Spectra CF

Version 3.0

User Guide
All Platforms

PRISMTECH

Spectra CF

User Guide

PRISMTECH

Part Number: CF-UG

Doc Issue 07, 27 Nov 14

Copyright Notice

Contents

Table of Contents

The Example Application

SharedLibrary Support

Spectra CF on Linux

Spectra CF on LynxOS 4.0 and 4.2

Spectra CF on VxWorks 5.5.1

Spectra CF on VxWorks 6.7 & 6.8

Spectra CF on GHS Integrity 5.0.x

Spectra CF on Windows

Preface

About the User Guide

This *User Guide* is included with the ***Spectra Core Framework*** (***CF***), part of the Spectra productivity suite for Software Defined Radio (SDR) developers.

The Spectra suite comprises:

Spectra CX, which is designed to provide SDR developers with an easy-to-use, graphical modeling environment that dramatically increases the productivity of waveform and radio platform development.

A range of language-specific ***Code Generators*** which produce source, descriptor, and test code seamlessly from the graphical model.

The ***Unit Test Framework*** verifies this generated code, if necessary well in advance of the availability of a radio platform.

Spectra CF is a high-performance, low-overhead runtime environment for SDR applications. It can be used in conjunction with Spectra CX to deploy generated platforms and waveforms.

Intended Audience

This *User Guide* is intended to be used by anyone who wishes to deploy SCA platforms and applications, whether created with Spectra CX or generated by third-party software.

Organisation

The first chapter has a brief outline of the Software Defined Radio (SDR) standard, and notes about how Spectra CF complies with the specifications.

The second chapter describes the components of Spectra CF.

The third chapter explains how to install and configure Spectra CF.

Chapter four describes the included example application.

Chapter five describes how Spectra CF supports shared libraries.

Subsequent chapters explain how to use Spectra CF to run applications (including the supplied example application) on specific platforms.

For descriptions and detailed examples of how to model waveforms and create applications using other applications in the Spectra suite, please refer to the relevant User Guides.

Conventions

The conventions listed below are used to guide and assist the reader in understanding the *User Guide*.

For simplicity and convenience, many descriptions in this *User Guide* use the general terms ‘display’ and ‘print’ to indicate information that is output to a console or other command-line interface; similarly, ‘enter’ and ‘type’ are used to indicate information or commands that are supplied to the program or system. Depending on your particular environment, this ‘console’ may or may not be a traditional VDU and keyboard attached to a PC.

Icons in the margins draw attention to important information such as differences between versions, and when specific standards or conditions apply.

Item contains helpful hint or special information.

Item of special significance or where caution needs to be taken. *This icon is also used to indicate information which can be critical to reliable operation on particular combinations of hardware and software.* This information may not be included in the hardware or operating system documentation.

Ruled lines are also used to indicate notes or cautions. They may also be used with the ‘caution’ icon (above) or the ‘environment’ icons (below) to add emphasis or draw attention to critical information.

SCA 2.2.2 Information relevant to SCA 2.2.2.

WIN Information applies to Windows (*e.g.* XP, 2003, Windows 7) only.

UNIX Information applies to Unix-based systems (*e.g.* Solaris) only.

C C language specific.

C++ C++ language specific.

Java Java language specific.

Hypertext links are shown as *blue italic underlined*.

On-Line (PDF) versions of this document: Items shown as cross references, *e.g.* ‘see *Contacts* on page xi’, act as hypertext links: click on the reference to go to the item.

```
%  Commands or input which the user enters on the
   command line of their computer terminal
```

`Courier` fonts indicate programming code, file names, and short keyboard entries.

Extended code fragments are shown in shaded boxes:

```
NameComponent newName[] = new NameComponent[1];

// set id field to "example" and kind field to an empty string
newName[0] = new NameComponent ("example", "");
```

Italics and ***Italic Bold*** indicate new terms, or emphasise an item.

Sans-serif and **sans-serif Bold** indicate Graphical User Interface (GUI) or Integrated Development Environment (IDE) elements and commands; for example, **File > Save** (a sequence of selections from a set of menus).

Step 1: One of several steps required to complete a task.

Contacts

PrismTech can be reached at the following contact points for information and technical support.

USA Corporate Headquarters

PrismTech Corporation
400 TradeCenter
Suite 5900
Woburn, MA
01801
USA

Tel: +1 781 569 5819

European Head Office

PrismTech Limited
PrismTech House
5th Avenue Business Park
Gateshead
NE11 0NG
UK

Tel: +44 (0)191 497 9900
Fax: +44 (0)191 497 9901

Web: http://www.prismtech.com
Technical questions: crc@prismtech.com (Customer Response Center)
Sales enquiries: sales@prismtech.com

SDR Specifications

CHAPTER

1 SDR Specifications

This chapter briefly describes how the Spectra Core Framework complies with the established standards.

1.1 Introduction

The idea behind *Software Defined Radio* (SDR) is that several different radios (with capabilities defined and limited by hardware) can be replaced by a single versatile radio (built with general-purpose hardware) which can be reconfigured through software. A *software-defined radio* uses software to control performance parameters, such as the wavelength, modulation or encryption used, which were previously controlled by means of special-purpose hardware.

Software Communications Architecture (SCA) is an open framework which defines how software and hardware components interoperate within a software defined radio. It defines several interfaces which a compliant application must implement. The SCA *Core Framework* (CF) is a standard component management API which makes interoperability possible between radios; SDR software is easily portable between different CF-compliant hardware sets.

SCA incorporates interoperability features and standards originally defined in the Common Object Request Broker Architecture (CORBA) specification developed by the Object Management Group (OMG).

The SCA standard is a critical element in the Joint Tactical Radio System (JTRS) being developed by NATO and the U.S. military, so although SCA itself does not specify a hardware configuration, one of the requirements for SCA certification is that SDR applications (often referred to as ‘waveforms’) must operate successfully on a particular hardware platform specified by the U.S. government.

1.2 Spectra CF compliance

Spectra CF is compliant with the SCA 2.2.2 specification.

1.2.1 SCA compliance references in this Guide

Issues arising from compliance with the SDR specifications are mentioned at relevant places in this Guide.

For example, although the DomainManager and DeviceManager can be started by other methods, using Spectra CF’s BootLoader to control the boot-up sequence guarantees SCA compliance. (See Section 2.2.1, *BootLoader*, on page 19.)

All Spectra CF operations work in compliance with SCA standards.

1.3 References

The official SCA and Joint Tactical Radio System (JTRS) site run by the U.S. Department of Defense:

http://jtnc.mil/sca/

The Object Management Group (OMG) develops standards for systems interoperability, including Common Object Request Broker Architecture (CORBA). (The CORBA Naming service, Lightweight Log service and Event service are referenced in the SCA specifications.)

http://www.omg.org/

The OMG publishes many documents, including both white papers and standards specifications, wihch can be freely downloaded. The specifications catalogue is at:

http://www.omg.org/technology/documents/spec_catalog.htm

The Wireless Innovation Forum is an international association which supports the development and deployment of Software Defined Radio.

http://www.wirelessinnovation.org

Spectra CF Components

CHAPTER

2 Spectra CF Components

This chapter describes the various components of the Spectra Core Framework.

2.1 Introduction to Spectra CF

Spectra CF is part of PrismTech's Spectra SDR Tools suite. It is a second generation, high-performance, low-overhead runtime environment for Software Defined Radio (SDR) applications.

2.2 Spectra CF components

Spectra CF includes the following binaries, which are described below:

```
BootLoader
DomainManager
DeviceManager
cfadmin
TargetLoader
FileSystemService
LogService
ExecutableDevice
```

Depending on the platform on which Spectra CF is deployed, it may be necessary to re-link these binaries. For example, on Green Hills Software INTEGRITY sizing and memory attributes often need to be reconfigured. Notes about platform-specific configuration considerations are included in the appropriate chapters.

Spectra CF is distributed with debug variants (identified by the '`_g`' suffix) of the binaries that can be used to help diagnose problems. These debug versions print a large amount of diagnostic output to the console when running; they also have larger files and slower operation than the non-debug versions.

2.2.1 BootLoader

The BootLoader is a launch utility that can be used to start the DomainManager, DeviceManager and SCA services such as the LogService and FileSystemService. The advantage of launching the SCA binaries using BootLoader instead of starting the binaries directly is that the execparams listed in the PRF descriptor file for that binary will automatically be passed as arguments to main at startup. In cases where multiple implementations are listed in the Software Package Descriptor (SPD) file for a binary, BootLoader will determine which should be launched by analysing the platform properties to find one that matches the target system.

The type of binary that BootLoader is to launch is determined by the descriptor file passed to it: to launch the DomainManager the DMD file is specified; for DeviceManagers the DCD file is specified; and services are started by passing the SPD file. Additional arguments may be required depending on which type of binary is being launched.

BootLoader accepts the following arguments:

-DMD_FILENAME <path to dmd file>
The BootLoader will parse the dmd xml file describing the DomainManager in order to locate its Software Package Descriptor (SPD) file. It will navigate through the implementation attributes of this SPD file and start the executable refered to by the implementation section that it deems appropriate for the target system.

-DCD_FILENAME <path to dcd file>
The BootLoader will parse the dcd xml file describing the DeviceManager in order to locate its Software Package Descriptor (SPD) file. It will navigate through the implementation attributes of this SPD file and start the executable refered to by the implementation section that it deems appropriate for the target system.

-ORBInitRef NameService=<IOR | corbaloc>
The IOR or corbaloc of the NameService context that the DomainManager is bound into. This argument is required when starting a DeviceManager.

-SPD_SERVICE_FILENAME <path to SPD file>
The BootLoader will parse the SPD file and start the executable referred to by the implementation section that it deems to be appropriate for the target system. When starting a service, the following two parameters must also be specified.

-SERVICE_NAME <name>
The name used to identify the service once it has registered with the DeviceManager.

-DEVICE_MGR_IOR <ior>
The IOR or corbaloc of the device manage that the service will register with.

2.2.2 DomainManager

The DomainManager component provides control, configuration and a system wide view of the SDR domain. DomainManager can either be spawned as a stand-alone process, or launched by a DeviceManager (see section *2.2.3.1* on page 27 for information on the latter approach). Once the DomainManager is running, all other components can register themselves with it.

The DomainManager executable that is required to start for a specific SDR platform is determined by the Domain Manager Descriptor (DMD) file and the BootLoader. The DMD file refers to a Software Package Descriptor (SPD) file which contains at least one implementation of the DomainManager and may contain many if supporting multiple platforms. The correct selection and activation of the appropriate DomainManager implementation and executable is done by the BootLoader component. The DomainManager registers itself into a Naming Service during its initialization process.

In order for a DomainManager to read its domain profile it must be accessible on a file system provided by the operating system. The file system location containing the domain profile is mounted into the DomainManager as `/dmfs` and is therefore visible to all SDR components. Unless otherwise configured by an `fsmount` option, the `/dmfs` mount point in DomainManager's FileManager will default to the current working directory of the underlying file system.

When the DomainManager is started, it will attempt to automatically restore any application factories that were installed the last time it ran. The DomainManager keeps track of which applications have been installed by writing to the file `/dmfs/<dm_id>/installed-apps`, where `<dm_id>` is the DCE UUID of the DomainManager (with colons replaced by underscores).

If the file system for an application factory is not available when the DomainManager starts, it will just continue its start-up, but if that file system is mounted at a later time it will then restore the application factory.

DomainManager accepts the following arguments:

`-fsmount <path>`
: The `fsmount` option changes the mapping of the `/dmfs` mount point in DomainManager to the underlying file system location specified by `<path>`. If `<path>` is not specified, the default is `/dmfs` mapped to the current working directory.

`-ORBInitRef NameService=<IOR>`
: A standard ORB argument for specifying the initial reference for a Naming Service. If this argument is specified, the DomainManager object is bound into this Naming Service, and an in-process NamingService is not created.

`-CFLogLevel ( D, I, W, E )`
: Set the output logging level for use by the DomainManager process. Specify one of the settings `D`, `I`, `W` or `E` to set the minimum severity level to `Debug`, `Info`, `Warning` or `Error` respectively. The default level is `W` (or `D` when running the debug _g version). Setting to a lower severity level implies all higher levels are set (therefore `D` reports all messages).

-DMD_FILENAME file.dmd.xml

The name of the DMD file name used to configure the DomainManager. The filename should be relative to the file system mounted by the DomainManager, which is the directory from which the DomainManager is run. When run outside of the BootLoader launching utility, the path must be prefixed with `/dmfs/` so the file can be located from the File Manager.

-ORBListenEndpoints <endpoint address>

A standard ORB argument for specifying an endpoint on which incoming requests will be listened for. This argument can be used to control or fixate the endpoint address for the CF components.

An example endpoint configuration for IIOP protocol would be

```
-ORBListenEndpoints iiop://10.0.0.1:2891
```

-ORBPOAEndpoints <POA_NAME>:<endpoint address>

A Spectra ORB argument for specifying an endpoint on which incoming requests will be listened for. In the context of the DomainManager, this argument is used to control or fixate the endpoint address of the in-process Name Service. The in-process Name Service is always created in an ORB POA called `NAMING_POA`, therefore the argument to `-ORBPOAEndpoints` must always start with `NAMING_POA` as the POA name.

An example endpoint configuration for IIOP protocol would be

```
-ORBPOAEndpoints NAMING_POA:iiop://10.0.0.1:2896
```

i **NOTE:** There is no distinction between command line arguments and execparams. All command line arguments can be added as execparams in the `PRF` file by omitting the leading '`-`' from the argument name. The arguments listed above may therefore be regarded as additional to the properties listed in *Table 1* below.

Table 1 DomainManager properties

Name	Type	Description	Default
Logging			
`PRODUCER_LOG_LEVEL`	simplesequence of long configure	Enabled log levels	DISABLED
Naming Service			

Table 1 DomainManager properties

Name	Type	Description	Default
NAMESERVICE_ENDPOINTS	simple string execparam	Set of endpoints the DomainManager naming service will listen on. The Format is identical to -ORBListenEndpoints (example: "iiop://10.1.0.46:12346,iiop://10.1.0.46:12347").	iiop://<IP>:<PORT> where IP is the IP address of the current machine and PORT is the port number.
EXTERNAL_NAMESERVICE	simple string execparam	If the EXTERNAL_NAMESERVICE property is supplied, the DomainManager will federate its Name Service into the specified naming service. If this property is specified, NAMESERVICE_BINDING must also be specified.	NOT SET
NAMESERVICE_BINDING	simple string execparam	The Name under which the DomainManager will bind itself into the external Naming Service	NameService
Thread Pools			
THREADPOOL_MAX	simple long execparam	Maximal number of threads in DomainManager thread pool. Set to 0 (zero) for unlimited queue	10
THREADPOOL_QUEUE_MAX	simple long execparam	Maximal number of jobs in DomainManager. Set to 0 (zero) for unlimited queue	100
THREADPOOL_SIZE	simple long execparam	Initial number of threads in DomainManager thread pool	10

Table 1 DomainManager properties

Name	Type	Description	Default
THREADPOOL_STACK_SIZE	simple long execparam	Per-thread stack size in DomainManager thread pool. Set to 0 (zero) to use OS default stack size for threads	0
THREADPOOL_NO_POOL	simple long execparam	If set to 1 the DomainManager will not use a thread pool for its operations	0
EORB_SOCKET_LINGERTIME	simple long execparam	Specifies the time to wait to ensure that all data is read once one end of the connection is closed. It may be useful to set this property to 0 on embedded systems with limited sockets so that they're made available for re-use more quickly.	NOT SET
CONFIG_DELEGATION	simple boolean execparam & config	When delegation is off, the ApplicationFactory will configure each resource individually during create, and only the assembly controller's properties will be passed to the assembly controller. When delegation is true, all configure properties are passed to the assembly controller.	false

2.2.3 DeviceManager

The DeviceManager component is used to manage Device and Service components. It controls the lifecycle of the Device and Service components described in the Device Configuration Descriptor (DCD) file.

In a similar way to the DomainManager, the DeviceManager executable that is required to start is chosen based on the operating system platform and properties of the implementation section as described in the Software Package Descriptor (SPD) file. This selection and activation of the appropriate implementation is peformed by the BootLoader.

Upon deployment, the DeviceManager will configure and activate its Devices and Services, and register itself and these child components to the DomainManager. The DeviceManager provided by Spectra CF locates the DomainManager using a Naming Service and the `domainmanager` element of the DCD file as the designated lookup id.

DeviceManager accepts the following arguments:

`-fsmount <path>`
: The `fsmount` option specifies the path on the local file system that the DeviceManager uses as the root of its `filesystem` attribute. The file system will then be mounted in the DomainManager's file manager when the DeviceManager registers itself. If the `fsmount` option is not specified, the file system will be rooted at the current working directory.

`-ORBInitRef NameService=<IOR | corbaloc>`
: The IOR or corbaloc of the NameService context that the DomainManager is bound into.

Table 2 DeviceManager execparams

Name	Type	Description	Default
Logging			
`PRODUCER_LOG_LEVEL`	simplesequence of long configure	Enabled log levels	DISABLED
DomainManager Registration			
`RESOLVE_MAX_RETRIES`	simple long execparam	The number or times the DeviceManager will attempt to resolve the DomainManager from the Name Service. Set to `-1` for continuous retries	`10`

Table 2 DeviceManager execparams

Name	Type	Description	Default
RESOLVE_SLEEP_TIME	simple long execparam	The amount of time to sleep between successive calls when resolving DomainManager from Name Service. Time is in milliseconds	2000
REGISTER	simple long execparam	DeviceManager will not register itself or its components with a DomainManager if this property is set to 0 (zero)	1
Thread Pools			
THREADPOOL_MAX	simple long execparam	Maximal number of threads in DeviceManager thread pool. Set to 0 (zero) for unlimited queue	10
THREADPOOL_QUEUE_MAX	simple long execparam	Maximal number of jobs in DeviceManager. Set to 0 (zero) for unlimited queue	100
THREADPOOL_SIZE	simple long execparam	Initial number of threads in DeviceManager thread pool	10
THREADPOOL_STACK_SIZE	simple long execparam	Per-thread stack size in DeviceManager thread pool. Set to 0 (zero) to use OS default stack size for threads	0
THREADPOOL_NO_POOL	simple long execparam	If set to 1 the DeviceManager will not use a thread pool for its operations	0

Table 2 DeviceManager execparams

Name	Type	Description	Default
SharedLibrary support			
CPP_ORB_LIBRARIES	simplesequence of string configure	Libraries dynamically loaded before the DeviceManager starts a C++ ORB in its address space	[libe_orb.so, libe_poa.so, libe_naming_c.so, libe_any.so, libe_lwevent_s.so, libe_lwlog_s.so, libcf_cpp_s.so]
CPP_ORB_ARGS	simple string execparam	Arguments passed to the co-located C++ ORB CORBA::ORB_init() function	-

The DeviceManager's thread pool settings determine the amount of parallelism to be employed when launching devices and services. Each device/service will be launched in its own thread, if there are sufficient threads available.

2.2.3.1 Launching DomainManager from DeviceManager

In order to launch the DomainManager as a component from the DeviceManager's DCD, a `componentfile` entity is added pointing to the DomainManager's SPD file and `componentinstantiation` is also added. For example:

```
<componentfiles>
   <componentfile id="DCE:b6200cee-ef65-44dc-ba72-064efa0e3b3c"
type="Software Package Descriptor">
      <localfile name="DomainManager.2.2.2.spd.xml"></localfile>
   </componentfile>
   <!-- Other component files here -->
</componentfiles>
<partitioning>
   <componentplacement>
      <componentfileref
refid="DCE:b6200cee-ef65-44dc-ba72-064efa0e3b3c"></componentfileref
>
      <componentinstantiation
id="DCE:cfc887ba-f200-4e58-8bfc-4c1084a259d7">
         <usagename>DomainManager</usagename>
      </componentinstantiation>
   </componentplacement>
   <!-- Other component placements here -->
</partitioning>
```

The DomainManager's `PRF` file must contain all the arguments that are usually passed to DomainManager on the command line, so an `execparam` for `DMD_FILENAME` would be added. For example:

```
<simple id="DMD_FILENAME" type="string" mode="readwrite"
name="DMD_FILENAME">
   <value>/dmfs/DomainManager.2.2.2.dmd.xml</value>
   <kind kindtype="execparam"/>
</simple>
```

When starting the DeviceManager, it still needs to know the object reference on the Naming Service so it can resolve the DomainManager instance. As this won't have been created yet, the following corbaloc template can be used: `corbaloc:iiop:1.1@<ip address>:2809/NameService`

2.2.4 Naming Service

It is not necessary to have a Naming Service running before starting the DomainManager, as it normally starts a Naming Service instance during its own initialization process, and binds itself to that.

If required this default behaviour can be overridden, enabling Spectra CF components to interact with third-party Naming Services or instances of a Naming Service that are already running. This is done with an extra command-line argument supplied to BootLoader:

```
   BootLoader -DMD_FILENAME DomainManager.dmd.xml -ORBInitRef
   NameService=<IOR>
```

The argument `-ORBInitRef NameService=<IOR>` contains a reference to an external Naming Service. BootLoader passes `-ORBInitRef=<IOR>` to any DomainManagers and DeviceManagers that it starts.

2.2.5 Cfadmin

The `cfadmin` utility is a basic but convenient tool that provides the capability to easily install, create, start, stop, release and uninstall applications. `Cfadmin` also has an option to view DomainManager registered components.

The `cfadmin` tool may be driven directly from the command line by passing the appropriate arguments to it (see Section 2.2.5.1, *Command-line usage of cfadmin* below) or controlled interactively through its own console menu system (see Section 2.2.5.2, *Interactive control of cfadmin* below).

`Cfadmin` normally requires an argument in the form of `-ORBInitRef NameService=<IOR | corbaloc>` in order to resolve the DomainManager from the given NameService. The default name that it will try to resolve is `CF_DomainManager/DomainManager`, which is used by the `connections` example. However, this name can be changed by specifying the `-NAME_BINDING` parameter followed by the required name. For example:

```
cfadmin -ORBInitRef
NameService=corbaloc:iiop:1.1@192.168.0.1:2809/NameService
-NAME_BINDING myDomain/DomainManager
```

If `cfadmin` is run without any arguments, a prompt will appear so that they can be typed in. This allows `cfadmin` to be run on embedded platforms where it is less practical to launch a program with command line arguments.

2.2.5.1 Command-line usage of cfadmin

The `cfadmin` tool may be driven directly from the command line by passing the appropriate arguments to it.

The argument `-v` displays the CF version, and `-h` or `-help` displays the following usage message:

```
usage: cfadmin [OPTIONS] -ORBInitRef NameService=<IOR|corbaloc>

Mandatory Parameters:
   <IOR|corbaloc>          - The CORBA Object reference of the NameService
                             containing the DomainManager.

Options:
   -NAME_BINDING <DM_NAME> - The fully-qualified name of the DomainManager
                             in the NameService. The default is
                             CF_DomainManager/DomainManager.
   -h                      - Displays this usage.

   -a                      - List applications.
   -f                      - List application factories.
   -m                      - List device managers
   -i <application>        - Install an Application using the specified
                             SAD file.
   -u <id>                 - Uninstall an Application (identified by ID)
   -c <id> <name>          - Create an application using application
                             factory <id> and <name>
   -o                      - Shutdown the domain manager
   -e <id>                 - Shutdown the device manager (identified by
                             device DCE ID)
   -s <name>               - Start an application.
   -t <name>               - Stop an application.
   -r <name>               - Release an application.
   -w <devices> <services>
      <sleep (ms)>
      <retry count>        - Process blocks until specified number of
                             devices and services are available. Will
                             retry specfied number of times, sleeping
                             between each.
                             Returns zero or one depending on success
                             or failure.
```

(Full descriptions of the commands are given in Section 2.2.5.2, *Interactive control of cfadmin*, starting on page 32.)

Below are a few example commands which illustrate how to install, create and start an application:

```
cfadmin -ORBInitRef
NameService=corbaloc:iiop:1.1@10.1.0.4:2809/NameService -i
/dmfs/Application.2.2.2.sad.xml
```

To discover the unique identifier of the Application Factory, list the factories as follows:

```
cfadmin -ORBInitRef
NameService=corbaloc:iiop:1.1@10.1.0.4:2809/NameService -f
Application Factories:
```

```
   1. ExampleApplication.sca
(DCE:86dafff3-6658-4a08-856a-ecdafd4f3dd1)
```

To create an application:

```
cfadmin -ORBInitRef
NameService=corbaloc:iiop:1.1@10.1.0.4:2809/NameService -c
DCE:86dafff3-6658-4a08-856a-ecdafd4f3dd1 TestApplication
Application created with name TestApplication and identifier
DCE:86dafff3-6658-4a08-856a-ecdafd4f3dd1: TestApplication
```

To list the applications:

```
cfadmin -ORBInitRef
NameService=corbaloc:iiop:1.1@10.1.0.4:2809/NameService -a
Applications:
1. TestApplication (DCE:86dafff3-6658-4a08-856a-ecdafd4f3dd1:
TestApplication)
```

Finally, start the application:

```
cfadmin -ORBInitRef
NameService=corbaloc:iiop:1.1@10.1.0.4:2809/NameService -s
TestApplication
Operation successful
```

2.2.5.2 Interactive control of cfadmin

`Cfadmin` accepts single-letter commands; valid commands are shown in a simple context-sensitive menu. For example, when `cfadmin` starts and successfully resolves the DomainManager, this menu is displayed:

```
   Main Menu Options
   =================

    * [I]nstall an application
    * List a device [m]anager
    * Shutdown a d[e]vice manager
    * Shutdown d[o]main manager
    * [V]iew domain objects
    * View [L]ogs
    * [Q]uit program

   Enter option:
```

Entering the letter in square brackets invokes the corresponding command. For example, enter `Q` to shut down `cfadmin` or `I` to install an application. When you enter `I` to install an application, you must then supply the absolute file path (as mounted in DomainManager's FileManager) to the appropriate Software Assembly Descriptor (SAD) file.

Once one or more applications have been installed in the domain, `cfadmin` displays this menu:

```
Main Menu Options
=================

 * [I]nstall an application
 * [U]ninstall an application
 * [C]reate an application
 * List application [f]actories
 * List a device [m]anager
 * Shutdown a d[e]vice manager
 * Shutdown d[o]main manager
 * [V]iew domain objects
 * View [L]ogs
 * [Q]uit program

Enter option:
```

When you choose `C` to create an application, `cfadmin` displays a numbered list of the installed application factories; enter the appropriate number to create the application required.

Stopping `cfadmin` with the `[Q]uit program` command does not affect any applications that have been installed or which are currently running. Running applications must be explicitly stopped and installed applications must be explicitly uninstalled using the appropriate commands.

`Cfadmin` is not *required* to manage applications in Spectra CF. Programmatic methods may of course be used to install, start, stop and uninstall applications, instead of (or in addition to) using `cfadmin`. However, `cfadmin` is a very simple, convenient and reliable tool with low overhead, so although it is an 'optional' component of Spectra CF we recommend that it is always kept available, especially for troubleshooting during application development and testing.

The full list of commands is provided below. The choices available at any given time depend on the state of the domain; for example, the `S` command to start an application only appears when one or more applications have been created. Most options present a numbered list of objects that the operation may be applied to, and the appropriate object can be selected by entering its number when prompted. Note that all commands are case-insensitve.

I *I*nstall an application. This command adds an application to the domain, making it available to run. When this option is selected, the fully-qualified name of the SAD profile to install must be entered. 'Fully-qualified' in this sense means an absolute path from a CF FileSystem that has been mounted in the DomainManager's FileManager. Typically, there will be a FileSystem for each registered DeviceManager as well as the DomainManager's FileSystem (`/dmfs`) available for use. The SAD xml file, and all files referenced from it, must all reside on the same file system.

U *U*ninstall an application. This command removes the selected application from the domain, making it unavailable for further instances to be run. This operation will fail if any application instances are still running (have not been released).

C *C*reate an application. This command starts an instance of an application that has been installed in the domain. When this option is selected, a numbered list of available applications is displayed. Enter the number corresponding to the application to be installed; `0` (zero) cancels the operation. Once an application has been selected, a name is requested to identify that application instance.

f List application *f*actories. This command lists the known application factories.

m List a device *m*anager. This command lists the device manager.

e Shutdown a d*e*vice manager. This command shuts down the selected DeviceManager. All devices and services registered to that DeviceManager are released and become unavailable for use. Any processes running on ExecutableDevices registered to that DeviceManager are terminated. Any applications using any of the registered devices or services should be released to avoid application errors.

o Shutdown d*o*main manager. This command shuts down the domain manager, releasing all components registered in the domain. Running applications are released, registered device managers are shut down, and installed applications are uninstalled.

V *V*iew domain objects. This command can be used at any time to view which objects are available for use in the domain. This operation will display the names and identifiers of Application Factories, Application instances, DeviceManagers, registered Devices and Services.

L View *l*ogs. This command displays the log records written by CF. The producer id, producer name, log level and message are shown for all records received so far.

Q *Q*uit program. This command exits `cfadmin` without affecting the state of the domain.

R *R*elease an application. This command stops the selected application, terminating all application components and releasing all associated system resources. After the operation completes, the application instance is unavailable for use. The application does not have to be already stopped in order for this operation to succeed.

S *S*tart an application. This command calls `start` on the chosen application. As an application carries no state about whether it is running, `cfadmin` does not prevent a running application from being started again. The AssemblyController's logic should handle this situation.

t S*t*op an application. This command calls `stop` on the chosen application. As an application carries no state about whether it is running, `cfadmin` does not prevent a non-running application from being stopped. The AssemblyController's logic should handle this situation.

2.2.6 TargetLoader

The TargetLoader is intended for use during development; it is not required for running any other component of Spectra CF.

The TargetLoader service runs on the target machine and allows files to be copied to a file system local to the target and processes to be executed remotely on the target. The TargetLoader should be run on a machine on which Core Framework (CF) components such as DomainManagers or DeviceManagers are to be deployed. The TargetLoader interface exposes a file system to the target platform to allow files to be copied using CORBA to the target system without relying on protocols such as scp or rcp, which may not be enabled in some embedded kernels. The TargetLoader also allows executables which have been copied onto the TargetLoader file system to be launched and terminated remotely on the target system. The TargetLoader interface is defined using the interface definition language (idl); the definition can be found in `$CFHOME/include/idl/TargetLoader.idl`.

TargetLoader accepts the following arguments:

`-h`
`--help`
Displays usage information.

`-p <port>`
`--port <port>`
The TCP port number that TargetLoader will listen on. If not specified, TargetLoader will listen on port `8277`.

-c <name>
--context <name>
When used in conjunction with a NameService, specifies a context name that the TargetLoader will be bound into. If the context does not already exist, it will be created.

-fsmount <path>
The path to where the root of the CORBA file system will be. If not specified, the directory TargetLoader is run from will be used.

-ORBInitRef NameService=<IOR | corbaloc>
The object reference, as `IOR` or `corbaloc`, of the Naming Service that the TargetLoader will bind itself into.

2.2.7 FileSystemService

The FileSystemService is an implementation of a CORBA file system that is compliant with the Core Framework (CF) specification. Particularly, the FileSytemService implements the `CF::FileSystem` interface. The FileSystemService can be started as a stand-alone service, like a Naming Service, or be launched by a DeviceManager component. The FileSystemService will mount itself into a DomainManager FileManager if the DomainManager Naming Service is set via `-ORBInitRef NameService=<IOR | corbaloc>` and the DomainManager context name is specified using the `DOMAINMANAGER_NS` execparam. The `$CFHOME/examples/FileSystemService` directory contains an example FileSystemService which can be launched with a command such as:

```
BootLoader -SPD_SERVICE_FILENAME
FileSystemService.2.2.2_g.spd.xml -SERVICE_NAME <name>
-DEVICE_MGR_IOR <IOR> -ORBInitRef NameService=<IOR | corbaloc>
```

where `<IOR | corbaloc>` is the DomainManager Naming Service `IOR` or `corbaloc`.

FileSystemService accepts the following arguments:

-fsmount <path>
FS_MOUNT <path>
Physical location the FileSystem will use to store its data. If this option is omitted the current working directory is used.

DOMAINMANAGER_NS <name>
DomainManager Naming Service name. If this property is set, the FileSystemService will attempt to look up the DomainManager and mount itself under the name specified in `FS_MOUNT_NAME`. If no `FS_MOUNT_NAME` parameter is provided the FileSystemService will mount itself under the mount point name “`extern`”.

FS_MOUNT_NAME <name>
Mount point the FileSystem will use during call to the `mount()` operation.

Table 3 FileSystem Service execparams

Name	Type	Description	Default
FS_MOUNT	simple string execparam	Physical location the FileSystem will use to store its data	current directory
DOMAINMANAGER_NS	simple string execparam	DomainManager NamingService name. If this property is set, the FileSystem will attempt to look up the DomainManager and mount itself under the name specified in FS_MOUNT_NAME	-
FS_MOUNT_NAME	simple string execparam	Mount point the FileSystem will use to call the `mount()` operation	extern

If the FileSystemService is added to the DeviceManager's DCD as a `componentinstantiation` entry, the file system can be mounted in the DomainManager's file manager automatically by adding a `filesystemnames` section to your DCD:

```
<filesystemnames>
   <filesystemname mountname="/fs_name" deviceid="DCE:..." />
</filesystemnames>
```

The `<filesystemnames>` entity comes after the `<domainmanager>` entity in the DCD file. The `mountname` will be used when mounting the filesystem into the file manager, and the `deviceid` references the `componentinstantiation` id of the file system service.

When the FileSystemService is started in this way, the `DOMAINMANAGER_NS` and `FS_MOUNT_NAME` parameters are not required.

2.2.8 LogService

The LogService is an SCA-compliant service implementing a log service which instantiates the `CosLwLog` interfaces.

When the LogService is started, it registers itself into the DeviceManager represented by the `DEVICE_MGR_IOR` argument with the name specified by the `SERVICE_NAME` argument. Once registered in the domain, the LogService can be used for application logging by specifying connections to it in an SAD or DCD XML profile, or it can receive log records from Spectra CF by specifying the service name in the services section of the Domain Manager Descriptor profile.

The lifetime of a service started via the BootLoader in this manner is independent of the lifetimes of the DomainManager and of the DeviceManager it is registered to. When the DeviceManager is shut down, the LogService will be unregistered from it, but the process will not be terminated.

The process can be terminated in the usual manner (on Linux systems) by either `Ctrl-C` or the `kill` command. Terminating the service when its DeviceManager is still running will cause the service to be unregistered from the domain before termination.

The LogService can also be started by including an appropriate `componentinstantiation` block in the DeviceManager's DCD profile. An example of this usage can be found in the `examples/connections/DeviceManager.dcd.xml` file. (Note that when the LogService is deployed in this manner, shutdown of the DeviceManager will both unregister the service and terminate its process.)

Starting from BootLoader:

```
BootLoader -SPD_SERVICE_FILENAME LogService.spd.xml SERVICE_NAME
<name>
DEVICE_MGR_IOR <ior>
```

2.3 Spectra ORB plugins link time configuration

All Spectra CF components have a compiled-in default set of Spectra ORB plugins. The default plugins are:

EORB_IIOP

EORB_POA

EORB_Any

The plugin set can be altered on a *per component* basis. This is necessary when a particular component needs to support (for example) an additional transport which is implemented as a Spectra ORB plugin.

Step 1: *Set up the environment*

Set up the Spectra CF environment (`EORBHOME`, `EORBENV`, `CFHOME`) as described in Section 6.2.1, *Environment Variables*, on page 63.

Step 2: *Spectra ORB plugin configuration*

Edit

```
$CFHOME/bin/configurable/<component>/init.c
```

where `<component>` is a Spectra CF component such as DomainManager or DeviceManager.

The default contents should look similar to this:

```
#include "init.h"
#include "eOrbC/PortableServer/POA.h"
#include "eOrbC/CORBA/any.h"

void init(void)
{
   EORB_plugin (EORB_IIOP);
   EORB_plugin (EORB_POA);
   EORB_plugin (EORB_Any);
}
```

Additional plugins can be added now. If additional header files are required they must be added here.

Step 3: *Re-linking*

This step will compile the previously-edited `init.c` and re-link the component binary. The re-linked component binary is then stored under `$CFHOME/bin`.

Note that the original component binary in `$CFHOME/bin` is overwritten.

If the additional Spectra ORB plugins need dedicated libraries, the commented Makefile

```
$CFHOME/bin/configurable/Makefile
```

must be edited.

The actual re-linking is done by:

```
$> cd  $CFHOME/bin/configurable/<component>
$> make release or $> make debug
```

i Debug binaries are always larger than release binaries, as they contain debug information.

2.3.1 Example

This example describes the addition of a UDP-based transport “DIOP” to the DomainManager. The DomainManager Naming Service will be capable of receiving requests using the DIOP transport.

Step 1: *Setting up the environment*

Set up the Spectra CF environment (`EORBHOME`, `EORBENV`, `CFHOME`, `LD_LIBRARY_PATH`) as described in Section 6.2.1, *Environment Variables*, on page 63.

Step 2: *Spectra ORB plugin configuration*

Change the contents of

```
$CFHOME/bin/configurable/DomainManager/init.c
```

to:

```
#include "init.h"
#include "eOrbC/PortableServer/POA.h"
#include "eOrbC/CORBA/any.h"

void init(void)
{

   EORB_plugin (EORB_IIOP);
   EORB_plugin (EORB_POA);
   EORB_plugin (EORB_Any);
   EORB_plugin (EORB_DIOP);
}
```

Step 3: *Re-linking*

The DIOP transport requires the DomainManager to be linked against the Spectra ORB libraries `ec_udp` and `ec_diop`.

In `$CFHOME/bin/configurable/DomainManager/Makefile`, change

```
# Add additional libraries to be linked.
#LD_LIBS +=
```

to

```
# Add additional libraries to be linked.
LD_LIBS += ec_udp ec_diop
```

Next, re-link the DomainManager:

```
$> cd  $CFHOME/bin/configurable/DomainManager
$> make release
```

The DomainManager binary `$CFHOME/bin/DomainManager` now supports DIOP. This can be verified using the connection example:

```
$> cd $CFHOME/examples/connections
$> cp $CFHOME/bin/DomainManager .
```

In order for the DomainManager to create listeners using both IIOP and DIOP protocols, an endpoint for each must be specified and passed to it. This can be done by editing the `DomainManager.2.2.2.prf.xml` properties descriptor and uncommenting the `ENDPOINTS` property. The value can then be set to the following:

```
<value>iiop://<ip>:12346,diop://<ip>:12347</value>
```

where `<ip>` is the IP address of your machine.

The DomainManager can then be started using BootLoader in the usual manner:

```
$> $CFHOME/bin/BootLoader -DMD_FILENAME
DomainManager.2.2.2.dmd.xml
```

Use Spectra ORB's `pior` utility to decode the Naming Service IOR printed to the screen by the DomainManager; this should show that the DomainManager's Naming Service is multi-homed and reachable via IIOP and DIOP.

Spectra CF Installation

CHAPTER

3 Spectra CF Installation

This chapter describes how to set up the Spectra CF product.

IMPORTANT NOTICE

3.1 Platforms

Spectra CF currently supports the host and target operating systems, compiler and chipset combinations listed in the release notes provided with the product distribution. The release notes can be accessed *via* `index.html` in the root directory where Spectra CF is installed.

Products such as INTEGRITY and VxWorks are cross-platform development packages for embedded systems which are installed on host systems. The Spectra CF embedded edition is installed onto the host system alongside the development package.

3.2 Spectra CF Directory Organisation

The default installation directory name is `cf`. The components of the Core Framework are located in the following sub-directories:

`bin`	contains DomainManager, DeviceManager, BootLoader and cfadmin executables, and others
`docs`	contains html and pdf documentation
`etc`	contains licence and configuration files

examples	contains an example of how to deploy SCA applications using Spectra CF; a further subdirectory (`examples/lib`) contains resources required to run the example.
include	contains IDL files and public header files
lib	contains library files

3.3 Installing Spectra CF

Note that immediately after installing Spectra CF (as described below), the user should read the Software License Agreement located at `docs/StdLicenseterms.pdf`. IF THE USER DOES NOT AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT, THEN THE USER MUST UNINSTALL THE SOFTWARE AND MUST NOT COPY OR USE IT.

3.3.1 Installing Spectra CF on Linux

Open a terminal window and change to the directory where you want to create the Spectra CF installation directory (such as `/usr/local/`).

For example:

```
> cd /usr/local/
> tar zxf cf-Ubuntu1404-CF_3_0.tar.gz
```

3.3.2 Installing Spectra CF on Windows

Open a command prompt window and change to the directory where you want to create the Spectra CF installation directory (such as `C:\`).

For example:

```
> cd C:\
> unzip cf-WinXP-CF_3_0.zip
```

(Some Windows installations do not have a command-line `unzip`; in this case use an approprate GUI or third-party un-zipping utility. It is sometimes necessary to select a ‘use directories’ or equivalent option to ensure that the correct directory structure is created when the files are extracted from the archive (see *Spectra CF Directory Organisation* on page 43).)

The Example Application

CHAPTER

4 The Example Application

This chapter describes how to use Spectra CF to run SDR applications.

4.1 Running an example

This section illustrates how to use the components of Spectra CF by referring to example files provided with the distribution. A simple example application and platform, compliant with the supported SDR specification, is located in `cf/examples/connections`. The example uses libraries in the `examples/lib` directory.

4.2 Platform differences

Spectra CF has been designed to be as platform independent as possible, and so have the example applications supplied. The instructions in this chapter apply to Linux systems. If the instructions differ for other target platforms, the reader will be directed to the individual platform chapters.

4.3 The 'connections' example

The example platform implementation consists of DomainManager, DeviceManager and an ExecutableDevice. The ExecutableDevice is used by the DomainManager as the host Device for running Application components.

The example application implementation consists of three `CF::Resource` executables: an AssemblyController, a Transmitter, and a Receiver. Connections specified in the Software Assembly Descriptor (SAD) file are established between these three components when the application is created. When the `start` operation is invoked on the deployed application, the AssemblyController invokes `start` on its resource port. Since this port is connected to the Transmitter and to the Receiver, the `start` operation will be invoked on both. The same is true for the `Application::stop` operation.

In this example, the DomainManager and DeviceManager are both started using BootLoader. This not only allows the correct binary to be detected from the Software Package Descriptor (SPD) file but also passes all execparam properties to the binary on startup. However, the DomainManager and DeviceManager binaries can be run without using BootLoader: just replace '`BootLoader`' with the correct binary name on the command line.

4.3.1 Prepare to run the example

Before running the connections application and platform, set up the environment as described in the relevant platform section, then make working copies of the example files from the original Spectra CF installation location (see *Spectra CF Directory Organisation* on page 43) to a location that can then be mounted or made visible to the target system.

For the purposes of this example, the connections example files have been copied to a physical directory called `/tmp/examples/connections`.

On systems that allow terminal access and filesystem navigation, starting the binaries in the `/tmp/examples/connections` directory will mount this `/tmp/examples/connections` directory as `/dmfs` in SDR FileSystem space.

Step 1: Start DomainManager

Start the DomainManager executable by running BootLoader with the `-DMD_FILENAME` argument. This will locate the appropriate DomainManager executable by comparing the target system’s operating system properties with the implementations that are available and described by the DomainManager’s profile.

On Linux-based systems, run the following command from the `/tmp/examples/connections` directory:

```
$> $CFHOME/bin/BootLoader -DMD_FILENAME DomainManager.2.2.2.dmd.xml
```

WIN The Windows command line is:

```
$> %CFHOME%\bin\BootLoader -DMD_FILENAME DomainManager.2.2.2.dmd.xml
```

Successful deployment of the DomainManager should result in output similar to the following:

```
Use the following line as an argument when starting
DeviceManager:
-ORBInitRef
NameService=corbaloc:iiop:1.0@10.1.0.9:2809/NameService
Or:
-ORBInitRef
NameService=IOR:010000002800000049444c3a6f6d672e6f72672f436f734
e616d696e672f4e616d696e67436f6e746578743a312e300001000000000000
004a0000000101000009000000031302e312e302e390000f90a3200000001820
a00654f5242000000000e446f6d61696e4d616e6167657200000b0000004e41
4d494e475f504f410000020000003000

DomainManager waiting for requests ....
```

When successfully deployed, the DomainManager creates a new instance of the Naming Service and binds a reference to itself into it. The name used to locate the DomainManager reference in the Naming Service can be formed by appending the string "`/DomainManager`" to the `domainmanagerconfiguration` attribute of the dmd xml. This id can then be used by DeviceManager or other components to look up the reference to the DomainManager from the Naming Service. In the connections example the id used to look up DomainManager from the Naming Service is `CF_DomainManager/DomainManager`.

DomainManager also displays the IOR and corbaloc of the Naming Service it is using on the console.

Step 2: ***Start a DeviceManager***

Start the DeviceManager executable by running the BootLoader with the `-DCD_FILENAME` argument. This will locate the appropriate DeviceManager executable by comparing the target system's operating system properties with the implementations that are available and described by the DeviceManager DCD file.

To start a DeviceManager the `-DCD_FILENAME` and `-ORBInitRef NameService=<IOR | corbaloc>` arguments must be supplied:

```
$> $CFHOME/bin/BootLoader -DCD_FILENAME
DeviceManager.2.2.2.dcd.xml -ORBInitRef NameService=<IOR |
corbaloc>
```

WIN The Windows command line is:

```
$> %CFHOME%\bin\BootLoader -DCD_FILENAME
DeviceManager.2.2.2.dcd.xml -ORBInitRef NameService=<IOR |
corbaloc>
```

In these commands `<IOR | corbaloc>` is the IOR or corbaloc string of the Naming Service used by and printed by the DomainManager on the console.

When successfully deployed, the DeviceManager will print out its IOR, which can be passed as an argument when starting stand-alone services, such as the LogService or FileSystemService.

Once all components launched by the DeviceManager have registered in the domain, the DeviceManager will print out the following message:

```
All components are up. DeviceManager waiting for requests ....
```

Step 3: ***Start cfadmin***

Start Spectra CF's `cfadmin` tool, which provides simple methods for basic management of applications on the domain. `cfadmin` locates the DomainManager by way of the `-ORBInitRef` parameter.

Applications can be installed, created, and started by using the command line interface provided by `cfadmin`. There is also the facility to stop, unload and uninstall applications.

Start the `cfadmin` tool with the following command:

```
$> $CFHOME/bin/cfadmin -ORBInitRef NameService=<IOR | corbaloc>
```

WIN The Windows command line is:

```
$> %CFHOME%\bin\cfadmin -ORBInitRef NameService=<IOR | corbaloc>
```

If the `cfadmin` utility successfully resolves the DomainManager, the following menu will be displayed:

```
Main Menu Options
=================

 * [I]nstall an application
 * List a device [m]anager
 * Shutdown a d[e]vice manager
 * Shutdown d[o]main manager
 * [V]iew domain objects
 * View [L]ogs
 * [Q]uit program

Enter option:
```

Entering the letter in square brackets invokes the corresponding command. For example, enter `Q` to shut down `cfadmin` or `I` to install an application. When you enter `I` to install an application, you must then supply the path to the appropriate Software Assembly Descriptor (SAD) file.

After each operation, `cfadmin` provides the option to view the domain. To do this, type `V` at the prompt, and an overview of the system will be displayed including the DeviceManagers, Devices, Services, Applications and Application Factories registered to the domain.

Step 4: Install an application

Install an application onto the domain by typing `I` at the prompt. The DomainManager mounts the physical filesystem to mountpoint `/dmfs`. To locate the SAD file on the filesystem, the fully-qualified name of the file is required.

```
/dmfs/Application.2.2.2.sad.xml
```

Note that cfadmin will install the SAD file if it can be located from the DomainManager's FileManager. If the DomainManager was started from `/tmp/examples/connections`, the SAD file can be located from the root of the DomainManager's file system (`/dmfs`).

Viewing the domain will now show an ApplicationFactory registered to the system.

Step 5: ***Create the application***

Create the application with the `ApplicationFactory::create` operation. The following options are available:

```
Main Menu Options
=================

 * [I]nstall an application
 * [U]ninstall an application
 * [C]reate an application
 * List application [f]actories
 * List a device [m]anager
 * Shutdown a d[e]vice manager
 * Shutdown d[o]main manager
 * [V]iew domain objects
 * View [L]ogs
 * [Q]uit program

Enter option:
```

Create the application by typing `C`. The tool displays the Application Factories that are available for selection. Select the ApplicationFactory to use by typing the corresponding item number and entering a suitable name for the application. The `cfadmin` tool will create an application based on these parameters which will be viewable from the next menu of options.

The options menu should now look like this:

```
Main Menu Options
=================

 * [I]nstall an application
 * [U]ninstall an application
 * [C]reate an application
 * List application [f]actories
 * List [a]pplications
 * [S]tart an application
 * S[t]op an application
 * [R]elease an application
 * List a device [m]anager
 * Shutdown a d[e]vice manager
 * Shutdown d[o]main manager
 * [V]iew domain objects
 * View [L]ogs
 * [Q]uit program

Enter option:
```

After an application is installed and registered with the domain, it can be started and stopped. Start and stop the application by typing `S` and `T` and then entering the appropriate application name.

4.3.2 To remove an application

To cleanly remove an application from the system, it should be released (with `R`) and then uninstalled (with `U`).

4.3.3 To shut down the platform

To shut down a device manager (and all devices and services that have registered with it), enter `E` at the menu and then select the device manager to shut down. To tear down the entire domain, enter `O` at the menu to terminate the DomainManager, all device managers, devices, services and applications.

SharedLibrary Support

CHAPTER

5 SharedLibrary Support

This chapter describes the use of shared libraries in Spectra CF .

5.1 Introduction

When a particular component is implemented in a shared library, it is specified by a Software Package Descriptor (SPD) implementation with a code type "`SharedLibrary`", like this:

```
<implementation id="DCE:234514ca-f5dc-4b40-93a2-1509930e413a">
<code type="SharedLibrary">
<localfile name="libMyLib.so"/>
<entrypoint>main_entry</entrypoint>
</code>
<os name="Linux"/>
</implementation>
```

In order to exectute such a component, the library must be loaded and then the symbol defined by `<entrypoint>main_entry</entrypoint>` must be executed.

Warning

A SharedLibrary component runs in the same address space as the facility which has loaded it. It is therefore possible for a SharedLibrary component to make the facility crash.

For example, if a resource has a code type of SharedLibrary, it will run in the same address space as the ExecutableDevice it has been deployed on, *i.e.* as part of the ExecutableDevice's process. If the resource attempts to access memory unavailable to it, it will cause the resource and the hosting device to crash. If the resource was Executable rather than SharedLibrary, only the resource itself would crash.

Devices can also be SharedLibrary type, in which case they share the same address space as the DeviceManager launching them. Any fatal errors in any of the components in DeviceManager's address space would cause DeviceManager to terminate.

On some systems, such as VxWorks 5.5.1 (which has no support for processes), *everything* runs in the same address space.

5.1.1 Supported Platforms

SharedLibrary support is provided by Spectra CF as shown on the list of host and target operating systems, compiler and chipset combinations given in the release notes provided with the product distribution. The release notes can be accessed *via* `index.html` in the root directory where Spectra CF is installed.

5.2 DeviceManager SharedLibrary support

In order to launch components (Devices, Services and Resources) that are implemented as shared libraries, Spectra CF supports the code type "`SharedLibrary`". Spectra CF will dynamically load the library and execute the entry point function according to that component's Software Package Descriptor (SPD) file implementation element. The entry point function must have the ANSI C main function `argc/argv` signature. The component may be written in C or C++.

When a shared library is loaded, a constructor function is located and run. The constructor takes no arguments, and returns an integer indicating the success or failure of the function. A return value of `0` indicates that the constructor ran successfully and the library is ready for use; any other value indicates failure. For GNU compilers, the constructor function is denoted using standard `__attribute__` directives, for example:

```
int __libMyLib__init() __attribute__ ((constructor));
```

For non-GNU compilers, the constructor function is found by using a naming convention based on the name of the library being loaded. In the example XML extract above, the library was named `libMyLib.so`. The "`.so`" extension is removed from the name, and the constructor name is created by pre-pending a double underscore "__" and appending "`__init`".

The intent of the constructor function is to initialise any global data needed by the library, such as a mutex to protect against concurrent access when multiple component instantiations are launched.

A destructor function is called just before the library is unloaded from the system, using the same naming convention as the constructor, for example:

```
int __libMyLib__fini() __attribute__ ((destructor));
```

The implementation of the destructor function should free any resources acquired in the constructor.

When a component is executed, the entry point function is called in a new thread so that the component that spawned it (a DeviceManager or ExecutableDevice) can continue regardless of how the entry point is implemented.

During component termination, another function is located based on a name derived from the entry point to handle the termination of that component instantiation. The `shutdown` function should be named `<entrypoint>_shutdown`. For example, if the entry point is named `main_entry`, the corresponding shutdown function will be called `main_entry_shutdown` and will have the following signature:

```
int main_entry_shutdown(char * id, int options);
```

The `id` argument is either the device's or the resource's component identifier, or the service's name. The `options` argument is reserved for future use. The `shutdown` implementation should clean up all resources for the component being terminated, and should deactivate the CORBA Object if it is not deactivated by the `releaseObject` operation.

For SharedLibrary components linked with Spectra ORB, all components running in the same address space will share the ORB singleton. A component can acquire a handle to the ORB by calling the `ORB_init()` function. This has two important implications. First, SharedLibrary components must not destroy the ORB. Second, Spectra ORB plugin statements have no effect, so SharedLibrary devices will use the DeviceManager's ORB plugins, and SharedLibrary resources will use their loading device's ORB plugins. The set of DeviceManager's ORB plugins can be altered using link-time configuration of the DeviceManager binary (see Section 2.3, *Spectra ORB plugins link time configuration*, on page 37). The default set of e*ORB plugins available for C SharedLibrary components is: `EORB_IIOP`, `EORB_POA`, and `EORB_Any`.

If the SharedLibrary component uses the C++ ORB, the DeviceManager will provide a C++ ORB instance. Since the DeviceManager is not linked against the C++ ORB libraries, they are loaded dynamically before a SharedLibrary component is started. The set of C++ ORB libraries can be modified using the DeviceManager `CPP_ORB_LIBRARIES` property. The `SharedLibraryCPP` example (Section 5.4, *SharedLibraryCPP Example*, on page 58) demonstrates the DeviceManager SharedLibrary component support. The set of C++ ORB plugins available to C++ SharedLibraries can be modified through link-time configuration of the library "`cpporb_launcher`", which is in the `$CFHOME/lib` directory (see also Section 2.3, *Spectra ORB plugins link time configuration*, on page 37). The default set of Spectra ORB plugins available to C++ SharedLibrary components is: `EORB_IIOP`, `EORB_POA`, and `EORB_Any`.

If no `CPP_ORB_LIBRARIES` property is specified, the DeviceManager will load the default set:

```
libe_orb.so, libe_poa.so, libe_naming_c.so, libe_any.so,
libe_lwevent_s.so, libe_lwlog_s.so, libcf_cpp_s.so.
```

The library `libcf_cpp_s.so` contains the generated C++ stub and skeleton code of the Core Framework (CF) interfaces and is located in `$CFHOME/examples/lib`.

i Note that VxWorks 5.5.1 does not require the C++ ORB libraries to be dynamically loaded. The `CPP_ORB_LIBRARIES` property will be ignored, if specified, as all libraries must be loaded onto the board before starting DeviceManager (please refer to Chapter 8, *Spectra CF on VxWorks 5.5.1*, on page 71 for full instructions).

5.3 ApplicationFactory SharedLibrary support

To facilitate wave forms implemented in shared libraries, the ApplicationFactory will pass (for SharedLibrary application components) the entry point name as the name argument to the `execute()` operation. As all symbols from the shared library will have been loaded into the global symbol table for that address space, the entry point name must be unique amongst all loaded libraries.

5.4 SharedLibraryCPP Example

This example is comprised of a DeviceManager with an EchoService which is implemented in a shared library. Requests are made to the EchoService using a client. The implementation of the EchoService can be found in

`$CFHOME/examples/SharedLibraryCPP/EchoService/echoServiceServer.cpp`

and the implementation of the EchoService client can be found in

`$CFHOME/examples/SharedLibraryCPP/EchoClient/echoServiceClient.cpp`

Step 1: *Start a DomainManager*

The process of starting a DomainManager is exactly the same as described in Chapter 4, *The Example Application*, page 48.

Step 2: *Start the DeviceManager with EchoService*

Start a DeviceManager by using the `DeviceManager.2.2.2.dcd.xml` descriptor from the `$CFHOME/examples/SharedLibraryCPP/EchoService` directory. On Linux, run the following commands:

```
$> cd $CFHOME/examples/SharedLibraryCPP/EchoService
$> $CFHOME/bin/BootLoader -DCD_FILENAME
DeviceManager.2.2.2.dcd.xml -ORBInitRef NameService=<IOR>
```

A successful DeviceManager launch will result in output similar to this:

```
Use the following line as an argument when starting LogServices or
Devices:
-DEVICE_MGR_IOR
IOR:010000001900000049444c3a43462f4465766963654d616e616765723a31
2e300000000001000000000000000440000000010101000a00000031302e312e30
2e323300fa81280000000192180065 4f5242edac8a38030000000e4465766963
654d616e6167657200000400000000b00000000000000
=== echoServiceServer.cpp(89) ::__init_lib()
=== echoServiceServer.cpp(172) ::main_entry of EchoService
All components are up. DeviceManager waiting for requests ....
-ORBInitRef
echoService=IOR:0153dcb70d00000049444c3a4563686f3a312e3000000000
0100000000000002c000000010101300a00000031302e312e302e3233007a94
1000000000800000010000001000000ec88fa3f00000000
=== echoServiceServer.cpp(281): exiting EchoService
```

On successful launch, the IOR for the EchoService will be written to the terminal.

Step 3: *Run the EchoService client*

The EchoService client must be started with the `-ORBInitRef` argument printed out when the EchoService started. An optional message argument can be passed to the client which will then be echoed by the EchoService.

On Linux, run the following commands to start the EchoService client:

```
$> cd $CFHOME/examples/SharedLibraryCPP/EchoClient
$> ./echoServiceClient -ORBInitRef echoService=<IOR> Hello
```

Executing the client with the argument "`Hello`" should result in output on the client side similar to this:

```
SharedObject example client starting
Hello
SharedObject example client exiting
```

Spectra CF on Linux

CHAPTER

6 Spectra CF on Linux

This chapter describes how to configure and use Spectra CF to run applications on a Linux x86 platform.

6.1 Platforms

This chapter describes the installation and use of Spectra CF on the Linux platforms listed in the release notes provided with the product distribution. The release notes can be accessed *via* `index.html` in the root directory where Spectra CF is installed.

Spectra CF will also run successfully on other common GNU/Linux platforms which include the Linux 2.4 (or later) kernel. Please call PrismTech for support or for porting to platforms other than those listed in the release notes.

6.2 Pre-requisites

Spectra ORB version 2.0 must be installed before Spectra CF can be used to run applications successfully.

6.2.1 Environment Variables

Spectra CF requires environment variables such as `LD_LIBRARY_PATH` to be set. The `EORB*` environment variables listed below must point to a working Spectra ORB installation.

One method of setting environment variables is to store the necessary settings in a file and then enable them by running the source command.

For example, when using Linux with the default bash shell, the file must contain the following values:

```
$ cat setup_cf.sh
export EORBHOME=<Spectra ORB install directory>
export EORBENV=linux-gcc-x86
export PATH=$PATH:$EORBHOME/bin/$EORBENV
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$EORBHOME/lib/$EORBENV
export CFHOME=<CF install directory>
export PATH=$PATH:$CFHOME/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CFHOME/lib
$ source setup_cf.sh
```

NOTE: The value of `EORBENV` varies depending on the target platform. For embedded Linux boards based on an ARM processor, the value of `EORBENV` will be `linux-gcc-arm-linux`. For boards based on a PowerPC processor, the value of `EORBENV` will be `linux-gcc-ppc-linux`.

6.3 Running the example

The only platform specific instruction for running the supplied example (see *The Example Application* on page 47) is to make sure that the directory `examples/lib` is added to the environment variable `LD_LIBRARY_PATH`. So if the example files have been copied to `/tmp` (as described in *Prepare to run the example* on page 48), the command will be:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/tmp/examples/lib
```

Spectra CF on LynxOS 4.0 and 4.2

CHAPTER

7 Spectra CF on LynxOS 4.0 and 4.2

This chapter describes how to configure and use Spectra CF to run applications on LynxOS 4.0.x and 4.2.x platforms.

7.1 Pre-requisites

Spectra ORB version 2.0 must be installed before Spectra CF can be used to run applications successfully.

7.1.1 Environment Variables

Spectra CF requires environment variables such as `LD_LIBRARY_PATH` to be set. The `EORB*` environment variables listed below must point to a working Spectra ORB installation.

One method of setting environment variables is to store the necessary settings in a file and then enable them by running the source command.

For example, when using LynxOS with the default bash shell the file must contain the following values:

```
$ cat setup_cf.sh
export EORBHOME=<Spectra ORB install directory>
export EORBENV=lynxos-gcc-x86
export PATH=$PATH:$EORBHOME/bin/$EORBENV
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$EORBHOME/lib/$EORBENV
export CFHOME=<CF install directory>
export PATH=$PATH:$CFHOME/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CFHOME/lib
$ source setup_cf.sh
```

Alternatively the environment settings can be set in shell startup scripts.

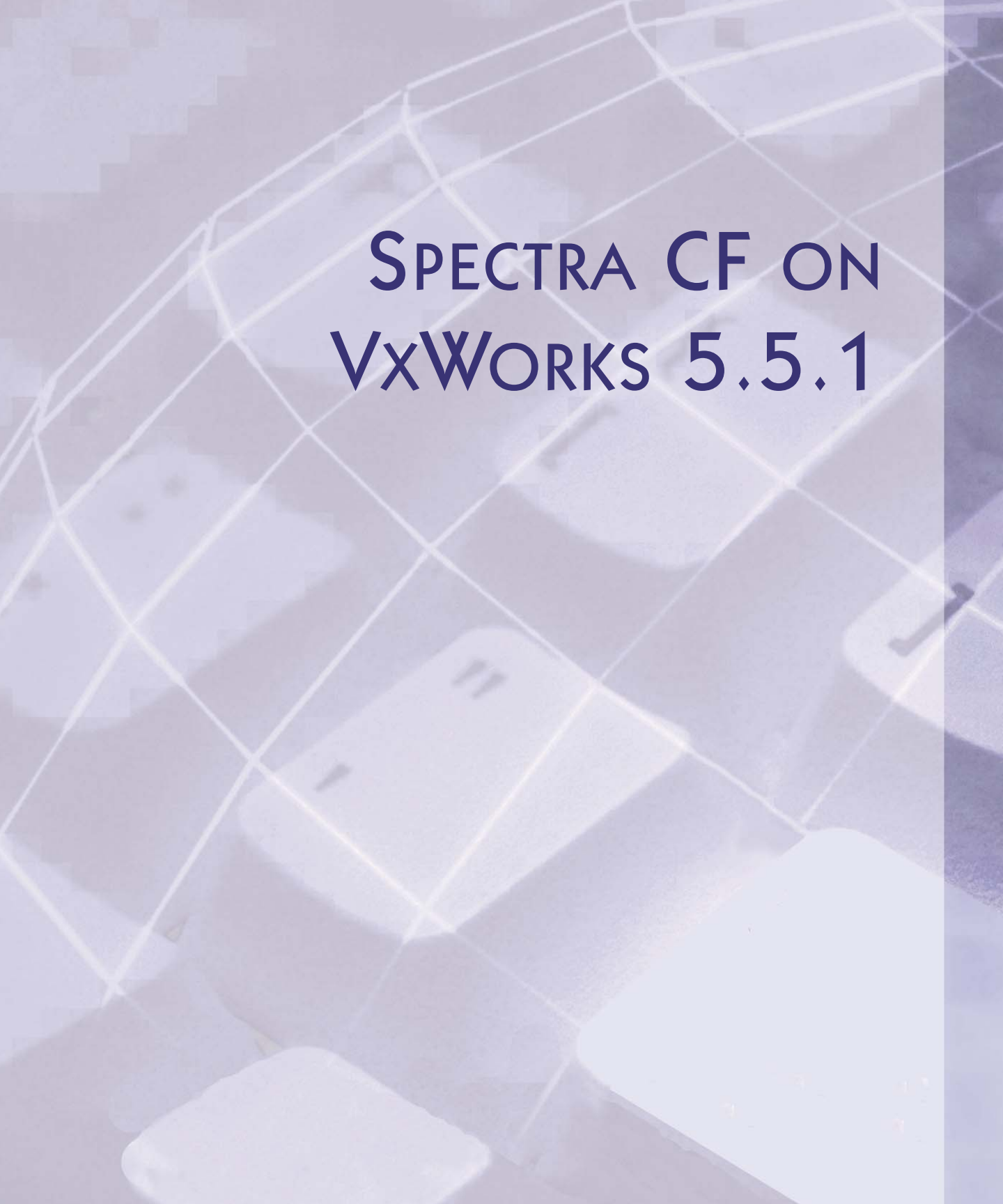
i **NOTE:** The value of `EORBENV` varies depending on the target architecture and host OS. For PPC builds hosted on Windows, the `EORBENV` will be `lynxos-gcc-ppc-win32`, whereas a Linux-hosted `EORBENV` would be `lynxos-gcc-ppc-linux`.

7.2 Running the example

Before running the supplied example (see *The Example Application* on page 47) make sure that the directory `examples/lib` is added to the environment variable `LD_LIBRARY_PATH`. So if the example files have been copied to `/tmp` (as described in *Prepare to run the example* on page 48), the command will be:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/tmp/examples/lib
```

There is a bug in the LynxOS kernel that causes the DeviceManager process to exit prematurely when one of its child processes terminates. Until LynuxWorks provides a patch for this bug, a workaround is to run the DeviceManager as the root user.



SPECTRA CF ON VXWORKS 5.5.1

CHAPTER

8 Spectra CF on VxWorks 5.5.1

This chapter describes how to configure and use Spectra CF *to run applications on the VxWorks 5.5.1 Operating System from Wind River.*

8.1 Platforms

Spectra CF has been built for the VxWorks 5.5.1 chipsets listed in the release notes provided with the product distribution. The release notes can be accessed *via* `index.html` in the root directory where Spectra CF is installed.

8.2 VxWorks 5.5.1 Deployment Model

VxWorks 5.5.1 has no notion of an "executable" or "process", instead the programming model is to run tasks (threads). Spectra CF for VxWorks 5.5.1 deploys all SDR components as VxWorks tasks, and the SDR type of execution is modeled as the "SharedLibrary" type.

This mode of deployment is analogous to the "SharedLibrary" type of execution, where core symbols are pre-loaded onto the board and code for the entry point of each Core Framework component is then loaded and invoked as required.

Spectra CF supports deployment on VxWorks 5.5.1 by providing shared library (`.out`) file versions of each Core Framework component which can be loaded onto the board as required. Before loading these components it is necessary to pre-load the core symbols from the ORB and Core Framework libraries which the component depends upon. *Table 4* lists the required ORB and Core Framework libraries.

The ORB and Core Framework libraries are provided as static libraries (`.a` files). Before these libraries can be loaded onto the board they must be linked into a `.out` shared library.

WARNING: Loading duplicate symbols onto a VxWorks target can result in unexpected errors when running components. To reduce the risk of duplicate symbols being loaded onto the VxWorks target it is highly recommended that users create a single `.out` library containing the common symbols from user components as well as the ORB and Core Framework libraries.

For convenience the Core Framework also provides the `examples-libs.out` library in the `examples/lib` directory which contains the common symbols (from the Core Framework and ORB libraries) that are necessary to run the bundled Core

Framework examples. This library should be pre-loaded on to the VxWorks target, before the shared libraries containing the entry points of the components are loaded and the entry points invoked.

For further details of how to deploy shared library components and an example entry point implementation, please refer to Chapter 5, *SharedLibrary Support*, on page 55.

Table 4 Libraries required for VxWorks deployment

Functionality	ORB Libraries	CF Libraries

C Support	`ec_os`
	`ec_orbs`
	`ec_util`
	`ec_orb`
	`ec_tcp`
	`ec_iiop`
	`ec_poa`
	`ec_any`
	`ec_lwlog_c`
	`ec_lwlog_s`
	`ec_lwevent_c`
	`ec_lwevent_s`
	`ec_naming_c`
	`ec_naming_s`
		`cf_os`
		`cf_impl`

C++ Support (in addition to C support)	`e_orb`
	`e_mpoa`
	`e_any`
	`e_iiop`
	`e_lwnaming_c`
		`cpporb_launcher`

Executable Device (in addition to C support)		`exdev`

All components in VxWorks 5.5.1 are deployed as type “SharedLibrary”, and an entry point must be specified for the component.

8.3 Kernel Configuration

This section describes the components that must be part of a VxWorks kernel to support the Spectra CF runtime. This Guide does *not* describe how to build a VxWorks kernel; please refer to the documentation supplied with VxWorks, which has comprehensive information about the kernel-building process and all of the available options.

8.3.1 Minimum requirements

A minimal kernel to support the Spectra CF runtime can be built with the default properties, together with the following components also enabled:

File System & Disk Utilities (`INCLUDE_DISK_UTIL`)
operating system components > IO system components > File System and Disk Utilities

Posix Timers (`INCLUDE_POSIX_TIMERS`)
operating system components > POSIX components > POSIX timers

NFS Mount (`INCLUDE_NFS_MOUNT_ALL` or `INCLUDE_NFS`)
network components > networking protocols > network filesystems

Synchronize Host and Target Symbols Table (`INCLUDE_SYM_TBL_SYNC`)
development tool components > symbol table components > synchronize host and target symbols table

C++ Components

It is also recommended to increase the network buffer sizes and number of available files to the following values:

```
NUM_SYS_64 1600
NUM_SYS_128 2400
NUM_SYS_256 1600
NUM_SYS_512 1600
```

network components > basic network initialization components > network buffer initialization

```
NUM_FILES 200
```

operating system components > IO system components > IO system

Note that the paths to enable each component have been described because there is no search facility in the Tornado 2.2.1 IDE. Note also that the C++ Components do not have an `INCLUDE` style name and are at the top level, so there is no path to describe for this component.

8.4 Network socket considerations

VxWorks' network connection implementation is such that when a network socket is closed, the default behaviour is that it moves from the `ESTABLISHED` state to the `TIME_WAIT` state. This `TIME_WAIT` state is a period of graceful shutdown where the socket is unavailable for binding to a new destination. The fast, concurrent nature of Spectra CF can mean that there is great demand for network sockets, particularly if there are many components deployed on the target. Because sockets are not

available for re-use for a period of time after being closed (because they are in the TIME_WAIT state), a situation may arise where no more sockets are available for binding.

There is a socket option, named SO_LINGER, that controls the time spent in the TIME_WAIT state when a connection is closed. If the linger time of the SO_LINGER option is 0 (zero), then the TIME_WAIT state is bypassed and the connection resources are immediately freed and available for re-use. Providing network sockets are closed appropriately, this reduces the likelihood of exhausting the available network connections.

The ORB used within Spectra CF is Spectra ORB, which provides a mechanism to set this SO_LINGER option by way of a socket linger policy. Spectra CF has been instrumented to set this policy for DomainManager and DeviceManager if the property named EORB_SOCKET_LINGERTIME is an execparam for the component. The file VxWorks551.prf.xml as provided in Spectra CF's example contains this property:

```
<simple id="EORB_SOCKET_LINGERTIME" type="long" mode="readonly"
name="EORB_SOCKET_LINGERTIME">
    <value>0</value>
    <kind kindtype="execparam"/>
    <action type="eq"/>
</simple>
```

In order to minimize the likelihood of exhausting the number of socket connections on VxWorks, PrismTech recommends using the EORB_SOCKET_LINGERTIME property as shown above for the DomainManager and DeviceManager components.

i Wind River's inetstatShow utility enables you to monitor the active connections in the system.

8.5 Running the Example

The following instructions describe how to run the "connections" example distributed with Spectra CF on VxWorks 5.5.1, using Tornado 2.2.1.

8.5.1 Provide target with access to domain profile

The example domain profile has to be accessible to the VxWorks target system. This example documents the use of NFS in order to do this, although other file systems may be used. Please refer to the *Release Notes* to view the list of supported file systems. To use the NFS technique, a directory containing the domain profile needs to be made available for NFS mounting on to the VxWorks target. The following steps outline this process:

On the host machine, create a directory to be NFS mounted as `SpectraCF`. Within this directory, create a `connections` directory. From the Spectra CF installation, copy the contents of `examples/connections` to this new `connections` directory. This means that the SDR components that are deployed within the `/SpectraCF/connections` directory on the target will have immediate access to the domain profile.

NFS file permission requirements may vary depending on the system configuration, but it has been seen that permissions for the "other" group of these directories need to be readable, writable and executable (VxWorks uses the "other" type of permissions by default). For example, to create the connections directory for the target the following commands would be issued from a cygwin terminal:

```
%  cd C:/
%  mkdir -p SpectraCF/connections
%  chmod o+rwx SpectraCF # so that the directory is available to
read, write and execute
%  chmod o+rwx SpectraCF/connections # so that the directory is
available to read, write and execute
```

This `SpectraCF` directory then needs to be made available to NFS.

8.5.2 Connecting to VxWorks target board using Tornado 2.2.1

Note that this process may vary depending on the board setup and configuration.

Step 1: Start Tornado 2.2.1 GUI.

Step 2: Run Tornado's FTP server (required for locating and booting of VxWorks kernels) **Start > Programs > Tornado2.2.1 > FTP Server**.

Step 3: Using the Tornado GUI, start **TargetServer** for connections to the target board. This is done with **Tools > Target Server > Configure...**. Refer to WindRiver documentation for details of this process.

Step 4: Using the Tornado GUI, attach a Tornado shell to the connected board above. This is done with **Tools > Shell...**. The shell command `nfsDevShow` will display all mounted NFS directories. The directory on the host must be made available to VxWorks by calling `nfsMount`:

```
%  hostAdd("<host name>", "<host ip>")
%  nfsMount("<host ip>", "/SpectraCF", "/SpectraCF")
```

8.5.3 Running the example using the Tornado shell

Using the Tornado shell, first load the `examples-libs.out` library and `BootLoader.out` provided with Spectra CF. To minimize path manipulation, perform a `chdir` to the `/SpectraCF/connections` directory that was mounted above so that all components that are created have local access to the example domain profile.

For example, if Spectra CF has been installed at `C:/PrismTech/SpectraCF/`, the VxWorks loading commands on the shell would be as follows (substitute the installation directory of Spectra CF as necessary):

```
%  ld 1,0,
   "C:/PrismTech/SpectraCF/cf/examples/lib/examples-libs.out"
%  ld 1,0, "C:/PrismTech/SpectraCF/cf/bin/BootLoader.out"
%  chdir ("/SpectraCF/connections")
```

i Note that `examples-libs.out` is the 'release' version of the CF library. It is highly optimized for fast efficient deployment of SDR components. As such, components that use symbols from this library will run with very little printed output. Spectra CF also provides a 'debug' version of this library (`examples-libs_g.out`) which gives more detailed output from each component that is deployed. The print statements slow the deployment process down considerably, but can be useful for debugging.

NOTE: When run from the host shell, no processing of backspace characters is performed so care must be taken when entering arguments at the BootLoader prompt. Alternatively, copy them from a text editor such as notepad.

Step 1: Start the DomainManager executable by running BootLoader with the `-DMD_FILENAME` argument.

Invoke the BootLoader's entry point:

```
%  BootLoader_main
```

When prompted, enter the following parameter:

```
-DMD_FILENAME DomainManager.2.2.2.dmd.xml
```

i (Note that there is an alternative technique for passing run time arguments to the BootLoader and `cfadmin` entry points. Please refer to the *Troubleshooting* section of this chapter on page 78 for details of a technique which precompiles these arguments into a library which can be preloaded on to the target.)

The DomainManager will then be started using the `DomainManager.2.2.2.dmd.xml` file that will be local to the target because of the chdir to `/SpectraCF/connections` that was done above. If successful, the string `DomainManager waiting for requests...` will be output. The IOR and corbaloc of the DomainManager's NamingService should also be output.

Step 2: Start DeviceManager.

The DeviceManager is ready to be deployed and registered with this DomainManager. Deploy the DeviceManager on the same node using the BootLoader in a similar manner as above. Invoke the BootLoader's entry point again:

```
%  BootLoader_main
```

When prompted, enter the `DCD_FILENAME`. For example:

```
-DCD_FILENAME DeviceManager.2.2.2.dcd.xml
```

Step 3: Start `cfadmin`.

Load the `cfadmin.out` entry point code provided with Spectra CF. `cfadmin` can be run on the same board as used above, or a different one if available. If using a different board, the `examples-libs.out` library will need to be loaded before `cfadmin.out` so that any missing symbols are resolved.

```
%  ld 1,0, "C:/cygwin/home/cf/bin/cfadmin.out"
```

Step 4: Invoke the `cfadmin`'s entry point, and pass the NameService parameter from above when prompted:

```
%  cfadmin_main
```

```
-ORBInitRef NameService=<IOR | corbaloc>
```

The `cfadmin` can then locate the DomainManager that is bound into the Naming Service at the specified location by `ORBInitRef`. Applications can then be installed, created, and started by using the command line interface provided by `cfadmin` on the shell terminal. Please refer to the description of `cfadmin` starting on page 29 of this Guide for details of the commands available.

8.6 Troubleshooting

8.6.1 SDR component is started but no output is seen on the Tornado shell

Because of the task model used in Spectra CF for VxWorks 5.5.1, SDR components are deployed within the system as tasks. The default configuration of these tasks is such that logging and debugging output is redirected to the serial port. In order to display the serial I/O, i.e. the CF output, it is recommended that the serial port from the VxWorks target system is connected to a serial I/O terminal such as GTKTerm on Linux, or some other equivalent package like HyperTerminal on Windows.

Alternatively, the VxWorks system can be configured to redirect all I/O to the Tornado shell. This is done by enabling **Redirect Target IO** in the **Console and Redirection** properties of the TargetServer configuration.

8.6.2 Alternative to pasting command line arguments into the Tornado shell

The components within the example need to be passed run-time arguments so that the domain profile and Naming Service can be located. The main entry point of each component can be invoked and the user will be prompted to enter the arguments (such as `-DMD_FILENAME` and `-ORBInitRef`, etc.). This is the simplest technique but it can become awkward to copy and paste strings into the Tornado shell.

For final (non-development) run-time deployment, these arguments can be precompiled into a library which can also be loaded on to the target. These arguments can then be referenced when the entry points of the components are invoked. The example here uses corbalocs rather than IORs because these can be known before the example is run. The corbaloc is formed using the IP address of the target and port from which the NamingService is started (defined within the DomainManager's property file).

For example, a file named `CFArguments.c` could contain the following:

```
char * domainManagerArguments [] =
{
   "BootLoader", "-DMD_FILENAME", "DomainManager.dmd.xml"
};
char * deviceManagerArguments [] =
{
   "BootLoader", "-DCD_FILENAME", "DeviceManager.dcd.xml",
"-ORBInitRef", "NameService=corbaloc:iiop:<ip>:2809/NameService"
};
char * cfAdminArguments [] =
{
   "cfadmin", "-ORBInitRef",
"NameService=corbaloc:iiop:<ip>:2809/NameService"
};
```

This can then be compiled with:

```
%  ccppc -c CFArguments.c
%  ldppc -X -r -o CFArguments.out CFArguments.o
```

This `CFArguments.out` file can be then loaded on to the target on the Tornado shell with the other `.out` files so these precompiled arguments can be passed directly to the components.

For example, BootLoader can be instructed to start the DomainManager by invoking the entry point in the usual argc, argv style:

```
%  BootLoader_main (3, &domainManagerArguments)
```

where `3` is `argc`, the length of the `argv` array.

This strategy enables automation of the loading and deployment of SDR components in the form of a Tornado shell script.

8.6.3 Preventing duplicate object keys

The object keys of the SDR Components that are deployed on to the target hardware are derived from an object identifier generated by the POA together with a random number generated from using the VxWorks system libraries. Although rare, it has been observed that two components deployed on two separate targets can have the same object key. This is because the same random number was actually generated by the two different targets.

The object references (IORs) will always differ because these contain host information as well as the object key data, but identical object keys for different objects can cause problems for CORBA comparison functions. To guard against this possibility, the ORB parameter `-ORBServerId` can be used to ensure that objects running from different ORBs can be distinguished. For example, add `"-ORBServerId target1"` to the BootLoader's options and all CORBA objects created from this ORB will contain the `"target1"` string in their object keys.

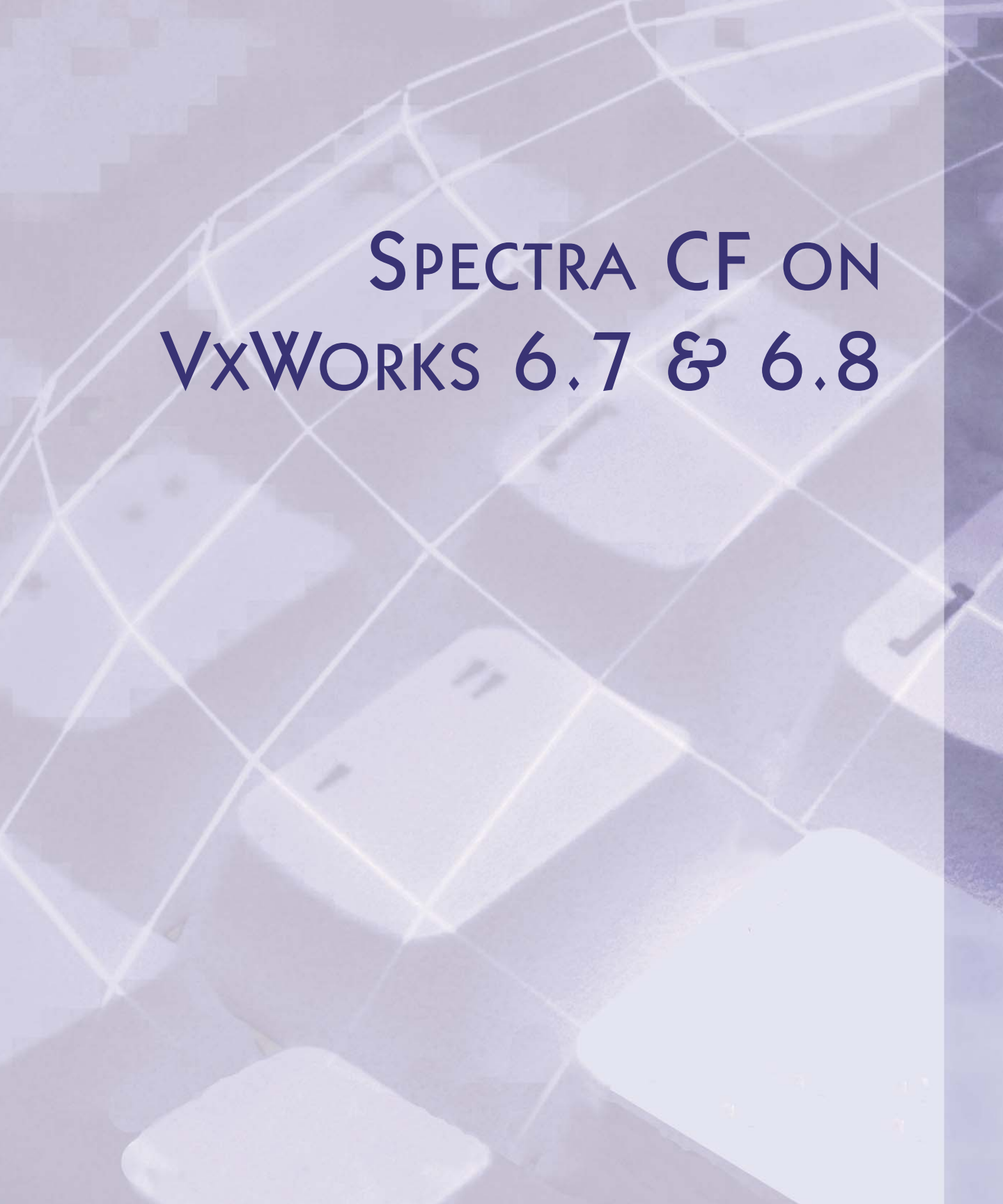
8.6.4 Configuring DomainManager's Name Service

The manner in which the DomainManager can be configured to start the Naming Service at a particular port or endpoint varies slightly from other operating systems. As described in previous chapters, the port or endpoint information is configured by the orb initialization exec params in `DomainManager.prf.xml`. These values will not take effect in VxWorks 5.5.1 because the singleton ORB has already been initialized when it comes to parsing the XML. For this reason, these parameters must be passed directly to the BootLoader; for example:

```
-DMD_FILENAME DomainManager.dmd.xml ORB_NAMESERVICE_PORT 12345
```

or

```
-DMD_FILENAME DomainManager.dmd.xml ORB_NAMESERVICE_LISTEN_ENDPOINTS
iiop://<ip>:12345,iiop://<ip>:12346
```



SPECTRA CF ON VXWORKS 6.7 & 6.8

CHAPTER

9 Spectra CF on VxWorks 6.7 & 6.8

This chapter describes how to set up and use Spectra CF to run applications on the VxWorks Real-Time Operating System from WindRiver.

9.1 Building a VxWorks Kernel for Spectra CF

This section describes the elements or modules that must be part of a VxWorks kernel to support the Spectra CF runtime. This Guide does not describe how to build a VxWorks kernel; please refer to the documentation supplied with VxWorks, which has comprehensive information about the kernel-building process and all of the available options.

Spectra CF for VxWorks 6.7/6.8 gives increased assurances of software reliability by taking advantage of new features in VxWorks 6.7/6.8 . Real-time processes (RTP) protects the kernel from applications and applications from each other.

The functionality required in a Spectra CF capable kernel can be broken down into two categories: *mandatory* base capability (the minimum needed by Spectra CF components), and *optional* capability that can speed up development or make development tasks simpler (such as debugging servers). It is up to the developer to decide how much extra or optional capability is added into a kernel, but the mandatory capability *must* be included for correct operation of Spectra CF.

The mandatory capability is defined by the default Configuration Profile (which is specified when a new kernel is created) plus extra features that must be explicitly added. These additional features are described below.

9.2 Configuration of VxWorks 6.7/6.8

Kernel components can be added to a kernel configuration using the Wind River Workbench IDE Kernel Configuration Viewer or by using the command-line configuration tool `vxprj`.

VxWorks components are identified by the names used in component description files, which are in the form of `INCLUDE_RTP`. Similarly, configuration parameters are identified by their configuration parameter names, such as `NUM_FOO_FILES`. Component and parameter names can be used directly to identify components and configure VxWorks. Wind River's Workbench provides more detailed descriptions of the components and their parameters in the Graphical User Interface (GUI). You

can also use a simple search facility to locate a component based on its component name. Once you have located the component, you can edit the component's parameters through the GUI.

The IDE component hierarchy uses folders (*e.g.* `FOLDER_RTP`) to group components logically. Instead of presenting all components in a flat list, it presents them hierarchically. For example, top-level folders might organize components under headings such as network, drivers, OS, and so on. Folder groupings can affect configuration when a folder is added, because components specified by a folder's `DEFAULTS` property are added all at once.

The IDE provides facilities for configuring VxWorks with selected components, setting component parameters, as well as automated mechanisms for determining dependencies between components during the configuration and build process. The command-line interface (CLI) configuration tool `vxprj` uses the naming convention that originated with configuration macros to identify individual operating system components. The convention identifies components with names that begin with `INCLUDE`. For example, `INCLUDE_MSG_Q` is the message queue component.

For information about the host IDE and CLI facilities used for configuring and building VxWorks, see the Wind River *Workbench User's Guide* and the *VxWorks Command-Line Tools User's Guide*.

9.2.1 Mandatory Capability

The features listed in Table 5, *Mandatory VxWorks kernel components*, on page 85 *must* be included for Spectra CF to run correctly. Note that these features are *in addition to* those provided by the default Configuration Profile.

If any of the listed components are not included in the VxWorks kernel image, linker errors may occur when loading Spectra CF or its applications.

Where several of the required components are available in a single logical grouping, the group or 'folder' name is shown in the table. Not all of the required components are available in such groups.

A specific type of file system must be explicitly enabled so that the SCA components can have access to their domain profiles. The NFS (Network File System) is suitable and convenient to use for this purpose; the following table includes the options required to enable NFS. Other types of file system may be used instead; different options will have to be selected if a file system other than NFS is to be used. Please refer to the Release Notes to view the list of supported file systems.

Table 5 *Mandatory* VxWorks kernel components

Folder	Components	Notes
FOLDER_POSIX	**INCLUDE_POSIX_AIO_SYSDRV** **INCLUDE_POSIX_AIO** **INCLUDE_POSIX_CLOCKS** **INCLUDE_POSIX_DIRLIB** **INCLUDE_POSIX_FTRUNC** **INCLUDE_POSIX_MQ** **INCLUDE_POSIX_MEM** **INCLUDE_POSIX_SCHED** **INCLUDE_POSIX_SEM** **INCLUDE_POSIX_SIGNALS** **INCLUDE_POSIX_PTHREADS** **INCLUDE_POSIX_TIMERS**	Required for SCA and SWRadio Application Environment Profile (AEP) support.
n/a	**INCLUDE_POSIX_PTHREAD_SCHEDULER**	Required for RTPs to schedule Posix threads.
n/a	**INCLUDE_DISK_UTIL**	Enables basic file system functionality (mkdir, open, etc.); specific file system types must also be enabled.
n/a	**INCLUDE_NUM_FILES**	Maximum number of file descriptors available to the entire VxWorks system. This value should be increased to accommodate the requirements of every component to be deployed.

Table 5 *Mandatory* VxWorks kernel components

Folder	Components	Notes
n/a	**INCLUDE_NFS_CLIENT_ALL** **INCLUDE_CORE_NFS_CLIENT** **INCLUDE_NFS2_CLIENT** **INCLUDE_NFS3_CLIENT**	Required in order to nfsMount remote directories into VxWorks' file system. See *Mounting the remote directory using NFS* on page 88.
n/a	**INCLUDE_NET_POOL**	Defines resources available within the TCP stack. TCP resource requirements vary with SDR platforms and waveforms; testing has shown the following values to be useful basic settings: NUM_SYS_32 1600 NUM_SYS_64 1600 NUM_SYS_128 2400 NUM_SYS_256 1600 NUM_SYS_512 800 VxWorks documentation contains memory configuration details.
n/a	**INCLUDE_IPDNSC**	DNS resolver may be required to resolve the NFS machine. **DNSC_DOMAIN_NAME** and **DNSC_PRIMARY_NAME_SERVER** should be set according to your network configuration. Alternatively, calling `hostAdd()` from the shell can associate a host name with an IP address. For example, this `nfsMount` command will work in the same manner: `% hostAdd ("ultra5", "10.1.0.55")`

In addition, if RTP support is required the components in *Table 6* must be included.

Table 6 Components required for RTP support

Folder	Components	Notes
FOLDER_RTP	**INCLUDE_RTP** **INCLUDE_RTP_HOOKS**	Enables real time processes (RTPs) to be managed by the system.
n/a	**INCLUDE_RTP_APPL_USER**	Useful in order to automate an nfsMount before any RTPs are started. See *Mounting the remote directory using NFS* on page 88.
n/a	**INCLUDE_RTP_FD_NUM_MAX**	The maximum file descriptors allowed for each RTP.

9.2.2 Optional Capability

The VxWorks kernel components listed in the table below are not mandatory, but their inclusion will make development quicker and more straightforward.

Table 7 *Optional* (but recommended) VxWorks kernel components

Folder	Components	Notes
FOLDER_SHELL	**INCLUDE_SHELL_INTERP_C** **INCLUDE_DEBUG** **INCLUDE_SHELL_BANNER** **INCLUDE_SHELL** **INCLUDE_SHELL_VI_MODE**	The default line length of `256` may not be long enough to paste IORs; set to `512` if running with the command line.
n/a	**INCLUDE_WDB_TSFS**	Provides the target with access to files on the host system.
FOLDER_NET_SHOW	**INCLUDE_NETSTAT**	Enables `netstat`, `mbufShow`, etc.
FOLDER_EDR	*n/a*	Useful for tracing causes of memory errors.
n/a	**INCLUDE_MEM_EDR**	Useful for tracing causes of memory errors.

Table 7 *Optional* (but recommended) VxWorks kernel components

Folder	Components	Notes
n/a	**INCLUDE_EDR_SHOW**	Use with the shell to see stack traces etc.
n/a	**INCLUDE_MEM_EDR_SHOW**	Use with the shell to see stack traces etc.
n/a	**INCLUDE_MEM_EDR_RTP_SHOW**	Use with the shell to see stack traces etc.

9.3 Mounting the remote directory using NFS

A remote file system directory can be NFS mounted using the kernel shell (if enabled). As an example, the following command will mount the directory `/var/sun16/users/cf/vxworksfs/` on a machine named `ultra5` as "`/cf`" on VxWorks:

```
%  nfsMount ("ultra5", "/var/sun16/users/cf/vxworksfs", "/cf")
```

To ensure that an RTP has "`/cf`" as its working directory, the following shell command can be used:

```
%  cd "/cf"
```

9.3.1 Using the RTP Startup Facility

An alternative to using the shell is to utilise VxWorks' RTP Startup Facility, where each RTP is launched from a user-defined function. To enable this, add the `INCLUDE_RTP_APPL_USER` component to the kernel configuration as shown above. The `usrRtpAppInit.c` file within the kernel configuration can then be edited so that code similar to the example below is executed automatically before an RTP is launched. (Note that the `usrRtpAppInit()` function is called *once*, when the kernel is starting. Additional code may be added to the function body after the call to `chdir` to launch RTPs (DomainManager/DeviceManager *etc.*) via calls to `rtpSpawn()`, passing in the necessary arguments.)

```
int res = nfsMount ("ultra5", "/var/sun16/users/cf/vxworksfs",
"/cf");
if (res == 0)
{
   printf ("usrRtpAppInit: nfsMount was successful\n");
   res = chdir ("/cf");
   if (res == 0)
   {
      printf ("usrRtpAppInit: chdir was successful\n");
   }
   else
   {
      printf ("usrRtpAppInit: chdir failed - errno is %d\n", errno);
   }
}
else
{
   printf ("usrRtpAppInit: nfsMount failed - errno is %d\n", errno);
}
```

Note that the following header includes will be required:

```
#include <stdio.h>
#include <nfs/nfsDriver.h>
#include <ioLib.h>
```

9.4 VxWorks Run-Time Considerations

9.4.1 Memory size considerations

The embedded nature of VxWorks means that memory sizing considerations are very important. For example, an RTP will halt if it exceeds the memory stack that it was allocated. By default, Spectra CF will deploy each SDR component as an RTP with a stack size of `0x10000`. This should be sufficient for most components; however, this stack size value can be overridden on an individual basis by adding the `stacksize` attribute to the relevant implementation of the component's Software Package Descriptor (SPD) xml file.

Spectra CF handles the run-time priority of the RTPs in a similar way. Each component is deployed with the default priority value of `100` (priorities range from `0` [the highest] to `255`). This value can be overriden by adding a priority value to the relevant implementation of the component's Software Package Descriptor (SPD) xml file.

Please refer to the VxWorks documentation for more detail about RTP stack size and priority considerations.

9.4.2 Network socket considerations

VxWorks' network connection implementation is such that when a network socket is closed, the default behaviour is that it moves from the ESTABLISHED state to the TIME_WAIT state. This TIME_WAIT state is a period of graceful shutdown where the socket is unavailable for binding to a new destination. The fast, concurrent nature of Spectra CF can mean that there is great demand for network sockets, particularly if there are many SDR components deployed on the target. Because sockets are not available for re-use for a period of time after being closed (because they are in the TIME_WAIT state), there may become a point where no more sockets are available for binding.

There is a socket option, named SO_LINGER, that controls the time spent in the TIME_WAIT state when a connection is closed. If the linger time of the SO_LINGER option is 0 (zero), then the TIME_WAIT state is bypassed and the connection resources are immediately freed and available for re-use. Providing network sockets are closed appropriately, this reduces the likelihood of exhausting the available network connections.

The ORB used within Spectra CF is Spectra ORB, which provides a mechanism to set this SO_LINGER option by way of a socket linger policy. Spectra CF has been instrumented to set this policy for DomainManager and DeviceManager if the property named EORB_SOCKET_LINGERTIME is an execparam for the component. The file VxWorks67.prf.xml or VxWorks68.prf.xml as provided in Spectra CF's example contains this property:

```
<simple id="EORB_SOCKET_LINGERTIME" type="long" mode="readonly"
name="EORB_SOCKET_LINGERTIME">
    <value>0</value>
    <kind kindtype="execparam"/>
    <action type="eq"/>
</simple>
```

In order to minimize the likelihood of exhausting the number of socket connections on VxWorks, PrismTech recommends using the EORB_SOCKET_LINGERTIME property as shown above for the DomainManager and DeviceManager components.

i Wind River's netstat utility enables you to monitor the active connections in the system. (Add INCLUDE_NETSTAT to the kernel to enable this; see *Table 7* on page 87.)

9.5 Running the Example

i This section describes the process of deploying the RTP version of the example. For downloadable kernel modules please refer to Chapter 8, *Spectra CF on VxWorks 5.5.1*, on page 71 for details about running the example.

The example can be run using either the VxWorks Kernel Shell or the Wind River Workbench GUI.

In either case, the example binaries and domain profile need to be made available to the VxWorks file system. This example describes how to use NFS as this file system; however, other file system mechanisms can also be used. Copy the example files from the original Spectra CF installation to the file system which is to be, or has already been, NFS-mounted. This location can be mounted into the VxWorks system either within kernel code or using the kernel shell, as discussed in *Configuration of VxWorks 6.7/6.8* on page 83.

Ensure the components are started from the NFS mounted location on the target so they have immediate access to the domain profile. For example if the directory has been mounted as `"/cf"`, navigate to `"/cf"` before running any SDR components, so that any components that are started will have `"/cf"` as their current working directory. This navigation can be done both within kernel code or by using the kernel shell.

9.5.1 To use the kernel shell (with NFS)

As well as the example binaries and domain profile mentioned above, the `bootloader.vxe` and `cfadmin.vxe` binaries will need to be copied to the NFS mounted directory.

Note that this example demonstrates use of the command shell (`cmd`).

Step 1: Start DomainManager:

```
[vxWorks *]# BootLoader.vxe -DMD_FILENAME
DomainManager.2.2.2.dmd.xml
Launching process 'BootLoader.vxe' ...
Process 'BootLoader.vxe' (process Id = 0x80d18010) launched.
Use the following line as an argument when starting DeviceManager:
-ORBInitRef
NameService=corbaloc:iiop:1.1@10.1.2.67:2809/NameService
Or:
-ORBInitRef
NameService=IOR:010000002800000049444c3a6f6d672e6f72672f436f734e
616d696e672f4e616d696e67436f6e746578743a312e3000010000000000000000
5000000010101000a00000031302e312e322e363700f90a3200000001820a00
654f5242000000000e446f6d61696e4d616e6167657200000b0000004e414d49
4e475f504f41000002000000300000000000000000

DomainManager waiting for requests ....
```

Step 2: Start DeviceManager:

```
[vxWorks *]# BootLoader.vxe -DCD_FILENAME DeviceManager.dcd.xml
-ORBInitRef NameService=<IOR | corbaloc>
```

Step 3: Start `cfadmin`:

```
[vxWorks *]# cfadmin.vxe -ORBInitRef NameService=<IOR | corbaloc>
```

Applications can then be installed, created, and started by using the command line interface provided by `cfadmin` on the shell terminal. Please refer to the description of `cfadmin` starting on page 29 of this Guide for details of the commands available.

9.5.2 To use the Wind River Workbench GUI (with TSFS and NFS)

The Workbench GUI can only start binaries which can be located from the Target Server File System (TSFS). As a result, the `BootLoader.vxe` and `cfadmin.vxe` binaries need to be copied to TSFS before they can be started. The location of the TSFS is specified when establishing the connection to the target below.

Step 1: Establish Connection to VxWorks Target:

In order to start processes from the Workbench GUI, a connection to the VxWorks target needs to be established. This is done using the Target Manager view. Please refer to the Wind River documentation for a description of target connections and the TSFS.

Step 2: Start DomainManager:

In the Target Manager view of the Wind River Workbench, select the connection to the VxWorks target that was created above. Right-click and select **Run > VxWorks Real Time Process...**. Add a new RTP configuration for the DomainManager. The **Exec Path on Target** should be set to `/tgtsvr/BootLoader.vxe`, and **Arguments** should be set to `-DMD_FILENAME DomainManager.2.2.2.dmd.xml`. The default values for **priority** and **Stack size** of `100` and `0x10000` should be sufficient for the example. Apply these settings and click **Run**.

i Note that the priority and stack size values apply *only* to the BootLoader and *not* to the DomainManager component that it starts. The priority and stack size settings for the DomainManager are obtained from the DomainManager's xml profile. Please refer to *VxWorks Run-Time Considerations* on page 89 for more details.

The BootLoader.vxe that was copied to the TSFS will be deployed onto the VxWorks target. Once running on the target, BootLoader will have access to the NFS-mounted file system containing the domain profile of the example. Using the `-DMD_FILENAME` argument, BootLoader will deploy the DomainManager.vxe

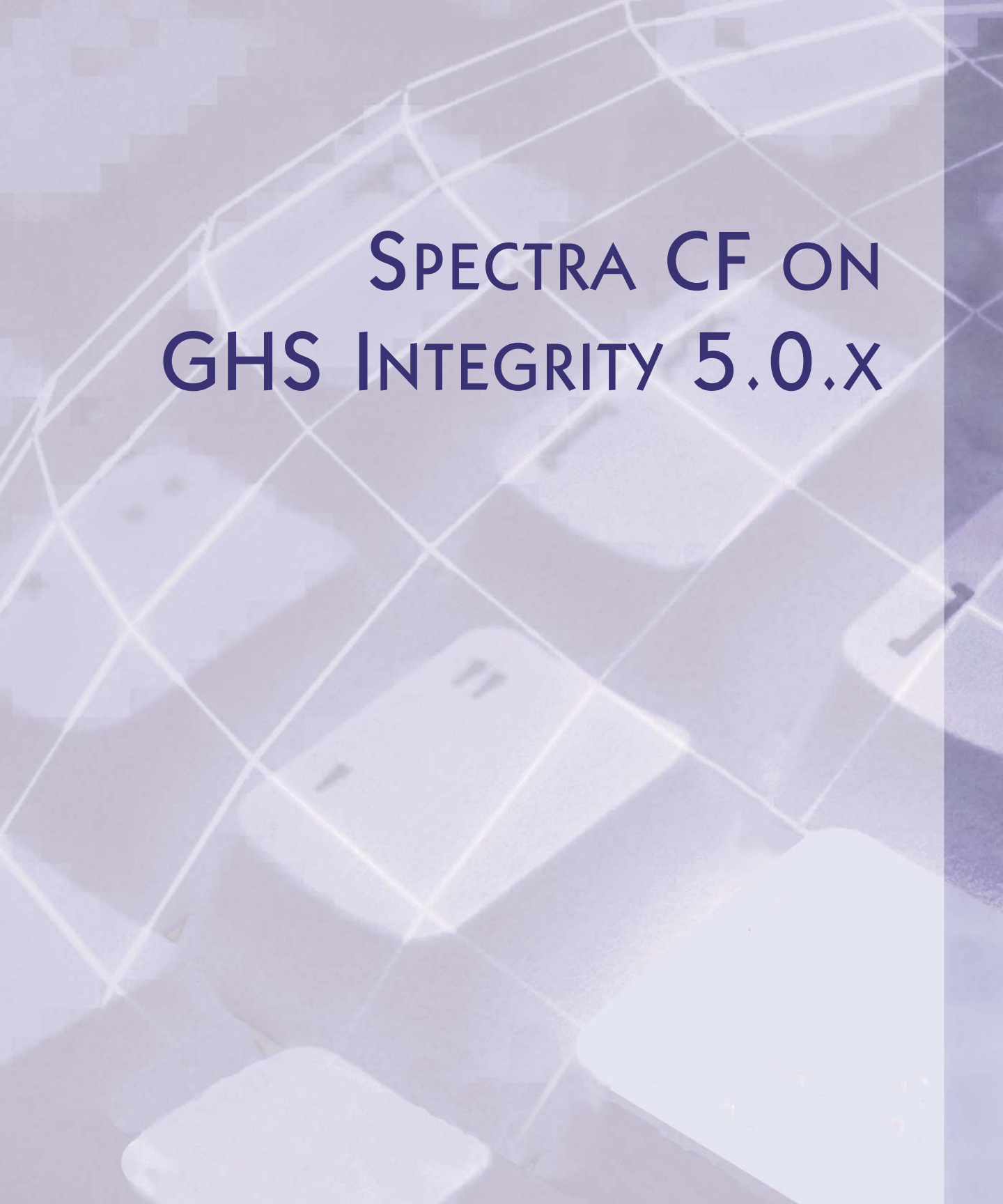
binary located on the NFS-mounted file system. BootLoader and DomainManager output will appear on the VxWorks target terminal. Note that it may take a few moments for the BootLoader to be copied from the TSFS to the VxWorks target.

Step 3: Start DeviceManager:

Add a new RTP configuration for the DeviceManager in the same way as above. The **Exec Path on Target** should be set to `/tgtsvr/BootLoader.vxe`, and **Arguments** should be set to `-DCD_FILENAME DeviceManager.2.2.2.dcd.xml -ORBInitRef NameService=<IOR | corbaloc>`. As above, **priority** and **Stack size** values of `100` and `0x10000` should be sufficient. Apply these settings and click **Run**. As above, this will deploy the `BootLoader.vxe` from the TSFS onto the VxWorks target, except this time it will deploy the `DeviceManager.vxe` from the NFS-mounted file system. In turn, the DeviceManager will deploy the ExecutableDevice and LogService binaries, which will register onto the DomainManager.

Step 4: Start `cfadmin`:

As with DomainManager and DeviceManager, add a new RTP configuration for `cfadmin`. **Exec Path on Target** should be set to `/tgtsvr/cfadmin.vxe`, and **Arguments** should be set to `-ORBInitRef NameService=<IOR | corbaloc>`. After applying these settings and running the configuration, the `cfadmin` will load and start on the VxWorks target. Because the Target Console was enabled before this was deployed, the `cfadmin` I/O will be on the Target Console. Please refer to the description of `cfadmin` starting on page 29 of this Guide for details of the commands available.



Spectra CF on GHS Integrity 5.0.x

CHAPTER

10 Spectra CF on GHS Integrity 5.0.8 and 5.0.10

This chapter describes how to configure and use Spectra CF to run applications on the INTEGRITY Real-Time Operating System from Green Hills Software.

10.1 INTEGRITY Setup and Configuration

10.1.1 Memory Settings using Integrate Configuration Files

Integrate is a post-linker utility supplied by Green Hills Software (GHS) that processes configuration files and ELF binary images to produce loadable ELF binaries. Integrate (`.int`) files contain settings for `heap`, `stack`, `task`, `AddressSpace` and `MemoryPoolSize`, amongst other configuration parameters.

Spectra CF provides integrated (`.mod`) versions of all binaries that are suitable for dynamic downloading onto already running kernels. Spectra CF also provides non-integrated versions of the binaries and example Integrate configuration files. These are required for the cases where configuration parameters (such as `MemoryPoolSize`) need to be modified or for the cases where kernels and multiple ELF binaries need to be integrated into a single loadable ELF binary.

Please refer to the GHS *INTEGRITY Integrate User's Guide* for full explanations on Integrate configuration files.

10.1.2 The POSIX system

INTEGRITY provides two forms of POSIX support:

A Single-AddressSpace POSIX
: POSIX for single-process systems

POSIX System
: Fully-compliant POSIX implementation allowing multi-process applications

Spectra CF requires the use of the full POSIX System to function correctly. Please refer to the chapter “21.4 Setting Up POSIX on INTEGRITY” in the *INTEGRITY Libraries and Utilities Reference Guide*[1] for full details on setting up POSIX. The POSIX System server functionality can be included in virtual AddressSpaces or

1. References are to the documentation supplied with INTEGRITY v5.0.7 and MULTI 4.2.3

built into the kernel. Spectra CF does not prescribe which way POSIX support is built into INTEGRITY. For the purposes of the “connections” example, POSIX support has been built into a kernel.

10.1.3 Shared Libraries

Please refer to section “21.4.2 Setting Up the POSIX System” of the *INTEGRITY Libraries and Utilities Reference Guide*.

Applications using the POSIX system must not be linked with shared libraries.

10.1.4 File Systems

The INTEGRITY VFS module allows different filesystem implementations to be plugged in and used through a common API. Ramdisk FFS and NFS are two filesystems available through INTEGRITY. To build a kernel using other filesystem implementations, please refer to the *INTEGRITY Libraries and Utilities User's Guide*.

i For information about which file systems and which platforms the Core Framework has been tested with, please consult the *Release Notes*.

10.2 Building an INTEGRITY Kernel

This section describes how to create an INTEGRITY kernel that can run the example application supplied with Spectra CF.

WARNING: A ‘CF-capable’ kernel *must* be built with the POSIX server libraries, the GHNet TCP/IP stack, and the Virtual File System (VFS) filesystem support.

For full instructions on how to build a kernel for your system, please consult the relevant Green Hills documentation and release notes for your BSP.

10.2.1 Kernel Requirements

In order to run the CF binaries, the following libraries must be linked into an INTEGRITY kernel image:

- ivfs
- net
- socket

Note that the load library, which is added by default to a new kernel project, *must* be removed due to symbol clashes with the posix library.

10.2.2 Posix System Support

The Core Framework binaries require a POSIX System Server to be built running in either KernelSpace or its own virtual AddressSpace. To build the POSIX System Server into KernelSpace:

Step 1: Link the kernel project with libposix_sys_server_kernel.a.

Step 2: Create a source file containing a user and group database, and add this to your kernel project. An example of what should appear in this file can be found in `INTEGRITY_installation_directory/modules/ghs/posix_sys/posix_sys_server_main.c`. Note that for KernelSpace builds the `main()` function is not required.

Step 3: Optionally increase the size of the kernel's .download section to accommodate any large binaries (> 1.6M). The .download section is used by the POSIX `exec()` call to temporarily store the binary image before being executed. If any waveform or platform components are larger than the default size, then the `__INTEGRITY_DownloadSize` constant in the kernel's `_default.ld` file needs to be increased accordingly.

10.2.3 File System Support

The virtual file system server can either be added to KernelSpace or be used in its own virtual AddressSpace. This guide will document building the VFS server into a KernelSpace project; for information about adding the VFS server to a virtual AddressSpace, please refer to Chapter 10 of the *INTEGRITY Libraries and Utilities User's Guide*.

To add VFS server support to the KernelSpace, the project must be linked with the `ivfsserver` library. Additional libraries must also be linked with the project depending on which file system implementation is desired; for NFS, the `nfs` and `rpc` libraries must be added.

Once the desired file system libraries have been added to the project, a mount table source file, such as `ffs_mounttable.c` (found in `modules/ghs/ffs` in your INTEGRITY installation directory), must be added to the kernel project.

10.2.4 TCP/IP Support

The GHNet TCP/IP stack library provides the ability to configure and IP address for an INTEGRITY kernel. There are several different ways to configure the network stack, please refer to Chapter 2 of the *INTEGRITY Networking Guide* and to the BSP usage notes for more details. The approach described in this section assumes that the network configuration is hard-coded into the kernel project.

Step 1: Link the kernel project with the `libitcpip.a` stack library.

Step 2: Open the `global_table.c` file found in the `_kernel.gpj` project, and uncomment the line:

```
#define HARD_CODE_NETWORK_CONFIGURATION
```

Step 3: Enter the IP address (`IP1..IP4`), network mask (`NM1..NM4`), gateway address (`GW1..GW4`), DNS server (`NS1..NS4`), and host name.

10.3 Running the example

This section describes how to run the '`connections`' example distributed with Spectra CF on Integrity.

Step 1: Copy the files in the `$CFHOME/examples/connections` directory to the file system made available to the Integrity kernel, and ensure that appropriate write permissions are granted to it.

Step 2: Launch Multi and connect to the target board.

Step 3: Select **Target > Load Module** from the menu and select `BootLoader.out` from the `bin` directory in the CF installation location.

Step 4: 4. The BootLoader will start automatically and print a '`BootLoader>`' prompt in the I/O console. Enter the following arguments:

```
-DMD_FILENAME DomainManager.2.2.2.dmd.xml
```

Step 5: Repeat step 4 but enter the following arguments to start the DeviceManager:

```
-DCD_FILENAME DeviceManager.2.2.2.dcd.xml -ORBInitRef
NameService=<IOR>
```

Step 6: Load `cfadmin.mod` from the `bin` directory in the same manner as in the previous steps. At the prompt enter the IOR for the NameService as follows:

```
-ORBInitRef NameService=<IOR>
```

Applications can then be installed, created, and started by using the command-line interface provided by `cfadmin` on the shell terminal. Please refer to the description of `cfadmin` in section *2.2.5* on page 29 of this Guide for details of the commands available.

Spectra CF on Windows

CHAPTER

11 Spectra CF on Windows

This chapter describes how to configure and use Spectra CF to run applications on Windows platforms.

11.1 Platforms

This chapter describes the installation and use of Spectra CF on the Windows platforms listed in the release notes provided with the product distribution. The release notes can be accessed *via* `index.html` in the directory where Spectra CF is installed.

11.1.1 Prerequisites

Spectra ORB version 2.0 must be installed before Spectra CF can be used to run applications successfully. Please refer to the Spectra ORB documentation for details.

11.1.2 Environment Variables

Spectra CF requires environment variables such as `PATH` to be set. In addition to the settings for Spectra ORB, you must set the `CFHOME` environment variable to the location where you installed Spectra CF, and you must add the top level `bin` and `lib` directories to the `PATH` environment variable.

One method for conveniently setting environment variables is to store the necessary settings in a batch file and then running them at the Command Prompt. For example:

```
C:\> more setup_cf.bat
set EORBHOME=<Spectra ORB install directory>
set EORBENV=win32-msdev-x86
set PATH=%EORBHOME%\bin\%EORBENV%;%PATH%
set PATH=%EORBHOME%\lib\%EORBENV%;%PATH%

set CFHOME=<CF install directory>
set PATH=%CFHOME%\bin;%PATH%
set PATH=%CFHOME%\lib;%PATH%
C:\> .\setup_cf.bat
```

11.2 Running the Example

The only platform-specific instruction for running the supplied example (see Chapter 4, *The Example Application*, on page 47) is to make sure that the directory `examples\lib` is added to the `PATH` environment variable, so that the additional DLL's for the example binaries can be found. So, if the examples have been copied to `C:\tmp` (as described in Section 4.3.1, *Prepare to run the example*, on page 48), the command line will be:

```
C:\> set PATH=C:\tmp\examples\lib;%PATH%
```