circuit board.

This test sets and then clears each bit to try to locate memory defects bounded by the starting and ending addresses entered in the **Start Address** and **End Address** fields. If an error occurs, the test stops and the error type and address will be displayed in the **Output** field.



NOTE: A complete Full RAM test would take several years to finish, so make sure you specify a very small region of memory to be tested.

Full RAM tests are designed to check for cell disturbance and addressing problems. These tests perform the following actions:

A **Single Pass** test will run the test only once. A **Continuous** test will repeat the test over the same address until you click **Stop**.

1. In the **Diagnostic** field, select **Simple RAM Test – Single Pass**.
2. Select the **Run on Target** checkbox.
3. In the **Start Address** field, enter **14000**.
4. In the **End Address** field, enter **14100**.

5. In the **Units** field, select **LONG**.
6. Click **Run**.

Workbench displays the test result in the **Output** field.

If the message **Test Complete** appears, then the diagnostics passed.

If the test fails, try re-seating the SDRAM module and repeat the test. If the test still fails, then run the **Address Bus Test** diagnostic and the **Data Bus Test** diagnostic to determine the cause of the failure. See [7.3.5 Bus Tests](#), p.36.

7.3.3 CRC Calculation

Workbench and the emulator support the calculation of a Cyclic Redundancy Check (CRC) on all addresses in the range specified. The CRC test will checksum a block of data on the target for the address range you specify in the **CRC Calculation** dialog. The CRC algorithm is based on the following polynomial:

$$x^{16} + x^{15} + x^2 + 1$$

Workbench uses this polynomial as follows:

Workbench reads a location and uses the value read, x , to calculate the CRC. Then Workbench adds the result to the value calculated for the previous address. This process continues until Workbench has checked the entire specified memory range.

The CRC sum will be returned if the communications with the emulator and target are working. To interrupt the test, click **Stop**.

7.3.4 Scope Tests

Read From Location

The Read From Location Scope Test performs a memory read of designated length from the address entered in the **From Address** field.

Write To Location

The Write To Location Scope Test performs a memory write of designated length of the value entered in the **Data Value** field to the address in the **To Address** field.

Write and Complement

The Write and Complement Scope Test performs a memory write of designated length of the value entered in the **Data Value** field to the address in the **To Address** field; the value is then complemented.

Write Rotating Value

The Write Rotating Value Scope Test performs a memory write of the value entered in the **Data Value** field to the address in the **To Address** field. The value is then rotated through all of the bit positions with respect to the designated length of the memory address.

Write Then Read

The Write and Read Scope Test performs a memory write of designated length of the value entered in the **Data Value** field to the address in the **To Address** field; the value is then read back.

7.3.5 Bus Tests

Address Bus Test

This test detects faults in the address bus over the range bounded by the starting and ending addresses entered in the **Start Address** and **End Address** fields. This test can be interrupted by clicking the **Stop** button.

Data Bus Test

This test detects faults in the data bus over the range bounded by the starting and ending addresses entered in the **Start Address** and **End Address** fields. This test can be interrupted by clicking the **Stop** button.

When you have tested your hardware successfully, you must test your ability to read and write memory, as described in [8. Testing Memory](#).

8

Testing Memory

[8.1 Introduction 37](#)

[8.2 Testing Memory 37](#)

8.1 Introduction

Before handling more complex application code, the target system must be able to handle low-level assembly instructions.

Wind River Workbench includes a simple diagnostic to test the target's ability to write to memory, set breakpoints, and run and step code. This diagnostic writes a loop of NOP instructions at a specified memory address.

8.2 Testing Memory

To run the memory diagnostic, use the following steps.

1. At the **>BKM>** prompt in the **OCD Command Shell**, enter **DF E 14000**.
This writes a NOP loop at address 0x14000.

2. Enter the command **DI 14000**.

This command disassembles the instructions at 0x14000.

3. Enter the command **SR PC 14000**.

This command sets the Program Counter to address 0x14000.

The output should resemble that shown below.

```
>BKM>df e 14000
>BKM>di 14000
$00014000 : 0xFFFFFFFF :arm swinv 0x00ffffff
$00014004 : 0xFFFFFFFF :arm swinv 0x00ffffff
$00014008 : 0xFFFFFFFF :arm swinv 0x00ffffff
$0001400C : 0xFFFFFFFF :arm swinv 0x00ffffff
$00014010 : 0xFFFFFFFF :arm swinv 0x00ffffff
$00014014 : 0xFFFFFFFF :arm swinv 0x00ffffff
$00014018 : 0xFFFFFFFF :arm swinv 0x00ffffff
$0001401C : 0xFFFFFFFF :arm swinv 0x00ffffff
$00014020 : 0xFFFFFFFF :arm swinv 0x00ffffff
$00014024 : 0xFFFFFFFF :arm swinv 0x00ffffff
$00014028 : 0xFFFFFFFF :arm swinv 0x00ffffff
$0001402C : 0xFFFFFFFF :arm swinv 0x00ffffff
$00014030 : 0xFFFFFFFF :arm swinv 0x00ffffff
$00014034 : 0xFFFFFFFF :arm swinv 0x00ffffff
$00014038 : 0xFFFFFFFF :arm swinv 0x00ffffff
$0001403C : 0xFFFFFFFF :arm swinv 0x00ffffff
$00014040 : 0xFFFFFFFF :arm swinv 0x00ffffff
$00014044 : 0xFFFFFFFF :arm swinv 0x00ffffff
$00014048 : 0xFFFFFFFF :arm swinv 0x00ffffff
$0001404C : 0xFFFFFFFF :arm swinv 0x00ffffff
>BKM>sr pc 14000
>BKM>
```

Now there is a simple program in the target's memory, and the Program Counter has been set to 0x14000.

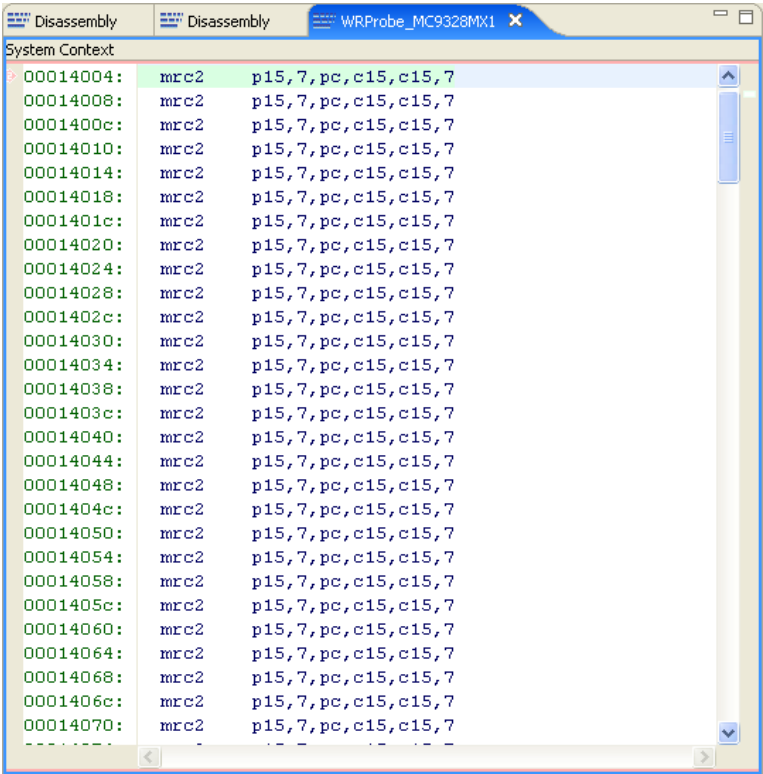
8.2.1 Stepping an Instruction

First, test to see if the system can handle the step instruction command.

In the **Debug** view, click the **Step Into** button.

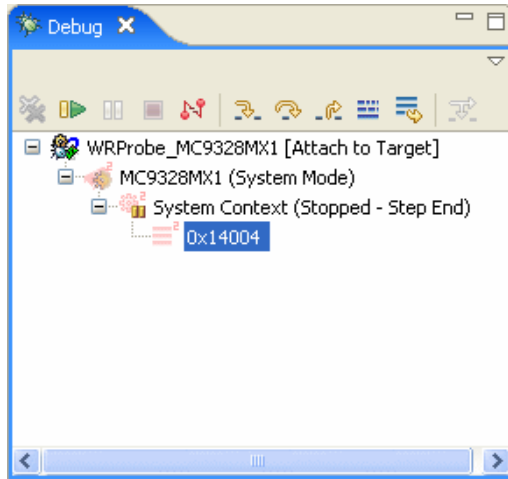
The **Disassembly** view opens, with the Program Counter now at 14004, as shown in [Figure 8-1](#).

Figure 8-1 Disassembly View



Also, the **System Context** in the **Debug** view now reads **0x14004**, as shown in [Figure 8-2](#).

Figure 8-2 **System Context**



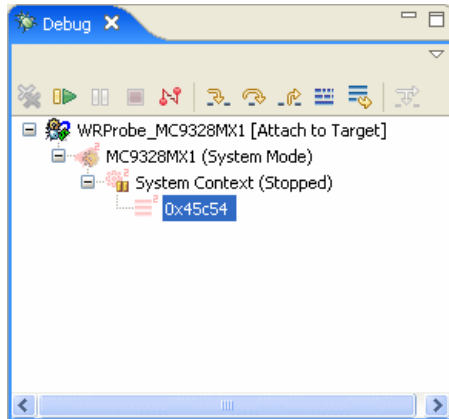
8.2.2 Running Code

Next, test to see if the processor can run the simple program at full bus speed.

In the **Debug** View, click the **Resume** button to start the target. In the **Debug** view, the **System Context** changes to **Running**, and a **>RUN>** prompt appears in the **OCD Command Shell**.

Wait a few seconds and then click the **Suspend** button to stop the target. In the **Debug** view, the **System Context** changes to **Stopped -- User Request**, as shown in [Figure 8-3](#).

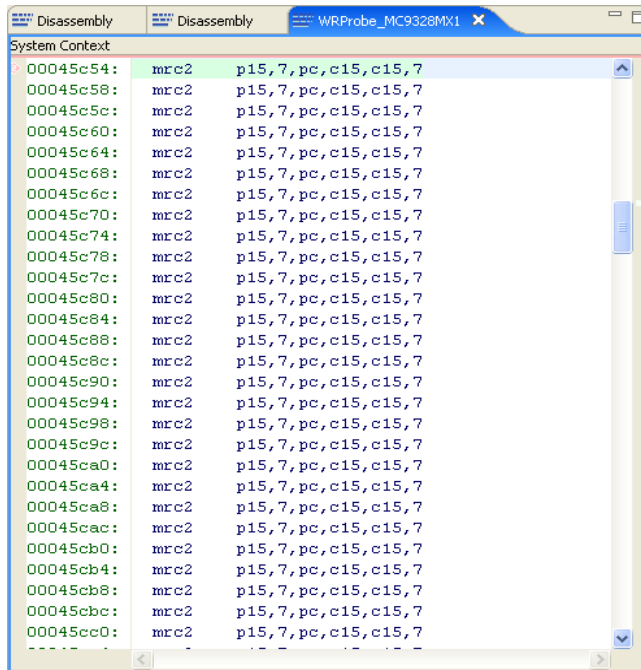
Figure 8-3 System Context



8

Also, the **Disassembly** view updates to show the new location of the Program Counter, as shown in [Figure 8-4](#).

Figure 8-4 Updated Program Counter

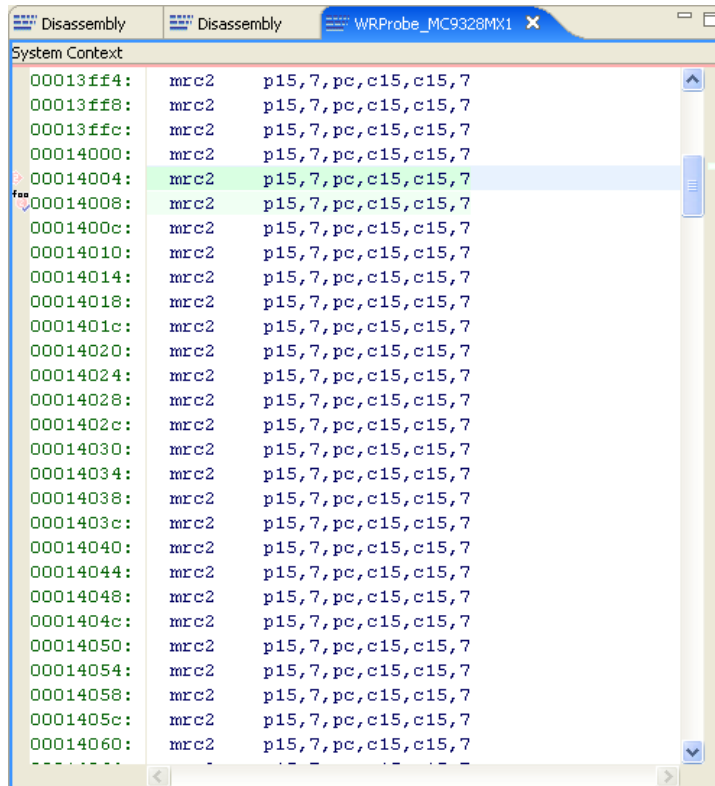


8.2.3 Setting Software Breakpoints

Next, test to see if the target can set a software breakpoint.

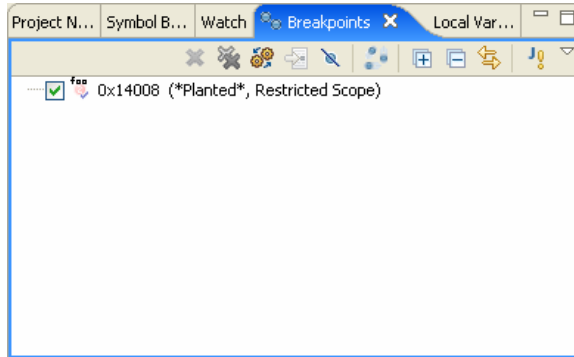
In the **Disassembly** view, double-click to the left of address 0x14008 (in the gutter.) Workbench places a software breakpoint at address 0x14008, as shown in [Figure 8-5](#).

Figure 8-5 **Planted Software Breakpoint**



The breakpoint at address 0x14008 appears in the **Breakpoints** view.

Figure 8-6 Breakpoints View



8

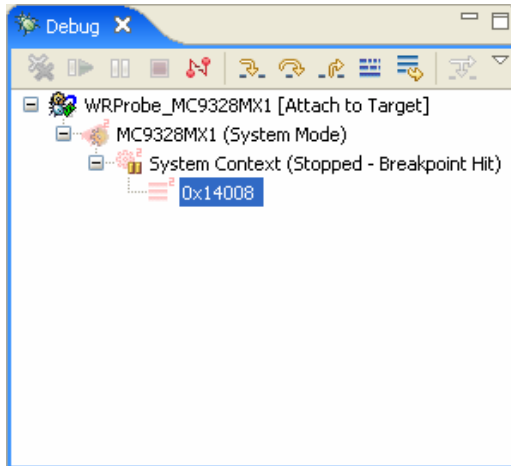
In the **Debug** view, click the **Resume** button to start the processor. The program will run until it hits the breakpoint. Output appears in the **OCD Command Shell**, showing that the system has stopped and showing the current location of the Program Counter, as shown below:

```
>RUN>  
  
!BREAK! - [msg12000] Software breakpoint; PC = 0x00014008 [EVENT Taken]  
>BKM>
```

This output shows that the software breakpoint at address 0x14008 has been hit.

In the **Debug** view, the **System Context** changes to **Stopped -- Breakpoint Hit**.

Figure 8-7 **System Context**



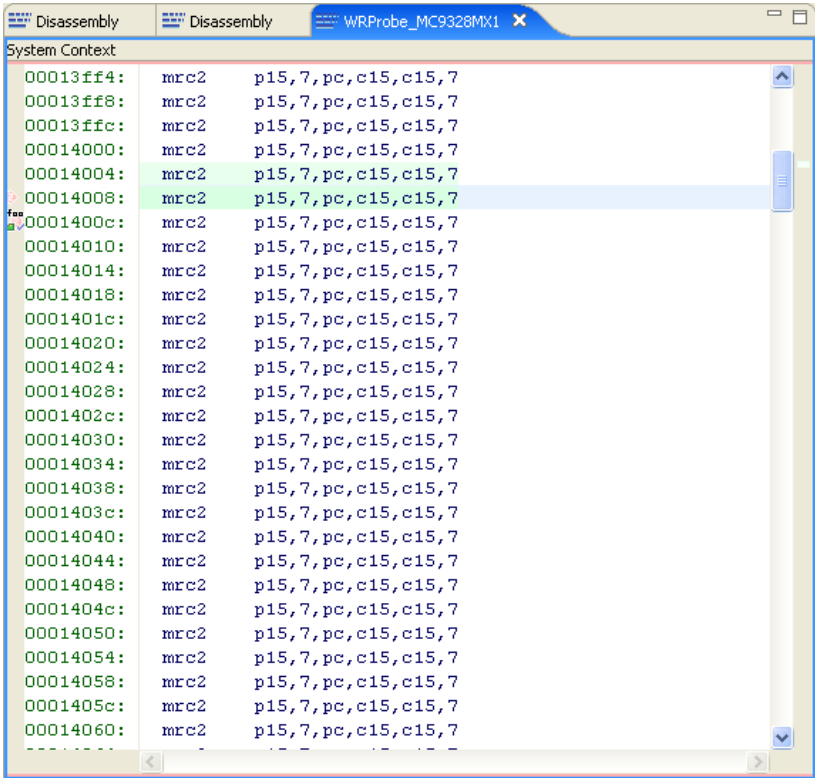
To remove the software breakpoint, double-click on the breakpoint icon to the left of address 0x14008 in the **Disassembly** view.

8.2.4 Setting Hardware Breakpoints

Next, test to see if the system can handle setting hardware breakpoints.

In the **Disassembly** view, right-click to the left of address 0x1400C (in the gutter) and select **Add Hardware Breakpoint**. Workbench places an internal hardware breakpoint at address 0x1400C, as shown in [Figure 8-8](#).

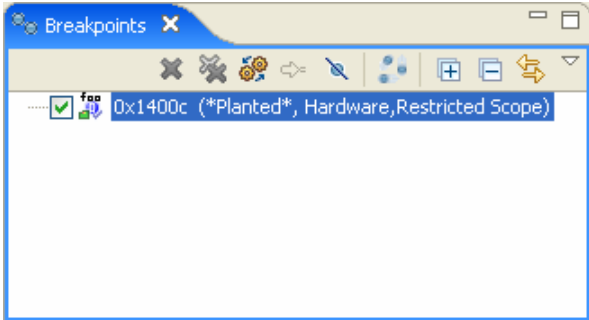
Figure 8-8 **Planted Hardware Breakpoint**



8

The hardware breakpoint at address 0x1400c appears in the **Breakpoints** view.

Figure 8-9 **Breakpoints View -- Hardware Breakpoint**



In the **Debug** view, click the **Resume** button to start the processor. The program will run until it hits the breakpoint. Output appears in the **OCD Command Shell**, showing that the system has stopped and showing the current location of the Program Counter, as shown below:

```
>RUN>  
  
!BREAK! - [msg11001] Internal hardware breakpoint; PC = 0x0001400c [EVENT  
Taken]  
>BKM>
```

This output shows that the hardware breakpoint at address 0x1400C has been hit.

In the **Debug** view, the **System Context** changes to **Stopped -- Breakpoint Hit**.

To remove the hardware breakpoint, double-click on the breakpoint icon to the left of address 0x1400C in the **Disassembly** view.

If all these steps perform successfully, the target can run and debug low-level assembly code. The next step is to run and debug application code, as described in [9. Debugging in RAM](#).

9

Debugging in RAM

- 9.1 Overview 47
- 9.2 Creating a Target Connection 47
- 9.3 Creating a Project 48
- 9.4 Downloading Code and Symbol Information 52
- 9.5 Debugging Code in RAM 55

9.1 Overview

This chapter describes the process of running and debugging application code in RAM using Wind River Workbench.

9.2 Creating a Target Connection

To download and run code and symbol information, you must have an active target connection.

To create a target connection, create projects, and download code, you can use a Wind River Probe or a Wind River ICE SX.

To create a target connection, use the procedure described in [4. OCD Connections](#).

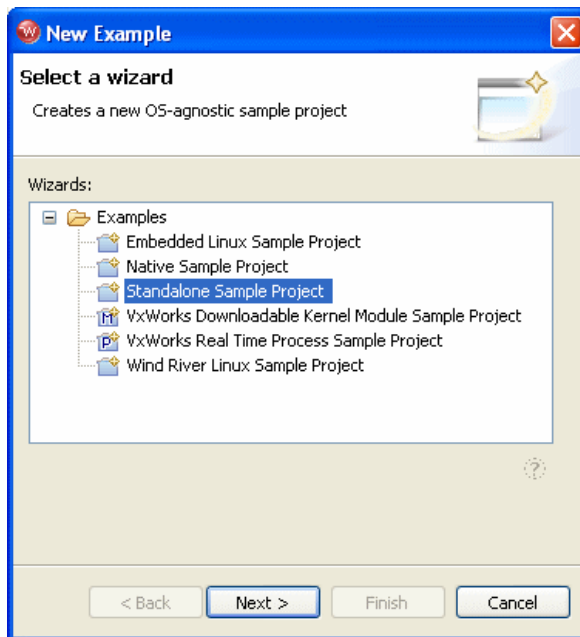
9.3 Creating a Project

In order to download and run code and symbol information in RAM, you must have an active project open.

Several example projects are included in Wind River Workbench for demonstration purposes. To open a new demonstration project, use the following steps:

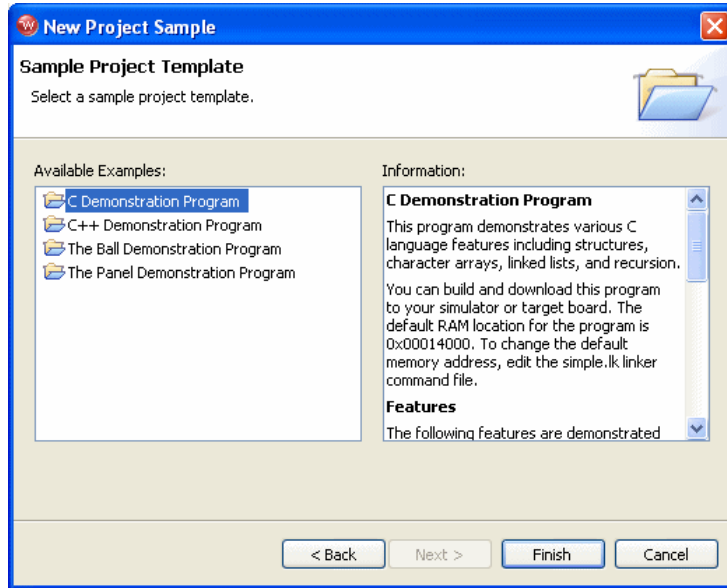
1. Select **File > New > Example**.

The **New Example** wizard appears.



2. Select **Standalone Sample Project** and click **Next**.

A sample project template appears.



3. Select **C Demonstration Program** and click **Finish**.

Workbench creates the sample project in the default **workspace** directory, and the project name **c_demo_sa** appears in the **Project Navigator** view.

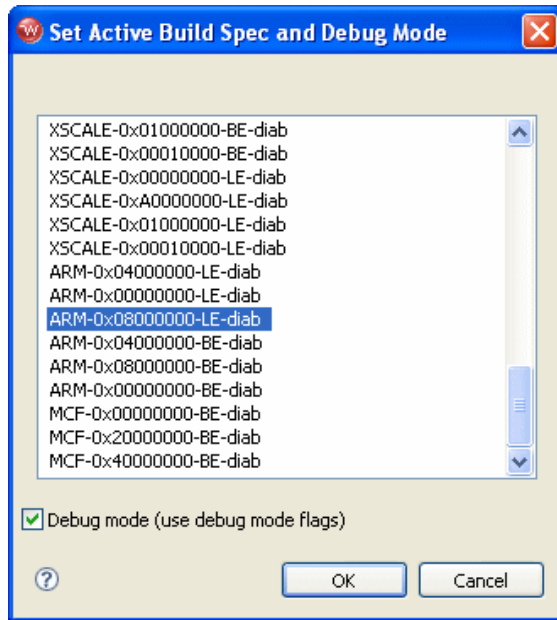
4. In the **Project Navigator** view, expand the **c_demo_sa** project.

A number of available build specs appear.



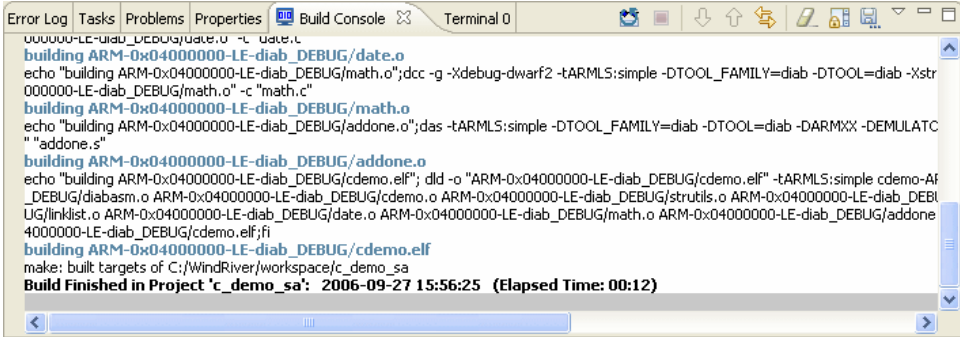
5. To build the sample project, right-click on the `c_demo_sa` top-level folder and select **Build Options > Set Active Build Spec**.

The **Set Active Build Spec and Debug Mode** dialog appears.



6. Select the build spec for your target family. This document uses the Wind River ARM MC9238MX1 for its examples, so this example shows an ARM build spec.
7. Select **Debug mode (use debug mode flags)** so Workbench will generate symbolic debug information.
8. Click **OK**.
9. Right-click on the `c_demo_sa` folder and select **Build Project**.

Workbench builds the sample project using the Wind River Compiler. The results of the project build appear in the **Build Console** view.



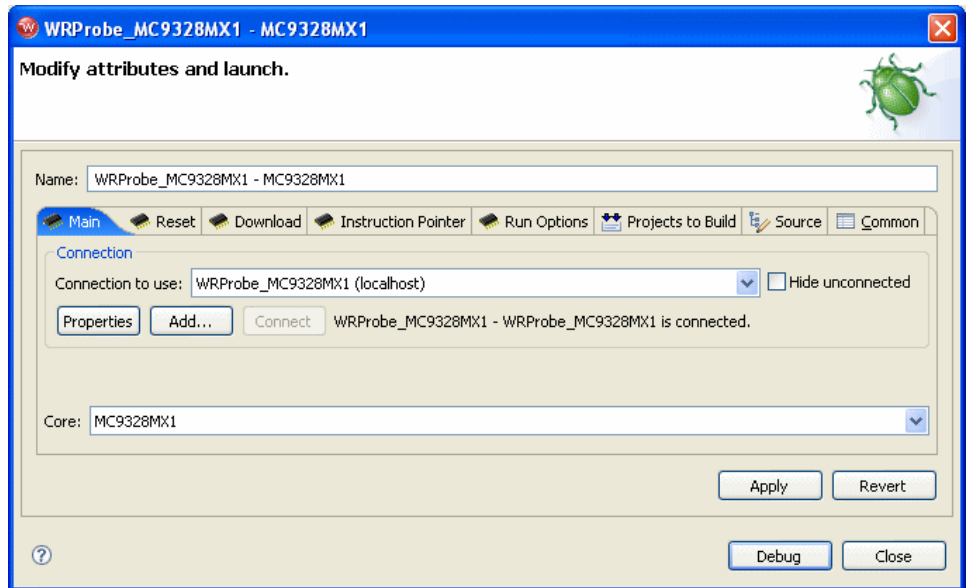
```
uuuuuu-LE-diab_DEBUG/date.o -c date.c
building ARM-0x04000000-LE-diab_DEBUG/date.o
echo "building ARM-0x04000000-LE-diab_DEBUG/math.o";dcc -g -Xdebug-dwarf2 -tARMLS:simple -DTOOL_FAMILY=diab -DTOOL=diab -Xstr
000000-LE-diab_DEBUG/math.o" -c "math.c"
building ARM-0x04000000-LE-diab_DEBUG/math.o
echo "building ARM-0x04000000-LE-diab_DEBUG/addone.o";das -tARMLS:simple -DTOOL_FAMILY=diab -DTOOL=diab -DARMXX -DEMULATC
" "addone.s"
building ARM-0x04000000-LE-diab_DEBUG/addone.o
echo "building ARM-0x04000000-LE-diab_DEBUG/cdemo.elf"; dld -o "ARM-0x04000000-LE-diab_DEBUG/cdemo.elf" -tARMLS:simple cdemo-Af
_DEBUG/diabasm.o ARM-0x04000000-LE-diab_DEBUG/cdemo.o ARM-0x04000000-LE-diab_DEBUG/strutils.o ARM-0x04000000-LE-diab_DEBI
UG/linklist.o ARM-0x04000000-LE-diab_DEBUG/date.o ARM-0x04000000-LE-diab_DEBUG/math.o ARM-0x04000000-LE-diab_DEBUG/addone
4000000-LE-diab_DEBUG/cdemo.elf;fi
building ARM-0x04000000-LE-diab_DEBUG/cdemo.elf
make: built targets of C:/WindRiver/workspace/c_demo_sa
Build Finished in Project 'c_demo_sa': 2006-09-27 15:56:25 (Elapsed Time: 00:12)
```

→ **NOTE:** When using projects other than the supplied demonstration projects: you must compile your programs using debugging symbols (the `-g` compiler option) to use most debugger features. The compiler settings used by the Wind River Workbench project facility's Managed Builds include debugging symbols. However, Workbench does not support code compiled with `-O2` optimization.

9.4 Downloading Code and Symbol Information

To run the sample code, right-click on `cdemo.elf` in the **Project Navigator** view and select **Reset and Download**.

The **Reset and Download** view appears.

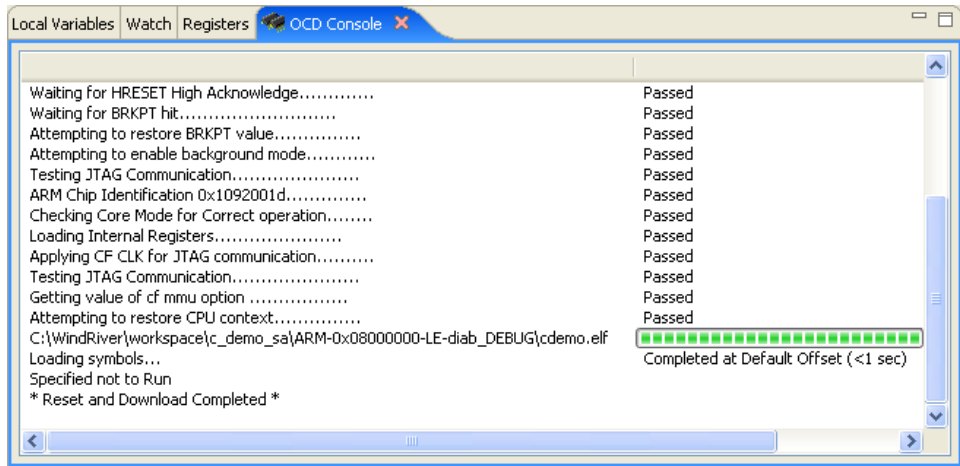


9

When opened from this folder, the **Reset and Download** view is pre-configured for this project.

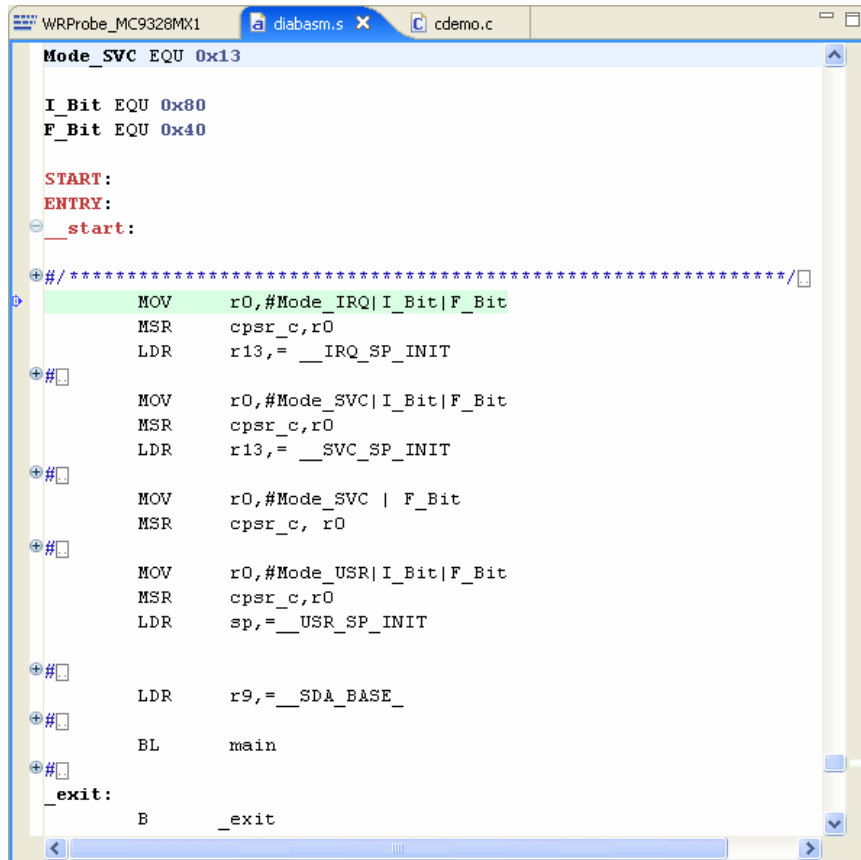
Leave the settings at their defaults and click **Debug**.

The **OCD Console** view opens.



The **OCD Console** view shows the progress of the download operation, as Workbench downloads the sample code to the target.

The Editor opens showing the Program Counter set at the beginning of the application code.



```
WRProbe_MC9328MX1  diabasm.s  demo.c
Mode_SVC EQU 0x13

I_Bit EQU 0x80
F_Bit EQU 0x40

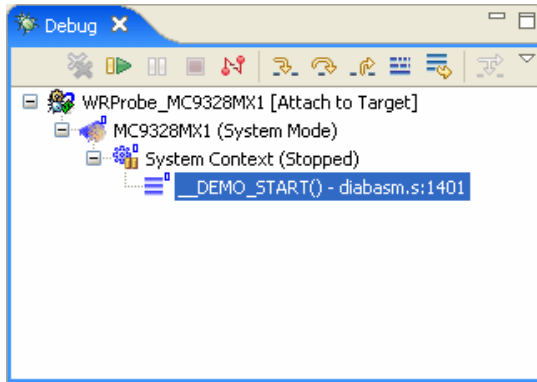
START:
ENTRY:
_start:
+ / *****/
MOV r0,#Mode_IRQ|I_Bit|F_Bit
MSR cpsr_c,r0
LDR r13,=__IRQ_SP_INIT
+ #[]
MOV r0,#Mode_SVC|I_Bit|F_Bit
MSR cpsr_c,r0
LDR r13,=__SVC_SP_INIT
+ #[]
MOV r0,#Mode_SVC | F_Bit
MSR cpsr_c, r0
+ #[]
MOV r0,#Mode_USR|I_Bit|F_Bit
MSR cpsr_c,r0
LDR sp,=__USR_SP_INIT
+ #[]
LDR r9,=__SDA_BASE_
+ #[]
BL main
+ #[]
_exit:
B _exit
```

You are now ready to run and debug the application.

9.5 Debugging Code in RAM

Use the **Debug** view to monitor, control, and manipulate the processes and tasks that you are actively debugging. The **Debug** view shows only the processes that are currently under debugger control.

Figure 9-1 **Debug View**



9.5.1 Monitoring Processes

When you start processes under debugger control, or attach the debugger to running processes, they appear in the **Debug** view labeled with unique colors and numbers. You can change the color assigned to a process or thread by right-clicking the process or thread and selecting **Color** > *specific color*.

9.5.2 Stepping Through Code

The Editor shows the source file **diabasm.s**, showing the **c_demo_sa** project's initialization assembly.

In the **Debug** view, click the **Step Into** button.

The Program Counter moves to the second assembly instruction. If you open the **Memory** view or the **Registers** view, you can see them update memory and register values as you step through instructions.

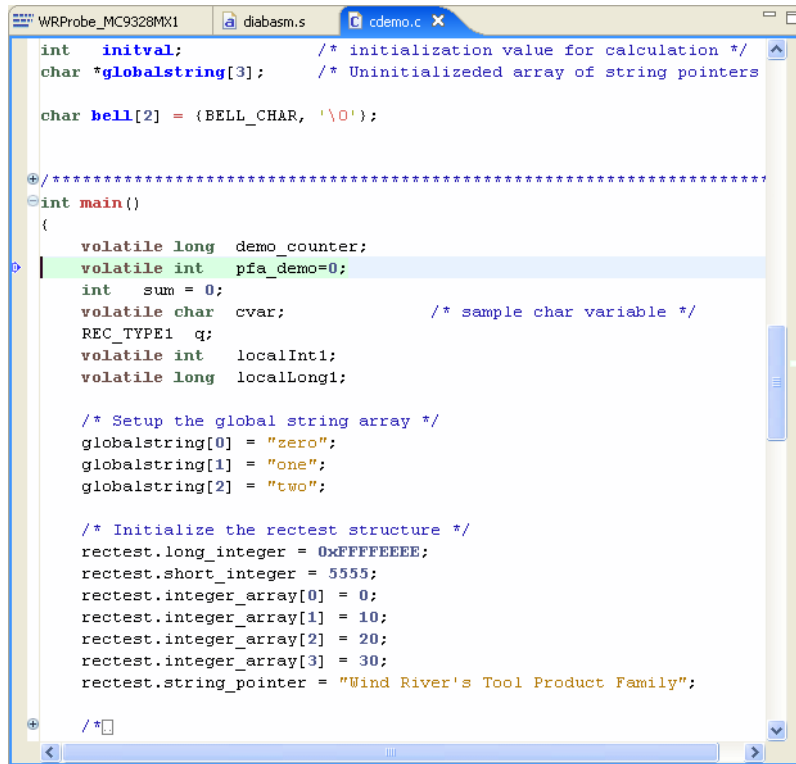
Click the **Step Into** button eleven more times, to step through all the initialization code and reach the first branch instruction:

```
BL main
```

This is where the application branches out of assembly into C code.

Click the **Step Into** button again.

The application branches into **main()** and the Editor opens the source file **cdemo.c**.



```
WRProbe_MC9328MX1  diabasm.s  cdemo.c x
int  initval;          /* initialization value for calculation */
char *globalstring[3]; /* Uninitialized array of string pointers

char bell[2] = (BELL_CHAR, '\0');

+-----+
int main()
{
    volatile long  demo_counter;
    volatile int   pfa_demo=0;
    int  sum = 0;
    volatile char  cvar;          /* sample char variable */
    REC_TYPE1  q;
    volatile int   localInt1;
    volatile long  localLong1;

    /* Setup the global string array */
    globalstring[0] = "zero";
    globalstring[1] = "one";
    globalstring[2] = "two";

    /* Initialize the rectest structure */
    rectest.long_integer = 0xFFFFFFFF;
    rectest.short_integer = 5555;
    rectest.integer_array[0] = 0;
    rectest.integer_array[1] = 10;
    rectest.integer_array[2] = 20;
    rectest.integer_array[3] = 30;
    rectest.string_pointer = "Wind River's Tool Product Family";

    /*
```

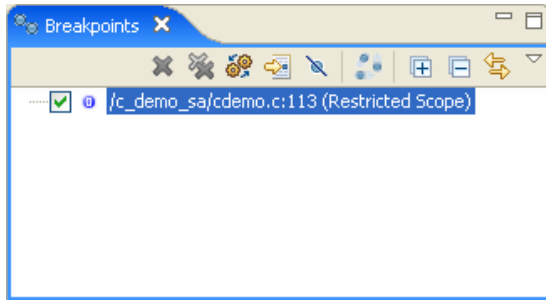
9.5.3 Setting a Software Breakpoint

Breakpoints allow you to stop a running program at particular places in the code or when specific conditions exist. For a full explanation of Workbench breakpoints, see *B. Internal Breakpoint Capabilities*.

In the left ruler of the Editor (the gutter), double-click to the left of the source line `globalstring[2] = "two";`

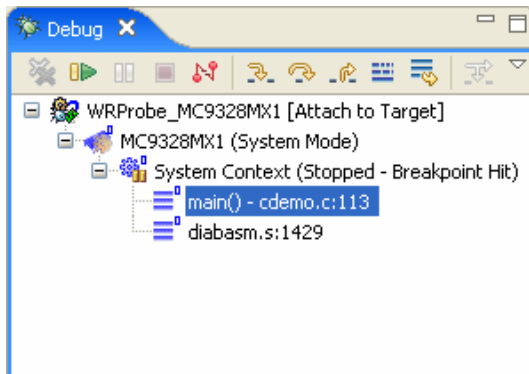
This sets a software breakpoint on that source line. The breakpoint appears in the **Breakpoints** view.

Figure 9-2 **Planted Software Breakpoint**



In the **Debug** view, click the **Resume** button. The program runs until it hits the breakpoint. The **System Context** changes to **Stopped -- Breakpoint Hit**.

Figure 9-3 **System Context -- Stopped**



Breakpoint information also appears in the **OCD Command Shell**:

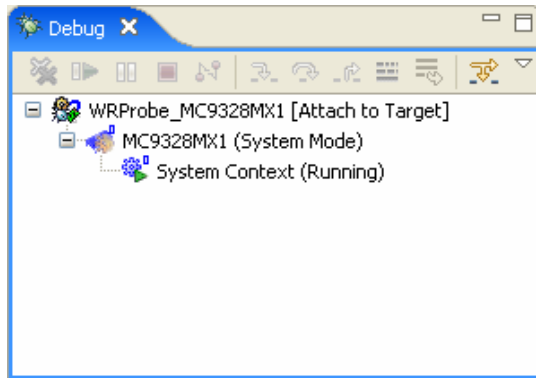
```
>RUN>  
  
!BREAK! - [msg12000] Software breakpoint; PC = 0x0800848c [EVENT Taken]  
>BKM>
```

9.5.4 Running a Program

To run your downloaded program, click **Resume** in the **Debug** view. The program will run until it hits a breakpoint. If there are no breakpoints or interrupts, the program will run to completion or until you click **Suspend**.

When the program is running, the System Context changes to **Running**, as shown in Figure 9-4, and a `>RUN>` prompt appears in the **OCD Command Shell**.

Figure 9-4 **System Context -- Running**



If there are no breakpoints, you can stop the program by clicking the **Suspend** button in the **Debug** view or by entering the `HA` command at the `>RUN>` prompt in the **OCD Command Shell**.

The Editor updates to show the current location of the Program Counter and the **System Context** in the **Debug** view changes to **Stopped -- User Request**.

9.5.5 Stepping Through a Program

To single-step without going into other subroutines, click **Step Over** instead of **Step Into**.

While stepping through a program, you may conclude that the problem you are interested in lies in the current subroutine's caller, rather than at the stack level where your process is suspended. In this situation, if you click **Step Return**, execution continues until the current subroutine completes, then the debugger regains control in the calling statement.

9.5.6 Setting a Hardware Breakpoint

The availability of hardware breakpoints varies by architecture. You can only set as many hardware breakpoints as there are debug registers available on your target.

Once a hardware breakpoint is trapped, the debugger will behave in the same way as for a standard breakpoint and stop for user interaction.

For a full description of hardware breakpoints in Workbench, see [B. Internal Breakpoint Capabilities](#).

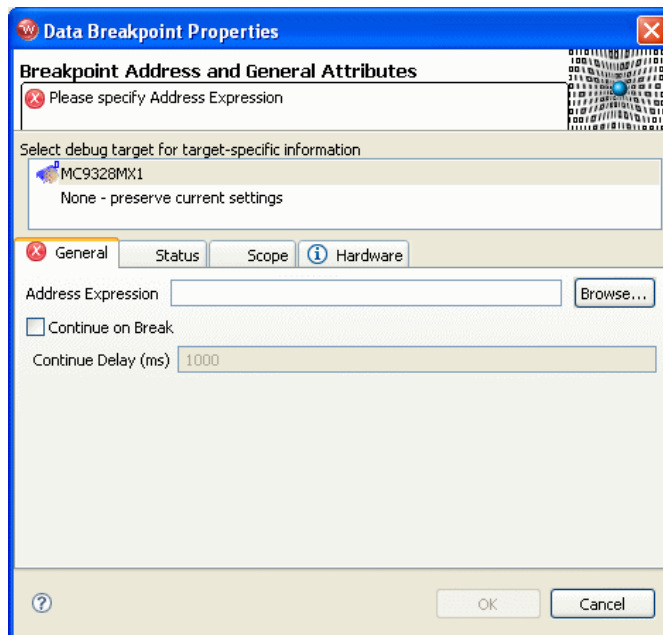
In the **Breakpoints** view, click on the **Menu** button and select **Add Data Breakpoint**.

The **Data Breakpoint** dialog appears, as shown in [Figure 9-5](#).

If an error message appears, you may have exceeded the number of allowed hardware breakpoints (four for most targets). Right-click in the **Breakpoints** view and select **Remove All**. Then select **Menu > Add Data Breakpoint** again.

If an error message still appears, your target may not support hardware breakpoints.

Figure 9-5 **Data Breakpoint Dialog**



You can use data hardware breakpoints to find out which routines are modifying a specific variable.

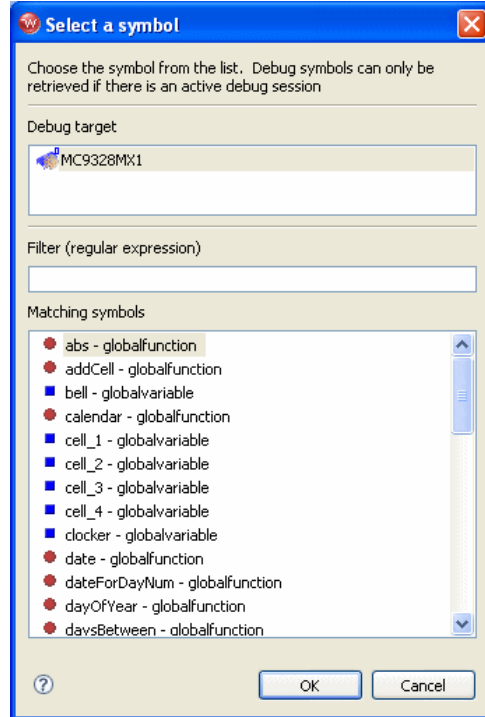
The **Address Expression** can be a symbol or a specific address in hex. You can use the address **0x0** in the **Address Expression** field to set a data hardware breakpoint to catch null pointers. You can set the **Address Expression** field to an address in the stack area to set a data hardware breakpoint to find out if the stack grew to that point.

The following example sets a symbol in the **Address Expression** field.

1. Click **Browse**.

The **Select Symbol** dialog appears, showing a list of available symbols that can take a hardware breakpoint.

Figure 9-6 **Select Symbol**

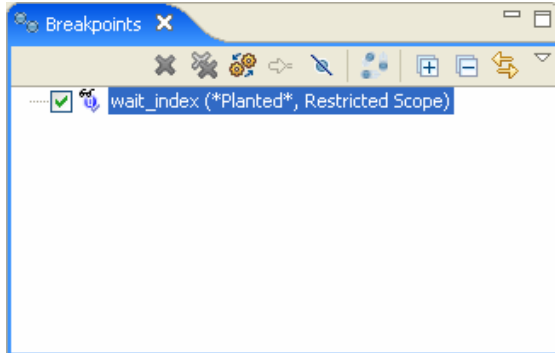


2. Scroll down and highlight the symbol **wait_index**.
3. Click **OK**.

The global variable **wait_index** is now the address for the data hardware breakpoint.

The hardware breakpoint on **wait_index** appears in the **Breakpoints** view.

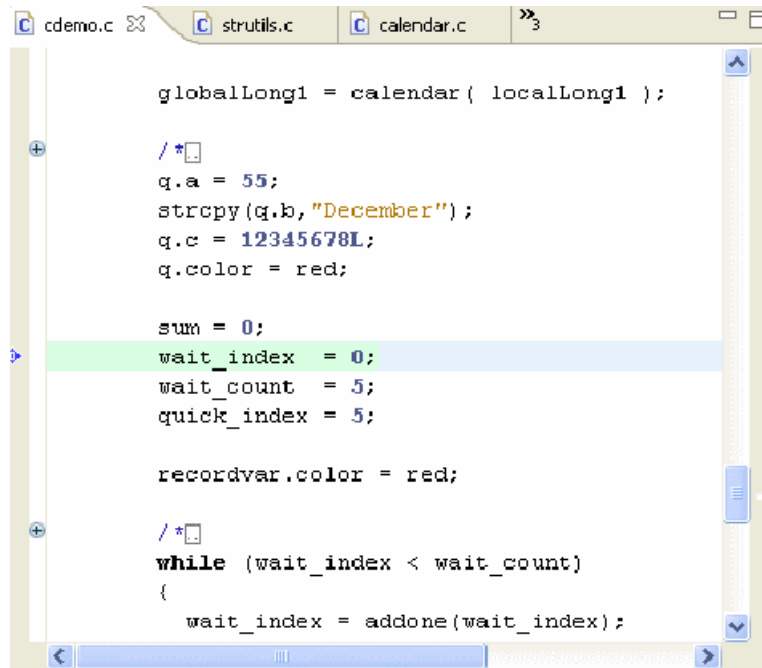
Figure 9-7 **wait_index**



In the **Debug** view, click **Resume**.

The program runs until it hits the hardware breakpoint. Workbench halts the processor when it locates **wait_index** and displays that source line in the Editor.

Figure 9-8 Hardware Breakpoint Hit



```
cdemo.c  strutils.c  calendar.c  >>3

globalLong1 = calendar( localLong1 );

/*
q.a = 55;
strcpy(q.b, "December");
q.c = 12345678L;
q.color = red;

sum = 0;
wait_index = 0;
wait_count = 5;
quick_index = 5;

recordvar.color = red;

/*
while (wait_index < wait_count)
{
    wait_index = addone(wait_index);
```

9

9.5.7 Disconnecting and Terminating Processes

Disconnecting from a process or core detaches the debugger, but leaves the process or core in its current state.

Terminating a process actually kills the process on the target.



NOTE: If the selected target supports terminating individual threads, you can select a thread and terminate only that thread.

10

Programming Flash Memory

- 10.1 Introduction 65
- 10.2 Testing Flash Workspace 66
- 10.3 Getting Started 67
- 10.4 Flash Configuration Tab 68
- 10.5 Flash Programming Tab 70
- 10.6 Flash Memory/Diagnostics Tab 76

10.1 Introduction

In order to erase and program target flash memory, you must first set up your target registers properly, as described in [6. Board Initialization](#).

The **Flash Programmer** view provides the ability to flash images into flash chips present on your target.

To program flash correctly you need to know the physical characteristics of your flash bank. For instance, your target may have one flash device connected to a 64-bit bus. Or it may have a bank of several flash devices, for example two flash devices, each wired at 16 bits, connected along a 32-bit bus. If you are using a Wind River-supported target, this information can be found in the file

installDir/vxworks-6.x/target/config/yourTarget/target.ref

If you are not using a Wind River-supported target, consult your target's documentation. The design primitives of your target board should be included in its board specification and schematics.

10.2 Testing Flash Workspace

The flash programming algorithm needs to run on the target. This requires a RAM workspace, to which the algorithm will download, and breakpoints, which are used to stop an erase and program operation at completion.

Reading and Writing Memory

Once you have established communications with the target, use the following procedure to make sure you can write to and read from the target. In this example we assume that the RAM workspace is 0x00F00200.



NOTE: A RAM workspace address of 0x00F00200 is not appropriate for all targets. For Wind River-supported targets, you can find the necessary RAM workspace in your target's **target.ref** file, located in *installDir/vxworks-6.x/target/config/yourTarget/target.ref*.

Wherever the RAM workspace is located on your target, you must make sure that memory is writable there.

At the **>BKM>** prompt, enter **dm 00f00200** and press **ENTER**. Doing so displays the memory on your target at address 0.

Next, enter **sm 00F00200 1234** and press **ENTER** to set the memory at address 0 to the value 1234. Enter **dm 00F00200** to display the memory at that address again.

If you are communicating properly with your target, output is similar to that shown below:

```
>BKM>dm 00f00200
00F00200: FF7C EFFE FEFF E3FE 0D01 0FBE F0FD BFB6 .|.....
>BKM>sm 00F00200 1234
>BKM>dm 00f00200
00F00200: 1234 EFFE FEFF E3FE 0D01 0FBE F0FD BFB6 .4.....
```

>BKM>

Occasionally, you may have difficulty programming flash memory on your target if software breakpoints are not being hit properly. Test this functionality before you continue.

To use the test, enter the following commands at the >BKM> prompt in the **OCD Command Shell**:

```
>BKM>df e 0

>BKM>di 0 6
$00000000 : 0xFFFFFFFF :arm swinv 0x00ffffff
$00000004 : 0xFFFFFFFF :arm swinv 0x00ffffff
$00000008 : 0xFFFFFFFF :arm swinv 0x00ffffff
$0000000C : 0xFFFFFFFF :arm swinv 0x00ffffff
$00000010 : 0xFFFFFFFF :arm swinv 0x00ffffff
$00000014 : 0xFFFFFFFF :arm swinv 0x00ffffff
>BKM>go 0
>RUN>dr pc
PC = 00000004
>RUN>dr pc
PC = 00000010
>RUN>sb 8

>RUN>

!BREAK! - [msg12000] Software breakpoint; PC = 0x00000008 [EVENT Taken]
>BKM>
>BKM>rb

>BKM>
```

10

10.3 Getting Started

Once you have connected to Wind River Workbench, as described in your emulator's Hardware Reference, and configured your target registers, as described in [6. Board Initialization](#), you are ready to begin programming flash.

1. In the toolbar, click on **Window**, then select **Show View > Flash Programmer**.

The **Flash Programmer** view appears.

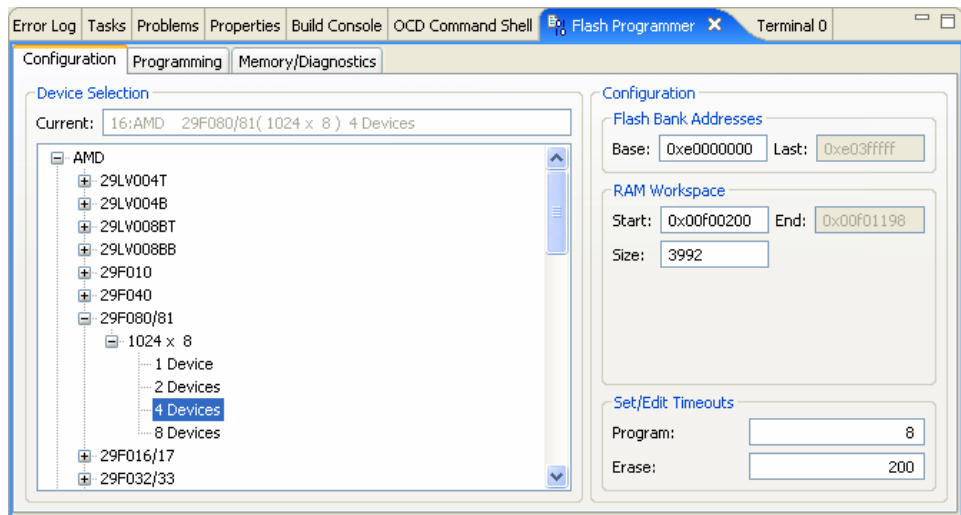
The **Flash Programmer** view has three tabs: **Configuration**, **Programming**, and **Memory/Diagnostics**. Use these tabs to configure your flash address and RAM

workspace, choose files for download, execute erase and program operations, and check the results of your operations.

10.4 Flash Configuration Tab

Use the **Configuration** tab to configure the base address and workspace address for flash memory erase operations. You can also enter the physical description of your flash devices.

Figure 10-1 **Configuration Tab**



10.4.1 Selecting a Flash Driver

In the **Device Selection** field, browse to a description of your flash bank. [Figure 10-1](#) shows an example of a flash bank consisting of four 8-bit AMD 29F0808 devices.



NOTE: For AMD flash devices, “F” and “LV” devices are interchangeable in Workbench.

If you attempt to move on to the **Programming** tab without selecting a flash bank description in the **Configuration** tab, Workbench displays an Invalid Flash Bank error and returns you to the **Configuration** tab.

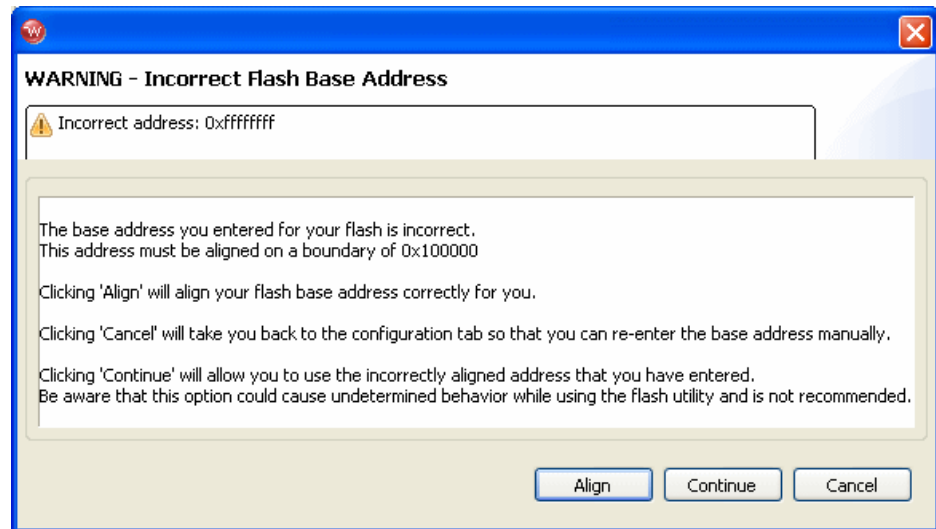
10.4.2 Configuring Flash Memory Bounds

In the **Configuration** field, enter the **Base** value for the area of flash memory you wish to erase. In [Figure 10-1](#) the address used is 0xe0000000. The **Last** field populates automatically.

→ **NOTE:** Workbench erases flash memory sector by sector. That means that no matter where the address you enter in the **Base** field is located within the flash sector, Workbench will still erase the entire sector.

If Workbench detects that the address you entered in the **Base** field is not correctly aligned with the flash sector boundary, it displays the following warning message:

Figure 10-2 **Incorrect Flash Base Address**



- To have Workbench align your base address, click **Align**. Workbench aligns the base address with the nearest preceding sector boundary.
- To go back to the **Configuration** tab and re-enter the address manually, click **Cancel**.

- To use the base address as you entered it, without aligning it with the flash boundary, click **Continue**.



CAUTION: Choosing **Continue** may cause unpredictable results in your flash programming operations. Wind River recommends that you align the base address with the flash sector boundary.

10.4.3 Configuring RAM Workspace

The flash programming algorithm needs to run on the target. This requires a RAM workspace, to which the algorithm will download.

In the **RAM Workspace** field, enter the **Start** value for the area of RAM you wish to use as the workspace. In the **Size** field, enter the desired size of the workspace in bytes. In [Figure 10-1](#) the starting address used is 0x00F00200 and the workspace size is 3992. The **End** field populates automatically.



NOTE: A RAM workspace address of 0x00F00200 is not appropriate for all targets. For Wind River-supported targets, you can find the necessary RAM workspace in your processor's **target.ref** file, located in *installDir/vxworks-6.x/target/config/yourTargetBoard/target.ref*, or **target.ref.linux** file, located at <http://www.windriver.com/support>.

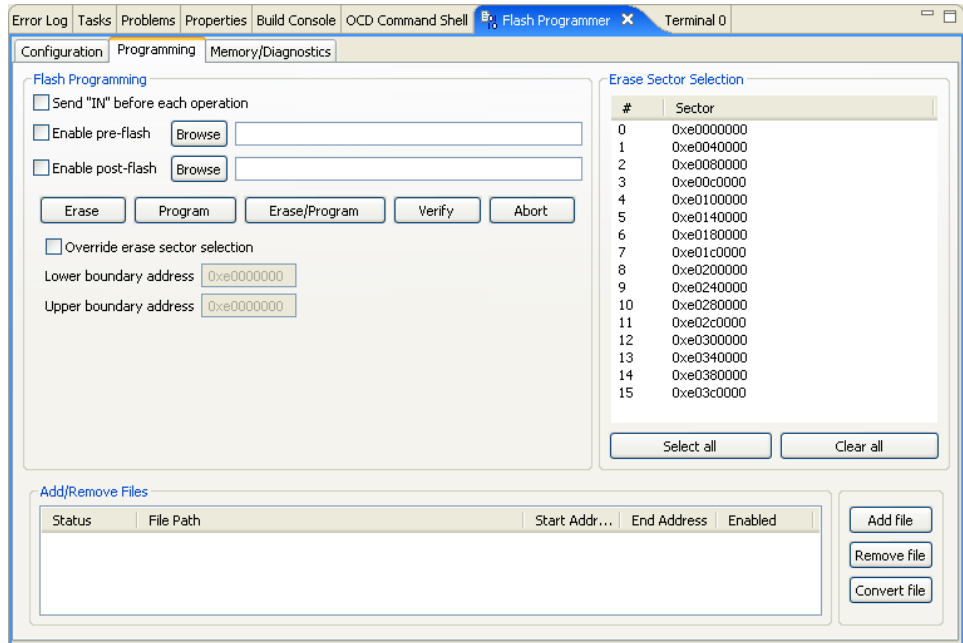
10.4.4 Setting Timeouts

To set a program or erase timeout, use the **Program** or **Erase** fields in the **Set/Edit Timeouts** area. Enter a timeout value in seconds. If you enter an invalid number, Workbench resets the timeout to its default setting.

10.5 Flash Programming Tab

Use the **Programming** tab to execute erase and program operations in flash and to specify files for download.

Figure 10-3 Programming Tab



10

10.5.1 Erasing and Programming Flash

To issue an IN initialization command before erase or program operations, select the **Send "IN" before each operation** checkbox.

Click **Erase** to erase the contents of the flash memory sectors you selected in the **Configuration** tab.

Click **Program** to program the flash memory with the files you selected in the **Add/Remove Files** area of the **Programming** tab.

Click **Erase/Program** to perform both operations. Workbench will erase all selected flash sectors before programming.

Click **Abort** to stop the erase or program operation.

10.5.2 Verifying Flash Contents

Click **Verify** to execute a byte-by-byte comparison between the file you just downloaded and the file already in memory. If there is a discrepancy, Workbench will break at that address and deliver an error message.

10.5.3 Running a Pre- or Post-Flash Script

You can specify a script to run before or after an erase or program operation. Select the **Enable pre-flash** or **Enable post-flash** checkboxes (you can select either or both for any operation). Next to the checkbox, click **Browse** and navigate to the script you wish to run.

10.5.4 Selecting Flash Sectors for Erasure

The **Sectors** field automatically populates with the starting addresses of sectors of flash memory, depending on which flash device you specified in the **Configuration** tab. Click on a sector to select it. You can select all sectors by clicking **Select All**. Click **Clear All** to deselect all sectors.

Before you erase all sectors, make sure you know what resides in the flash. For example, if your target processor reads its reset configuration word from the flash device, erasing the entire device may cause problems with resetting the board.

10.5.5 Manually Configuring Flash Memory Erasure Bounds

Workbench allows greater user control by allowing manual configuration of the flash memory bounds for erase operations.

You can manually configure the flash memory bounds by checking the **Override erase sector selection** checkbox. When this box is checked, Workbench will allow you to enter any addresses in the **Lower boundary address** and **Upper boundary address** fields.



NOTE: If the values you enter result in a memory address range that is outside your target board's flash programming area, erase operations will not perform correctly.

10.5.6 Adding Files

To add a **.bin** file, click **Add File**. This opens the **Choose File for Flash Download** browser window. Workbench automatically looks for a folder labeled **firmware**, located in *installDir/workbench-2.x/dfw/version/host/firmware*, where *version* is the installed version of the debugger middleware. If your **.bin** files are stored in another folder, use the browser to navigate to it. Select the file you want and click **Open**. The file will appear in the **File Path** field.

10.5.7 Removing Files

To remove a file from the list, highlight it and then click **Remove File**.

10.5.8 Converting Files To Wind River Flash Binary Format

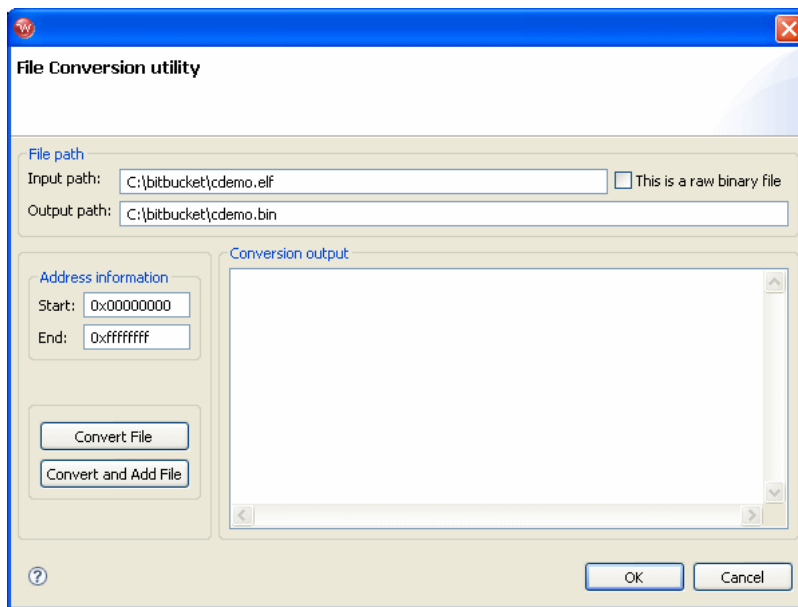
In order to use a file to program flash, you must convert it to a Wind River binary format that the **Flash Programmer** can use. Workbench can convert any of the following file types to Wind River binary format:

- **elf** files
- **hex** files
- **srec** files
- any headerless flat binary (RAWBIN) file

To convert a file to Wind River binary format, use the following steps:

1. In the **Programming** tab, select **Convert File**.
2. In the browser window that opens, navigate to the file you want to convert and click **Open**.

The **Convert** utility appears.

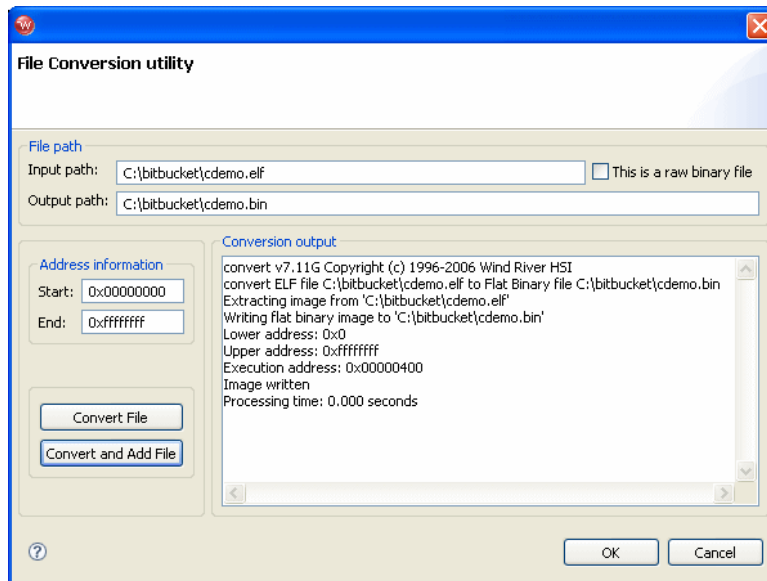


Converting the file to Wind River binary format does not delete the original file.

By default, Workbench stores the new binary file in the same location as the original file. If you want the new binary file stored somewhere else, enter the path to the desired location in the **Output path** field.

3. Select **Convert and Add File**.

Workbench converts the selected file to Wind River binary format and adds it to the file list in the **Programming** tab.



NOTE: To convert the selected file to Wind River binary format without adding it to the file list in the **Programming** tab, select **Convert File**.

4. Click **OK**.

You are returned to the **Programming** tab. The file you just converted now appears in the **File Path** field.

10.5.9 Setting The Download Offset Of A File

In some cases, before you program the file into flash, you may need to set a memory offset bias to divert the data to other areas of the flash bank.

Each file is built with a start address. This start address may or may not be the address where you want the image to reside on the board. If you subtract the start address of the image from the address where you want the image to reside on the board, then you end up with the proper bias address.

For example, if the image was built with a start address of 0x10000000 and you wanted the image to reside at the reset vector 0x10000100, then the offset bias would be 0x00000100.

You can use the **Add/Remove Files** area to edit the starting address of a **.bin** file to offset the file into flash. Click on the value under the **Start Address** heading to highlight it. Edit the value as needed.

10.5.10 Enabling A File For Download

Enable a file by clicking on the checkbox under the **Enabled** heading. If the file address is outside your specified address range, an error message appears:

```
Cannot enable for download.  
Part of this file falls outside your flash address range.
```

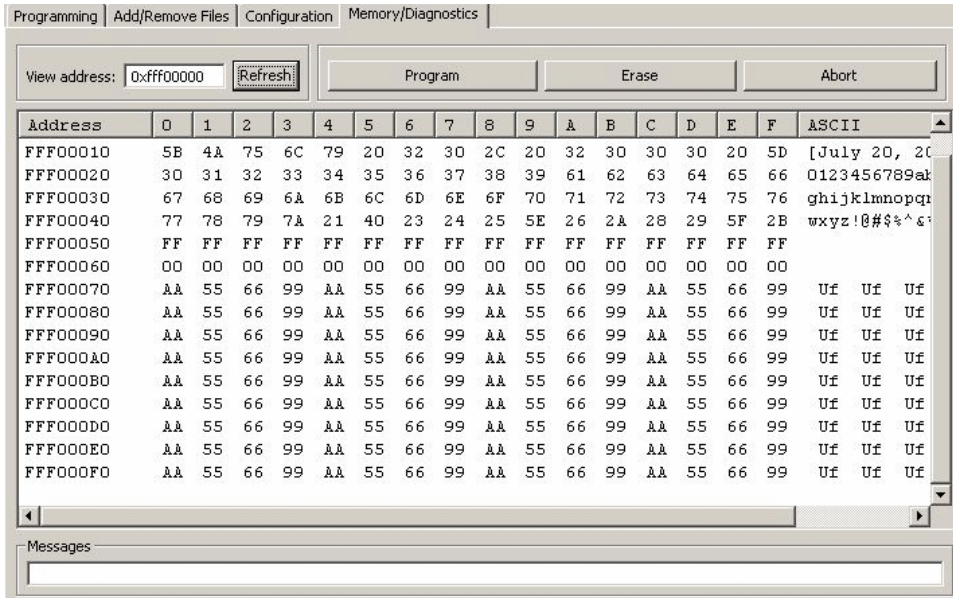
To correct this error, you must either change the start address of your file or use the **Configuration** tab to change your flash address range.

10.6 Flash Memory/Diagnostics Tab

Use the **Memory/Diagnostics** tab to view the contents of flash memory and to run diagnostic tests to verify your ability to write and erase flash.

You must set up the **Configuration** tab before using the **Memory/Diagnostics** tab.

Figure 10-4 Memory/Diagnostics Tab



10.6.1 Viewing Memory

Enter the address you wish to view in the **View Address** field. The area below displays the bit-level detail. To change the view, edit the address in the **View Address** field and click **Refresh**. You can also use the scrollbar on the right to scroll up and down from the starting address to the end address.

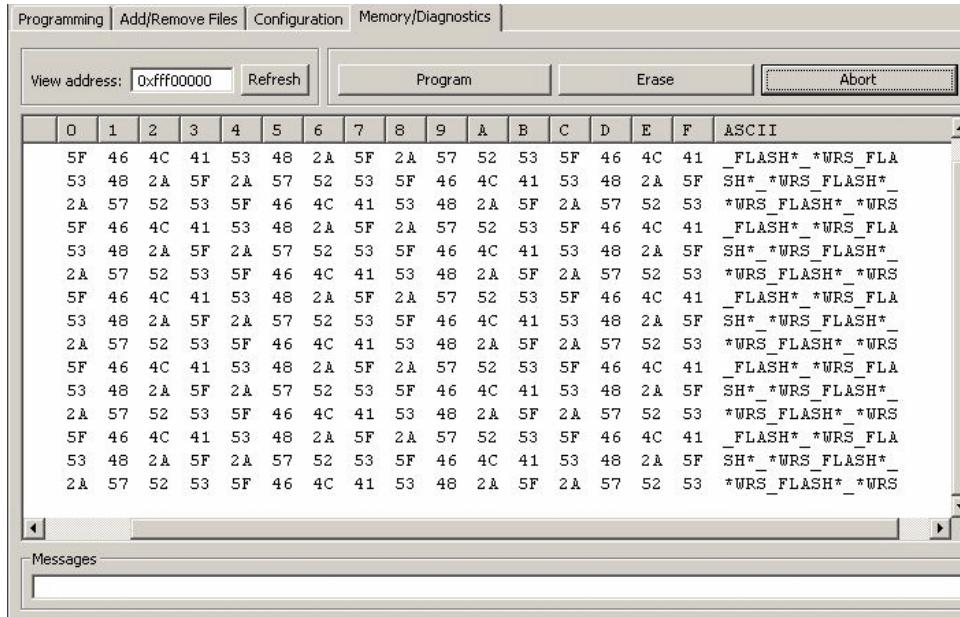
10.6.2 Running Diagnostic Tests

To test your ability to write to flash memory, click the **Start Program Diagnostic** button. This writes a bit pattern to flash.

You may see a **Target Exception** message. This requires no action.

If the write operation is successful, you should see the pattern ***WRS_FLASH*** repeated under the **ASCII** heading in the **Memory/Diagnostics** tab, as shown in [Figure 10-5](#).

Figure 10-5 Successful Program Diagnostic



If the write operation is unsuccessful, the diagnostic will never complete. You will need to click the **Abort Diagnostic** button to stop the write operation. Check to make sure that you have the right flash device selected in the **Device Selection** area in the **Configuration** tab, and that you are using the correct base address.

To test your ability to erase flash memory, click the **Start Erase Diagnostic** button. This will erase the selected flash sectors.

You may see a **Target Exception** message. This requires no action.

If the erase operation is successful, the selected sectors will be erased and the space under the **ASCII** heading in the **Memory/Diagnostics** view will be empty.

If the erase operation is unsuccessful, the diagnostic will never complete. You will need to click the **Abort Diagnostic** button to stop the erase operation. Check to make sure that you have the right flash device selected in the **Device Selection** area in the **Configuration** tab, and that you are using the correct base address.

11

Debugging in ROM

11.1 Overview 79

11.2 Getting Started 80

11.3 Debugging in ROM 80

11.1 Overview

The procedure described in *9. Debugging in RAM* uses software breakpoints. Software breakpoints work by replacing the destination instruction with an interrupt; therefore it is impossible to debug code in ROM using software breakpoints.

To debug code in ROM you must use hardware breakpoints, which work by setting a break condition and comparing the condition with the execution stream. This chapter describes using Workbench to debug code in ROM with hardware breakpoints.

11.2 Getting Started

To debug code in ROM, you must have an active target connection.

To create an active target connection, follow the steps described in [4. *OCD Connections*](#).

You do not need to have an active project to debug code in ROM.

11.3 Debugging in ROM

Use the **Debug** view to monitor, control, and manipulate the processes and tasks that you are actively debugging. The **Debug** view shows only the processes that are currently under debugger control.

1. In the **Target Manager** view, right-click on your target name and select **Attach to Core**.

The **Disassembly** view opens, with the Program Counter set to the start of the reset vector.



NOTE: The reset vector will vary between target processors. The example used in this chapter is only one of many possible examples.

```

System Context
00000000: 11    r3, 2
fff00104: nop
fff00108: bl    0xFFFF00138
fff0010c: bcl   0x1E, 0xF, 0xFFFF07184
fff00110: andi. r9, r19, Cx67e8
fff00114: andis. r0, r1, 0x3135
fff00118: addi  r1, r20, Cx2D32
fff0011c: addic r1, r16, Cx3220
fff00120: clrslwi r9, r27, Cx26, 0xD
fff00124: subfic r2, r18, Cx6976
fff00128: oris  r18, r11, 0x2C53
fff0012c: .lorq 0x75737465
fff00130: xoris r19, r11, 0x2C20
fff00134: ba    0x1eE632C
fff00138: mr    r11, r3
fff0013c: xor   r4, r4, r4
fff00140: mr    r0, r4
fff00144: isync
fff00148: mtmsr r4
fff0014c: ioyrc
fff00150: xor   r0, r0, r0
fff00154: mtspr sprg0, r0
fff00158: mtspr sprg1, r0
fff0015c: mtspr sprg2, r0
fff00160: mtspr sprg3, r0
00000164: xor   r4, r4, r4
fff00168: mtsr  0, r4
  
```

2. In the **Target Manager** view, select **OCD Reset and Download**.
3. Select the **Reset** tab.
4. Set the reset type to **INN -- Reset**.

This will initialize the target, but leave the target registers as close to reset value as possible.



NOTE: Because the target registers are not set, the target software watchdog timers are still active. This can cause some targets to drop out of background mode.

5. Leave the **Play register file** box unchecked.
6. Select the **Download** tab.
7. Click **Add Files...**

8. In the browser window that appears, navigate to the boot ROM file for your target.

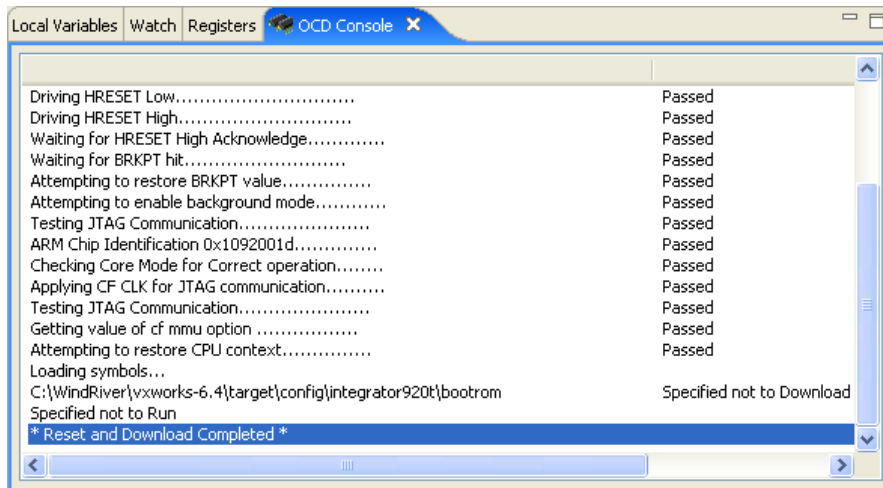
If you are using a Wind River-supported target, the boot ROM file is located in *installDir/vxworks-6.x/target/config/your_target*. If you are supplying your own boot ROM file, navigate to the directory where it is located.

9. Click **Open**.

You are returned to the **Download** tab.

10. Uncheck the **Download** checkbox.
11. Make sure the **Load Symbols** checkbox is selected and the **Verify** field is set to **None**.
12. Select the **Instruction Pointer** tab.
13. Uncheck the **Set instruction pointer after download** checkbox.
14. Select the **Run Options** tab.
15. Make sure the **Do not run** checkbox is selected.
16. Click **Debug**.

The **OCD Console** view opens to show the results.



11.3.1 Stepping Through Boot Code

In this example, the first line of the reset vector is a Load Immediate command:

```
fff00100 li    r2, r3
```

1. In the **Debug** view, click the **Step Into** button.

The Program Counter moves to the No-Operation command on the next line:

```
fff00104 nop
```

In the **Debug** view, the **System Context** changes to **Stopped -- Step End**, and the view updates to show the new location of the Program Counter.

If you have data views, such as the **Watch** view, **Memory** view, or **Registers** view open, you can see them update as you step through code.

2. In the **Debug** view, click **Step Into** again.

The Program Counter moves to the Branch and Link instruction on the next line:

```
fff00108 bl    0xFFFF00138
```

3. In the **Debug** view, click **Step Into** one more time.

The branch instruction executes and the Program Counter jumps to address FFF00138. The **Debug** view updates to show the Program Counter at address FFF00138.

11.3.2 Setting Hardware Breakpoints

Breakpoints allow you to stop a running program at particular places in the code or when specific conditions exist. Use the **Breakpoints** view to keep track of your breakpoints and their conditions.

To debug in ROM you must use hardware breakpoints. The availability of hardware breakpoints varies by architecture. You can only set as many hardware breakpoints as there are debug registers available on your target.

Once a hardware breakpoint is trapped, the debugger will behave in the same way as for a standard breakpoint and stop for user interaction.

For a full description of hardware breakpoints in Workbench, see [B. Internal Breakpoint Capabilities](#).

1. In the **Disassembly** view, right-click in the left ruler (the gutter) to the left of the Exclusive Or instruction at address FFF00150:

```
fff00150 xor    r0,r0,r0
```

2. From the context menu that appears, select **Add Hardware Breakpoint**.

The breakpoint appears in the **Disassembly** view and is displayed in the **Breakpoints** view.

3. In the **Debug** view, click the **Resume** button.

The code runs until it hits the hardware breakpoint at address FFF00150.

In the **Debug** view, the **System Context** changes to **Stopped --Breakpoint Hit**.

The following message appears in the **OCD Command Shell**:

```
>RUN>
```

```
!BREAK! - [msg11001] Internal hardware breakpoint; PC = 0xffff00150 [EVENT  
Taken]
```

```
>BKM>
```

4. To remove the hardware breakpoint, double-click on the breakpoint icon in the **Disassembly** view gutter, or right-click on the breakpoint in the **Breakpoints** view and select **Remove**.

A

Pins Mapped to Common Signals

[A.1 Introduction 85](#)

[A.2 ARM Processors -- JTAG 85](#)

A.1 Introduction

This appendix describes mapped pins to common signals for Wind River-supported ARM and ARMX (XScale) processor families.

For all families described in this appendix, “n” is set as an ACTIVE LOW.

A.2 ARM Processors -- JTAG

Table A-1 **ARM -- JTAG**

Pin Number	Function	Description
1	JTAG_VIO	JTAG Voltage Output

Table A-1 **ARM -- JTAG**

Pin Number	Function	Description
2	NC	Not Connected
3	nTRST	Test Reset (reset JTAG clock)
4	GND	Ground
5	TDI	Test Data In
6	GND	Ground
7	TMS	Test Mode Select
8	GND	Ground
9	TCK	Test Clock
10	GND	Ground
11	RTCK	Return Test Clock
12	GND	Ground
13	TDO	Test Data Out
14	GND	Ground
15	nRESET	Processor Reset
16	GND	Ground
17	DBGRQ	Debug Request
18	GND	Ground
19	DBGACK	Debug Acknowledge
20	GND	Ground

B

Internal Breakpoint Capabilities

Emulators use breakpoints to implement single stepping, since the embedded processor's single step mode, if it has one, is not useful for stepping through C code.

Software breakpoints work by replacing the destination instruction by a software interrupt. Therefore, it is impossible to debug code in ROM using software breakpoints.

Hardware breakpoints work by setting a break condition and comparing it against the execution stream. You can use hardware breakpoints to debug code in RAM, ROM, flash memory, or even unused address spaces.

Complex breakpoints involve conditions. An example might be, "Break if the program writes *value* to *variable* if and only if *function_name* was called first." A software-only debugger setting a complex breakpoint must interpret the program while watching for the trigger condition, which slows performance. Emulators implement complex breakpoints in hardware, so there is no performance penalty.

In Wind River Workbench, you can use the **Breakpoints** view to keep track of all breakpoints, along with any conditions.

You can create breakpoints in different ways: by double-clicking or right-clicking in the Editor's left overview ruler (also known as the gutter), by opening the various breakpoint dialogs from the pull-down menu in the **Breakpoints** view itself, or by selecting one of the breakpoint options from the **Run** menu.

Wind River Workbench supports three kinds of breakpoints: line breakpoints, expression breakpoints, and hardware breakpoints.

Line Breakpoints

Set a line breakpoint to stop your program at a particular line of source code.

To set a line breakpoint with an unrestricted scope (that will be hit by any process or task running on your target), double-click in the left gutter next to the line on which you want to set the breakpoint. A solid dot appears in the gutter, and the **Breakpoints** view displays the file and the line number of the breakpoint. You can also right-click in the gutter and select **Add Global Line Breakpoint**.

To set a line breakpoint that is restricted to just one task or process, right-click in the Editor gutter and select **Add Breakpoint for selected thread**. If the selected thread has a color in the **Debug** view, a dot with the same color will appear in the Editor gutter, with the number of the thread inscribed inside it.

Either of these actions opens the **Line Breakpoint** dialog, where you can create and adjust the properties of the breakpoint.

Expression Breakpoints

Set an expression breakpoint using any C expression that will evaluate to a memory address. This could be a function name, a function name plus a constant, a global variable, a line of assembly code, or just a memory address. Expression breakpoints appear in the Editor's gutter only when you are connected to a task.

Breakpoint conditions are evaluated after a breakpoint is triggered, in the context of the stopped task or process. Functions in the condition string are evaluated as addresses and are not executed. Other restrictions are similar to the C/C++ restrictions for calculating the address of a breakpoint using the **Expression Breakpoint** dialog.

Select **Add Expression Breakpoint** from the pull-down menu in the **Breakpoints** view to open the **Expression Breakpoint** dialog, where you can create and adjust the properties for the breakpoint.

Hardware Breakpoints

Some processors provide specialized registers called debug registers that can be used to specify an area of memory to be monitored. For instance, IA-32 processors have four debug address registers, which can be used to set data breakpoints or control breakpoints.

Hardware breakpoints are useful if you want to stop a process when a specific variable is written or read. For example, with hardware data breakpoints, a hardware trap is generated when a write or read occurs in a monitored area of memory. Hardware breakpoints are fast, but their availability is machine-dependent. On most ARM CPUs that do support them, only two to six debug registers are provided, so you can only watch a maximum of two to six memory locations in this way.

There are two types of hardware breakpoints:

- A hardware *data* breakpoint occurs when a specific variable is read or written.
- A hardware *instruction* breakpoint or *code* breakpoint occurs when a specific instruction is read for execution.

Once a hardware breakpoint is trapped—either an instruction breakpoint or a data breakpoint—the debugger will behave in the same way as for a standard breakpoint and stop for user interaction.

Adding Hardware Instruction Breakpoints

There two ways to add a new hardware instruction breakpoint:

In the gutter on the left of the source file, right-click and select **Add Hardware Code Breakpoint**. Alternately, double-click in the gutter to add a standard breakpoint and then, in the **Breakpoints** view, right-click the breakpoint you just added and select **Properties**. In the last pane (**Hardware**) of the **Properties** dialog, select **Enable Hardware Breakpoint**.

Adding Hardware Data Breakpoints

Set a hardware data breakpoint when:

- The debugger should break when an event (such as a read or write of a specific memory address) or a situation (such as data at one address matching data at another address) occurs.
- Threads are interfering with each other, or memory is being accessed improperly, or whenever the sequence or timing of runtime events is critical (hardware breakpoints are faster than software breakpoints).

Select **Add Data Breakpoint** from the pull-down menu in the **Breakpoints** to open the **Hardware Data Breakpoint** dialog, where you can create and adjust the properties for the breakpoint.

➔ **NOTE:** Some hardware breakpoint register implementations only support hardware instruction breakpoints in a subset of the address space. For example, the ARM Cortex-M3 only supports setting hardware instruction breakpoints in the flash memory region.

Converting Line or Expression Breakpoints Into Hardware Code Breakpoints

To cause the debugger to request that a line or expression breakpoint be a hardware code breakpoint, select the **Hardware** check box on the **Hardware** tab of the **Line Breakpoint** or **Expression Breakpoint** dialog.

This request does not guarantee that the hardware code breakpoint will be planted; that depends on whether the target supports hardware breakpoints, and if so, whether or not the total number supported by the target have already been planted. If the target does not support hardware code breakpoints, an error message will appear when the debugger tries to plant the breakpoint.

➔ **NOTE:** Workbench will set only the number of code breakpoints, with the specific capabilities, supported by your hardware.

➔ **NOTE:** If you create a breakpoint on a line that does not have any corresponding code, the debugger will plant the breakpoint on the next line that does have code. The breakpoint will appear on the new line in the Editor gutter. In the Breakpoints view, the original line number will appear, with the new line number in square brackets [] after it.

Importing Breakpoints

To import breakpoint properties from a file:

1. Select **File > Import > Import Breakpoints**, then click **Next**. The **Import Breakpoints** dialog appears.
2. Select the breakpoint file you want to import, then click **Next**. The **Select Breakpoints** dialog appears.
3. Select one or more breakpoints to import, then click **Finish**. The breakpoint information will appear in the **Breakpoints** view, and the next time the context for that breakpoint is active in the **Debug** view, the breakpoint will be planted.

Exporting Breakpoints

To export breakpoint properties to a file:

1. **Select File > Export > Export Breakpoints**, then click **Next**. The **Export Breakpoints** dialog appears.
2. Select the breakpoint whose properties you want to export, and type in a file name for the exported file. Click **Finish**.

Refreshing Breakpoints

Right-click a breakpoint in the **Breakpoints** view and select **Refresh Breakpoint** to cause the breakpoint to be removed and reinserted on the target. This is useful if something has changed on the target (for example, you downloaded a new module) and the breakpoint is not automatically updated.

To refresh all breakpoints in this way, select **Refresh All Breakpoints** from the pull-down menu in the **Breakpoints** view.

B

Disabling Breakpoints

To disable a breakpoint, clear its check box in the **Breakpoints** view. This retains all breakpoint properties, but ensures that it will not stop the running process. To re-enable the breakpoint, select the box again.

Removing Breakpoints

To remove a breakpoint:

- Right-click on a breakpoint in the Editor gutter and select **Remove Breakpoint**.
- Select a breakpoint in the **Breakpoints** view and select **Remove**.

C

Pin Terminations

C.1 JTAG Pin Terminations

C.1.1 ARM 14-Pin JTAG Connector



NOTE: This is a legacy connector. It is not suitable for new designs.

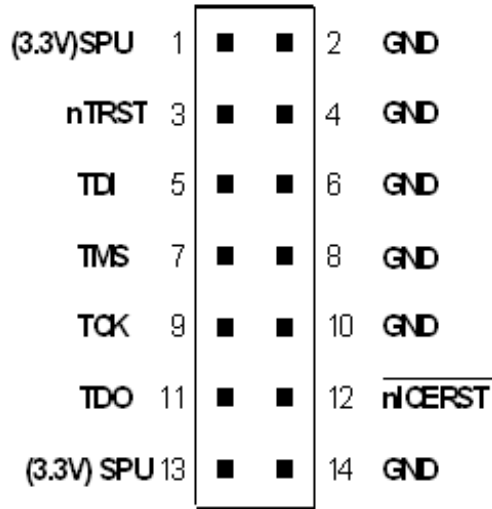
The connector type on your board should be as follows:

- 14 (2 by 7) 0.025" square posts
- 0.10" between centers of adjacent posts
- 0.23" height of each post

A sample connector is Samtec part number TSW-107-07-S-D

The pin-outs of this connector are shown in [Figure C-1](#).

Figure C-1 ARM 14-Pin JTAG Connector Pinouts



JTAG Terminations

The following table contains the Wind River-recommended pull-up/pull-down resistor values that must be placed on the target. Signals not shown should not have pull-ups or pull-downs.

Table C-1 ARM 14-Pin JTAG Terminations

Signal Name	Description
TCK - Test Clock	External 5K pull-down
TMS - Test Mode	External 5K pull-up on PCB
TDI - Test Data In	External 5K pull-up on PCB
TDO - Test Data Out	External 5K pull-up on PCB
nTRST - Reset TAP Controller	External 2K pull-up on PCB
$\overline{\text{nICERST}}$ (nPOR)	External 2K pull-up on PCB



NOTE: These are the termination values for the Wind River reference design. It is important that you verify that the pull-up and pull-down values you use are appropriate for the board that you are using.

C.1.2 ARM 20-Pin JTAG Connector

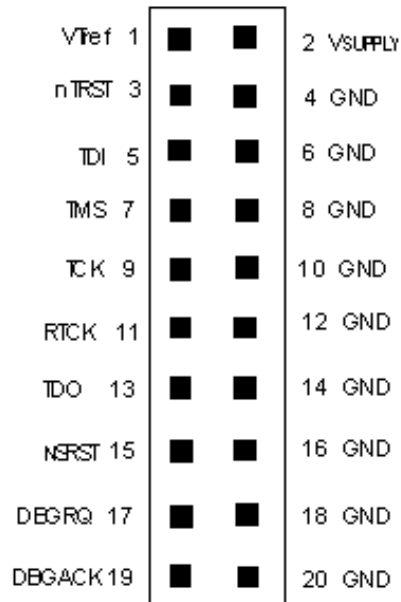
The connector type on your board should be as follows:

- 20 (2 by 10) 0.025" square posts
- 0.10" between centers of adjacent posts
- 0.23" height of each post

A sample connector is Samtec part number TSM-110-01-S-DV.

The pin-outs of this connector are shown in [Figure C-2](#).

Figure C-2 **ARM 20-Pin JTAG Connector**



JTAG pull-ups and pull-downs are not provided for the 20-pin connector. Please refer to the specifications from the board manufacturer to determine the appropriate pull-ups and pull-downs for your board.

C.1.3 ARMX (XScale) 20-Pin JTAG Connector

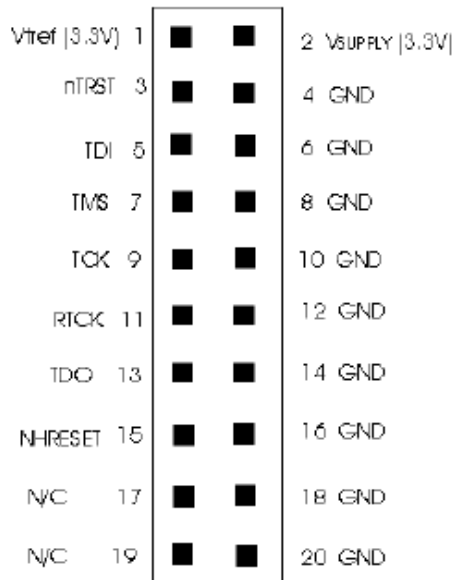
The connector type on your board should be as follows:

- 20 (2 by 10) 0.025" square posts
- 0.10" between centers of adjacent posts
- 0.23" height of each post

A sample connector is Samtec part number TSM-110-01-T-DV.

The pin-outs of this connector are shown in [Figure C-3](#).

Figure C-3 ARMX 20-Pin JTAG Pinout



JTAG Terminations

The following table contains the Wind River-recommended pull-up/pull-down resistor values for this target. Signals not shown should not have pull-ups or pull-downs.

Table C-2 **ARMX (XScale) JTAG Terminations**

Signal Name	Description
/TRST	External 4.7K pull-down
/HRESET	External 2.7K pull-up



NOTE: These are the termination values for the Wind River reference design. It is important that you verify that the pull-up and pull-down values you use are appropriate for the board that you are using.

Index

A

Adding Hardware Data Breakpoints 89
Adding Hardware Instruction Breakpoints 89
Address Bus Test 36
ARM 14-Pin JTAG Connector 93
ARM 20-Pin JTAG Connector 95
ARM Processors -- JTAG 85
ARMX (XScale) 20-Pin JTAG Connector 96
Attempting JTAG communication 23
Attempting to restore CPU context 24

B

Background Mode 22
Bit-Level Detail 29
Board Bring-Up 7
Board Initialization 21
breakpoints
 verifying with target 67
Breakpoints view 83
Bus Tests 36

C

Clock Rate 16

Comparing Target CPU With CF Setting 24
Configuring Registers Manually 28
Converting Files To .bin Format 73
Converting Line or Expression Breakpoints Into
 Hardware Code Breakpoints 90
CPU Reset Type 18
CRC Calculation 35
Creating a Project 48
Creating a Target Connection 9, 47

D

Data Bus Test 36
Debug Connections 9
debugger
 disconnecting and terminating processes 63
Debugging Code in RAM 55
Debugging in RAM 47
Debugging in ROM 79, 80
Diagnostic Functions 32
Disabling Breakpoints 91
Disconnecting and Terminating Processes 63
Downloading a Register File 26
Downloading Code and Symbol Information 52
Drive TRESET Line 17
Driving HRESET to be High 23
Driving HRESET to be Low 23

E

- Emulator HRESET Control [18](#)
- Enabling and Disabling Register Groups [27](#)
- Exporting Breakpoints [91](#)
- Expression Breakpoints [88](#)

F

- Flash Programmer view [70](#)
 - Configuration tab [68](#)
 - getting started [67](#)
 - Memory/Diagnostics tab [76](#)
- Flash programming
 - erasing flash [71](#)
 - setting timeouts [70](#)
 - verifying flash contents [72](#)
- Full RAM Tests [34](#)

G

- Goals and Objectives [7](#)

H

- Hardware Breakpoints [88](#)

I

- Importing Breakpoints [90](#)
- Internal Breakpoint Capabilities [87](#)
- Introduction [1, 15, 21, 31, 37, 65, 85](#)

J

- JTAG Pin Terminations [93](#)
- JTAG Terminations [94, 97](#)

L

- Line Breakpoints [88](#)
- Loading Internal Registers [24](#)

M

- Monitor Target Reset [17](#)
- Monitoring Processes [56](#)

N

- New Connection Wizard [11](#)

O

- OCD Connections [9](#)
- On-Chip Debugging [3](#)
- Overview [47, 79](#)

P

- Pin Terminations [93](#)
- Pins Mapped to Common Signals [85](#)
- processes
 - disconnecting debugger [63](#)
- Programming Flash Memory [65](#)

R

- Read From Location [35](#)
- Reading and Writing Memory [66](#)
- Refreshing Breakpoints [91](#)
- register groups
 - disabling [27](#)
 - enabling [27](#)
- Registers [25](#)

Registers view 28
 Removing Breakpoints 91
 Running a Program 58
 Running Code 40

S

Saving Changes 19
 Scope Tests 35
 Set Verbose On 22
 Setting a Workspace 31
 Setting Breakpoints 83
 Setting Hardware Breakpoints 44
 Setting Software Breakpoints 42
 Setting Up a Project 80
 Simple RAM Test 32
 Stepping an Instruction 38
 Stepping Through a Program 59

T

target
 software breakpoints, verifying 67
 Testing Communications to Hardware Interface 23
 Testing Flash Workspace 66
 Testing for target STOP State 24
 Testing JTAG Communication 24
 Testing Memory 37
 The IN Command 22
 The INN Command 25
 The Registers View 28
 Tool Configuration 15, 16

V

Verifying Hardware 31

W

Waiting for HRESET to be Released 23
 Workbench
 views
 Breakpoints 83
 Write and Complement 36
 Write Rotating Value 36
 Write Then Read 36
 Write To Location 35