

# VxWorks® 653

## CONFIGURATION AND BUILD GUIDE

### 2.2

---

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

<b>1</b>	<b>Overview .....</b>	<b>1</b>
1.1	Introduction .....	1
1.2	Quick Start .....	2
1.3	Organization of This Documentation .....	2
1.4	Conventions Used in This Documentation .....	3
<b>2</b>	<b>Understanding VxWorks 653 .....</b>	<b>5</b>
2.1	VxWorks 653 .....	5
2.2	Certification .....	7
<b>3</b>	<b>Configuration System .....</b>	<b>9</b>
3.1	Understanding VxWorks 653 Configuration .....	9
3.2	XML Configuration Files .....	13

<b>4</b>	<b>Build System .....</b>	<b>19</b>
4.1	Understanding the Build Process .....	19
4.2	Planning a Build .....	20
4.3	Building for VxWorks 653 .....	21
<b>5</b>	<b>Memory .....</b>	<b>23</b>
5.1	Understanding Memory .....	23
5.2	Planning Memory .....	27
5.3	Configuring Memory .....	28
<b>6</b>	<b>Core OS .....</b>	<b>33</b>
6.1	Understanding the Core OS .....	33
6.2	Planning the Core OS .....	34
6.3	Configuring the Core OS .....	36
6.4	Building the Core OS .....	41
<b>7</b>	<b>User Configuration Records .....</b>	<b>43</b>
7.1	Understanding User Configuration Records .....	43
7.2	Planning User Configuration Records .....	44
7.3	Configuring User Configuration Records .....	44
7.4	Building a User Configuration Record .....	46

<b>8</b>	<b>Partition OSs .....</b>	<b>47</b>
8.1	Understanding Partition OSs .....	47
8.2	Planning Partition OSs .....	48
8.3	Configuring a Partition OS .....	50
8.4	Building a Partition OS .....	55
<b>9</b>	<b>Shared Libraries .....</b>	<b>57</b>
9.1	Understanding Shared Libraries .....	57
9.2	Planning Shared Libraries .....	58
9.3	Configuring Shared Libraries .....	59
9.4	Building a Shared Library .....	65
<b>10</b>	<b>Shared Data Regions .....</b>	<b>67</b>
10.1	Understanding Shared Data Regions .....	67
10.2	Planning a Shared Data Region .....	68
10.3	Configuring a Shared Data Region .....	68
10.4	Building a Shared Data Region .....	71
<b>11</b>	<b>Shared I/O Regions .....</b>	<b>73</b>
11.1	Understanding Shared I/O Regions .....	73
11.2	Planning Shared I/O Regions .....	74
11.3	Configuring Shared I/O Regions .....	74
11.4	Building a Shared I/O Region .....	75

<b>12</b>	<b>ACE .....</b>	<b>77</b>
12.1	Understanding ACE .....	77
12.2	Planning ACE .....	78
12.3	Configuring ACE .....	78
12.4	Building ACE .....	82
<b>13</b>	<b>Platforms .....</b>	<b>85</b>
13.1	Understanding Platforms .....	85
13.2	Planning a Platform .....	86
13.3	Building a Platform .....	87
13.4	Packaging a Platform .....	87
<b>14</b>	<b>Applications .....</b>	<b>89</b>
14.1	Understanding Applications .....	89
14.2	Planning Applications .....	90
14.3	Configuring Applications .....	92
14.4	Building an Application .....	97
14.5	Packaging an Application .....	98
<b>15</b>	<b>Partitions .....</b>	<b>101</b>
15.1	Understanding Partitions .....	101
15.2	Planning Partitions .....	102
15.3	Configuring Partitions .....	104
15.4	Building Partitions .....	104

15.5	Configuring a Module for Online-Loaded Partitions .....	105
15.6	Building an Online-Loaded Partition .....	106
<b>16</b>	<b>Ports and Channels .....</b>	<b>107</b>
16.1	Understanding Ports and Channels .....	107
16.2	Planning Ports and Channels .....	111
16.3	Configuring Ports and Channels .....	111
16.4	Building Ports and Channels .....	112
<b>17</b>	<b>Schedules .....</b>	<b>113</b>
17.1	Understanding Schedules .....	113
17.2	Planning Schedules .....	114
17.3	Configuring Schedules .....	116
17.4	Building Schedules .....	116
<b>18</b>	<b>Health Monitor .....</b>	<b>119</b>
18.1	Understanding the Health Monitor .....	119
18.2	Planning Health Monitoring .....	123
18.3	Configuring the Health Monitor .....	126
<b>19</b>	<b>Modules .....</b>	<b>129</b>
19.1	Understanding Modules .....	129
19.2	Planning Modules .....	130
19.3	Configuring a Module .....	132
19.4	Building a Module .....	134

<b>20</b>	<b>Configuration Record .....</b>	<b>135</b>
20.1	Understanding the Configuration Record .....	135
20.2	Planning the Configuration Record .....	136
20.3	Configuring the Configuration Record .....	137
20.4	Building the Configuration Record .....	139
<b>21</b>	<b>System Images .....</b>	<b>141</b>
21.1	Understanding System Images .....	141
21.2	Planning a System Image .....	144
21.3	Configuring a Network-Loadable System Image .....	144
21.4	Configuring a RAM Payload System Image .....	147
21.5	Configuring a ROM Payload System Image .....	149
21.6	Building a System Image .....	152
<b>22</b>	<b>Reference Process .....</b>	<b>155</b>
22.1	Introduction .....	155
22.2	Quick Start .....	158
22.3	Hello World .....	159
22.3.1	Hello World cert .....	176
22.4	Module OS .....	179
22.4.1	Building and Exporting a Basic Module .....	180
22.4.2	Module OS Cert Build .....	180
22.4.3	Module OS with ACE .....	180
22.4.4	Module OS with Binary Components .....	183
22.4.5	Module OS with Source Components .....	184



<b>22.5</b>	<b>Partition OS .....</b>	<b>185</b>
22.5.1	Building and Exporting a Basic Module .....	186
22.5.2	Partition OS with Binary Components .....	186
22.5.3	Partition OS with Source Components .....	188
22.5.4	Partition OS with Shared Data Region .....	189
<b>22.6</b>	<b>Application .....</b>	<b>191</b>
22.6.1	Building and Exporting a Basic Module .....	192
22.6.2	Application in C++ .....	192
22.6.3	Two Applications .....	194
<b>22.7</b>	<b>Shared Library .....</b>	<b>196</b>
22.7.1	Building and Exporting a Basic Module .....	197
22.7.2	Hello from the Shared Library .....	197
22.7.3	Shared Library Versioning .....	201
<b>22.8</b>	<b>Integration .....</b>	<b>203</b>
22.8.1	Building and Exporting a Basic Module .....	204
22.8.2	Network-Loadable System Image .....	204
22.8.3	RAM Payload System Image .....	205
22.8.4	ROM Payload System Image .....	206
<b>A</b>	<b>Glossary .....</b>	<b>209</b>



# 1

## Overview

- 1.1 Introduction 1
- 1.2 Quick Start 2
- 1.3 Organization of This Documentation 2
- 1.4 Conventions Used in This Documentation 3

### 1.1 Introduction

This documentation describes the configuration and build process for VxWorks 653. The configuration and build process is designed to be flexible and to meet the needs of diverse organizations. In particular, it is designed to support distributed development of systems that comply with the ARINC 653 *Avionics Application Software Standard Interface*, Supplement 2, Part 1, and to make it as easy as possible for organizations to develop configuration and build systems that enable them to manage distributed development and certification of ARINC 653 modules.

The configuration and build of VxWorks 653 systems involves a collection of XML documents, source code files, binary components, and command-line tools. The system offers flexibility in the use of these elements, and allows significant customization of the configuration and build process.

To illustrate the possibilities offered by the configuration and build process, VxWorks 653 includes a reference process that contains a number of different use

cases that illustrate the configuration and build of the major components of an ARINC 653 module and the most commonly used optional features. You may develop your own configuration and build system either by adapting one of the use cases in the reference process to your needs, or by creating the required files from scratch.

## 1.2 Quick Start

To get started on building a project right away, see the quick start in the reference process ([22.2 Quick Start](#), p.158).

## 1.3 Organization of This Documentation

This document is organized as follows:

- Chapter [1](#) (this chapter) provides an overview of the system and the documentation.
- Chapter [2](#) provides an overview of VxWorks 653.
- Chapter [3](#) describes the configuration system.
- Chapter [4](#) describes the build system.
- Chapters [5](#) through [21](#) describe the components of an ARINC 653 module and how they are configured and built in the VxWorks 653 build system.
- Chapter [22](#) describes the use cases that make up the VxWorks 653 configuration and build reference process.

This document is supported by the *VxWorks 653 Configuration and Build Reference*, which contains detailed reference information for the XML files and binary components that are used to configure and build a VxWorks 653 system.

## 1.4 Conventions Used in This Documentation

This documentation uses the following conventions:

- Directory paths that are relative to the directory in which VxWorks 653 is installed on your system are indicated with the prefix *installDir*.
- Filenames shown in *italic*, for example, *my-ApplicationDescription.xml* refer to files that are named by the user. You, or the people providing the files, may give them different names.
- File names shown in **bold**, for example, **configRecord.xml**, refer to files that have a set name in VxWorks 653, either as part of the VxWorks 653 installation, or as a generated file that has a fixed name in the VxWorks 653 configuration and build process.
- Several XML files are used in configuring a VxWorks 653 system. In this documentation, each of these files is referred to by its XML document type. The document type is the name of the root element of the document. Thus, a document whose document type is **ApplicationDescription** would have a root element called **ApplicationDescription**. It is referenced in this documentation as the ApplicationDescription document.
- Elements within an XML document are referred to using XPath expressions. An XPath expression uses a syntax similar to that used to describe file paths. Thus the **title** element of an HTML page would be identified by an XPath expression like this:

**/html/head/title**

An attribute of an element is identified by @ before the name. For example, the following XPath expression identifies the **Name** attribute of an **Application** element in a **Module** document:

**/Module/Applications/Application/@Name**

The root element of an XPath expression corresponds to the document type of the document. In the *VxWorks 653 Configuration and Build Reference*, each document type is listed separately and individual XML elements and attributes are identified by Xpath expressions. To locate information on a particular element, start with the document type and then navigate the element tree to find the element you are interested in.



# 2

## *Understanding VxWorks 653*

[2.1 VxWorks 653 5](#)

[2.2 Certification 7](#)

### **2.1 VxWorks 653**

VxWorks 653 is an OS designed for safety-critical applications. In a safety-critical system, applications and operating systems must be protected from the errant behavior of other applications and libraries. To achieve this, VxWorks 653 divides the software system into partitions. Each partition is protected from applications running in other partitions. A set of partitions running on a common core OS is referred to as a module.

The module is controlled by the core OS. The task of the core OS is to schedule the partitions and to provide services to the applications running in partitions. To protect the core OS from errant behavior by applications, applications can communicate only with the partition OS running in the application's partition. The partition OS in turn communicates with the core OS through a controlled API.

VxWorks 653 includes a configurable partition OS called vThreads. VxWorks 653 also supports the development and use of user-provided custom partition OSs based on the core OS interface library (COIL).

You can have more than one type of partition OS in the same module. For example, you can have:

- one or more partitions running different subsets of vThreads
- one or more partitions running a user-defined partition OS based on COIL
- one or more partitions based on a different user-defined partition OS based on COIL

A VxWorks 653 module is made up of domains. A domain is a software container. Each element of the system—the core OS, partitions, and shared libraries—exists in a domain. Domains are established at runtime by the core OS based on information provided in the configuration documents. [Table 2-1](#) summarizes the types of domains in a module.

Table 2-1    **Types of Domains in a Module**

Type of Domain	Contents of Each
Kernel domain	Core OS
ACE domain	ACE
Configuration record domain	System configuration record
User configuration record domains	User configuration record
Application domains	Partition and its associated application
Shared library domains	Shared library
Shared data domains	Shared data region or shared I/O region

The configuration of a VxWorks 653 module is expressed in a configuration record which is used by the core OS at boot time to correctly configure all the domains of the system.

For information about programming for VxWorks 653, vThreads, and COIL, see the *VxWorks 653 Programmer's Guide*.



## 2.2 Certification

A subset of VxWorks 653 is available that is certifiable to Level A of the RTCA/DO-178B avionics software standard. The build system supports building modules in either cert or debug modes.



# 3

## *Configuration System*

[3.1 Understanding VxWorks 653 Configuration 9](#)

[3.2 XML Configuration Files 13](#)

### 3.1 Understanding VxWorks 653 Configuration



---

**NOTE:** The VxWorks 653 build system is not compatible with the Wind River Workbench 2.6.1 build system. To successfully configure and build a VxWorks 653 module you must use the command-line tools.

---

The VxWorks 653 configuration and build system is designed to minimize the cost associated with developing a safety-critical system by allowing you to manage the cost of change.

First, the configuration system is designed to minimize dependencies between elements of the module. This allows you to build and certify different elements of a module separately, and to avoid the need to rebuild the entire module if a change is required in one element. This also allows organizations to manage the cost of certifying systems by allowing each component to be certified separately, and to avoid the need to recertify the whole module if one applications changes.

Second, the configuration system is designed to support incremental building of module components and the module as a whole, reducing the number of files that have to be rebuilt when a change is made. This documentation contains diagrams

that illustrate the configuration and build process for each component of a module. You can use these diagrams as dependency graphs to determine which elements of a particular module component need to be rebuilt following a change in a particular file.

Third, the configuration and build system is designed to facilitate the independent development of different components of the module by different groups of developers. This allows development to be distributed between development groups playing different roles. In this documentation, tasks are described as the responsibility of specific developer roles. However, the configuration and build system does not enforce these distinctions. Rather, it is designed to allow development organizations the maximum flexibility in distributing development roles in a way that best suits their organization and objectives, as well as the ability to develop their build system in a way that enforces the developer roles that best suit their organization.

### **The Configuration Record**

The configuration of a VxWorks 653 module is expressed in a configuration record that forms part of the payload installed on the target. The core OS reads the information in the configuration record at boot time and run-time to configure the module.

The configuration record is a separate binary file that can be built independently of the other components of the module. To facilitate this, the configuration documents contain all of the resource and memory information for each component. Resources are thus pre-allocated for each application, library, and shared data region. This means that the certification of the configuration record is separate from the certification of the applications and libraries. You can certify each of the applications and libraries independently and then certify the configuration record. Alternatively, configuration data can be certified and the configuration record built before applications and libraries are complete.

The finished applications and libraries must conform to the resource allocations specified in the configuration documents. In particular, the configuration and build system requires you to pre-assign blocks of memory to individual components so that the memory map of the module as a whole can be fixed independently of the development of components.

Each of the components of a module is built separately. Building the module itself consists of creating the configuration record, assembling the other built components, and generating an appropriate system image to boot the target.

## User Configuration Records

VxWorks 653 provides for the inclusion of user configuration records in the core OS. The structure of user configuration records is entirely in the hands of the developer.

## Development Roles

To understand the design of the configuration and build system, it is useful to understand the developer roles it is designed to support, the way in which it supports separate build of different system components, and the organization of the files that are used to configure a module.

Developing a complete embedded system from platform development to final system integration is a complex task that usually involves the cooperation of developers playing different roles.

Three principal roles are defined for purposes of this documentation:

- *platform provider*, who is responsible for developing the *platform*
- *application developer*, who is responsible for developing *applications*
- *system integrator*, who is responsible for the design and specification of the *module* and for integrating a set of applications with a platform to create a module.

In order to work together effectively, developers playing these roles need to negotiate requirements and constraints to determine the characteristics of the various components that will go together to make up a system.

## Models of Development

There are many different ways in which development responsibilities can be distributed and VxWorks 653 is designed to be as flexible as possible in allowing you to create a model of development that suits your organization. For clarity and consistency, this documentation is built around a single model of development. This model is illustrated in the examples in the reference process and is assumed in the discussion of the configuration and build of each of the elements of the module.

This model is based on the assumption that development is divided between developers playing the three roles specified above, and that the platform provider and application developers work independently to create a core OS, libraries, and applications which the system integrator must then integrate to create a complete module. In this model, it is the responsibility of the platform provider to determine the needs of their core OS, partition OSs, and shared libraries and to write and

provide to other developers the CoreOSDescription and SharedLibraryDescription documents as well as the system module (.sm) files for the core OS, partition OSs, and shared libraries. Likewise, in this model, it is the responsibility of the application developer to decide on the requirements for their application and to write and provide to other developers the ApplicationDescription document and system module (.sm) file for their application. It is then the responsibility of the system integrator to assemble these files, write a Module document to incorporate them, and to build the configuration record and system image of the final module.

Another possible model of development is this: The system integrator designs the entire module and writes all of the configuration documents for the core OS, partition OSs, shared libraries, shared data regions, and applications. They then use these documents to compile and certify a configuration record and distribute the configuration files and configuration record to the platform provider and application developers who must then provide a platform and applications that meet the constraints expressed in the pre-certified configuration provided by the system integrator.

These are only two of the many development models that are possible with VxWorks 653. Bear in mind when reading this documentation that the model of development that it presents is only one of the many possible models. You are free to devise and implement a model of development that meets your own needs.

## **Components of the Configuration System**

The configuration system consists of the following components:

### **XML Configuration Files**

The configuration information that is used to build the configuration record of a module is specified in a set of XML files.

### **Binary Component Files**

VxWorks 653 is packaged as a collection of binary component files. To configure a particular module, core OS, partition OS, or application, you will need to specify the components to include, resolve any dependencies of your chosen components, and provide appropriate values for any initialization parameters required by these components. This is generally done through makefiles.

### **Configlettes**

Many binary components have compile-time configuration options. Since the compile-time options cannot be set in binary files, source files, called configlettes,

are distributed for the portions of the components that are compile-time configurable. Sometimes the entire component is contained in a configlet and there is no precompiled binary. Some components have no compile-time options and thus include no configlet code. When including a component in a build, you must make sure that both the binary file and the configlet are included (where they exist), and that the compile-time parameters, if any, are set appropriately. For information on each component and its configlets, see the VxWorks 653 *Configuration and Build Reference*.

### **Makefiles**

The configuration and build of each of the elements that make up a module is managed by makefiles. VxWorks 653 includes makefile variable and rules files to make it easier for you to write your makefiles.

## **3.2 XML Configuration Files**

The VxWorks 653 configuration and build system uses a number of different XML files. Each XML file is based on an XML document type defined in one of two XML schemas. The schemas are:

- VxWorks 653 Configuration Schema, which has an XML namespace identifier of:  
"http://www.windriver.com/vxWorks653/ConfigRecord"
- VxWorks 653 Shared Library API schema, which has an XML namespace identifier of:  
"http://www.windriver.com/vxWorks653/SharedLibraryAPI"

The VxWorks 653 Configuration Schema is defined in these schema definition files:

- **Module.xsd** describes the configuration of the module itself, including schedules, health monitoring, and ports.
- **CoreOS.xsd** describes the configuration of the core OS and ACE.
- **Application.xsd** describes the configuration of applications, shared libraries, system shared libraries, and shared data and I/O regions.
- **Partition.xsd** describes the configuration of partitions.

- **TypeLib.xsd** contains type definitions used by the other schemas.
- **WR\_ConfigRecord.xsd** contains the definition of the configuration record document type, which is used to assemble the various parts of the configuration record.
- **WR\_Extensions.xsd** contains the definition of resource allocations calculated by the build tools.
- **XInclude.xsd** is the W3C XInclude schema, which is used to incorporate the different configuration documents to form a complete configuration record.

The VxWorks 653 Shared Library API Schema is defined in these schema definition files:

- **SharedLibraryAPI.xsd** describes the configuration of a shared library interface.
- **XInclude.xsd** is the W3C XInclude schema, which is used to incorporate interface subset definitions to create a complete interface definition.

The document types defined by these schemas have a hierarchical relationship to one another, meaning that one document type fits inside another document type at a particular point. For instance, the **ApplicationDescription** document type fits inside the **Module** document type in the **/Module/Applications/Application** element. These relationships are normally expressed by using an **xi:include** element to include the document by reference. However, it is also possible to include the document inline.

[Table 3-1](#) lists the document types, the schemas that define them, and the schema files in which the particular document type is defined.

Table 3-1 **Document Types and the Schemas that Define Them**

Document Type	Schema	Schema File
Module	VxWorks 653 Configuration	<b>Module.xsd</b>
Ace	VxWorks 653 Configuration	<b>CoreOS.xsd</b>
ApplicationDescription	VxWorks 653 Configuration	<b>Application.xsd</b>
PartitionDescription	VxWorks 653 Configuration	<b>Partition.xsd</b>



Table 3-1 Document Types and the Schemas that Define Them (cont'd)

Document Type	Schema	Schema File
PseudoPartitionDescription	VxWorks 653 Configuration	Partition.xsd
SharedLibraryDescription	VxWorks 653 Configuration	Application.xsd
CoreOSDescription	VxWorks 653 Configuration	CoreOS.xsd
SharedDataDescription	VxWorks 653 Configuration	Application.xsd
SharedIODescription	VxWorks 653 Configuration	Application.xsd
ConfigRecord	VxWorks 653 Configuration	WR_ConfigRecord.xsd
SharedLibraryAPI	VxWorks 653 Shared Library API	SharedLibraryAPI.xsd
InterfaceSubset	VxWorks 653 Shared Library API	SharedLibraryAPI.xsd

For information on creating any of these document types, see the *VxWorks 653 Configuration and Build Reference*. Examples of the configuration documents, together with detailed explanations of what they achieve, are found in [22Reference Process](#), p.155.

In the default build process, the ConfigRecord document is generated by the build tools based on the information contained in the Module document and its subsidiary configuration documents. If you substitute your own build process, you will need to generate a valid ConfigRecord document or write one by hand.

Creating XML Files

You can create XML configuration files using any editor you choose. If you use a schema-aware XML editor, it will guide you through the creation of the files and make sure that your files comply with the schema.

To create XML files with a schema-aware editor:

1. Following the instructions in your editor documentation, open the appropriate schema for the file you are creating. The schemas are located in:

*installDir/vxworks653-2.2/target/config/xml/cleanschema*

2. Following the instructions in your editor documentation, create or edit the appropriate files. You can also choose a file from the reference process that is closest to the file you want to create and edit it. The reference process files are located in:

*installDir/vxworks653-2.2/target/reference/helloWorld*

3. Check your document for conformance to the business rules used by the configuration and build process using VerIMAx. For information on VerIMAx, see the *Wind River Workbench User's Guide, (VxWorks 653 Version)*.

To create XML files with an editor that is not schema-aware:

1. Choose a file from the reference process that is closest to the file you want to create or create the file from scratch following the instructions for the appropriate document type in the *VxWorks 653 Configuration and Build Reference*. The reference process files are located in:

*installDir/vxworks653-2.2/target/reference/helloWorld*

2. Name the file to suit your project.
3. Open the file in your editor.
4. Edit the document as required to achieve your desired configuration.
5. Validate your XML document against the appropriate schema using the validation tool of your choice.
6. Check for conformance to the business rules used by the configuration and build process using VerIMAx. For information on VerIMAx, see the *Wind River Workbench User's Guide*.

## Extending the Document Types

The VxWorks 653 Configuration Schema allows you to create your own extensions to the configuration document types at specific points. You may want to make use of this facility to include your own configuration parameters in the configuration files. To see where you can insert your extensions, look for the **Extensions** elements in the schema.

You will need to create custom tools to act on this custom configuration data. Extensions will not be included in the configuration record in the payload. You can create your own configuration record if you want and store it in a user

configuration record region of core OS memory. Only the core OS has access to this configuration record region. Information that is needed by applications can be placed in a shared data region. For information on user configuration records, see the *VxWorks 653 Programmer's Guide*. For information on shared data regions, see [10. Shared Data Regions](#).



# 4

## *Build System*

4.1 Understanding the Build Process 19

4.2 Planning a Build 20

4.3 Building for VxWorks 653 21

### 4.1 Understanding the Build Process

VxWorks 653 has a command-line build system designed to support separate building of the various components of a system (core OS, shared libraries, applications, shared data regions, and system images). The build system is driven by makefiles.



---

**NOTE:** The VxWorks 653 build system is not compatible with the Workbench 2.6.1 build system. To successfully configure and build VxWorks 653 you must use the command-line tools.

---

There are a number of supporting tools that are used in the build process. For details on the build tools, see the *Wind River Workbench User's Guide, (VxWorks 653 Version)*.

The invocation of the build tools is managed by makefiles supplied with VxWorks 653. Unless you intend to customize the build system, you do not need to understand how these tools work.

There are several parts to the build of a VxWorks 653 module:

- core OS build
- ACE build
- user configuration record build
- shared library build
- shared data region build
- application build
- integration build

Each build part can be performed independently of the others. Individual elements of the module can be compiled separately. However, in order to link each element into a system module (**.sm**) file, the following dependencies must be met:

- The shared library build requires that the virtual address of the shared library be specified. The virtual address depends on the memory configuration of the core OS. It also depends on the size and virtual address of the other shared libraries in the module. When building a shared library, therefore, you need to create a virtual memory map for the module in order to calculate a reasonable virtual address for each shared library.
- The application build requires stubs files for all the shared libraries used by the application. The stubs files are created as part of the shared library build.

## 4.2 Planning a Build

The following are some of the key questions to keep in mind when planning your build strategy:

### **Are you building a cert or debug image?**

You must choose whether to build a cert or a debug image. A cert image is an image that is suitable for deployment as a certified system. A debug image contains debugging code that cannot be included in a cert image. (ACE can be used to debug a cert image. For more information on debugging, see the *Wind River Workbench Users Guide, VxWorks 653 version*.) A debug image will bind in a different set of libraries. An error will occur if you attempt to include a debug

component in a cert build. When you choose a cert image, the **CERT** macro is defined when compiling C, C++, and assembly language files.

#### Which CPU are you building for?

You must choose a build spec for the CPU that you are building for. This will cause the build to produce code appropriate for the selected CPU. The available build specs will depend on the BSPs that you have installed. You select a cert or debug build by choosing the appropriate build spec for your CPU, build tool, and image type. The names of build specs encode these pieces of information, so that, for instance, the build spec **PPC604gnu.debug** specifies a PPC604 target, the gnu tool chain, and a debug image

#### Which system image type will you build?

VxWorks 653 supports three different image types: network loadable, RAM payload, and ROM payload. Each is used for a different part of the development and deployment cycle. The type of image that you build will be determined by the build target that you choose in your system image makefile. For more information on system images, see [21. System Images](#).

## 4.3 Building for VxWorks 653

In VxWorks 653 the building of the various components of a module, and the module itself, is handled by makefiles. Explanations of the makefiles used for each of the components can be found in topics on each component and in [22Reference Process](#), p.155.

#### Makefile.vars and Makefile.rules

VxWorks 653 provides a number of makefiles to help you build the components of a module. For each type of module component that you can build, there are two makefiles, **Makefile.vars**, which contains variable definitions, and **Makefile.rules**, which contains rules for building different outputs. You will include one or both of these makefiles in your makefiles for each module component.





# 5

## *Memory*

5.1 Understanding Memory 23

5.2 Planning Memory 27

5.3 Configuring Memory 28

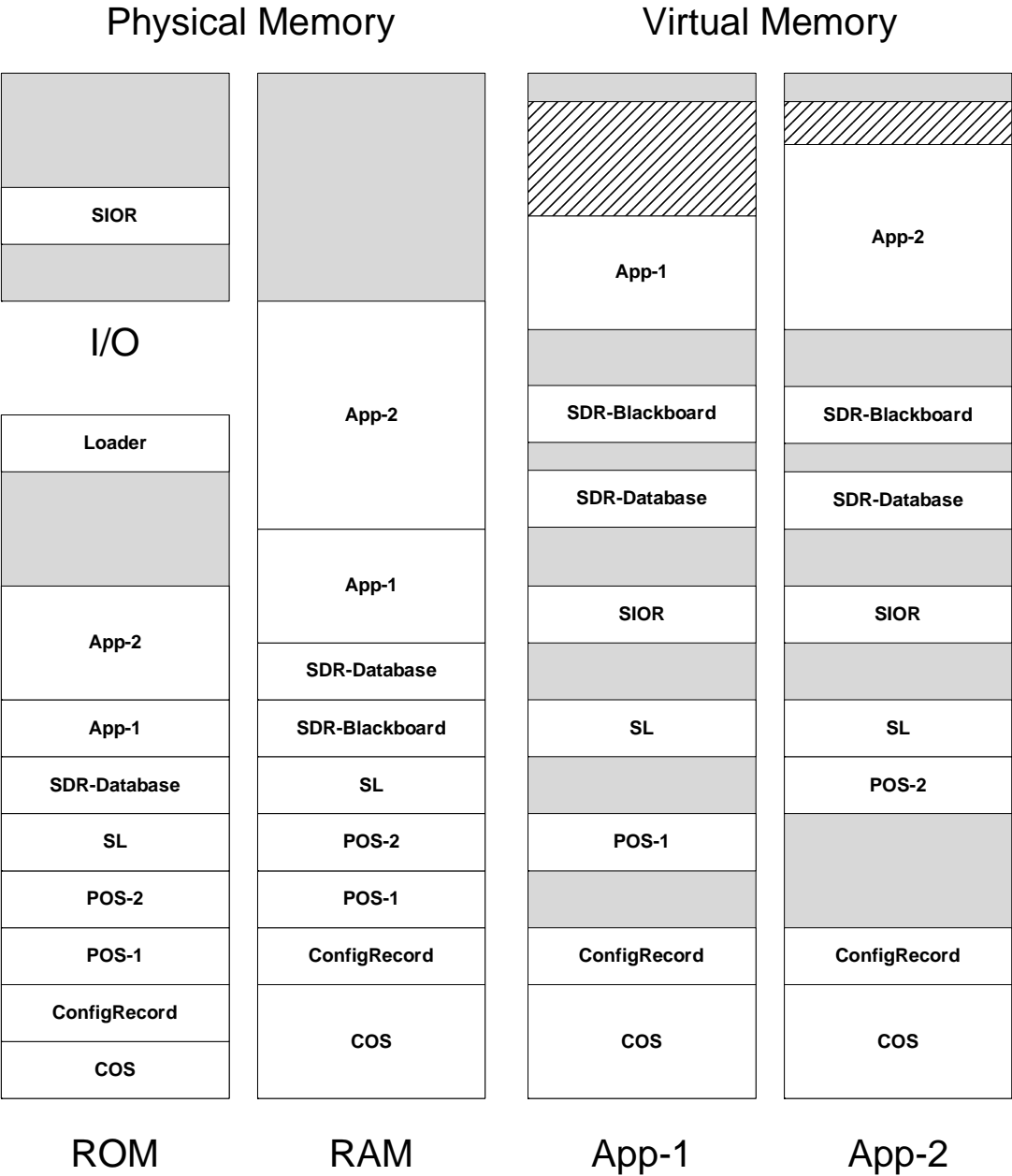
### 5.1 Understanding Memory

There are two parts to the memory configuration of a VxWorks 653 module: the physical memory configuration and the virtual memory configuration. [Figure 5-1](#) shows the organization of physical and virtual memory in a VxWorks 653 module. The diagram shows a module with two applications: App-1 and App-2. The physical memory map on the left shows the organization of physical memory on the target for a sample system. (This sample includes some optional elements but not all of the elements that can possibly occur in a VxWorks 653 module.) The two virtual memory maps on the right show how virtual memory appears to App-1 and App-2.

#### ROM

The ROM section of the diagram shows how the components of a module might be arranged in a ROM payload. The core OS and applications are smaller in ROM than they are in RAM, since the copies in RAM require additional space for stack

Figure 5-1 Physical and Virtual Memory Allocation



and heap. There is no sharing of memory in VxWorks 653, so each application requires its own stack and heap space, as does the core OS. Each application must also set aside memory for the stack and heap requirements of each shared library that it accesses, including the partition OS that it uses. The ROM section also contains the loader, which is typically located at the top of ROM and is used to load the system image from ROM to RAM at boot time.

## **I/O**

The I/O section of the diagram represents the I/O hardware on the target board. One part of this I/O address space has been mapped to a shared I/O region (SIOR).

## **RAM**

The RAM section of the diagram shows how the components of the system might be located in RAM after they have been loaded. The space required by each application and by the core OS has been set by the information contained in the configuration record, which includes their stack and heap requirements. The space required by the partition operating systems (POS-1 and POS-2) and by the shared library (SL) have not been increased, as their stack and heap space is provided by each application that uses them.

RAM contains two shared data regions: SDR-Database and SDR-Blackboard. SDR-Database is a loadable shared data region, meaning that it contains pre-compiled data that is loaded into RAM at boot time. Thus SDR-Database occurs in the ROM payload as one of the items to be loaded into RAM at boot time. SDR-Blackboard, however, is a non-loadable shared data region, meaning that it contains no precompiled data. It is simply an area of memory that is set aside for the use of two or more applications. Applications may use a non-loadable shared data region as a place to exchange data.

App-1 and App-2 are loaded into separate areas of RAM. There is no sharing of memory between applications (except in the form of shared data regions) and each application receives its full allocation of memory at load time.

## **Virtual Memory**

In virtual memory, every component of the system has a fixed address. However, all applications have the same virtual address. Each application starts at the same address in virtual memory. Applications are not aware of the existence of other applications (though they may be aware of resources that are shared with other applications). Thus, an application is configured almost as if it were the only application in the system. The size of the virtual address space available for

applications is the same for all applications, even if some applications are larger than others. The shared I/O region is also mapped into virtual memory space.

Applications cannot see all of the module components that exist in virtual memory, however. Each application exists in a virtual container called a partition, and the partition configuration governs which resources are available to a particular application. Each application uses a different partition OS. Thus App-1's virtual memory space includes POS-1, but not POS-2, and similarly for App-2. Both applications have access to shared library (SL) so it is visible to both of them. Similarly, the shared I/O region and both shared data regions are visible to both.

The core OS and configuration record are not directly visible to the application, but are shown in the diagram to show their location in the overall virtual memory space of the module. Applications are not visible to each other at all since they occupy the same location in virtual memory. However, applications can communicate with each other through shared data regions or through ARINC ports, if they exist in the module.

## **Partitions**

Applications know nothing about the other applications in the module. Each application resides in a container called a partition. The partition is responsible for the application's interactions with the rest of the module and for the objects that the application can access in the module. Partitions provide the memory in which each application runs. The partition also determines which shared resources (libraries, I/O, data regions) the application has access to. Because each application must execute in a separate area of physical memory, each partition must also provide the read/write memory required to execute any shared library code that the application has access to.

## **Domains**

VxWorks 653 is designed to protect each application and the core OS from interference by other parts of the module. This protection requires that memory access for each object in the system must be restricted to that object itself (except where specific and limited permissions are given, as in the case of shared libraries, I/O, and data). This protection is provided by domains. Domains are also referred to as protection domains because their role is to protect the elements of the system that they contain. The core OS, shared libraries, shared data regions, and partitions all exist in separate domains and have no access to each other except as specifically permitted and configured in the module configuration record.

## 5.2 Planning Memory

Planning memory usage for a module involves looking at the memory configuration of the target as well as the memory requirements of the various components of a module. Different targets reserve different portions of memory for different purposes. Some processors have limits on how close together different types of memory may be located. All of this must be taken into consideration when you are deciding where to place different components of your module in memory. Many build errors result from a failure to consider all the factors affecting memory configuration. You should study your BSP documentation carefully to determine what the constraints and requirements are for memory configuration on your chosen target.

When planning memory, you should consider the following questions:

### **How will physical memory be organized?**

The physical memory of the target must be configured for the use of the various components of the module. In many cases, the settings in the template `CoreOSDescription` document will be sufficient. In some cases you may need to adjust memory allocations if the memory regions defined are too small for some part of your module.

### **How will virtual memory be organized?**

As part of the configuration of a module, you will need to determine where each element of the module is located in virtual memory. This requires that you make a virtual memory map of your module, taking into consideration the specific requirements of your target and the size and type of each of the components in your module. For shared libraries and shared data regions, VxWorks 653 provides a tool that can be used to suggest available virtual addresses in your module configuration. For information, see the reference entry for **XMLGen**.

### **How will memory configuration affect certification?**

VxWorks 653 is designed to reduce costs associated with certification by allowing each component of a module, and the module configuration as a whole, to be certified separately, and to isolate the different components from the impact of changes in other components. A large part of this isolation is achieved through the use of memory black boxes which predefine the memory resources that each component will need. You can help to minimize certification costs by making your memory calculations as accurate as possible, and also by making sure that enough memory is assigned to each component to accommodate unforeseen changes in

that component so that if the component does have to change, this change does not impact other components of the module.

## 5.3 Configuring Memory

The configuration of memory is divided into several parts.

### Physical Memory Configuration

The configuration of physical memory is done as part of configuration of the core OS by specifying the appropriate value in the CoreOSDecription document in the element **CoreOSDescription/HardwareConfiguration/PhysicalMemory**. This element contains settings for the following memory regions:

#### kernelMemoryRegion

Contains the memory for the core OS black box and the core OS task stacks and heap.

#### kernelPgPool

Optional area used only when dynamic loading is enabled in a debug build.

#### kernelRegion

Contains memory for user regions which must reside in the core OS memory context. This is primarily used for online-loaded partitions. For information on online-loaded partitions, see [15. Partitions](#).

#### portRegion

Contains memory for the heap of the APEX ports component. The size requirement for this region depends on the number of ports in the module. For information on APEX ports, see [16. Ports and Channels](#) and the *VxWorks 653 Programmer's Guide*.

#### hmLogRegion

Contains memory set aside for storing health monitor logs. For information on health monitoring, see [18. Health Monitor](#).

#### userConfigRecordRegion

Contains memory set aside for a user configuration record. For information on user configuration records, see the *VxWorks 653 Programmer's Guide*.

**ramPayloadRegion**

Contains the memory allocated for the use of a RAM payload. For information on RAM payloads, see [21. System Images](#).

**aceMemoryRegion**

Contains the memory allocated for the use of the ACE. For information on ACE, see [12. ACE](#).

**kernelConfigRecordRegion**

Contains the memory allocated for the use of the system configuration record. For information on the system configuration record, see [20. Configuration Record](#).

**userMemoryRegion**

Contains the memory allocated for every partition, shared data pool, and shared library pool in the module.

VxWorks 653 includes a CoreOSDescription document for each supported BSP which contains a set of defaults for these values that are appropriate to the particular BSP. You may need to adjust one or more of these values depending on the physical memory requirements of your module.

## Virtual Memory Configuration

VxWorks 653 is designed to facilitate the independent building of different components of the module, and to minimize the impact that changes in one component of the module have on other components. To this end, virtual memory assignment for each component can be determined separately from the actual build of the component. This allows a component to be changed and rebuilt, without affecting its memory assignment, or the location of its various sections in memory. This helps reduce the development and certification costs for certified systems by allowing the memory configuration of the module to be certified independently of the certification of the components. This means that as long as each component continues to fit into its assigned memory area, the memory configuration of the module does not have to be recertified when an individual component changes.

### Black Boxes

This pre-determined memory configuration is achieved using black boxes. A black box is a section of memory set aside for a component of the system (core OS, shared library, application, partition, or shared region). Memory allocation for the component as a whole, and for each section of the component ELF file, is based on black box sizes, not on the actual memory footprint of the components. For instance, in the **MemorySize** element of an application or library configuration,

each section of an ELF file is specified with a separate black box value, as in this example:

```
<MemorySize
  MemorySizeBss="0x1000"
  MemorySizeData="0x1000"
  MemorySizePersistentBss="0x1000"
  MemorySizePersistentData="0x1000"
  MemorySizeRoData="0"
  MemorySizeText="0x1000" />
```

The actual sizes of these sections in the application or library ELF file must be less than or equal to the black box sizes in the configuration document. The black box sizes are used to assign memory for the component they describe, regardless of the actual sizes of the respective ELF sections.

Each section aligns on the start of its black box. This allows jump tables to be calculated even if the application or library is not finalized. This helps to ensure that changes to one part of a module do not lead to the need to recertify the entire module. A linker script is generated from the black box data to align the sections on the black box boundaries.

Heap and stack space for applications and the core OS is not defined in black boxes, but must be added to the overall memory allocation of the partition or core OS.

For shared libraries, including the system shared libraries that contain the partition OS, the read-only portions of the library are shared, and space is allocated for them in the shared library configuration. The read/write requirements for each library, including stack and heap, are provided separately by each partition that uses the library. As part of the configuration of a partition, you will designate which libraries a particular partition will use. It is important that a partition designate only those libraries that it actually uses, or memory will be wasted.

### **Distributed Configuration**

The component black boxes, and of total core OS and partition memory, are configured with the configuration of the individual components. The settings affecting virtual memory are found in the following locations:

#### **CoreOSDescription/KernelConfiguration/ @addressSpaceRegionAllocationUnit**

The size of the virtual memory allocation unit for the target.

#### **CoreOSDescription/KernelConfiguration/@addressSpaceSize**

The total size of the virtual memory address space. This is specific to the target.



**CoreOSDescription/KernelConfiguration/@kernelVirtualAddress**

The virtual address of the kernel.

**CoreOSDescription/MemorySize**

The black box for the core OS.

**CoreOSDescription/KernelConfiguration/@partitionVirtualAddress**

The virtual address of every application.

**ApplicationDescription/MemorySize**

The black box for an application.

**PartitionDescription/Settings/@RequiredMemorySize**

The total size of the partition memory, which must include the application black box, the application stack and heap space, and read/write memory space for the partition OS and all shared libraries accessed by the partition.

**SharedLibraryDescription/MemorySize**

The black box for a partition OS or shared library.

**SharedLibraryDescription/@VirtualAddress**

The virtual address of a partition OS or shared library.

**SharedDataDescription/@Size**

The size of a shared data region.

**SharedDataDescription/@VirtualAddress**

The virtual address of a shared data region.

**SharedIODescription/@VirtualAddress**

The virtual address of a shared I/O region.

**CoreOSDescription/HardwareConfiguration/sharedIO/@Size**

The size of a shared I/O region.

**Payload Memory Configuration**

Payload memory configuration for ROM payloads is specified in the Module configuration document in the element **/Module/Payloads**. For more information, see the *VxWorks 653 Configuration and Build Reference*. Payload memory is based on the virtual memory configuration for the module. The only additional configuration required is the assignment of an appropriate amount of RAM as a RAM payload region in the CoreOSDescription document in the element **CoreOSDescription/HardwareConfiguration/PhysicalMemory/ramPayloadRegion**.



# 6

## *Core OS*

6.1 Understanding the Core OS 33

6.2 Planning the Core OS 34

6.3 Configuring the Core OS 36

6.4 Building the Core OS 41

### **6.1 Understanding the Core OS**

The core OS is responsible for partition scheduling, resource allocation, and health monitoring for the applications running in partitions. It also facilitates communication between partitions by means of ARINC ports. It reports hardware events and other information to the partitions by way of pseudo-interrupts. It manages partition access to shared resources such as shared data regions and shared I/O regions.

The overall memory configuration of a module is determined by the core OS configuration. The core OS memory configuration includes the provision of kernel memory regions, the RAM payload region, and the size of the application memory area. It also determines the location into which online-loaded partitions are loaded, and provides space for ACE, if ACE is being used.

A typical platform will include two versions of the core OS: a debug version, and a cert version.

The core OS is the responsibility of the platform provider. For information on development roles, see [3. Configuration System](#).

## 6.2 Planning the Core OS

When planning the core OS, you should consider the following questions:

### **Which components will be included, and where?**

A core OS for a particular platform is made up of a collection of components that provide various OS services. You can configure the core OS with different capabilities by including different components in the core OS build.

The default core OS build process produces a core OS project that includes a number of components by default. Default components include operating system, device support, and development tool support components. Other available components include support for POSIX, and additional debug tools. For a cert system, a reduced set of components is available for the core OS and a separate set of component binaries is used.

Several core OS components are commonly required to provide one feature or set of functionality in the core OS. To make it easier to select the components required for a particular feature, component bundles are provided.

For more information on core OS components and component bundles, see the *VxWorks 653 Configuration and Build Reference*.

### **Will shared I/O regions be required?**

In order for applications to access memory-mapped I/O devices (such as LEDs) on a target, you must configure those memory regions as shared I/O regions.

### **Will user configuration records be required?**

If your core OS requires user configuration records, memory must be allocated for each user configuration record. A user configuration record is a named memory pool set aside for user configuration data. The form of that data and the method for accessing it are entirely up to the user.

### Will support for online-loaded partitions be required?

If your platform is to support online-loaded partitions, you must configure a kernel memory region into which each partition can be loaded. You must also ensure that you add the required components to the core OS to support the loading of online-loaded partitions, and you must provide a partition loader. For information on writing a partition loader, see the *VxWorks 653 Programmer's Guide*. For information on building online-loaded partitions, see [15. Partitions](#).

### Will kernel memory regions be required?

If the custom code that you are adding to the core OS requires kernel memory regions, you must configure them appropriately.

### Will ACE support be required?

If ACE support is required, you must configure an ACE memory region in the core OS. For information on configuring ACE itself, see [12. ACE](#).

### Will RAM payload support be required?

If your platform is going to support RAM payloads, RAM payload memory must be configured. The template CoreOSDescription document for your BSP may already define a RAM payload region. If you do not want RAM payload support, you may need to remove it from the configuration. For information on RAM payloads, see [21. System Images](#).

### Will boot loaders be required?

If your core OS is to support RAM or ROM payloads, you must also build boot loaders for RAM and ROM payloads. This will be specified through a command in the core OS makefile. In almost all cases, boot loaders will be required. For information on payloads, see [21. System Images](#).

### Will more than 32 partitions be required?

The template CoreOSDescription document in the BSP is configured to support up to 32 partitions. If your platform needs to support more than 32 partitions, you will need to adjust the core OS configuration accordingly.

To support more than 32 partitions, you may need to increase some or all of the following settings under the

**CoreOSDescription/HardwareConfiguration/PhysicalMemory** element:

- **kernelConfigRecordRegion**

- **portRegion**
- **hmLogRegion**
- **userMemoryRegion**

You may need to change the following core OS parameters:

- **PD\_MAX\_NUMBER\_OF\_PDS**, which limits the number of protection domains in a module, must be greater than or equal to the number of partitions, shared libraries, and shared data regions, plus one (for the core OS). The default value is 64.
- **NUM\_FILES**, which sets the number of files that can be open at once. The default value is 50 and the maximum is 1024. If the project is built with the **INCLUDE\_PARTITION\_TOOL** component, the kernel will set up three standard pseudo-I/Os for each partition. When there are multiple partitions, the kernel may reach the default system limit.

## 6.3 Configuring the Core OS

Figure 6-1 summarizes the core OS configuration and build process.

The following are the inputs and outputs of the core OS configuration and build process.

### Outputs of the Core OS Configuration and Build Process

The outputs of the core OS configuration and build process are as follows:

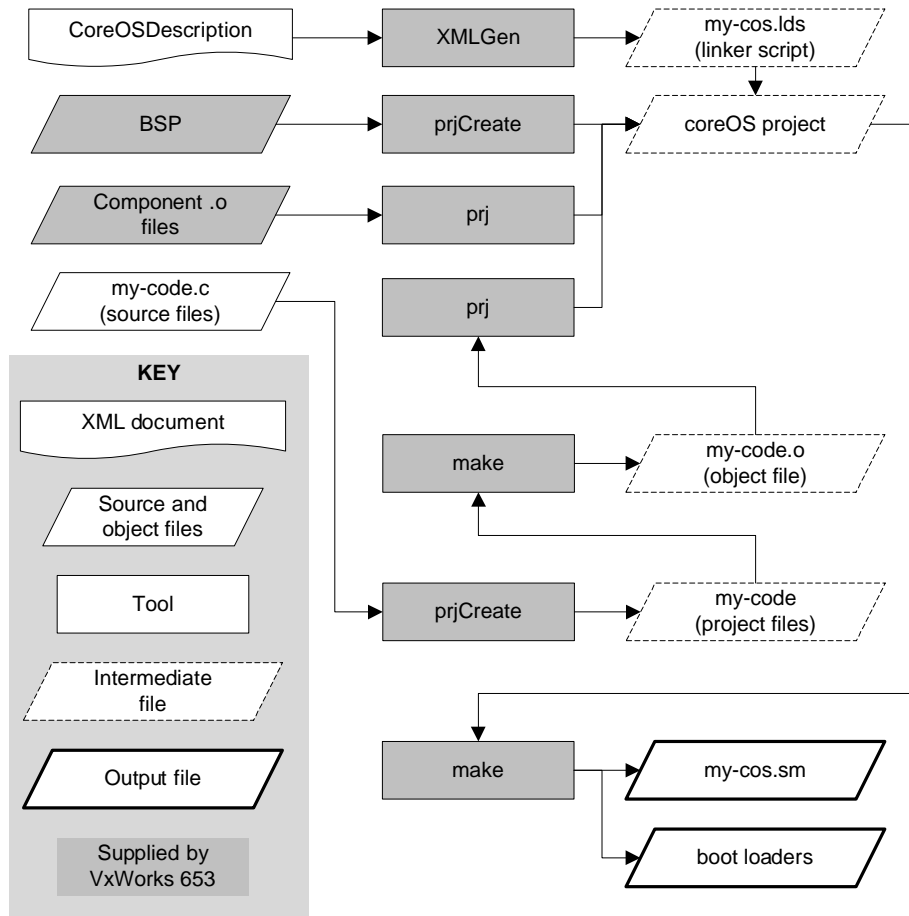
#### Core OS System Module File

The core OS system module file (*my-cos.sm* in the diagram) is a linked and located ELF file that is ready to be included in a the build of a system image.

#### Boot Loaders

The ROM and RAM payload builds require boot loaders to load the system image into memory. These boot loaders are specific to a particular core OS configuration (since the core OS configuration includes the physical memory allocations for the target). The core OS build process therefore produces ROM and RAM payload boot loaders, named **payloadObjs\_rom.o** and **payloadObjs\_ram.o**. These files must be

Figure 6-1 Core OS Configuration and Build Process



delivered to the system integrator as part of the platform. For more information on RAM and ROM payload builds, see [21. System Images](#).

### Inputs to the Core OS Configuration and Build Process

The inputs to the core OS configuration and build process are as follows:

### CoreOSDescription document

The CoreOSDescription document is an XML document conforming to the CoreOSDescription document type defined in the VxWorks 653 Configuration Schema.

This document contains configuration information for the core OS in the following areas:

- kernel configuration
- core OS memory black box
- physical memory configuration
- I/O configuration

Much of the configuration of a core OS involves settings that are specific to the target hardware. For this reason, a template CoreOSDescription file is included with each supported BSP. This file should be your starting point for configuring the core OS. You may be able to use the file from the BSP unchanged, or you may need to make your own CoreOSDescription document based on the CoreOSDescription document in the BSP.

For detailed information on creating or modifying the CoreOSDescription document, see the *VxWorks 653 Configuration and Build Reference*.

### Core OS Project

The core OS is built using a core OS project. The creation and configuration of this project is handled by a set of project commands. These commands create the files necessary to build the core OS. All the files necessary to create the core OS, with the exception of the CoreOSDescription document, must be placed in the core OS project.

The command to create a core OS project is **prjCreate**. The following is a typical **prjCreate** command:

```
prjCreate -domtype kernel -prj my-kernel -bsp wrSbc750gx -name coreOS
```

This line calls the prjCreate utility to create a core OS project. It specifies the type of project to create (kernel), the directory to create it in (my-kernel) the BSP to use (wrSbc750gx), and the name of the project (coreOS).

To create a cert kernel, you use the following command:

```
prjCreate -domtype kernel -prj my-kernel -bsp wrSbc750gx -name coreOS -ddf  
certKernel
```

For additional **prjCreate** options, see the reference entry for **prjCreate**.



## Core OS Build Spec

The core OS build will access different libraries depending on whether you are building a cert or debug core OS. You set the build spec for the core OS project with the **prj projBuildSet** command. The following is an example of the **prj projBuildSet** command that specifies a debug build for the PPC604 target using the gnu tool chain.

```
prj projBuildSet -prj my-kernelDir PPC604gnu.debug
```

The default build spec for any project is based on the BSP used to create the project. It uses the gnu tool chain and creates a debug image. For this reason, you only need to specify the build spec if you are changing the tool chain or changing the build type from debug to cert.

The binary files produce by building the core OS project will be located in a directory whose name corresponds to the name of the build spec. So, for instance, the binary files created by building the project created with the commands in this example would be located in a directory **my-kernelDir/PPC604gnu.debug**.

## Core OS Components

There are a number of optional core OS components that you can include in your core OS project. You can add components to your core OS using the **prj domComponentAdd** command:

```
prj domComponentAdd -prj my-kernelDir "REQUIRED_COMPONENTS"
```

The default project created by the **prjCreate** command includes a number of components by default. You can remove unneeded components with the **prj domComponentRemove** command:

```
prj domComponentRemove -prj my-kernelDir "UNNEEDED_COMPONENTS"
```

## Component Bundles

Components are packaged into bundles by function. You can add a component bundle with the **prj domComponentBundleAdd** command, specifying the project directory (with the **-p** option) and the location of the **.ddf** file that describes the bundle. For information on component bundles, see the *VxWorks 653 Configuration and Build Reference*.

```
prj domComponentBundleAdd -prj my-kernelDir \  
$(WIND_BASE)/target/config/comps/vxWorks/sysTemplates/vxKernel/windview.ddf
```

## Component Parameters

There are a number of core OS configuration parameters that are set with **prj** commands. These parameters are listed in the *VxWorks 653 Configuration and Build Reference*. The following example changes the value of the **NUM\_FILES** parameter:

```
prj domParameterValueSet -prj my-kernelDir NUM_FILES 128
```

## Component Dependencies

Many components have dependencies on other components. You can check to make sure all the required components have been included with the **ADD\_NEEDED** command. Once again, the core OS project directory is specified, along with the **ADD\_NEEDED** command.

```
make -C my-kernelDir ADD_NEEDED
```

## Custom Core OS Components

If you want to add your own code to the core OS, you must assemble your code into a component and add that component to the core OS project. To do this, use the following procedure:

1. Create the project for your component by adding a **prjCreate** command to the coreOS build rule of your core OS project makefile:

```
prjCreate -type kernelComponent \  
-prj my-componentDir \  
-build PPC604gnu.debug \  
-srcfiles "/src/my-component.c /src/other.c"
```

2. Add the component to your core OS project by adding a **prj domComponentAdd** command to your build rule:

```
prj domComponentAdd -p my-kernelDir my-componentDir
```

3. Set any attributes required by your component. At minimum, you will need to set the **INIT\_RTN** parameter to specify the init routine for your component:

```
prj compAttributeSet -p my-componentDir INIT_RTN "myComponentInit();"
```

For additional information on the **prjCreate** and **prj** commands used to introduce custom code into the core OS, see the reference entries for **prjCreate** and **prjScriptLib**.

## Core OS Linker Script

Because the memory configuration of the core OS is specified by a black box defined in the CoreOSDescription document, a custom linker script is required to align the sections of the core OS ELF file on the correct boundaries. The linker

script can be generated from the CoreOSDescription file using the **XMLGen** tool. The following is a typical **XMLGen** linker script command:

```
xmlgen --ldScript --arch ppc -o my-kernelldir/coreOS.lds my-coreOS.xml
```

The `--ldScript` option tells **XMLGen** to generate a linker script.

The `--arch` option specifies the target architecture ("ppc"). In a makefile that imports **Makefile.vars**, the architecture can be specified with the variable `$(TOOLARCH)`.

The `-o` option specifies the name and location of the output file (my-kernelldir/coreOS.lds). The output file must be placed in the project directory created by **prjCreate**. The name of the output file must match the value of `/CoreOSDescription/@KernelName` in your CoreOSDescription document, and the extension must be `.lds`.

The command-line parameter specifies the name of the CoreOSDescription document (*my-coreOS.xml*).

For a complete list of **XMLGen** options, see the reference entry for **XMLGen**.

## 6.4 Building the Core OS

To build the core OS, use the following procedure:

### Step 1: Create a core OS project makefile.

Create a makefile using the editor of your choice. For more information on makefiles, see the **make** documentation.

The following is a typical core OS project makefile for a cert core OS. For other examples of core OS project makefiles, see [22. Reference Process](#).

```
CPU=PPC604
include $(WIND_BASE)/target/config/make/Makefile.vars

coreOS:
    prjCreate -domtype kernel -prj my-kernelldir -bsp wrSbc750gx \
    -name coreOS -ddf certKernel
    prj projBuildSet -p my-kernelldir PPC604gnu.cert
    xmlgen --ldScript --arch $(TOOLARCH) -o my-kernelldir/coreOS.lds \
    my-coreOS.xml
```

**Step 2: Open the VxWorks 653 Development Shell.**

The VxWorks 653 build tools require a specific build environment which is provided by the VxWorks 653 Development Shell. To open the shell, from your program list select:

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

**Step 3: Create the core OS project.**

To build the core OS project, run **make**, specifying the name of your core OS project target:

```
make coreOS
```

**Step 4: Build the core OS.**

The core OS project contains a makefile that will build the core OS. To build the core OS, run **make** specifying the **-C** option with the core OS project directory:

```
make -C my-kernel_dir
```

Rather than running this command separately, you may choose to add this line to the core OS project makefile.

**Step 5: Build the boot loaders.**

The core OS project makefile also contains a target to build the boot loaders. To build the core OS, run **make** specifying the **-C** option with the core OS project directory, and the **payloadObjs** target:

```
make -C my-kernel_dir payloadObjs
```

Rather than running this command separately, you may choose to add this line to the core OS project makefile.

# *User Configuration Records*

[7.1 Understanding User Configuration Records](#) 43

[7.2 Planning User Configuration Records](#) 44

[7.3 Configuring User Configuration Records](#) 44

[7.4 Building a User Configuration Record](#) 46

## **7.1 Understanding User Configuration Records**

A user configuration record is a loadable data region in the core OS that can be used for storage of a user-defined configuration record or other data to be used by a user extension to the core OS. A user configuration record is loaded into a user configuration record region defined in the core OS configuration.

For information on accessing a user configuration record, see the *VxWorks 653 Programmer's Guide*.

User configuration records are part of a platform. For information on platforms, see [13. Platforms](#).

Memory for user configuration record regions is assigned as part of the configuration of the core OS. For information on the core OS, see [6. Core OS](#).

User configuration records are usually the responsibility of the platform provider. For information on development roles, see [3. Configuration System](#).

## 7.2 Planning User Configuration Records

VxWorks 653 allows you to set aside user configuration record regions, and to load precompiled user configuration records as part of the module payload. The structure and access mechanism for user configuration records is up to the core OS developers who will be using the data.

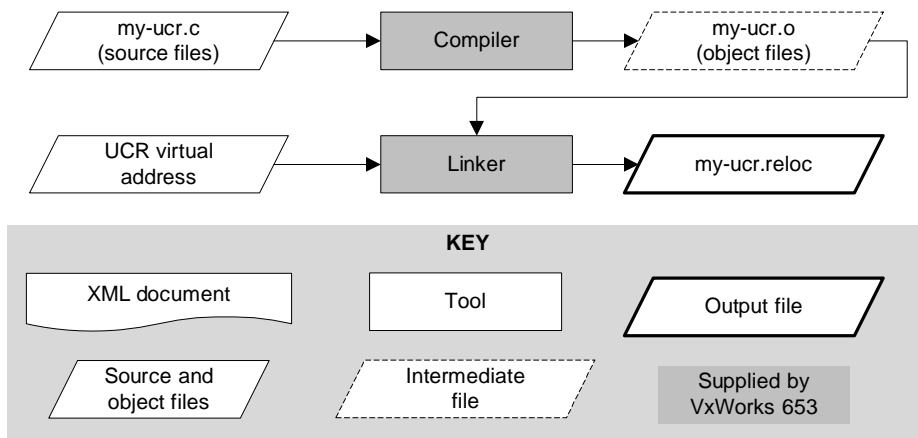


**CAUTION:** You are responsible for the certification of your user configuration record binary. The system configuration record is compiled to binary form by VerIMAX, a qualified tool. Since the semantics of your user configuration record are determined by you, there is no pre-existing qualified tool to compile your user configuration record to binary format. You are responsible for the certification of your user configuration record, and for the qualification of any tools you use to build it.

## 7.3 Configuring User Configuration Records

Figure 7-1 summarizes the configuration and build of a user configuration record.

Figure 7-1 User Configuration Record Configuration and Build



The inputs and outputs of the user configuration record configuration and build are as follows:

### User Configuration Record Outputs

The following are the outputs of the user configuration record configuration and build process:

#### User Configuration Record System Module File

The user configuration record system module file contains the compiled data for a user configuration record. The required extension for the user configuration record system module file is **.reloc**.

The following is a sample makefile rule for building a user configuration record system module file. As noted above, you are responsible for the certification of your user configuration record and for the qualification of any tools used to build it. The example shown here is designed to address build issues only and does not cover certification issues.

```
userCfgRgn1.reloc: userCfgRgn1.c
    $(CC) $(CFLAGS) -c $< -o userCfgRgn1.o
    $(LD) userCfgRgn1.o -Tdata $(USERCFGREGION_ADDR) -e 0xffffffff \
    -o userCfgRgn1.reloc
```

The entry point of 0xffffffff specified by the -e option is required by the system and must not be changed.

### User Configuration Record Inputs

The following are the inputs for the user configuration record build:

#### User Configuration Record Source or Object Files

You will need the source or object files that comprise your user configuration record. You must specify your source files as a dependency on the user configuration record object file in the user configuration record makefile:

```
userCfgRgn1.reloc: userCfgRgn1.c
```

#### Physical Address for the User Configuration Record Region

You need to specify the physical address at which the user configuration record region is to be located in the RAM. The value given here must match the value given in your CoreOSDescription file in the attribute **CoreOSDescription/HardwareConfiguration/PhysicalMemory/userConfigRecordRegion/@Base\_Address**. This value is represented in the makefile rule above by the

variable `$(USERCFGREGION_ADDR)`, which can be defined in the makefile or supplied on the command line.

## 7.4 Building a User Configuration Record

To build the user configuration record, use the following procedure:

**Step 1: Create a user configuration record makefile.**

A typical user configuration record makefile looks like this:

```
include $(WIND_BASE)/target/vThreads/config/make/Makefile.vars

userCfgRgn1.reloc: userCfgRgn1.c
    $(CC) $(CFLAGS) -c $< -o userCfgRgn1.o
    $(LD) userCfgRgn1.o -Tdata $(USERCFGREGION_ADDR) -e 0xffffffff \
    -o userCfgRgn1.reloc

%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<
```

**Step 2: Open the VxWorks 653 Development Shell.**

The VxWorks 653 build tools require a specific build environment which is provided by the VxWorks 653 Development Shell. To open the shell, from your program list, select:

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

**Step 3: Build the user configuration record.**

To build the user configuration record, run **make**:

```
make USERCFGREGION_ADDR=0x0f000000
```



# 8

## *Partition OSs*

[8.1 Understanding Partition OSs 47](#)

[8.2 Planning Partition OSs 48](#)

[8.3 Configuring a Partition OS 50](#)

[8.4 Building a Partition OS 55](#)

### **8.1 Understanding Partition OSs**

A partition OS provides the operating system for an application running in a partition. To conserve resources, partition OSs are located in a special kind of shared library called a system shared library. Each partition that uses a particular partition OS references the same system shared library. However, to maintain the required isolation between partitions, each partition must maintain a separate copy of all read/write sections of the partition OS, as well as the stack and heap space required to run the partition OS. A system shared library can only contain one partition OS and a partition can only access one system shared library. Every module must contain at least one partition OS. For information on applications, see [14. Applications](#). For information on partitions, see [15. Partitions](#). For information on shared libraries, see [9. Shared Libraries](#).

Because shared libraries and applications are built separately, the location of the partition OS code is not known when the applications is built. VxWorks 653 manages the lookup of partition OS code at run time. When the partition OS is

built, stubs files are created with routine stubs to which applications can link. As part of the partition OS build process, entry-point tables are created that correspond to the routine stubs in the stubs file. The entry-point tables are built into the partition OS. When the application is initialized, VxWorks 653 resolves the calls to the stubs to the real partition OS routines using the entry-point tables.

VxWorks 653 includes a partition OS called vThreads. It also includes a number of optional components that can add functionality to vThreads, and which allow for more than one flavor of the vThreads partition OS to be created and used. A module may contain more than one partition OS, and can have different partitions running different partition OSs, including different flavors of the vThreads partition OS and user-provided partition OSs based on the core OS interface library (COIL).

In addition to the components supplied with VxWorks 653, you can add your own code to the partition OS. You should keep in mind the certification implications of adding your own code to the partition OS.

Partition OSs are the responsibility of the platform provider. For information on development roles, see [3. Configuration System](#).

## 8.2 Planning Partition OSs

When planning a partition OS, you should consider the following issues:

### **What will be included with the partition OS?**

There are a number of components that can be added to a partition OS to provide added functionality. While all the optional components supplied with VxWorks 653 (with the exception of the C++ components) can be included in the partition OS, it can be useful to include those components in a shared library instead. Reasons for doing this include:

- If you have more than one flavor of partition OS in your module, but those partition OSs contain some of the same components, you can reduce the memory required by removing the common components from both partition OS variants and placing them in a shared library which can be accessed by partitions running either flavor of partition OS. Note that for this to work, the shared library must not make calls to the partition OS.

- If you have only one flavor of partition OS, but not all applications use all the components in the partition OS, you can move the components that are not used by all applications to shared libraries. Only the partitions whose applications use those components need to reference the shared libraries and set memory aside for their use.
- If you want to restrict the API available to certain applications, you can move that part of the partition OS to a shared library and deny that application's partition access to that shared library.

C++ support components are available but can only be included in application builds, not shared libraries or the partition OS or core OS.

For a cert system, a reduced set of components is available for shared libraries and a separate set of component binaries is used.

For a list of available components, see the *VxWorks 653 Configuration and Build Reference*.

#### **Will multiple interfaces be required?**

You may provide more than one interface to a partition OS. A shared library API definition can be used to select a subset of the routines available in a library for use in a particular application, and to map new routine names to the names in the library.

You can create interfaces to provide backward compatibility for applications written to use an earlier version of the partition OS or to provide cert and debug interfaces to the same partition OS. To define multiple interfaces for your partition OS, you create multiple interface definitions in the `Shared_Library_API` configuration document. You must define at least one interface.

You can also restrict the API available to an application by linking it to a limited version of the library's interface defined in the `Shared_Library_API` document. For example, in some cases you may want to provide a partition OS that only allows the application developer to use the APEX API. In this case you can effectively hide the native vThreads API by not including it in the library interface definition in the `Shared_Library_API` document.

## 8.3 Configuring a Partition OS

Figure 8-1 summarizes the configuration and build process for a partition OS.

The inputs and outputs of a partition OS build are as follows. Most of these are common to both partition OSs and regular shared libraries.

### Outputs of a Partition OS Build

The following are the outputs of a partition OS build:

#### Partition OS System Module File

The partition OS build process produces a system module (**.sm**) file for the shared library. This is an ELF file that can be included in a system image build. It is produced by a rule like this:

```
pos.sm: sslMain.o vThreadsComponent.o pos-ept.o pos.lds
    $(LD) $(LDFLAGS) -T pos.lds -o $@ $(filter %.o,$^)
```

#### Partition OS Stubs Files

The shared library stubs file contains the stubs that are required to build an application that uses the library. Normally, the stubs file has the same name as the shared library object file, but with the suffix **-stubs.o** in place of **.sm**. Thus a system shared library object file named **vThreads.sm** would have a stub file called **vThreads-stubs.o**.

If you have provided more than one interface for your shared library, you will need to provide a different stubs file for each interface. For example, you may provide separate cert and non-cert interfaces for your library. In this case you would need to build separate cert and non-cert stubs files:

```
posv1-stubs.c: pos-api.xml
    xmlgen --linkage --arch ppc --api-version "cert" --output-stubs $@ $<
```

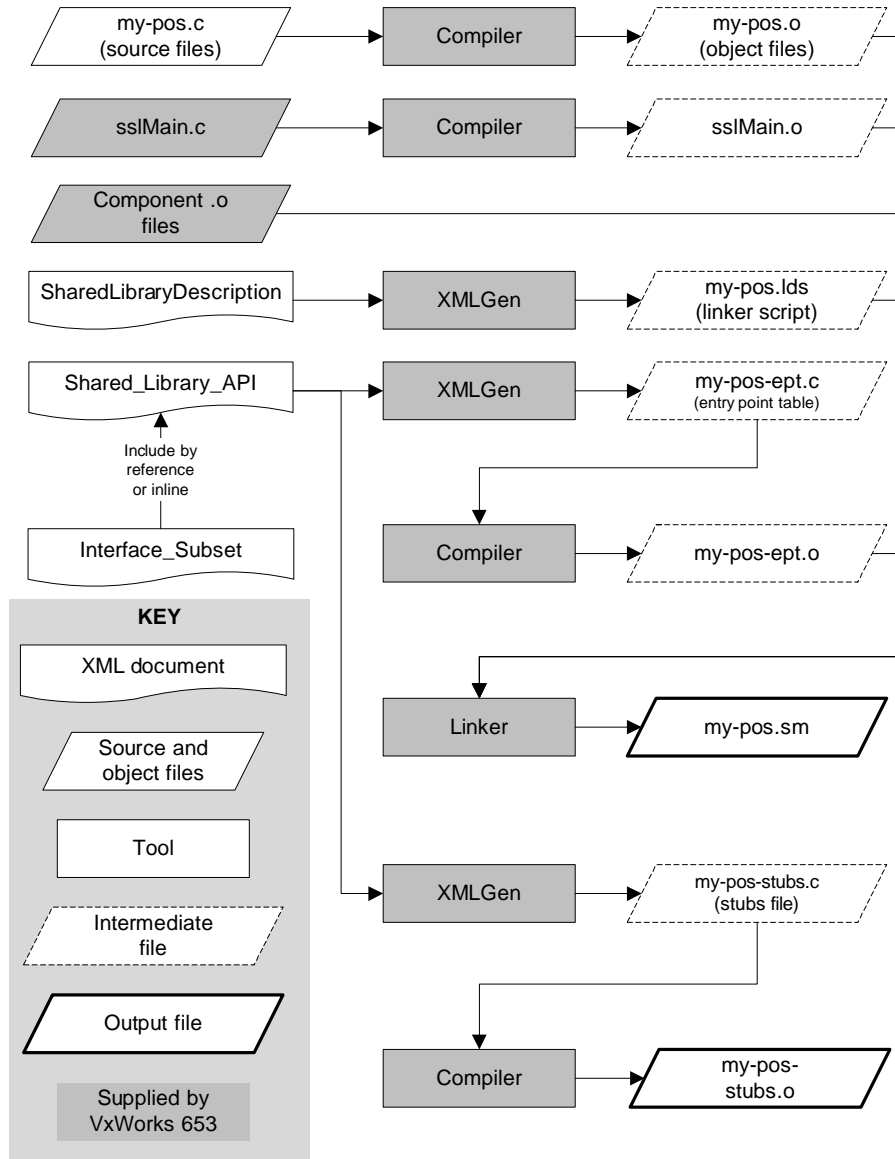
```
posv2-stubs.c: pos-api.xml
    xmlgen --linkage --arch ppc --api-version "non-cert" --output-stubs $@ $<
```

The names of the interfaces (e.g. “cert”) must match the names of the interfaces given in **Shared\_Library\_API/Interface/Version/@Name** in the **SharedLibraryInterface** document.

### Inputs of a Partition OS Build

The following are the inputs to a partition OS build:

Figure 8-1 Partition OS Configuration and Build



### SharedLibraryDescription Document

The SharedLibraryDescription document is an XML document conforming to the SharedLibraryDescription document type defined in the VxWorks 653 Configuration Schema.

The SharedLibraryDescription document contains configuration information for the partition OS in the following areas:

- virtual address of the partition OS
- memory requirements of the partition OS
- a flag that indicates that the shared library is a system shared library. A shared library that contains a partition OS must be flagged as a system shared library.

For examples of a SharedLibraryDescription document for a partition OS, see [22.5 Partition OS](#), p.185. For detailed information on creating or modifying the SharedLibraryDescription document, see the *VxWorks 653 Configuration and Build Reference*.

### Shared\_Library\_API Document

The Shared\_Library\_API document is an XML document conforming to the Shared\_Library\_API document type defined in the VxWorks 653 Shared Library API Schema.

This document defines one or more interfaces for the shared library. For examples of a Shared\_Library\_API document, see [22.5 Partition OS](#), p.185. For detailed information on creating or modifying the Shared\_Library\_API document, see the *VxWorks 653 Configuration and Build Reference*.

### Component API Interface\_Subset Documents

An Interface\_Subset document is an XML document conforming to the Interface\_Subset document type defined in the VxWorks 653 Shared Library API Schema. VxWorks 653 provides an Interface\_Subset definition document for the vThreads operating system. It is located at:

*installDir\vxworks653-2.2\target\vThreads\config\comps\xml\vthreads.xml*

The cert version of the interface is defined in **vthreads\_cert.xml** in the same location. For a list of components and their corresponding default Interface\_Subset documents, the *VxWorks 653 Configuration and Build Guide*.

If you are including components in your partition OS, and you want to make the routines in those components available to application developers, you must add these interfaces to your Shared\_Library\_API document as an Interface\_Subset

definition. This can be done inline, or by reference to an external Interface\_Subset document. VxWorks 653 includes Interface\_Subset documents for the components it supplies. They are located in:

```
installDir\vxworks653-2.2\target\vThreads\config\comps\xml
```

You may also create Interface\_Subset documents to describe all or part of an interface for your own library code. For detailed information on creating or modifying the Interface\_Subset document, see the *VxWorks 653 Configuration and Build Reference*.

### System Shared Library Initialization File

VxWorks 653 includes the system shared library initialization file **sslMain.c** which is required for building a system shared library. You must include **sslMain.o** as the first item in the dependency list for your partition OS in the partition OS makefile. You must also include a rule in the makefile to automatically compile **.o** files from **.c** files and a vpath statement to specify the location of **sslMain.c**:

```
vpath %.c $(WIND_BASE)/target/vThreads/config/comps/src

pos.sm: sslMain.o ...
    $(LD) $(LDFLAGS) -T pos.lds -o $@ $(filter %.o,$^)

%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<
```

### Partition OS Components

Your partition OS must include either the **vThreadsComponent.o** component that contains the vThreads partition OS or **coilComponent.o**, which supports user-supplied partition OS based on the COIL library.

Either the vThreads or COIL component must be included in your partition OS in the shared library makefile. You must also add a vpath statement to your partition OS makefile to specify the location of the component files:

```
vpath %.o $(WIND_BASE)/target/vThreads/lib/obj$(CPU)gnuvx

pos.sm: sslMain.o vThreadsComponent.o ...
```

Component object files are stored in different directories, depending on the target, the tool chain, and whether they are cert or non-cert. You must choose the directory that matches the build spec you are using. The directory name is formed using the following pattern:

“obj” + <CPU name> + <toolchain name> + “.” + [“cert” | “vx”]

As a result, for instance, the directory for the PPC604 CPU with the GNU tool chain and the debug build spec is objPPC604gnuvx.

### Additional Components

VxWorks 653 provides several components that provide additional functionality and can be included either in a partition OS or shared library.

All the component files to be included in your partition OS must be listed as dependencies for your partition OS in the partition OS makefile. You must also add a vpath statement to your partition OS makefile to specify the location of the component object files:

```
vpath %.o $(WIND_BASE)/target/vThreads/lib/obj$(CPU)gnuvx

pos.sm: sslMain.o vThreadsComponent.o vThreadsLibcMathComponent.o ...
```

### Partition OS Entry-Point Tables

An entry-point table for the partition OS is required to enable applications to link to partition OS routines. **XMLGen** builds the entry-point table from the information in the Shared\_Library\_API definition file. You must add a target to your partition OS makefile to build the entry-point table:

```
pos-ept.c: my-pos-Shared_Library_API.xml
xmlgen --linkage --output-entrypoints $@ $<
```

You also need to add the entry-point table to the dependencies of the partition OS in the partition OS makefile:

```
pos.sm: sslMain.o vThreadsComponent.o vThreadsLibcMathComponent.o \
pos-ept.o ...
```

### Partition OS Source Files

If you are adding your own code to the partition OS, your source files must be stated as dependencies on the partition OS in the partition OS makefile. The following example show the rule for building a partition OS with all of the common dependencies:

```
pos.sm: sslMain.o vThreadsComponent.o vThreadsLibcMathComponent.o \
pos-ept.o my-code.o ...
```

### Partition OS Linker Script

Because the memory configuration of a partition OS is specified by a black box defined in its SharedLibraryDescription document, a custom linker script is required to align the sections of the partition OS ELF file on the correct boundaries. The linker script can be generated from the SharedLibraryDescription document.



To cause the linker script to be built, you specify it as a target in the partition OS makefile and give the SharedLibraryDescription document as a dependency:

```
my-pos.lds: my-pos.xml
    xmlgen --ldScript --arch $(TOOLARCH) -o $$ $<
```

## 8.4 Building a Partition OS

To build a partition OS, use the following procedure:

8

### Step 1: Create a makefile for the partition OS.

A typical partition OS makefile looks something like this.

```
all: pos.sm pos-stubs.o

include $(WIND_BASE)/target/vThreads/config/make/Makefile.vars

vpath %.c    $(WIND_BASE)/target/vThreads/config/comps/src
vpath %.o    $(WIND_BASE)/target/vThreads/lib/obj$(CPU)gnuvx

pos.sm: sslMain.o vThreadsComponent.o pos-ept.o pos.lds
    $(LD) $(LDFLAGS) -T pos.lds -o $$ $(filter %.o,$^)

%.o: %.c
    $(CC) $(CFLAGS) -c -o $$ $<

pos-ept.c: pos-api.xml
    xmlgen --linkage --output-entrypoints $$ $<

pos-stubs.c: pos-api.xml
    xmlgen --linkage --arch $(TOOLARCH) --output-stubs $$ $<

pos.lds: hello-pos.xml
    xmlgen --ldScript --arch $(TOOLARCH) -o $$ $<
```

For other sample partition OS makefiles, see [22.5 Partition OS](#), p.185.

### Step 2: Open the VxWorks 653 Development Shell.

The VxWorks 653 build tools require a specific build environment which is provided by the VxWorks 653 Development Shell. To open the shell, from your program list, select:

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

**Step 3: Build the shared library.**

To build the shared library, run **make**, specifying the CPU to build for:

```
make all CPU=PPC604
```

# 9

## *Shared Libraries*

[9.1 Understanding Shared Libraries 57](#)

[9.2 Planning Shared Libraries 58](#)

[9.3 Configuring Shared Libraries 59](#)

[9.4 Building a Shared Library 65](#)

### **9.1 Understanding Shared Libraries**

If you want to share code between more than one application, that code can be placed in a shared library. Shared libraries in VxWorks 653 are not dynamically loaded. In order to comply with the requirements of DO-178B certification, all software must be loaded into fixed and non-overlapping sections of memory at boot time. All libraries are loaded into memory when the module is booted. All partitions that access a library have to set aside memory for the read/write sections of the library, as well as the stack and heap space required to run the library routines. Minimizing the memory footprint of a library therefore, depends on the correct analysis and configuration of the module as a whole at design time. For more information on applications, see [14. Applications](#).

You can add your own code to shared libraries, but in many cases, you will build shared libraries entirely from components supplied with VxWorks 653. This allows you to package VxWorks 653 functionality in the way that is optimal for your platform. In some cases, all the shared code required by the applications in a

module may be contained in the partition OS. However, if you have more than one partition OS in your module, and you have some shared code that you want all applications to be able to access, you can avoid duplicating it in multiple partition OSs and including it in a single shared library. Even if you have only one partition OS in your module, you may want to limit the libraries that some applications can access by separating that functionality into shared libraries that only some partitions can access.

Because shared libraries and applications are built separately, the location of the library code is not known when the applications is built. VxWorks 653 manages the lookup of shared library code at run time. When libraries are built, stubs files are created with routine stubs to which applications can link. As part of the library build process, entry-point tables are created that correspond to the routine stubs in the stubs file. The entry-point tables are built into the library. When the application is initialized, VxWorks 653 resolves the calls to the stubs to the real library routines using the entry-point tables.

A special kind of shared library called a system shared library is used to contain a partition OS. For information on partition OSs, see [8. Partition OSs](#).

Shared libraries are usually the responsibility of the platform provider. For information on development roles, see [3. Configuration System](#).

## 9.2 Planning Shared Libraries

When planning a shared library, you should consider the following issues:

### **How will the shared code be organized?**

There are a number of components that can be placed in a partition OS or a shared library. In most cases, it is appropriate to put all the components required by a partition into the partition OS. However, if you have more than one partition OS in your platform, you may want to place components required by partitions running different partition OSs into a single shared library to avoid duplication. Note that this is only possible if the components placed in the shared library do not depend on code in a particular partition OS.

#### Which binary components will be included?

For shared libraries, a small set of optional components is available. None are included by default. For a list of available components, see the *VxWorks 653 Configuration and Build Reference*.

C++ support components are available, but can only be included in application builds, not shared libraries or the core OS.

For a cert system, a reduced set of components is available for shared libraries and a separate set of component binaries is used.

#### Will multiple interfaces be required?

You can provide more than one interface to a shared library. A shared library interface can be used to select a subset of the routines available in a library for use in a particular application, and to map new routine names to the names in the library. You can create interfaces to provide backward compatibility for applications written to use an earlier version of the library or to provide cert and debug interfaces to the same library. To define multiple interfaces for your library, you create multiple interface definitions in the `Shared_Library_API` configuration document. You must define at least one interface.

9

## 9.3 Configuring Shared Libraries

[Figure 9-1](#) summarizes the shared library configuration and build process.

The inputs and outputs of the shared library configuration and build process are as follows:

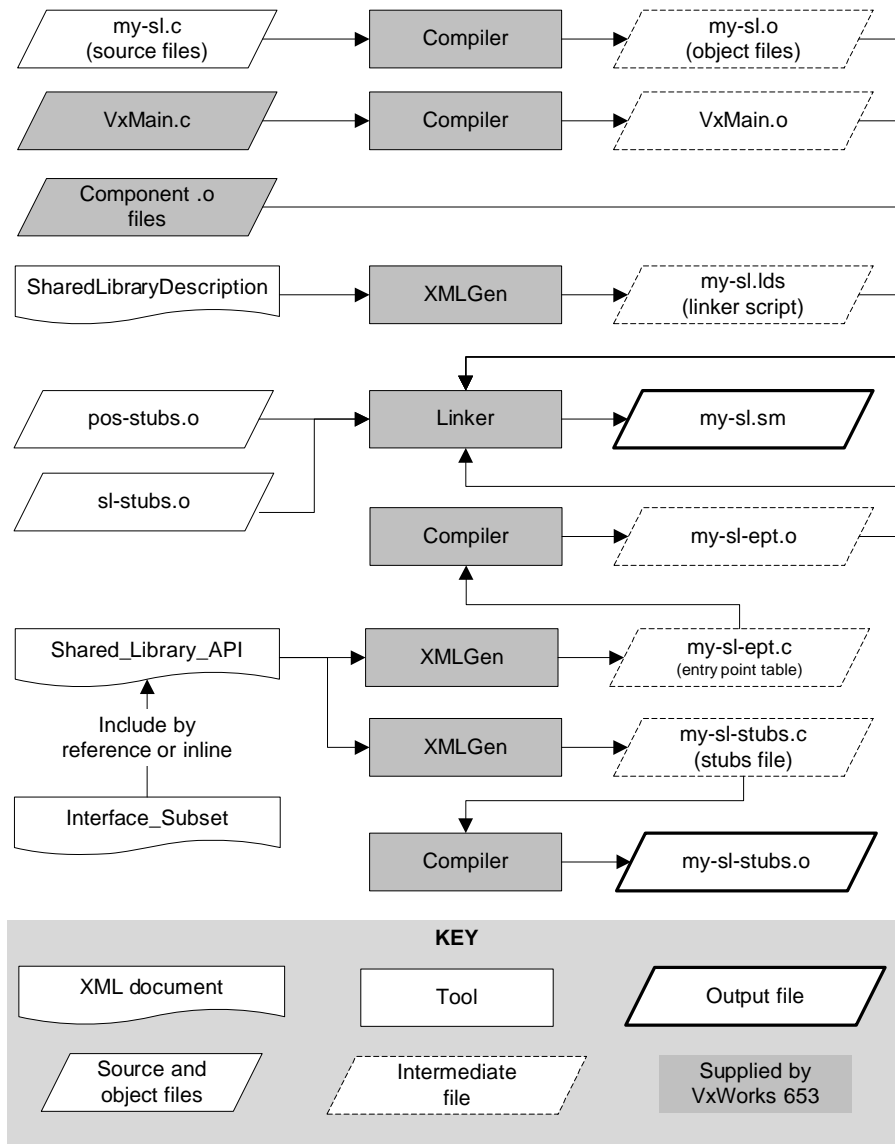
#### Outputs of the Shared Library Configuration and Build process

The outputs of the shared library build process are:

##### Shared Library System Module File

The shared library build process produces a system module (`.sm`) file for the shared library. This is an ELF file that can be included in a system image build. For information on building system images, see [21. System Images](#).

Figure 9-1 Shared Library Configuration



A typical build rule for a shared library system module file looks something like this:

```
sl.sm: vxMain.o my-code.o sl-stubs.o sl-ept.o sl.lds  
$(LD) $(LDFLAGS) -T sl.lds -o $$ $(filter %.o,$^)
```

The inputs to this rule are explained below.

### Shared Library Stubs Files

The shared library stubs file contains the stubs that are required to build an application that uses the library. Normally, the stubs file has the same name as the system module file, but with the suffix **-stubs.o** in place of **.sm**. Thus a system shared library object file named **my-sl.sm** would have a stub file called **my-sl-stubs.o**.

A typical build rule for a shared library stubs file looks like this:

```
slv1-stubs.c: sl-api.xml  
xmlgen --linkage --arch $(TOOLARCH) --output-stubs $$ $<
```

The inputs to this rule are explained below.

If you have provided more than one interface for your shared library, you will need to provide a different stubs file for each interface. For example, you may provide separate cert and non-cert interfaces for your library. In this case you would need to build separate cert and non-cert stubs files.

```
slv1-stubs.c: sl-api.xml  
xmlgen --linkage --arch $(TOOLARCH) --api-version "cert" /  
--output-stubs $$ $<  
  
slv2-stubs.c: sl-api.xml  
xmlgen --linkage --arch $(TOOLARCH) --api-version "non-cert" /  
--output-stubs $$ $<
```

The names of the interfaces (e.g. "cert") must match the names of the interfaces given in **Shared\_Library\_API/Interface/Version/@Name** in the SharedLibraryInterface document.

### Inputs to the Shared Library Configuration and Build Process

The following are the inputs to the shared library configuration and build process:

#### SharedLibraryDescription Document

The SharedLibraryDescription document is an XML document conforming to the SharedLibraryDescription document type defined in the VxWorks 653 Configuration Schema.

The SharedLibraryDescription document contains configuration information for the shared library in the following areas:

- virtual address of the shared library
- memory requirements of the shared library
- a flag indicating whether the shared library is a system shared library (a system shared library is used to contain a partition OS)

For examples of a SharedLibraryDescription document, see [22.7 Shared Library](#), p.196. For detailed information on creating or modifying the SharedLibraryDescription document, see the *VxWorks 653 Configuration and Build Reference*.

### **Shared\_Library\_API Document**

The Shared\_Library\_API document is an XML document conforming to the Shared\_Library\_API document type defined in the VxWorks 653 Shared Library API Schema.

This document defines one or more interfaces for the shared library. For examples of a Shared\_Library\_API document, see [22.7 Shared Library](#), p.196. For detailed information on creating or modifying the Shared\_Library\_API document, see the *VxWorks 653 Configuration and Build Reference*.

### **Component API Interface\_Subset Documents**

A Interface\_Subset document is an XML document conforming to the Interface\_Subset document type defined in the VxWorks 653 Shared Library API Schema.

If you are including components in your shared library, and you want to make the routines in those components available to application developers, you must add these interfaces to your Shared\_Library\_API document as an Interface\_Subset definition. This can be done inline, or by reference to an external Interface\_Subset document. VxWorks 653 includes Interface\_Subset documents for the components it supplies. You may also create Interface\_Subset documents to describe all or part of an interface for your own library code. For detailed information on creating or modifying the Interface\_Subset document, see the *VxWorks 653 Configuration and Build Reference*.

### **Library Initialization Files**

VxWorks 653 includes the library initialization file **vxMain.c** which is required for building a shared library. You must include **vxMain.c** as a dependency for your



shared library in the shared library makefile. The dependency is stated as a dependency on **vxMain.o**. Note that **vxMain.o** must be first item in the dependency list.

```
my-sl.sm: vxMain.o ...
```

You should provide a generic rule for compiling **.c** file to **.o** files using the variables defined in **Makefile.vars**.

```
%.o: %.c
    $(CC) $(CFLAGS) -c -o $$@ $<
```

You must also add a **vpath** statement to your shared library makefile to specify the location of **vxMain.c**:

```
vpath %.c $(WIND_BASE)/target/vThreads/config/comps/src
```

### Shared Library Entry-Point Tables

For each shared library, an entry-point table is required. **XMLGen** builds the entry-point table from the information in the **Shared\_Library\_API** definition file. You must add a target to your shared library makefile to build the entry-point table:

```
my-sl-ept.c: my-SharedLibraryInterface.xml
    xmlgen --linkage --output-entrypoints $$@ $<
```

You must also add the entry-point table to the dependencies of the shared library in the shared library makefile:

```
my-sl.sm: vxMain.o my-sl-ept.o ...
```

### Shared library Linker Script

Because the memory configuration of a shared library is specified by a black box defined in the **SharedLibraryDescription** document, a custom linker script is required to align the sections of the shared library ELF file on the correct boundaries. **XMLGen** generates the linker script from the **SharedLibraryDescription** document:

```
my-sl.lds: my-sl.xml
    xmlgen --ldScript --arch $(TOOLARCH) -o $$@ $<
```

You must also add the linker script to the dependencies of the shared library in the shared library makefile.

```
my-sl.sm: vxMain.o my-sl-ept.o my-sl.lds ...
```

## Shared Library Component Object Files

VxWorks 653 provides several components that provide packaged functionality that can be included in a shared library. All the component files to be included in your shared library must be listed as dependencies for your shared library in the shared library makefile.

```
my-sl.sm: vxMain.o vThreadsLibcMathComponent.o ...
```

You must also add a `vpath` statement to your shared library makefile to specify the location of the component object files:

```
vpath %.o $(WIND_BASE)/target/vThreads/lib/obj$(CPU)gnuvx
```

Component object files are stored in different directories, depending on the target, the tool chain, and whether they are cert or non-cert. You must choose the directory that matches the build spec you are using. The directory name is formed using the following pattern:

“obj” + <CPU name> + tool + [“cert” | “vx”]

so that, for instance, the directory for the PPC604 BSP with the GNU tool chain and the debug build spec is “objPPC604gnuvx”. However, the CPU name can be represented by the `$(CPU)` variable, giving “obj\$(CPU)gnuvx”

## Shared Library Source Files

If you are adding your own code to the shared library, your source files must be stated as dependencies on the shared library in the shared library makefile. The following example show the rule for building a shared library with all of the common dependencies:

```
my-sl.sm: vxMain.o my-sl.o my-sl.lds file1.o file2.o \  
vThreadsAmioComponent.o usrAmio.o usrAmioRedirect.o my-sl-ept.o
```

## Partition OS Stubs File

If the shared library depends on routines in the partition OS, you must provide the stubs files for the partition OS. Partition OS stubs files must be listed as dependencies for the shared library in the shared library makefile. Note that if a shared library depends on routines in a partition OS it can only be used in partitions that use that partition OS, or that use a partition OS that uses the same interface name and interface definition as the partition OS for which it was compiled.

### Shared Library Stubs File

If the shared library depends on routines in another shared library, you must provide the stubs files for all the libraries that the shared library accesses. Note that in order for the shared library to access the routines in the second library, it is necessary that the application's partition be configured to permit access to both libraries in the PartitionDescription document.

Shared library stubs files must be listed as dependencies for the shared library in the shared library makefile.

## 9.4 Building a Shared Library

To build the shared library, use the following procedure:

### Step 1: Create a makefile for the shared library.

A typical shared library makefile looks something like this.

```
all: sl.sm sl-stubs.o

include $(WIND_BASE)/target/vThreads/config/make/Makefile.vars

vpath %.c $(WIND_BASE)/target/vThreads/config/comps/src
vpath %.o $(WIND_BASE)/target/vThreads/lib/obj$(CPU)gnuvx

sl.sm: vxMain.o vThreadsComponent.o sl-ept.o sl.lds
$(LD) $(LDFLAGS) -T sl.lds -o $@ $(filter %.o,$^)

%.o: %.c
$(CC) $(CFLAGS) -c -o $@ $<

sl-ept.c: sl-api.xml
xmlgen --linkage --output-entrypoints $@ $<

sl-stubs.c: sl-api.xml
xmlgen --linkage --arch $(TOOLARCH) --output-stubs $@ $<

sl.lds: hello-sl.xml
xmlgen --ldScript --arch $(TOOLARCH) -o $@ $<
```

For other sample shared library and system shared library makefiles, see [22.7 Shared Library](#), p.196.

**Step 2: Open the VxWorks 653 Development Shell.**

The VxWorks 653 build tools require a specific build environment which is provided by the VxWorks 653 Development Shell. To open the shell, from your program list, select:

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

**Step 3: Build the shared library.**

To build the shared library, run **make**, specifying the CPU to build for:

```
make all CPU=PPC604
```

# 10

## *Shared Data Regions*

[10.1 Understanding Shared Data Regions 67](#)

[10.2 Planning a Shared Data Region 68](#)

[10.3 Configuring a Shared Data Region 68](#)

[10.4 Building a Shared Data Region 71](#)

### **10.1 Understanding Shared Data Regions**

A shared data region is an area of memory that can be accessed by more than one partition. A shared data region may be either a blank area of memory that applications can use to exchange data or a pre-compiled database that contains information that is used by more than one application. A shared data region that contains pre-compiled data is referred to as a loadable shared data region, since the database becomes part of the payload. A blank shared data region is referred to as a non-loadable shared data region, since it is not part of the payload.

Shared data regions contain shared data that can be used by multiple applications. For more information on applications, see [14. Applications](#).

Shared data regions may be part of a platform. For information on platforms, see [13. Platforms](#).

Access to shared data regions by applications is configured as part of the partition configuration. For information on partitions, see [15. Partitions](#).

Shared data regions are usually the responsibility of the system integrator. They may also be created by the platform provider or the application developer. For information on development roles, see [3. Configuration System](#).

## 10.2 Planning a Shared Data Region

VxWorks 653 provides for the setting aside of shared data regions, and for the loading of precompiled databases as part of the module payload. The structure and access mechanism for both loadable and non-loadable shared data regions are up to the application developers who will be sharing the data.

Developers of applications that share a shared data region must communicate with each other to determine the structure and method of access for the shared data region. The system integrator must communicate with the application developers and the platform provider to determine the number and size of the shared data regions to be provided by the platform.

## 10.3 Configuring a Shared Data Region

[Figure 10-1](#) summarizes the configuration and build of a loadable shared data region.

The inputs and outputs of the shared data region configuration and build are as follows:

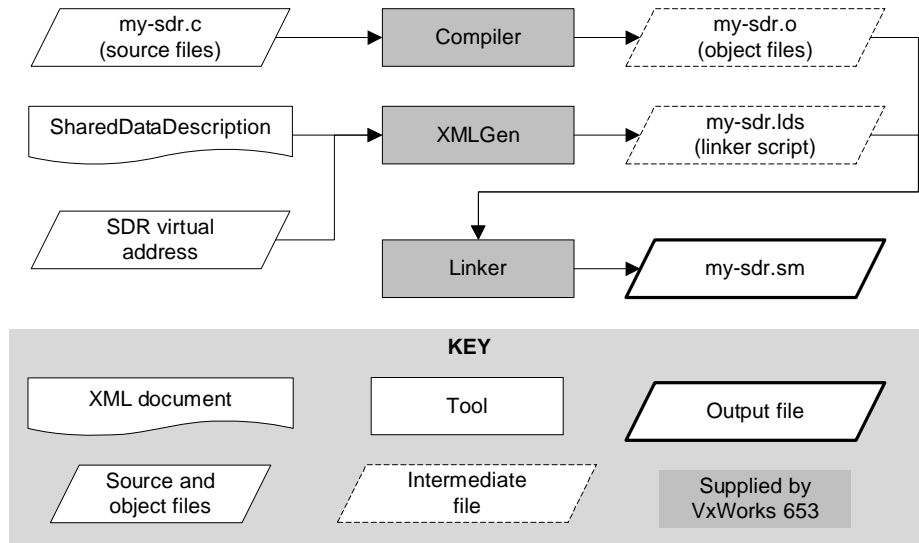
### Shared Data Region Outputs

The following are the outputs of the shared data region configuration and build process:

#### Shared Data Region System Module File

The shared data region system module file contains the compiled database for a loadable shared data region. If the shared data region is non-loadable, that is, it consists simply of an area of memory that is shared between applications, then no

Figure 10-1 **Shared Data Region Configuration and Build**



system module is created and no build is required. The following rule builds a shared data region system module file:

```
sdr.sm: sdr.o sdr.lds
$(LD) $(LDFLAGS) -T sdr.lds -o $$@ $(filter %.o,$^)
```

### Shared Data Region Inputs

The following are the inputs to the user shared data region configuration and build process:

#### SharedDataDescription Document

The SharedDataDescription document is an XML document conforming to the SharedDataDescription document type defined in the VxWorks 653 Configuration Schema.

This document contains configuration information for the shared data region in the following areas:

- cache policy for the shared data region
- data type for the shared data region

- size of the shared data region
- access rights to the shared data region for the core OS
- virtual address of the shared data region

For an example of a SharedDataDescription document, see [22.5.4 Partition OS with Shared Data Region](#), p.189. For detailed information on creating or modifying the SharedDataDescription document, see the *VxWorks 653 Configuration and Build Reference*.

### Virtual Address for the Shared Data Region

You need to know the virtual address at which the shared data region is to be located in the module. This information is contained in the Module document for the module. If you have access to a Module document, you can use it to provide the address to the build process. If you do not have a Module document, you can specify the virtual address via the **LD\_FLAGS\_EXTRA** variable in the shared data makefile:

```
SD_ADDR = 0
LD_FLAGS_EXTRA = -Tdata $(SD_ADDR)
```

### Shared Data Region Source or Object files

You will need the source or object files that comprise your shared data region. The format and access to the shared data regions is entirely up to the developer of the shared data region. You must specify your source files as a dependency on the shared data region object file in the shared data region makefile:

```
my-sdr.sm: my-sdr.o ...
```

### Shared Data Region Linker Script

A custom linker script is required to align the sections of the shared data region ELF file on the correct boundaries. The linker script can be generated from the SharedDataDescription document using commands contained in **Makefile.rules**.

To cause the linker script to be built, you specify it as a target in the shared library makefile and give the SharedLibraryDescription document as a dependency:

```
my-sdr.lds: my-sdr.xml
    xmlgen --ldScript --arch $(TOOLARCH) --address $(SD_ADDR) -o $@ $<
```

You must also add the linker script to the dependencies of the shared data region in the shared data makefile:

```
my-sdr.sm: my-sdr.o my-sdr.lds
```



## 10.4 Building a Shared Data Region

To build the shared data region, use the following procedure:

**Step 1: Create a shared data region makefile.**

The makefile for your shared data region will look something like this:

```
all: my-sdr.sm

include $(WIND_BASE)/target/vThreads/config/make/Makefile.vars

my-sdr.sm: my-sdr.o my-sdr.lds
    $(LD) $(LDFLAGS) -T my-sdr.lds -o $$ $(filter %.o,$^)

%.o: %.c
    $(CC) $(CFLAGS) -c -o $$ $<

my-sdr.lds: my-sdr.xml
    xmlgen --ldScript --arch $(TOOLARCH) --address $(SD_ADDR) -o $$ $<
```

If the shared data region contains pointers (and therefore needs to be located at a specific address), set **SD\_ADDR** to the value of **SharedData/SharedDataDescription/@VirtualAddress** in the SharedDataDescription file. Otherwise the value of **SD\_ADDR** is 0.

For an example of a shared data region makefile, see [22.5.4 Partition OS with Shared Data Region](#), p.189.

**Step 2: Open the VxWorks 653 Development Shell.**

The VxWorks 653 build tools require a specific build environment which is provided by the VxWorks 653 Development Shell. To open the shell, from your program list, select:

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

**Step 3: Build the shared data region.**

To build the shared data region, run **make**, specifying the CPU to build for:

```
make all CPU=PPC604
```



# 11

## *Shared I/O Regions*

[11.1 Understanding Shared I/O Regions](#) 73

[11.2 Planning Shared I/O Regions](#) 74

[11.3 Configuring Shared I/O Regions](#) 74

[11.4 Building a Shared I/O Region](#) 75

### **11.1 Understanding Shared I/O Regions**

A shared I/O region represents an I/O device on the target (such as an LED) that the core OS makes available to be shared by applications in the module.

Shared I/O regions make target hardware I/O devices available to one or more applications. For more information on applications, see [14. Applications](#).

Target I/O regions are shared as part of the configuration of the core OS. For information on the core OS, see [6. Core OS](#). Applications access to shared I/O regions is part of the partition configuration. For information on partitions, see [15. Partitions](#).

Creating Shared I/O regions is the responsibility of the platform provider. Permitting applications to access shared I/O regions is the responsibility of the system integrator. For information on development roles, see [3. Configuration System](#).

## 11.2 Planning Shared I/O Regions

The platform provider, application developer, and system integrator each have a role in planning shared I/O regions. The platform provider needs to tell the application developer and system integrator what shared I/O regions are available on the platform. The application developer needs to tell the system integrator which shared I/O regions are used by the application so that the system integrator can configure access to the shared I/O region for the application.

## 11.3 Configuring Shared I/O Regions

There are two parts to the configuration of a shared I/O region, one belonging to the platform provider and one belonging to the system integrator.

The platform provider defines the memory area of the shared I/O region and determines whether that area is shared or not. This is done by adding a **CoreOSDescription/HardwareConfiguration/sharedIO** element to the CoreOSDescription document for the core OS. For information on configuring the core OS see [6. Core OS](#). For detailed information on creating or modifying the CoreOSDescription document, see the *VxWorks 653 Configuration and Build Reference*.

The system integrator configures the address and access rights for the shared I/O region within the module by writing a SharedIODescription document. The SharedIODescription document is an XML document conforming to the SharedIODescription document type defined in the VxWorks 653 Configuration Schema.

For detailed information on creating or modifying the SharedIODescription document, see the *VxWorks 653 Configuration and Build Reference*.

## **11.4 Building a Shared I/O Region**

Shared I/O regions are not built because they represent hardware on the target. Shared I/O configuration information is used as part of the configuration and build of the core OS and partitions.



# 12

## ACE

*(Agent for the Certified Environment)*

12.1 Understanding ACE 77

12.2 Planning ACE 78

12.3 Configuring ACE 78

12.4 Building ACE 82

## 12.1 Understanding ACE

The agent for the certified environment (ACE) allows you to load the WDB (Wind River debug) agent separately from the core OS. This allows you to certify the core OS independently of whether the WDB agent is included in the module. ACE is supported only when the core OS is built with the cert subset. If you are using a debug version of the core OS, you should not configure your module to use ACE.

Since ACE limits the range of debugging options available, the platform provider should generally ship both a debug and cert version of the core OS so that application developers can do the majority of their debugging using the full debugging capabilities of VxWorks 653 with the debug version of the core OS, and use ACE only for final checking in the cert environment.

ACE is a debugging facility that works with a certified core OS. For information on the core OS, see [6. Core OS](#). For information on debugging, see the *Wind River Workbench Users's Guide*.

ACE is part of a platform. For information on platforms, see [13. Platforms](#).

ACE is the responsibility of the platform provider. Including ACE in a module is the responsibility of the system integrator. For information on development roles, see [3. Configuration System](#).

## 12.2 Planning ACE

The overall configuration of ACE is fixed and cannot be altered by the user. The only choice that the user has to make concerns the communication channel between ACE and the host. Two methods are available: network and serial. The network communications method requires that the IP address of the target be written into the ACE configuration at build time. This means that ACE must be rebuilt for each target. If application developers or the system integrator need to use ACE, they can either use a version that has been configured for serial communication or they can establish a private network around the target for debugging. If the platform provider is providing ACE as part of the platform, it is generally advisable to provide a version configured for serial communication and/or to provide a version configured for network communication that uses an IP address in the private network range (for example 192.168.x.x).

## 12.3 Configuring ACE

[Figure 12-1](#) summarizes the configuration and build of ACE. For an example of building ACE, see [22.4.3 Module OS with ACE](#), p.180.

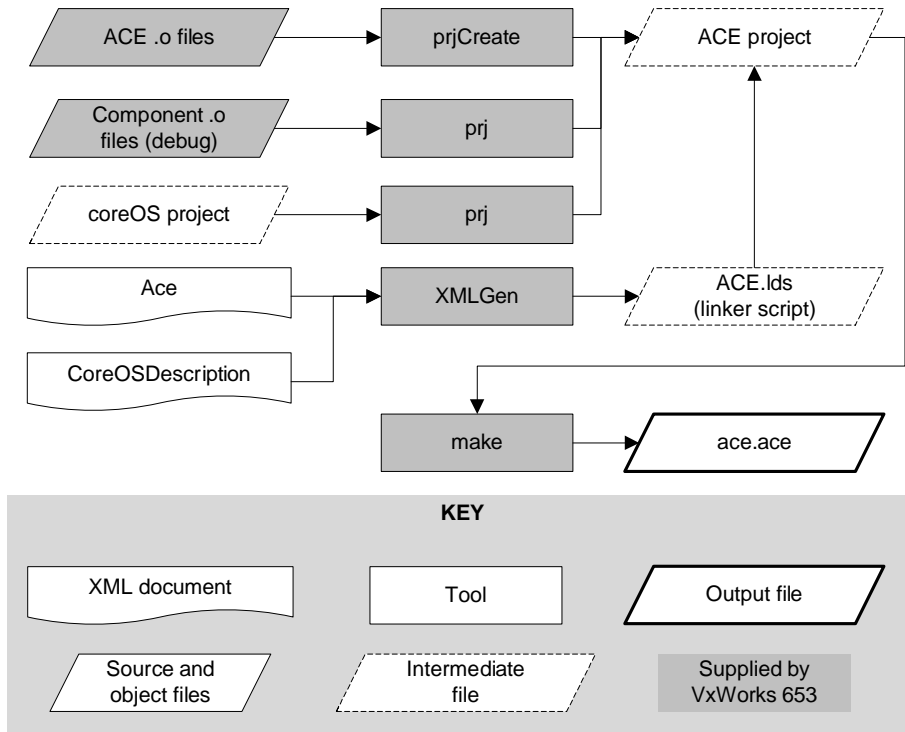
The inputs and outputs of the ACE build process are as follows:

### Outputs of the ACE Build

The outputs of the ACE build process are:



Figure 12-1 Configuration of ACE



### ACE System Module

The ACE system module (.ace) file is the ACE binary that is loaded on the target.

### Inputs to the ACE Build

#### An ACE Project

To build ACE you must create an ACE project. You can create an ACE project using the **prjCreate** command. For example:

```
prjCreate -domtype ace -prj my-acedir -build PPC604gnu.cert -name my-ace
```

The **prjCreate** command requires the following options:

#### -domtype

Specifies the type of project to create. For ACE, this is "ace".

**-prj**

Specifies the directory in which the project will be built. This value will be used by subsequent commands to locate the project files.

**-build**

Specifies the build spec to be used to build ACE. This must be a cert build spec, and may not be a simulator build spec.

**-name**

Specifies the name of the ACE project. This must match **/Module/Ace/@Name**.

For additional **prjCreate** options, see the reference entry for **prjCreate**.

### The Core OS Project

The ACE project needs to know the name and project location of the core OS that this ACE is to work with. For information on creating a core OS project, see [6. Core OS](#). You can provide this information to the ACE project by adding **prj projBuildTagValueSet** commands to the ACE build rule in your ACE makefile:

```
prj projBuildTagValueSet -prj my-acedir \  
COREOS_DIR my-kernelldir/PPC604gnu.cert  
prj projBuildTagValueSet -prj my-acedir COREOS_NAME coreOS.sm
```

Note that the core OS directory is the directory under the core OS project directory with the directory name corresponding to the build spec used to build the core OS. The core OS name must match the value of **/CoreOSDescription/@KernelName** in your *my-coreOS.xml* file, and must have an extension of **.sm**.

### Ace Configuration Document

The build process needs the memory black box information for ACE in order to correctly configure memory for ACE. This information is contained in the Ace configuration document (**hello-ace.xml**). Since the size of ACE is fixed, the content of the ACE black box is fixed also:

```
<Ace  
  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/Module.xsd"  
  name="ace">  
    <MemorySize  
      MemorySizeBss="0x10000"  
      MemorySizeText="0x30000"  
      MemorySizeData="0x1000"  
      MemorySizeRoData="0x1000"/>  
    </MemorySize>  
  </Ace>
```

The value of the ace name can be changed, provided that all references to it are changed accordingly, however it is recommended that it be left as “ace”.

### CoreOSDescription Document

In order to align correctly with the core OS, the ACE build needs the core OS memory black box information contained in the CoreOSDescription configuration document for the target core OS. For information on the core OS configuration, see [6. Core OS](#).

### Linker Script for ACE

Because memory sizing and allocation in VxWorks 653 is based on memory black box information contained in configuration files, all VxWorks 653 components require a linker script to correctly align ELF file sections on the boundaries specified by the black boxes. You must create a linker script that will be used to link ACE. **XMLGen** can generate this script for you from the Module configuration document. To cause the linker script to be built and used:

```
xmlgen --ldScript --arch ppc --blackbox my-ace -o my-acedir/my-ace.lds \
my-ace.xml my-coreos.xml
```

The **XMLGen** command requires the following parameters:

#### **--ldScript**

Specifies the creation of a linker script. This option must be first in the list.

#### **--arch**

Specifies the target architecture. This must match the architecture of the core OS. The simulator does not support ACE.

#### **-o**

Specifies the output file. The output file must be placed in the project directory created by **prjCreate**. The name of the output file must match the value of the **/Module/Ace/@Name** attribute and must have a **.lds** extension.

You must specify both the CoreOSDescription document and the ACE configuration document on the **XMLGen** command line:

Note that the name given in **Ace/@Name**, the name given in the **--blackbox** parameter, and the name given in the **-name** parameter of the **prjCreate** command that created the ACE project must all be the same.

## ACE Communication Components

You can configure your ACE for either network or serial communication. Since network communication requires configuring ACE with the IP address of the target, you should choose an address in the private network range (for example 192.168.0.1) so that the application developer or system integrator can set up a private network around the target, allowing it to be accessed at the pre-compiled address.

If you are configuring ACE for network communication:

1. Add the `INCLUDE_WDB_COMM_MINI_UDP` component to the ACE project.
2. Add the `INCLUDE_MINIMAL_MV_END` component to the core OS project (on supported BSPs only).
3. Set the IP address parameter `WDB_MINI_UDP_IP_ADDR` to your target IP address "xxx.xxx.xxx.xxx" (include the quotation marks):

```
prj domParameterValueSet -p my-acedir WDB_MINI_UDP_IP_ADDR "192.168.0.1"
```

If you are configuring ACE for serial communications:

1. Add `INCLUDE_WDB_COMM_SERIAL` to the ACE project.
2. Set the parameter `WDB_TTY_CHANNEL` to an appropriate serial I/O channel (the default is 0).

## 12.4 Building ACE

### Step 1: Create an ACE project makefile.

You will need a makefile to make an ACE project. The ACE project will contain a makefile that will build ACE. Here is a typical ACE project makefile:

```
ace:
prjCreate -domtype ace -prj my-acedir -build $(CPU)gnu.cert -n my-ace
prj projBuildTagValueSet -prj my-acedir COREOS_DIR my-kernelldir/
$(CPU)gnu.cert
prj projBuildTagValueSet -prj my-acedir COREOS_NAME coreOS.sm
xmlgen --ldScript --arch ppc -o my-acedir/my-ace.lds my-ace.xml
prj domComponentAdd -prj my-acedir INCLUDE_WDB_COMM_SERIAL
prj domParameterValueSet -prj my-acedir WDB_TTY_CHANNEL 0
```

**Step 2: Open the VxWorks 653 Development Shell.**

The VxWorks 653 build tools require a specific build environment which is provided by the VxWorks 653 Development Shell. To open the shell, from your program list, select:

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

**Step 3: Run the ACE project makefile.**

To create the ACE project, run the ace project makefile:

```
make
```

**Step 4: Build ACE.**

To build ACE, run the makefile created by the ACE project, specifying the CPU to build for:

```
make -C my-acedir CPU=PPC604
```



# 13

## *Platforms*

13.1 Understanding Platforms 85

13.2 Planning a Platform 86

13.3 Building a Platform 87

13.4 Packaging a Platform 87

### 13.1 Understanding Platforms

A platform is the basis for developing an ARINC 653 module. Designing the platform is the responsibility of the platform provider. A platform may be designed to be generic, to support many different systems, or to meet the needs of one specific module.

A platform includes the following:

- A core OS. A platform may include more than one version of the core OS. It will usually include a full core OS with debugging information (debug) and a certifiable subset of the core OS (cert). It may also include routines developed by the platform provider, such as drivers or health monitor event and notification handlers. The debug version may include development tools such as the target agent and the target shell. For more information on the core OS, see [6. Core OS](#).

- ACE, the agent for the certified environment. ACE is optional and is supported only with a cert core OS. For information on ACE, see [12. ACE](#).
- One or more partition OSs based either on vThreads or COIL. A partition OS may also contain common code that is needed by all applications that use a particular partition OS. For more information on partition OSs, see [8. Partition OSs](#).
- Zero or more shared libraries containing common code that can be used by more than one partition. For more information on shared libraries, see [9. Shared Libraries](#).
- Zero or more user configuration records. For information on user configuration records see [6. Core OS](#).
- Zero or more loadable shared data regions, and/or the configuration of zero or more non-loadable shared data regions. For more information on shared data regions, see [10. Shared Data Regions](#).
- Configuration of zero or more shared I/O regions. For information on shared I/O regions, see [11. Shared I/O Regions](#).

A platform is the foundation on which a module is built. For information on modules, see [2. Understanding VxWorks 653](#) and [19. Modules](#).

The platform is the responsibility of the platform provider. For information on development roles, see [3. Configuration System](#).

## 13.2 Planning a Platform

In planning a platform you may need to consider the following issues:

### **Which partition operating systems will be provided?**

VxWorks 653 allows for multiple partition OSs to be used in a single module. Partition OS choices include different combinations of vThreads partition OS with different components (such as certified and debug versions) as well as user-defined partition OSs based on the COIL library.

For information on components that can be added to a partition OS, see the *VxWorks 653 Configuration and Build Reference*.



For more information on COIL, see the *VxWorks 653 Programmer's Guide*.

**Which shared libraries will be provided?**

In order to reduce memory requirements, commonly used code can be placed in a shared library. If you provide shared libraries, you must also provide linkage information so that applications can link to the shared code. Linkage information will be generated from the Shared\_Library\_API document that you will write as part of configuring your shared library.

For information on components that can be included in a shared library, see the *VxWorks 653 Configuration and Build Reference*.

## 13.3 Building a Platform

Building a platform consists of building each of the components in your platform. For information on building each of the components, see the appropriate topics.

13

## 13.4 Packaging a Platform

Once your platform is complete it should be packaged for delivery to the system integrator and application developers. To package your platform, choose the appropriate packaging method (.zip file, tar ball, install package, etc.) and include the following:

- core OS system module (.sm) file (including different versions of the core OS, if you are supplying separate debug and cert versions of the core OS)
- CoreOSDescription file
- ACE system module (.sm) file (including different versions of ACE if you are providing both network and serial versions)
- ACE configuration file
- RAM and ROM payload boot loader code files:

- **payloadObjs\_ram.o**
- **payloadObjs\_rom.o**
- partition OS system module (**.sm**) files for each partition OS
- shared library system module (**.sm**) files for each shared library
- shared library stubs (**-stubs.o**) files for each partition OS and shared library
- SharedLibraryDescription (**.xml**) documents for each partition OS and shared library (the Shared\_Library\_Interface documents are not needed for build purposes, but may be useful to the application developer as a means of discovering the interface to the shared libraries)
- data region system module (**.sm**) files
- SharedDataDescription documents
- user configuration record **.reloc** files
- any documentation of your core OS or shared libraries that will be needed by the system integrator or the application developer

# 14

## *Applications*

14.1 Understanding Applications 89

14.2 Planning Applications 90

14.3 Configuring Applications 92

14.4 Building an Application 97

14.5 Packaging an Application 98

### 14.1 Understanding Applications

An application is a process or set of processes that runs inside a partition. An application may be developed specifically for a particular module, or it may be a generic application designed to run on any platform which supports an API that is supported in VxWorks 653, such as APEX, POSIX, or vThreads. For information on partitions, see [15. Partitions](#). For information on programming applications, see the *VxWorks 653 Programmers Guide*.

An application runs in a protected environment and is not directly aware of other applications running in the module, though it may be able to communicate with other modules using ARINC ports or shared data regions. For more information on the application's view of its environment, see [5. Memory](#). For information on ARINC ports, see [16. Ports and Channels](#) and the *VxWorks 653 Programmer's Guide*.

You can write your application using VxWorks 653 or you may bring in an externally written application that was written to run on an API provided by the partition OS. You can then bring it into a VxWorks 653 module by building it as a VxWorks 653 application. Applications intended for certified systems are commonly written to the APEX API.

Configuring an application to run in a VxWorks 653 module means providing information about the application that is required to integrate the application into the module. This includes memory requirements, APEX port usage, health monitoring requirements, scheduling requirements, library/API dependencies, and shared resource requirements.

The health of applications is monitored by the health monitor. For information on the health monitor, see [18. Health Monitor](#).

The execution of applications is controlled by schedules. For information on schedules, see [17. Schedules](#).

Applications are the responsibility of the Application developer. For information on development roles, see [3. Configuration System](#).

## 14.2 Planning Applications

When planning the configuration of an application, you must consider the resources available in the system and any constraints imposed by the system configuration. The issues you need to consider include the following:

### **How does the module configuration affect my application?**

If you are creating a generic application, you will only need to consider the constraints of the partition OS API for which your application is written. You will need to document the memory requirements, time requirements, library usage, port usage, and any access to shared resources required by your application so that the system integrator who builds your application into a module will be able to configure the module correctly to receive it.

If you are creating an application for a specific module, you will need to determine if there are any restrictions imposed on your application by the memory requirements, time requirements, library availability, port usage, and available shared resources in the proposed module. The system integrator may have

provided this information in the form of a Module configuration document along with configuration documents for other parts of the module. For details on the meaning of the various settings in this file, see the *VxWorks 653 Configuration and Build Reference* and [19. Modules](#).

#### **Does my application require binary components?**

If you want to add Wind River-supplied binary components to your application (for instance, because those components are used by this application alone, or because they are not suitable for inclusion in a shared library), you should add these components to the application's build. See [4. Build System](#) for details.

#### **What if my application is written in C++?**

VxWorks 653 supports applications written in C++. However, the C++ support components must be added separately to each application, by adding the C++ components to the application's build. Applications cannot access the C++ support components in a shared library.

#### **Will my application need to use ARINC ports?**

Applications can use ARINC ports to communicate with other applications in the module or with the outside world. In the application configuration, you must configure the ports that your application needs to send or receive data. Configuring the channels that connect ports to one another is the job of the system integrator.

You must make sure that your source and destination ports are correctly configured to communicate with the respective destination and source ports in their home module and in other modules.

#### **Does my application require a process health monitor?**

VxWorks 653 provides a comprehensive health monitoring system that routes messages and events to the handlers at the appropriate level. For events and messages that you want to handle at the application level, you should provide your own health monitor routine and register that routine with the VxWorks 653 health monitor system as the process health monitor for your application. For more information on health monitoring, see [18. Health Monitor](#).

#### **Does my application involve periodic processes?**

VxWorks 653 applications share time with other applications on a schedule determined by the system integrator. An application cannot run except in its designated time slot, and cannot exceed its time allotment. If your application

involves a periodic process, that is, one that must run on a regular schedule in real-world time, the system integrator will need to make sure that the application is always in its scheduled window when the periodic process is due to run, and that enough time remains in the application's time allotment for the periodic process to complete. You must inform the system integrator of the period and duration of any periodic processes in your application. For more information on schedules and periodic processes, see [17. Schedules](#).

#### **Does my application depend on health monitor notifications?**

The system integrator is responsible for configuring partitions to allow health monitor notifications to be handled on their behalf. The system integrator needs to know if application developers have made use of, or relied upon, the existence and operation of notification handlers in the core OS. For information on health monitoring, see [18. Health Monitor](#).

## **14.3 Configuring Applications**

[Figure 14-1](#) shows the process of configuring and building an application.

The inputs and outputs of the application configuration and build process are as follows:

#### **Outputs of the Application Build Process**

The outputs of the application build process are:

##### **Application System Module File**

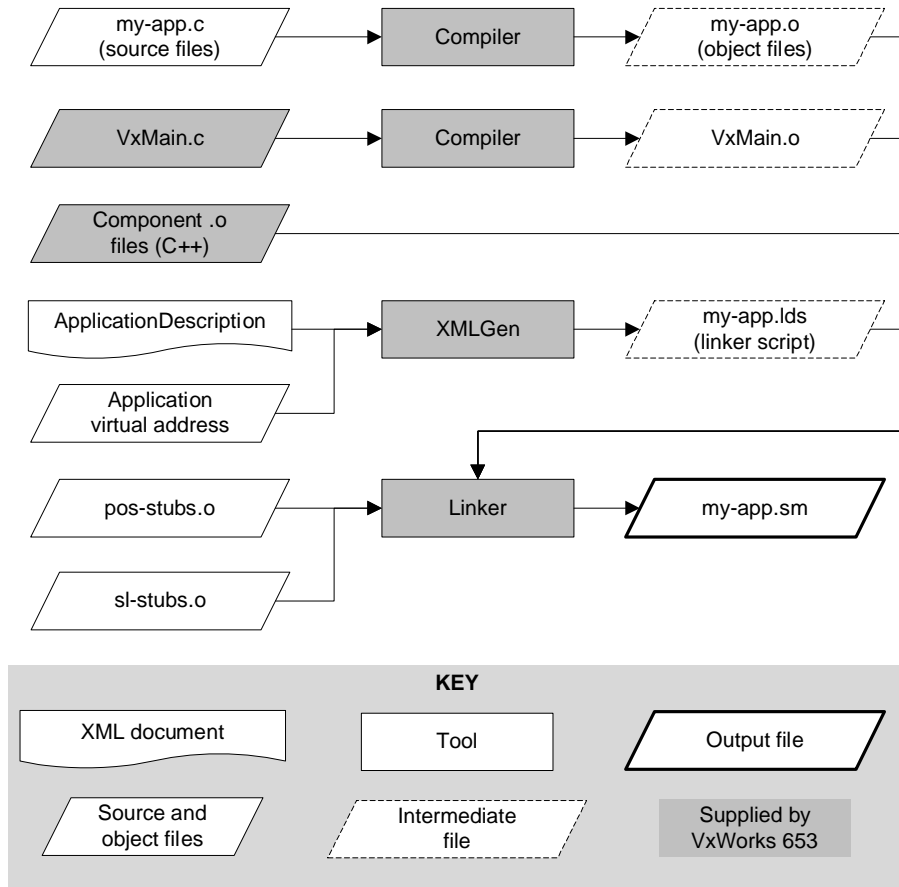
The application system module (.sm) file is the application binary that can be included in a system image. The following is an example of a rule that builds an application system module file:

```
my-app.sm: vxMain.o my-app.o pos-stubs.o my-app.lds
    $(LD) $(LDFLAGS) -T my-app.lds -o $$@ $(filter %.o,$^)
```

#### **Inputs to the Application Build Process**

The following are the inputs to the application build process:

Figure 14-1 Application Configuration and Build



### ApplicationDescription Document

The ApplicationDescription document is an XML document conforming to the ApplicationDescription document type defined in the VxWorks 653 Configuration Schema.

This document contains configuration information for the application in the following areas:

- memory requirements of the application

- ARINC port configuration of the application

For examples of an ApplicationDescription document, see [22.6 Application](#), p.191. For detailed information on creating or modifying the ApplicationDescription document, see the *VxWorks 653 Configuration and Build Reference*.

### Application Source or Object Files

You may compile your application to an object (.o) file as a separate step. Alternatively, you can use your application source code and have it compiled as part of the build process. Your application files must be listed as dependencies on you application in the application makefile. The dependencies must be stated as .o files, whether or not you are providing .c or .o files. If you supply .c files, they will be compiled to .o files by the rules in the application makefile. You will also need to specify the path to your .c files with a vpath statement in the application makefile.

By default, the application initialization routine (contained in **vxMain.o**) assumes that the application's init routine is called **usrAppInit()**. If your application has a different init routine, you must tell **vxMain.o** about it by redefining the preprocessor macro **USER\_APPL\_INIT**:

```
CFLAGS_EXTRA = "-DUSER_APPL_INIT=myInitFunc() "
```

If you have provided a custom init routine that requires parameters to be passed to it when the application is started, you can specify them here:

```
CFLAGS_EXTRA = "-DUSER_APPL_INIT=myInitFunc(1, 0, FALSE) "
```

### Application Linker Script

Because the memory configuration of an application is specified by a black box defined in the ApplicationDescription document, a custom linker script is required to align the sections of the application ELF file on the correct boundaries. The linker script can be generated from the ApplicationDescription document using **XMLGen**. A rule to build a linker script looks like this:

```
my-app.lds: my-app.xml  
    xmlgen --ldScript --arch $(TOOLARCH) --address $(PARTADDR) -o $@ $<
```

The linker script must also be listed as a dependency for the application in the application makefile.

### Partition Virtual Address

All applications in a module must be linked using the same partition virtual address. This value must match the value specified in **CoreOSDescription/KernelConfiguration/@partitionVirtualAddress** in the CoreOSDescription



document. This value may have been communicated to you by the platform provider or the system integrator. To supply the partition virtual address you must supply a value for the **\$(PARTADDR)** variable (as used in the rule above that generates the application linker script). You can supply this value in the makefile or on the command line when you run **make**.

### **vxMain.o**

**vxMain.o** provides initialization for the libraries used by the application, calls any C++ constructors, and calls the application's init routine. VxWorks 653 includes a suitable **vxMain.c** file, or you can provide your own. **vxMain.o** must be the first item in the dependency list for the application in the application makefile.

You must specify the location of the source files for **vxMain.c** using a **vpath** statement in the application makefile:

```
vpath %.c $(WIND_BASE)/target/vThreads/config/comps/src
```

### **Partition OS Stubs File**

In order for your application to access the partition OS that it runs on, you must provide the stubs files for the partition OS. Note that in order for the application to access the partition OS it is also necessary that the application's partition be configured to permit access to the partition OS in the PartitionDescription document.

Partition OS stubs files must be listed as dependencies for the application in the application makefile:

```
my-app.sm: vxMain.o my-app.o my-app.lds pos-stubs.o
```

### **Shared Library Stubs File**

In order for your application to access the shared libraries that it uses you must provide the stubs files for all the libraries that the application accesses. Note that in order for the application to access these libraries in a running system it is also necessary that the application's partition be configured to permit access to those libraries in the PartitionDescriptionDocument.

Shared library stubs files must be listed as dependencies for the application in the application makefile:

```
my-app.sm: vxMain.o my-app.o my-app.lds sl-stubs.o
```

## C++ Components

If your application is written in C++ and you make C++ library calls, you must add one of the C++ support components to the application build by specifying it as a dependency in the application build rule. The available components are:

- **vThreadsCplusComponent.o:** Provides basic C++ support and is suitable for inclusion in cert applications.
- **vThreadsCplusLibraryComponent.o:** Provides extended C++ support, but cannot be included in cert applications.

This example adds basic C++ support for the application:

```
my-app.sm: vxMain.o my-app.o my-app.lds pos-stubs.o vThreadsCplusComponent.o
$(LD) $(LDFLAGS) -T part1.lds -o $@ $(filter %.o %.pm,$^)
```

```
%.o: %.cpp
$(CC) $(C++FLAGS) -c -o $@ $<
```

If your application contains C++ global constructors, you must replace the lines above with these lines:

```
my-app.pm: my-app.o ssl-stubs.o vThreadsCplusComponent.o
$(LD) $(LDFLAGS_PARTIAL) -o $@ $^
```

```
my-app.sm: vxMain.o my-app.pm my-app.lds my-app-ctors.o
$(LD) $(LDFLAGS) -T my-app.lds -o $@ $(filter %.o %.pm,$^)
```

```
%.o: %.cpp
$(CC) $(C++FLAGS) -c -o $@ $<
```

```
%-ctors.c: %.pm
$(NM) $< | $(MUNCH) $(MUNCHFLAGS) > $@
```

These lines break the build into two separate stages. C++ applications with global objects require the OS to call the object constructors before starting the application. This is accomplished with a three-step linking process:

1. The C++ application is linked into a **.pm** file (pm stands for partially-linked module).
2. The **\$(MUNCH)** tool generates the list of global constructors from the **.pm** file.
3. The list of constructors and the **.pm** file are linked into the system module (**.sm**) file.

```
%.o: %.cpp
$(CC) $(C++FLAGS) -c -o $@ $<
```

This rule provides a generic rule for building C++ files.

```
%-ctors.c: %.pm
$(NM) $< | $(MUNCH) $(MUNCHFLAGS) > $@
```

This rule provides a generic rule for building global constructors files from **.pm** files.

## 14.4 Building an Application

To build the application, use the following procedure:

### Step 1: Create an application makefile.

Create a makefile for your application. A typical makefile is shown below:

```
all: my-app.sm

include $(WIND_BASE)/target/vThreads/config/make/Makefile.vars

POS_DIR    = ../../pos/demo

vpath %.c   $(WIND_BASE)/target/vThreads/config/comps/src
vpath %.o   $(POS_DIR)

my-app.sm: vxMain.o my-app.o pos-stubs.o my-app.lds
    $(LD) $(LDFLAGS) -T my-app.lds -o $$@ $(filter %.o,$^)

%.o: %.c
    $(CC) $(CFLAGS) -c -o $$@ $<

my-app.lds: my-app.xml
    xmlgen --ldScript --arch $(TOOLARCH) --address $(PARTADDR) -o $$@ $<
```

A typical makefile for a C++ application is shown below:

```
all: my-app.sm

include $(WIND_BASE)/target/vThreads/config/make/Makefile.vars

POS_DIR    = ../../pos/demo

vpath %.c   $(WIND_BASE)/target/vThreads/config/comps/src
vpath %.o   $(POS_DIR)

my-app.sm: vxMain.o my-app.pm my-app-ctors.o my-app.lds
    $(LD) $(LDFLAGS) -T my-app.lds -o $$@ $(filter %.o %.pm,$^)

my-app.pm: my-app.o ssl-stubs.o
    $(LD) $(LDFLAGS_PARTIAL) -o $$@ $^

%.o: %.c
    $(CC) $(CFLAGS) -c -o $$@ $<
```

```
%o: %.cpp
$(CC) $(C++FLAGS) -c -o $$ $<

%-ctors.c: %.pm
$(NM) $< | $(MUNCH) $(MUNCHFLAGS) > $$

my-app.lds: my-app.xml
xmlgen --ldScript --arch $(TOOLARCH) --address $(PARTADDR) -o $$ $<
```

**Step 2: Open the VxWorks 653 Development Shell.**

The VxWorks 653 build tools require a specific build environment which is provided by the VxWorks 653 Development Shell. To open the shell, from your program list, select:

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

**Step 3: Build the application.**

To build the application, run **make**, specifying the CPU to build for:

```
make all CPU=PPC604
```

## 14.5 Packaging an Application

Once your application is complete you should document it and package it for delivery to the system integrator.

**Step 1: Document your application.**

There are several pieces of information that the system integrator will need in order to integrate your application into a module. These include:

- The partition OS or API that the application runs on.
- The version of the partition OS API and stubs file that the application was linked against.
- The memory requirements of the application. The application's black box can be read from the ApplicationDescription document, but you must also specify the required stack and heap space to run the application.
- A list of the shared libraries that the application requires.

- A list of any shared data regions or shared I/O regions that the application requires.
- The time requirements of the application, including any periodic processes it creates.
- Which health monitor event types the application expects to handle for itself.
- Whether the application depends on health monitor notifications.
- Source and destination ports used by the application.

**Step 2: Package the application.**

To package your application, choose an appropriate packaging method (.zip file, tar ball, install package, and so on) and include the following:

- application system module file (.sm) file
- ApplicationDescription file
- any documentation for your application that the system integrator may need



# 15

## *Partitions*

15.1 Understanding Partitions	101
15.2 Planning Partitions	102
15.3 Configuring Partitions	104
15.4 Building Partitions	104
15.5 Configuring a Module for Online-Loaded Partitions	105
15.6 Building an Online-Loaded Partition	106

### 15.1 Understanding Partitions

A partition is a container for an application. The partition provides the partition operating system on which the application runs. (The partition OS is contained in a system shared library which the partition references.) The partition also regulates the application's access to other shared libraries and shared data and I/O regions. The partition supplies the stack and heap space required to run the application and any shared library code that the application uses. For information on applications, see [14. Applications](#).

Pseudo-partitions represent the outside world for purposes of communication via ARINC ports. Partitions can also communicate with the outside world using partition direct-access ports. For information on ports, see [16. Ports and Channels](#).

Partitions are the responsibility of the system integrator. For information on development roles, see [3. Configuration System](#).

## 15.2 Planning Partitions

In planning a partition, you will need to consider the following issues:

### **Which partition operating system will be used?**

Each partition must reference exactly one system shared library that contains the partition OS for the partition. You must determine from the application developer what partition OS or API (such as APEX) the application requires, and ensure that the partition OS that you provide includes all the services required by the application. The partition OS must export the correct API set and version required by the application. (Essentially this means that the partition OS must include an entry-point table that corresponds to the partition OS stubs file included in the application.)

### **Which shared libraries does the application need?**

The partition must reference each shared library that is required by the application it contains. The system integrator must determine from each application developer which libraries the application needs to access. Since each shared library that a partition references requires additional memory space in the partition, a partition should not reference a shared library unless it is required by the application.

### **How much memory does the partition require?**

Each partition occupies a separate section of physical memory. You must specify the amount of memory required by the partition. The partition will require sufficient memory for the application and for the read/write sections of each of the shared libraries which it references. These values will be found in the appropriate application and library configuration files supplied by the application developer and the platform provider respectively. In addition, the application will require stack and heap space in which to execute. These space requirements are not specified in the configuration file for the application. You must get this information from the application developer.



**Does the application require access to shared data regions?**

The application in the partition may need access to one or more shared data regions. The system integrator must determine from the application developer which shared data regions the application needs to access.

**Does the application require access to a partition direct-access port?**

Partition direct-access ports allow an application to communicate with the outside world using a port driver in the partition operating system. For information on partition direct-access ports, see [16. Ports and Channels](#).

**How will the health of the application be monitored?**

Each partition must specify the partition health monitor table to be used for health monitor events that are to be handled at the partition level.

The application in a partition may have an interest in the health of other partitions. While a partition cannot directly receive information on the health of other partitions, the health monitor does allow a notification handler in the core OS to receive and act on notifications of events in other partitions on behalf of a potentially affected partition. You need to determine if the application needs notifications to be processed on its behalf. For more information on health monitoring, see [18. Health Monitor](#).

**Will the partition be online loaded?**

The code of the online-loaded partition is made available to the core OS after system start. In some cases, this may be after all the regular partitions are already running. The partition code can be provided on a removable medium such as a PC card. To support online-loaded partitions, the platform must provide a partition loader to load the online-loaded partition code from the removable device and install it in the location in RAM that was defined when configuring the partition. The implementation of this partition loader is the responsibility of the platform provider and depends on the format used to store the partition code on the removable device. An online-loaded partition is configured the same way as a normal partition. However, the configuration of the payload is different for an online-loaded partition. For information on configuring and building online-loaded partitions, see [21. System Images](#).

**Are pseudo-partitions required?**

A pseudo-partition is a virtual representation, in the current module, of a data source in the outside world. Pseudo-partitions are used to map communication channels between partitions in external data sources.

## 15.3 Configuring Partitions

To configure a partition, write a PartitionDescription document for the partition.

The PartitionDescription document is an XML document conforming to the PartitionDescription document type defined in the VxWorks 653 Configuration Schema. The PartitionDescription document includes configuration information on the following items:

- identity of the application that is to reside in the partition
- memory required for the partition
- partition direct-access ports required by the partition's application
- shared data regions that the partition's application can access
- shared libraries that the partitions's application can access
- a number of settings affecting the runtime behavior of the partition

The structure and content of this file is defined in the *VxWorks 653 Configuration and Build Reference*. For examples of a PartitionDescription document, see [22. Reference Process](#).

## 15.4 Building Partitions

There is no separate build for a partition. Partitions are created at boot time using information in the configuration record, which is created from the PartitionDescription document as part of the module build process.

The exception to this is online-loaded partitions which are built separately from the module and loaded at runtime. For information on configuring and building online-loaded partitions, see [21. System Images](#).

## 15.5 Configuring a Module for Online-Loaded Partitions

To configure your module for an online-loaded partition, your core OS must be configured to support online-loaded partitions. For information on configuring the core OS to support online-loaded partitions, see [6. Core OS](#).

To configure an online-loaded partition, you must designate the partition payload as an online-loaded partition, and you must specify the memory location where the online-loaded partition will be loaded at run time.

To configure an online-loaded partition:

1. Open the `CoreOSDescription` document for your platform. Locate the name of the memory pool set aside for the online-loaded partition. It is located in a `CoreOSDescription/HardwareConfiguration/PhysicalMemory/kernelRegion/@PoolName` attribute. (There may be several `kernelRegions` defined, so be sure that you identify the correct one. If in doubt, consult your platform provider.)
2. Generate the `configRecord.xml` file. This step is required so that the tools can determine the correct memory location for the kernel region pool. If you change your platform memory configuration after configuring the online-loaded partition, you will need to repeat the configuration steps to determine the new address of the kernel region pool. For information on generating `configRecord.xml`, see [20. Configuration Record](#).
3. Open the `configRecord.xml` file and search for an `MmuInformation` element with a `Name` attribute that matches the kernel region pool name with the word "Pool" appended to it.
4. Locate the `VirtualAddress` attribute of the `MmuInformation` element.
5. Create a `PartitionPayload` element for the partition (for more information, see [21.4 Configuring a RAM Payload System Image](#), p.147). Set the `Base_Address` attribute to the virtual address from the previous step and set the `Online` attribute to `true`:

```
<PartitionPayload
  NameRef="my-application-partition"
  Base_Address="poolVirtualBaseAddress"
  Online="true"/>
```

## 15.6 Building an Online-Loaded Partition

To build an online-loaded partition:

1. Configure the online-loaded partition as described in [15.5 Configuring a Module for Online-Loaded Partitions](#), p.105.
2. Build a RAM or ROM payload system as described in [21.6 Building a System Image](#), p.152.
3. Write and build an online-partition loader to load the partition code from a removable device to the defined RAM address. (For example code, see the *VxWorks 653 Programmer's Guide*.)

For information on loading an online-loaded partition, see the *Wind River Workbench User's Guide, VxWorks 653 Version*.

# 16

## *Ports and Channels*

[16.1 Understanding Ports and Channels 107](#)

[16.2 Planning Ports and Channels 111](#)

[16.3 Configuring Ports and Channels 111](#)

[16.4 Building Ports and Channels 112](#)

### **16.1 Understanding Ports and Channels**

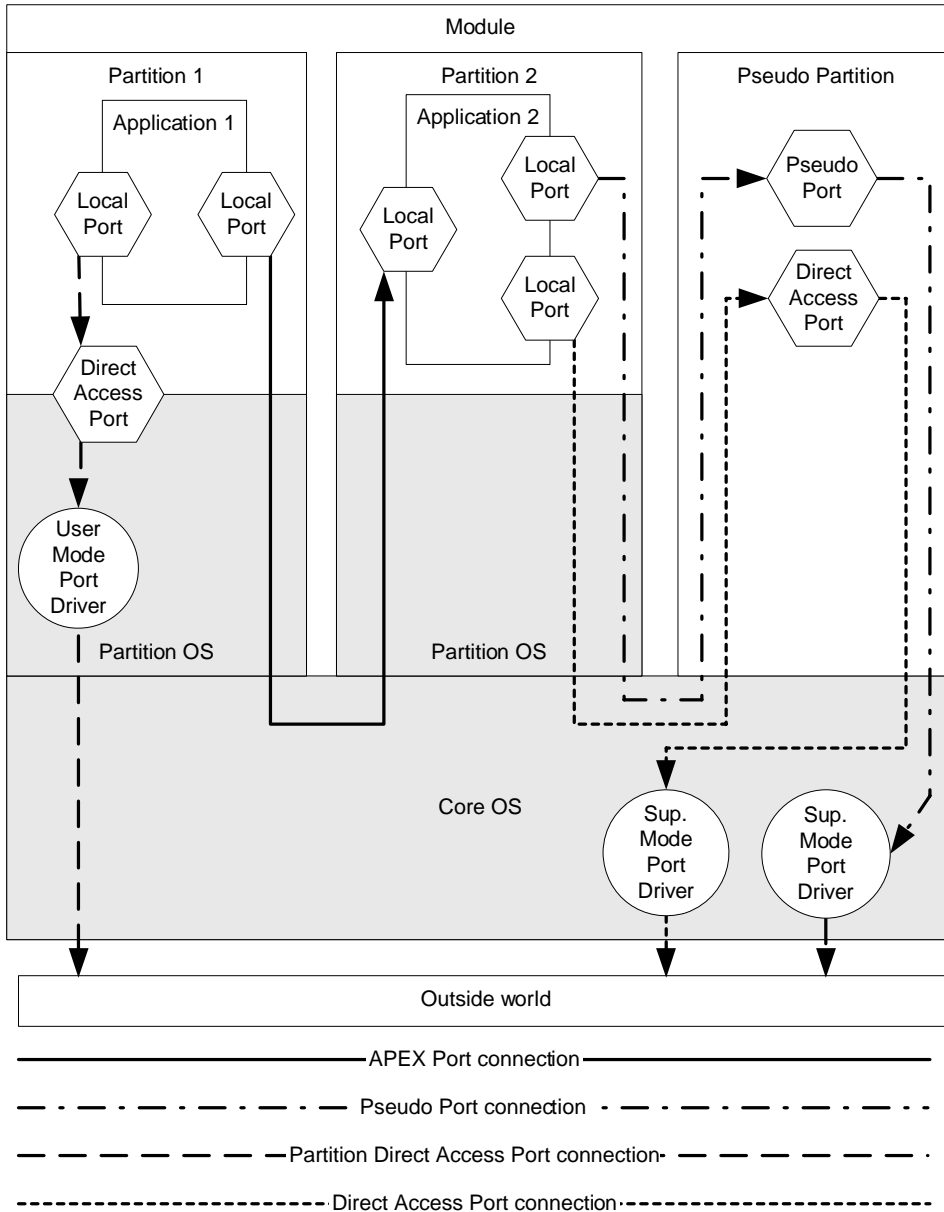
Applications can communicate with each other and with the outside world using ARINC ports. In order for port communications to work, you must configure connections between the ports at the module level. Connections consist of channels that connect one sending port to one or more receiving ports. Channels can also be configured for communication with data sources in the outside world generally.

VxWorks 653 supports four types of ports which are illustrated in [Figure 16-1](#).

#### **ARINC Ports**

ARINC ports connect one application to another using ARINC port services in the core OS. ARINC ports support one to one and one to many communication. For information on programming applications to use ARINC ports, see the *VxWorks 653 Programmer's Guide*.

Figure 16-1 **Ports Overview**



### **Pseudo-Ports**

An ARINC pseudo-port looks like a local ARINC port to other ARINC ports within a module, but it actually communicates, via a custom driver, with data sources outside the module. Pseudo-ports represent data sources in the outside world. A pseudo-port can be used for one-to-one or one-to-many communication. Because they communicate outside the module, they require a user supplied port driver to handle communication with the external data source. For information on port drivers, see the *VxWorks 653 Programmer's Guide*.

### **Direct-Access Ports**

A direct-access port is a type of pseudo-port that does not use the software buffering of the ARINC port services, but communicates directly with the communications hardware, relying on the buffering capabilities of the hardware itself. A direct-access port can only be used for one-to-one communication.

### **Partition Direct-Access Ports**

Pseudo-ports and direct-access ports use port drivers that reside in the core OS and operate in supervisor mode. Partition direct-access ports allow you to place the driver in the partition OS where it will operate in user mode. Note that regular pseudo-ports are not permitted in a partition. Only direct-access ports can be used in a partition. A partition direct-access port can only connect to a port in the application that resides in the partition. Partition direct-access ports cannot be used for communication between two partitions.

### **Pseudo-Partitions**

Pseudo-ports and direct-access ports reside in the core OS, but for configuration purposes they are represented as belonging to a pseudo-partition.

### **Configuring Port Types**

In general, applications do not know what kind of port is on the other end of a channel. Ports in an application are always configured as local ports. The type of port communication that occurs is determined by the type of port at the other end of the channel. If the channel connects a local port in an application to a local port in another application, a regular ARINC port communication occurs. If the channel connects a local port in an application to a direct-access port in its partition, for instance, partition direct-access port communications occur.



**WARNING:** While applications are theoretically unaware of what is on the other end of a channel and regard all communication as occurring over a local ARINC port, timing issues with different types of ports may affect how the application reads information from the local port. For more information, see the *VxWorks 653 Programmer's Guide*.

**Null Ports**

During development, it may be useful to configure a port as a null port, which simply means that the port creates no data and consumes all data it receives.

Ports in applications, partitions, and pseudo-partitions, can only be of types appropriate to their role in port communication, as shown in [Table 16-1](#).

Table 16-1 **Allowed Port Types by Location**

Location	Allowed Port Types
Application	local null
Partition	direct-access null
Pseudo-partition	pseudo direct-access null

**Queuing vs. Sampling Ports**

A port may be either a queuing port or a sampling port. A queueing port is a port on which messages are queued and can be read one by one. A sampling port has only one message at a time. To deal with the situation in which its message queue is full, a queuing port may use one of two protocols. In the sender block protocol, the source port cannot send a message until all the destination ports have room in their queues to receive the message. In the receiver discard protocol, the message is always sent and any destination port that does not have room in its queue discards the message.

Application ports are configured as part of the application configuration. For information on applications, see [14. Applications](#).

Partition ports are configured as part of the partition configuration. For information on partitions, see [15. Partitions](#).



Pseudo-ports are configured in the pseudo-partition configuration. For information on pseudo-partitions, see [15. Partitions](#).

For information on development rules, see [3. Configuration System](#).

## 16.2 Planning Ports and Channels

Planning ports and connections requires collaboration between the platform provider, application developer, and system integrator.

The platform provider must provide a core OS and partition OS that includes the appropriate port drivers.

Application developers must communicate the ARINC ports used by their applications to the system integrator so that the system integrator can configure connections to those ports.

The system integrator must develop and communicate a communications plan for the module as a whole so that the platform provider and the various application developers can configure ports that will take advantage of the communications channels provided by the module. The system integrator must ensure that ports at the ends of channels are compatible. For the rules for matching ports, see the *VxWorks 653 Configuration and Build Reference*.

16

## 16.3 Configuring Ports and Channels

The configuration of ports and connections is distributed as follows:

Application ports are configured in the ApplicationDescription document in the element **ApplicationDescription/Ports** and are the responsibility of the application developer.

Partition ports are configured in the PartitionDescription document in the element **PartitionDescription/Ports** and are the responsibility of the system integrator.

Pseudo-partition ports are configured in a **PseudoPartitionDescription** document and are the responsibility of the system integrator.

Channels to connect the ports are configured in the **Module/Connections** element of the Module configuration document and are the responsibility of the system integrator.

## 16.4 Building Ports and Channels

There is no separate build process for ports and channels. Ports and channels are instantiated at boot time based on information in the configuration record. The ports and channels information in the configuration record is created from the ports and channels information in the Module, PartitionDescription, PseudoPartitionDescription, and ApplicationDescription documents by the configRecord build. For more information, see [20. Configuration Record](#) and [21. System Images](#).

# 17

## *Schedules*

[17.1 Understanding Schedules 113](#)

[17.2 Planning Schedules 114](#)

[17.3 Configuring Schedules 116](#)

[17.4 Building Schedules 116](#)

### **17.1 Understanding Schedules**

In VxWorks 653, applications are run on a time-deterministic schedule. Applications cannot run outside of their scheduled time. Each schedule consists of a set of partition windows, each of which is a time slice in which a specified partition can run. You can schedule a partition more than once in a schedule, and you do not have to schedule every partition in every schedule. The schedule with the ID of 0 is run at startup.

#### **Spare Time**

In addition to scheduling partitions, you can also add spare time to a schedule by using “SPARE” as the name of the partition to run. Spare time is needed if your platform includes device drivers or kernel tasks that run in the core OS and that require time to run outside of a partition time slice.

### Release Points

Each partition window can be designated as a release point. A release point is a means for synchronizing periodic processes with a partition's schedule. A periodic process is a process within a partition that is scheduled in wall clock time. That is, the schedule for the periodic process is counted whether the partition is running in its scheduled partition window or not. It is important that the partition be in its scheduled window when the periodic process is scheduled to run, and that there is enough time remaining in the partition window for the periodic process to run to completion. To help accomplish this synchronization, when a partition spawns a periodic process, that process is not started until the next release point. A release point occurs at the start of each partition window that is designated a release point.

Schedules determine the time allotments for applications. For information on applications, see [14. Applications](#).

Schedules are the responsibility of the system integrator. For information on development roles, see [3. Configuration System](#).

## 17.2 Planning Schedules

It is important that you obtain accurate information on the scheduling requirements of the application from the application developer so that the application can be scheduled appropriately. Required information includes the amount of time needed for the application to execute in each partition window, and the period and duration of any periodic processes created by the application. It may also include time for processes that run in the core OS using the partition's time slice. For example, a health monitor notification handler is run in the core OS on behalf of (and in the time slice of) a partition.

Partitions run as tasks in the core OS. It is important to understand the impact that activity in the core OS can have on your schedules. The following summarizes the major categories of core OS activity and how they can affect the timing of partitions.

- The system is booting and the core OS has not yet started the partitions. This has no impact on partition schedules, since they are not yet running.
- The core OS is servicing a request from a partition (that is, the partition made a system call). This is part of the normal time calculation for an application and

should be accounted for in the application developer's duration calculation. This includes all the operations that the core OS performs in handling communication between the partitions. For instance, if partition A sends data to a port, that operation is handled by the core OS in partition A's time slice. If partition B reads data from a port, that operation is handled by the core OS in partition B's time slice.

- The core OS is running a partition-level health monitor event handler for a health monitor event that was injected in a partition. The handling the event will not impact the scheduling of other partitions. If there is not sufficient time in the current partition window to complete the event handler, the event handler process will be suspended until the next partition window for the affected partition. It is also important to remember that a partition may have more than one health monitor event queued for handling when its time slice begins.
- The core OS is running a module-level health monitor event handler for a health monitor event that was injected in a partition. Health monitor event handlers running at the module level are not interrupted by the scheduler.
- The core OS is running a health monitor notification handler on behalf of a partition. The handler is running in the time slice of the notified partition and can potentially cause it to miss its deadline. If you configure a partition to accept notifications, it is important to calculate the possible impact on your schedule. You should also remember that for any event type for which you accept notification, you will also get those notifications for core OS events as well as events in your trusted partitions. It is also important to remember that a partition may have more than one health monitor notification queued for handling on its behalf when its time slice begins.
- The core OS is running an event handler for an error in the health monitor itself. Such errors are handled at a very high priority level, meaning they can disrupt partition schedules. Such events should only occur under exceptional circumstances.
- The core OS is running a user-supplied driver or other user-supplied core OS code. If this code has a lower priority than the partitions, then it will only run in a SPARE partition window so it will not affect the scheduling of partitions. However, since the code is running in the core OS, it has the privileges of a core OS process, including the ability to change schedules. If the code is running at a higher priority than partitions, it will preempt the partitions, meaning the partitions will not run as scheduled. There is nothing you can do in the schedule configuration to prevent this. In a safety-critical system, is important

to make sure that there is no code introduced at the platform level that can interfere with or preempt schedules.

- The core OS is servicing interrupts. Time to service interrupts is taken from the currently executing process, which will usually be a partition. However interrupts are processed very quickly—they are translated into pseudo-interrupts and queued for the appropriate partitions, which will then process them in their own partition window.
- The core OS is running any other tasks that have a higher priority than the partitions. This includes tasks like the target agent or the target shell. These will preempt a partition when they become ready to run. These types of tasks should generally be removed from a certified system. When the partition scheduler is preempted by a higher priority task in the core OS, the scheduler continues to keep track of the passage of time and will resume the schedule at the correct point in time when the higher priority task ends. In other words, it will not suspend the schedule and allow the preempted partition to finish. The schedule keeps running and execution of scheduled partitions resumes wherever they fall in the schedule. If any partition is unable to start at its scheduled time, an `HME_PARTITION_OVERFLOW` health monitor event is injected.

## 17.3 Configuring Schedules

Schedules are configured by completing the Schedules element of the Module configuration document. For details of the Schedules element, see the *VxWorks 653 Configuration and Build Reference*. For an example of a Module document with schedule information, *Building the C++ Application*, p.193.

## 17.4 Building Schedules

There is no separate build process for schedules. Schedules are instantiated at boot time based on information in the configuration record. The schedule information

in the configuration record is created from the schedule information in the Module document by the configRecord build.





# 18

## *Health Monitor*

[18.1 Understanding the Health Monitor 119](#)

[18.2 Planning Health Monitoring 123](#)

[18.3 Configuring the Health Monitor 126](#)

### **18.1 Understanding the Health Monitor**

The health monitor provides a central dispatch system for events that represent either alarms or messages. All events raised in the system are handled by the health monitor. The health monitor then hands off the events to event handling routines either in the core OS or in the partitions.

There are four components of the health monitoring system: the system, module, partition, and process health monitors.

#### **System Health Monitor**

The system health monitor is the dispatcher for the health monitoring system. For each event type, the system health monitor determines if that event type will be handled by the module health monitor, the partition health monitor, the process health monitor, or not handled at all.

### Module Health Monitor

The module health monitor handles events dispatched to the module level by the system health monitor and directs them to the appropriate event handlers based on their type. There is one module health monitor for the entire module. Event handlers invoked by the module health monitor run in the core OS context (that is, supervisor mode). They are not subject to the ARINC scheduler and may therefore cause delays in the execution of the schedule.

### Partition Health Monitor

The partition health monitor handles events dispatched to the partition level by the system health monitor and directs them to the appropriate event handlers based on their type. There is one partition health monitor for each partition, though several partition health monitors can use the same partition health monitor configuration. Event handlers invoked by the partition health monitor execute in the context of the partition where the event was injected (in supervisor mode), and in the time slice of that partition.

### Process Health Monitor

The process health monitor handles events dispatched to the process level by the system health monitor. The process health monitor is not configured at the module level. Instead, the process health monitor is provided as a routine in the application code of the partition. During initialization, the application must register its process health monitor routine with the system health monitor. See the *VxWorks 653 Programmer's Guide* for details. If the process health monitor is not installed, events are dispatched to the partition health monitor instead.

## Event Types

Health monitor events are divided into types, following the types specified in the ARINC 653 specification. VxWorks 653 represents event types by their ARINC 653 code names prefixed by **HME\_**. A number of VxWorks 653 specific types are also included. Event types are defined in **hmTypes.h**.

## Messages

The **HM\_MSG** event type is used to convey messages and does not normally represent a fault. The routing of health monitor messages is hard coded and cannot be configured. **HM\_MSG** events that occur in a partition are handled by the partition health monitor and those that occur at the module level are handled by the module health monitor. While the routing of messages is hard-coded, you must explicitly configure the handling of messages by supplying a handler for **HM\_MSG**

events at the module and partition levels. Unless you have specific needs for handling **HM\_MSG** events, you can map them to the **hmDH\_EventLog()** routine provided by **hmDefaultHandlers.c**. If you do not provide a handler for **HM\_MSG** events, the handler configured for the **HME\_DEFAULT** event will be used. Turning on autologging of events does not automatically log **HM\_MSG** events.

## Event Handlers

An event handler is a routine located either in the application code (in the case of the process health monitor) or in the core OS (in the case of the partition and module health monitors). VxWorks 653 provides a set of event handler routines in the file **hmDefaultHandlers.c** which you can use if more suitable event handlers have not been provided as part of the platform. Note that the word “default” in the name of this file does not mean that these handlers will be used automatically if you do not configure alternatives. These are merely basic event handlers that you can configure your module to use if you want. The comments in **hmDefaultHandlers.c** indicate which of the routines it contains are likely to be appropriate for certain types of events, but these are guidelines only. It is up to you to decide which event handlers are appropriate for your module in all situations.

If you do not configure an error handler for a particular type of event, the handler configured for the **HME\_DEFAULT** event is used. You must configure a handler for **HME\_DEFAULT** events. The handler supplied in **hmDefaultHandlers.c** for use with **HME\_DEFAULT** events (**hmDefaultHandler()**) restarts the partition (if called at the partition level) or the module (if called at the module level).

## Event Queues

To minimize the time penalty associated with running a health monitor process, the system health monitor dispatcher (which runs at a very high priority level) does not call the module, partition, or process health monitors directly. Instead, it dispatches events to the queue for the appropriate health monitor, to be handled in the appropriate time slice for the particular health monitor. Each health monitor, therefore, has an associated event queue that must be configured appropriately.

## Event Logging

The module and partition health monitors have logs where events and messages can be recorded. The logs can then be read by applications using the health monitoring API or by the developer using **windSh** commands. Event handlers can log events by calling the appropriate API routines. This is the responsibility of the platform provider or application developer who writes the event handler. As system integrator, you can cause all events to be logged automatically (with the

exception of **HM\_MSG** events, which must be logged explicitly) by turning on autologging.

Event logs are stored in volatile memory and do not survive a system restart. If you want logs to be kept in non-volatile memory, you (or the platform provider) must provide handlers that log events to non-volatile memory, or that periodically empty the logs to non-volatile memory.

### Health Monitor Notification

In addition to dispatching events, the health monitor can also dispatch health monitor notifications. A health monitor notification is a message that a health monitor event has occurred. It can be used to handle any impact that the occurrence of an event in one partition may have on other partitions. (For instance, if partition A supplies data to partition B, and partition A experiences a fault and must be restarted, partition B may need to react to the fact that its source of data has been interrupted.)

Notifications are not sent to partitions themselves. Notifications are sent to notification handlers in the core OS, which handle the notification on behalf of the affected partition. Notification handlers run in the time slice of the partition on whose behalf they run, not the partition that injected the event. (The partition that injected the event does not receive notification of that event—it receives the event itself.)

The core OS can also receive notification of events in particular partitions. Such notifications are handled in the time slice of the partition in which the event was injected.

Since notification handlers run in the core OS, they can use any of the core OS facilities to communicate any necessary information to the affected partition (such as communicating via a port or a pseudo-interrupt). This communication is entirely the responsibility of the platform provider and the application developer and is not part of the health monitor configuration.

The health monitor monitors the health of the module. For information on modules, see [2. Understanding VxWorks 653](#) and [19. Modules](#).

Configuration of the health monitor is the responsibility of the system integrator. For more information on development roles, see [3. Configuration System](#).

## 18.2 Planning Health Monitoring

When planning your health monitoring strategy, you need to consider the following issues:

### At which level will events be handled?

Since there is only a single system health monitor table for the entire system, and therefore only one table that governs which types of events will be dispatched to the process level, all applications in the module will have the same set of event types dispatched to them. If the application does not provide a process health monitor that handles a particular type of event that is dispatched to it, the event will be dispatched to the partition health monitor. It is important, therefore, that the system integrator communicate with the application developers to determine the set of event types to be mapped to the process health monitor level in the system health monitor table.

If an event cannot be handled at the level to which it would normally be dispatched, it will be dispatched to the next highest level. Circumstances that could cause events to be dispatched to a higher level include an application that fails in a way that prevents the failure from being properly reported, the absence of a process health monitor in a particular partition, a full queue at any health monitor level, or the injection of an event from a process with a priority higher than that of the designated health monitor. For this reason you should plan for the possibility that, under certain circumstance, an event that is mapped to the process level may be dispatched to the partition or module level, and an event mapped to the partition level may be dispatched to the module level. To prepare for this possibility, ensure that any event handlers that are configured for an event at a lower level are also configured for that event at all higher levels or that an appropriate default event handler is configured at all higher levels.

### How will events be logged?

You can configure logging separately for the module health monitor and each partition health monitor. When configuring logging for a module or partition health monitor, you need to consider the following questions:

- Should all events be logged automatically, or should logging be left to each event handler?
- How many log entries should the log hold? (The size of a log entry depends on the size of the **hm\_event** type, which can vary based on certain configuration parameters. See **hmTypes.h** for details.)

- What should happen when the log is full? Once the log is full, each new entry overwrites the oldest entry in the log. Optionally, you can configure a log threshold that will trigger an **HME\_HMQ\_OVERFLOW** event when the log reaches a certain size. You can then configure a handler for the **HME\_HMQ\_OVERFLOW** event that empties the log (transferring entries to more permanent storage, perhaps) before it reaches its maximum size and entries are overwritten.

#### **How will the health monitor queues be managed?**

While the health monitor is a high priority task and therefore services most events as soon as they are injected, it is possible that events might not be handled in the same time slice in which they are injected. This can happen if the partition runs out of time before the event is handled. It is also possible for a core OS process running on behalf of a partition to inject events into the partition's health monitor. If these processes run at a high priority, the health monitor may not have time to run the event handler in the current time slice. These events would also need to be queued to be handled in the next time slice. A queue size of ten is considered adequate in most circumstances.

#### **How will notification be managed?**

The use of health monitor notifications is optional, and requires communication between the platform provider, the application developer, and the system integrator.

Notification handlers are part of the core OS and are generally created by the platform provider and provided as part of the platform.

Application developers do not need to be directly aware of health monitor notifications; however, if the behavior of the notification handler provided by the platform provider is to communicate information to the application, the application developer needs to be aware of this, and provide the appropriate facilities to handle the communication.

The system integrator is responsible for configuring partitions to allow health monitor notifications to be handled on their behalf. The system integrator needs to know if application developers have made use of, or relied upon, the existence and operation of notification handlers in the core OS. They also need to consider the time implications of allowing notification handlers to run in a partition's time slice, and to ensure that notifications are appropriately filtered so that the time impact is minimized.

Health monitor notification does not take place automatically. The person who writes the event handler for an event must specifically create a notification by

calling the **hmNotificationSend()** routine. When configuring notification it is necessary to determine if the event handlers supplied as part of a platform or application are sending notifications.

Health monitor notifications occur based on the level of the event handler that sends the notification. That is, if an event is injected in partition A, but that type of event is routed to the module level in the system health monitor table, the event will then be processed at the module health monitor level. Notification of that event generated by a module-level handler will be treated as coming from the core OS and will be sent to all partitions that accept notification of this event type, whether or not they trust the partition in which the event was injected.

Because a notification handler runs in the time slice of the affected partition, partitions need to be able to control the number of notifications that are processed on their behalf by the core OS. In effect, the partition needs a spam filter to make sure that it surrenders time only for the processing of the notifications that are relevant to it.

This spam filter has two parts. The first is a white-list of partitions from which it is willing to accept notifications. This is set up in the Module document as a list of trusted partitions. The second is a subject filter, based on the types of events that it is willing to accept notifications of. This is set up in the Module document as an event filter mask.

By default, a partition does not allow any notifications to be processed on its behalf. You must explicitly configure a partition to allow notifications to be processed on its behalf.

Health monitor notifications are placed on a queue for the affected partition. When the affected partition receives its next time window, the notifications in the queue are processed. This means that the partition may be giving up time to process more than one notification in any given time slice. You can control the number of notifications that can be queued. You can also determine whether you want an event to be raised if the number of notifications to be queued exceeds the length of the queue.

#### **How will health monitor errors be handled?**

The health monitor itself, as well as the event and notification handlers that it calls, may experience faults. This will result in events being injected from the health monitor itself, which will then be routed according to the appropriate mapping in the system health monitor table. As part of your health monitoring strategy, you must plan how to handle health monitor errors. The health monitor may also generate events to indicate that some resource, such as a log or a queue is full. You will need to develop an appropriate recovery strategy for these types of situations.

Correctly written event handlers will either inject an event of the appropriate type when they encounter an error, or will return **ERROR**. When an event handler returns **ERROR**, the health monitor injects an **HME\_HM\_ERROR** event and dispatches it according to the mapping of that event type in the system health monitor table. For this reason it is important to provide an explicit routing for **HME\_HM\_ERROR** in the system health monitor table. Note that if the module health monitor returns **ERROR**, the **HME\_HM\_ERROR** handler is called directly by the dispatcher (thus bypassing the health monitor queues). Since the health monitor dispatcher runs at a very high priority level, this could result in partition schedules being disrupted.

If any of the health monitor queues become full, and a new event is dispatched to that queue, the event is reformatted as an **HME\_HM\_ERROR** event and dispatched to the module level where it is handled as described above. If you configure a threshold for each queue, and an event is placed on the queue that causes it to hold more events than the threshold number, the event is placed on the queue and an **HME\_HMQ\_OVERFLOW** event is injected and also placed on the queue. (This is why the recommended threshold value is two less than the queue size, to allow room for the event that breached the threshold and the **HME\_HMQ\_OVERFLOW** event.) The **HME\_HMQ\_OVERFLOW** event is a record of the fact that an overflow of the threshold occurred. It does not indicate whether or not the queue itself subsequently overflowed. The health monitor sees the **HME\_HMQ\_OVERFLOW** event only after the other events on the queue have been processed. This is useful principally as a debugging device to check that all queues are of adequate size. It is not a suitable mechanism for recovering from a queue overflow at run-time.

## 18.3 Configuring the Health Monitor

To configure the health monitor, use the following procedure:

### Step 1: Configure the system health monitor.

Configure the system health monitor by completing the **/Module/HealthMonitor/SystemHealthMonitorTable** element of the Module configuration document. The content of the **SystemHealthMonitorTable** element is detailed in the *VxWorks 653 Configuration and Build Reference*.



**Step 2: Configure the module health monitor.**

Configure the module health monitor by completing the **/Module/HealthMonitor/ModuleHealthMonitorTable** element of the Module configuration document. The content of the **ModuleHealthMonitorTable** element is detailed in the *VxWorks 653 Configuration and Build Reference*.

**Step 3: Configure partition health monitors.**

Configure each partition health monitor by adding a **/Module/HealthMonitor/PartitionHealthMonitorTable** element to the Module configuration document. The content of the **PartitionHealthMonitorTable** element is detailed in the *VxWorks 653 Configuration and Build Reference*.

**Step 4: Assign partition health monitors to partitions.**

As part of the configuration of a partition you must specify the partition health monitor table that will be used for that partition's health monitor. The **PartitionHMTable** attribute of the **/PartitionDescription/Settings** element specifies the name of the partition health monitor to be used. The name of the partition health monitor table is specified by the **Name** attribute of the **/Module/HealthMonitor/PartitionHMTable** element.

**Step 5: Configure the process health monitors.**

The process health monitor is written by the application developer. Its configuration is up to the application developer.



# 19

## *Modules*

[19.1 Understanding Modules 129](#)

[19.2 Planning Modules 130](#)

[19.3 Configuring a Module 132](#)

[19.4 Building a Module 134](#)

### 19.1 Understanding Modules

An ARINC module is a set of partitions controlled by a single core OS. In VxWorks 653, a module build creates the system configuration record and a ROM payload, RAM payload, or network-loadable system image that can be installed on a target and executed. Configuration of a module, therefore, involves bringing together all the elements of the system, ensuring that they have the resources required to run, scheduling them appropriately, and providing appropriate health monitoring to ensure that they are running correctly.

A module contains the following:

- A core OS, which may be a certified OS or may include debug support.
- Zero or more user configuration record regions.
- One or more partition OSs.
- Zero or more shared libraries.

- Zero or more shared data regions.
- Zero or more shared IO regions.
- One or more partitions, each of which contains an application. Some of these partitions may be online-loaded partitions.
- ACE, the debug agent for use in a certified environment, if required.

Designing the module is the responsibility of the system integrator. The system integrator works in cooperation with the platform provider and the application developers to create a platform and a set of applications that work correctly together as a module. For information on development roles, see [3. Configuration System](#).

A module is a complete VxWorks 653 system. The module configuration brings together all the elements that make up a module. For more information on modules, see [2. Understanding VxWorks 653](#).

The configuration of a module is expressed in a configuration record. For information on the configuration record, see [20. Configuration Record](#).

A module is instantiated as a system image. For information on system images, see [21. System Images](#).

## 19.2 Planning Modules

In planning a module, you should consider the following issues:

### **Which platform will the module use?**

A module is built on a platform. Platforms provide a specific set of resources that the module and its applications can use. You will need to decide if your module will be built on an existing preconfigured platform or if it will be necessary to commission a platform with specific features to support this module. You may need to communicate with the application developers to determine the features they will require, and with the platform provider to determine if the platform can support those features. The platform requirements for the various features of a module are discussed in the sections that follow.

### What applications will there be?

A module contains one or more applications. Each application resides in a separate partition. Applications are written to run on a specific partition OS, or to run on a specific API supported by a partition OS, such as APEX. Applications may require access to system resources such as ports, shared libraries, or shared data regions. You will need to determine what APIs and resources each application in the system will require and configure your module appropriately.

VxWorks 653 supports up to 255 application partitions (256 total partitions including the core OS). However, the operating system has a number of default settings that are suitable for modules of 32 partitions or less. If you configure a module with more than 32 partitions, you will need to make sure that your platform has been configured to support the appropriate number of partitions. For information on configuring a core OS for more than 32 partitions, see [6. Core OS](#).

### What shared memory regions will be required?

Most of the memory in a VxWorks 653 system is assigned exclusively to individual partitions and is not shared with any other partitions or the core OS. For some purposes, however, it is necessary to share memory between applications and/or the core OS.

There are two kinds of shared memory region: shared I/O regions and shared data regions. A shared I/O region is a memory location assigned to a hardware I/O device, such as an LED, that is accessed by more than one application. A shared data region is a region that contains data, such as a database, that is used by multiple applications.

You must determine if any shared memory regions are required, and configure them with the correct access rights for the applications that will use them.

### What shared I/O regions will be required?

To configure a shared I/O region you need to determine the following:

- The name of the shared I/O pool in the platform configuration. The set of available shared I/O pools can be determined by consulting the **CoreOSDescription/HardwareConfiguration/sharedIO** elements in the appropriate CoreOSDescription document.
- The level of access the core OS will have to the shared I/O region. This requires you to communicate with the platform provider to determine what level of access the core OS requires to the region. (Partition access to the shared I/O region is configured in the partition configuration, which is described in [15. Partitions](#).)

- The virtual address that the shared I/O region will occupy in the module configuration.
- The appropriate memory caching policy for the shared I/O region.

#### **Will online-loaded partitions be required?**

Online-loaded partitions are partitions that are not part of the regular payload but are loaded into a system while it is running. The partition is not loaded during the boot sequence for the module, but the system reserves the resources required to run the partition, as specified in its configuration. For information on configuring online-loaded partitions, see [15. Partitions](#).

#### **How will the applications be scheduled?**

The core OS allots time to partitions according to a schedule. The schedule is established in Module configuration document. You can establish up to 16 schedules. The core OS, or an application with the appropriate syscall permissions (as configured in **PartitionDescription/Settings/@syscallPermissions**), can switch from one schedule to another at run-time. For information on schedules, see [17. Schedules](#).

#### **How will the health of the system be monitored?**

VxWorks 653 provides a health monitor that can be used to dispatch events to appropriate handlers based on their type, the partition in which they were injected, and the state of the system at the time they were injected. You will need to configure the health monitor tables to route events appropriately for your module. For information on health monitoring, see [18. Health Monitor](#).

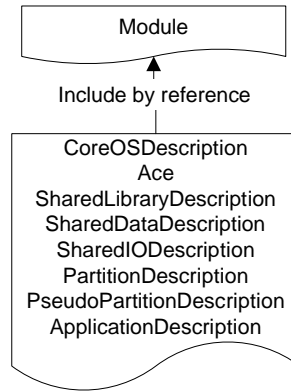
## **19.3 Configuring a Module**

[Figure 19-1](#) summarizes the configuration of a module.

The configuration of a module is expressed in a Module document. The Module document is an XML document conforming to the Module document type defined in the VxWorks 653 Configuration Schema.

This document contains configuration information for the module in the following areas:

Figure 19-1 **Configure a Module**



The Module document includes the following configuration documents by reference:

- CoreOSDescription document (see [6. Core OS](#))
- Ace configuration document (see [12. ACE](#))
- SharedLibraryDescription documents (see [9. Shared Libraries](#))
- ApplicationDescription documents (see [14. Applications](#))
- PartitionDecscription documents (see [15. Partitions](#))
- SharedDataDescription documents (see [10. Shared Data Regions](#))
- SharedIODescription documents (see [11. Shared I/O Regions](#))

In addition, the Module document contains the following elements inline:

- **Schedules** element (see [17. Schedules](#))
- **HealthMonitor** element (see [18. Health Monitor](#))
- **Connections** element (see [16. Ports and Channels](#))
- **Payloads** element (see [21. System Images](#))

## 19.4 Building a Module

The module configuration is expressed in the build of the configuration record or a payload. For information on building the configuration record, see [20. Configuration Record](#). For information on building a payload, see [21. System Images](#).



# 20

## *Configuration Record*

20.1 Understanding the Configuration Record 135

20.2 Planning the Configuration Record 136

20.3 Configuring the Configuration Record 137

20.4 Building the Configuration Record 139

### 20.1 Understanding the Configuration Record

The configuration record contains configuration information for a module. It is used at boot time to correctly set up the module and each of its components, and to regulate many aspects of the runtime behavior of the module. For more information on modules, see [19. Modules](#).

The configuration record is compiled from information contained in the configuration documents for the module and all its constituent parts. Generating the configuration record is a two step process. In the first step the Module configuration document, and all the configuration documents that it references is processed by the VxWorks 653 build tools to calculate those values that were not specified in the component and module configuration documents, and to assign each component its own memory location and resource access. This produces a document called **configRecord.xml**.

To create the binary configRecord that can be included in the image loaded on the target, **configRecord.xml** is processed by the VerIMAX tool. The output of this

process is the binary file **configRecord.reloc** or **configRecord.bin**. Whether the output is **configRecord.bin** or **configRecord.reloc** depends on the kind of system image that you are creating. For a downloadable system image, a **.reloc** file is created. For a RAM or ROM payload file, a **.bin** file is created.

The configuration record is part of a system image. For information on system images, see [21. System Images](#).

The configuration record is the responsibility of the system integrator. For information on development roles, see [3. Configuration System](#).

## 20.2 Planning the Configuration Record

The configuration record is a compilation of the information used to configure each of the components of the module, and the module itself. As such, no separate planning is required for the configuration record. However, there are certification considerations which can affect how you organize your configuration and build process for the configuration record.

### How will certification be handled?

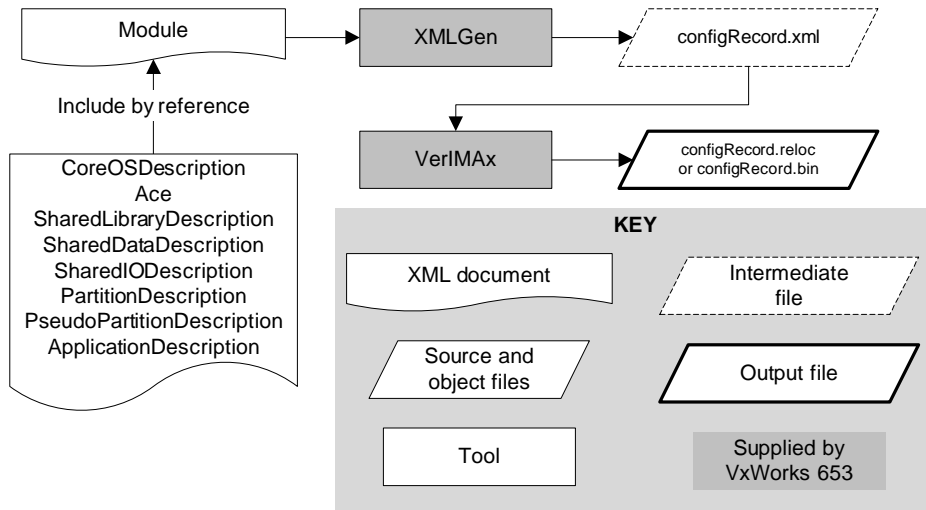
Because VerIMAx is a qualified tool, its outputs are considered to be certified as long as its inputs are certified. The input to VerIMAx is **configRecord.xml**. **configRecord.xml** is created by the build tools from the set of configuration files brought together in the Module configuration document. In creating a certified system, it is necessary to certify the Module configuration document and all of the configuration files that it references. VerIMAx verifies that **configRecord.xml** had been created correctly from the Module configuration file, however, you must examine **configRecord.xml** to certify that the XInclude statements that it uses to bring in the other configuration documents point to the correct documents.

In some cases, it may be desirable to certify a module configuration before all the components of the system are finished. Also, it is useful to be able to certify the module configuration independently of the components of the module so that changes in one component do not require re-certification of the entire module. The separation of configuration and build elements supported by VxWorks 653 facilitates this. In some cases, you may want to build and certify the module configuration and distribute the certified binary **configRecord.bin** to the other members of the development team.

## 20.3 Configuring the Configuration Record

Figure 20-1 summarizes the build of the configuration record.

Figure 20-1 Building the configuration record



The inputs and outputs of the configuration record build process are as follows:

### Outputs of the Configuration Record Build

The outputs of the configuration record build are:

#### **configRecord.reloc**

**configRecord.reloc** is the binary configuration record that can be included in a system image. For information on system images, see [21. System Images](#).

#### **configRecord.xml**

**configRecord.xml** is an XML document conforming to the ConfigRecord document type defined in the VxWorks 653 Configuration Schema. ConfigRecord.xml contains memory configuration information calculated from the information in the Module configuration document and the associated configuration documents for the other components of the module.

**configRecord.xml**, in turn, is used to configure the memory allocation of the module build.

In a full configuration record build, **configRecord.xml** is an intermediary file. However, in a certified system, you must certify that the XInclude statements in **configRecord.xml** identify the correct configuration documents before it is processed by the qualified tool, **VerIMAx**, to produce the module binaries. In the case of a certified build, therefore, **configRecord.xml** is the output of the first part of the configuration record build and an input to the second part of the build.

**Makefile.rules** contains the rules needed to build **configRecord.xml**.

### Inputs to the Configuration Record Build

The following are the inputs to the configuration record build:

#### Module Configuration Document

For information on creating the Module configuration document, see [19. Modules](#) and the *VxWorks 653 Configuration and Build Reference*. For examples of a Module document, see [22. Reference Process](#).

#### Configuration Files for Module Components

The module build process requires all of the configuration files for the components of the module. The Module configuration document must identify the location of these files via **xi:include** elements in the appropriate places.

- CoreOSDescription document (see [6. Core OS](#))
- Ace configuration document (see [12. ACE](#))
- SharedLibraryDescription documents (see [9. Shared Libraries](#))
- ApplicationDescription documents (see [14. Applications](#))
- PartitionDescription documents (see [15. Partitions](#))
- SharedDataDescription documents (see [10. Shared Data Regions](#))
- SharedIODescription documents (see [11. Shared I/O Regions](#))

## 20.4 Building the Configuration Record

To build the configuration record, use the following procedure:

**Step 1: Create a configuration record makefile.**

To build a configuration record you must create a makefile. Since building the configuration record is part of the process of building a system image, most system integrators will write one makefile that can be used to build the configuration record as well as the different payload image types. The following makefile example contains only the information required to build the configuration record:

```
include $(WIND_BASE)/target/config/make/Makefile.vars

XML_FILE = hello.xml

include $(WIND_BASE)/target/config/make/Makefile.rules
```

There are no targets for **configRecord.xml** and **configRecord.reloc** in this makefile. Those targets are contained in **Makefile.rules**.

**Step 2: Open the VxWorks 653 Development Shell.**

The VxWorks 653 build tools require a specific build environment which is provided by the VxWorks 653 Development Shell. To open the shell, select from the program list:

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

**Step 3: Build the configuration record.**

To generate **configRecord.xml**, run **make**, specifying **configRecord.xml** as the build target:

```
make configRecord.xml
```

To generate **configRecord.reloc**, run **make**, specifying **configRecord.reloc** as the build target:

```
make configRecord.reloc
```



# 21

## *System Images*

- 21.1 Understanding System Images 141
- 21.2 Planning a System Image 144
- 21.3 Configuring a Network-Loadable System Image 144
- 21.4 Configuring a RAM Payload System Image 147
- 21.5 Configuring a ROM Payload System Image 149
- 21.6 Building a System Image 152

### 21.1 Understanding System Images

To run a module on a target, you must transfer to the target the image files of the various components that make up the module. Image files for the various components of the module are produced by separate build processes as system module (**.sm**) files. Collectively, the various images that make up a module are referred to as the system image. For information on modules, see [2. Understanding VxWorks 653](#) and [19. Modules](#).

To run the module, you must transfer the system image to the target, load it into target memory, and boot it. The software that loads and boots the system image on the target is referred to as a loader. You can use the loader supplied with VxWorks 653 or you can create your own loader. For information on loading and

running a system image, see the *Wind River Workbench User's Guide (VxWorks 653 Version)*.

VxWorks 653 supports three methods for loading the system image on the target:

- network-loadable
- RAM payload
- ROM payload

### Network-loadable

The simplest way of loading the image onto the target is to load it over the network from the host. The method is used mainly used for development and debugging.

Network loading is supported by the boot ROM supplied with VxWorks 653. This boot ROM must be installed on the target in order to use network-loadable or RAM payload system images. To support network loading, the host must be configured with an FTP server to which the boot ROM will connect to download the image.

You may replace the Wind River boot ROM with a custom boot ROM. If you create your own boot ROM, you can use any method you want to load the system image.

To guide the loading and booting of the system image, the boot ROM requires a manifest that tells it which files are part of the system image. This file is called **boot.txt**. **boot.txt** is generated by the network-loadable image build. **boot.txt** is also used by the target agent to communicate with the target for debugging purposes.

Since the target has only the copy of the payload that is being executed, and not an original copy (which would contain, for example, initial values of variables) restart is not available with a network-loadable system image. The only way to restart the system is to reboot, which will download the payload again from the host.

Online-loaded partitions are not supported with a network-loadable system image.

The network-loadable system image is the only type supported by the simulator.

### RAM Payload

A RAM payload system image is one that resides in target RAM rather than on the host. As with a network-loadable image, a RAM payload image is transferred from the host to the target RAM over the network by the boot ROM. However, instead of being loaded and run directly, the payload is loaded in a section of RAM configured as a RAM payload area. Once the RAM payload is loaded in RAM, a copy program is initiated that copies the image into the area of RAM where it will be executed, and then executes it.



RAM payload images are used for development systems where capabilities not supported by the network loaded image are required. These capabilities include cold and warm restart of the module and the use of online-loaded partitions.

Because two copies of a RAM payload image reside in RAM at the same time, the use of a RAM payload requires approximately twice the memory of other image types.

RAM and ROM payloads also reduce the image load time by stripping extraneous ELF sections. Since non-executable sections (**.comment**, **.debug\_info**, and so on) are removed from the image, the size and load time of the final image are reduced.

### **ROM Payload**

A ROM payload is an image that is stored in ROM on the target. The ROM payload includes a boot ROM generated specifically to boot that payload. The boot ROM replaces the default boot ROM that was used to download other image types over the network. ROM payloads are used for deployed systems.

The payload resides permanently on the target in ROM, so no network connection to the host is required to boot the system. When the target is booted, the payload is copied from ROM to RAM and executed. Restart is supported because the original payload is available in ROM.

This image supports:

- all restart types (including power off/on)
- online-loaded partitions

Unlike the network loadable image, which consists of all the individual system module files in the module, and the RAM payload image, which is a single large image file, the ROM payload lets you configure the number and size of the binary files that will make up the system image.

Building a system image is the responsibility of the system integrator. For information on development roles, see [3. Configuration System](#).

## 21.2 Planning a System Image

Generally you will choose the type of system image to build based on where you are in the development process. However, you should bear the following limitation in mind when choosing a downloadable system image:

When using the VxWorks 653 generated **bootApp** to load a network-loadable system image, the boot loader is limited to a maximum of 320 system module (.sm) files. The maximum number of system module files is composed of:

- one for the core OS
- one for each partition
- one for each system shared library
- one for each loadable shared data region
- one for each shared library
- one for the configuration record
- one for each user configuration record

This sum must be less than or equal to 320, otherwise the boot loader will exit with the following error message:

```
Modules to load exceeds limits, abort ...
```

If you have sufficient RAM on your target, you can use a RAM payload system image instead. Otherwise, you will need to use a ROM payload.

## 21.3 Configuring a Network-Loadable System Image

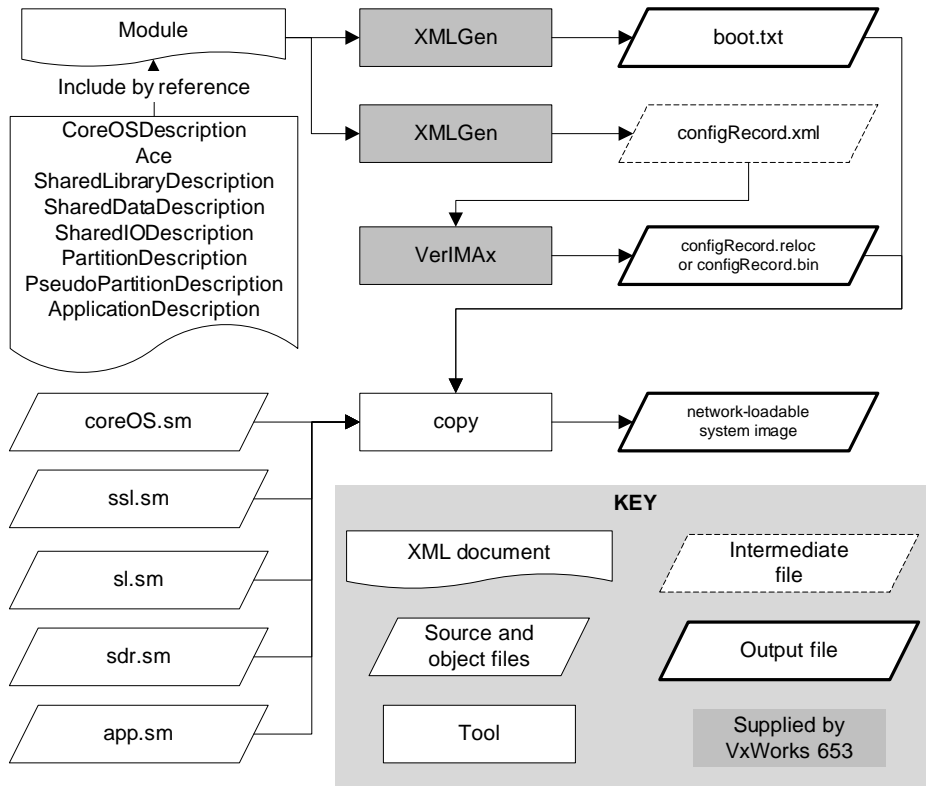
[Figure 21-1](#) summarizes the configuration and build of a network-loadable system image.

The inputs and outputs of the configuration and build of a network loadable image are as follows:

### Outputs of the Network-Loadable System Image Build

The following are the outputs of a network loadable system image build:

Figure 21-1 Configuring and Building Network-Loadable System Image



### boot.txt

boot.txt describes the structure of the system image. It is used by the boot loader to load and boot the image on the target. It is also used by the target server to communicate with the target for debugging purposes.

### Network-loadable System Image

The network-loadable system image is a collection of the system modules for each of the module components, plus the configuration record and **boot.txt**.

### Inputs of the Network-Loadable System Image Build

The following are the inputs of a network loadable system image build:

## The Module Configuration Document

The Module configuration document contains configuration information for the module as a whole. For more information on the Module configuration document, see [19. Modules](#).

## Configuration Files for Module Components

The module build process requires all of the configuration files for the components of the module. The Module configuration document must identify the location of these files via **xi:include** elements in the appropriate places.

- CoreOSDescription document (see [6. Core OS](#))
- Ace configuration document (see [12. ACE](#))
- SharedLibraryDescription documents (see [9. Shared Libraries](#))
- ApplicationDescription documents (see [14. Applications](#))
- PartitionDescription documents (see [15. Partitions](#))
- SharedDataDescription documents (see [10. Shared Data Regions](#))
- SharedIODescription documents (see [11. Shared I/O Regions](#))

## Object Files for Module Components

Your system image brings together all the object files for your core OS, shared libraries, shared data regions, and partitions, as well as the boot loader files for your RAM and ROM payloads. You must provide a rule for each of these files to import them into your build project. In the example below, the variable **\$(IMPORTS)** stands for the location of each of the files to be imported. This may be different in each case.

```
coreOS.sm:          $(IMPORTS)/coreOS.sm          ; $(CP) $< $@
payloadObjs_ram.o:  $(IMPORTS)/payloadObjs_ram.o   ; $(CP) $< $@
payloadObjs_rom.o:  $(IMPORTS)/payloadObjs_rom.o   ; $(CP) $< $@
ssl.sm:             $(IMPORTS)/ssl.sm              ; $(CP) $< $@
part1.sm:           $(IMPORTS)/part1.sm            ; $(CP) $< $@
part2.sm:           $(IMPORTS)/part2.sm            ; $(CP) $< $@
```

Your makefile must define the following variables exactly as shown:

```
SM_FILES = $(shell $(XMLGEN_FILES) $(XML_FILE))
BIN_FILES = $(shell $(XMLGEN_FILES) --bin $(XML_FILE))
```

## configRecord.xml

**configRecord.xml** is an XML file that contains the configuration information calculated by the build tools based on the module configuration document and the

configuration documents for each of the components of the module. For more information on **configRecord.xml**, see [20. Configuration Record](#).

#### **configRecord.reloc or configrecord.bin**

**configRecord.reloc** or **configRecord.bin** is the binary representation of the configuration record. A cert build creates **configRecord.bin**. A debug build produces **configRecord.reloc**. For more information on **configRecord.reloc** and **configRecord.bin**, see [20. Configuration Record](#).

## 21.4 Configuring a RAM Payload System Image

[Figure 21-2](#) summarizes the configuration and build of a RAM payload system image.

Configuring and building a RAM payload system image involves the following items, in addition to those described under [21.3 Configuring a Network-Loadable System Image](#), p.144:

#### **payloadObjs\_ram.o**

**payloadObjs\_ram.o** is the boot loader for a RAM payload system image. It is created as part of the core OS build and should be included in the platform you received from the platform provider.

#### **boot.txt**

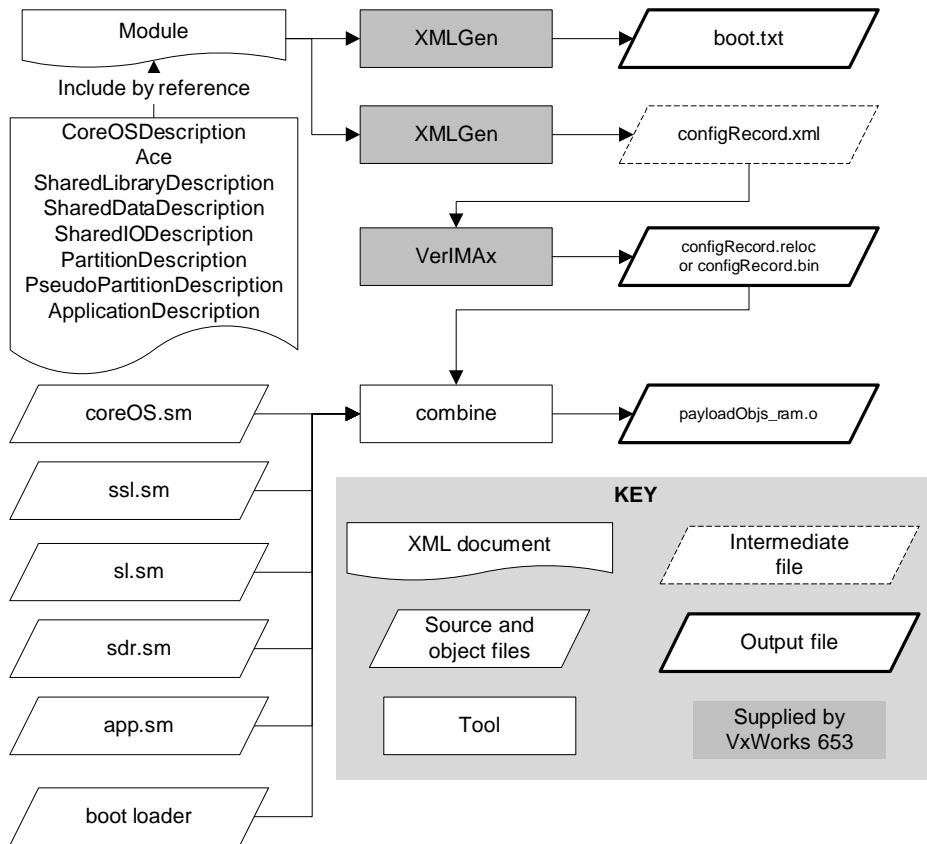
While **boot.txt** is not used to load the system image, as it is with a network-loadable image, it is still required to enable the target server to connect to the target for debugging purposes.

#### **CoreOSDescription Document**

To configure a RAM payload image, you must allocate a RAM payload region in the configuration of the core OS. This requires that you edit the CoreOSDescription document for your platform. If you have questions about this file, consult your platform provider.

To configure a RAM payload region, supply appropriate values for the **Base\_Address** and **Size** attributes of the **CoreOSDescription/**

Figure 21-2 Configuring and Building a RAM Payload System Image



**HardwareConfiguration/PhysicalMemory/ramPayloadRegion** element of *my-coreOS.xml*. For example:

```

<PhysicalMemory>
  <ramPayloadRegion
    Base_Address = "0x0fc00000"
    Size = "0x00400000"/>
</PhysicalMemory>

```

For information on calculating the size of the RAM payload region, see the *VxWorks 653 Configuration and Build Reference* entry for **/CoreOSDescription/HardwareConfiguration/PhysicalMemory/ramPayloadRegion**.

If you need to adjust other core OS settings to accommodate the RAM payload region, see [6. Core OS](#) for details.

### Boot Loader

You must include a RAM payload boot loader in the RAM payload. You can use the boot loader supplied with the platform you are using, or create your own.

## 21.5 Configuring a ROM Payload System Image

[Figure 21-3](#) summarizes the configuration and build of a ROM payload system image.

Configuring and building a ROM payload system image involves the following items, in addition to those described under [21.3 Configuring a Network-Loadable System Image](#), p.144:

### CoreOSDescription Document

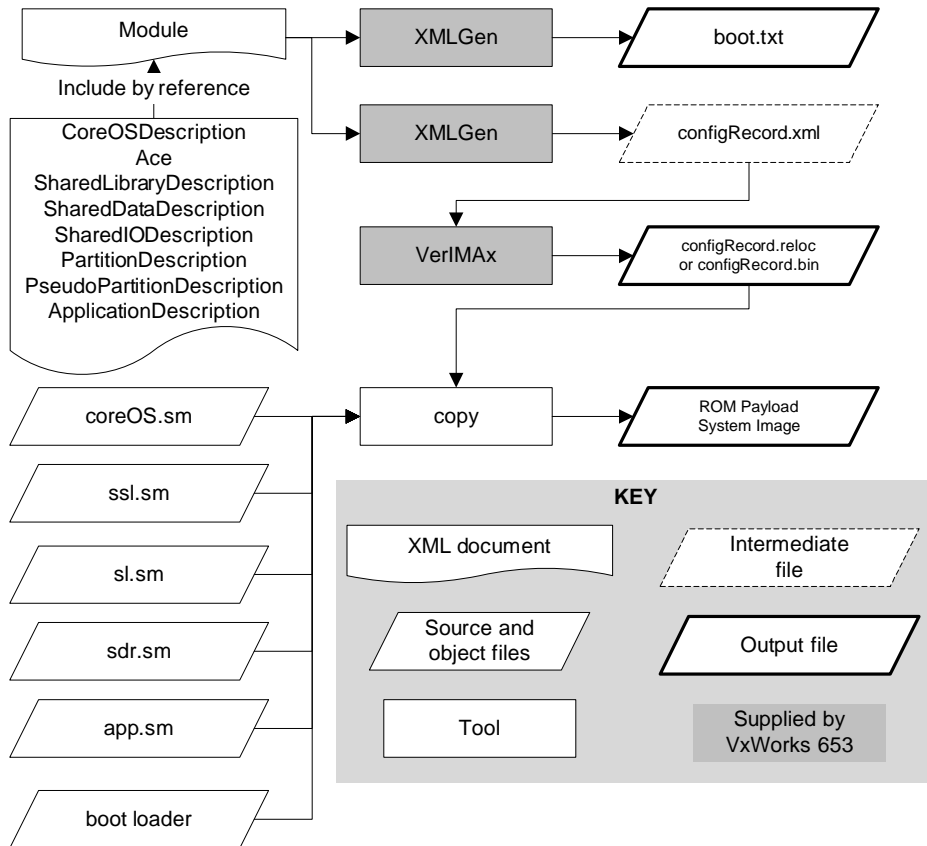
Because a ROM payload image is burned into ROM, you must configure the addresses in ROM where each binary file will be located. The build process will create a separate **.bin** file for each component of your system (core OS, shared library, partition, etc.). You can specify the location of each component separately, or you can allow the build tools to locate some or all components contiguously.

Locate the payload memory configuration of the CoreOSDescription file for your platform. It is located at **CoreOSDescription/HardwareConfiguration/payloadMemory**.

If only one payload memory region is defined, then you can choose either to let the build system locate payload sections or you can locate them yourself. If more than one section is defined, you will need to specify the base addresses of some of your payloads in order to assign them to the correct payload memory region.

If you are configuring a system with more than one bank of payload memory, you must set the base address of the first payload assigned to each bank to match the base address of that bank. You must also be sure that the payloads assigned to each individual bank fit within that bank. If the base address of a payload is not

Figure 21-3 Configuring and Building a ROM Payload System Image



specified, it is placed immediately after the preceding payload in the order in which payloads are defined in the configuration document.

### Module Configuration Document

The **Payloads** element of the Module configuration document defines the location of each payload in ROM. For information on the **Payloads** element, see the *VxWorks 653 Configuration and Build Reference*.



**boot.txt**

While **boot.txt** is not used to load the system image, as it is with a network-loadable image, it is still required to enable the target server to connect to the target for debugging purposes.

**Boot Loader**

You must include a ROM payload boot loader in the ROM payload. You can use the boot loader supplied with the platform you are using, or create your own.

**ROM Payload System Image**

A ROM payload system image consists of the following:

- **payloadObjs\_rom.o** is linked with the payload map to create **sms\_romPayload.hex**.
- *component-name*.**bin**. For each component of the system (core OS, shared library, partition), the build process creates a separate **.bin** file.
- **sms\_romPayload.hex**. This is the bootstrap code that will be called by the board during power up and will copy the payload into RAM and then jump to the start address of the system in RAM.
- **configRecord.bin**. This is the binary configuration record for the system.

Your **.bin** output files must be padded to the size of the black boxes for their components. This is accomplished by adding the following lines, exactly as shown, to your makefile. (**configRecord.xml** is a generated file. This name should not be changed.)

```
BINFLAGS_EXTRA = $(shell $(XMLGEN_BINFLAGS) -j $* configRecord.xml)
$(BIN_FILES): configRecord.xml
```

If binary files do not fit in their defined black boxes, the build will fail. You can add a rule to your makefile to check that the binaries fit their black boxes:

```
check: $(SM_FILES)
    xmlgen --bbCheck $(addprefix -j ,$(basename $^)) configRecord.xml
```

## 21.6 Building a System Image

### Step 1: Create a module makefile.

To build a system image you must create a makefile that will collect the necessary files and call the build tools to generate the image. The following is a typical system image makefile:

```
all: net rom ram checkSize

include $(WIND_BASE)/target/config/make/Makefile.vars

XML_FILE = ../../my-module.xml
SM_FILES = $(shell $(XMLGEN_FILES) $(XML_FILE))
BIN_FILES = $(shell $(XMLGEN_FILES) --bin $(XML_FILE))
SYM_FILES = $(shell $(XMLGEN_FILES) --sym $(XML_FILE))

include $(WIND_BASE)/target/config/make/Makefile.rules

coreOS.sm: $(IMPORTS)/coreOS.sm ; cp $< $@
ssl.sm:    $(IMPORTS)/ssl.sm    ; cp $< $@
part1.sm:  $(IMPORTS)/part1.sm  ; cp $< $@

# Downloadable symbol tables

net: $(SYM_FILES)

checkSize: $(SM_FILES)
    xmlgen --bbCheck $(addprefix -j ,$(basename $^)) configRecord.xml
```

For other examples of a system image makefile, see [22.8 Integration](#), p.203.

### Step 2: Open the VxWorks 653 Development Shell.

The VxWorks 653 build tools require a specific build environment which is provided by the VxWorks 653 Development Shell. To open the shell, from your program list, select:

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

### Step 3: Build the system image.

To build the system image, run **make**, specifying the correct target for the image you want to build.

For a network loadable image:

```
make net
```

For a RAM payload image:

```
make ram
```

For a ROM image:

```
make rom
```

**Step 4: Generate the boot.txt file.**

If you intend to connect to a running RAM or ROM payload system image via the target server, you will need a **boot.txt** file. To generate a **boot.txt** file, run **make** with the **boot.txt** target:

```
make boot.txt
```



# 22

## *Reference Process*

22.1 Introduction	155
22.2 Quick Start	158
22.3 Hello World	159
22.4 Module OS	179
22.5 Partition OS	185
22.6 Application	191
22.7 Shared Library	196
22.8 Integration	203

### **22.1 Introduction**

This chapter describes the VxWorks 653 configuration and build reference process.

There are eight parts to the build of a VxWorks 653 module:

- module OS build
- ACE build (if ACE is used)
- partition OS build
- shared library build (if shared libraries are used)

- shared data build (if loadable shared data regions are used)
- user configuration record build (if user configuration records are used)
- application build
- integration build (which includes the configuration record and system image builds)

The reference process provides working examples of each of these parts and the common variants of these builds. You can use the reference process as a means to learn about the VxWorks 653 build process or as a starting point for creating your own build system for your projects. For more information on the build process, see [4. Build System](#).

The reference process consists of the following examples, which are located at:

*installDir/vxworks653-2.2/target/reference/helloWorld*

### **introduction**

The introduction example contains a basic Hello World program. It consists of a module OS, partition OS, application, and integration build.

### **moduleOS**

The moduleOS example contains several ways of building a module OS, including:

- hello-ref, a basic module OS build (the same one shown in the introduction)
- hello-ace, a module OS with ACE
- hello-bincomp, a module OS with optional binary components
- hello-srccomp, a module OS with optional source components
- hello-full, the full hello-world example set up to export various components of the hello-world module for use with the different variations of the module OS
- hello-cert, a module OS built in cert mode

### **partitionOS**

The partition OS example contains several ways of building a partition OS

- hello-ref, a basic partition OS build (the same one shown in the introduction)
- hello-sd, a partition OS that uses a shared data region
- hello-bincomp, a partition OS with optional binary components

- hello-srccomp, a partition OS with optional source components
- hello-full, the full hello-world example set up to export various components of the hello-world module for use with different variations of the partition OS
- hello-cert, partition OS built in cert mode

### **application**

The application example contains several ways of building an application.

- hello-ref, a basic application build (the same one shown in the introduction)
- hello-cpp, the Hello World application written in C++
- hello-two, an example with two Hello World programs running in separate partitions
- hello-full, the full hello-world example set up to export various components of the hello-world module for use with different variations of the application
- hello-cert, an application built in cert mode

### **sharedLibrary**

The sharedLibrary example contains the following examples of the use of a shared library

- hello-ref, the hello-two example from the application example, showing two Hello World programs with duplicated code
- hello-sl, an example in which the Hello World code is moved to a shared library and is called by the two applications
- hello-version, an example which shows how you can define more than one interface for a library
- hello-full, the full hello-world example set up to export various components of the hello-world module for use with different variations of the shared library

### **integration**

The integration example shows the different types of integration builds:

- hello-net, a downloadable system image build
- hello-ram, a RAM payload build
- hello-rom, a ROM payload build

- `hello-full`, a full build of all the elements of a downloadable system image build

This chapter will walk you through these examples so that you can understand how they work. For additional information on any of the settings here, see the topic on the specific feature in this document, or consult the appropriate setting in the *VxWorks 653 Configuration and Build Reference*.

## 22.2 Quick Start

You will receive the maximum benefit from the reference process if you work your way thorough each part of the build process separately, observing how the different parts relate to each other. However, if you want to get something running on your board as quickly as possible, you can use the following quick start procedure to run the entire build process and produce a network loadable system image that you can run on your target:

1. Choose which target you want to build for and identify the appropriate BSP files in:

`installDir/vxworks653-2.2/target/config`

If you are unsure which BSP to use, consult the platform provider.

2. In the BSP directory for your chosen BSP, open the default `SharedLibraryDescription` file for the partition OS, which is named `BSP_Name_default.xml` (for example, `wrSbc750gx_default.xml`). Note the value of the attribute `SharedLibraryDescription/@VirtualAddress`. This value is the default virtual address for the partition OS for your target. You will need this value to build all the use cases in the reference process.

3. Open this file:

`installDir/vxworks653-2.2/target/reference/helloWorld/introduction/hello-full/pos/hello-pos.xml`

and locate the attribute `SharedLibraryDescription/@VirtualAddress`. Its current value is `"$(SSLADDR)"`. Replace this value with the partition OS virtual address that you identified in step 2.

4. Determine the CPU for your target. For instance, if your BSP is `wrSbc750gx`, your CPU will be `PPC604`. If you are unsure of the name of the CPU for your



BSP, open the **Makefile** in the BSP directory and find the value of the CPU variable.

5. Determine the partition virtual address for your target. It can be found in the attribute **CoreOSDescription/KernelConfiguration/@partitionVirtualAddress** in the default CoreOSDescription document. This file is found in your BSP directory and has the same name as the BSP directory with an **.xml** extension. For the wrSbc750gx, the value is 0x40000000.
6. Open the VxWorks 653 Development Shell.
7. Change your current directory to **introduction/hello-full**.
8. Run **make**, specifying the values of the variables **CPU**, **BSP**, and **PARTADDR** (partition virtual address). For example:

```
make BSP=wrSbc750gx CPU=PPC604 PARTADDR=0x40000000
```

The makefile will run the makefiles for each part of the build. This will create a network-loadable system image in the directory:

```
installDir/vxworks653-2.2/target/reference/helloWorld/introduction/hello-full/int/demo
```

For instructions on how to load and run this system image on your target, see the *Wind River Workbench User's Guide 2.6.1 (VxWorks 653 edition)*.

9. If you want to remove the built files so that you can perform each of the build steps separately, run:

```
make clean
```

## 22.3 Hello World

The reference process is built around a simple Hello World program. The introduction section of the reference process contains all the build steps required to build a basic no-frills Hello World application, as well as a partition OS for the application to run on and a core OS to manage the module and schedule the Hello World application to run.

There are four parts to the build of this example:

- module OS build

- partition OS build
- application build
- integration build

This example illustrates an important feature of VxWorks 653. Once the core OS is built, its binary file can be integrated with the other components without change. The same is true for each of the other components. With only a minimum of constraints, each can be built independently of the others and integrated without change to create a complete module. This means that once a component is built and certified, it does not have to be rebuilt or recertified when another component changes.

## Module OS Build

In a VxWorks 653 module, the module OS is provided by the VxWorks 653 core OS. For detailed information on the building of the core OS and the various files needed to build a core OS, see [6. Core OS](#).

The module OS for the Hello World example consists of the default core OS configuration provided by VxWorks 653 and the default set of core OS components with no added user code. This means that there is no need to create a CoreOSDescription document for this example. You can use the default CoreOSDescription document from the appropriate BSP.

The module OS example therefore requires only a makefile. That makefile looks like this:

```
include $(WIND_BASE)/target/config/make/Makefile.vars

all:

# Create the core OS project
prjCreate -domtype kernel -prj demo/ -bsp $(BSP) -name coreOS

# Build the core OS
cp $(WIND_BASE)/target/config/$(BSP)/$(BSP).xml demo/bsp.xml
xmlgen --ldScript --arch $(TOOLARCH) -o demo/coreOS.lds demo/bsp.xml
make -C demo ADD_NEEDED
make -C demo

clean:
rm -rf demo
```

There are a number of things to note in this makefile:

```
include $(WIND_BASE)/target/config/make/Makefile.vars
```

**Makefile.vars** is a makefile provided with VxWorks 653 that contains a number of variable definitions that are used to build various components of a VxWorks 653

module. You must include the appropriate version of **Makefile.vars** in your makefile. The variable **\$(WIND\_BASE)** is defined in the environment of the VxWorks 653 Development shell and refers to the directory where VxWorks 653 is installed on your system.

```
all:
```

```
# Create the core OS project
prjCreate -domtype kernel -prj demo/ -bsp $(BSP) -name coreOS
```

The VxWorks 653 core OS is built using the VxWorks 653 project facility, which creates and populates a project structure for the core OS and produces a makefile to build the core OS.

The **prjCreate** utility creates a core OS project. The **-domtype** options specifies that we are creating a kernel project. The **-prj** option specifies the location in which the core OS project is created. The **-bsp** options specify the BSP to build for. You must supply the value of the **\$(BSP)** variable on the command line. The **-name** option specifies the name of the coreOS.

```
cp $(WIND_BASE)/target/config/$(BSP)/$(BSP).xml demo/bsp.xml
```

The **cp** command copies the correct CoreOSDecription document for your target into the project directory. (If you needed to change the configuration of the core OS, you would need to create your own version of the CoreOSDecription document rather than copying the one supplied with the BSP for your target.)

```
xmlgen --ldScript --arch $(TOOLARCH) -o demo/coreOS.lds demo/bsp.xml
```

The **xmlgen --ldScript** command generates a linker script for the core OS project. This is required to support the black box feature that is used to configure memory in VxWorks 653. For more information on black box memory configuration, see [5. Memory](#).

```
make -C demo ADD_NEEDED
```

Calling the **ADD\_NEEDED** target in the generated makefile makes sure that all component dependencies in the core OS configuration are accounted for, and adds any components that are needed.

```
make -C demo
```

Finally, calling the generated makefile causes the core OS to be built. The built files will be placed in a directory named for the build spec used. This directory will be located under the project directory. For instance, if you specified the PPC604 CPU as the value of the **\$(CPU)** variable on the command line, the built files will be found in the directory **demo\coreos\PPC604gnu.debug**. The file that contains the core OS system image is **coreOS.sm** (unless you changed the name of the project

in the **prjCreate** command, in which case the file name will match the value you gave there).

```
clean:
    rm -rf demo
```

The clean target will remove the demo directory and its contents.

### Building the Module OS

To build the module OS, use the following procedure:

1. Open the VxWorks 653 Development Shell.
2. Change your current directory to **introduction/hello-full/mos**.
3. Run **make**, specifying the values of the variables **\$(CPU)** and **\$(BSP)**. For example:

```
make BSP=wrSbc750gx CPU=PPC604
```

The makefile will create a core OS project and then run the makefile that project contains, creating a system module file and supporting files for the core OS.

### Partition OS Build

The partition OS for the Hello World program is provided by the vThreads partition OS that is part of VxWorks 653. The partition OS is contained in a system shared library. The Hello World example uses the default vThreads partition OS without any additional components.

The Partition OS build involves the following files:

#### SharedLibraryDescription Document

The SharedLibraryDescription document (**hello-pos.xml**) contains configuration information for the library that will form part of the system configuration record. It looks 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
Application.xsd"
  SystemSharedLibrary="true"
  VirtualAddress="$(SSLADDR) ">
<MemorySize
  MemorySizeBss="0x10000"
  MemorySizeText="0x40000"
  MemorySizeData="0x10000"
```

```
MemorySizeRoData="0x10000"
/>
</SharedLibraryDescription>
```

For detailed information on the content of this file, see the *VxWorks 653 Configuration and Build Reference* entry for the SharedLibraryDescription document type.

There are three things to notice about this file:

```
SystemSharedLibrary="true"
```

This specifies that the library is a system shared library rather than a regular shared library. A partition OS resides in a system shared library.

```
VirtualAddress="$ (SSLADDR) ">
```

This line specifies the virtual address of the system shared library within the module. To build the example, you will need to substitute an appropriate value for your module in place of “\$(SSLADDR)”. An appropriate value can be found in the default BSP Module configuration file (*BSP\_default.xml*) located in the BSP directory for your BSP.

```
<MemorySize
MemorySizeBss="0x10000"
MemorySizeText="0x40000"
MemorySizeData="0x10000"
MemorySizeRoData="0x10000"
/>
```

These lines define the black box for the system shared library. The values shown are correct for the **simpc** BSP and the default vThreads partition OS object file, **vThreadsComponent.o**.

### **Shared\_Library\_API Document**

The Shared\_Library\_API document (**pos-api.xml**) contains information that is used to build entry-point tables to enable applications to link to the routines in the shared library. It looks like this:

```
<Shared_Library_API
xmlns="http://www.windriver.com/vxWorks653/SharedLibraryAPI"
xmlns:xi="http://www.w3.org/2001/XInclude"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.windriver.com/vxWorks653/SharedLibraryAPI"
Name="vThreads">
<Interface>
<Version Name="hello"/>
<Interface_Subset>
<Routine Name="printf"/>
</Interface_Subset>
</Interface>
</Shared_Library_API>
```

There are three things to notice about this file:

```
Name="vThreads"
```

This is the name of the shared library API, which is not the same thing as the name of the library. The name of the API is used at runtime to identify the correct shared library entry-point table for a routine.

```
<Version Name="hello"/>
```

This is the name of the version of the API. It is possible to define more than one version of an API, each with a different interface. When more than one interface is defined, the version name is used to identify the interface to select when building the entry-point table.

```
<Interface_Subset>  
  <Routine Name="printf"/>  
</Interface_Subset>
```

These lines define an interface subset. An interface is made up of one or more interface subsets. An interface subset is made up of one or more routines. In this case, only one routine is needed to support the Hello World applications, the **printf()** routine. While the vThreads partition OS contains many other routines, only those defined in the selected **Interface** in the Shared\_Library\_API document are available to the application.

### Partition OS Makefile

The partition OS makefile (**makefile.pos**) contains the commands required to build the partition OS. It looks like this:

```
all: pos.sm pos-stubs.o

include $(WIND_BASE)/target/vThreads/config/make/Makefile.vars

vpath %.c $(WIND_BASE)/target/vThreads/config/comps/src
vpath %.o $(WIND_BASE)/target/vThreads/lib/obj$(CPU)gnuvx

pos.sm: sslMain.o vThreadsComponent.o pos-ept.o pos.lds
$(LD) $(LDFLAGS) -T pos.lds -o $$@ $(filter %.o,$^)

%.o: %.c
$(CC) $(CFLAGS) -c -o $$@ $$<

pos-ept.c: pos-api.xml
xmlgen --linkage --output-entrypoints $$@ $$<

pos-stubs.c: pos-api.xml
xmlgen --linkage --arch $(TOOLARCH) --output-stubs $$@ $$<

pos.lds: hello-pos.xml
xmlgen --ldScript --arch $(TOOLARCH) -o $$@ $$<
```

There are several things to note about this file:

```
all: pos.sm pos-stubs.o
```

The partition OS build process produces two outputs, the partition OS system module file (**.sm** extension) and the stubs file for the shared library (**-stubs.o** extension). The system module file is a fully linked object file in ELF format that is ready to be used in a VxWorks 653 module. The stubs file must be linked into any application that uses the partition OS.

```
include $(WIND_BASE)/target/vThreads/config/make/Makefile.vars
```

**Makefile.vars** is provided with VxWorks 653. It includes a number of variables used in building parts of a VxWorks 653 module.

```
vpath %.c $(WIND_BASE)/target/vThreads/config/comps/src
vpath %.o $(WIND_BASE)/target/vThreads/lib/obj$(CPU)gnuvx
```

These lines establish the path to the object files supplied with VxWorks 653 that will be built into the system shared library

```
pos.sm: sslMain.o vThreadsComponent.o pos-ept.o pos.lds
```

The shared library object file depends on the following:

- **sslMain.o**, a file supplied as part of VxWorks 653, which provides initialization for system shared libraries.
- **vThreadsComponent.o**, the vThreads component file, which is supplied with VxWorks 653
- **pos-ept.o**, the entry-point table, which is generated from the information in the Shared\_Library\_API document
- **pos.lds**, the linker script, which is generated from information in the SharedLibraryDescription document, and which ensures that the sections of the shared library binary file are aligned on the proper boundaries specified in the black box

```
$(LD) $(LDFLAGS) -T pos.lds -o $@ $(filter %.o,$^)
```

This line calls the linker.

```
pos-ept.c: pos-api.xml
xmlgen --linkage --output-entrypoints $@ $<
```

These lines generate the entry-point table from information in the Shared\_Library\_API document.

```
pos-stubs.c: pos-api.xml
xmlgen --linkage --arch $(TOOLARCH) --output-stubs $@ $<
```

These lines generate the stubs file from information in the Shared\_Library\_API document.

```
pos.lds: hello-pos.xml
        xmlgen --ldScript --arch $(TOOLARCH) -o $@ $<
```

These lines generate the linker script from information in the SharedLibraryDescription document.

For general information on the shared library configuration and build process, see [9. Shared Libraries](#).

### Project makefile

The project makefile (**makefile**) contains rules to establish a project to build the partition OS. It looks like this:

```
all:

# Create the partition OS project
mkdir -p demo
cp Makefile.pos demo/Makefile
cp hello-pos.xml demo
cp pos-api.xml demo

# Build the partition OS
make -C demo

clean:
rm -rf demo
```

The “all” rule copies all the required file to the demo directory and calls **make** in that directory.

### Building the Partition OS

To build the partition OS, use the following procedure:

1. Edit the SharedLibraryDescription document to insert the appropriate value for the **VirtualAddress** attribute.
2. Open the VxWorks 653 Development Shell.
3. Change your current directory to **introduction/hello-full/pos**.
4. Run **make**, specifying the values of the variables **\$(CPU)** and **\$(BSP)**. For example:

```
make BSP=wrSbc750gx CPU=PPC604
```

The **pos.sm** and **pos-stubs.o** files will be created in the project directory.



## Application Build

The application part of the introduction example is found in the directory **hello-full/app**. It includes the following files:

### Application Source File

The Hello World application is a simple C language file:

```
#include <stdio.h>

void usrAppInit (void)
{
    printf ("Hello, world!\n");
}
```

The main thing to note about this file is the name of the routine. It is called **usrAppInit**. This is the default name for the initialization routine in VxWorks 653. VxWorks 653 will call the **usrAppInit** routine of each application when it is started. You can change the name of this routine if you want, but if you do so, you will need to tell VxWorks 653 about the new routine name. For an example of this, see [14. Applications](#).

### ApplicationDescription Document

The ApplicationDescription document (**hello-app.xml**) contains configuration information for the library that will form part of the system configuration record. It looks like this:

```
<ApplicationDescription
  xmlns="http://www.windriver.com/vxWorks653/ConfigRecord"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.windriver.com/vxWorks653/ConfigRecord
Application.xsd">
  <MemorySize
    MemorySizeBss="0x10000"
    MemorySizeText="0x10000"
    MemorySizeData="0x10000"
    MemorySizeRoData="0x10000"/>
</ApplicationDescription>
```

The ApplicationDescription document contains the memory black box definition for the Hello World application.

### Application Makefile

The application makefile (**makefile.app**), contains the commands needed to build the application. It looks like this:

```
all: hello.sm
```

```
include $(WIND_BASE)/target/vThreads/config/make/Makefile.vars

POS_DIR    = ../../pos/demo

vpath %.c   $(WIND_BASE)/target/vThreads/config/comps/src
vpath %.o   $(POS_DIR)

hello.sm: vxMain.o hello.o pos-stubs.o hello.lds
    $(LD) $(LDFLAGS) -T hello.lds -o $$ $(filter %.o,$^)

%.o: %.c
    $(CC) $(CFLAGS) -c -o $$ $<

hello.lds: hello-app.xml
    xmlgen --ldScript --arch $(TOOLARCH) --address $(PARTADDR) -o $$ $<
```

There are several things to note about this makefile:

```
POS_DIR = ../../pos/demo
```

The application makefile needs to access files from the partition OS build. The **\$(POS\_DIR)** variable identifies the location of the partition OS build files. Note that since the application makefile will be copied into the application project directory and executed there, this path must be relative to the application project directory, in this case, **app/demo**.

```
vpath %.c   $(WIND_BASE)/target/vThreads/config/comps/src
```

This line establishes the path to the vThreads code. This is needed to locate the file **vxMain.c**, which contains initialization code for applications. This file must be compiled into all VxWorks 653 applications.

```
vpath %.o $(POS_DIR)
```

This line provides the path to the directory where the partition OS files reside so that the build can locate the **pos-stubs.o** file.

```
hello.sm: vxMain.o hello.o pos-stubs.o hello.lds
```

The application system module file (**hello.sm**) is created by linking **vxMain.o**, the VxWorks 653 application initialization code; **hello.o**, the application object file; **pos-stubs.o**, the partition OS stubs file, using **hello.lds**, the application linker script.

```
$(LD) $(LDFLAGS) -T hello.lds -o $$ $(filter %.o,$^)
```

This line calls the linker to create the application system module file.

```
hello.lds: hello-app.xml
    xmlgen --ldScript --arch $(TOOLARCH) --address $(PARTADDR) -o $$ $<
```

These lines create the application linker script (**hello.lds**) from information in the ApplicationDescription document. The partition virtual address of the core OS is also required to link the application. The partition virtual address is defined in the

core OS configuration in the CoreOSDescription document in **CoreOSDescription/KernelConfiguration/@partitionVirtualAddress**. You should refer to the CoreOSDescription document for your core OS (which, in this case, is **bsp.xml** in the **mos/demo** directory) to determine the correct partition virtual address. You supply the value of the **\$(PARTADDR)** variable on the command line.

## Building the Application

To build the application, use the following procedure:

1. Open the CoreOSDescription document for your platform (**hello-full/mos/bsp.xml**). Locate the value for the partition virtual address. It is located in the attribute:

**CoreOSDescription/KernelConfiguration/@partitionVirtualAddress**

2. Open the VxWorks 653 Development Shell.
3. Change your current directory to **introduction/hello-full/app**.
4. Run **make**, specifying the values of the variables **CPU**, **BSP**, and **PARTADDR**. For example:

```
make BSP=wrSbc750gx CPU=PPC604 PARTADDR=0x40000000
```

The application system module file (**hello.sm**), is created.

## Integration Build

The final step in creating a VxWorks 653 module is the integration build. The integration build creates the system configuration record and the system image which you load onto your target and execute.

The integration build files are located in the directory **hello-full/int**. They include:

### Module Configuration Document

The Module configuration document (**hello-module.xml**) contains configuration information for the module that will form part of the system configuration record. It looks like this:

```
<Module
  xmlns="http://www.windriver.com/vxWorks653/ConfigRecord"
  xmlns:xi="http://www.w3.org/2001/XInclude"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.windriver.com/vxWorks653/ConfigRecord
Module.xsd">
  <CoreOS>
    <xi:include href="bsp.xml"/>
```

```
</CoreOS>
<Applications>
  <Application Name="hello">
    <xi:include href="hello-app.xml" />
  </Application>
</Applications>
<SharedDataRegions/>
<SharedLibraryRegions>
  <SharedLibrary Name="pos">
    <xi:include href="hello-pos.xml" />
  </SharedLibrary>
</SharedLibraryRegions>
<Partitions>
  <Partition Name="hello" Id="1">
    <xi:include href="hello-part.xml" />
  </Partition>
</Partitions>
<Schedules>
  <Schedule Id="0">
    <PartitionWindow PartitionNameRef="hello" Duration="0.10" />
  </Schedule>
</Schedules>
<HealthMonitor>
  <SystemHMTable Name="systemHm">
    <SystemState SystemState="HM_PARTITION_MODE">
      <ErrorIDLevel
        ErrorIdentifier="HME_DEFAULT"
        ErrorLevel="HM_PARTITION_LVL" />
    </SystemState>
    <SystemState SystemState="HM_MODULE_MODE">
      <ErrorIDLevel
        ErrorIdentifier="HME_DEFAULT"
        ErrorLevel="HM_MODULE_LVL" />
    </SystemState>
    <SystemState SystemState="HM_PROCESS_MODE">
      <ErrorIDLevel
        ErrorIdentifier="HME_DEFAULT"
        ErrorLevel="HM_PROCESS_LVL" />
    </SystemState>
  </SystemHMTable>
  <ModuleHMTable Name="moduleHm">
    <SystemState>
      <ErrorIDAction ErrorIdentifier="HME_DEFAULT" ErrorAction="" />
    </SystemState>
    <Settings
      stackSize="0x0400"
      maxQueueDepth="2"
    />
  </ModuleHMTable>
  <PartitionHMTable Name="helloHm">
    <SystemState>
      <ErrorIDAction ErrorIdentifier="HME_DEFAULT" ErrorAction="" />
    </SystemState>
    <Settings
      stackSize="0x0400"
      maxQueueDepth="2"
```

```

        />
    </PartitionHMTTable>
</HealthMonitor>
<Payloads>
    <CoreOSPayload/>
    <SharedLibraryPayload NameRef="pos"/>
    <ConfigRecordPayload NameRef="configRecord"/>
    <PartitionPayload NameRef="hello"/>
</Payloads>
</Module>

```

There are several things to note in this file:

```

<CoreOS>
    <xi:include href="bsp.xml"/>
</CoreOS>

```

These lines import the CoreOSDescription document for the core OS into the module configuration. The project makefile will copy this document into the project directory.

```

<Applications>
    <Application Name="hello">
        <xi:include href="hello-app.xml"/>
    </Application>
</Applications>

```

These lines name the application and import the ApplicationDescription document. Note that applications and shared libraries are not named in their SharedLibraryDescription and ApplicationDescription files respectively. They are given names in the Module configuration document for a particular module. Both shared libraries and applications may be used in more than one module and they are named in the module that use them in a way that is appropriate for that particular module. This avoids the possibility of name collisions when shared libraries and applications are reused.

```

<SharedDataRegions/>

```

The VxWorks 653 Configuration Schema requires that the **SharedDataRegions** element be present in the Module configuration. Since there is no shared data region in this module, the **SharedDataRegions** element is empty.

```

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

```

These lines import the SharedLibraryDescription document for the system shared library.

```

<Partitions>
    <Partition Name="hello" Id="1">

```

```
<xi:include href="hello-part.xml" />
</Partition>
</Partitions>
```

These lines name the partition and import the PartitionDescription document. For more information on partitions, see [15. Partitions](#).

```
<Schedules>
  <Schedule Id="0">
    <PartitionWindow PartitionNameRef="hello" Duration="0.10"/>
  </Schedule>
</Schedules>
```

These lines establish the schedule for the module. Since there is only one application in this module, there is only one entry in the schedule. Since the Hello World application does not do anything other than print “Hello World” when it is initialized, the duration of its schedule has no impact on its operation. However, the application must be scheduled if it is to run at all. The only way that an application can run in a VxWorks 653 module is if it is scheduled in the current schedule.

It is possible to define more than one schedule, and the core OS can change schedules at run time. For the Hello World module, however, only one schedule is required. For more information on schedules, see [17. Schedules](#).

```
<HealthMonitor>
  <SystemHMTable Name="systemHm">
    ...
  </SystemHMTable>
  <ModuleHMTable Name="moduleHm">
    ...
  </ModuleHMTable>
  <PartitionHMTable Name="helloHm">
    ...
  </PartitionHMTable>
</HealthMonitor>
```

These lines set up a health monitor for the module. Note that the PartitionHMTable is named “helloHm”. This is the health monitor table name that was used in the PartitionDescription document to identify the partition health monitor table to be used for the partition. For more information on health monitoring, see [18. Health Monitor](#).

```
<Payloads>
  <CoreOSPayload/>
  <SharedLibraryPayload NameRef="pos"/>
  <ConfigRecordPayload NameRef="configRecord"/>
  <PartitionPayload NameRef="hello"/>
</Payloads>
```

These lines configure the payload regions for the module. In this example, no payload settings are specified, which leaves the location and size of payloads up to

the build tools. The shared library, config record and partition payloads have a **NameRef** attribute to identify the component that will reside in the payload. There is no application payload, since the application resides in the partition. There is no **NameRef** attribute on the **CoreOSPayload** element, since there is only one core OS in a module. There is a **NameRef** on the **ConfigurationRecordPayload** element because while there is only one system configuration record, and its name is always “configRecord”, VxWorks 653 also supports user configuration records, which also have to be mapped to configuration record payloads, if they exist. For more information on payloads, see [21. System Images](#).

### PartitionDescription Document

The PartitionDescription document (**hello-part.xml**) describes the configuration of the partition in which the application will reside. It looks like this:

```
<PartitionDescription
  xmlns="http://www.windriver.com/vxWorks653/ConfigRecord"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.windriver.com/vxWorks653/ConfigRecord
Partition.xsd">
  <Application NameRef="hello"/>
  <SharedLibraryRegion NameRef="pos"/>
  <Settings
    RequiredMemorySize="0x100000"
    PartitionHMTTable="helloHm"
    numFiles="0xffffffff"
    numDrivers="0xffffffff"
    isrStackSize="0xffffffff"
    maxEventQStallDuration="INFINITE_TIME"/>
</PartitionDescription>
```

There are several things to note in this file:

```
<Application NameRef="hello"/>
```

The **Application** element identifies the application that will reside in the partition. In this case the value is “hello”, which is the name given to the application in the Module configuration document.

```
<SharedLibraryRegion NameRef="pos"/>
```

The **SharedLibraryRegion** element identifies the system shared library that contains the partition OS for the partition. In this case the value is “pos”, which is the name given to the system shared library in Module configuration document.

```
RequiredMemorySize="0x100000"
```

The **RequiredMemorySize** attribute defines the total memory requirement for the partition. This includes the total size of the application, as defined by its memory black box, as well as the stack and heap space required to run the application and any library code that the application uses.

```
PartitionHMTTable="helloHm"
```

The **PartitionHMTTable** attribute identifies the partition health monitor table that will be use to route health monitor events that are dispatched to the partition level. The value in this case is “helloHm”, which must match the value of a partition health monitor table name in the Module configuration document.

```
numFiles="0xffffffff"  
numDrivers="0xffffffff"  
isrStackSize="0xffffffff"
```

The value “0xffffffff” sets these values to the system defaults. For information on these settings, see the *VxWorks 653 Configuration and Build Reference*.

```
maxEventQStallDuration="INFINITE_TIME" />
```

For information on this setting, see the *VxWorks 653 Configuration and Build Reference*.

### Project makefile

The project makefile (**makefile**) creates the project directory, copies files from the various sources to the project directory, and runs the integration makefile. It looks like this:

```
# Create the integration project  
mkdir -p demo  
cp Makefile.int demo/Makefile  
cp hello-module.xml demo  
cp hello-part.xml demo  
cp ../mos/demo/bsp.xml demo  
cp ../pos/demo/hello-pos.xml demo  
cp ../app/demo/hello-app.xml demo  
  
#Integrate  
make -C demo  
  
clean:  
rm -rf demo
```

### Integration makefile

The integration makefile (**makefile.int**) contains the commands required to combine the various elements of the module together to form a payload that can be transferred to a target and executed. It also builds the system configuration record. It looks like this:

```
all: net checkSize  
  
include $(WIND_BASE)/target/config/make/Makefile.vars  
  
XML_FILE = hello-module.xml
```



```
SM_FILES = $(shell $(XMLGEN_FILES) $(XML_FILE))
SYM_FILES = $(shell $(XMLGEN_FILES) --sym $(XML_FILE))

MOS_DIR = ../../mos/demo
POS_DIR = ../../pos/demo
APP_DIR = ../../app/demo

include $(WIND_BASE)/target/config/make/Makefile.rules

coreOS.sm: $(MOS_LOCATION)/$(CPU)gnu.debug/coreOS.sm ; cp $< $@
pos.sm:    $(POS_LOCATION)/pos.sm ; cp $< $@
hello.sm:  $(APP_LOCATION)/hello.sm ; cp $< $@

# Downloadable symbol tables

net: $(SYM_FILES)

checkSize: $(SM_FILES)
    xmlgen --bbCheck $(addprefix -j ,$(basename $^)) configRecord.xml
```

There are several things to note in this file:

```
all: net checkSize
```

The integration makefile creates a network loadable payload and runs a check to make sure that the elements of the payload fit into their assigned black boxes.

```
XML_FILE = hello-module.xml
SM_FILES = $(shell $(XMLGEN_FILES) $(XML_FILE))
SYM_FILES = $(shell $(XMLGEN_FILES) --sym $(XML_FILE))
```

These commands call **XMLGen** to create a list of system module files and a list of symbol files. The list of system module files is used to assemble all the components of the module in the system image. The list of symbol files is used by the target shell and by the debugger (running on the host) to locate symbols in the runtime (running on the target).

```
MOS_DIR = ../../mos/demo
POS_DIR = ../../pos/demo
APP_DIR = ../../app/demo
```

These variables point to the location of the module OS, partition OS, and application build files. The integration makefile will be copied to the project directory before it is executed, therefore these paths are expressed relative to the project directory (demo), which is one level down the directory tree.

```
include $(WIND_BASE)/target/config/make/Makefile.rules
```

This line includes **Makefile.rules**. **Makefile.rules** is provided with VxWorks 653 and defines rules for building portions of the module.

```
coreOS.sm: $(MOS_LOCATION)/$(CPU)gnu.debug/coreOS.sm ; cp $< $@
pos.sm:    $(POS_LOCATION)/pos.sm ; cp $< $@
```

```
hello.sm: $(APP_LOCATION)/hello.sm ; cp $< $@
```

These lines copy the system module files for the core OS, system shared library, and application to the current directory.

```
net: $(SYM_FILES)
```

This line specifies the creation of symbol files.

```
checkSize: $(SM_FILES)
    xmlgen --bbCheck $(addprefix -j ,$(basename $^)) configRecord.xml
```

This rule is used to check that the sizes of the ELF sections of the system module files fit into the black boxes specified in the configuration documents. The sizes are checked against the generated configuration file **configRecord.xml**, which is created as part of the build.

### Building the System Image

To build the system image, use the following procedure:

1. Open the VxWorks 653 Development Shell.
2. Change your current directory to **introduction/hello-full/int**.
3. Run **make**, specifying the values of the variable CPU. For example:

```
make CPU=PPC604
```

A network loadable system image for the Hello World module is created.

For instructions on running this system image on your target, see the *Wind River Workbench User's Guide 2.6.1, VxWorks 653 Version*.

#### 22.3.1 Hello World cert

The introduction example also contains a cert version of Hello World. The cert version of a module differs from a debug version in the following ways:

- Different compiler flags are used to compile source components.
- Different object files are used to include binary components.
- Some binary components may not be available in cert.

The differences between the debug and cert version of Hello World are as follows:

## Module OS cert Differences

To build a cert version of the VxWorks 653 core OS, the following lines are added or changed in the core OS makefile:

```
prjCreate -domtype kernel -ddf certKernel -prj demo/coreos -bsp $(BSP) \
  -name coreOS
prj projBuildSet -p demo/coreos $(CPU)gnu.cert
```

The **prjCreate** command gains an added option: **-ddf certKernel**. This option tells the **prjCreate** command to create a cert version of the core OS. This means that a different set of default components will be included, and that a different set of binaries will be used for those components.

The **prj projBuildSet** command sets the build spec to be used for building the core OS. This line was not needed in the debug example (**hello-full**) because the appropriate debug build spec would be selected automatically based on the CPU used. To build a cert version of the core OS, however, you must specify the use of the appropriate cert build spec for the CPU.

### Building the cert Module OS

To build the cert module OS, use the following procedure:

1. Open the VxWorks 653 Development Shell.
2. Change your current directory to **introduction/hello-cert/mos**.
3. Run **make**, specifying the values of the variables **\$(CPU)** and **\$(BSP)**. For example:

```
make BSP=wrSbc750gx CPU=PPC604
```

The makefile will run the create a core OS project and then run the makefile that project contains, creating a system module file and supporting files for the core OS.

## Partition OS cert Differences

The cert version of the partition OS makefile contains two changes. First, it redefines the **\$(CERT)** variable, which is used in **Makefile.vars** to set the appropriate compiler flags for a cert build:

```
CERT=1
```

Second, it changes the **vpath** statement used to locate the object files for partition OS binaries:

```
vpath %.o $(WIND_BASE)/target/vThreads/lib/obj$(CPU)gnucert
```

This change will cause the build to include the cert version of the partition OS binaries that are listed as dependencies of the partition OS system module file. In the Hello World example, all the components used have both debug and cert versions. In your own project, if you have debug components for which no cert versions exist, they will have to be removed from the dependency list in order to build a cert version.

### Building the cert Partition OS

To build the cert partition OS, use the following procedure:

1. Edit the SharedLibraryDescription document to insert the appropriate value for the **VirtualAddress** attribute.
2. Open the VxWorks 653 Development Shell.
3. Change your current directory to **introduction/hello-cert/pos**.
4. Run **make**, specifying the values of the variables **\$(CPU)** and **\$(BSP)**. For example:

```
make BSP=wrSbc750gx CPU=PPC604
```

The **pos.sm** and **pos-stubs.o** files will be created in the project directory.

### Application cert Differences

The makefile for the cert version of the application redefines the **\$(CERT)** variable, which is used in **Makefile.vars** to set the appropriate compiler flags for a cert build:

```
CERT=1
```

### Building the cert Application

To build the cert application, use the following procedure:

1. Open the CoreOSDescription document for your core OS (**hello-cert/pos/bsp.xml**). Locate the value for the partition virtual address. It is located in the attribute:

**CoreOSDescription/KernelConfiguration/@partitionVirtualAddress**

2. Open the VxWorks 653 Development Shell.
3. Change your current directory to **introduction/hello-cert/app**.
4. Run **make**, specifying the values of the variables **\$(CPU)**, **\$(BSP)**, and **\$(PARTADDR)**. For example:

```
make BSP=wrSbc750gx CPU=PPC604 PARTADDR=0x40000000
```

The application system module file, **hello.sm**, is created.

### **Building the cert System Image**

To build the cert system image, use the following procedure:

1. Open the VxWorks 653 Development Shell.
2. Change your current directory to **introduction/hello-cert/int**.
3. Run **make**, specifying the values of the variable **CPU**. For example:

```
make CPU=PPC604
```

A network loadable system image for the Hello World module is created.

For instructions on running this system image on your target, see the *Wind River Workbench User's Guide 2.6.1, VxWorks 653 Version*.

## **22.4 Module OS**

The module OS examples are located in:

*installDir/vxworks653-2.2/target/reference/helloWorld/moduleOS*

The illustrate the common variants of a VxWorks 653 core OS build:

hello-cert

The hello-cert example illustrates how to build a certified core OS.

hello-ace

The hello-ace example illustrates how to build a core OS that includes the agent for certified environment (ACE).

hello-bincomp

The hello-bincomp example illustrates how to build a core OS the includes additional binary components (beyond those that are included by default).

hello-srccomp

The hello-srccomp example illustrates how to build a core OS that includes source components.

#### hello-ref

The hello-ref example is a reference example that is identical to the core OS build in the introduction example. You can use this example as a reference point to diff the files in the other examples to see what changes have been made to support the new features they include.

#### hello-full

The hello-full example contains all the files for a complete module build. It is configured to export all the components of a module so that they can be used with all of the different Module OS examples to build and run a complete Hello World module.

### 22.4.1 Building and Exporting a Basic Module

The first step in exploring the module OS examples is to build a basic Hello World module and export its components so that they can be used to build complete modules on top of the different varieties of module OS.

To build and export the basic module:

1. Build the project following the directions for building a complete module in [22.2 Quick Start](#), p.158, substituting references to the directory **helloWorld/introduction/hello-full** with **helloWorld/ModuleOS/hello-full**.
2. Export the module components by running:

```
make export
```

### 22.4.2 Module OS Cert Build

The cert module OS build is the same as that provided in the introduction (**helloWorld/introduction/hello-cert/mos**). It is provided here to allow you to explore the differences between a cert and debug build by comparing it to the module OS build in the reference project (**helloWorld/moduleOS/hello-ref/mos**).

To build the cert module OS, follow the directions under [22.3.1 Hello World cert](#), p.176, substituting path names as appropriate.

### 22.4.3 Module OS with ACE

The hello-ace example shows how to build a module OS with ACE. Actually, the ACE part of the project is separate from the core OS, as this is the way ACE works.

It provides debug support in a system with a certified core OS by providing a debug agent that exists outside of the operating system.



**NOTE:** ACE is not supported on the simulator. Therefore, you cannot run this part of the reference process on the simulator. You must run it on a real target.

The example therefore consists of a makefile that combines two separate projects, a cert core OS project and an ACE project. You can see the differences between a regular cert core OS project and this project by comparing the files in this project with the files in the **moduleOS\hello-cert** example.

### ACE Differences

The **ace** directory of the ACE example contains the following files:

#### Project Makefile

The project makefile (**Makefile**) contains lines to create and build an ACE project. The lines that create the ACE project are as follows:

```
prjCreate -domtype ace -prj demo/ace -build $(CPU)gnu.cert -name ace
prj projBuildTagValueSet -p demo/ace COREOS_DIR \
.../.../mos/demo/$(CPU)gnu.cert
prj projBuildTagValueSet -p demo/ace COREOS_NAME coreOS.sm
```

The **prjCreate** command creates the project. In this case, the project type, specified by the **-domtype** option, is “ace”. The **-build** option sets the build spec to cert. The **projBuildTagValueSet** commands are then use to specify the name and location of the core OS build files for the core OS that ACE is to work with.

The lines that build the ACE project are as follows:

```
prj domComponentAdd -p demo/ INCLUDE_WDB_COMM_SERIAL
prj domParameterValueSet -p demo/ WDB_TTY_CHANNEL 0
xmlgen --ldScript --arch $(TOOLARCH) -o demo/ace.lds hello-ace.xml \
/mos/demo/bsp.xml
make -C demo ADD_NEEDED
make -C demo
```

**prj** commands are used to include the serial interface component (**INCLUDE\_WDB\_COMM\_SERIAL**) and set the TTY channel for serial communications to 0.

The **XMLGen** command creates the linker script for ACE. To do this, it needs to access the CoreOSDescription document for the core OS that ACE will work with. In this case, that document is **bsp.xml**.

## ACE Configuration Document

The ACE example contains an Ace configuration document that contains the memory black box for ACE:

```
<Ace
  xmlns="http://www.windriver.com/vxWorks653/ConfigRecord"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.windriver.com/vxWorks653/ConfigRecord
CoreOS.xsd"
  name="ace">
  <MemorySize
    MemorySizeBss="0x10000"
    MemorySizeText="0x30000"
    MemorySizeData="0x1000"
    MemorySizeRoData="0x1000" />
</Ace>
```

## Running the ACE Build

You can perform each of the build steps for ACE separately, or you can build the entire ACE module at once.

To build the ACE project in one step:

1. Make sure that you have built and exported the moduleOS/hello-full project according to the directions at [22.4.1 Building and Exporting a Basic Module](#), p.180.
2. Open the VxWorks 653 Development Shell.
3. Change your current directory to **moduleOS/hello-ace**.
4. Run **make**, specifying the values of the \$(BSP) and \$(CPU) variables. For example:

```
make BSP=wrSbc750gx CPU=PPC604
```

A network loadable system image for the Hello World module is created in the directory **moduleOS/hello-ace/int/demo**.

For instructions on running this system image on your target, see the *Wind River Workbench User's Guide 2.6.1, VxWorks 653 Version*.

To build the ACE project as separate steps:

1. Make sure that you have built and exported the **moduleOS/hello-full** project according to the directions at [22.4.1 Building and Exporting a Basic Module](#), p.180.
2. Open the VxWorks 653 Development Shell.



3. Change your current directory to **moduleOS/hello-ace/mos**.
4. Run **make**, specifying the values of the **\$(BSP)** and **\$(CPU)** variables. For example:

```
make BSP=wrSbc750gx CPU=PPC604
```

5. Change your current directory to **moduleOS/hello-ace/ace**.
6. Run **make**, specifying the value of the **\$(BSP)** variable. For example:

```
make BSP=wrSbc750gx
```

7. Change your current directory to **moduleOS/hello-ace/int**.
8. Run **make**, specifying the value of the **\$(BSP)** variables. For example:

```
make BSP=wrSbc750gx
```

A network loadable system image for the Hello World module is created in the directory **moduleOS/hello-ace/int/demo**.

For instructions on running this system image on your target, see the *Wind River Workbench User's Guide 2.6.1, VxWorks 653 Version*.

## 22.4.4 Module OS with Binary Components

A default core OS includes a number of binary components. In addition to the default set, there are a number of components that you can include to add additional functionality to your core OS. The **moduleOS/hello-bincomp** example illustrates how to add binary components to the core OS. For a complete listing of available components, see the *VxWorks 653 Configuration and Build Reference*.

To see the changes that the use of binary components introduces into a core OS build, you can compare the files in this example to the files in the files in the **moduleOS/hello-ref** example.

### Module Binary Component Differences

The changes are as follows:

#### Makefile Differences

The example adds the following lines to the makefile:

```
prj domComponentBundleAdd -prj demo/
$(WIND_BASE)/target/config/comps/vxWorks/sysTemplates/vxKernel/targetTools.dd
f
prj domComponentRemove      -prj demo/ INCLUDE_SHELL_VI_MODE
prj domComponentAdd         -prj demo/ INCLUDE_SHELL_EMACS_MODE
```

```
prj domParameterValueSet -prj demo/ SHELL_STACK_SIZE 0x8000
```

These lines perform the following operations:

```
prj domComponentBundleAdd -prj demo/  
$(WIND_BASE)/target/config/comps/vxWorks/sysTemplates/vxKernel/targetTools.dd  
f
```

This line adds a bundle of components to the core OS. The contents of the bundle are described in the file **targetTools.ddf**. They include target-side debugging tools. As with other modifications to a core OS project, this command specifies the location of the core OS project to which the components will be added using the **-prj** option. For information on the content of the **targetTools** bundle, see the *VxWorks 653 Configuration and Build Reference*.

```
prj domComponentRemove -prj demo/ INCLUDE_SHELL_VI_MODE  
prj domComponentAdd -prj demo/ INCLUDE_SHELL_EMACS_MODE
```

These lines remove the component that provides for vi-mode editing in the target shell and replace it with a component that provides emacs-mode editing.

```
prj domParameterValueSet -prj demo/ SHELL_STACK_SIZE 0x8000
```

This line changes the value of the parameter **SHELL\_STACK\_SIZE**, which controls the size of the stack for the shell component.

### Running the Binary Component Build

As with the ACE build ([22.4.3 Module OS with ACE](#), p.180), you can build the entire module at once by running **make** in the **hello-bincomp** directory, or you can perform each of the build steps for binary component build separately by running **make** in the **mos** and **int** directories. In each case you must specify the appropriate value for the BSP variable on the **make** command line.

## 22.4.5 Module OS with Source Components

The **hello-srccomp** example shows how to add your own source components to the core OS. In this case, the code that prints the hello message is included in the core OS component. The following is the code for that component:

```
#include <stdio.h>  
  
void kernelHello (void)  
{  
    printf ("Hello from the core OS!\n");  
}
```

To build this code as a core OS component, the following lines are added to the core OS project makefile:

```
prjCreate -type kernelComponent -prj hello/ -build $(CPU)gnu.debug
cp khello.c hello
prj compFileAdd -prj hello/ hello/khello.c
prj compAttributeSet -prj hello/ INIT_RTN "kernelHello();"
prj domComponentAdd -prj demo/ hello/ INCLUDE_HELLO
```

The **prjCreate** command creates a project for the new component. In this case, the **-type** option is set to “kernelComponent”. The **cp** command copies the source file to the project directory and the **prj compFileAdd** is used to add the source file to the component project. Then the **prj compAttributeSet** command is used to set the **INIT\_RTN** parameter for the component to the name of the main routine in the source code, “kernelHello();”. Finally, the **prj domComponentAdd** command is used to add the new component to the core OS project. No additional commands are required to compile the **.c** file. The makefile created by the **prj** commands will take care of all the details.

In this example, the hello message is not printed by the application but by the core OS itself. The code will be run when the core OS initializes the component. If you integrate this core OS with the Hello World application and run it, you will see one hello message from the core OS and a second from the application.

## 22.5 Partition OS

The partition OS examples are located in:

*installDir/vxworks653-2.2/target/reference/helloWorld/partitionOS*

They illustrate the common variants of a vThreads partition OS build:

hello-cert

The hello-cert example illustrates how to build a certified partition OS.

hello-bincomp

The hello-bincomp example illustrates how to build a partition OS that includes additional binary components (beyond those that are included by default).

hello-srccomp

The hello-srccomp example illustrates how to build a partition OS that includes source components.

#### hello-sd

The hello-sd example illustrates how to build a partition OS that includes a shared data region containing the text of a hello world message.

#### hello-ref

The hello-ref example is a reference example that is identical to the partition OS build in the introduction example. You can use this example as a reference point to diff the files in the other examples to see what changes have been made to support the new features they include.

#### hello-full

The hello full example contains all the files for a complete module build. It is configured to export all the components of a module so that they can be used with all of the different Module OS examples to build and run a complete Hello World module.

### 22.5.1 Building and Exporting a Basic Module

The first step in exploring the Partition OS examples is to build a basic Hello World module and export its components so that they can be used to build complete modules on top of the different varieties of partition OS.

To build and export the basic module:

1. Build the project following the directions for building a complete module in [22.2 Quick Start](#), p.158, substituting references to the directory **helloWorld/introduction/hello-full** with **helloWorld/partitionOS/hello-full**.
2. Export the module components by running:

```
make export
```

### 22.5.2 Partition OS with Binary Components

The binary component example (**hello-bincomp**) adds the POSIX component to the partition OS. The POSIX component adds support for the POSIX API. For information on POSIX, see the *VxWorks 653 Programmer's Guide*. For information on the binary components supplied with VxWorks 653, see the *VxWorks 653 Configuration and Build Reference*.

Adding this component to the partition OS means that the example has to change the “pos.sm” rule in the partition OS makefile to add the component object files to the dependencies list:

```
pos.sm: sslMain.o vThreadsComponent.o vThreadsPosixComponent.o \
vThreadsPosixInit.o \
pos-ept.o pos.lds
```

The example also adds the following lines, which cause the value of the **AIO\_TASK\_STACK\_SIZE** parameter to be passed to the compiler when the **vThreadsPosixInit** configlet is compiled:

```
CFLAGS_EXTRA = $(CFLAGS_$$)
CFLAGS_vThreadsPosixInit.o = -DAIO_TASK_STACK_SIZE=0x10000
```

Another important thing to notice about this example is what does not change. The **SharedLibraryDescription** file (**hello-pos.xml**) contains the memory black box information for the partition OS:

```
<MemorySize
  MemorySizeBss="0x10000"
  MemorySizeText="0x40000"
  MemorySizeData="0x10000"
  MemorySizeRoData="0x10000"
/>
```

This is the same black box used in the basic Hello World module (compare this black box to the one in the **partitionOS/hello-ref** project). Why does the size of the black box not change when the POSIX component is added? The reason is that the black box was originally sized to allow for growth. This means that a configuration record based on this black box would not have to be revised or re-certified to accommodate the addition of a component to the partition OS. By allowing you to reserve memory for future growth, black boxes allow you to reduce the cost of change and certification for your module.

## Building the Partition OS with Binary Components

You can perform each of the build steps for the partition OS separately, or you can build the entire Hello World module at once.

To build the entire module in one step:

1. Make sure that you have built and exported the **partitionOS/hello-full** project according to the directions at [22.4.1 Building and Exporting a Basic Module](#), p.180.
2. Edit the **SharedLibraryDescription** document (**partitionOS/hello-bincomp/pos/hello-pos.xml**) to replace the string “\$(SSLADDR)” with the shared library virtual address. You can use the same address that you used for the **partitionOS/hello-full** project.
3. Open the VxWorks 653 Development Shell.
4. Change your current directory to **partitionOS/hello-bincomp**.

5. Run **make**, specifying the values of the variable CPU. For example:

```
make CPU=PPC604
```

A network loadable system image for the Hello World module is created in the directory **partitionOS/hello-bincomp/int/demo**.

For instructions on running this system image on your target, see the *Wind River Workbench User's Guide 2.6.1, VxWorks 653 Version*.

To build the module as separate steps:

1. Make sure that you have built and exported the **partitionOS/hello-full** project according to the directions at [22.4.1 Building and Exporting a Basic Module](#), p.180.
2. Edit the SharedLibraryDescription document (**partitionOS/hello-bincomp/pos/hello-pos.xml**) to replace the string “\$(SSLADDR)” with the shared library virtual address. You can use the same address that you used for the **partitionOS/hello-full** project.
3. Open the VxWorks 653 Development Shell.
4. Change your current directory to **partitionOS/hello-bincomp/pos**.
5. Run **make**, specifying the values of the variable CPU. For example:

```
make CPU=PPC604
```

6. Change your current directory to **partitionOS/hello-bincomp/int**.
7. Run **make**, specifying the values of the variable CPU. For example:

```
make CPU=PPC604
```

A network loadable system image for the Hello World module is created in the directory **partitionOS/hello-bincomp/int/demo**.

For instructions on running this system image on your target, see the *Wind River Workbench User's Guide 2.6.1, VxWorks 653 Version*.

### 22.5.3 Partition OS with Source Components

As with the source component example in the core OS, the source component example for the partition OS moves the hello message code into a source component for the partition OS. Here is the code:

```
#include <vxWorks.h>
#include <stdio.h>
```

```
void sslHello (void);
void *helloInit _VTH_COM_INIT = sslHello;

void sslHello (void)
{
    printf ("Hello from the partition OS!\n");
}
```

For information on writing partition OS components, see the *VxWorks 653 Programmer's Guide*.

The only change to the rest of the configuration files is the addition of the object file for the component to the **pos.sm** rule in the partition OS makefile and the addition of a generic rule to compile **.o** files from **.c** files:

```
pos.sm: sslMain.o vThreadsComponent.o poshello.o pos-ept.o pos.lds

%.o: %.c
    $(CC) $(CFLAGS) -c -o $$@ $<
```

### Running the Source Component Build

As with the binary component build ([22.5.2 Partition OS with Binary Components](#), p.186), you can build the entire module at once by running **make** in the **hello-srccomp** directory, or you can perform each of the build steps for binary component build separately by running **make** in the **pos** and **int** directories. In each case you must edit the SharedLibraryDescription document to insert the shared library virtual address and you must specify the appropriate value for the BSP variable on the **make** command line.

## 22.5.4 Partition OS with Shared Data Region

A shared data region is a region of memory set aside for data shared between applications. The shared data region may be empty, allowing the applications to use it to exchange data, or it may contain a database that contains information that the applications can share. In this example, the hello world message will be read from a shared data region. (Empty shared data regions do not need to be built. They are simply configured in the SharedDataDescriptions document.)

A shared data region is a separate project that requires its own source, configuration, and makefiles.

### Shared Data Source

The shared data source file looks like this:

```
char sd[] = "Hello inside the Shared Data region";
```

This line simply defines a character string. VxWorks 653 does not define a particular access mechanism for shared data regions, but leaves their structure and access methods up to the individual developer.

### Partition OS Source Component

The application from the shared data region example has been changed to access the text of the hello message from the shared data region.

```
#include <vxWorks.h>
#include <stdio.h>
#include <sdRgnLib.h>

void sslHello (void);
void *helloInit _VTH_COM_INIT = posHello;

void posHello (void)
{
    char *sdPtr = sdRgnAddrGet("sd");
    printf ("%s accessed within the partition OS!\n", sdPtr);
}
```

For more information on programming partition OS components and shared data regions, see the *VxWorks 653 Programmer's Guide*.

### SharedDataDescription Document

The configuration information for the shared data region is contained in the SharedDataDescription document:

```
<SharedDataDescription
  xmlns="http://www.windriver.com/vxWorks653/ConfigRecord"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.windriver.com/vxWorks653/ConfigRecord
Application.xsd"
  Size="0x00001000"
  DataType="DATABASE"
  VirtualAddress="0"
  CachePolicy="DEFAULT"
  SystemAccess="READ_WRITE"
/>
```

For information on the individual fields in the SharedDataDescription document, see the *VxWorks 653 Configuration and Build Reference*.

### Shared Data Makefile

The shared data makefile looks like this:

```
all: sd.sm

include $(WIND_BASE)/target/vThreads/config/make/Makefile.vars
```



```
sd.sm: sd.o sd.lds
      $(LD) $(LDFLAGS) -T sd.lds -o $@ $(filter %.o,$^)

%.o: %.c
      $(CC) $(CFLAGS) -c -o $@ $<

sd.lds: hello-sd.xml
      xmlgen --ldScript --arch ppc --api-version sd -o $@ $<
```

The structure of the makefile is the same as for a shared library. The shared data region is compiled by the default rules, a linker script is generated from the black box information in the SharedDataDescription document, and the shared data region is built into a system module file (**sd.sm**).

The other files are the same as in the source component example. When you build this example, the hello message from the shared data region will be shown when the partition is initialized.

### Running the Shared Data Build

As with the binary component build ([22.5.2 Partition OS with Binary Components](#), p.186), you can build the entire module at once by running **make** in the **hello-sd** directory, or you can perform each of the build steps for binary component build separately by running **make** in the **sd**, **pos** and **int** directories. In each case you must edit the SharedLibraryDescription document to insert the shared library virtual address and you must specify the appropriate value for the BSP variable on the **make** command line.

## 22.6 Application

The application examples are located in:

*installDir/vxworks653-2.2/target/reference/helloWorld/application*

They illustrate the common variants of an application build:

hello-cert

The hello-cert example illustrates how to build an application for a certified environment. Note that building an application for a cert environment does not in itself imply that the application is certified. The application must be certified separately according to the applicable certification procedures.

#### hello-cpp

The hello-cpp example illustrates how to build an application written in C++.

#### hello-two

The hello-two example illustrates the case in which a module includes more than one application.

#### hello-ref

The hello-ref example is a reference example that is identical to the application build in the introduction example. You can use this example as a reference point to diff the files in the other examples to see what changes have been made to support the new features they include.

#### hello-full

The hello full example contains all the files for a complete module build. It is configured to export all the components of a module so that they can be used with all of the different Application examples to build and run a complete Hello World module.

### 22.6.1 Building and Exporting a Basic Module

The first step in exploring the Application examples is to build a basic Hello World module and export its components so that they can be used to build complete modules using the different varieties of application.

To build and export the basic module:

1. Build the project following the directions for building a complete module in [22.2 Quick Start](#), p.158, substituting references to the directory **helloWorld/introduction/hello-full** with **helloWorld/Application/hello-full**.
2. Export the module components by running:

```
make export
```

### 22.6.2 Application in C++

The C++ application (**hello.cpp**) adds a time statement to the Hello World program. For information on programming in C++ for VxWorks 653, see the *VxWorks 653 Programmers Guide*.

Adding the C++ application involves a number of changes to the application makefile. Here are the changed lines:

```
hello.sm: vxMain.o hello.pm hello-ctors.o hello.lds
```

```
$(LD) $(LDFLAGS) -T hello.lds -o $@ $(filter %.o %.pm,$^)  
  
hello.pm: hello.o pos-stubs.o  
$(LD) $(LDFLAGS_PARTIAL) -o $@ $^  
  
%-ctors.c: %.pm  
$(NM) $< | $(MUNCH) $(MUNCHFLAGS) > $@
```

These lines break the build into two separate stages. C++ applications with global objects require the OS to call the object constructors before starting the application. This is accomplished with a three-step linking process:

1. The C++ application is linked into a **.pm** file (pm stands for partially-linked module)
2. The **\$(MUNCH)** tool generates the list of global constructors from the **.pm** file
3. The list of constructors and the **.pm** file are linked into the system module (**.sm**) file.

```
%.o: %.cpp  
$(CC) $(C++FLAGS) -c -o $@ $<
```

This rule provides a generic rule for building C++ files.

```
%-ctors.c: %.pm  
$(NM) $< | $(MUNCH) $(MUNCHFLAGS) > $@
```

This rule provides a generic rule for building global constructors files from **.pm** files.

## Building the C++ Application

You can build the entire Hello World module at once, or you can perform each of the build steps for the C++ application separately.

To build the entire module in one step:

1. Make sure that you have built and exported the **application/hello-full** project according to the directions at [22.6.1 Building and Exporting a Basic Module](#), p.192.
2. Open the VxWorks 653 Development Shell.
3. Change your current directory to **application/hello-cpp**.
4. Run **make**, specifying values for the BSP. For example:

```
make CPU=PPC604 TOOL=gnu BSP=wrsbc750gx PARTADDR=0x40000000
```

To determine **PARTADDR** for other BSPs, see [22.2 Quick Start](#), p.158.

A network loadable system image for the Hello World module is created in the directory **application/hello-cpp/int/demo**.

For instructions on running this system image on your target, see the *Wind River Workbench User's Guide 2.6.1, VxWorks 653 Version*.

To build the module as separate steps:

1. Make sure that you have built and exported the **application/hello-full** project according to the directions at [22.6.1 Building and Exporting a Basic Module](#), p.192.

2. Open the VxWorks 653 Development Shell.

3. Change your current directory to **application/hello-cpp/app**.

4. Run **make**, specifying values for the BSP. For example:

```
make CPU=PPC604 TOOL=gnu BSP=wrSbc750gx PARTADDR=0x40000000
```

To determine **PARTADDR** for other BSPs, see [22.2 Quick Start](#), p.158.

5. Change your current directory to **application/hello-cpp/int**.

6. Run **make**, specifying values for the BSP. For example:

```
make CPU=PPC604 TOOL=gnu BSP=wrSbc750gx PARTADDR=0x40000000
```

To determine **PARTADDR** for other BSPs, see [22.2 Quick Start](#), p.158.

A network loadable system image for the Hello World module is created in the directory **application/hello-cpp/int/demo**.

For instructions on running this system image on your target, see the *Wind River Workbench User's Guide 2.6.1, VxWorks 653 Version*.

### 22.6.3 Two Applications

The **hello-two** example illustrates how to create a module with two applications.

There are several new or changed files in this example:

#### Second application file

The second application is a simple Hello World program:

```
#include <stdio.h>

void usrAppInit (void)
{
    printf ("Hello from the second application!\n");
}
```

```
}
```

### Second ApplicationDescription Document

The ApplicationDescription document for the second application (**hello2.c**) is new, but it is identical to the ApplicationDescription document for the first application (**hello.c**). (Not surprisingly, the black box size is the same for both applications.)

### Second Application Makefile

Other than the names of the files, the makefile for the second application (**makefile.app**) is identical to the makefile for the first application (**makefile.app2**).

### Project Makefile

The project makefile (**Makefile**) has changed to create the project for the second application:

```
all:

# Create the application project
  mkdir -p demo
  cp Makefile.app2 demo/Makefile
  cp hello-secondapp.xml demo
  cp hello2.c demo

# Build the application
  make -C demo
```

### PartitionDescription Document

The PartitionDescription document (**hello-secondpart.xml**) for the second application's partition is identical to the PartitionDescription of the first application (**hello-part.xml**), except that it names a different application and a different health monitor table.

### Module Configuration Document

There are several changes to the Module configuration document (**hello-module.xml**):

```
<Application Name="hello2">
  <xi:include href="hello-secondapp.xml"/>
</Application>
```

These lines include the ApplicationDescription document for the second application.

```
<Partition Name="hello2" Id="2">
  <xi:include href="hello-secondpart.xml"/>
```

```
</Partition>
```

These lines include the PartitionDescription document for the second application's partition.

```
<Schedule Id="0">  
  <PartitionWindow PartitionNameRef="hello1" Duration="0.10"/>  
  <PartitionWindow PartitionNameRef="hello2" Duration="0.10"/>  
</Schedule>
```

These lines add the second application to the default schedule.

```
<Payloads>  
  <CoreOSPayload/>  
  <SharedLibraryPayload NameRef="pos"/>  
  <ConfigRecordPayload NameRef="configRecord"/>  
  <PartitionPayload NameRef="hello1"/>  
  <PartitionPayload NameRef="hello2"/>  
</Payloads>
```

These lines define a payload for the second application.

### Integration Makefile

The only change in the integration makefile (**makefile.int**) is the line to copy the system module file for the second application:

```
hello2.sm: $(APP2_DIR)/hello2.sm    ; cp $< $@
```

### Running the Two Applications Build

As with the C++ application build ([22.6.2 Application in C++](#), p.192), you can build the entire module at once by running **make** in the **hello-two** directory, or you can perform each of the build steps for binary component build separately by running **make** in the **app1**, **app2** and **int** directories. In each case you must specify the appropriate value for the BSP variable on the **make** command line.

## 22.7 Shared Library

The shared library examples are located in:

*installDir/vxworks653-2.2/target/reference/helloWorld/sharedLibrary*

They illustrate the common variants of a shared library build. Earlier versions of the Hello World module have not used a shared library, so all the shared library variants are new. The shared library example includes the following examples:

#### hello-ref

In the shared library example, the ref example does not include a shared library. Instead it reproduces the hello-two example from the application example as a starting point for creating a solution that uses a shared library to support the two applications. You can use it as a reference point to compare with the shared library examples.

#### hello-sl

The hello-sl example shows how common functionality can be placed in a shared library and accessed by more than one application.

#### hello-version

The hello-version example shows how to create a shared library with more than one version of its interface.

#### hello-full

The hello full example contains all the files for a complete module build. It is configured to export all the components of a module so that they can be used with all of the different shared library examples to build and run a complete Hello World module.

### 22.7.1 Building and Exporting a Basic Module

The first step in exploring the shared library examples is to build a basic Hello World module and export its components so that they can be used to build complete modules using the different varieties of shared library.

To build and export the basic module:

1. Build the project following the directions for building a complete module in [22.2 Quick Start](#), p.158, substituting references to the directory **helloWorld/introduction/hello-full** with **helloWorld/sharedLibrary/hello-full**.
2. Export the module components by running:

```
make export
```

### 22.7.2 Hello from the Shared Library

The shared library example (**hello-sl**) places the routine that displays the hello world message into a shared library and accesses it from two different applications. It includes a number of new or changed files:

## Shared Library Source File

The shared library source file (**hellolib.c**) looks like this:

```
#include <stdio.h>

void hello (void)
{
    printf ("Hello, world!\n");
}

void usrAppInit (void)
{
}
```

What is important to note about this program is that it is not designed to run when the library is initialized (as was the case in the partition OS example described in [22.5.3 Partition OS with Source Components](#), p.188) but when it is called from an application. Thus the initialization routine, **usrAppInit()** is an empty routine and the Hello World functionality is contained in a separate routine, **hello()**.

## SharedLibraryDescription Document

The SharedLibraryDescription document for the shared library looks 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
Application.xsd"
  SystemSharedLibrary="false"
  VirtualAddress="$ (SLADDR) ">
  <MemorySize
    MemorySizeBss="0x10000"
    MemorySizeText="0x10000"
    MemorySizeData="0x10000"
    MemorySizeRoData="0x10000"
  />
</SharedLibraryDescription>
```

Just as you did with the system shared library that holds the partition OS in the introduction example, you must calculate the correct value for the **VirtualAddress** parameter and update the SharedLibraryDescription file accordingly. For a production system, you will need to calculate the virtual memory map of the module to choose appropriate addresses for all shared libraries. For building this example, however, you can safely choose an address that is one megabyte greater than the shared library virtual address of the partition OS.

For information on the meaning of each of the settings on the SharedLibraryDescription document, see the *VxWorks 653 Configuration and Build Guide*.



## Shared\_Library\_API Document

The Shared\_Library\_API document describes the interface of the shared library.

```
<Shared_Library_API
  xmlns="http://www.windriver.com/vxWorks653/SharedLibraryAPI"
  xmlns:xi="http://www.w3.org/2001/XInclude"
  Name="Hello Library"
>
  <Interface>
    <Version Name="version1"/>
    <Interface_Subset>
      <Routine Name="hello"/>
    </Interface_Subset>
  </Interface>
</Shared_Library_API>
```

For more information on the meaning of the fields in the Shared\_Library\_API document, see the *VxWorks 653 Configuration and Build Reference*. For more information on shared library interfaces, see [9Shared Libraries](#), p.57.

## Shared Library Makefile

The shared library makefile is used to build the shared library. Structurally it is identical to the partition OS makefile.

```
all: sl.sm sl-stubs.o

include $(WIND_BASE)/target/vThreads/config/make/Makefile.vars

vpath %.c    $(WIND_BASE)/target/vThreads/config/comps/src
vpath %.o    ../../../../exports

sl.sm: vxMain.o hellolib.o pos-stubs.o sl-ept.o sl.lds
      $(LD) $(LDFLAGS) -T sl.lds -o $@ $(filter %.o,$^)

%.o: %.c
      $(CC) $(CFLAGS) -c -o $@ $<

sl-ept.c: sl-api.xml
      xmlgen --linkage --output-entrypoints $@ $<

sl-stubs.c: sl-api.xml
      xmlgen --linkage --arch ppc --output-stubs $@ $<

sl.lds: hello-sl.xml
      xmlgen --ldScript --arch ppc --api-version sl -o $@ $<
```

## Application Source Files

The source code for the first application (**hello.c**) changes to call the library routine:

```
#include <stdio.h>
```

```
void usrAppInit (void)
{
    printf ("Starting the first application\n");
    hello ();
}
```

Similarly for the second application (**hello2.c**):

```
#include <stdio.h>

void usrAppInit (void)
{
    printf ("Starting the second application\n");
    hello ();
}
```

For information on writing applications that access shared library code, see the *VxWorks 653 Programmer's Guide*.

### PartitionDescription Files

The partition description files for both applications change to include a reference to the shared library. The following example shows the partition description for the first application:

```
<PartitionDescription
  xmlns="http://www.windriver.com/vxWorks653/ConfigRecord"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.windriver.com/vxWorks653/ConfigRecord
Partition.xsd">
  <SharedLibraryRegion NameRef="pos"/>
  <SharedLibraryRegion NameRef="sl"/>
  <Application NameRef="hello"/>
  <Settings
    RequiredMemorySize="0x100000"
    PartitionHMTTable="helloHm"
    numFiles="0xffffffff"
    numDrivers="0xffffffff"
    isrStackSize="0xffffffff"
    maxEventQStallDuration="INFINITE_TIME"
  />
</PartitionDescription>
```

An application can only access those shared libraries that are referenced by its partition. Since a partition must provide stack and heap space for the execution of each shared library that it references, a partition should only reference those libraries that are used by its application.

### Application Makefiles

The application makefiles change to include the shared library stubs in the application build and to add a vpath statement to locate the stubs files:

```
vpath %.o    ../../sl/demo

hello1.sm: vxMain.o hello.o pos-stubs.o sl-stubs.o part1.lds
```

### Building the Hello World Shared Library Example

You can build the entire module at once by running **make** in the **hello-sl** directory, or you can perform each of the build steps for binary component build separately by running **make** in the **sl**, **app1**, **app2** and **int** directories. In each case, you must specify the appropriate value for the BSP variable on the **make** command line.

## 22.7.3 Shared Library Versioning

VxWorks 653 allows you to define more than one version of the interface of a library. This can be used for backwards compatibility or to limit access to library routines for applications using different interfaces.

The reference point for the shared library version example (**hello-version**) is the shared library example (**hello-sl**). The following files have changes from the **hello-sl** example:

### Shared Library Source File

The shared library source file (**hellolib.c**) adds a second Hello World routine:

```
void hello2 (void)
{
    printf ("Hello too!\n");
}
```

### Shared\_Library\_API Document

The Shared\_Library\_API document defines two separate interfaces:

```
<Interface>
  <Version Name="version1"/>
  <Interface_Subset>
    <Routine Name="hello"/>
  </Interface_Subset>
</Interface>

<Interface>
  <Version Name="version2"/>
  <Interface_Subset>
    <Routine Name="hello" InternalName="hello2"/>
  </Interface_Subset>
</Interface>
```

The first interface, named “version1” is the same as the interface in the **hello-sl** example.

The second interface, named “version2”, defines an interface that looks the same to the application but accesses a different routine in the library. That is, it associates the routine name “hello” with the internal routine “hello2”. This means that whether an application uses version1 or version2 of the interface, they will use the same routine name, “hello”, but depending on which version they use, a different routine will be called in the library.

The value of the **Name** attribute cannot contain spaces.

### Shared Library Makefile

Since we now have two different interfaces to the shared library, we need to generate two different version of the shared library stubs files, one expressing each of the interfaces. To do this, the shared library makefile is changed to generate the two different stubs files.

The first change is to add both stubs files to the “all” rule:

```
all: sl.sm slv1-stubs.o slv2-stubs.o
```

The second change is to define rules to build each of the stubs files:

```
slv1-stubs.c: sl-api.xml
    xmlgen --linkage --arch ppc --api-version "version1" --output-stubs $@ $<
```

```
slv2-stubs.c: sl-api.xml
    xmlgen --linkage --arch ppc --api-version "version2" --output-stubs $@ $<
```

Notice that the `--api-version` option is used to select the interface version to build in each case. The name given here must match one of the interface names given in the `Shared_Library_API` document.

### Application Makefiles

The application makefiles also change to add the version specific stubs files to the application dependencies. Thus the build rule for the first application becomes:

```
hello1.sm: vxMain.o hello.o pos-stubs.o slv1-stubs.o hello1.lds
```

Since the name of the Hello World routine remains the same in both interfaces, the application source files do not change. However, each will print a different message, depending on which version of the library they link to.

### Running the Shared Library Version Build

As with the Hello World shared library build ([22.7.2 Hello from the Shared Library](#), p.197), you can build the entire module at once by running **make** in the **hello-version** directory, or you can perform each of the build steps for binary component build separately by running **make** in the **sl**, **app1**, **app2** and **int** directories. In each case you must specify the appropriate value for the BSP variable on the **make** command line.

## 22.8 Integration

The integration examples are located in:

*installDir/vxworks653-2.2/target/reference/helloWorld/integration*

They illustrate the build of the three system image types supported by VxWorks 653. The examples are:

hello-net

The hello-net example illustrates how to build a network loadable system image.

hello-ram

The hello-ram example illustrates how to build a RAM payload system image.

hello-rom

The hello-rom example illustrates how to build a ROM payload system image.

hello-full

The hello full example contains all the files for a complete module build. It is configured to export all the components of a module so that they can be used with all of the different integration examples to build and run a complete Hello World module.

For more information on system images, see [21. System Images](#).

## 22.8.1 Building and Exporting a Basic Module

The first step in exploring the integration examples is to build a basic Hello World module and export its components so that they can be used to build complete modules using the different varieties of integration.

To build and export the basic module:

1. Build the project following the directions for building a complete module in [22.2 Quick Start](#), p.158, substituting references to the directory **helloWorld/introduction/hello-full** with **helloWorld/integration/hello-full**.
2. Export the module components by running:

```
make export
```

## 22.8.2 Network-Loadable System Image

The network-loadable system image is the one that has been built for all the other examples in the reference process. It is identical to the **introduction/int** example. You can use it as a reference source to compare with the other integration examples to see where the changes are in building each type of system image.

### Building the Network-Loadable System Image

To build the network-loadable system image:

1. Make sure that you have built and exported the **integration/hello-full** project according to the directions at [22.8.1 Building and Exporting a Basic Module](#), p.204.
2. Open the VxWorks 653 Development Shell.
3. Change your current directory to **integration/hello-net**.
4. Run **make**, specifying the values of the variable **\$(BSP)**. For example:

```
make BSP=wrSbc750gx
```

A network-loadable system image for the Hello World module is created in the directory **integration/hello-net/int/demo**.

For instructions on running this system image on your target, see the *Wind River Workbench User's Guide 2.6.1, VxWorks 653 Version*.

### 22.8.3 RAM Payload System Image

The RAM payload system image example is identical to the network loadable image example, except for the integration makefile (**Makefile.int**):

```
all: ram checkSize

include $(WIND_BASE)/target/config/make/Makefile.vars

XML_FILE = hello.xml
SM_FILES = $(shell $(XMLGEN_FILES) $(XML_FILE))
BIN_FILES = $(shell $(XMLGEN_FILES) --bin $(XML_FILE))
SYM_FILES = $(shell $(XMLGEN_FILES) --sym $(XML_FILE))

MOS_DIR = ../../../../exports
POS_DIR = ../../../../exports
APP_DIR = ../../../../exports

include $(WIND_BASE)/target/config/make/Makefile.rules

payloadObjs_ram.o: ../../../../exports/payloadObjs_ram.o ; cp $< $@
coreOS.sm: $(MOS_DIR)/coreOS.sm ; cp $< $@
pos.sm: $(POS_DIR)/pos.sm ; cp $< $@
hello.sm: $(APP_DIR)/hello.sm ; cp $< $@

# use XML black boxes to pad the .bin files

BINFLAGS_EXTRA = $(shell $(XMLGEN_BINFLAGS) -j $* configRecord.xml)
$(BIN_FILES): configRecord.xml

checkSize: $(SM_FILES)
    xmlgen --bbCheck $(addprefix -j ,$(basename $^)) configRecord.xml
```

The changes in this file are as follows:

```
all: ram checkSize
```

The all target specifies the “ram” target rather than the “net” target. Like the “net” target, the “ram” target is specified in **Makefile.rules**.

```
BIN_FILES = $(shell $(XMLGEN_FILES) --bin $(XML_FILE))
```

This line creates a list of binary files that will be created by the “net” target.

```
payloadObjs_ram.o: ../../../../exports/payloadObjs_ram.o ; cp $< $@
```

This rule copies the payload loader for the RAM payload system image from the exports directory. The payload loader is created as part of the core OS build. The loader is part of the payload that is downloaded to the target. On the target, it is used to load the payload into normal RAM.

```
# use XML black boxes to pad the .bin files
```

```
BINFLAGS_EXTRA = $(shell $(XMLGEN_BINFLAGS) -j $* configRecord.xml)
$(BIN_FILES): configRecord.xml
```

These lines pad out the binary files that constitute the RAM payload so that they align on the boundaries specified in their respective black boxes.

### Building the RAM Payload System Image

To build the RAM payload system image:

1. Make sure that you have built and exported the **integration/hello-full** project according to the directions at [22.8.1 Building and Exporting a Basic Module](#), p.204.
2. Open the VxWorks 653 Development Shell.
3. Change your current directory to **integration/hello-ram**.
4. In the **exports** directory, edit **bsp.xml** to specify appropriate values for the **Size** and **Base\_Address** attributes of the **ramPayloadRegion** element (example values are shown):

```
ramPayloadRegion Size="0x00600000" Base_Address="0xfa00000"
```

5. Run **make**, specifying the values of the variable **\$(BSP)**. For example:

```
make BSP=wrSbc750gx
```

A network loadable system image for the Hello World module is created in the directory **integration/hello-ram/int/demo**.

For instructions on running this system image on your target, see the *Wind River Workbench User's Guide 2.6.1, VxWorks 653 Version*.

## 22.8.4 ROM Payload System Image

The ROM payload system image example is identical to the RAM payload system image example except for the following changes in the integration makefile:

The all target specifies the **rom** target rather than the **ram** target:

```
all: rom checkSize
```

The name of the payload loader changes to "payloadObjs\_rom.o":

```
payloadObjs_rom.o: ../../../../exports/payloadObjs_rom.o ; cp $< $@
```

### Building the ROM Payload System Image

To build the ROM payload system image:



1. Make sure that you have built and exported the **integration/hello-full** project according to the directions at [22.8.1 Building and Exporting a Basic Module](#), p.204.
2. Open the VxWorks 653 Development Shell.
3. Change your current directory to **integration/hello-rom**.
4. Run **make**, specifying the values of the variable **\$(BSP)**. For example:

```
make BSP=wrSbc750gx
```

A network loadable system image for the Hello World module is created in the directory **integration/hello-rom/int/demo**.



# A

## *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](http://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.