

Wind River® Workbench

USER'S GUIDE

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, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. The Wind River logo is a trademark 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	Overview	1
1.1	Introduction	1
2	Setup	5
2.1	Setting up Your Development Environment	5
2.2	Planning a Cross-Development Environment	9
2.3	Setting up a Cross-Development Environment	11
2.3.1	Installing bootApp	12
2.3.2	Configuring the Host	12
2.3.3	Configuring bootApp	14
2.4	Installing a ROM Payload System Image	18
3	Boot	21
3.1	Booting VxWorks 653	21
3.2	Booting VxWorks 653 on the Simulator	22
3.3	Booting a Network Loadable System Image	23

3.4	Booting RAM Payload System Images	24
3.5	Booting ROM Payload System Images	24
3.6	Booting an online-loaded partition	25
4	Connect	27
4.1	Connecting Workbench to the Running Target	27
4.1.1	Connecting Workbench to the Simulator	27
4.1.2	Connecting to the Target via the Network	28
4.1.3	Setting up a Host-Target Connection via a Serial Connection	30
5	Debug	33
5.1	Understanding Cross-development Debugging	33
5.2	Understanding ARINC 653 Debugging	35
5.3	Understanding the Workbench Debugger	35
5.4	Planning Debugging	37
	System mode or task mode?	38
	Do you need to debug C++ code?	39
5.5	Planning Debugging in a Certified Environment	40
	Enable debugging for applications post deployment	40
5.6	Using the Debugger	41
5.7	Controlling Execution	43
5.8	Viewing and Manipulating Data	47
5.8.1	Special debugging situations	48
	How do I overcome the constraints of partitions?	48
	How do I debug application initialization?	49
5.8.2	Examining Memory	51

5.9	Monitoring Resources	51
5.10	Configuring the Debugger	52
6	Tools	53
6.1	Introduction	53
6.2	Boot Program	54
6.2.1	Description of Boot Parameters	56
6.3	wrMonitor	58
6.4	Shells	59
6.4.1	Host Shell	59
6.4.2	Target Shell	59
6.4.3	vThreads Shell	60
	Strengths	61
	Limitations	61
6.5	Monitoring Tools	62
6.5.1	Memory Usage Monitoring	62
6.5.2	Performance Monitoring	62
	Parameters	65
6.5.3	Port Monitoring	66
	System Impact	69
	Using the Port Monitoring Tool	69
6.6	VxWorks 653 Simulator	71
6.6.1	Running the Simulator	71
6.6.2	File Systems	72
6.6.3	Building a Module for the Simulator	72
6.6.4	Differences between the Simulator and VxWorks 653	72
	Architecture Considerations	74

6.7	Configuration and Build Tools	79
6.7.1	XMLGen	80
6.7.2	crDump	81
6.7.3	VxWorks 653 2.2 Development Shell	82
6.7.4	VerIMax	82
6.7.5	Table Viewer	83
	Generating a Complete Set of Reports	83
	Generating Individual Reports	84
6.7.6	VeroStyle	88
A	Debugger Tutorial	89
A.1	Introduction	89
A.2	Debugger Tutorial	89
B	Glossary	105

1

Overview

1.1 Introduction

Wind River Workbench is a suite of cross-development tools for developing embedded systems. A cross-development environment consists of a target (the embedded systems board for which you are developing), a host (the workstation on which you will do your development work), one or more communication channels between the target and the host, and a suite of development tools designed for cross-development.

When an embedded system is running in production mode, its software consists of an operating system and one or more applications running on that operating system. During development an additional piece of software is required on the target to manage communications between the target and the host. This is the target agent. When you debug a program running in the host, it is the target agent that manages breakpoints and watches on the target side and transfers the information to the host. The host in turn uses the target agent to issue debugging commands to the target, such as stepping through code.

On the host side, communication with the target agent is managed by a target server. This communication can take place over either a serial or network connection, or using a hardware emulator such as Wind River Probe or Wind River ICE.

Once the target operating system is loaded, the host communicates with the target for debugging applications. How this communication takes place, and therefore the host setup required to support this communication, depends on the configuration of the target OS and the target agent.

The default target agent uses a network connection to communicate with the host. The host uses two software components to establish communication with a network target, the Registry and the Target Server.

The Registry maintains a list of the targets available on the network. You can communicate with a registry running on your own machine or with a shared registry running on the network.

The target server manages the connection to a target and communicates with the target agent using the WDB protocol.

To set up your host to communicate with the target, you must first launch the registry (which is done automatically when you launch Workbench) and then create and configure a target server to communicate with the target. You set up a target server using the Workbench Target Manager.

All the Workbench tools that need to talk to the target do so using the target server. Once the target server is running and connected, you are ready to begin development.

About this document

This document covers Wind River Workbench 2.6.1 for VxWorks 653 2.2. It includes information on:

- how to setup your development environment
- how to load and run a VxWorks 653 module on a target
- how to debug a VxWorks 653 module using the Workbench debugger
- a number of tools included with Workbench and VxWorks 653.

For information on configuring and building a VxWorks 653 module, see the *VxWorks 653 Configuration and Build Guide*. For information on programming for VxWorks 653, see the *VxWorks 653 Programmer's Guide*.

Conventions Used in This Book

This document uses the following conventions to refer to files and directories that may have different names or locations on your system:

- *installDir* refers to the directory in which your Wind River software is installed.
- *projDir* refers to the directory in which project files are located.
- Configuration files are referred to by their XML document type names. For more information, see the *VxWorks 653 Configuration and Build Guide*.

Specific information on individual Workbench views and dialogues can be found by pressing the help key.

2

Setup

Setting Up Your Development Environment

2.1	Setting up Your Development Environment	5
2.2	Planning a Cross-Development Environment	9
2.3	Setting up a Cross-Development Environment	11
2.4	Installing a ROM Payload System Image	18

2.1 Setting up Your Development Environment

To set up your development environment you must accomplish the following:

- Set up your target (the embedded system board your module will run on)
- Set up your host (the workstation you use to do development)
- Set up the necessary connections between the host and target.

When your target is turned on, it attempts to load its operating system by running its boot ROM code. To set up your target, you install either a VxWorks 653 ROM payload loader or a VxWorks 653 network boot loader as the bootROM of the target. The ROM payload loader loads the VxWorks 653 system image that is resident in target ROM as part of the ROM payload. The network boot loader transfers the VxWorks 653 system image from the host to the target over a network connection.

In a development environment, having the board boot VxWorks 653 from ROM makes it difficult to make changes to the code, especially because, in the case of an VxWorks 653 module, the entire module must be booted at once and applications cannot be downloaded to a running OS.

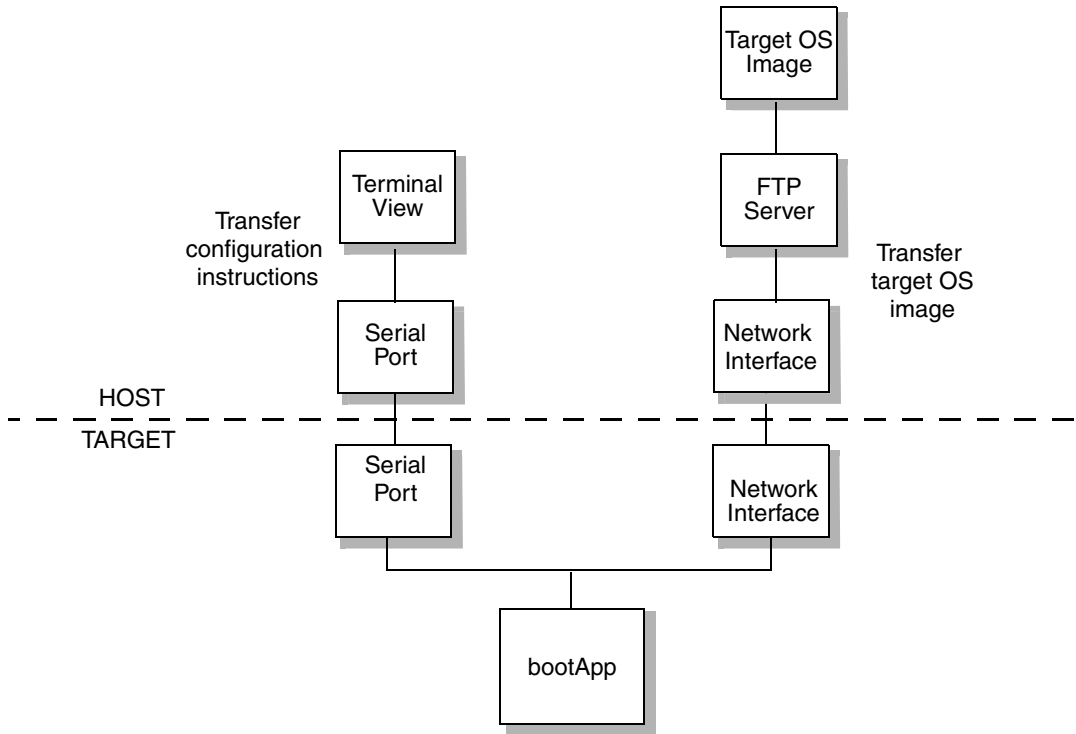
To make it easier to make changes in the module, VxWorks 653 allows you to load the system image from the host over a network or serial connection. To facilitate this, VxWorks 653 supplies a special boot loader program called `bootApp` with every BSP it supports. `bootApp` is provided in both binary and source. You can use `bootApp` as is, modify it to suit your needs, or write your own boot loader.

When loading a network-loadable system image, `bootApp` loads the image over a network connection from the host using FTP or RSH. Before `bootApp` can download the system image, however, it must be configured with the information required to locate the system image on the host and to retrieve it.

To configure `bootApp` with this information, you must create a serial connection to the target. `bootApp` will communicate over the serial connection and prompt you for the information it needs to download the system image over the network. [Figure 2-1](#) illustrates the setup for booting the target using the default boot loader.

Once `bootApp` receives the booting instructions over the serial connection, it is ready to download the target operating system from the host and boot VxWorks 653.

Figure 2-1 Host-Target Communication for Booting



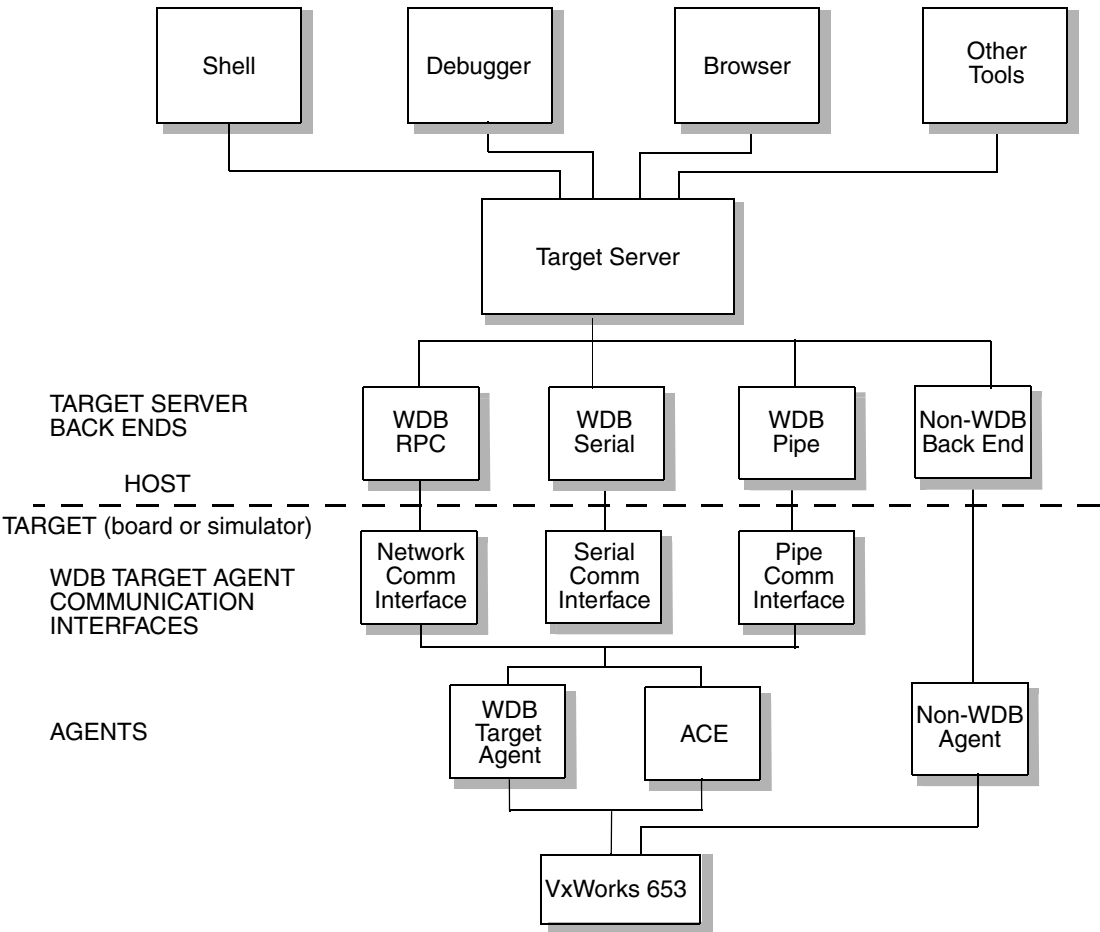
Once the target has been booted the first time, bootApp will store the communication parameters in NV-RAM, meaning that the serial connection will not be necessary for subsequent booting (unless the board is not equipped with NV-RAM, or you need to change the boot parameters).

Notice that it is not necessary for the host that contains the downloadable target operating system to be the same workstation as the host used to establish the serial connection with the target. The serial connection is used to transmit communication parameters and booting instructions to bootApp. You can tell bootApp to load the operating system from any suitably configured machine on the network.

Development Configuration Overview

Figure 2-2 illustrates the setup for developing and debugging with Wind River Workbench. It illustrates both the default configuration using a network connection, and alternate configurations using other types of connections.

Figure 2-2 Host-Target Communication for Development and Debugging



Developing Using the Simulator

It is also possible to develop applications without access to a physical target using the Wind River Simulator. The simulator is essentially a version of the target OS running on your host machine (on top of, not instead of, your workstation's OS).

Specialized Connection Types

Specialized development tasks, such as developing device drivers, operating system components, or boot loaders may require specialized low level connection methods such as JTAG and specialized hardware such as Wind River ICE or Wind River Probe. See the *Wind River On Chip Debugging User's Guide* and the Wind River ICE and Wind River Probe documentation for more information on setting up your development environment for use with these tools.

2.2 Planning a Cross-Development Environment

Before you proceed with setting up your development environment, you may find it useful to spend some time planning your development strategy. Planning your development strategy involves answering a number of questions about your environment and the kinds of development work you will be doing. Answering the following questions will help lead you to the correct development strategy for your project.

Which type of system image will you use

VxWorks 653 can produce three different types of system image:

- Network loadable system image
- RAM payload image
- ROM payload image

For information on the image types, their advantages and disadvantages, see the *VxWorks 653 Configuration and Build Guide*.

How should the target acquire the system image?

You need to decide how the system image will get be loaded on to the target and booted. Available options are as follows:

Network from the Host

The principal advantage of a network connection is speed. This method is supported by bootApp. You may develop a custom network boot ROM if your board is unable to store configuration parameters between sessions or if you have some particular network configuration or booting problem you need to resolve. This method can be used with a network loadable system image or a RAM payload system image.

From Flash on the Target

Loading the OS from flash on the target is quick, but it can be very cumbersome to transfer an updated copy of the target OS to the target during development. This method is used with ROM payload system images and is more commonly used for production systems rather than for development purposes.

Via an Emulator

An emulator can give you very low level access to the target hardware when other communication channels are unavailable. It is principally useful when you are developing low level drivers and OS components. For information on using Wind River ICE or Wind River Probe to load the target OS onto your target, see *Wind River ICE for Wind River Workbench Hardware Reference* or *Wind River Probe for Wind River Workbench Hardware Reference*.

Will you use the Wind River target agent?

While Wind River supplies a target agent as part of the Workbench suite, some organizations prefer to develop their own agent, or to use another generally available agent. If you decide to use a different agent, you must also select or develop an appropriate back end to communicate with your custom target agent.

What debug modes do I need to support?

The default target agent supports two debug modes: task mode and system mode. Task mode is used for debugging individual applications. System mode is used for debugging the target operating system itself and low level code such as device drivers. For more information on debug modes, see [5. Debug](#).

The reason that you need to consider this question when designing your development environment is that system mode debugging requires a communication channel that is capable of working in polled mode rather than the default interrupt mode. (Interrupts are suspended when code is stopped at a breakpoint in system mode.) Most Wind River BSPs supply a default driver capable of running in polled mode, and the target agent switches automatically

between interrupt mode and polled mode as required. However, not all the available drivers support polled mode. This should not be an issue for you as long as you stick to the default communication drivers for the target agent.

How will the host and the target communicate for debugging?

The issue of how the host and target communicate for development and debugging is essentially unrelated to the issue of how they communicate for booting the target. The two processes may use the same physical connections, or different ones. You can use different host machines and different communication channels on the target for development than you used for booting.

For development and debugging purposes, the choices of communication channels are as follows:

Network

The normal method for communication between host and target during debugging is over a network.

Serial

You might choose a serial connection if a network connection were not available, if you needed to reduce the size of the target agent by removing networking support, or if you did not have access to a network driver that supported polled mode communication, which is needed to support system mode debugging.

Emulator

You might choose to use an emulator if you are doing programming that requires low level access to the target hardware or if you are debugging a problem related to higher level communication protocols, or which occurs before higher level communication protocols are available in the boot sequence.

2.3 Setting up a Cross-Development Environment

The following is an overview of the steps required to prepare a target and host for development. Some of these steps may already have been performed for you in your development organization, or your organization may be using a custom

setup. Consult the person in charge of your development environment to determine the specific steps to be followed in your environment.

2.3.1 Installing bootApp

The first step to setting up your development environment is to install bootApp on your board.

VxWorks 653 includes both the source code and binary for bootApp for each supported BSP. Unless you want to make changes to bootApp, you can use the precompiled binaries for your BSP. The bootApp for each BSP is found in `$(WIND_BASE)/target/proj/$(BSP)_bootApp`.

Instructions for installing bootApp on a particular board are found in the BSP documentation for your board.



CAUTION: Follow proper procedures to protect your equipment from electrostatic discharge (ESD). Failure to take proper ESD precautions may seriously damage your board. It can also degrade target hardware over time, leading to intermittent errors or hardware failure. Damage caused by ESD can be difficult to diagnose, resulting in significant loss of time trying to track down the causes of subtle failures.

BSPs Requiring TFTP on the host

Some boards require the TFTP protocol on the host in order to burn a new VxWorks 653 image into flash. Workbench ships with a version of TFTP. See your BSP documentation on how to burn flash for your board.

2.3.2 Configuring the Host

You must set up the host to communicate with the target. There are two parts to this communication, and they may occur on different host machines or the same one.

The first part of the host/target communication occurs over a serial connection and is used to configure bootApp to download the VxWorks system image over a network connection.

The second part is to configure the host to respond to a request from bootApp to download the VxWorks 653 system image.

Configuring Terminal View

You can use any terminal emulator to configure bootApp. Workbench supplies a suitable terminal emulator in the **Terminal** view.

To configure **Terminal** view to receive a connection from bootApp:

1. Launch Workbench on the host.
2. Locate the **Terminal** view.
3. Click the Settings button on the **Terminal** view toolbar.
4. Configure your host system serial port for a full-duplex (no local echo), 8-bit connection with one stop bit and no parity bit. The line speed must match whatever is configured into your target agent. See your BSP documentation for specific settings.
5. Click **OK**. Workbench opens the port and awaits a connection from the target.

Configuring FTP on the host

bootApp downloads the target OS image from the host using FTP or RSH. Your host must be configured to respond to the request from the target to download the system image.

For Windows, Workbench includes an FTP server application, **WFTPD**.



NOTE: You can run **WFTPD** as a restricted user, but you cannot add new users and passwords if you are a restricted user. A non-restricted user must add the new users and passwords for you.

To configure **WFTPD**:

1. Select **Programs > Wind River > VxWorks 653 2.2 > FTP Server**. The WFTPD application appears.
2. Select **Security > Users/rights**. The **User / Rights Security Dialog** is displayed.
3. Select **New User**. The **New User** dialog is displayed.
4. Enter the target user name.
5. Click **OK**. The **Change Password** dialog is displayed.
6. Enter the target password and reenter it to verify what you have typed.
7. Click **OK**. You will be returned to the **User / Rights Security Dialog**.

8. Fill in the **Home Directory** text box. The home directory specifies the directory that will be displayed to an FTP user when they first log in.
9. If the **Rights for user** panel is not displayed, click **Rights >>**. Assign this user sufficient privileges to permit the download of the system image.
10. Click **Done**.
11. To enable logging of FTP activities, select **Logging > Log Options** and select the types of activities you want to log.

When you have finished configuring your FTP settings, leave the FTP server running. It must be running on your host when your target tries to access the system image.

You are now ready to boot a VxWorks 653 system image. For information on booting a VxWorks 653 image, see [3. Boot](#).

2.3.3 Configuring bootApp

Once bootApp is installed and your host is configured, you can configure bootApp to download the VxWorks system image from the host. Once the image has been booted for the first time, the configuration will be saved and you will not need to use this procedure again unless:

- You want to change the configuration
- The board has been used to boot a different system, overwriting the configuration data stored in its NVRAM.
- You want to load a different type of system image.

For this example, we will use the values for the boot parameters shown in [Table 2-1](#):

Table 2-1 **Sample Boot Parameters**

Parameter	Value
Host IP address	90.0.0.1
Host name	mars
Target IP address	90.0.0.50
Target name	phobos

Table 2-1 **Sample Boot Parameters** (cont'd)

Parameter	Value
Image location	c:\project\integration\boot.txt
Target user name	fred
Target password	secret

Step 1: Determine communication parameters

Determine the correct values for the parameters that govern target-host communication during the booting process.

Host IP Address

The IP address of the host computer. This is the address of the computer on which the target OS image resides. You will need this to configure the boot loader.

Target IP Address

The IP address of the target. Ask your network administrator to assign an IP address for use with your target. You will need this to configure the boot loader.

If you intend to use ACE over a network connection, your ACE will be configured with a specific network address. You may need to configure a private network around your target in order to assign it an address that matches the address configured into ACE.



CAUTION: If you are in a networked environment, do not pick arbitrary IP addresses for your host and target as they could be assigned to someone else. Contact your network administrator for available IP addresses.

Target Name

The network name assigned to the target. This is optional. bootApp always uses the IP address of the target.

Image Location

The location of the target OS image on the host. Note that this path must be expressed relative to the root of the FTP server running on the host. You will need this to configure the boot loader. For a network-loadable system image, this

parameter should point to the **boot.txt** file produced by the build. For a RAM payload system image, it should point to **sms_ramPayload**.

Target User Name

The user name that the boot loader will use to download the target OS image from the host. This can be any name you like. You will need this to set up the FTP or RSH server on the host and to configure the boot loader.

Target Password

The password that the boot loader will use to connect to the FTP server to download the target OS image. You will need this to set up the FTP server on the host and to configure the boot loader. A target password is not required if you use RSH to load your image.

Step 2: Power up the target

Switch on the target power according to the directions in the hardware documentation.

Step 3: Configure the boot loader

If your system is properly configured, the target will start the boot loader and the boot loader will communicate with the host over the serial port. The boot display will look something like this:

```
VxWorks 653 System Boot
```

```
Copyright (c) 1984-2007 Wind River Systems, Inc.
```

```
CPU: MPC8560 - Wind River SBC  
Version: 2.2  
BSP version: 1.3/1  
Creation date: Aug 24 2007, 03:37:56
```

```
Press any key to stop auto-boot...  
1
```

The boot loader starts a seven-second countdown (after which it will proceed with the boot using the default configuration).

Stop the countdown by pressing any key. The boot program displays the VxWorks 653 boot prompt:

```
[VxWorks 653 Boot]:
```

Step 4: Review the existing boot parameters

To display the current (default) boot parameters, type **p** at the boot prompt:

```
[VxWorks 653 Boot]: p
```

A display similar to the following appears. The meaning of each of these parameters is described in [6.2 Boot Program](#), p.54. The **p** command does not actually display the lines with blank fields, although this example shows them for completeness.

```
boot device           : ln
unit number          : 0
processor number      : 0
host name             : mars
file name             : c:\project\integration\boot.txt
inet on ethernet (e)  : 90.0.0.50:ffffff00
inet on backplane (b) :
host inet (h)         : 90.0.0.1
gateway inet (g)      :
user (u)              : fred
ftp password (pw) (blank=use rsh) :secret
flags (f)             : 0x0
target name (tn)      : phobos
startup script (s)    :
other (o)             :
```

Step 5: Enter new boot parameters

You can now configure the boot loader to download and execute the target OS. For a complete list of boot loader parameter and commands, see [6.2 Boot Program](#), p.54.

1. Type **c** at the boot prompt:

```
[VxWorks 653 Boot]: c
```

The boot loader prompts you for each parameter in turn.

2. If a particular field has the correct value already, press **ENTER**. To clear a field, enter a period (.), then press **ENTER**. To go back to change the previous parameter, enter a dash (-), then press **ENTER**. If you want to quit before completing all parameters, type **CTRL+D**.

If your target has non-volatile RAM (NV-RAM), the boot parameters are stored there and retained even if power is turned off. For each subsequent power-on or system reset, the boot program uses these stored parameters for the automatic boot configuration.

Step 6: Initiate booting

You can test your connection by initiating booting by typing the @ command:

```
[VxWorks 653 Boot]: @
```

The boot loader will download the target OS image and boot it on the board. Note that this will only be successful if there is a correctly built VxWorks 653 network-loadable or RAM payload system image located at the location you specified in your configuration. For information on building VxWorks 653 system images, see the *VxWorks 653 Configuration and Build Guide*.

2.4 Installing a ROM Payload System Image

To install a VxWorks 653 ROM Payload system image, you must flash each component of the image to the correct location in ROM. Those locations are the ones that you configured in the **Payloads** section of your Module configuration document. If a set of components is configured to be located contiguously in ROM, you must concatenate the **.bin** files for each contiguously located component into a single bin file that can be flashed at the appropriate location in ROM.

This example is based on a system that includes the following components:

- *my-coreOS*
- *my-ssl*
- *my-sl-A*
- *my-sl-B*
- *my-sdr*
- *my-application-A-partition*
- *my-application-B-partition*
- *my-application-C-partition*

The *my-application-B-partition* component will be configured as an online-loaded partition.

The payload memory is configured as follows:

```
<Payloads>  
  <CoreOSPayload
```



```

    Base_Address="0xc0000000"
    Size="0x100000"/>
<SharedLibraryPayload
  NameRef="my-ssl"/>
<SharedLibraryPayload
  NameRef="my-sl-A"
  Base_Address="0xc0200000"
  Size="0x100000"/>
<SharedLibraryPayload
  NameRef="my-sl-B"
  Size="0x100000"/>
<SharedDataPayload
  NameRef="my-sdr"/>
<ConfigRecordPayload
  NameRef="configRecord"
  Base_Address="0xc0400000"
  Size="0xC00000"/>
<PartitionPayload
  NameRef="my-application-A-partition"/>
<PartitionPayload
  NameRef="my-application-B-partition"
  Online="true"/>
<PartitionPayload
  NameRef="my-application-C-partition"/>
</Payloads>

```

Unless you specify a specific base address for a payload, the system will locate the payloads contiguously. Unless you specify a specific size for a payload, the build system will calculate the size base on the black box for the component.

Given this configuration, the build process will create the following binary files:

- *my-coreOS.bin*
- *my-ssl.bin*
- *my-sl-A.bin*
- *my-sl-B.bin*
- *my-sdr.bin*
- **configRecord.bin**
- *my-application-A-partition.bin*
- *my-application-B-partition.bin*
- *my-application-C-partition.bin*

You would then concatenate these files as follows. The names of the concatenated files are up to you. You will use them with your flash utility to flash files to ROM:

1. Concatenate *my-coreOS.bin* and *my-ssl.bin* to *my-platform.bin*. On UNIX you can use the **cat** command to concatenate files. On Windows, you can use the **copy** command, as follows:

```
% copy /b my-coreOs.bin+my-ssl.bin my-platform.bin
```

2. Concatenate *my-sl-A.bin*, *my-sdr.bin*, and *my-sl-B.bin* to *my-shared.bin*.
3. Concatenate **configRecord.bin**, *my-application-A-partition.bin*, and *my-application-C-partition.bin* to *my-apps.bin*.

Although *my-application-A-partition.bin* and *my-application-C-partition.bin* are not consecutive in the XML document, *my-application-C-partition.bin* follows *my-application-A-partition.bin* because *my-application-B-partition.bin* is defined as an online-loaded partition.

4. Convert **sms_romPayload.hex** to **sms_romPayload.bin** following the directions in your BSP documentation.
5. Flash the ROM image to the correct addresses in ROM using the flash utility of your choice. The addresses used must be those specified in the **payloadMemory** element of the CoreOSDescription document, as described in [2.12.2 Configuring a ROM Payload](#), p.52.

Following the example given above, you should flash the files as follows:

- Flash **sms_romPayload.bin** to the location specified by the **ROM_TEXT_ADRS** of the BSP.
- Flash *my-platform.bin* starting at 0xc0000000.
- Flash *my-shared.bin* starting at 0xc0200000.
- Flash *my-apps.bin* starting at 0xc0400000.

Because *my-application-B-partition.bin* is defined as an online-loaded partition it is not flashed.

3

Boot

Loading and Running VxWorks 653 on your Target

- 3.1 Booting VxWorks 653 21
- 3.2 Booting VxWorks 653 on the Simulator 22
- 3.3 Booting a Network Loadable System Image 23
- 3.4 Booting RAM Payload System Images 24
- 3.5 Booting ROM Payload System Images 24
- 3.6 Booting an online-loaded partition 25

3.1 Booting VxWorks 653

A VxWorks 653 system image always contains a complete module including all libraries, applications, and shared data. With the limited exception of online-loaded partitions, you cannot add or subtract anything from a running VxWorks 653 system. Running VxWorks 653, therefore, consists of transferring a complete system image to the target and booting it. The core OS will then automatically create all the partitions, load the applications, and run them according to the default schedule.

How you boot a VxWorks 653 system image will depend on which type of image you are using and whether you are running on a real target or on the simulator.

Since the only image type that works on the simulator is the network-loadable system image, there are therefore four possible boot routines:

- Booting a network-loadable system image on the simulator
- Booting a network-loadable system image on a target
- Booting a RAM payload system image on a target
- Booting a ROM payload system image on a target

If you are using a network-loadable image, bootApp will load the image from a host over the network and then boot the image.

If you are using a RAM payload image, bootApp loads the image into a RAM payload region of the target RAM and then transfer control to the payload loader that is part of the RAM payload image, which then loads and boots the system image.

If you are using a ROM payload image, you flash the various components of the ROM payload image to target ROM. You will also flash the payload loader that is part of the ROM payload to the boot ROM address of the target (replacing bootApp, if it is installed). You then power on the target and VxWorks 653 will boot.

For more information on system images, schedules, and, online-loaded partitions see the *VxWorks 653 Configuration and Build Guide*.

3.2 Booting VxWorks 653 on the Simulator

You can run your system image on the simulator either through Workbench or by invoking VxSim on the command line.

Booting the Simulator via Workbench

To run your module on the simulator using Workbench:

1. Build a network loadable system image as described in the *VxWorks 653 Configuration and Build Guide*.
2. Open Workbench.
3. Select **Targets > New Connection**. The **New Connection Wizard** appears.

4. Select the **Simulator Connection**. The wizard prompts you for a boot file name.
5. Select **Custom Simulator** and enter the path to the **boot.txt** file in the integration directory of your build tree. (The name and location of this directory will depend on how you designed your build system.)
6. Complete the wizard, supplying whatever parameters are appropriate to your application. Workbench will launch your module and connect to the simulator.

For more information on the managing targets, select the **Target Manager** view and press the help key.

Booting the Simulator in the Command Line

You can run the simulator from the command line:

1. Open the VxWorks 653 2.2 Development Shell.
2. Change the current directory to the project integration directory that contains **boot.txt**.
3. Type **vxsim**.

For more vxsim option, type **vxsim -help**.

3.3 Booting a Network Loadable System Image

1. Build a network loadable system image as described in the *VxWorks 653 Configuration and Build Guide*.
2. Configure your host and target to boot a network loadable system image as described in [2. Setup](#).
3. Initiate booting by typing the @ command in the Terminal view that is connected to bootApp on the target:

```
[VxWorks 653 Boot]: @
```

The boot loader will download the VxWorks 653 system image and boot it on the board.

Rebooting VxWorks 653

When VxWorks 653 is running, there are several ways you can reboot it. Rebooting by any of these means restarts the attached target server on the host as well



CAUTION: Be sure to follow ESD precautions specified in the hardware documentation whenever you work with your target.

- Entering **CTRL+X** in the **Terminal View** (other Windows terminal emulators do not pass **CTRL+X** to the target, because of its standard Windows meaning.)
- Invoking **reboot()** from the host shell.
- Pressing the reset button on the target system.
- Turning the target's power off and on.

3.4 Booting RAM Payload System Images

The procedure for loading a RAM Payload System Image is identical to that for loading a network-loadable system image, with one exception: In loading the network-loadable system image, you configure the bootApp to load **boot.txt**. To boot a RAM payload image, you configure bootApp to load **sms_ramPayload**. (Note however, that you will still use **boot.txt**, not **sms_ramPayload**, when configuring the target server.)

3.5 Booting ROM Payload System Images

To boot a ROM payload system image, install the ROM payload system image on the target as described in [2. Setup](#) and power up the board.

3.6 Booting an online-loaded partition

The method for loading an online partition in a production system is entirely up to the platform provider and the system integrator. For information on how to build a loader for an online-loaded partition, see the *VxWorks 653 Programmer's Guide*.

To test an online loaded partition during development, you may use the following procedure.

1. Boot the system image.
2. Establish a target server connection to the target.
3. In the target shell, verify that there is a pool with the name of the online-loaded partition:

```
[coreOS] -> rgmShow
```

4. View information on partitions, noting that the partition mode for the online-loaded partition is **IDLE**:

```
[coreOS] -> partitionShow
```

5. Load the module containing the online-loaded partition loader:

```
[coreOS] -> ld < c:\onlinePartitionLoad.o
```

6. Load the code for the online-loaded partition loader (and spawn the task):

```
[coreOS] -> sp onlinePartitionLoad, "c:\part2.bin", 2
```

7. Start the partition:

```
[coreOS] -> partitionModeSet 2, 1, 0, 0, 0
```

8. Verify that the partition mode for the online-loaded partition is **COLD_START** instead of **IDLE**:

```
[coreOS] -> partitionShow
```


4

Connect

Connecting the host to the running target

4.1 Connecting Workbench to the Running Target

In order to debug your application using Workbench, you must connect the Workbench tools to the VxWorks 653 system image running on the target.

If you need a JTAG or other emulator connection, see the *Wind River ICE for Wind River Workbench Hardware Reference* or the *Wind River Probe for Wind River Workbench Hardware Reference* for information about making emulator connections to your target.

4.1.1 Connecting Workbench to the Simulator

If you booted your system image on the simulator as described in [3. Boot](#), it will already be connected to Workbench. If you booted your system image on the simulator by starting VxSim on the command line, you can connect Workbench to the Simulator using the same procedure as described in [3. Boot](#). Workbench will connect to the existing simulator rather than launching a new instance of the simulator.

4.1.2 Connecting to the Target via the Network

To create a connection to the target for development using a network connection:

Step 1: Ensure that the registry is running

Make sure that the registry is running on your host, or that you have access to an appropriate network registry. The registry is started automatically on your host when you run Workbench. If you are using a network registry, talk to the person responsible for your development environment.

Step 2: Configure the target server

Configure the target server settings necessary to connect to the target. This procedure will cover the steps necessary to create a basic connection with a default configuration. For more information on target server options press the help key in the **Target Manager** view or the **New Connection** dialog.

1. Open Workbench.
2. Select **Target > New Connection**. The **New Connection** wizard appears. The **New Connection** wizard supports many different types of connections. The connection types available on your system will depend on the set of product that you have installed.
3. Select **Wind River VxWorks 653 Target Server Connection** and click **Next**. The **Target Server Options** page appears.

New Connection

Target Server Options

Please enter a valid target name or IP address.

Backend settings

Backend: wdbrpc Cpu: (default from target)

Target name / IP address: Check... Port:

Kernel image

☒ File path from target (if available)

☐ File: Browse...

☐ Bypass checksum comparison

Advanced target server options

☒ Verbose target server output

Options: -R C:/WindRiver/workspace -RW -Bt 3 -A Edit...

Command Line:

```
tgtsvr -V -R C:/WindRiver/workspace -RW -Bt 3 -A
```

< Back Next > Finish Cancel

4. For **Backend**, select the **wdbrpc** back end.
5. For **Target name / IP Address**, enter the target IP address or the target name.
6. In the **Kernel Image** panel, select **File** and enter the location of the **boot.txt** file in the integration directory of your build tree. Note that when you connect the target manager to a RAM payload system image, you do not use **sms_ramPayload** here, you use **boot.txt** just as with the network-loadable image.
7. In the **Advanced target server options** section, select the **Verbose target server output**.

The command line section show the command line that will be used to create the target server.

8. Click **Next** through the following pages, then click **Finish**. Your new target server connection definition will appear in the **Target Manager** connection list.

Step 3: Connect to the target

The target manager will attempt to connect to your target. If everything is set up properly, you will see **connected - target server running** in the status bar.

If the target manager does not connect automatically, or to connect to a target using a previously defined connection:

1. Right click on the connection in the **Target Manager** view.
2. Select **Connect**.

4.1.3 Setting up a Host-Target Connection via a Serial Connection

The procedure for using a serial connection for host-target communication during development is:

Step 1: Ensure that you have a working serial connection between the host and the target

Make sure that you have a working serial connection between the host and the target. If you were able to boot the target as described in [3Boot](#), p.21, or if you were able to successfully boot the target using a custom serial-only boot procedure, then your connection is working correctly.

Step 2: Configure the target agent for serial connection

Using a serial connection is not the default configuration, therefore, you will have to modify the target agent configuration in order to use a serial connection.



NOTE: If the target has only one serial port and the target server is therefore using the same port as the terminal program used for booting, you must terminate the terminal connection before trying to connect with the target server. If the terminal connection is open, the target server will not be able to connect.

Step 3: Configure the target server

To configure the target server for a serial connection.

1. Open Workbench.

2. Select **Target > New Connection**. The **New Connection** wizard appears. For more information on the **New Connection** wizard, press the help key.

The **New Connection** wizard supports many different types of connections. The connection types available on your system will depend on the set of product that you have installed.

3. Select **Wind River VxWorks 653 Target Server Connection** and click **Next**. The **Target Server Connection** dialog appears.
4. For **Back End**, select the **wdbserial** back end.
5. For **Host serial device**, select the serial port on the host that is connected to the target.
6. For **Serial device speed**, select the highest speed supported by your target.
7. In the **Advanced Target Server Options** section, select **Verbose target server output**.

The command line section show the command line that will be used to create the target server. Your command line should look like this:

```
tgtsvr -V -d comport -bps speed -B wdbserial ipaddress
```

8. Click **Next** through the following screens, then click **Finish**. Your new target server connection definition will appear in the Target Manager connection list.

Step 4: Connect to the target

The target manager will attempt to connect to your target. If everything is set up properly, you will see **connected - target server running** in the status bar.

If the target manager does not connect automatically, or to connect to a target using a previously defined connection:

1. Right click on the connection in the **Target Manager** view.
2. Select **Connect**.

5

Debug

- 5.1 Understanding Cross-development Debugging 33
- 5.2 Understanding ARINC 653 Debugging 35
- 5.3 Understanding the Workbench Debugger 35
- 5.4 Planning Debugging 37
- 5.5 Planning Debugging in a Certified Environment 40
- 5.6 Using the Debugger 41
- 5.7 Controlling Execution 43
- 5.8 Viewing and Manipulating Data 47
- 5.9 Monitoring Resources 51
- 5.10 Configuring the Debugger 52

5.1 Understanding Cross-development Debugging

Debugging in a cross-development environment involves debugging an application running on a target using debugging tools running on a host. This requires an agent on the target to communicate with the debugging tools on the host and to perform debugging actions on the target, such as setting breakpoints, starting and stopping execution, and inspecting the values of variables.

The following components are used to establish an environment for cross-development debugging:

- The target: the board that the software to be debugged is running on.
- The target image: the system image that is running on the target.
- The host image: an identical system image to the target image, but resident on the host.
- The source code: the code from which the system image was created, also resident on the host.
- Symbol files: files listing the symbols in the image, resident on the host.
- The debug agent: a component running on the target that communicates with the debugger to execute debug actions on the target and to communicate debug information to the debugger.
- The debugger: a debugger running on the host that communicates with the debug agent to debug the software running on the target.
- The target server: a component running on the host that manages communication between the debugger and the debug agent.

The debugger requires access to the host image and the source code in order to formulate and communicate instructions to the target agent and interpret information sent by the target agent. The general procedure for debugging in a cross development environment is:

1. Build a system image with debug information and include the debug agent in the image.
2. Transfer the image to the target and boot the target.
3. Establish a connection from the host to the target and attach the debugger to the image running on the target via the debug agent.
4. Debug the system.

The debugging options available depend on several factors, including the capabilities of the debugger, the capabilities of the debug agent, the debug support available in the target hardware, and the characteristics of the target software. In the case of an ARINC 653 compliant system, the strict protections in place between partitions and the core OS, and the constraints that exist in a certified environment, present some unique debugging challenges.

5.2 Understanding ARINC 653 Debugging

Debugging ARINC 653 compliant systems presents a number of particular challenges, in addition to the normal challenges of debugging in a cross-development environment. VxWorks 653 provides a number of tools and strategies to help you cope with these challenges. The principal challenges of an ARINC 653 compliant system are:

Challenges of a Partitioned System

Applications reside in partitions which are strictly separated from the core OS. This restricts the access that the target agent has to code running in a partition. Applications and partitions cannot be downloaded to a running system. All applications and partitions (with the exception of online-loaded partitions) must be part of the payload at the time that the system is booted. At boot time, the core OS creates all partitions and starts them according to a schedule.

Challenges of a Certified Environment

VxWorks 653 applications are usually intended to be certified. The debug agent typically runs as a component of the core OS on the target. However, a debug agent cannot be included in a certified core OS. This means that during development, the system is running a non-certified core OS. To alleviate this problem, VxWorks 653 includes the Agent for Certified Environment (ACE), which allows the target agent to run outside of the core OS. ACE provides a restricted set of debugging facilities. In most cases, it is used for final testing and debugging of a system in cert configuration, after principal development is complete.

For information on including ACE in your system, see the *VxWorks 653 Configuration and Build Guide*.

5.3 Understanding the Workbench Debugger

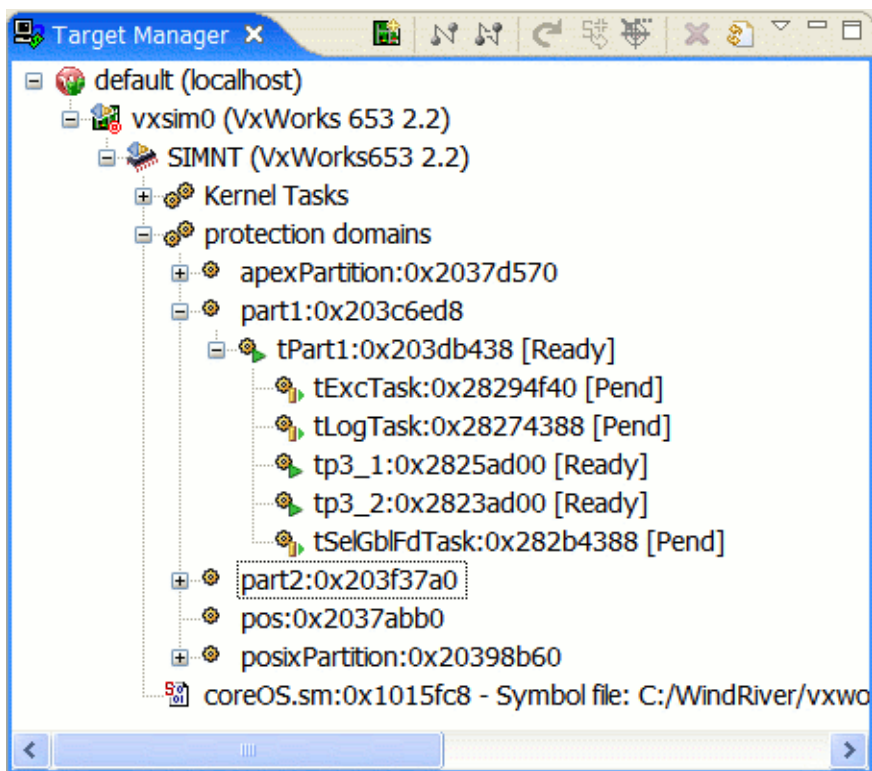
For general information on the Workbench debugger, press the help key in the debugger perspective.

For a tutorial on the debugger with VxWorks 653, see [A. Debugger Tutorial](#).

Understanding the Target Manager Display

Managing a system running on a target and attaching the debugger to tasks running in a system is done through the Target Manager. It is important to understand the Target Manager display for a VxWorks 653 system. A sample display is shown in [Figure 5-1](#).

Figure 5-1 Sample Target Manager Display



This is what each level of the hierarchy in [Figure 5-1](#) represents:

default(localhost)

This is the registry that you used to connect to your target.

vxsim0 (VxWorks653 2.2)

This is the target that your module is running on. In this case, the target is the simulator.

SIMNT (VxWorks653 2.2)

This is your module.

Kernel Tasks

This item collects all the kernel tasks. Expanding this item will show all the tasks running in the kernel (not shown here to reduce space).

protection domains

This item collects all the partitions, shared libraries, and other protection domains in your system.

apexPartition:0x2037d570

This item represents one partition, called **apexPartition**. The hex number following the name is the location of the partition in memory.

part1:0x203c6ed8

This item represents a second partition, called **part1**. It is expanded to show its contents.

tPart1:0x203db438 [Ready]

This item represents the partition task. The partition task is a kernel task that runs the partition. The current state of this task is Ready. Below the partition task are listed all the threads running in the partition. Note that the threads will only be displayed if the partition is part of the current schedule. If the partition is not part of the current schedule, its state will be shown as Suspend and the threads will not be shown.

When attaching the debugger to the module, you will attach either to a kernel task or to a partition task.

tExcTask:0x28294f40 [Pend]

This item represents a thread running in a partition.

5.4 Planning Debugging

When planning your debugging strategy, these are some of the issues you will need to bear in mind.

System mode or task mode?

Two debugging modes are available, task mode and system mode. In system mode, hitting a break point stops the entire system. In task mode, hitting a breakpoint stops a single running task or partition. Other partitions and the core OS continue to run normally according to the schedule. Note that the stopped task is still subject to the scheduler. HM events could occur if it does not meet its deadlines. Periodic processes could be thrown off.

In both modes, if a breakpoint is encountered in a thread in a partition, the whole partition is stopped. Individual threads in the partition are visible, but you cannot control them individually.

Advantages of System Mode

In task mode, stopping one partition does not stop other partitions which may depend on the stopped partition. If you stop partition A, and partition B is waiting for information from partition A, partition B will run on its normal schedule and may inject health monitor events as a result of not receiving information from partition A. These in turn could affect the behavior of partition A, or even result in a partition restart or module restart. In system mode, stopping partition A stops the whole system, avoiding problems of this sort.

This preserves the scheduling behavior of multi-task applications. In task mode, when a task hits a debugger breakpoint, other tasks in the application or operating system may be scheduled to run when otherwise they would not. For many applications, this represents a significant change in the behavior and characteristics of the application. By putting all the tasks into the break state together, system mode preserves the scheduling characteristics of the application.

This also supports third-party emulators for operating system bring-up. If an emulator is present, a pseudo-protection domain and pseudo-task are shown in the control view prior to the operating system starting.

System mode allows you to debug interrupt service routines (ISR) by attaching the debugger to the kernel and setting a breakpoint in the ISR.

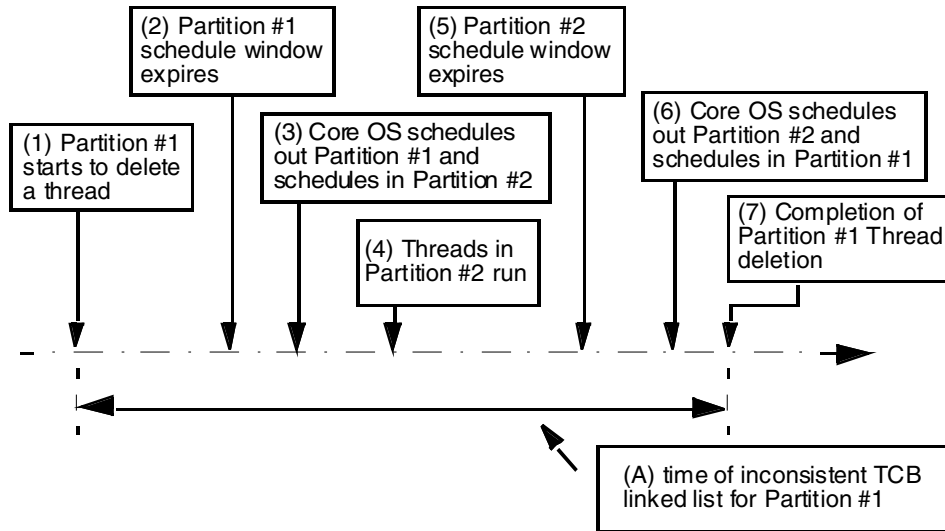
Limitations of System Mode

When the debugger is in system mode:

- When the host tools access data, it is possible that the data may be in an inconsistent state. For example, [Figure 5-2](#) shows a system of two partitions. Partition #1 starts to delete a thread, but its time slice expires before it completes the action. Partition #2 runs for its time frame. If a system-mode

breakpoint fires while Partition #2 is running, Partition #1's thread linked list will be in an inconsistent state. While the frequency of this type of event is low, when it occurs the behavior of the tools may be unpredictable.

Figure 5-2 **Timeline of Hypothetical Debug Scenarios**



Unusable Tasks

In bimodal systems, selecting system mode when a task is in the **break** state causes that task to remain in the break state until task mode is selected again. Such tasks are marked as unusable. You cannot change their state from the debugger or the target agent until you switch back to task mode.

Do you need to debug C++ code?

If your project contains C++ code, refer also to [Inline Functions and Default Constructors and Destructors](#), p.47. This section discusses compiling options for inline functions and coding requirements for constructors and destructors that you want to debug.

5.5 Planning Debugging in a Certified Environment

VxWorks 653 provides a number of tools and techniques to assist in debugging in a certified environment.

Using ACE

Because the ACE debug agent is separate from the core OS, you can include ACE in a system with a cert core OS. While the use of ACE restricts the number of shell routines that are available, it does allow you to inspect many system properties and perform a number of debugging activities. For information on including ACE in your module, see the *VxWorks 653 Configuration and Build Guide*.

Debugging Applications Built with Certifiable Subsets

In order to debug an application that is built with certifiable subsets, you can debug it against the debug core OS image. In doing so, you benefit from all the Workbench tools, including the full WDB debug agent without the limitations of ACE. To do this, simply include a debug core OS rather than a cert core OS in your integration build.

Taking Advantage of the CERT Macro

Because the CERT macro is defined when compiling with the **cert** build specification, it is possible to have the debug code conditionally compiled in the application, as shown in the following example:

```
numItems = getNewItems(items, MAX_ITEMS); /* get new items */
#ifdef CERT /* full VxWorks AE653 */
printf ('debug: numItems returned = %d\n', numItems);
#endif
```

Enable debugging for applications post deployment

Debug code cannot be included in a deployed certified system. However, it is sometimes useful to be able to attach a system to a debugger after initial deployment so as to diagnose a problem encountered in the field.

To facilitate this, VxWorks 653 allows you to optionally start ACE based on a hardware signal. This means that ACE can be included in a deployed system, but not started. When the board is placed on a test rack, a hardware signal can be asserted, causing ACE to start, and enabling debug access to the system.

This functionality is provided by a function pointer `_func_usrAceStartupConditionGet`, which is set to NULL by default. If the function pointer is set to NULL, ACE starts. The function pointer can be set to point to a function that returns TRUE or FALSE. If the functions returns FALSE, ACE does not start.

The system integrator or platform provider can set this function pointer to point to a function that will return TRUE or FALSE depending on a hardware signal. In a deployed system, the function should return FALSE, so that ACE does not start. In a test bed, the signal should be asserted and the function should return TRUE, causing ACE to start.

5.6 Using the Debugger

This is the general procedure for using the debugger. For a tutorial on the debugger, see [A. Debugger Tutorial](#).

To run your system with the debugger:

Step 1: Build your module in debug mode or included ACE

You must either build your system image in debug mode or include ACE in your system image. For information on building a system image in debug mode or including ACE in your system image, see the *VxWorks 653 Configuration and Build Guide*.

Step 2: Run your system on the target

Transfer your system image to the target and boot the target as described in [3. Boot](#). You are now ready to attach the debugger to a running task.

Step 3: Attach the debugger to a task

You can attach the debugger to a kernel (core OS) task or to a task running in a partition. You can also spawn a new kernel task and attach it to the debugger.

Spawning a Kernel Task

To spawn a kernel task and attach it to the debugger:

1. In the **Target Manager**, locate your module. For help in locating a module in the **Target Manager**, see [Figure 5-1](#).
2. Select **Target > Debug > Debug Kernel Task**. The **Debug** dialog box appears.
3. In the **Kernel Task to Run** section of the **Main** tab, choose the entry point to use and supply any arguments required by the entry point.
4. In the **Debug Options** tab, check the **Break on Entry** and **Automatically attach spawned Kernel Tasks** check boxes.
5. Click **Debug**. The debugger attaches to the task. The task is added to the list of attached tasks in the **Debug** view.

Attaching the Debugger to a Running Kernel Task

To attach the debugger to a kernel task:

1. In the **Target Manager**, locate the kernel task you want to debug. For help in locating a kernel task in the **Target Manager**, see [Figure 5-1](#).
2. Right click the kernel task and select **Attach to Kernel Task**. The task is added to the list of attached tasks in the **Debug** view.

Attaching the Debugger to a Partition Task

To attach the debugger to a partition task:

1. In the **Target Manager**, locate the partition task that you want to debug. (For help in identifying a partition task in the **Target Manager**, see [Figure 5-1](#).)
2. If the partition task is in the **Suspend** state, use the target console to change to a schedule that includes the partition using the **arincSchedSet** command. After changing the schedule, right click the partition task and select **Refresh**. The state of the partition task should change from **Suspend** to another state.
3. Select the partition task and select **Target > Attach to Protection domain task**. The partition task, and its threads, are displayed in the **Debug** view.

Viewing symbols

You can view the symbols in tasks attached to the debugger using the **Symbol Browser**.

To view symbols, select **Show Debugger Symbols** from the **Symbol Browser** menu. The symbols in all tasks attached to the debugger are displayed. You can filter the symbols by name and type.

To bring up the code that corresponds to a symbol, double click the symbol. The code (assembly or source, as available) will be shown in the **Editor** view.

For more information on the **Symbol Browser**, press the help key.

Associating source code with the task

If your system was compiled in another location, the debugger may not be able to associate the task that you are debugging with its source code. In this case, you can manually associate source code with a task in the debugger.

To associate source code with a task in the debugger:

1. Right click on the task in the debug view and select **Edit Source Lookup**. The **Edit Source Lookup Path** dialog is displayed.
2. Click Add.
3. Select the correct type of source location for your source file.
4. Select the directory that contains your source file.
5. Click **OK** to close the dialog.



NOTE: The core OS and vThreads binaries have been pre-compiled without debug symbols. Thus you cannot associate core OS and vThreads tasks with their source code files unless you first recompile them with debug symbols.

5.7 Controlling Execution

This section describes how to set breakpoints, step through code, and examine the execution state.

Setting Breakpoints

To set a breakpoint:

1. Open the source file for the task in which you want to set a breakpoint.
2. Right click in the editor gutter next to the line where you want to set the breakpoint and select **Breakpoints > Add Breakpoint**. The **Line Breakpoint Properties** dialog is displayed.

3. Click the **Source Lookup** tab and ensure that the source lookup settings are correct.
4. Click on the **Scope** tab and ensure that the correct scope for the breakpoint is set.
5. If you want to place a hardware breakpoint, click on the **Hardware** tab and select the appropriate settings for a hardware breakpoint.
6. Click **OK**.

For more information on the **Line Breakpoint Properties** dialog, press the help key.

You can manage your breakpoints in the Breakpoints view. You can also set expression and data breakpoints. For more information, press the help key.

Break on Data Access

You can set a hardware breakpoint that will break on access to a variable when that variable has a particular value. This means that the breakpoint will be hit either when the variable is read and the current value matches the specified value, or when the variable is written and the new value matches the specified value.

These breakpoints cannot be set in Workbench. They must be set from either the host shell or the target shell. For more information, see the reference entries for **bhv()** and **pdbhv()**.

Stepping Through Source Code

You can step through code using the step button in the **Debug** view. Note that you can step through a kernel or partition task, but you cannot step through a thread. For more information, press the help key.

Navigating the Call Stack

You can navigate the call stack using the **Stack Trace** view. For more information, press the help key.

Viewing Thread Activity

You can view thread activity in Workbench or in WindSh.

In Workbench, you can view thread activity in the **Target Manager**.

The WindSh command **partitionTaskInfoShow()** displays thread information in a partition. There is no equivalent in the partitions to the **ti()** command in the core OS. It is also possible to display a stack trace for threads by using the **partitionTt()**

command in a partition. The **partitionW()** command, when used in a partition, shows the objects each thread is pending on, if any. For more information, see [6. Tools](#).

Displaying APEX and POSIX Objects

WindSh commands are provided to display APEX and POSIX objects in partitions. Alternatively, the **Object Browser** view displays the same information.

5

OS Exception State

If the operating system puts a task into the exception state, which is actually a variation of pended, a pop-up window indicates that an exception has been trapped. In the debugger, the **runstate** icon for the task changes from a running icon to an exception icon. In this state you can use all of the features in the debugger to examine the state of the task when it triggered an exception.

Object Memory Allocation

C++ objects can be allocated in three different ways:

- automatic allocation
- heap allocation (using the **new** operator)
- static allocation

You can toggle the display of these types of objects in the object relationships view.

Global Class

Functions that are not class member functions, and global data, are shown as being members of the **Global** class.

Debugging Constructors and Destructors

This section describes settings and debugger behavior for C++ constructors and destructors.

Constructor and Destructor Calling Ordering

The debugger shows the calling sequence from derived down to base class constructor, for the purpose of passing the initialization parameters. The constructors then execute the code and return in the correct order, from base class constructor through to derived class constructor.

Destructors are called in the reverse order, the most derived executing first, most base class executing last. The debugger shows this order of invocation directly.

Missing Constructors and Destructors

To use the debugger optimally for C++ applications, it is best to provide constructors and destructors for all classes. These can even be null body constructors, if in-lining has been switched off using a compiler flag. For example:

```
class Derived: public Base
{
    public:
        Derived() {}
        ~Base(){}
};
```

They can be created by the compiler by default. For example, as shown below, the C++ compiler generates a default constructor and destructor for any class that is derived from, or an aggregate of, a base class, in order to call that class constructor and destructor:

```
class Base
{
    public:
        Base();
        ~Base();
};
class Derived : public Base
{
    // body goes here but does not have a constructor or destructor
};
```

It is good practice to explicitly provide constructors and destructors for all classes that have member functions. This also allows the debugger to check the validity of all object IDs used in the program. In the case of an invalid object ID value, the debugger generates an error message.

If the debugger is used on a program that does not have default constructors or destructors for all classes or, if static filtering criteria are changed to include previously excluded objects, the debugger may generate unnecessary information messages about unexpected object IDs. This happens the first time it encounters the ID for an object of a class for which it has not detected the invocation of a constructor. If the value is a valid address, the debugger assumes that the address is a valid object address. It then uses that value as the object address in the future.

The debugger does not detect the destruction of a heap-based object that is instantiated from a class that has no destructor, unless one of its base classes has a destructor. Then the debugger detects the deletion of the object.

Inline Functions and Default Constructors and Destructors

To monitor object creation and destruction, constructors and destructors must be defined and implemented for each class. If they are not, the debugger cannot show the object being created or destroyed. This is also true for constructors and destructors that are implemented as *inline*. Inline functions cannot be monitored because they are not implemented as functions. To fully monitor your C++ objects, ensure that all of your classes have constructors and destructors, and that you compile your application with inlining turned off.

Debugging Shared Libraries

Applications linked to a shared library (including the partition OS) share the **TEXT** sections (that is the code). Each has its own per-client copy of the **DATA** sections of the shared library. In the **Target Manager** view, any attached shared library module icons within an application domain are there to indicate that the application is linked to that library.

Debugging of shared libraries is done in the context of application domains. It is in the application domain that you read debug information for the shared library. You cannot read debug information for a shared library in the shared library domain itself.

5.8 Viewing and Manipulating Data

You can modify all data and software in the module using the debugger.



NOTE: You can potentially crash the system by writing invalid data into core OS memory.

Browsing and Editing Data

You can browse and edit data using the **Variables** view and the **Registers** view. For more information on these views, press the help icon in each.

Setting Watches

You can set watches using the **Watch** view. You can watch a variable or a complex expression.

To set a watch:

1. Highlight the variable or expression to watch.
2. Right click on the highlighted expression
3. Select **Watch**. The highlighted expression is added to the watch table in the **Watch** view.

You can also set a watch by entering the expression to watch:

1. Right click in the watch view.
2. Select **Add**. An empty cell is highlighted in the watch table.
3. Type an expression into the empty cell.

For more information on the **Watch** view, press the help key.

5.8.1 Special debugging situations

How do I get information about applications in partitions?

In a partitioned system it is difficult to get access to the standard output of an application running in a partition. If standard output is attached to a serial port, for example, the output of different partitions and the core OS would be interspersed, making it difficult to determine what information came from each partition. To address this problem, VxWorks 653 provides application multiplexed I/O (AMIO). AMIO allows you to monitor the output of multiple partitions over a single serial connection. You can enable AMIO by including its component in your partition. To view AMIO data, VxWorks 653 provides an AMIO-enabled serial monitor, *wrMonitor*. For information on *wrMonitor*, see [6.3 *wrMonitor*](#), p.58.

How do I overcome the constraints of partitions?

If an application uses a standard API like POSIX or APEX, which is supported in VxWorks 653 partitions, you can develop and debug your application in another environment that supports that API and then move it to VxWorks 653 later. This may make it easier to debug the application without the constraints imposed by an ARINC 653-compliant system.

How do I debug application initialization?

Because all partitions are started automatically by the core OS according to a schedule, it can be difficult to debug the initialization routines of applications. You can use the following techniques to debug application initialization.

Debugging Application Initialization Using Warm Restart

Manual invocation of a warm restart of the application allows you to observe failures that occur during initialization.

The procedure for using this debugging technique is:

1. Build and boot a payload image system.
2. Set a system or task mode breakpoint on the application entry point using the debugger or the shell.
3. Invoke a **WARM** restart from the host or target shell:

```
partitionModeSet (partition_number, 2, 0, 0, 0)
```

Debugging Initialization Prior to OS Initialization

The software debugger is not available until the core OS initialization routine starts the target debug agent. To debug code that executes during board hardware initialization or the preliminary stages of core OS initialization, you must use a hardware debugger.

Debugging Application Initialization Using the Scheduler

If the initial automatic execution of the application causes catastrophic application failure, consider using the scheduler to debug application startup. By assigning the time allotted to your application to the core OS instead, you can load the application without ever running it. Once it is loaded, you can set a breakpoint on the entry point using the debugger or a shell. Then, by changing to a schedule that allots time to the application, the breakpoint is fired.

This technique does not require restarting your application and therefore works with any build specification. However, since, in a ROM or RAM payload system, partitions are not loaded until they are scheduled, it is necessary to use a hardware rather than a software breakpoint to initially stop the system. If you are using a downloadable image, partitions are loaded when the image is loaded and before the system is started. Therefore you can use a software breakpoint with a downloadable image.

The procedure for this debugging technique is:

1. In your Module configuration document, change the schedule ID of the default schedule from 0 to the next available number. (In this example, 1 will be used).
2. Add a new default schedule that assigns all time to the core OS by establishing a single partition window with a partition name reference of "SPARE". The changes are highlighted:

```
<Schedules>
  <Schedule Id="0">
    <PartitionWindow
      PartitionNameRef="SPARE"
      Duration="0.25"
      ReleasePoint="true"/>
    </Schedule>
  <Schedule Id="1">
    <PartitionWindow
      PartitionNameRef="my-application-A-partition"
      Duration="0.001"
      ReleasePoint="true"/>
    <PartitionWindow
      PartitionNameRef="my-application-B-partition"
      Duration="0.00025"
      ReleasePoint="true"/>
    <PartitionWindow
      PartitionNameRef="my-application-C-partition"
      Duration="0.002"
      ReleasePoint="true"/>
    </Schedule>
  </Schedules>
```

3. Build and boot the system. If you use a downloadable image, the partition will be loaded but not run. If you use a ROM or RAM payload image, the partition will not be loaded.
4. If you are using a downloadable image, set a breakpoint at the appropriate line of code using the debugger.

If you are using a ROM or RAM payload, look up the virtual address of the partition's system shared library in the PartitionDescription document. Use the host shell to set a hardware breakpoint at that address. For information on the host shell, see [6.4 Shells](#), p.59.

5. From the host or target shell, switch to the schedule that includes the partition to be debugged (in this case, schedule 1):

```
arincSchedSet (1, 2)
```

The breakpoint will be hit, stopping the partition. You can now set any additional breakpoints you need to debug the application.

Debugging Application Initialization by Disabling the Scheduler

You can gain access to the initialization routines of partitions by disabling the initial startup of the scheduler, allowing you to set breakpoints before the schedules are started.

The parameter **FUNCPTR_func_arincSchedEnableHook** has been added to the **arincSchedEnable()** routine. By default, this function pointer is set to NULL. You can set this function pointer to point to a function that sets **arincSchedEnabled** to TRUE or FALSE. If **arincSchedEnabled** is FALSE, the schedule will not start. You can use a hardware signal, or any other trigger you like, to control whether scheduling is started.

Once you have control over the starting of scheduling, the procedure for debugging application initialization is as follows:

1. Boot the system with the scheduler disabled. Once the boot is complete, the core OS, system shared libraries, and shared libraries have been loaded. The partitions have been created, but not started.
2. Connect the debugger to the target.
3. Using the host shell, set a hardware breakpoint at the start of the partition OS by connecting to a partition that uses that partition OS and setting the breakpoint at the virtual address of the start of the partition OS.

Start the scheduler by using the shell to set **arincSchedEnabled** to TRUE. The scheduler will start and the task will break when the breakpoint is hit.

5.8.2 Examining Memory

You can view the contents of target memory in the **Memory** view. For more information, press the help key in the **Memory** view.

5.9 Monitoring Resources

For information on monitoring resources, see [6.5 Monitoring Tools](#), p.62.

5.10 Configuring the Debugger

For information on configuring the debugger, press the help key in the debugger perspective.

6

Tools

- 6.1 Introduction 53
- 6.2 Boot Program 54
- 6.3 wrMonitor 58
- 6.4 Shells 59
- 6.5 Monitoring Tools 62
- 6.6 VxWorks 653 Simulator 71
- 6.7 Configuration and Build Tools 79

6.1 Introduction

This chapter describes some of the tools included with Workbench.

6.2 Boot Program

Boot ROMs for each BSP are located at `$(WIND_BASE)/target/proj/$(BSP)_bootApp`. For information on flashing the boot ROM, see your BSP documentation.

The VxWorks 653 boot program, `bootApp`, is used to bring up a target board and to load a VxWorks 653 system image over the network. To see a list of available commands, type either **h** or **?** at the boot prompt, followed by **ENTER**:

```
[VxWorks 653 Boot]: ?
```

[Table 6-1](#) describes each of the VxWorks 653 boot commands and their arguments.

Table 6-1 **VxWorks 653 Boot Commands**

Command	Description
h	Help command—print a list of available boot commands.
?	Same as h .
@	Boot using the current boot parameters.
p	Print the current boot parameter values.
c	Change the boot parameter values.
l	Load the file using current boot parameters, but without executing.
g <i>adrs</i>	Go to (execute at) hex address <i>adrs</i> .
d <i>adrs</i>[, <i>n</i>]	Display <i>n</i> words of memory starting at hex address <i>adrs</i> . If <i>n</i> is omitted, the default is 64.
m <i>adrs</i>	Modify memory at location <i>adrs</i> (hex). The system prompts for modifications to memory, starting at the specified address. It prints each address, and the current 16-bit value at that address, in turn. You can respond in one of several ways: ENTER : Do not change the address, but continue prompting at the next address. <i>number</i> : Set the value to a 16-bit <i>number</i> .

Table 6-1 VxWorks 653 Boot Commands (cont'd)

Command	Description
.	(dot): Do not change that address, and quit.
f <i>adrs, nbytes, value</i>	Fill <i>nbytes</i> of memory, starting at <i>adrs</i> with <i>value</i> .
t <i>adrs1, adrs2, nbytes</i>	Copy <i>nbytes</i> of memory, starting at <i>adrs1</i> , to <i>adrs2</i> .
s [0 1]	Turn the CPU system controller ON (1) or OFF (0) (only on boards where the system controller can be enabled by software).
e	Display a synopsis of the last occurring VxWorks 653 exception.
v	Display BSP and boot ROM version.
n	Set MAC address of the target board.

Changing Boot Parameters

To change boot parameters:

1. Type **c** at the boot prompt. The first parameter is displayed.
2. Enter a new value for this parameter or press enter to keep the existing value.
3. Repeat for each parameter in turn.

Instead of being prompted for each of the boot parameters, you can supply the boot program with all the parameters on a single line at the boot prompt ([VxWorks Boot]:) beginning with a dollar sign character (“\$”). For example:

```
$ln(0,0)mars:c:\temp\vxWorks e=90.0.0.50 h=90.0.0.1 u=fred pw=...
```

The order of the assigned fields (those containing equal signs) is not important. Omit any assigned fields that are irrelevant. The codes for the assigned fields correspond to the letter codes shown in parentheses by the **p** command. For a full description of the format, see the reference entry for **bootStringToStruct()** in **bootLib**.

Non-volatile RAM (NV-RAM)

If your target CPU has non-volatile RAM (NV-RAM), all the values you enter in the boot parameters are retained in the NV-RAM. In this case, you can let the boot

program auto-boot without having a terminal program connected to the target system.

6.2.1 Description of Boot Parameters

Each of the boot parameters is described below. The letters in parentheses after some parameters indicate how to specify the parameters in the command line boot procedure described in [Changing Boot Parameters](#), p.55.

boot device

The type of device to boot from. This must be one of the drivers included in the boot ROMs (for example, **enp** for a CMC controller). Due to limited space in the boot ROMs, only a few drivers can be included. A list of included drivers is displayed at the console (type **?** or **h**).

unit number

The unit number of the boot device, starting at zero.

processor number

A unique numerical target identifier for systems with multiple targets on a backplane. The backplane master must have its processor number set to zero. For boards not connected to a backplane, a value of zero is typically used but is not required.

host name

The name of the host machine to boot from. This is the name by which the host is known to VxWorks 653; it need not be the name used by the host itself.

file name

The full path name of the VxWorks 653 image to be booted. This path name is also reported to the host when you start a target server, so that it can locate the host-resident image of VxWorks 653. The path name is limited to a 160 byte string, including the null terminator. If the same path name is not suitable for both host and target—for example, if you boot from a disk attached only to the target—you can specify the host path separately to the target server, using the **-c filename** option in the **Advanced Target Server Options** field of the **New Target Server Connection** dialog.

inet on ethernet (e)

The Internet Protocol (IP) address of a target system Ethernet interface, as well as the subnet mask used for that interface. The address consists of the IP address, in dot decimal format, followed by a colon, followed by the mask in hex format (for example, 90.0.0.50:ffffff00).

inet on backplane (b)

The Internet address of a target system with a backplane interface (blank in the example).

host inet (h)

The Internet address of the host to boot from (90.0.0.1 for example).

gateway inet (g)

The Internet address of a gateway node if the host is not on the same network as the target.

user (u)

The user ID that VxWorks 653 uses to access the host for the purpose of boot loading the file specified by the **filename** boot parameter. That user must have permission to read the VxWorks 653 boot-image file.

The user must have FTP or **rsh** access. The **ftp password** boot parameter described below controls how the boot loader accesses the host. For **rsh**, the user must be granted access by adding the user ID to the host's **/etc/host.equiv** file, or more typically to the user's **.rhosts** file (**~userName/.rhosts**).

ftp password (pw)

The **user** password used by the boot loader to access the host using FTP for the purpose of boot loading the file specified by the **filename** boot parameter.

flags (f)

Configuration options specified as a numeric value that is the sum of the values of selected option bits defined below.

- 0x01** = Do not enable the system controller, even if the processor number is 0. (This option is board specific; refer to your target documentation.)
- 0x02** = Load all VxWorks 653 symbols, instead of just globals.
- 0x04** = Do not auto-boot.
- 0x08** = Auto-boot fast (short countdown).
- 0x20** = Disable login security.
- 0x40** = Use BOOTP to get boot parameters.
- 0x80** = Use TFTP to get boot image.
- 0x100** = Use proxy ARP.
- 0x200** = Use WDB agent.
- 0x400** = Set system to debug mode for the error detection and reporting facility.

target name (tn)

The name of the target system to be added to the host table.

startup script (s)

If the kernel shell is included in the downloaded image, this parameter allows you to pass to it the path and filename of a startup script to execute after the system boots. A startup script file can contain only the shell's C interpreter commands. This parameter can also be used to specify process-based applications to run automatically at boot time, if VxWorks 653 has been configured with the appropriate components.

other (o)

This parameter is generally unused and available for applications. It can be used when booting from a local SCSI disk to specify a network interface to be included.

6.3 wrMonitor

The wrMonitor serial-port monitor lets you observe the output from application multiplexed I/O (AMIO). AMIO lets you perform serial text I/O with multiple applications running in different partitions, and with the core OS, over a single serial connection. For more information on AMIO, see the *VxWorks 653 Programmer's Guide*.

To start wrMonitor, in the **Command Prompt** window, move to the following directory:

installDir/**host**/*hostType*/**bin**

Type the following:

wrMonitor

The default configuration is brought up. This includes a window for the core OS.

Partitions will be displayed in new tabs as soon as they generate output. To display partitions that have not yet produced output, click **Setup > Add Partition** and add partitions by number. The **Setup** menu includes other configuration items, such as configuration parameters for serial ports.

The wrMonitor tool includes help.

6.4 Shells

VxWorks 653 requires multiple shells to provide full access to the partitions. In addition to the target shell in the core OS (see the reference entry for **shellLib**) and the host shell (WindSh), the vThreads shell runs in a partition time slot and provides access to more information about the partition than is available to the other shells. There is no COIL shell, however, the target shell of the core OS will provide useful information about COIL-based partitions.

Shell routines that require spawning a task cannot be issued to partitions from either the host or the target shell. They can be used from the core OS only. The partition version of the shell permits using tools such as **i()** and **tt()** within partitions.

Shells cannot call routines in a partition context, only in the core OS context.

6.4.1 Host Shell

The host shell (WindSh) provides the default shell interface to the target. This is the shell you should use unless you need the specific capabilities of the other shells. Each APEX and POSIX partition can be debugged from the shell as a single instance of a core OS task.

6.4.2 Target Shell

The target shell provides access to core OS task information and system memory. However, system mode debugging is not supported from the target shell. Note that the target shell is an unbreakable task, so breakpoints cannot be set on it, or on functions that it calls directly.

The target shell cannot display the Altivec register set. Only a single instance of the target shell, running as a single task, can be run on a VxWorks 653 system. The target shell consumes target resources (memory and CPU time) when it performs control and information functions. When the target shell encounters a string literal in an expression, it allocates space for the string. This memory is never freed unless you explicitly free it.

For more information about the target shell, see the *VxWorks 653 Programmer's Guide*.

6.4.3 vThreads Shell

The vThreads shell runs as a thread in a vThreads partition. Like other vThreads threads, it runs only during the partition time slot.

Configuring the vThreads shell

To configure your application to include the vThreads shell:

1. Ensure that the core OS provided by your platform provider includes the **INCLUDE_PARTITION_SHELL** and **INCLUDE_VAL_WORKER_TASKS** components.
2. Ensure that the partition OS provided by the platform provider includes the **vThreadsShellComponent.o** component. If you want to have the partition OS symbols in the partition symbol table, you must also ensure that the partition OS symbol table (*my-pos-syms.c*) was included in the partition OS build.
3. Include the **vThreadsShell.c** component in the application build.
4. Include the **usrSymTbl.c** component in the application build.
5. Generate the application symbol table (*my-application-syms.c*) and include it in the application. The symbol tables are generated from a **.pm** file containing all your object modules, so you must change how your **.sm** is built. First, build a **.pm** file, then generate the symbol table from the **.pm**, and finally link the **.pm** and symbol table files into the **.sm** file. **Makefile.rules** contains a rule for generating a symbol table from a **.pm** file, so you need only supply the dependency list for the **.pm** and **.sm** files:

In your application make file, change this line (in which `$(PART_OBJS)` represents all the **.o** files that make up your application):

```
my-application.sm: vxMain.o $(PART_OBJS)
```

To this:

```
my-application.pm: $(PART_OBJS) vThreadsShell.o usrSymTbl.o
my-application.sm: vxMain.o my-application.pm my-application-syms.o
```

6. Configure at least one worker task in the **PartitionDescription/Settings/@numWorkerTasks** attribute of the **PartitionDescription** document.
7. Build your application as described in the *VxWorks 653 Configuration and Build Guide*.

Invoking the vThread shell

To invoke the vThreads shell, execute the **attach** command from the shell prompt:

```
[coreOS] -> attach firstPartition
Switched to firstPartition's Shell (Press CTRL+W to exit)

-> iosFdShow
fd name                                drv
 3 /globalIo/0                         1 in
 4 /globalIo/1                         1 out
 5 /globalIo/2                         1 err
value = 50 = 0x32 = '2'
->
```

6

Strengths

The vThreads shell has the following strengths:

- Has visibility into a partitions mapped space, but not outside it.
- Supports all standard show routines.
- Displays thread status.
- Creates and deletes threads.
- Checks the thread stack.
- Runs a stack trace on threads.
- Examines and modifies memory.
- Generates partition memory usage statistics.
- Has a C interpreter with function call capability.

Limitations

The vThreads shell has the following limitations:

- No breakpoint support.
- No dynamic loading of code.

Many partitions can have a shell thread running, but only one can be active at a given time. The core OS redirects input to the active partition.

6.5 Monitoring Tools

The following monitoring tools are available.

6.5.1 Memory Usage Monitoring

The memory usage monitor runs as part of the host shell, and reports the memory usage of various areas of the operating system, including heaps, stacks, ports, and health monitoring. Information displays on a per partition basis in a consistent, easily readable format.



NOTE: The memory usage monitor does not introduce any performance issues to your system. Any changes in performance are due exclusively to the use of WindSh.

To use the memory usage monitor, make sure that you have established a target connection, and that the host shell is running. No additional configuration is required to use the memory usage monitor.



NOTE: Memory usage monitoring is not supported in COIL-based partitions.

For information on the memory usage monitoring commands, consult the API reference for the following routines:

- `memoryUsageHeapShow`
- `memoryUsageStackShow`
- `memoryUsageHmShow`
- `memoryUsagePortShow`
- `memoryUsageAllShow`

6.5.2 Performance Monitoring

The performance monitoring tool allows you to monitor CPU usage in order to analyze a system or part of a system. Use it to monitor the amount of time used in a module, either in the core OS or the partition OS.

Two time monitoring components make up the performance monitoring tool—one for the core OS and one for the partition OS.

The performance monitoring tool is included by default and cannot be removed. If you want to view performance data using the target shell, you must include the component `INCLUDE_KERNEL_SHOW`. For information about including binary components, see [Using the Performance Monitoring Tool](#), p.66.



NOTE: Performance monitoring is not supported for COIL-based partitions. Performance monitoring is not supported in the Simulator.

How the Performance Monitoring Works

The data acquisition portion of the performance monitoring tool is included by default and is controlled by the **timeMonitorLib** routines. Monitoring sessions start when the core OS starts, or when a partition is activated the first time. Monitoring can be stopped on request and started again. Core OS and partition OS monitoring are independent. The core OS sees the partition as a task, and the partition does not see the core OS at all.

Start or stop commands in a partition do not cause any negative impact on the performance of the core OS. Likewise, start and stop commands in the core OS do not cause any performance issues for the partition OS.

Partitions may be switched out without notification. This means that within a partition, there is no way to update current task CPU usage before the partition is switched out. Since the core OS controls this switch, it is responsible for delivering the partition switch-out time to the relevant partition.

Monitoring the Core OS

In the core OS, three elements are monitored:

- **Task Time**

The performance monitoring tool monitors the cumulative amount of time the CPU spends executing tasks from the time that monitoring starts. The **tPartOS** task that runs the partitions is included. There is one time accumulator per task.

Monitoring CPU time for a task starts when a task becomes active and stops when a task becomes inactive. A task may become inactive when another task becomes active or when an interrupt occurs. When the interrupt is finished executing, the task starts again.

- **Idle Time**

The performance monitoring tool records the cumulative amount of time that the system is idle from the time that monitoring starts. Monitoring idle time

starts whenever the system enters an idle loop and stops when it exits the idle loop or when an interrupt occurs.

- **Interrupt Time**

The performance monitoring tool records the amount of time that the system spends in an interrupt from the time that monitoring starts. As with tasks, there is one time accumulator per interrupt. Monitoring CPU time for an interrupt starts when the system enters the ISR. It stops when the system exits the ISR.

vThreads Partition OS

In the partition OS, four elements are monitored:

- **Thread Time**

The performance monitoring tool monitors the cumulative amount of time the CPU spends executing threads from the time that monitoring starts. There is one time accumulator per thread.

In a partition, monitoring starts when the thread becomes active and stops when a pseudo-interrupt occurs or when the partition is scheduled out. If a thread stops because of a pseudo-interrupt, it will start again when the pseudo-interrupt finishes executing. Monitoring CPU time within a partition is not stopped when the partition OS task is preempted by a core OS task.

- **Idle Time**

The performance monitoring tool records the cumulative amount of time that the system has been idle since monitoring started. Monitoring idle time starts when the system enters an idle loop. It stops when exiting the loop, when a pseudo-interrupt occurs, or when the partition is scheduled out.

- **Pseudo-Interrupt Time**

The tool records the amount of time that the system spends in a pseudo-interrupt once monitoring is started. As with tasks, there is one accumulator per pseudo-interrupt. Monitoring CPU time starts when the pseudo-interrupt starts executing, and stops when the pseudo-interrupt execution exits or when the partition is scheduled out.

- **Partition Window**

The performance monitoring tool records the amount of time that a partition is active since monitoring started. Monitoring a partition window starts when a partition becomes active and stops when it becomes inactive. Data collected

in this accumulator is used to compute information like CPU usage inside a partition or percentage of CPU usage for a thread.

Viewing Acquired Data

To view the data acquired by the performance monitoring tool, use the Get routines in **timeMonitorLib**. These functions are used to read the time accumulator values of the system or a task or thread. A series of **Show** routines can also be included (added as a binary component **INCLUDE_KERNEL_SHOW**) that allows you to view the collected data through the target shell.

6

Parameters

The following parameters are available to configure the behavior of the Performance Monitor. These parameters are available in both the core OS and the vThreads partition OS:

Table 6-2 Performance Monitor Parameters

Parameter	How Used
TIME_MONITOR_ENABLE	When this parameter is set to TRUE , performance monitoring starts at start up time. This is equivalent to calling timeMonitorStart() when the core OS starts or when a partition is activated.
MAX_NUM_OF_ACCU	<div>This parameter defines the maximum number of time accumulators available in the core OS and partitions. Time accumulators are required as follows:<ul style="list-style-type: none">▪ 1 per task (all tasks including system tasks)▪ 1 per partition task (vThreads partitions only)▪ 4 extras (idle, ISR, reference, and notUsed)</div>

System Impact

The performance monitoring tool is designed to be non-intrusive for CPU consumption. It runs permanently and takes a fixed overhead during the partition window.

Since the reporting task involves the use of a communication media (such as a network stack and device), it does use some CPU time.

Using the Performance Monitoring Tool

The monitoring portion of the tool is included and cannot be removed. You do not have to make any changes or do any additional configuration to use it.

To view the acquired data from the target shell, you need to include the **INCLUDE_TIME_MONITOR_SHOW** binary component. To view data in the core OS, include the component in your kernel. To view data in a partition OS, include **timeMonitorShowLib.o** in the partition OS.

Once the **INCLUDE_TIME_MONITOR_SHOW** binary component is included, you can use four different **show** commands in the target shell to control the performance monitoring tool and display output.



NOTE: For data to display correctly, you need to use the target shell to execute the **show** commands. Since the **start**, **stop**, and **clear** commands do not return any output, you can enter these commands from WindSh (the host shell) if you prefer. Note that commands entered from WindSh apply to the core OS.

For information on the port time monitoring command, see the API reference for the following routines:

- **timeMonitorClear**
- **timeMonitorStart**
- **timeMonitorStop**
- **timeMonitorShow**
- **timeMonitorAPEXShow**

6.5.3 Port Monitoring

The port monitoring tool is provided so that developers can selectively log log port activity occurring in a system.

The port monitoring tool is composed of three pieces. The first is the low-level instrumentation that is embedded in the sampling and queuing port libraries. This code collects data during port operation and sends it to the monitoring portion of the code. It cannot be removed.

The second piece is the monitoring component, which gets the collected data from the ports and stores it in a buffer. This buffer is allocated to the port specific memory region. The monitoring component can be added or removed from the

system as required. Interaction between the instrumentation and the monitoring components occurs when monitoring is first enabled.



NOTE: Port monitoring is supported for COIL-based partitions.

How Does Port Monitoring Work?

Monitoring

The monitoring portion of the tool oversees a set of connected ports called *channels*. Within each channel a sender point, a distribution point, and a receiver point are monitored.

- **Sender Point**

At the sender point, all messages that are sent by a port and put into a source port are monitored. Information captured about each message at this point includes:

- port type
- timestamp
- message length
- source port identifier
- message identifier

- **Distribution Point**

At the distribution point, all messages taken from the sender port and sent to the destination port are monitored.



NOTE: Sampling ports use a zero-copy algorithm, so there is no distribution point for sampling channels.

Information collected about each message includes:

- port type
- timestamp
- message length
- source port identifier
- destination port identifier
- message identifier



NOTE: When using the **RECEIVER_DISCARD** function, the event of discarding a message when the destination buffer is full is recorded at the distribution point.

▪ **Receiver Point**

At the receiver point, all messages read from a destination port are monitored. Information collected about each message includes:

- port type
- timestamp
- message length
- destination port identifier
- message identifier

Message data monitoring consists of identifying each message and logging its identifier. If reporting is enabled, message data is shown if the message is still in the port buffer at the time of reporting.

Monitored data is stored in a ring buffer, and there is one buffer for the whole system. Specify buffer size at system startup during the call to **portMonitorLibInit()**.



NOTE: Stopping port monitoring does not clear the data collected in the monitoring buffer. If monitoring is not enabled, data is collected in a waste buffer.

Reporting

The reporting portion of the port monitoring tool collects data from the monitoring buffer and writes it to the virtual I/O (VIO) channel of the Wind River Debug (WDB) agent. The VIO channel is a feature of the WDB agent that provides fast, asynchronous communication between the target and the host. Using this channel lets reporting code rely exclusively on the WDB back end without requiring configuration for a specific communication media. For more information on the VIO channel and the WDB agent, see the *Tornado API Programmer's Guide: The WTX Protocol*.

The reporting task is spawned at configlet initialization (**usrPortMonitorInit()**). If monitoring is disabled, the reporting task suspends.

The code for the reporting task is provided as source code in a configlet called **usrPortMonitor.c**. The code can be modified as required, or removed from the system entirely.

For example, a platform provider may wish to modify the code to avoid using the VIO channel and send a UDP broadcast packet through a socket instead.



NOTE: If the bandwidth of the media used for streaming is not large enough, messages may be lost because the monitoring buffer wraps. Any loss of messages is indicated in the report.

System Impact

6

The port monitoring tool is designed to be non-intrusive for CPU consumption. The monitoring portion runs permanently and takes a fixed overhead during the partition window. When data is available, the monitoring portion tells the reporting function that data is available. This exchange is fast and non-blocking regardless of whether the reporting task is included.

Since the reporting task involves the use of a communication media (such as a network stack and device), it does use some CPU time. To avoid affecting the partition timing, the reporting task is scheduled as a low priority task that only runs when the system is idle, or is scheduled within its own time window, set aside in the major time frame.

Using the Port Monitoring Tool

The use of monitoring and reporting is handled slightly differently. Enable or disable the monitoring of a channel using routines made from the host shell or the target shell, or programmed into the core OS.

Reporting is also enabled and disabled using routines; however, calls to these can be made only from a shell (either the host shell or the target shell). Reporting routines cannot be programmed into the core OS because the reporting function is part of the tools, and tools can be included or removed from the system with no impact on the kernel domain.

The display used for the reporting tool is **wtxConsole**, which is included with Workbench, or the **portMonitorLib.tcl** application included in *installDir/host/resource/tcl/*. Launch **wtxConsole** with the option to display all data coming from VIO channel 99.

For assistance in writing a C or TCL application that reads data from the VIO channel, see the API reference entry for **wtxVioLink**. This type of code could be used to filter the data or log it to a file.

Executing Port Monitoring Commands

There are two main port monitoring actions available:

- Start or stop port monitoring of a channel
- Enable or disable port monitoring reporting

These two actions are independent of each other and do not need to be synchronized. The following examples illustrate different ways of combining the monitoring and reporting commands.



NOTE: These examples show the results of various commands in a general way. For command syntax, see [Port Monitoring Functions](#), p.70.

Enabling Reporting First

1. Enable reporting.
2. Start monitoring on Channel A. Events from Channel A start being displayed.
3. Start monitoring on Channel B. Events from Channel A and Channel B are displayed.
4. Disable reporting.

Start Monitoring First

1. Start monitoring on Channel A. Nothing is displayed.
2. Start monitoring on Channel B. Nothing is displayed.
3. Enable reporting. Events from Channel A and Channel B are displayed.
4. Stop monitoring on Channel B. Events from Channel A are displayed.
5. Disable reporting.

Port Monitoring Functions

For information on the port monitoring commands, see the API reference entry for **portMonitorLib**.

6.6 VxWorks 653 Simulator

The VxWorks 653 simulator (VxSim) is a port of VxWorks 653 to the Windows host architectures. It provides a simulated target for use as a prototyping and test-bed environment. In most regards, its capabilities are identical to a true VxWorks 653 system running on target hardware.

In the simulator, the image is executed on the host computer as a host process. There is no emulation of instructions, because the code is for the host's own architecture.

The host-based simulation environment supports partitioning and provides behavior-level simulation for the APIs of the operating system (including POSIX and APEX APIs). However, the host simulation environment does not provide instruction set simulation.

The host simulator and partitions are built with the standard VxWorks 653 Pentium compiler, which generates objects in ELF format.

6.6.1 Running the Simulator

Starting the Simulator

You can launch the simulator from within Workbench as described in [3.2 Booting VxWorks 653 on the Simulator](#), p.22.

The simulator can also be started from the command line by typing **vxsim** (the executable is found in *installDir/host/hostType/bin*). By default, it looks for the file to boot (**boot.txt**) in the current directory.

For additional options when running vxsim from the command line, run

```
vxsim -help
```

Rebooting the Simulator

As with other targets, you can reboot the simulator by typing **CTRL+X** in the shell.

Exiting the Simulator

Close the simulator window.

6.6.2 File Systems

The simulator can use any VxWorks 653 file system.

The default file system is the pass-through file system, **passFs**, which is unique to the simulator. **passFs** allows direct access to any files on the host. Essentially, the VxWorks 653 functions **open()**, **read()**, **write()**, and **close()** eventually call the host equivalents in the host library **libc.a**. With **passFs**, you can open any file available on the host, including NFS-mounted files. By default, **NTPASSFS** is included in the kernel protection domain to cause this file system to be mounted on startup.

In order to prevent the simulator from misinterpreting the colon (:) in the path name for a VxWorks 653 device, you need to specify, in the target shell only, the host name at the beginning of a full path. In the following examples, note the use of the forward slash (/):

```
[vxKernel] -> cd "myHostName:c:/myPath"  
[vxKernel] -> ml < myHostName:d:/projDir/test/CPUgnu.debug/test.o
```

For more information on **passFs**, see the reference entry for **passFsLib**. For more information on other VxWorks 653 file systems, see the *VxWorks 653 Programmer's Guide*.

6.6.3 Building a Module for the Simulator

In order to build successfully for the simulator, you must adjust the memory map of your module to conform with the memory map of the simulator. This involves adjusting all memory values that must align on CPU page sizes to the page size of the simulator, and assigning virtual addresses that do not conflict with those used by the simulator. For more information, see the documentation for the **simpc** BSP.

To build your module for the simulator you specify "SIMNT" as the name of the CPU and "simpc" as the name of the BSP when building each component of the module. For information on configuring and building modules, see the *VxWorks 653 Configuration and Build Guide*.

6.6.4 Differences between the Simulator and VxWorks 653

In comparison to the VxWorks 653 operating system running on a hardware target, the simulator has several limitations:

- The execution speed of the simulator is dependent on your host, on what simulator services you are using, and on what else is running on the host. Thus you cannot predict the performance of your application on a hardware target based on its performance on the simulator.
- For the simulator, there is no difference between user and supervisor modes. All the tasks, including threads running in partitions, run in the equivalent of supervisor mode. This means that code executing in a partition can access the core OS on the simulator, which it cannot do in real targets.
- There is no stack overflow detection for core OS tasks.
- The only clock available on the simulator is the system clock. This means that the time resolution will not be better than 100 ticks per second.
- No memory auto-sizing is available for the simulator. The size of the memory available for the simulator is specified during kernel configuration and must be rebuilt to take effect.
- Only **COLD** starts are available on the simulator; no restart or **WARM** start is available.
- Only a limited number of device drivers are available with the simulator.
- The performance monitor is not supported.
- Online loaded partitions are not supported.
- ROM and RAM payloads are not supported. The simulator does not support payload images. For this reason, it also does not support partition restart.
- The certifiable build image of vThreads and the core OS are not supported.
- The simulator does not provide a full MMU as real targets do. Associated limitations are described in [Protection Domains and Partitions](#), p.74.
- The simulator defaults to using a pass-through file system (passFs) to access files directly on the workstation. (See the online reference for **passFsLib** under **VxWorks AE Reference Manual > Libraries**.)

Because target hardware interaction is not possible, device driver development may not be suitable for simulation. However, the VxWorks 653 scheduler is implemented in the host process, maintaining true tasking interaction with respect to priorities and preemption. This means that any application that is written in a portable style and with minimal hardware interaction should be portable between the simulator and VxWorks 653.

Architecture Considerations

The information in this section highlights differences between the simulator and other VxWorks 653 BSPs. These differences should be taken into consideration as you develop applications on the simulator that will eventually be ported to another target architecture.

The simulator uses the VxWorks 653 scheduler, which behaves the same way as for any other VxWorks 653 architecture. The BSP is extensible. For example, pseudo-drivers can be written for additional timers, and serial drivers.

The rest of this section discusses some details of the simulator implementation. Differences between the simulator and other VxWorks 653 environments are noted where appropriate.

Supported Configurations

Most of the optional features and device drivers for VxWorks 653 are supported by the simulator. The few that are not are hardware devices (SCSI, Ethernet) and ROM and RAM payloads.

Protection Domains and Partitions

The simulator provides partial support for protection domains and, therefore, for partitions. Because there is no difference between user and supervisor modes, it is not possible to prevent one domain from directly accessing the memory of another. Moreover, there is no stack overflow detection mechanism.

Simulator Timeout

Occasionally a simulator session loses its target server connection due to the many things competing for CPU time on the host. If you find that your application is frequently losing its target server connection, adjust the back end timeout (**-Bt**) and back end retry (**-Br**) parameters.

1. In the **Target Manager** view, right-click the active session, select **Properties**.
2. Select the **Target Server Options** tab.
3. In the **Advanced Target Server Options** panel, click the **Edit** button.
4. Enter appropriate values for **Backend Request Timeout** and **Backend Request Resent Number**.

The BSP Directory

Aside from the following exceptions, the **simpc** BSP is similar to other VxWorks 653 BSPs:

- The **sysLib.c** module contains the same essential functions: **sysModel()**, **sysHwInit()**, and **sysClkConnect()** through **sysNvRamSet()**. Because there is no bus, **sysBusToLocalAdrs()** and related functions have no effect.
- The file **00kernel.ddf** contains the default kernel domain configuration. By default, it removes the networking facility, which is not supported by the simulator in VxWorks 653.
- You may experience address conflicts with associated error messages if the default addresses used by the simulator are not available in the simulator address space. This might occur, for example, if a DLL is already mapped to that address or if you modified the memory configuration and the default address areas no longer fit.
- The BSP file **sysLib.c** can be extended to emulate the eventual target hardware.

Correcting Memory Map Errors

[Table 6-3](#) illustrates the Windows simulator memory map. Physical and virtual address spaces are mixed into the simulator process address space. Physical addresses are determined by attributes in CoreOSDescription document for the core OS.

Table 6-3 Simulator Memory Map

Physical address space	Starting address (Default)	How calculated
KernelMemPoolRgn	0x20000000	PhysicalMemory/@BaseAddress
ConfigRecordPoolRgn	0x20400000	kernelMemPoolRgn start address + PhysicalMemory/kernelMemoryRegion/@Size
KernelPgPoolRgn	0x20410000	configRecordPoolRgn start address + PhysicalMemory/kernelConfigRecordRegion/@Size
portPgPoolRgn	0x20810000	kernelPgPoolRgn start address + PhysicalMemory/kernelPgPool/@Size
hmlogPgPoolRgn	0x20a10000	portPgPoolRgn start address + PhysicalMemory/portRegion/@Size
SdPgPoolRgn	Allocated consecutively starting at PhysicalMemory/userMemoryRegion/@Base_Address	
SlPgPoolRgn		
Part1PgPoolRgn		
Part2PgPoolRgn		
Virtual address space (Space for static protection domains; should be completely available)	Start: PhysicalMemory/@Base_Address (0x20000000) End: PhysicalMemory/@Base_Address + KernelConfiguration/@addressSpaceSize (0x80000000)	
Partitions	Start: kernelConfiguration/@partitionVirtualAddress (0x28000000) End: kernelConfiguration/@partitionVirtualAddress + PD_DEFAULT_SIZE (0x30000000)	
Ending address (Default):	PhysicalMemory/@Base_Address + KernelConfiguration/@addressSpaceSize (0x80000000)	

KernelMemPool Region Start Address

If the default values do not fit on your host, the simulator exits showing a message similar to the following:

```
Failed to allocate 0x400000 at 0x20000000
Trying to find a valid configuration:
Please update memory configuration of your kernel domain.
Update Physical Regions Parameters with the following values :
KERNEL_MEM_POOL_RGN_LOG_START: 0x20400000
...
If kernelMemPool is modified then you also have to modify TEXT_VIRT_ADDR
and VIRT_ADDR parameters in kernel domain's attributes.
```

To update the **KernelMemPool** region start address, you need to update the **HardwareConfiguration/PhysicalMemory/@BaseAddress** attribute in the **CoreOSDescription** document.

Core OS Memory Attributes

If you modified the kernel memory pool configuration, you must also update related core OS attributes in the **CoreOSDescription** document. Change the value of the attribute:

KernelConfiguration/@kernelVirtualAddress

to be equal to:

HardwareConfiguration/PhysicalMemory/@Base_Address + 0x10000

Application Base Address Not Available

The default application domain base address may also be unavailable. In such a case, the following error message is output to the simulator console:

```
Can't reserve user PD address space at 0x60000000
Please update PD_DEFAULT_BASE_ADRS parameter (PD component) with the
value 0x60600000.
```

Update the attribute **KernelConfiguration/@partitionVirtualAddress** in the **CoreOSDescription** document to the specified value.

ADR_SPACE_BASE

The last parameters shown on the memory map are **PhysicalMemory/@Base_Address** and **KernelConfiguration/@addressSpaceSize**. The run-time side checks and isolates blocks in this range already used by the process. However, when building static protection domains, they are mapped starting at **PhysicalMemory/@Base_Address**, by default. In such a case, the following message results:

```
Can't use virtual address space at 0x22400000 for domain simpc_eliza_lib

Please update ADR_SPACE_BASE parameter (ADR_SPACE_LIB component) and
PD_BASE_ADDR (PD component) with the value 0x22600000.

Domain init failed for `simpc_eliza_lib`
pdBootCreate: 0x7f0004
Simulator stopped
Hit any key to Exit
```

To fix this problem, modify the **KernelConfiguration/@addressSpaceSize** attribute of the CoreOSDescription document.

Interrupts

Windows messages are used to simulate hardware interrupts. For example, the simulator uses messages 0xc000 through 0xc010 to simulate interrupts from the pipe back end. The messages are the simulator equivalent to ISRs on other VxWorks 653 targets. You can install ISRs in the simulator to handle these simulated interrupts. Not all VxWorks 653 routines can be called from ISRs; see *VxWorks AE Programmer's Guide: Multitasking*. To run ISR code during a future system clock interrupt, use the watchdog timer facilities. To run ISR code during auxiliary clock interrupts, use the **sysAuxClkxxx()** functions.

Table 6-4 shows how the message table is set up.

Table 6-4 Interrupt Assignments

Interrupts	Assigned To
0xc000-0xc010	host messages
0xc011 on	available for user messages

Pseudo-drivers can be created to use these interrupts. Interrupt code must be connected with the standard VxWorks **intConnect()** mechanism.

For example, to install an ISR that logs a message whenever host message **WM_TIMER_CLOCK** arrives, execute the following from the shell:

```
[vxKernel] -> intConnect (0xc011, logMsg, "Help!\n")
```

Then send message 0xc011 to the simulator from a host task. Every time the message is received, the ISR (**logMsg()** in this case) runs.

If a simulator task reads from a host device, the task would normally block while reading; however, this would stop the simulator process entirely until data is ready. Instead, the device is put into asynchronous mode so that a message is sent

whenever data becomes ready. In this case, an input ISR reads the data, puts it in a buffer, and unblocks some waiting task.

Since the simulator uses the task's stack when taking interrupts, the task stacks are artificially inflated to compensate.

Clock and Timing Issues

The execution times of simulator functions are not, in general, the same as on a real target. For example, the VxWorks **intLock()** routine is normally very fast because it just writes to the processor status register. However, under the simulator, **intLock()** is relatively slow because it takes a host semaphore, allowing other processes to run.

The simulator is not a target, and, therefore, has no target clock. Clock facilities are provided by the host thread sending messages for both the system and auxiliary clocks. This technique produces inaccurate timings when the simulator is swapped out as a host process. However, in general, the timing of the simulator is different from an actual target.



NOTE: Because the simulator is a host process, it shares resources with all other processes and is swapped in and out. In addition, the kernel's idle loop has been modified to suspend the simulator until a signal arrives (rather than busy waiting), thus allowing other processes to run.

The **spy()** facility is built on top of the auxiliary clock. The task monitoring occurs during each interrupt of the auxiliary clock to see which task is executing or whether the kernel is executing. Because the profiling timer includes host system time and user time, discrepancies can occur, especially if intensive host I/O occurs.

6.7 Configuration and Build Tools

This section lists the tools used in the configuration and build process for VxWorks 653



NOTE: Not all the applications used in the configuration and build process are invoked directly by the user. In some cases, the tools are invoked by other tools.

6.7.1 XMLGen

XMLGen is wrapper around a collection of tools used to build different elements of a system. These tools read XML configuration files and create various files required by the build. For help, see the reference entry for **XMLGen**. For help on the underlying tools, see the reference entries for the following:

- **xmlBinFlagsGen**
- **xmlCfgGen**
- **xmlFileListGen**
- **xmlLinkageGen**
- **xmlLdsGen**
- **xmlMemMacrosGen**
- **xmlNetbootGen**
- **bbSizeCheck**
- **xmlVirtAddrGen**

Shared library Virtual Address

You can use XMLGen to calculate a virtual address for a shared library:

1. If you have a Module configuration file for your module, you can use it in the steps that follow. If not, create an XML file named **temp-module.xml** in the same directory as your shared library files list each of your shared libraries, following the format of a normal Module configuration document:

```
<Module>
  <CoreOS>
    <xi:include href="bsp.xml" />
  </CoreOS>
  <SharedLibraryRegions>
    <SharedLibrary Name="pos">
      <xi:include href="hello-pos.xml" />
    </SharedLibrary>
  </SharedLibraryRegions>
</Module>
```

2. Open your shared library configuration file and completely remove the **SharedLibraryDescription/@VirtualAddress** attribute. Make sure that you remove both the attribute and its value, not just the value.
3. Open the **VxWorks 653 Development Shell** and change the current directory to your shared library project directory.
4. Run the following command:

```
xmlgen --virtAddr --region pos temp-module.xml
```

The command will print out a hexadecimal number. For example:

0x10000000

This value is a virtual address that is suitable for loading the system shared library that contains the partition operating system.

5. Open the SharedLibraryDescription document for your shared library and restore the virtual address attribute in its original place. For example, the file will now look something like this:

```
<SharedLibraryDescription
  xmlns="http://www.windriver.com/vxWorks653/ConfigRecord"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.windriver.com/vxWorks653/ConfigRecord
target/config/xml/cleanschema/Application.xsd"
  SystemSharedLibrary="true"
  VirtualAddress="0x10000000">
    <MemorySize
      MemorySizeBss="0x4000"
      MemorySizeText="0x40000"
      MemorySizeData="0x8000"
      MemorySizeRoData="0x4000"
    />
  </SharedLibraryDescription>
```

6

6.7.2 crDump

The configuration and build procedures no longer produce the following files:

- **configRecord.c**
- **prjMemConfig.c**
- **payloadMap.c**
- **payloadMap_ram.c**
- **payloadMap_rom.c**

Since these files can be useful for debugging, you can use the **crDump** utility to create these files from **configRecord.reloc**, **payloadMap_ram.o**, and **payloadMap_rom.o**.

crDump is provided in two versions based on the platform for which the system was built. The two versions are:

- **crDumppentium**
- **crDumpppc**

To create **configRecord.c** and **prjMemConfig.c** enter the following command:

```
crDumppentium configRecord.reloc
```

or

```
crDumpppc configRecord.reloc
```

To create the payload map files, enter the following commands:

```
crDumppentium payloadMap_ram.o  
crDumppentium payloadMap_rom.o
```

or

```
crDumpppc payloadMap_ram.o  
crDumpppc payloadMap_rom.o
```

6.7.3 VxWorks 653 2.2 Development Shell

The VxWorks 653 2.2 Development Shell is used to run the VxWorks 653 configuration and build tools. You can start the Development Shell from the program list

Wind River > VxWorks 653 2.2 > VxWorks 653 2.2 Development Shell.

6.7.4 VerIMAx

VerIMAx is an XML processing and validation tool for the XML configuration schemas. It is used for the following purposes:

Generating the Configuration Record

VerIMAx is used by the build tools to create a binary configuration file (**configRecord.reloc/configRecord.bin**) and a payload manifest (**payloadMap_ram.o** or **payloadMap_rom.o**) from the Module configuration document.

Checking Consistency

You can use VerIMAx independently of the build process to check your configuration files for errors. The check ensures that your configuration file follows the business rules implemented by VerIMAx. To use the checker feature you must use a special version of the schemas which includes the business rule information used by VerIMAx. These schema are found in the directory *installDir/target/config/xml/schema/checker/arch/[ppc | simnt]*. They have the same names as the regular schemas, but the names are prefixed by "VXCC_", so that **Module.xsd** becomes **VXCC_Module.xsd**.

To check your Module configuration document, run the following command:


```
VerIMax
installDir/target/config/xml/schema/checker/arch/[ppc|simnt]/VXCC_Module.xsd
my-module.xml
```

You can also use VerIMax to check any of the component XML configuration files. For instance, to check your ApplicationDescription document, run the following command:

```
VerIMax
installDir/target/config/xml/schema/checker/arch/[ppc|simnt]/VXCC_Application.xsd
my-application.xml
```

6.7.5 Table Viewer

Table Viewer is a combination of two tools, VerIMax and VeroStyle which are used together to generate and format reports on various aspects of the configuration of a module and its constituent parts.

The Table Viewer can generate reports for each of the following:

- ACE
- applications
- core OS
- the health monitor
- memory layout
- modules
- partitions
- payloads
- port connections
- pseudo-partitions
- schedules
- shared data
- shared libraries
- Wind River extensions

For each report, an XML report file is generated. The XML report file can be translated into HTML and Microsoft Word (DOC) format for easier printing and reviewing using **VeroStyle**.

Generating a Complete Set of Reports

You can use the **sample** script to generate a complete set of reports using the following procedure:

1. Create **configRecord.xml** as described in the *VxWorks 653 Configuration and Build Guide*.
2. Locate the directory that contains the **configRecord.xml** file in the build output. In the instructions that follow, this directory will be referred to as *configDir*.
3. Copy the *installDir/target/config/xml/tabgen* directory and all its contents to *configDir*, so that you have a *configDir/tabgen* directory.
4. Copy *configDir/tabgen/sample.bat* to *configDir*.
5. Open a command prompt and run:

```
installDir/host/WIND_HOST_TYPE/bin/torVars
```
6. Change to *configDir* and run the sample script, specifying the name of the **configRecord.xml** file:

```
sample configRecord.xml
```

The reports, in HTML and DOC format, are created in *configDir*.

To create a subset of the reports, you may edit the sample script to specify only those reports that you want to create. You may also generate reports from any of the XML configuration files (ApplicationDescription, CoreOSDescription, etc.) however, you must use **configRecord.xml** if you want a report that includes the memory allocations created by the build tools. An error will occur if you request a report and specify a configuration file that does not contain the information required for the report.

Generating Individual Reports

To create an individual report, use the following procedure:

Step 1: Select the appropriate input file.

Reports are based on XML configuration files. You can generate all reports from the **configRecord.xml** file generated by the build process. You may also generate reports from any of the XML configuration files (ApplicationDescription, CoreOSDescription, etc.) however, you must use **configRecord.xml** if you want a report that includes the memory allocations created by the build tools. An error will occur if you request a report and specify a configuration file that does not contain the information required for the report.

In the instructions that follow, **configRecord.xml** will be used to illustrate the command. You may substitute any appropriate configuration file.

For information of building **configRecord.xml**, see the *VxWorks 653 Configuration and Build Guide*.

Step 2: Select the report to run.

Each report is created using a different schema file. Select the report you want to run from the list in [Table 6-5](#) and note the associated schema file. In the steps that follow, the module table report will be created using the **moduleTable/VXMT_WR_ConfigRecord.xsd** schema.

6

Table 6-5 XML Report Types and Filenames

Report Type	schemaFile	Report Output File
ACE table	ACETable/VXACET_WR_ConfigRecord.xsd	AceTable.xml
Application tables	applicationTable/VXAT_WR_ConfigRecord.xsd	ApplicationTable.xml
Core OS table	coreOSTable/VXCOST_WR_ConfigRecord.xsd	CoreOSTable.xml
Port connection data	connectionTable/VXCT_WR_ConfigRecord.xsd	ConnectionTable.xml
Health management tables	healthMonitorTables/\VXHMT_WR_ConfigRecord.xsd	HealthMonitorTables.xml
Memory layout tables	memoryLayoutTable/VXMLT_WR_ConfigRecord.xsd	MemoryLayoutTable.xml
Module tables	moduleTable/VXMT_WR_ConfigRecord.xsd	ModuleTable.xml
Payload tables	payloadsTable/VXPLT_WR_ConfigRecord.xsd	PayloadsTable.xml
Pseudo-partition table	pseudoPartitionTable/VXPPT_WR_ConfigRecord.xsd	PseudoPartitionTable.xml
Partition tables	partitionTable/VXPT_WR_ConfigRecord.xsd	PartitionTable.xml
Shared data tables	sharedDataTable/VXSDT_WR_ConfigRecord.xsd	SharedDataTable.xml

Table 6-5 XML Report Types and Filenames (cont'd)

Report Type	schemaFile	Report Output File
Shared library tables	sharedLibraryTable/VXSLT_WR_ConfigRecord.xsd	SharedLibraryTable.xml
Schedule tables	schedulesTable/VXST_WR_ConfigRecord.xsd	SchedulesTable.xml
Wind River extension table	WindRiverExtensionsTable/VXWRET_WR_ConfigRecord.xsd	WRExtensionsTable.xml

Step 3: Generate XML Report Files.

Generate an XML report file using **VerIMAx.exe**:

1. Open the VxWorks 653 Development Shell.
2. Change to the directory that contains your **configRecord.xml** file.
3. Create the XML report file by running VerIMAx and specifying the schema that corresponds to the report you want to create and the XML configuration file to report on:

```
VerIMAx %WIND_BASE%/target/config/xml/schema/tabgen/moduleTable/  
VXMT_WR_ConfigRecord.xsd configRecord.xml
```

The XML report file is generated in the current directory. The filenames of XML report files are listed in [Table 6-5](#).

Step 4: Format the Report Files.

You can format the reports in HTML or DOC format using **VeroStyle**. **VeroStyle** requires a driver file for each report file. These driver files are shown in [Table 6-6](#).

Table 6-6 XML Report Files, Driver Files, and Output Files

XML Report File	driverFile	Output Files
AceTable.xml	tabgen/drivers/AceTable.txt	AceTable.doc AceTable.htm
ApplicationTable.xml	tabgen/drivers/ApplicationTable.txt	ApplicationTable.doc ApplicationTable.htm
CoreOSTable.xml	tabgen/drivers/CoreOSTable.txt	CoreOSTable.doc CoreOSTable.htm

Table 6-6 XML Report Files, Driver Files, and Output Files (cont'd)

XML Report File	driverFile	Output Files
ConnectionTable.xml	tabgen/drivers/ConnectionTable.txt	ConnectionTableApp.doc ConnectionTableApp.htm ConnectionTablePart.doc ConnectionTablePart.htm
HealthMonitorTables.xml	tabgen/drivers/HMTable.txt	HealthMonitorTables.doc HealthMonitorTables.htm
MemoryLayoutTable.xml	tabgen/drivers/MemoryLayoutTable.txt	MemoryLayoutTable.doc MemoryLayoutTable.htm
ModuleTable.xml	tabgen/drivers/ModuleTable.txt	ModuleTable.doc ModuleTable.htm
PayloadsTable.xml	tabgen/drivers/PayloadsTable.txt	PayloadsTable.doc PayloadsTable.htm
PseudoPartitionTable.xml	tabgen/drivers/PseudoPartitionTable.txt	PseudoPartitionTable.doc PseudoPartitionTable.htm
PartitionTable.xml	tabgen/drivers/PartitionTable.txt	PartitionTable.doc PartitionTable.htm
SharedDataTable.xml	tabgen/drivers/SharedDataTable.txt	SharedDataTable.doc SharedDataTable.htm
SharedLibraryTable.xml	tabgen/drivers/SharedLibraryTable.txt	SharedLibraryTable.doc SharedLibraryTable.htm
SchedulesTable.xml	tabgen/drivers/SchedulesTable.txt	SchedulesTable.doc SchedulesTable.htm
WRExtensionsTable.xml	tabgen/drivers/WRExtensionsTable.txt	WRExtensionsTable.doc WRExtensionsTable.htm

The **tabgen/drivers/ModuleTable.txt** driver file will be used in the procedure that follows.

1. Copy the *installDir/target/config/xml/tabgen* directory and all of its contents to the directory that contains the XML report file (**ModuleTable.xml** in this example).

2. Format the report by running **VeroStyle** and specifying the name of the driver file to use:

```
VeroStyle tabgen/drivers/ModuleTable.txt
```

The formatted HTML and DOC files are created in the current working directory.

6.7.6 VeroStyle

VeroStyle is a formatting application used with the table viewer function of **VerIMax**. See [6.7.4 VerIMax](#), p.82 for details.

A

Debugger Tutorial

[A.1 Introduction](#) 89

[A.2 Debugger Tutorial](#) 89

A.1 Introduction

This tutorial is intended to assist you in understanding the steps required to attach the debugger to your system running on a target. Because VxWorks 653 projects are not supported on Workbench 2.6.1, you cannot use Workbench projects and launch configurations to attach the debugger to your system. For more information, see [5. Debug](#).

A.2 Debugger Tutorial

This tutorial will use the demo project found at *installDir/vxworks653-2.2/target/src/demos/simple653Module*.

To run the demo:

Step 1: Build the demo project

You must build the demo project for the target you are using. Since each target has a different memory map, some memory values have to be specified during the build process. Depending on the target and BSP you are using, you may find that you have to make additional configuration changes to get the demo to build. For more information on the configuration and build system, see the *VxWorks 653 Configuration and Build Guide*.

To build the demo project:

1. From the program list, select **Wind River > VxWorks 653 2.2 > VxWorks 653 2.2 Development Shell**. The VxWorks 653 Development Shell opens.

2. Change to the demo directory:

```
cd vxworks653-2.2/target/src/demos/simple653Module
```

3. Type **make** specifying the “create” target and giving the appropriate values for the **BSP** and **CPU** build variables. For instance, to build for the simulator, you would run:

```
make CPU=SIMMT BSP=simpc create
```

This creates a build project in the directory **vxworks653-2.2/target/proj/BSP_simple653Module**. For example, if you were building for the simulator, the build project would be created in **vxworks653-2.2/target/proj/simpc_simple653Module**.

4. In the project directory created by the previous step, locate and open the file **pos.xml**. This file is the SharedLibraryDescription document for the partition OS.
5. Replace the string **\$(SSL_ADDR)** with an appropriate virtual address. You can find a suitable default in the BSP directory for your BSP in the file **BSP_default.xml**. For example, the default value for the simulator, as defined in the file *installDir/vxworks653-2.2/target/config/simpc/simpc_default.xml* is **0x50000000**.
6. Determine the default partition virtual address for your BSP. This is found in the CoreOSDescription document for your BSP, which is named **BSP.xml**. For example, the name of the CoreOSDescription document for the simulator is **simpc.xml**. The value of the partition virtual address is found in the attribute **/CoreOSDescription/@partitionVirtualAddress**. For example, the default value for the simulator, as defined in the file *installDir/vxworks653-2.2/target/config/simpc/simpc.xml* is **0x28000000**.

7. In the VxWorks Development Shell enter the **make** command using the **buildproj** target and specifying the appropriate values for the **CPU**, **BSP**, and **PARTADDR** (partition virtual address) build variables. For instance, to build for the simulator you would use the following command:

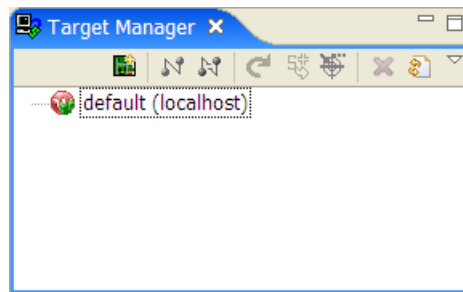
```
make CPU=SIMNT BSP=simpc PARTADDR=0x28000000 buildproj
```

The project will be built.

Step 2: Run the demo system on the simulator

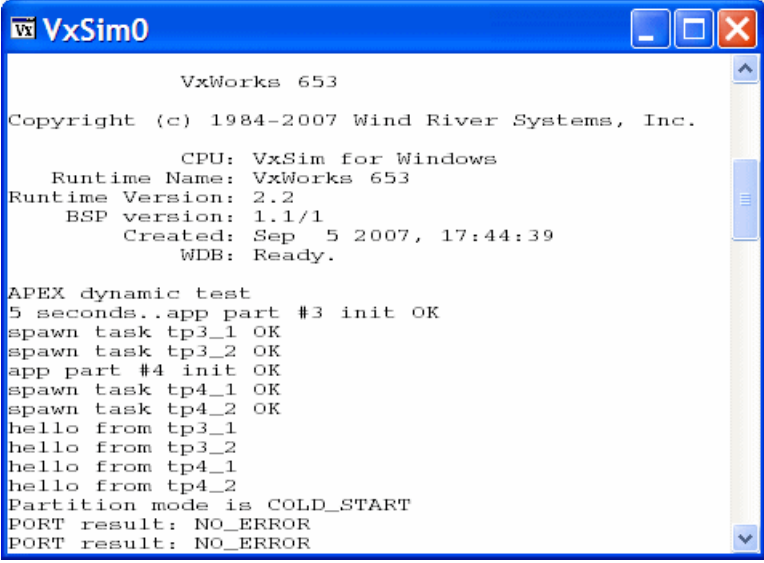
To run the demo project on the simulator:

1. Open Workbench.
2. Bypass any introductory screens until you get to the **Application Development** perspective. If you are asked to select a workspace, accept the default. For information on the **Application Development** perspective, press the help key.
3. Locate the **Target Manager** view. It should appear similar to the illustration below.



4. Select **Target > New Connection**. The **New Connection** dialog appears.
5. Select **Wind River VxWorks 653 Simulator Connection** and click **Next**.
6. Select the **Custom Simulator** option. The **VxWorks 653 Boot File** field becomes available.
7. Click **Browse** and select the file *installDir/vxworks653-2.2/target/proj/CPU_simple653Module/integration/boot.txt*.
8. Click **Next** until the **Target State Refresh** page is displayed.

9. Check all the items under the heading **Initial Target State Query Settings**.
10. Click **Next** until the **Connection Summary** page is displayed.
11. Verify that **Immediately Connect to Target if Possible** is checked.
12. Click **Finish**. The simulator console opens, showing the output as the module boots and the applications in the project begin to run and produce output.

A screenshot of a Windows-style window titled "VxSim0". The window contains a text area with the following output:

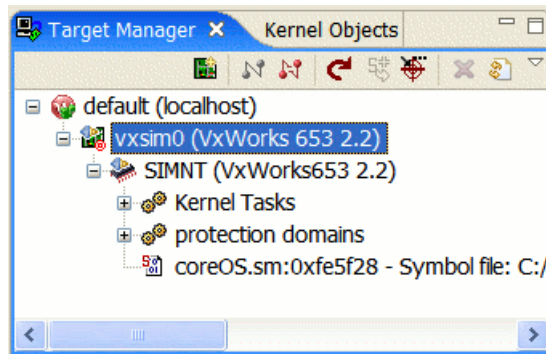
```
VxWorks 653
Copyright (c) 1984-2007 Wind River Systems, Inc.

      CPU: VxSim for Windows
Runtime Name: VxWorks 653
Runtime Version: 2.2
  BSP version: 1.1/1
    Created: Sep  5 2007, 17:44:39
      WDB: Ready.

APEX dynamic test
5 seconds..app part #3 init OK
spawn task tp3_1 OK
spawn task tp3_2 OK
app part #4 init OK
spawn task tp4_1 OK
spawn task tp4_2 OK
hello from tp3_1
hello from tp3_2
hello from tp4_1
hello from tp4_2
Partition mode is COLD_START
PORT result: NO_ERROR
PORT result: NO_ERROR
```

The window has a blue title bar with standard Windows controls (minimize, maximize, close) on the right.

Within Workbench, the connection is displayed in the **Target Manager**. It may take some time for the simulator to finish booting and for the connection to the target to be completed. When the connection is complete, the **Target Manager** view will look something like this:

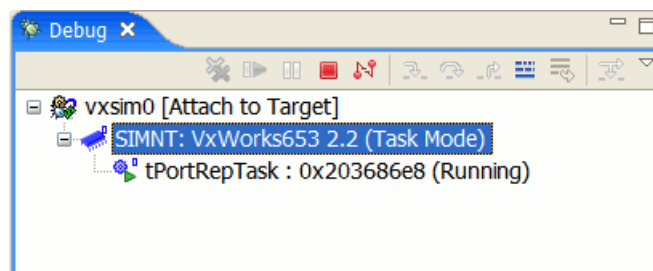


For an explanation of the **Target Manager** display for VxWorks 653 systems, see [Understanding the Target Manager Display](#), p.36. For more information on the Target Manager, see the *User Interface Reference*.

Step 3: Attach a kernel task (task mode)

You are now ready to debug a kernel task. Workbench allows you to debug an existing kernel task or to spawn and debug a new kernel task. In this step you will debug an existing kernel task. In the next step you will spawn and debug a new kernel task.

1. In order to debug an existing task, you must attach the debugger to the running task. To do this, expand the **Kernel Tasks** item in the **Target Manager** and select **tPortRepTask**.
2. Select **Target > Attach to Kernel Task**. The debugger attaches to the task and the task is displayed in the **Debug** view.



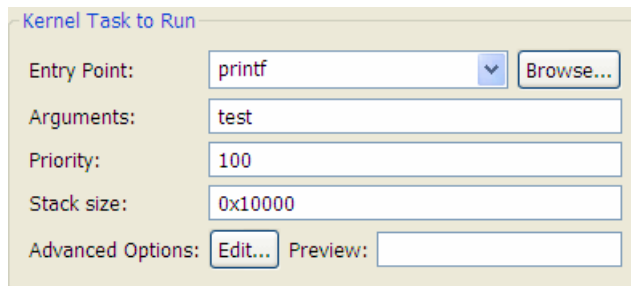
3. Select **tPortRepTask** in the debug view.

4. Click the **Suspend** button on the **Debug** view toolbar. The task is stopped. Its stack trace is displayed in the **Debug** view, and the **Assembly** view opens showing the current execution point. (Source code view for kernel tasks is not available unless you have access to the kernel source and have compiled the kernel with debug symbols.)
5. Click the **Resume** button on the **Debug** view toolbar.
6. Click the **Disconnect** button on the **Debug** view toolbar. The debugger connection to the task is terminated
7. Click the **Remove All Terminated** button in the **Debug** view. This clears the **Debug** view.

Step 4: Spawn and attach a kernel task (task mode)

To spawn and debug a kernel task:

1. In the **Target Manager**, select the **Kernel Tasks** item.
2. Select **Target > Debug > Debug Kernel Task**. The **Debug** dialog is displayed.
3. On the **Main** tab, enter “printf” in the **Entry Point** field and “test” in the **Arguments** field (without the quotes).



Kernel Task to Run

Entry Point: printf

Arguments: test

Priority: 100

Stack size: 0x10000

Advanced Options: Preview:

4. In the **Debug Options** tab, make sure that **Break on Entry** and **Automatically attach spawned Kernel Tasks** are checked.
5. Click **Debug**. The task is added to the **Debug** view and the assembly code is shown in the **Assembly** view.

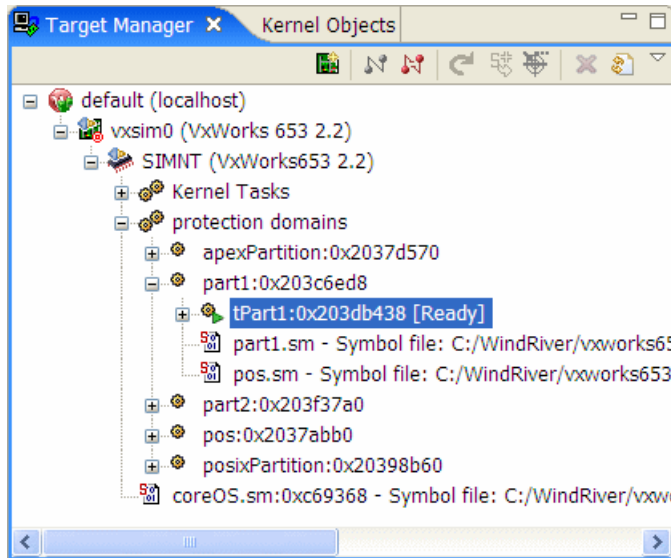
Step 5: Step through the task

1. Use the step controls on the **Debug** view toolbar to step through the code. Observe that the recently executed lines are shaded in progressively lighter tones as you step.
2. Click the **Resume** button on the **Debug** view toolbar. The string “test” is printed in the simulator console and the task exits. The debugger connection to the task is terminated.
3. Click the **Remove All Terminated** button in the **Debug** view. This clears the **Debug** view.

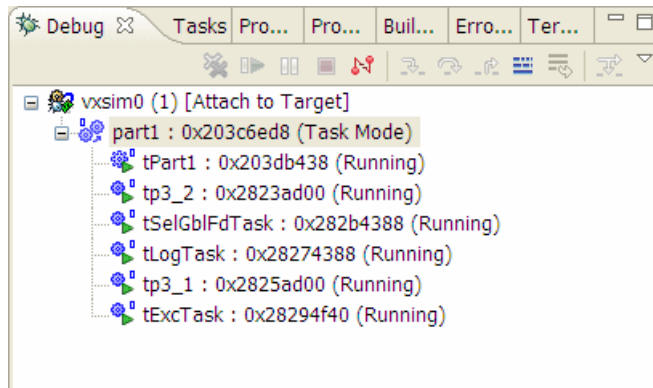
Step 6: Attach to a partition task

You can attach the debugger to a partition task.

1. In the **Target Manager**, collapse the list of kernel tasks.
2. Expand the **protection domains** item. The list of protection domains is displayed. Protection domains include partitions, shared libraries, and other domains.
3. Expand the **part1** partition item. The partition task, **tPart1** is shown. Select **tPart1**. (If **tPart1** is not listed, right click **part1** and select **Refresh**. **tPart1** should appear.)



4. Select **Target > Attach to protection domain task**. The task is displayed in the **Debug** view.



Note that the **tPart1** task is shown at the same level as the threads running within the task. It is the first item in the list and is distinguished by a different icon.

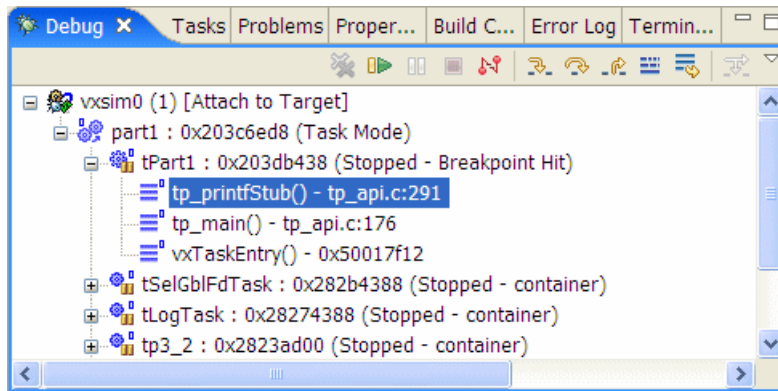
Step 7: Open a source code file and set a breakpoint

You can now open the source code file for this task and associate it with the debug session. Depending on how your project is configured, Workbench may be able to associate the source code files with the project automatically. However, following the procedure below assures that the association is made.

1. Select **File > Open File** and open the file `installDir/vxworks653-2.2/target/proj/BSP_simple653Module/part1/tp_api.c`. The file is displayed in the **Editor** view.
2. Locate the **tp_printfStub** function in the source code file.
3. Locate the first executable line of the function (the first **if** statement).
4. Right-click in the gutter next to that line (not on the line itself, but in the gutter) and select **Breakpoints > Add breakpoint**.
5. Select the **Source Lookup** tab.
6. In the **Source Lookup** tab, click the **Edit Source Lookup** button. The **Edit Source Lookup Dialog** is displayed.
7. Click **Add**. The **Add Source** dialog is displayed.
8. Select **File System Directory** and click **OK**. The **Directory Selection** dialog is displayed.
9. Select `installDir/vxworks653-2.2/target/proj/simple653Module/part1` and click **OK**.
10. Click **OK** to close the **Edit Source Lookup Path** dialog.
11. Click **OK** to close the **Line Breakpoint Properties** dialog. The breakpoint is set.
12. Wait for the system to hit the breakpoint. This should happen momentarily.

Step 8: Step through the function

You can now step through the function using the step controls on the **Debug** view toolbar. (For more information on the step controls, press the help key).



1. Experiment with the step controls.
2. When you have finished experimenting, remove the breakpoint by selecting **Run > Remove All Breakpoints**.
3. Resume the thread by clicking the **Resume** button on the **Debug** view toolbar.
4. Close the **tp_api.c** file in the **Editor** view.

Step 9: Set an expression breakpoint

1. Ensure that **tPart1** is selected in the **Debug** view.
1. Select **Run > Breakpoints > Add Expression Breakpoint**. The **Expression Breakpoint Properties** dialog is displayed.
2. In the **Location Expression** field of the **General** tab, enter the function name "tp_printfStub"
3. Select the **Scope** tab.
4. Ensure that **Enable restricting of breakpoint scope** is checked and that **Enable scoping to debug targets and threads** is not checked.
5. Ensure the **Show only active launch configurations** is checked. The list of launch configurations will be restricted to those active in the debugger.
6. Expand the **part1** partition and ensure that the **tPart1** partition task is selected.
7. Click **OK**. The breakpoint is set.
8. Wait for the system to hit the breakpoint. This should happen momentarily. When the breakpoint is hit, Workbench will open the source file and display

the line where the breakpoint was hit. (If Workbench cannot locate the file, it will prompt you to specify its location.)

9. Remove the breakpoint, resume the partition, and close the source file.

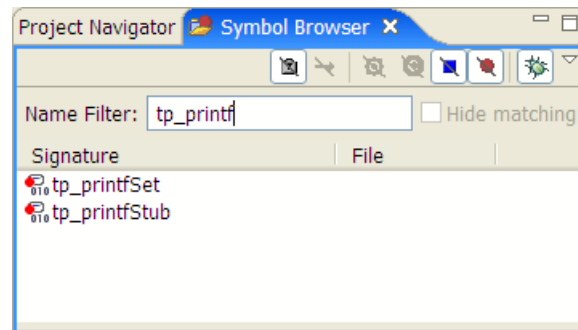
Step 10: Use the symbol browser

You can use the symbol browser to navigate the symbols in the tasks that are attached to the debugger.

1. Locate the **Symbol Browser** view. If it is hidden behind another view, click its tab to bring it forward. If it is not visible, select it from **Window > Show View**.
2. Click the **Show Symbols Loaded by the debugger** button on the **Symbol Browser** tool bar. (It is the one at the right with a bug on it.) The view is populated with a long list of symbols.
3. In the **Name Filter** field of the **Symbol Browser** toolbar, enter:

tp_printfStub

The list of symbols is restricted to those whose names start with **tp_printfStub**.



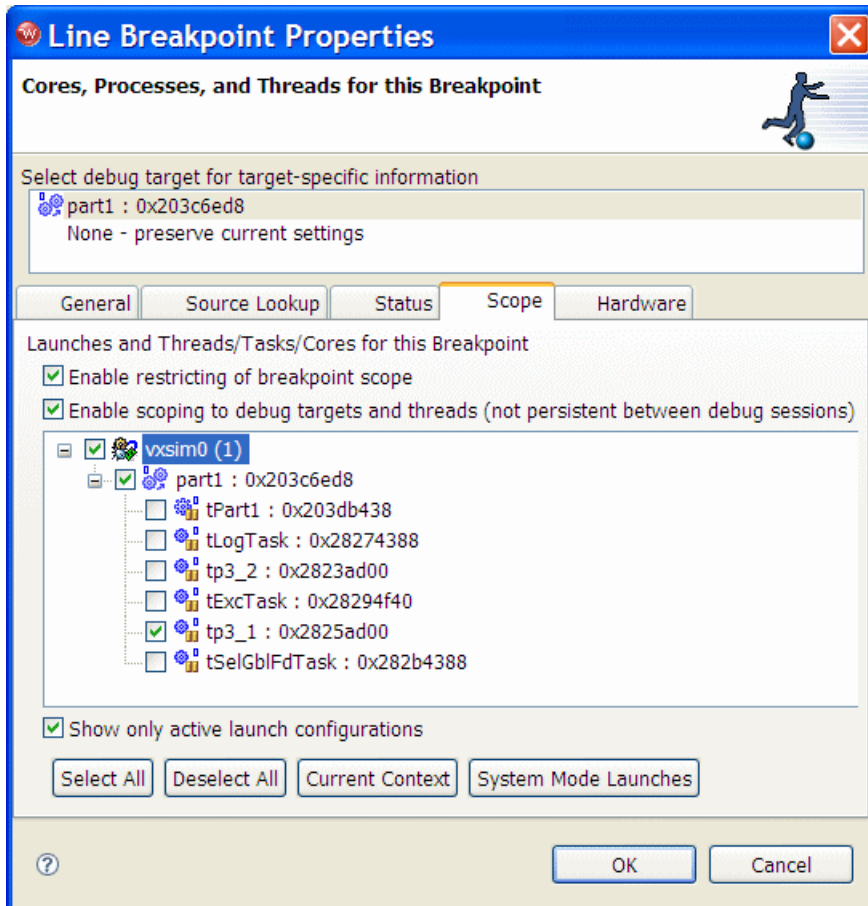
4. Double-click on the **tp_printfStub** symbol. Workbench opens the source file in the **Edit** view and displays the function definition for **tp_printfStub**.

Step 11: Set a breakpoint in a thread

While it is not possible to step through a thread, it is possible to set a breakpoint in a thread. (This means that the breakpoint will be hit only when the code is executing in the thread specified, not when that code is executed by another thread.)

1. Locate the **tp_printfStub** function in the source code file.

2. Locate the first executable line of the function (the first **if** statement).
3. Right click in the gutter next to that line (not on the line itself, but in the gutter) and select **Breakpoints > Add Breakpoint**.
4. Select the **Source Lookup** tab and confirm that the source lookup settings are correct.
5. Select the **Scope** tab.
6. Check **Enable scoping to debug targets and threads**. This allows you to specify which threads the breakpoint applies to.
7. Check thread **tp3_1** and clear all the other threads.



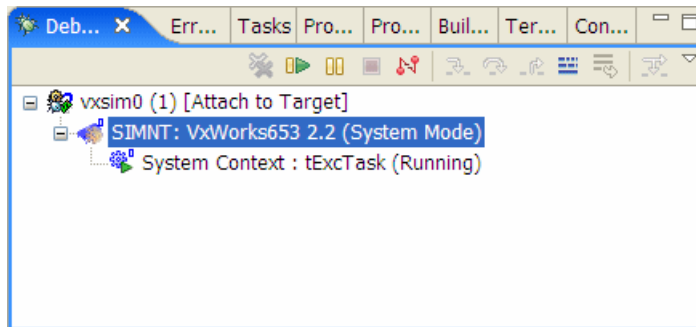
8. Click **OK**. The breakpoint is set.
9. Wait for the system to hit the breakpoint. This should happen momentarily.
10. Click the **Resume** button on the **Debug** view toolbar. You cannot use the step controls because stepping through a thread is not supported.
11. Click the **Disconnect** button on the **Debug** view toolbar. The debugger connection to the task is terminated.
12. Click the **Remove All Terminated** button on the **Debug** view toolbar. This clears the **Debug** view.

A

Step 12: Debug the kernel in system mode

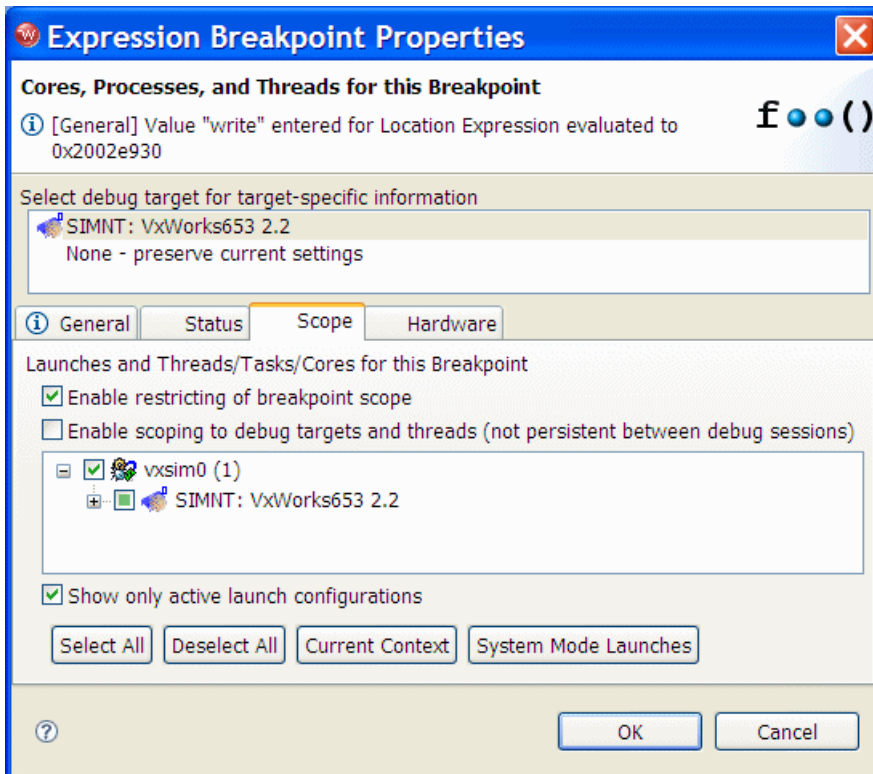
In this step you will debug the kernel in system mode. In system mode, you can attach to a system task or the whole system, but when you step, you step the whole system, not the individual task.

1. Switch the debugger to system mode by selecting the system item (the item labeled **SIMNT (VxWorks653 2.2)** in the **Target Manager** and selecting **Target > Target Mode > System**.
2. Attach the debugger to the kernel by selecting the system item in the **Target Manager** and selecting **Target > Attach to Kernel (System Mode)**. The debugger attaches to the task and the task is displayed in the **Debug** view. The **Debug** view displays the system:

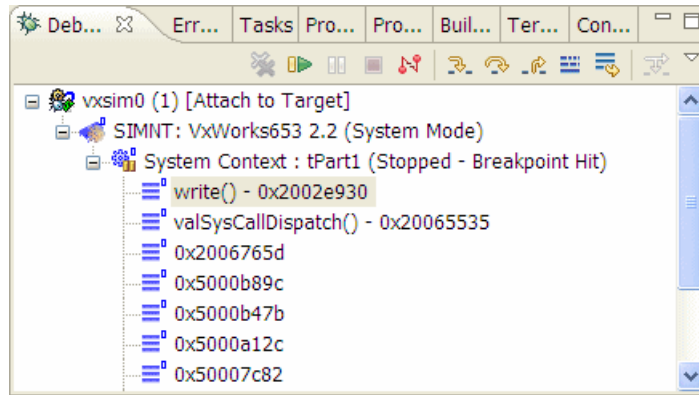


3. You will now set a breakpoint on the **write** routine in the kernel. Select **Run > Breakpoints > Add Expression Breakpoint**. The **Expression Breakpoint Properties** dialog is displayed.

4. In the **Location Expression** field of the **General** tab, enter "write" (without the quotes). The information panel of the dialog will show the location of the symbol.
5. On the **Scope** tab of the dialog, ensure that **Enable restricting of breakpoint scope** is checked and that the system you are debugging is checked.



6. Click **OK**. The breakpoint is set.
7. Wait for the breakpoint to be hit. This should happen momentarily. The **Debug** view shows the system stopped and displays a stack trace.



8. You can now experiment with stepping through code and viewing values.
9. When you have finished experimenting, you can close the connection to the debugger. If the system is stopped, remove the breakpoint and click the **Resume** button on the **Debug** view toolbar.
10. Click the **Disconnect** button on the **Debug** view toolbar. The debugger connection to the task is terminated.
11. Click the **Remove All Terminated** button on the **Debug** view toolbar. This clears the **Debug** view.

Step 13: Debugging a partition in system mode

You can debug a partition in system mode.

1. In the **Target Manager**, select **SIMNT (vxworks653 2.2)**.
2. Select **Target > Attach to kernel (System Mode)**. The **Debug** view will show the system.
3. In the **Target Manager**, select the **tPart1** item.
4. Select **Target > Attach to Protection domain task (System Mode)**. If the menu shows **Attach to Protection domain task (Task Mode)**, then select **Target > Target Mode > System**, then select **Target > Attach to Protection domain task (System Mode)**. The **Debug** view will show the partition.

5. You can now set a breakpoint in the partition. Select **Run > Breakpoints > Add Expression Breakpoint**. The **Expression Breakpoint Properties** dialog is displayed.
6. In the **Location Expression** field of the **General** tab, enter the expression "tp_printfStub" (without the quotes). The information panel of the dialog will show the location of the symbol.
7. On the Scope tab of the dialog, ensure that **Enable restricting of breakpoint scope** is checked.
8. Ensure that **tPart1** is part of the selected scope (the scope cannot be limited to **tPart1** alone).
9. Click **OK**. The breakpoint is set.
10. Wait for the breakpoint to be hit. This should happen momentarily. The **Debug** view will show that the system is stopped and will display a stack trace.
11. You can now step the system. In system mode you can only step the system. You cannot step **tPart1**. Experiment with stepping and viewing variables.
12. When you are finished, resume the system and disconnect the debugger.

B

Glossary

acceptance

Acceptance is the acknowledgement by a certification authority that the ARINC 653 module, application, or system meets its defined requirements.

ACE

ACE: Agent for the Certified Environment.

AFDX

AFDX: Avionics Full Duplex Switched Ethernet. It is defined by the ARINC 664 specification, Part 7.

alarm

In the context of health monitoring, an alarm is an event. See also message.

AMIO

Applications multiplexed I/O (AMIO) allows you to provide input to and view output from multiple partitions over a single serial connection.

APEX

APEX: Application/Executive. The general-purpose interface between an OS and application software, specified by the ARINC 653 specification. The specification includes the list of services that lets the application control scheduling, communication, and status information of its internal processing elements.

APEX port

APEX port: see port.

API

API: application programming interface.

application

An application is a collection of software components that together perform a specific function in an embedded system. See also application partition.

application developer

An application developer develops one or more applications that will reside in a partition. This person or group may also be responsible for developing data binaries, which contain any databases used by the application. See also platform provider and system integrator.

application partition

An application partition is a partition that includes an application.

APPS

APPS: ARINC PPS. It is the module-wide scheduling scheme for partitions. This is a combination of ARINC 653 scheduling (TPS) and PPS scheduling in which the PPS scheme is used during idle time within the TPS scheme. The scheduling scheme applies to all PPS-enabled partitions in the module.

ARINC 653

ARINC 653 refers to ARINC Specification 653: the “Avionics Application Software Standard Interface.”

ARINC 653 scheduling

ARINC 653 scheduling is the scheduling that is specified by the ARINC 653 specification. It is time-preemptive scheduling (TPS). See also APPS scheduling and PPS scheduling.

ARINC PPS

ARINC PPS: see APPS scheduling.

black box

A black box is a set of configuration parameters that represent the memory requirements of an application, a shared library, or the core OS. The use of black boxes allows a VxWorks 653 module to be configured before all the applications and libraries are available. Applications, libraries, and the core OS must fit within the memory limits set by their black boxes.

board support package

BSP: board support package. It provides the libraries required to support a platform on a particular board. The BSP, along with the kernel and user-supplied extensions, makes up the core OS.

BSP

BSP: see board support package.

BSP developer

A BSP developer is a person or organization responsible for the development of a board support package.

BSS

BSS: block started by symbol. It is a data section in an ELF file that contains uninitialized global and static variables that are zeroed.

build spec

A build spec specifies compiler and linker options to produce particular output, such as cert, debug, and release.

callback routine

In the context of health monitoring, a callback routine is called when an event arrives at a partition health monitor task or module health monitor task. It is called before the handler for the given event is called.

CDF

CDF: component description file. It has the **.cdf** extension. It uses the component description language (CDL) to name and give values to the parameters of VxWorks 653 components.

cert

cert is the build spec that produces a certifiable image.

certifiable

An image that is certifiable can be certified to a specific level of the DO-178B avionics software standard.

certifiable subset

A certifiable subset is a subset of the core OS or a partition OS that can be certifiable to Level A of the DO-178B avionics software standard.

certification

Certification refers to certification to a specific level of the DO-178B avionics software standard.

channel

A channel defines a logical link between one source port and one or more destination ports. It also defines the message transfer mode and the characteristics of the messages. Channels are used for inter-partition communication, which can be between local partitions and/or pseudo-partitions. Channels conform to the ARINC 653 specification.

COIL

COIL: core OS interface library. A partition OS that provides a library of routines independent of the vThreads partition OS. The library supports the management of interrupts and exceptions, device I/O, interpartition messaging, and injection of health monitoring events.

COIL partition

A COIL partition is a partition whose partition OS is based on COIL. See also vThreads partition.

cold restart

A cold restart occurs when a module or partition is restarted and all data is reloaded. A cold restart takes longer than a warm restart.

configlette

A configlette is a component or part of a component that is distributed in source form, allowing compile time parameters to be set when the component is included in a build.

configuration parameter

A configuration parameter is used to change the configuration of VxWorks 653 component.

configuration record

A configuration record is a record of the information that makes up the configuration of a VxWorks 653 module or a part of it. Configuration records include both the system configuration record and user configuration records.

core OS

The core OS is the core operating system for a VxWorks 653 module. It provides fundamental operating system services and schedules partitions.

core OS interface library

Core OS interface library: see COIL.

CPU page size

The CPU page size is the smallest addressable unit of memory for the MMU. It is also called MMU page size. The page size depends on the CPU and is generally not configurable.

cross-development tools

Cross-development tools are programs that run on a host computer (running, for example, Windows or UNIX) and that are used to develop, debug, or control software running on an embedded processor, which is running a real-time operating system (for example, VxWorks 653). For VxWorks 653, the cross-development tools are based on Workbench. See also run-time software.

current partition

The current partition is the partition that is running. In an APPS scheduling environment, the current partition and the TPS partition may not be the same.

default schedule

The schedule that will be run when the module is booted.

destination port

A destination port is one of possibly many ports at the receiving end of a channel. See also source port.

direct-access port

A direct-access port is a type of pseudo-port which does not use software buffering. Buffering support is assumed to be provided by the communications hardware.

DO-178B

DO-178B: "Software Considerations in Airborne Systems and Equipment Certification." The avionics software standard developed by RTCA.

domain

A domain is a software container. Each element of a VxWorks 653 module—the core OS (kernel), partitions (applications), shared libraries, system shared libraries, and shared data regions—exists in a domain.

dynamic memory allocation

Dynamic memory allocation refers to allocating memory from the heap at runtime.

EABI

EABI: Embedded Application Binary Interface.

ELF

ELF: Executable and Linking Format. It is an object module format used to encapsulate compiled software.

error handler process

See process health monitor.

event

In the context of health monitoring, an event is the base unit that is injected into the event handling framework. It could represent an alarm or a message, depending on the event code.

event code number

In the context of health monitoring, an event code number is the value of the event code, as defined in the **HM_CODE** enumeration type in **hmTypes.h**.

event queue

The module health monitor table and partition health monitor table each have an event queue. The module and partition health monitor event queues are sometimes called, simply, the module and partition health monitor queues. An event queue holds the events that have been dispatched to its associated health monitor for handling. Event queues are serviced before health monitor notification queues are serviced.

FAA

FAA: U.S. Federal Aviation Administration.

FIFO

FIFO: first-in, first-out queuing.

global file descriptor

Global file descriptors (standard in, standard out, and standard error) are available to all tasks in a partition. Their global assignment is controlled by the **ioGlobalStdSet()** and **ioGlobalStdGet()** routines, but may be overridden by the **ioTaskStdSet()** and **ioTaskStdGet()** routines.

GUI

GUI: graphical user interface.

health monitor

Health monitoring provides a framework to raise and handle events (which can be alarms or messages) in a VxWorks 653 module. Alarms are injected to represent faults, and handlers provide the opportunity to perform recovery actions. See module health monitor, partition health monitor, process health monitor, and system health monitor.

hosted function supplier

Hosted function supplier: see application developer.

IDE

IDE: integrated development environment.

injection

Injection is the act of creating a health monitor alarm event or message event.

interface subset

An interface subset defines part of the interface of a shared library. The use of interface subsets allows you to reuse parts of the interface definition among libraries that share some parts of their interface. For example, two different **vThreads** libraries containing different components would share the core **vThreads** interface.

interrupt level

Saying an event is injected at an interrupt level means the event is injected from an interrupt execution context.

ISR

ISR: interrupt service routine.

jitter

Jitter is a variation or deviation in the frequency of an expected occurrence. See also partition switch jitter.

kernel

Kernel is another term for the core OS.

kernel I/O region

A kernel I/O region is a region of target memory that corresponds to the address of an I/O device on the target and can be accessed only by the core OS.

Level A

Level A is the highest certification level for the DO-178B software standard.

loadable shared data region

A loadable shared data region is a data source, such as a database, that can be loaded into a shared data region as part of the module payload.

local partition

A local partition is a partition that is local to a VxWorks 653 module. Unless it might be confused with a pseudo-partition, it is called, simply, a partition.

local port

A local port is a port that is attached to a local partition. Unless it might be confused with a pseudo-port, it is called, simply, a port. See also null port.

log queue

The module health monitor and partition health monitor each have a log queue (sometimes called simply a log). Health monitor messages are always logged, whereas alarms are logged only if health monitor logging is enabled. If an event is injected from within a partition (**HM_PROCESS_MODE** or **HM_PARTITION_MODE**), the event is logged to the partition health monitor log. If the event is injected from outside the partition (**HM_MODULE_MODE**), the event is logged to the module health monitor log.

major frame

Each schedule consists of a major frame, which is divided into a series of variable-length minor frames.

message

In the context of health monitoring, a message is an event. See also alarm.

minor frame

Each schedule consists of a major frame, which is divided into a series of variable-length minor frames. Each minor frame defines the partition to run, its allowed duration, and whether or not the minor frame is a release point.

MMU

MMU: memory management unit.

module

A module is the “system” controlled by one RTOS, and in VxWorks 653, that RTOS is the core OS.

module health monitor

The module health monitor is present in parallel with all partitions in a VxWorks 653 module, and hence all partition health monitors in the module. The module health monitor is not part of any partition window and has priority over all partitions. The module health monitor resides in the core OS. It is associated with the module health monitor table, which among other things, defines notification queues, a log queue, and an event queue. See also system health monitor, partition health monitor, and process health monitor.

namespace

An XML namespace provides a unique identifier which can be associated with an XML element by means of a prefix. The namespace uniquely identifies the XML schema in which the element is defined.

NMI

NMI: non-maskable interrupt.

normal mode

Normal mode is the partition mode during which processes/threads are scheduled. (Other partition modes include idle, cold start, and warm start.)

notification

In the health monitoring context, notification is the act of informing another partition health monitor or the module health monitor of an event that has occurred in a given partition.

notification queue

The module health monitor table and partition health monitor table each have notification queues, one for each partition that wants to accept notification of events. Notification queues are serviced after health monitor event queues are serviced.

null port

A null port is a port that is created at system initialization time, but is not used. It is always considered to be empty when read from and have space when written to. A null port can be attached to a partition or the core OS of a VxWorks 653 module or to a pseudo-partition. See also local port and pseudo-port.

NVM

NVM: non-volatile memory.

online-loaded partition

With online-loaded partitions, the core OS does not install the partition code from flash or RAM into its final domain location in RAM as it does during the system initialization phase for regular partitions. Instead, an empty application domain is created for an online-loaded partition during the core OS initialization phase. The code of the online-loaded partition is made available to the core OS only at a later stage. In some cases this may not be until after all the regular partitions are already running.

OS

OS: operating system.

partition

A partition is a container for an application. An application running in a partition cannot interfere with applications in other partitions or with the core OS.

partition direct-access port

A partition direct-access port is a type of direct-access port residing in a partition. A partition direct access port can communicate only with a local port in the application resident in the partition.

partition health monitor

The partition health monitor is the health monitor that is present in parallel with vThreads to handle vThreads partition errors and events that may affect the operation of vThreads within the partition. The partition health monitor is scheduled as part of the partition window. It is associated with the partition health monitor table, which among other things, defines notification queues, a log queue, and an event queue. See also system health monitor, module health monitor, and process health monitor.

partition OS

A partition OS is a user-level software library running within a partition that provides operating system services to the partition. See also vThreads and COIL.

partition OS scheduler

The partition OS scheduler is the scheduler in a partition OS that allocates CPU time to threads in the partition. The partition OS scheduler in a vThreads partition is a priority-preemptive scheduler and is not related to the ARINC schedule.

partition port

Partition port: see local port.

partition scheduler

The partition scheduler is the scheduler in the core OS that allocates CPU time to partitions, allowing CPU time to become available to threads in those partitions. By default, the partition scheduler uses ARINC 653 (TPS) scheduling, but can optionally schedule designated partitions with APPS scheduling. See also partition OS scheduler.

partition switch jitter

Partition switch jitter is a variation or deviation in the configured partition switching schedule. For example, partition switch jitter might be caused by hardware latencies or when the core OS locks interrupts.

partition window

A partition window is the time in which a partition is allowed to run before being scheduled out.

payload

A payload is an image file (or files) that contains the code for a VxWorks 653 module in a form that is suitable for running on a target.

payload region

A payload region is the region of RAM or ROM where a payload is loaded.

periodic process

A periodic process is a process within a partition that is run on a schedule based on the passage of wall clock time (that is, the countdown to the next invocation of periodic process runs even when the partition itself is not scheduled).

PersistentBSS

A BSS section that is persistent across a warm restart.

platform

A platform is software on which applications can be built and from which a VxWorks 653 module can be developed.

platform provider

A platform provider is responsible for configuring the base system on which application developers will build their applications.

port

A port is one end of a channel, which is used for inter-partition communication. Ports have attributes, for example, direction (source or destination), mode (queuing or sampling), protocol (receiver discard, sender block, or none), and refresh rate. Ports conform to the ARINC 653 specification and its APEX interface and are also called APEX ports. See also pseudo-port.

POS

POS: See Partition Operating System.

POSIX

POSIX: Portable Operating Systems Interface. In this documentation, POSIX refers to the standard for real-time extensions (1003.1b), which specifies a set of interfaces to OS facilities. The POSIX API can be included in a vThreads partition if the APEX API is not included.

PPS

PPS: priority-preemptive scheduling. It allows for scheduling of partitions in a module-wide priority-preemptive scheme during the idle time within an ARINC 653 (TPS) schedule. See also APPS scheduling.

PPS-enabled

A PPS-enabled partition is a partition that is configured to indicate that it should be considered during APPS scheduling.

preemption locking

Preemption locking disables the scheduling of processes/threads/tasks, and only the current process/thread/task can be run until it decrements the lock level back to zero.

priority-preemptive scheduling

Priority-preemptive scheduling: see PPS.

process

Process is the APEX term for a thread. In the vThreads context, the term thread is preferred. See also task.

process health monitor

The process health monitor is the health monitor that is present within vThreads to handle process-related errors and events. It is also known as the error handler process. See also system health monitor, module health monitor, and partition health monitor.

pseudo-partition

A pseudo-partition is a communications object that is outside a VxWorks 653 module. See also local partition and pseudo-port.

pseudo-port

The term pseudo-port applies generally to any port that represents a data source or destination outside the current module. The term pseudo-port is also used in a more restrictive sense for a type of pseudo-port that uses software buffering. In this sense it is contrasted with direct-access port which is a type of pseudo-port that does not use software buffering. See also local port and null port.

queuing port

A queuing port is a port in queuing mode. In queuing ports, messages are queued. A protocol is required to manage the queues. See also sampling port.

RAM

RAM: random access memory.

RAM payload

A RAM payload is a payload that is designed to be downloaded into RAM on the target.

real-world time

Real-world time: see wall clock time.

receiver discard protocol

Receiver discard protocol is a port message protocol. If one of the channel's destination ports is full, the source port discards the message for that port. Therefore, if all the destination ports are full, the message might be lost. When a message is so discarded, the port's overflow flag is set to notify the application of the discarded (lost) message. See also sender block protocol.

refresh rate

The refresh rate (in seconds) indicates the maximum acceptable age of a valid message, from the time it was received by the port. It applies to destination sampling ports only.

release point

A release point is a way to synchronize a periodic process with the partition window of a partition. A periodic process spawned in a partition will be started only at the next release point.

ROM

ROM: read-only memory.

ROM payload

A ROM payload is a payload that is designed to be installed in ROM on the target.

root element

The root element is the element of an XML document that contains all the other elements in the document.

RTCA

RTCA: Radio Technical Commission for Aeronautics. The private, not-for-profit corporation that develops recommendations on communications, navigation, surveillance, and air-traffic management issues. RTCA developed the DO-178B avionics software standard.

RTOS

RTOS: real-time operating system.

run-time software

Run-time software is the operating system and application software that together run on a target. See also cross-development tools.

sampling port

A sampling port is a port in sampling mode. In sampling ports, messages are not queued. A message remains in the source port until it is sent or overwritten. Each new message overwrites the previous one when it reaches the destination port and remains there until it is overwritten itself. Sampling ports have refresh rates. See also queuing port.

SAP port

A service access point (SAP) is a special kind of queuing port. It is different from a normal queuing port because it allows access to addressing information when sending and receiving messages. The SAP services are similar to the ARINC 653 queuing port services but will have additional parameters to support address information.

schedule

Schedules define how the core OS schedules partitions. Each schedule consists of a major frame.

scheduler

See partition scheduler and partition OS scheduler.

select operation

The select operation refers to calling **select()** to pend on a set of file descriptors.

sender block protocol

Sender block protocol is a port message protocol. A queuing message is sent to all the channel's destination ports. If any one is full, the message is queued in the source port in FIFO order. When the source port is full and if a timeout was specified, sender processes are blocked during the **SEND_QUEUING_MESSAGE** service. When a destination port is emptied, retransmission is attempted. Whether it succeeds depends on the state of the channel's other destination ports. See also receiver discard protocol.

service access point

Service access point: see SAP port.

shared data region

A shared data region (sometimes called a shared data domain) is a data region that can be used by applications within partitions to share data. Outside a shared data region, applications have no access to the data of other applications. See also loadable shared data region.

shared I/O region

A shared I/O region is a region of target memory that corresponds to the address of an I/O device on the target and can be shared by partitions and the core OS.

shared library

A shared library is a library that contains code that can be shared by multiple applications. See also system shared library.

shared library region

A shared library region is the area of RAM that holds a shared library.

source port

A source port is the one port at the sending end of a channel. See also destination port.

standard port

Standard port: see local port.

static module

A static module file is a fully located object file that has been compiled and linked for use in a VxWorks 653 module. A static module file has a .sm file extension.

straight-line code

Straight-line code is code that does not use threads.

system call

A system call is a call from a partition to the core OS.

system clock

System clock refers to the system clock for a VxWorks 653 module.

system configuration record

The system configuration record is the record of all the configuration parameters in a VxWorks 653 module. During the configuration process, configuration information is expressed in the **Module** configuration document. The build process produces a binary version of this information in **configRecord.reloc** or **configRecord.bin**.

system health monitor

The system health monitor is the dispatcher for the health monitoring system. See also module health monitor, partition health monitor, and process health monitor.

system heap

System heap refers to the heap for the core OS.

system initialization

System initialization refers to the initialization of a VxWorks 653 module.

system integrator

A system integrator is responsible for integrating the applications created by the application developers with the platform created by the platform provider to create the final module.

system memory

System memory refers to memory controlled by the core OS.

system object

A system object is an object created by the core OS (or vThreads) for use by the core OS (or vThreads). An example is a semaphore.

system resource

A system resource is a resource allocated by the core OS for use by the core OS.

system restart

System restart refers to restarting a VxWorks 653 module.

system shared library

A system shared library is a special shared library that contains the code for a partition OS.

system start

System start refers to starting a VxWorks 653 module.

target

The target is the board for which you are developing an embedded system.

task

A task is an execution context. In VxWorks 653, it refers to a core OS object. See also thread.

TCB

TCB: task control block. The structure that contains critical runtime information for a single task.

thread

A thread is an execution context. It is the preferred term for what is sometimes called a process. A thread is a programming unit contained within a vThreads partition. It runs concurrently with other threads of the same partition. See also task and process.

time-preemptive scheduling

Time-preemptive scheduling: see TPS.

TLB

TLB: translation look-aside buffer. It is a specialized cache that holds a table of physical addresses as generated from the virtual addresses that program code uses.

TPS

TPS: time-preemptive scheduling. It is also called ARINC 653 scheduling. See also APPS scheduling and PPS scheduling.

TPS partition

A TPS partition is the partition that has been scheduled to be run by the ARINC 653 (TPS) scheduler. In an APPS scheduling environment, the current partition and the TPS partition may not be the same.

trusted partition

From the point of view of a given partition, a trusted partition is a partition from which it will allow the health monitor to accept health monitor notifications on its behalf. Since health monitor notifications are processed in the time slice of the partition on whose behalf they are received, limiting the number of partitions that a partition trusts limits the effect of health monitor notifications on the partition's time allotment.

user configuration record

A user configuration record is a collection of data that can be used for configuring user extensions to the core OS.

user memory region

The user memory region is that area of RAM that is needed for memory other than health monitor logs, core OS configuration records, core OS memory, core OS page pools, core pools, ports, and RAM payload.

user partition OS

A user partition OS is a partition OS that is based on COIL, augmented to perform other functions that are required by the application.

VAL

VAL: vThreads abstraction layer. It is a layer of the core OS. When a vThreads partition makes a system call, it communicates with this layer. It is a concept internal to VxWorks 653.

validation

In XML terms, validation is a process that ensures that an XML file is well formed according to the rules of XML and adheres to the structure specified in the appropriate XML schema. Validation is performed by an XML validator.

VME

VME: Versa Module Europa. VME is an open-ended bus system that makes use of the Eurocard standard. The VME bus was intended to be a flexible environment, supporting a variety of computing-intensive tasks, and has become a popular protocol in the computer industry. It is defined by the IEEE 1014-1987 standard.

vThreads

vThreads is the priority-preemptive OS that serves as a partition OS.

vThreads partition

A vThreads partition is a partition whose partition OS is based on vThreads. See also COIL partition.

vThreads scheduler

vThreads scheduler: see partition OS scheduler.

VxWorks 5.5

VxWorks 5.5 is the Wind River operating system on which the vThreads partition OS of VxWorks 653 is based.

VxWorks 653

VxWorks 653 is the Wind River operating system that supports the ARINC 653 specification.

W3C

W3C refers to the World Wide Web consortium at www.w3.org.

wall clock time

Wall clock time is time as measured in the real world by the clock on the wall. (As opposed, for instance, to the time elapsed in a particular application's partition window.)

warm restart

A warm restart occurs when a module or partition is restarted but persistent data is retained, shortening the time required for the restart.

WDB

WDB refers to the Wind River debug agent.

Wind

Wind is the adjective applied to certain OS objects to distinguishes them from POSIX objects. For example, Wind semaphores to distinguishes from POSIX semaphores.

WindSh

WindSh is a host shell.

Workbench

Workbench is the Wind River Workbench development environment.

worker task

A worker task is a core OS task that is associated with a specific partition. Worker tasks perform blocking operations (typically blocking I/O) on behalf of the partition they are associated with.

write-protect

To write-protect is to guard an entity by a mechanism that prevents it from being changed or erased. For example, memory can be write-protected by using an MMU.

XInclude

XInclude is a W3C standard for including one XML file in another.

XML

XML: Extensible Markup Language. It is a standard for defining markup languages.

XML attribute

An XML attribute is an additional piece of information added to an XML element in the form of a key/value pair.

XML declaration

The XML declaration identifies a file as an XML document and contains information such as the version of XML used and the character encoding used in the file.

XML document

A document written using XML syntax.

XML document type

An XML document type is the grammar of a particular XML file as defined by the applicable XML schema.

XML editor

An XML editor is a program that provides support for editing XML files. This usually includes support for inserting tags and for validating the file against an XML schema.

XML element

An XML document consists of XML elements, each of which may contain data content and/or other elements. The elements allowed in a particular document type is determined by the applicable XML schema.

XML file

An XML file is an instantiation of an XML schema.

XML schema

An XML schema is a document that defines the structure of an XML document. It defines what elements are permitted in an XML document, the order and nesting of elements, and the types of data each element can contain.

XML schema file

An XML schema file is a file that contains all or part of the definition of an XML schema. An XML schema file can include other schema files by reference to construct a complete schema definition.

XPath

XPath is a W3C standard for expressing the location of an element or attribute in an XML file.