Virtual Disk Development Kit Programming Guide

vSphere Storage APIs for Data Protection (VADP) 6.5 Virtual Disk Development Kit (VDDK) 6.5

This document supports the version of each product listed and supports all subsequent versions until the document is replaced by a new edition. To check for more recent editions of this document, see http://www.vmware.com/support/pubs.

EN-002082-03



You can find the most up-to-date technical documentation on the VMware Web site at:

http://www.vmware.com/support/

The VMware Web site also provides the latest product updates.

If you have comments about this documentation, submit your feedback to:

docfeedback@vmware.com

Copyright © 2008–2017 VMware, Inc. All rights reserved. This product is protected by U.S. and international copyright and intellectual property laws. VMware products are covered by one or more patents listed at http://www.vmware.com/go/patents.

VMware is a registered trademark or trademark of VMware, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies.

VMware, Inc. 3401 Hillview Ave. Palo Alto, CA 94304 www.vmware.com

Contents

About This Book 9

VDDK Components 11

1 Introduction to the Virtual Disk API 11 About the Virtual Disk API 11

	Virtual Disk Library 12
	Disk Mount Library 12
	Virtual Disk Utilities 12
	Backup and Restore on vSphere 12
	Backup Design for vCloud Director 12
	Use Cases for the Virtual Disk Library 12
	Developing for VMware Platform Products 12
	Managed Disk and Hosted Disk 13
	Advanced Transports 14
	VDDK and VADP Compared 14
	Platform Product Compatibility 14
	Redistributing VDDK Components 14
	O I
2	Installing the Development Kit 15
_	-
	Prerequisites 15
	Development Systems 15
	Programming Environments 15
	Visual Studio on Windows 15
	C++ and C on Linux 15
	Java Development for VADP 15
	VMware Platform Products 15
	Storage Device Support 16
	Installing the VDDK Package 16
	Repackaging VDDK Libraries 17
	How to Find VADP Components 17
_	THE 1011 TO 10
3	Virtual Disk Interfaces 19
	VMDK File Location 19
	Virtual Disk Types 19
	Persistence Disk Modes 20
	VMDK File Naming 20
	Thin Provisioned Disk 21
	Internationalization and Localization 21
	Virtual Disk Internal Format 21
	Grain Directories and Grain Tables 21
	Data Structures in Virtual Disk API 21
	Credentials and Privileges for VMDK Access 22
	Adapter Types 23
	Virtual Disk Transport Methods 23
	Local File Access 23
	SAN Transport 23
	HotAdd Transport 24

About the HotAdd Proxy 25 NBDSSL Transport 25 NFC Session Limits 26 SSL Certificates and Security 26

4 Virtual Disk API Functions 27

Virtual Disk Library Functions 27 Alphabetic Table of Functions 28

Start Up 28

Initialize the Library 29

Connect to a Workstation or Server 29

VMX Specification 29

Disk Operations 30

Create a New Hosted Disk 30

Open a Local or Remote Disk 30

Read Sectors From a Disk 30

Write Sectors To a Disk 30

Close a Local or Remote Disk 30

Get Information About a Disk 31

Free Memory from Get Information 31

Metadata Handling 31

Read Metadata Key from Disk 31

Get Metadata Table from Disk 31

Write Metadata Table to Disk 31

Check and Repair Sparse Disk Metadata 31

Disk Chaining and Redo Logs 32

Create Child from Parent Disk 32

Attach Child to Parent Disk 32

Opening in a Chain 33

Redo Logs and Linked Clone Backup 33

Cloning a Virtual Disk 34

Compute Space Needed for Clone 34

Clone a Disk by Copying Data 34

Error Handling 34

Return Error Description Text 34

Free Error Description Text 34

Administrative Disk Operations 34

Rename an Existing Disk 34

Grow an Existing Local Disk 35

Defragment an Existing Disk 35

Shrink an Existing Local Disk 35

Unlink Extents to Remove Disk 35

Shut Down 35

Disconnect from Server 35

Clean Up and Exit 35

Advanced Transport APIs 35

Initialize Virtual Disk API 36

Location of Log Files 37

List Available Transport Methods 38

Connect to VMware vSphere 38

Get Selected Transport Method 39

Prepare For Access and End Access 39

SAN Mode on Linux Uses Direct Mode 39

Clean Up After Disconnect 40

Updating Applications for Advanced Transport 40

	Algorithm for vSphere Backup 40
	Backup and Recovery Example 41
	Multithreading Considerations 42
	Multiple Threads and VixDiskLib 42
	Capabilities of Library Calls 42
	Support for Managed Disk 42
	Support for Hosted Disk 42
	Support for Hosted Disk 42
_	Winteral Diale A DI Commission de 12
5	Virtual Disk API Sample Code 43
	Compiling the Sample Program 43
	Visual C++ on Windows 43
	SLN and VCPROJ Files 43
	C++ on Linux Systems 44
	Makefile 44
	Library Files Required 44
	Usage Message 44
	Walk-Through of Sample Program 45
	Include Files 45
	Definitions and Structures 45
	Dynamic Loading 46
	Wrapper Classes 46
	Command Functions 46
	DoInfo() 46
	.,
	DoCreate() 47
	DoRedo() 47
	Write by DoFill() 47
	DoReadMetadata() 47
	DoWriteMetadata() 47
	DoDumpMetadata() 47
	DoDump() 48
	DoTestMultiThread() 48
	DoClone() 48
	SSL Certificate Thumbprint 48
6	Practical Programming Tasks 49
	Scan VMDK for Virus Signatures 49
	Creating Virtual Disks 50
	Creating Local Disk 50
	Creating Remote Disk 51
	Special Consideration for ESXi Hosts 51
	VMDK File Versions 51
	Working with Virtual Disk Data 52
	Reading and Writing Local Disk 52
	Reading and Writing Remote Disk 52
	Deleting a Disk (Unlink) 52
	Effects of Deleting a Virtual Disk 52
	Renaming a Disk 52
	Effects of Renaming a Virtual Disk 53
	Managing Child Disks 53
	Creating Redo Logs 53
	Virtual Disk in Snapshots 53
	Windows 2000 Read-Only File System 53
	RDM Disks and Virtual BIOS 54
	Restoring RDM Disks 54

Restoring the Virtual BIOS or UEFI 54 Interfacing With VMware vSphere 55 The VIX API 55 Virus Scan all Hosted Disk 55 The vSphere Web Services API 55 Virus Scan All Managed Disk 56 Read and Write VMDK Using vSphere API 56 Back Up First Class Disk (FCD) 56 Backing Up Virtual Disks in vSphere Design and Implementation Overview 57 The Backup Process 57 Communicating With the Server 58 Information Containers as Managed Objects 58 More About Managed Objects 58 Managed Object References 59 Unique ID for a Different vCenter 59 Gathering Status and Configuration Information 59 PropertyCollector Data 59 Useful Property Information 60 Doing a Backup Operation 60 Prerequisites for Backup 60 Create a Temporary Snapshot on the Target Virtual Machine 60 Changed Block Tracking 61 Extract Backup Data from the Target Virtual Machine 61 Delete the Temporary Snapshot 61 The Restore Process 61 Doing a Restore Operation 62 Prerequisites for Restore 62 Restoring an Existing Virtual Machine to a Previous State 62 Creating a New Virtual Machine 63 Accessing Files on Virtual Disks 63 More VADP Details 64 Low Level Backup Procedures 64 Communicating with the Server 64 The PropertyCollector 64 PropertyCollector Arguments 64 Getting the Data from the PropertyCollector 67 Identifying Virtual Disks for Backup and Restore 68 Creating a Snapshot 69 Backing Up a Virtual Disk 69 Deleting a Snapshot 70 Changed Block Tracking on Virtual Disks 70 Enabling Changed Block Tracking 71 Gathering Changed Block Information 71 Troubleshooting 73 Limitations on Changed Block Tracking 73 Checking for Namespace 73 Low Level Restore Procedures 73 Restoring a Virtual Machine and Disk 73 Creating a Virtual Machine 74 Using the VirtualMachineConfigInfo 80 Editing or Deleting a Device 80

6 VMware, Inc.

Restoring Virtual Disk Data 80

Raw Device Mapping (RDM) Disks 80 Restore of Incremental Backup Data 81 Restore Fails with Direct Connection to ESXi Host 81 Tips and Best Practices 81 Best Practices for SAN Transport 82 Best Practices for HotAdd Transport 82 Best Practices for NBDSSL Transport 83 General Backup and Restore 83 Backup and Restore of Thin-Provisioned Disk 83 About Changed Block Tracking 84 HotAdd and SCSI Controller IDs 84 Encrypted VM Backup and Restore 84 Windows Backup Implementations 86 Disabling Automount in Windows Proxy 86 Security and Remote Desktop 86 Working with Microsoft Shadow Copy 86 Application-Consistent Backup and Restore 88 New VSS Support Added in vSphere 6.5 89 The VMware VSS Implementation 89 Linux Backup Implementation 89 Backing Up vApps in vCloud Director 91 Introduction to Tenant vApps 91 Prerequisites 92 Other Information 92 Conceptual Overview 92 The Backup Process 93 The Restore Process 93 Use Cases Overview 94 Managing Credentials 94 Finding a vApp 95 Inventory Traversal 95 Using the Query Service 95 Protecting Specified vApps 95 Recovering an Older Version of a vApp 95 Recovering a Deleted vApp 95 Recovering a Single Virtual Machine 95 Backing Up vCloud Director 95 vCloud API Operations 96 Getting Access to vCloud Director 96 Inventory Access 96 Admin Views 97 Admin Extensions 97 Retrieving Catalog information 100 Retrieving vApp Configuration 101 Methods To Retrieve vApp Configuration 101 Virtual Machine Information 102 Preventing Updates to a vApp During Backup or Restore 102 Associating vCloud Resources with vSphere Entities 103 Restoring vApps 105 Conclusion 106 A Virtual Disk Mount API 107

The VixMntapi Library 107 Types and Structures 107

Operating System Information 107 Disk Volume Information 108 Function Calls 108 VixMntapi_Init() 108 VixMntapi_Exit() 108 VixMntapi_OpenDiskSet() 109 VixMntapi_OpenDisks() 109 VixMntapi_GetDiskSetInfo() 109 VixMntapi_FreeDiskSetInfo() 110 VixMntapi_CloseDiskSet() 110 VixMntapi_GetVolumeHandles() 110 VixMntapi_FreeVolumeHandles() 110 VixMntapi_GetOsInfo() 110 VixMntapi_FreeOsInfo() 111 VixMntapi_MountVolume() 111 VixMntapi_DismountVolume() 111 VixMntapi_GetVolumeInfo() 111 VixMntapi_FreeVolumeInfo() 111 Programming with VixMntapi 112 File System Support 112 Read-Only Mount on Linux 112 Sample VixMntapi Code 113 Restrictions on Virtual Disk Mount 113

B Errors Codes and Open Source 115

Recent Changes 115
Finding Error Code Documentation 115
Association With VIX API Errors 115
Interpreting Errors Codes 115
Troubleshooting Dynamic Libraries 116
Open Source Components 116

Glossary 117

Index 119

About This Book

The VMware[®] Virtual Disk Development Kit Programming Guide introduces the Virtual Disk Development Kit (VDDK) and the vSphere Storage APIs – Data Protection (VADP). For VDDK it describes how to develop software using a virtual disk library that provides a set of system-call style interfaces for managing virtual disks on ESXi hosts. For VADP it describes how to write backup and restore software that can be managed by vCenter Server[®] for vSphere.

Revision History

Table 1 summarizes the significant changes in each version of this guide. To view previous versions of this book and other VMware SDK documentation, go to http://www.vmware.com/support/pubs/sdk_pubs.html.

Table 1. Revision History

Revision	Description
2017-06-12	Added section to tips and best practices about backing up encrypted virtual disks; with source code.
2017-01-11	Corrections regarding 2TB virtual disk limits, more about ESXi connections.
2016-11-15	Final version for VDDK 6.5. New support for encryption, NBDSSL compression, and VSS extensions.
2016-03-25	Removed statement that disks not associated with a virtual machine can be mounted.
2016-02-02	Fallback to NBDSSL, SCSI order for disk mount, no linked clone SAN restore, EUC non-support.
2015-07-13	Clarified connection port and RAID support, more whitelist details, added Linux freeze and thaw.
2015-05-05	Documented whitelist and blacklist for device path selection, corrected install procedures.
2014-12-26	Final version for VDDK 6.0. Virtual volumes support. SSL certificate checking now mandatory.
2014-04-08	VDDK 5.5.1 supports GPT. Snapshot quiesce and memory are incompatible. Describe VMDK version 3.
2013-11-08	Fixed several errors involving roles and licensing, physical or virtual proxy, and log level.
2013-10-14	Corrections regarding 32-bit Windows and PackageCode. Removed Reparent and Combine APIs.
2013-09-22	Final version for the vSphere 5.5 release, with new chapter on vApp backup for vCloud Director.
2012-12-21	Bug fix version of the vSphere 5.1 manual: numeric change ID policy, mount restrictions.
2012-10-05	Final version of this manual for the vSphere 5.1 release.
2011-11-18	Bug fix version for 4Q 2011 refresh of the VMware vSphere Documentation Center.
2011-08-22	Final version for the VDDK 5.0 release, subsuming <i>Designing Backup Solutions</i> technical note.
2010-10-12	Bug fix revision for the VDDK 1.2.1 release
2010-08-05	Version for vSphere 4.1 and the VDDK 1.2 public release.
2009-05-29	Final version for the VDDK 1.1 public release.
2008-04-11	Updated version for release 1.0 of the Virtual Disk Development Kit.
2008-01-31	Initial version of the Virtual Disk Development Kit for partner release.

Intended Audience

This guide is intended for developers who are creating applications that manage virtual storage, especially backup and restore applications. It assumes knowledge of C and C++ programming. For VADP development, this guide assumes knowledge of Java.

Supported Platform Products

You can develop VDDK programs using either Linux or Windows, and test them using VMware Workstation or ESXi and vSphere. To develop and test VADP programs, you need a vCenter Server and ESXi hosts, preferably with shared cluster storage.

Document Feedback

VMware welcomes your suggestions for improving our developer documentation. Send your feedback to docfeedback@vmware.com.

VMware Technical Publications

VMware Technical Publications provides a glossary of terms that might be unfamiliar to you. For definitions of terms as they are used in VMware technical documentation go to http://www.vmware.com/support/pubs.

To access the current versions of VMware manuals, go to http://pubs.vmware.com/vsphere-50/index.jsp.

Introduction to the Virtual Disk API

1

This chapter presents the following topics:

- "About the Virtual Disk API" on page 11
- "VDDK Components" on page 11
- "Use Cases for the Virtual Disk Library" on page 12
- "Developing for VMware Platform Products" on page 12

The virtual disk development kit (VDDK) is an SDK to help developers create applications that access storage on virtual machines. The VDDK package is based on the virtual disk API, introduced in this chapter.

The VMware Storage APIs – Data Protection (VADP) use the virtual disk API and a subset of vSphere APIs to take snapshots of virtual machines running on ESXi, enabling full or incremental backup and restore. VADP replaces VMware Consolidated Backup (VCB).

About the Virtual Disk API

The virtual disk API, or VixDiskLib, is a set of function calls to manipulate virtual disk files in VMDK format (virtual machine disk). Function call semantics are patterned after C system calls for file I/O. Using the virtual disk API, you can write programs to manage VMDK files directly from your software applications.

These library functions can manipulate virtual disks on VMware Workstation or similar products (hosted disk) or virtual disks residing on file system volumes of an ESXi host managed by vCenter Server (managed disk). Hosted is a term meaning that the virtualization platform and disk are hosted by a guest operating system such as Windows or Linux.

The VDDK package installs on either Windows or Linux, so you can write VDDK and VADP applications using either system. Applications can manipulate the virtual disks of any operating system that runs on a supported VMware platform product. You may repackage VDDK binaries into your software application after signing a redistribution agreement. See the *VDDK Release Notes* for a list of supported platform products and development systems.

The VDDK and VADP enable you to develop applications that work effectively across multiple virtual disks from a central location.

VDDK Components

The virtual disk development kit includes the following components:

- The virtual disk library, a set of C function calls to manipulate VMDK files
- The disk mount library, a set of C function calls to remote mount VMDK file systems
- C++ code samples that can be compiled with Visual Studio or the GNU C compiler
- PDF manuals and online HTML reference

Virtual Disk Library

VixDiskLib is a standalone wrapper library to help you develop solutions that integrate into VMware platform products. The virtual disk library has the following capabilities:

- It allows programs to create, convert, expand, defragment, shrink, and rename virtual disk files.
- It can create redo logs (parent-child disk chaining, or deltas) and it can delete VMDK files.
- It permits random read/write access to data anywhere in a VMDK file, and reads metadata.
- It can connect to remote vSphere storage using advanced transports, SAN or HotAdd.

For Windows, the virtual disk kernel-mode driver should be 64-bit. User libraries could be 32-bit because Windows On Windows 64 can run 32-bit programs without alteration. VMware provides only 64-bit libraries.

Disk Mount Library

The virtual disk mount library, vixMntapi, allows programmatic access of virtual disks as if they were mounted disk partitions. For more information see Appendix A, "Virtual Disk Mount API," on page 107. The vixMntapi library is packaged in the VDDK with vixDiskLib.

Virtual Disk Utilities

The Virtual Disk Development Kit used to include two command-line utilities for managing virtual disk files: disk mount and virtual disk manager. They were last delivered in the VDDK 5.0 release. For more information see the old *Disk Mount and Virtual Disk Manager User's Guide*, still available on the Web.

Backup and Restore on vSphere

The VMware Storage APIs – Data Protection (VADP) is a collection of APIs that are useful for developing or extending backup software so it can protect virtual machines running on ESXi hosts in VMware based datacenters. For more information see Chapter 7, "Backing Up Virtual Disks in vSphere," on page 57.

Backup Design for vCloud Director

With VMware vCloud[®], the self-service capabilities of vCloud Director provide three levels of data protection. Backup providers can offer vApp protection at the system level, the tenant level, or the end-user level. For information about vCloud data protection, see the technical note *Backup Design for vCloud Tenant vApps*.

Use Cases for the Virtual Disk Library

The library provides access to virtual disks, enabling a range of use cases for application vendors including:

- Back up a particular volume, or all volumes, associated with a virtual machine.
- Connect a backup proxy to vSphere and back up all virtual machines on a storage cluster.
- Read virtual disk and run off-line anti-virus scanning, or package analysis, of virtual machines.
- Write to virtual disk to perform off-line centralized patching of virtual machines.
- Manipulate virtual disks to defragment, expand, convert, rename, or shrink the file system image.
- Perform data recovery or virus cleaning on corrupt or infected off-line virtual machines.

Developing for VMware Platform Products

In a VMware based data center, commercial backup software is likely to access virtual disks remotely, perhaps from a backup proxy. The proxy can be a virtual machine or a physical machine with backup-restore software installed and access to alternate storage such as a tape autochanger or equivalent.

At a given point in time, during the backup window, backup software:

- 1 Snapshots virtual machines in a cluster, one by one, or in parallel. Virtual machines run off the snapshot.
- 2 Copies the quiesced base disk, or (for incremental backup) only changed blocks, to backup media.
- 3 Records the configuration of virtual machines.
- 4 Reverts and deletes snapshots, so virtual machines retain any changes made during the backup window.

In the above procedure, the virtual disk library is used in the second step only. The other steps use a portion of the vSphere API (called VADP) to snapshot and save configuration of virtual machines. The virtual disk in a cluster is "managed" by vSphere.

Managed Disk and Hosted Disk

Analogous to a hard disk drive, virtual disk files represent the storage volumes of a virtual machine. Each is named with .vmdk suffix. On a system running VMware Workstation, file systems of each guest OS are kept in VMDK files hosted on the system's physical disk. VMDK files can be accessed directly on the host.

With the virtual machine file system (VMFS) on ESXi hosts, VMDK files again represent storage volumes of virtual machines. They are on VMFS, which often resides on shared storage in a cluster. The vCenter Server manages the cluster storage so it can migrate (vMotion) virtual machines from one ESXi host to another without moving VMDK files. VMFS storage is therefore called managed disk.

VMFS disk can reside on a storage area network (SAN) attached to ESXi hosts by Fibre Channel, iSCSI, or SAS connectors. It can also reside on network attached storage (NAS), or on directly attached disk.

Figure 1-1 depicts the arrangement of managed disk (in this case VMDK on a SAN-hosted VMFS file system) and hosted disk (VMDK files on physical disk).

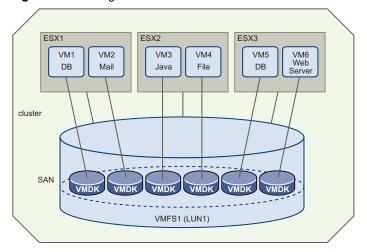
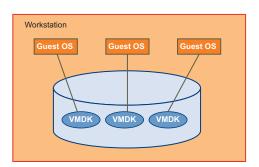


Figure 1-1. Managed Disk and Hosted Disk



The VDDK supports both managed disk and hosted disk, although some functions are not supported for managed disk, and other facilities are not supported for hosted disk. Exceptions are noted in documentation.

Managed virtual disk files larger than 2TB are supported by vSphere 5.5 and later.

Advanced Transports

With managed disk, VDDK applications can make use of advanced transports to perform many I/O operations directly on the SAN, rather than over the LAN. This improves performance and saves network bandwidth.

VDDK and VADP Compared

The Virtual Disk Development Kit (VDDK) includes a set of C library routines for manipulating virtual disk (VixDiskLib) and for mounting virtual disk partitions (VixMntapi). The VDDK focuses on efficient access and transfer of data on virtual disk storage.

The vSphere Storage APIs for Data Protection (VADP) is a marketing term for a subset of the vSphere API that enables backup and restore applications. The snapshot-based VADP framework allows efficient, off-host, centralized backup of virtual machine storage. After taking a snapshot to quiesce virtual disk, software can then back up storage using VDDK library routines.

The vSphere API is an XML-based Web service that provides the interfaces for vCenter Server management of virtual machines running on ESXi hosts.

Developers need both VDDK and VADP to write data protection software. VADP is presented in Chapter 7, "Backing Up Virtual Disks in vSphere," on page 57.

Platform Product Compatibility

To support a new release of vSphere, in most cases you should update and recompile your software with a corresponding new release of VDDK. This is because VDDK is continually updated to support new features in vSphere. As of 5.0, the version number of VDDK matches the version number of vSphere.

Since its inception in 2008, VDDK has been backward compatible with virtual platform products such as VMware Workstation, ESXi, and vCenter Server. VMware Fusion was never supported.

Redistributing VDDK Components

After you use the VDDK to develop software applications that run on VMware platform products, you might need to repackage library components that are compiled into your software.

To qualify for VDDK redistribution, you must be in the VMware TAP program at Select level or above, and sign a redistribution agreement. Contact your VMware alliance manager to request the VDDK redistribution agreement. VMware would like to know how you use the VDDK, in what products you plan to redistribute it, your company name, and your contact information.

Installing the Development Kit

To develop virtual disk applications, install the VDDK as described in this chapter. For backup applications, VADP development also requires the vSphere Web Services SDK.

- "Prerequisites" on page 15
- "Installing the VDDK Package" on page 16

Prerequisites

This section covers what you need to begin VDDK and VADP development.

Development Systems

The VDDK has been tested and is supported on the following systems:

- Windows 64-bit (x86-64) systems
- Linux 64-bit (x86-64) systems

See the VDDK Release Notes for specific versions, which change over time. Mac OS X is not supported.

Programming Environments

You can compile the sample program and develop vSphere applications in the following environments:

Visual Studio on Windows

On Windows, programmers can use the C++ compiler in Visual Studio 2005, Visual Studio 2008, and later. Along with Visual Studio, you also need to install the 64-bit debugging tools.

C++ and C on Linux

On Linux, programmers can use the GNU C compiler, version 4 and higher. The sample program compiles with the C++ compiler g++, but VDDK also works with the C compiler gcc.

Java Development for VADP

When developing backup and restore software to run on vSphere, VMware recommends Eclipse with Java, on both Windows and Linux. The vSphere Web Services SDK now includes both Axis and JAX-WS bindings. You can call C or C++ code with wrapper classes, as in Java Native Interface (JNI).

VMware Platform Products

Software applications developed with the VDDK and VADP target the following platform products:

- vCenter Server managing ESXi hosts
- ESXi hosts directly connected

See the VDDK Release Notes for specific versions, which change over time.

Hosted products including VMware Workstation are neither tested nor supported.

Storage Device Support

VMware Consolidated Backup (VCB) had knowledge base article http://kb.vmware.com/kb/1007479 showing the support matrix for storage devices and multipathing. VMware does not provide a similar support matrix for VDDK and VADP. Customers must get this information from you, their backup software vendor.

Installing the VDDK Package

The VDDK is packaged as a compressed archive for Windows 64-bit and for Linux 64-bit. The VDDK packages include the following components:

- Header files vixDiskLib.h and vm_basic_types.h in the include directory.
- Function library vixDiskLib.lib (Windows) or libvixDiskLib.so (Linux) in the lib directory.
- HTML reference documentation in the doc directory and sample program in doc/samples.



CAUTION In the VDDK 5.5 release, VMware has discontinued the Windows installer and 32-bit executables for Windows and Linux. The VDDK is delivered as a ZIP archive for 64-bit systems only.

To install the package on Windows

- On the Download page, choose the .zip file for Windows and download it to your development system.
- 2 Place the .zip file in a folder under Program Files you can choose the name and unpack it:

```
cd C:\Program Files\VMware\VDDK550
unzip VMware-vix-disklib-*.zip
```

- 3 Go to the bin subfolder, locate the vstor2install.bat script, and double-click to run it. The batch script should be run in place so that the current directory for execution is the bin subfolder. By running it, you implicitly accept the VMware license terms.
- 4 Edit the Windows registry with regedit and check for the following key. If this key exists from a previous VDDK install, right-click to delete it. Add a registry entry with the following key:

```
\label{thm:hammar} \begin{tabular}{ll} HKEY\_LOCAL\_MACHINE\SOFTWARE\Wow6432Node\VMware, Inc.\VMware\ Virtual\ Disk\ Development\ Kit\ Node \Local Color Col
```

For convenience you might want to edit the Windows Path environment to include the VDDK installation folder, C:\Program Files\VMware\VDDK550\bin in the example above.

To Install the package on Linux

- 1 On the Download page, choose the binary tar.gz for 64-bit Linux.
- 2 Unpack the archive with tar to create the vmware-vix-disklib-distrib subdirectory.

```
tar xvzf VMware-vix-disklib-*.tar.gz
```

3 Change to the newly created directory to see its subdirectories:

```
$ cd vmware-vix-disklib-distrib; ls
bin64 doc FILES include lib32 lib64
```

The bin64 subdirectory contains a reporting program and virtual disk manager. License terms are in the doc/EULA file. The sample program is under doc/samples. Header files in the include subdirectory are for compiling your VDDK applications. Normally the lib64 components are installed under /usr/lib,/usr/lib/vmware-vix-disklib/lib64 for instance.

4 Install components as needed. You might want to edit your LD_LIBRARY_PATH environment to include the library installation path. Alternatively, you can add the library location to the list in /etc/ld.so.conf and run ldconfig as the superuser.

Repackaging VDDK Libraries

After you develop an application based on VDDK, you might need the VDDK binaries to run your application.

As described in "Redistributing VDDK Components" on page 14, partners can sign a license agreement to redistribute VDDK binaries that support VADP applications.

To enable VDDK binaries on Windows virtual machines without VDDK installed

- 1 Install the Microsoft Visual C++ (MSVC) redistributable, possibly as a merge module. The latest MSVC runtime works as side-by-side component, so manually copying it might not work on Vista. See details on the Microsoft Web site for the redistributable package, x86 processors or x64 processors. Side-by-side is also explained on the Microsoft Web site.
- Install VMware executables and DLLs from the \bin and \lib folders of the installed VDDK, and the vstor2-mntapi10.sys driver into the Windows\system\drivers folder or equivalent.
- 3 Create and install your application, compiled in a manner similar to the vixDiskLibSample.exe code, discussed in Chapter 5, "Virtual Disk API Sample Code," on page 43.

How to Find VADP Components

ESXi hosts and vCenter Server similarly implement managed objects that support inventory traversal and task requests. Before you write VADP software in Java, you need to download the vSphere Web Services SDK. You can find documentation and ZIP file for download on the VMware Web site.

Virtual Disk Development Kit Programming Guide

Virtual Disk Interfaces

VMware offers many options for virtual disk layout, encapsulated in library data structures described here.

- "VMDK File Location" on page 19
- "Virtual Disk Types" on page 19
- "Data Structures in Virtual Disk API" on page 21
- "Virtual Disk Transport Methods" on page 23

VMDK File Location

On ESXi hosts, virtual machine disk (VMDK) files are usually located under one of the /vmfs/volumes, perhaps on shared storage. Storage volumes are visible from the vSphere Client, in the inventory for hosts and clusters. Typical names are datastore1 and datastore2. To see a VMDK file, click Summary > Resources > Datastore, right-click Browse Datastore, and select a virtual machine.

On Workstation, VMDK files are stored in the same directory with virtual machine configuration (VMX) files, for example /path/to/disk on Linux or C:\My Documents\My Virtual Machines on Windows.

VMDK files store data representing a virtual machine's hard disk drive. Almost the entire portion of a VMDK file is the virtual machine's data, with a small portion allotted to overhead.

Virtual Disk Types

The following disk types are defined in the virtual disk library:

- VIXDISKLIB_DISK_MONOLITHIC_SPARSE Growable virtual disk contained in a single virtual disk file.
 This is the default type for hosted disk, and the only setting in the Chapter 5 sample program.
- VIXDISKLIB_DISK_MONOLITHIC_FLAT Preallocated virtual disk contained in a single virtual disk file. This takes time to create and occupies a lot of space, but might perform better than sparse.
- VIXDISKLIB_DISK_SPLIT_SPARSE Growable virtual disk split into 2GB extents (s sequence). These files can to 2GB, then continue growing in a new extent. This type works on older file systems.
- VIXDISKLIB_DISK_SPLIT_FLAT Preallocated virtual disk split into 2GB extents (f sequence). These files start at 2GB, so they take a while to create, but available space can grow in 2GB increments.
- VIXDISKLIB_DISK_VMFS_FLAT Preallocated virtual disk compatible with ESX 3 and later. Also known as thick disk. This managed disk type is discussed in "Managed Disk and Hosted Disk" on page 13.
- VIXDISKLIB_DISK_VMFS_SPARSE Employs a copy-on-write (COW) mechanism to save storage space.
- VIXDISKLIB_DISK_VMFS_THIN Growable virtual disk that consumes only as much space as needed, compatible with ESX 3 or later, supported by VDDK 1.1 or later, and highly recommended.
- VIXDISKLIB_DISK_STREAM_OPTIMIZED Monolithic sparse format compressed for streaming. Stream optimized format does not support random reads or writes.

Persistence Disk Modes

In **persistent** disk mode, changes are immediately and permanently written to the virtual disk, so that they survive even through to the next power on.

In **nonpersistent** mode, changes to the virtual disk are discarded when the virtual machine powers off. The VMDK files revert to their original state.

The virtual disk library does not encapsulate this distinction, which is a virtual machine setting.

VMDK File Naming

Table 3-1 explains the different types of virtual disk. The first column corresponds to "Virtual Disk Types" on page 19 but without the VIXDISKLIB_DISK prefix. The third column gives the possible names of VMDK files as implemented on Workstation and ESXi hosts.

NOTE When you open a VMDK file with the virtual disk library, always open the one that points to the others, not the split or flat sectors. The file to open is most likely the one with the shortest name.

For information about other virtual machine files, see section "Files that Make Up a Virtual Machine" in the *VMware Workstation User's Manual*. On ESXi hosts, VMDK files are type VMFS_FLAT or VMFS_THIN.

Table 3-1. VMDK Virtual Disk Files

Disk Type in API	Virtual Disk Creation on VMware Host	Filename on Host
MONOLITHIC_SPARSE	In Select A Disk Type , accepting the defaults by not checking any box produces one VMDK file that can grow larger if more space is needed. The <vmname> represents the name of a virtual machine.</vmname>	<vmname>.vmdk</vmname>
	On VMFS partitions, this is name of the disk descriptor file.	
MONOLITHIC_FLAT or VMFS_FLAT or VMFS_THIN	If you select only the Allocate all disk space now check box, space is pre-allocated, so the virtual disk cannot grow. The first VMDK file is small and points to a much larger one, whose filename says flat without a sequence number.	<vnname>-flat.vmdk</vnname>
	Similarly on VMFS partitions, this is the virtual disk file that points to virtual disk data files, either thick or thin provisioned.	
SPLIT_SPARSE	If you select only the Split disk into 2GB files check box, virtual disk can grow when more space is needed. The first VMDK file is small and points to a sequence of other VMDK files, all of which have an s before a sequence number, meaning sparse. The number of VMDK files depends on the disk size requested. As data grows, more VMDK files are added in sequence.	<vmname>-s<###>.vmdk</vmname>
SPLIT_FLAT	If you select the Allocate all disk space now and Split disk into 2GB files check boxes, space is pre-allocated, so the virtual disk cannot grow. The first VMDK file is small and points to a sequence of other files, all of which have an f before the sequence number, meaning flat. The number of files depends on the requested size.	<vnname>-f<###>.vmdk</vnname>
MONOLITHIC_SPARSE or SPLIT_SPARSE is taken of a virtual machine, or with the virtual disk library. Snapshot Snapshot is taken of a virtual machine, or with the virtual disk library. Snapshot file numbers are in sequence, without an s or f prefix. The numbered VMDK file stores changes made to the virtual disk <diskname> since the original parent disk, or previously numbered redo log (in other words the previous snapshot).</diskname>		<diskname>-<###>.vmdk</diskname>
SE_SPARSE	Space-efficient sparse (seSparse) format. In vSphere 5.1 and later, used by VMware View to optimize linked clone templates. In the vSphere API, see data object SeSparseVirtualDiskSpec. Use of seSparse as a base disk is neither documented nor supported.	
n/a	Snapshot of a virtual machine, which includes pointers to all its .vmdk virtual disk files.	<vnname>Snapshot.vmsn</vnname>

For lazy zeroed thick disk, all blocks are allocated, and data written to used blocks, however unused blocks are left as-is, so they may contain data from previous use. Many storage systems will zero-out unused blocks in the background. With eager zeroed thick disk, unused blocks are zeroed-out at allocation time.

Thin Provisioned Disk

With thin provisioned disk, the vSphere Client may report that provisioned size is greater than disk capacity.

Provisioned size for a thin disk is the maximum size the disk will occupy when fully allocated. Actual size is the current size of the thin disk. Overcommit means that if all thin disks were fully provisioned, there would not be enough space to accommodate all of the thin disks.

Internationalization and Localization

VDDK libraries are not localized, but backup partners can support any locale that uses UTF-8 encoding. Other than Unicode, VDDK does not support multibyte character encodings such as Extended Unix Code (EUC) for Chinese, Japanese, and Korean (ISO-2022).

The path name to a virtual machine and its VMDK can be expressed with any character set supported by the host file system. As of vSphere 4 and Workstation 7, VMware supports Unicode UTF-8 path names, although for portability to various locales, ASCII-only path names are recommended.

Windows 2000 systems (and later) use UTF-16 for localized path names. For example, in locale FR (Français) the VDDK sample code might mount disk at C:\Windows\Temp\vmware-Système, where è is encoded as UTF-16 so the VixMntapi library cannot recognize it. In this case, a workaround is to set the tmpDirectory configuration key with an ASCII-only path before program start-up; see "Initialize the Library" on page 29.

For programs opening arbitrary path names, Unicode offers a GNU library with C functions iconv_open() to initialize codeset conversion, and iconv() to convert UTF-8 to UTF-16, or UTF-16 to UTF-8.

Virtual Disk Internal Format

The *Virtual Disk Format 5.0* technical note provides possibly useful information about the VMDK format, and is available at this URL:

http://www.vmware.com/support/developer/vddk/vmdk_50_technote.pdf

Grain Directories and Grain Tables

SPARSE type virtual disks use a hierarchical representation to organize sectors. See *Virtual Disk Format* 5.0 referenced in "Virtual Disk Internal Format" on page 21. In this context, grain means granular unit of data, larger than a sector. The hierarchy includes:

- Grain directory (and redundant grain directory) whose entries point to grain tables.
 - Grain tables (and redundant grain tables) whose entries point to grains.
 - Each grain is a block of sectors containing virtual disk data. Default size is 128 sectors or 64KB.

Data Structures in Virtual Disk API

Here are important data structure objects with brief descriptions:

■ VixError – Error code of type uint64.

VixDiskLibConnectParams – Public types designate the virtual machine credentials vmxSpec (possibly through vCenter Server), the name of its host, and the credential type for authentication. For details, see "VMX Specification" on page 29. The credType can be VIXDISKLIB_CRED_UID (user name / password), VIXDISKLIB_CRED_SESSIONID (the HTTP session ID), VIXDISKLIB_CRED_TICKETID (vSphere ticket ID), or VIXDISKLIB_CRED_SSPI (Windows only, current thread credentials).

```
typedef char * vmxSpec
typedef char * serverName
typedef VixDiskLibCredType credType
```

■ VixDiskLibConnectParams::VixDiskLibCreds - Credentials for either user ID or session ID.

VixDiskLibConnectParams::VixDiskLibCreds::VixDiskLibUidPasswdCreds – String data fields represent user name and password for authentication.

VixDiskLibConnectParams::VixDiskLibCreds::VixDiskLibSessionIdCreds – String data fields represent the session cookie, user name, and encrypted session key.

VixDiskLibConnectParams::VixDiskLibCredsb::VixDiskLibSSPICreds – String data fields represent security support provider interface (SSPI) authentication. User name and password are null.

■ VixDiskLibCreateParams – Types represent the virtual disk (see "Virtual Disk Types" on page 19), the disk adapter (see "Adapter Types" on page 23), VMware version, and capacity of the disk sector.

```
typedef VixDiskLibDiskType diskType
typedef VixDiskLibAdapterType adapterType
typedef uint hwVersion
typedef VixDiskLibSectorType capacity
```

■ VixDiskLibDiskInfo – Types represent the geometry in the BIOS and physical disk, the capacity of the disk sector, the disk adapter (see "Adapter Types" on page 23), the number of child-disk links (redo logs), and a string to help locate the parent disk (state before redo logs).

```
VixDiskLibGeometry biosGeo
VixDiskLibGeometry physGeo
VixDiskLibSectorType capacity
VixDiskLibAdapterType adapterType
int numLinks
char * parentFileNameHint
```

■ VixDiskLibGeometry – Types specify virtual disk geometry, not necessarily the same as physical disk.

```
typedef uint32 cylinders
typedef uint32 heads
typedef uint32 sectors
```

Credentials and Privileges for VMDK Access

Local operations are supported by local VMDK. Access to ESXi hosts is authenticated by login credentials, so with proper credentials VixDiskLib can reach any VMDK on an ESXi host. VMware vSphere has its own set of privileges, so with the proper privileges (see below) and login credentials, VixDiskLib can reach any VMDK on an ESXi host managed by vCenter Server. VixDiskLib supports the following:

- Both read-only and read/write modes
- Read-only access to disk associated with any snapshot of online virtual machines
- Access to VMDK files of offline virtual machines (vCenter restricted to registered virtual machines)
- Reading of Microsoft Virtual Hard Disk (VHD) format

With vCenter Server, the Role of the backup appliance when saving data must have these privileges for all the virtual machines being backed up:

- VirtualMachine > Configuration > Disk change tracking
- VirtualMachine > Provisioning > Allow read-only disk access and Allow VM download
- VirtualMachine > State > Create snapshot and Remove snapshot

On the backup appliance, the user must have the following privileges:

- Datastore > Allocate space
- VirtualMachine > Configuration > Add new disk and Remove disk
- VirtualMachine > Configuration > Change resource and Settings

The user must have these privileges for vCenter Server and all ESXi hosts involved in backup:

- Global > DisableMethods and EnableMethods
- Global > License

All privileges must be applied at the vCenter Server level. Otherwise the error message returned will be somewhat misleading: "The host is not licensed for this feature."

Adapter Types

The library can select the following adapters:

- VIXDISKLIB_ADAPTER_IDE Virtual disk acts like ATA, ATAPI, PATA, SATA, and so on. You might select this adapter type when it is specifically required by legacy software.
- VIXDISKLIB_ADAPTER_SCSI_BUSLOGIC Virtual SCSI disk with Buslogic adapter. This is the default on some platforms and is usually recommended over IDE due to higher performance.
- VIXDISKLIB_ADAPTER_SCSI_LSILOGIC Virtual SCSI disk with LSI Logic adapter. Windows Server 2003 and most Linux virtual machines use this type by default. Performance is about the same as Buslogic.

Virtual Disk Transport Methods

VMware supports file-based or image-level backups of virtual machines hosted on an ESXi host with SAN or NAS storage. Virtual machines read data directly from a shared VMFS LUN, so backups are efficient and do not put significant load on production ESXi hosts or the virtual network.

VMware offers interfaces for integration of storage-aware applications, including backup, with efficient access to storage clusters. Developers can use VDDK advanced transports, which provide efficient I/O methods to maximize backup performance. VMware supports four access methods: local file, NBD (network block device) over LAN with SSL encryption (NBDSSL), SAN, and SCSI HotAdd.

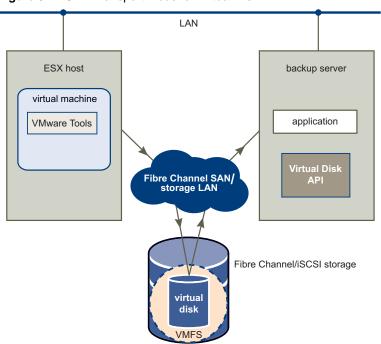
Local File Access

The virtual disk library can read virtual disk data from /vmfs/volumes on ESXi hosts, or from the local file system on hosted products. This file access method is built into VixDiskLib, so it is always available on local storage. However it is not a network transport method, and is seldom used for vSphere backup.

SAN Transport

SAN mode requires applications to run on a backup server with access to SAN storage (Fibre Channel, iSCSI, or SAS connected) containing the virtual disks to be accessed. As shown in Figure 3-1, this method is efficient because no data needs to be transferred through the production ESXi host. A SAN backup proxy must be a physical machine. If it has optical media or tape drive connected, backups can be made entirely LAN-free.

Figure 3-1. SAN Transport Mode for Virtual Disk



In SAN transport mode, the virtual disk library obtains information from an ESXi host about the layout of VMFS LUNs, and using this information, reads data directly from the storage LUN where a virtual disk resides. This is the fastest transport method for software deployed on SAN-connected ESXi hosts.

In general, SAN transport works with any storage device that appears at the driver level as a LUN (as opposed to a file system such as NTFS or EXT). SAN mode must be able to access the LUN as a raw device. The real key is whether the device behaves like a direct raw connection to the underlying LUN. SAN transport is supported in Fibre Channel, iSCSI, and SAS based storage arrays (SAS means serial attached SCSI). SAN storage devices can contain SATA drives, but currently there are no SATA connected SAN devices on the VMware hardware compatibility list.

SAN transport is not supported for backup or restore of virtual machines residing on VVol datastores.

VMware Virtual SAN (VSAN), a network based storage solution with direct attached disks, does not support SAN transport. Because VSAN uses modes that are incompatible with SAN transport, if the virtual disk library detects the presence of vSAN, it disables SAN mode. Other advanced transports do work.

HotAdd Transport

HotAdd is a VMware feature where devices can be added "hot" while a virtual machine is running. Besides SCSI disk, virtual machines can add additional CPUs and memory capacity.

If backup software runs in a virtual appliance, it can take a snapshot and create a linked clone of the target virtual machine, then attach and read the linked clone's virtual disks for backup. This involves a SCSI HotAdd on the ESXi host where the target VM and backup proxy are running. Virtual disks of the linked clone are HotAdded to the backup proxy. The target virtual machine continues to run during backup.

VixTransport handles the temporary linked clone and hot attachment of virtual disks. VixDiskLib opens and reads the HotAdded disks as a "whole disk" VMDK (virtual disk on the local host). This strategy works only on virtual machines with SCSI disks and is not supported for backing up virtual IDE disks. HotAdd transport also works with virtual machines stored on NFS partitions.

HotAdd is a good way to get virtual disk data from a virtual machine to a backup appliance (or backup proxy) for sending to the media server. The attached HotAdd disk is shown in Figure 3-2.

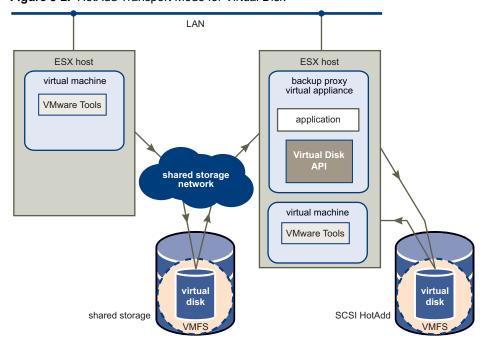


Figure 3-2. HotAdd Transport Mode for Virtual Disk

Running the backup proxy as a virtual machine has two advantages: it is easy to move a virtual machine to a new media server, and it can back up local storage without using the LAN, although this incurs more overhead on the physical ESXi host than when using SAN transport mode.

About the HotAdd Proxy

The HotAdd backup proxy must be a virtual machine. HotAdd involves attaching a virtual disk to the backup proxy, like attaching disk to a virtual machine. In typical implementations, a HotAdd proxy backs up either Windows or Linux virtual machines, but not both. For parallel backup, sites can deploy multiple proxies.

The HotAdd proxy must have access to the same datastore as the target virtual machine, and the VMFS version and data block sizes for the target VM must be the same as the datastore where the HotAdd proxy resides.

If the HotAdd proxy is a virtual machine that resides on a VMFS-3 volume, choose a volume with block size appropriate for the maximum virtual disk size of virtual machines that customers want to back up, as shown in Table 3-2. This caveat does not apply to VMFS-5 volumes, which always have 1MB file block size.

Table 3-2. VMFS-3 Block Size for HotAdd Backup Proxy

VMFS Block Size	Maximum Target Disk Size
1MB	256GB
2MB	512GB
4MB	1024GB
8MB	2048GB

NBDSSL Transport

When no other transport is available, networked storage applications can use LAN transport for data access, with NBD (network block device) protocol and SSL encryption, called NBDSSL. NBD is a Linux-style kernel module that treats storage on a remote host as a block device. NBDSSL is the VMware variant that uses SSL to encrypt all data passed over the TCP connection. The NBDSSL transport method is built into the virtual disk library, so it is always available, and is the fall-back when no other transport method is available.

VMware libraries, and backup applications, often fall back to NBDSSL when other transports are unavailable.

ESX host

virtual machine

VMware Tools

application

Virtual Disk
API

Figure 3-3. LAN (NBDSSL) Transport Mode for Virtual Disk

In this mode, the ESXi host reads data from storage and sends it across a network to the backup server. With LAN transport, large virtual disks can take a long time to transmit. This transport mode adds traffic to the LAN, unlike SAN and HotAdd transport, but NBDSSL transport offers the following advantages:

- The ESXi host can use any storage device, including local storage or remote-mounted NAS.
- The backup proxy can be a virtual machine, so customers can use vSphere resource pools to minimize the performance impact of backup. For example, the backup proxy can be in a lower-priority resource pool than the production ESXi hosts.

When VDDK opens a non-snapshot disk for NBDSSL transfer (read-only or read/write) it selects the ESXi host where the disk's virtual machine currently resides.

However when VDDK opens a snapshot for NBDSSL transfer, the common backup case, VDDK passes the datastore to vCenter Server, which consults its list of ESXi hosts with access to the datastore; vCenter picks the first host with read/write access. The list of hosts is unordered, so the host chosen for NBDSSL transfer of the snapshot is not necessarily the ESXi host where the snapshot's virtual machine resides.

NBDSSL Performance When reading disk data using NBDSSL transport, VDDK makes synchronous calls. That is, VDDK requests a block of data and waits for a response. The block is read from disk and copied into a buffer on the server side, then sent over the network. Meanwhile, no data gets copied over the network, adding to wait time. To some extent, you can overcome this limitation by using multiple streams to simultaneously read from a single disk or multiple disks, taking advantage of parallelism.

As of vSphere 6.5, NBDSSL performance can be significantly improved using data compression. Three types of compression are available – zlib, fastlz, and skipz – specified as flags when opening virtual disks with the VixDiskLib_Open() call. See "Open a Local or Remote Disk" on page 30.

NFC Session Limits

NBDSSL employs the network file copy (NFC) protocol. Table 3-3 shows limits on the number of connections for various host types. These are host limits, not per process limits. Additionally vCenter Server imposes a limit of 52 connections. VixDiskLib_Open() uses one connection for every virtual disk that it accesses on an ESXi host. Clone with VixDiskLib_Clone() also requires a connection. It is not possible to share a connection across physical disks. These NFC session limits do not apply to SAN or HotAdd transport.

Table 3-3. NFC Session Connection Limits

Host Platform	When Connecting	Limits You To About
vSphere 4	to an ESX host	9 connections directly, 27 connections through vCenter Server
vSphere 4	to an ESXi host	11 connections directly, 23 connections through vCenter Server
vSphere 5 and vSphere 6	to an ESXi host	Limited by a transfer buffer for all NFC connections, enforced by the host; the sum of all NFC connection buffers to an ESXi host cannot exceed 32MB. 52 connections through vCenter Server, including the above per-host limit.

SSL Certificates and Security

The VDDK 5.1 release and later were security hardened, with virtual machines set to check SSL certificates.

On Windows VDDK 5.1 and 5.5 required the VerifySSLCertificates and InstallPath registry keys under HKEY_LOCAL_MACHINE\SOFTWARE to check SSL certificates. On Linux VDDK 5.1 and 5.5 required adding a line to the VixDiskLib_InitEx configuration file to set linuxSSL.verifyCertificates = 1.

As of VDDK 6.0 both SSL certificate verification and SSL thumbprint checking are mandatory and cannot be disabled. The Windows registry and Linux SSL setting are no longer checked, so neither has any effect.

Specifically VDDK 6.0 and later use X.509 certificates with TLS cryptography, replacing SSLv3.

The following library functions enforce SSL certificate checking: InitEx, PrepareForAccess, EndAccess, GetNfcTicket, and the GetRpcConnection interface that is used by all advanced transports. SSL verification may use thumbprints to check if two certificates are the same. The vSphere thumbprint is a cryptographic hash of a certificate obtained from a trusted source such as vCenter Server, and passed in the SSLVerifyParam structure of the NFC ticket.

Virtual Disk API Functions

This chapter provides an overview of functions in the Virtual Disk API and includes the following sections:

- "Virtual Disk Library Functions" on page 27
- "Start Up" on page 28
- "Disk Operations" on page 30
- "Metadata Handling" on page 31
- "Cloning a Virtual Disk" on page 34
- "Disk Chaining and Redo Logs" on page 32
- "Error Handling" on page 34
- "Administrative Disk Operations" on page 34
- "Shut Down" on page 35
- "Advanced Transport APIs" on page 35
- "Updating Applications for Advanced Transport" on page 40
- "Multithreading Considerations" on page 42
- "Capabilities of Library Calls" on page 42

After a presentation of Virtual Disk API functions in alphabetic order, sections focus on what the functions do, in the normal order they would appear in a program, except advanced transport functions (SAN and HotAdd) appear after the shutdown functions.

Virtual Disk Library Functions

You can find the *VixDiskLib API Reference* by using a Web browser to open the doc/index.html file in the VDDK software distribution. As in most reference manuals, functions are organized alphabetically, whereas in this chapter, functions are ordered by how they might be called.

When the API reference says that a function supports "only hosted disks," it means virtual disk images hosted by VMware Workstation or similar products. Virtual disk stored on VMFS partitions managed by ESXi or vCenter Server is called "managed disk."

The functions described in this chapter are based on concepts and employ data structures documented in Chapter 3, "Virtual Disk Interfaces," on page 19.

If the library accesses virtual disk on VMFS, I/O by default goes through the ESXi host, which manages physical disk storage. To use function calls that provide direct access to SAN storage, start your program by calling the VixDiskLib_ConnectEx() function, as described in "Advanced Transport APIs" on page 35.

Alphabetic Table of Functions

Function calls in the Virtual Disk API are listed alphabetically in Table 4-1.

Table 4-1. Virtual Disk API Functions

Function	Description
VixDiskLib_Attach	Attaches the child disk chain to the parent disk chain.
VixDiskLib_Cleanup	Removes leftover transports. See "Clean Up After Disconnect" on page 40.
VixDiskLib_Clone	Copies virtual disk to some destination, converting formats as appropriate.
VixDiskLib_Close	Closes an open virtual disk. See "Close a Local or Remote Disk" on page 30.
VixDiskLib_Connect	Connects to the virtual disk library to obtain services. See also ConnectEx.
VixDiskLib_ConnectEx	Connects to optimum transport. See "Connect to VMware vSphere" on page 38
VixDiskLib_Create	Creates a virtual disk according to specified parameters.
VixDiskLib_CreateChild	Creates a child disk (redo log or delta link) for a hosted virtual disk.
VixDiskLib_Defragment	Defragments the sectors of a virtual disk.
VixDiskLib_Disconnect	Disconnects from the library. See "Disconnect from Server" on page 35.
VixDiskLib_EndAccess	Notifies a host that it may again relocate a virtual machine. See page 39.
VixDiskLib_Exit	Releases all resources held by the library. See "Clean Up and Exit" on page 35.
VixDiskLib_FreeErrorText	Frees the message buffer allocated by GetErrorText.
VixDiskLib_FreeInfo	Frees the memory allocated by GetInfo.
VixDiskLib_GetErrorText	Returns the text description of a library error code.
VixDiskLib_GetInfo	Retrieves information about a virtual disk.
VixDiskLib_GetMetadataKeys	Retrieves all keys in the metadata of a virtual disk.
VixDiskLib_GetTransportMode	Gets current transport mode. See "Get Selected Transport Method" on page 39.
VixDiskLib_Grow	Increases size of an existing virtual disk.
VixDiskLib_Init	Initializes the old virtual disk library. Replaced by InitEx function.
VixDiskLib_InitEx	Initializes new virtual disk library. See "Initialize Virtual Disk API" on page 36.
VixDiskLib_ListTransportModes	Available transport modes. See "List Available Transport Methods" on page 38.
VixDiskLib_Open	Opens a virtual disk. See "Open a Local or Remote Disk" on page 30.
VixDiskLib_PrepareForAccess	Notifies a host to refrain from relocating a virtual machine. See page 39.
VixDiskLib_Read	Reads from an open virtual disk. See "Read Sectors From a Disk" on page 30.
VixDiskLib_ReadMetadata	Retrieves the value of a given key from disk metadata.
VixDiskLib_Rename	Changes the name of a virtual disk.
VixDiskLib_Shrink	Reclaims blocks of zeroes from the virtual disk.
VixDiskLib_SpaceNeededForClone	Computes the space required to clone a virtual disk, in bytes.
VixDiskLib_Unlink	Deletes the specified virtual disk.
VixDiskLib_Write	Writes to an open virtual disk. See "Write Sectors To a Disk" on page 30.

Start Up

The VixDiskLib_Init() and VixDiskLib_Connect() functions must appear in all virtual disk programs. VixDiskLib_Init() has been superseded by VixDiskLib_InitEx(). See "Initialize Virtual Disk API" on page 36.

Initialize the Library

VixDiskLib_Init() initializes the old virtual disk library. The arguments majorVersion and minorVersion represent the VDDK library's release number and dot-release number. The optional third, fourth, and fifth arguments specify log, warning, and panic handlers. DLLs and shared objects may be located in libDir.

VixError vixError = VixDiskLib_Init(majorVer, minorVer, &logFunc, &warnFunc, &panicFunc, libDir);

You should call VixDiskLib_Init() only once per process because of internationalization restrictions, at the beginning of your program. You should call VixDiskLib_Exit() at the end of your program for cleanup. For multithreaded programs you should write your own logFunc because the default function is not thread safe.

In most cases you should replace VixDiskLib_Init() with VixDiskLib_InitEx(), which allows you to specify a configuration file. For information about InitEx, see "Initialize Virtual Disk API" on page 36.

Connect to a Workstation or Server

VixDiskLib_Connect() connects the library to either a local VMware host or a remote server. For hosted disk on the local system, provide null values for most connection parameters. For managed disk on vSphere, specify virtual machine name, vCenter Server or ESXi host, user name, password, and port number.

vixError = VixDiskLib_Connect(&cnxParams, &srcConnection)

Always call VixDiskLib_Disconnect() before the end of your program.

Calling vixDiskLib_Disconnect() invalidates any open file handles, so if you use the VixMntapi library, call VixMntApi_CloseDiskSet() before calling disconnect.

You can opt to use the VixDiskLibSSPICreds connection parameter to enable Security Support Provider Interface (SSPI) authentication. SSPI has the advantage of not storing plain text passwords in configuration files or in the registry. In order to be able to use SSPI, the following conditions must be met:

- Connections must be made directly to a vCenter Server instance.
- Applications and their connections must employ one of two user account arrangements. The connection must be established either using the same user context with the same user name and password credentials on both the proxy and the vSphere Server, or using a domain user. Attempts by applications to establish connections using the Local System account context will fail.
- User contexts must have administrator privileges on the proxy and have VCB Backup User role assigned in vSphere (by the vCenter Server).

If your setup meets all these conditions, you can enable SSPI authentication by setting USERNAME to __sspi__. For SSPI, the password must be set, but it is ignored. It can be set to "" (null string).

VMX Specification

On VMware platform products, .vmx is a text file (usually located in the same directory as virtual disk files) specifying virtual machine configuration. The Virtual Machine eXecutable (VMX) process is the user-space component (or "world") of a virtual machine. The virtual disk library connects to virtual machine storage through the VMX process.

When specifying connection parameters (see "Data Structures in Virtual Disk API" on page 21) the preferred syntax for vmxSpec is as follows:

Managed object reference of the virtual machine, an opaque object that you obtain programmatically using the PropertyCollector managed object:

```
moRef=<moref-of-vm>
```

The moRef of a virtual machine or disk snapshot on an ESXi host is likely different than the moRef of the same virtual machine or disk snapshot as managed by vCenter Server. Here are two example moRef specifications, one for ESXi and one for vCenter Server, both referring to the same snapshot:

moref=153

Disk Operations

These functions create, open, read, write, query, and close virtual disk.

Create a New Hosted Disk

VixDiskLib_Create() locally creates a new virtual disk, after being connected to the host. In createParams, you must specify the disk type, adapter, hardware version, and capacity as a number of sectors. This function supports hosted disk. For managed disk, first create a hosted type virtual disk, then use VixDiskLib_Clone() to convert the virtual disk to managed disk.

```
vixError =
```

VixDiskLib_Create(appGlobals.connection, appGlobals.diskPath, &createParams, NULL, NULL);

Currently VixDiskLib_Create() enforces a 4GB limit for virtual disks on FAT32 and FAT file systems, a 16TB - 54KB (hex FFFFFFF0000) limit on NTFS file systems, and a 2^64 - 1 limit (more than an exabyte) on ReFS and exFAT file systems. Hosted virtual disk > 2TB is not supported.

POSIX based file systems including NFS version 3 no longer have a 2GB file size limit. Although various checks are done to avoid creating impossibly large files, it becomes the customer's responsibility to cope with 2GB limits on NFS version 2 or Linux kernel 2.4 (EFS).

Open a Local or Remote Disk

After the library connects to a workstation or server, VixDiskLib_Open() opens a virtual disk. With SAN or HotAdd transport, opening a remote disk for writing requires a pre-existing snapshot.

```
vixError =
```

VixDiskLib_Open(appGlobals.connection, appGlobals.diskPath, appGlobals.openFlags, &srcHandle);

The following flags modify the open instruction:

- VIXDISKLIB_FLAG_OPEN_UNBUFFERED Disable host disk caching.
- VIXDISKLIB_FLAG_OPEN_SINGLE_LINK Open the current link, not the entire chain (hosted disk only).
- VIXDISKLIB_FLAG_OPEN_READ_ONLY Open the virtual disk read-only.

As of vSphere 6.5, the following additional flags are available:

- VIXDISKLIB_FLAG_OPEN_COMPRESSION_ZLIB Open for NBDSSL transport, zlib compression.
- VIXDISKLIB_FLAG_OPEN_COMPRESSION_FASTLZ Open for NBDSSL transport, fastlz compression.
- VIXDISKLIB_FLAG_OPEN_COMPRESSION_SKIPZ Open for NBDSSL transport, skipz compression.

Read Sectors From a Disk

VixDiskLib_Read() reads a range of sectors from an open virtual disk. You specify the beginning sector and the number of sectors. Sector size could vary, but is defined in <vixDiskLib.h> as 512 bytes because VMDK files have that sector size.

```
vixError = VixDiskLib_Read(srcHandle, i, j, buf);
```

Write Sectors To a Disk

VixDiskLib_Write() writes one or more sectors to an open virtual disk. This function expects the fourth parameter buf to be VIXDISKLIB_SECTOR_SIZE bytes long.

```
vixError = VixDiskLib_Write(newDisk.Handle(), i, j, buf);
```

Close a Local or Remote Disk

VixDiskLib_Close() closes an open virtual disk.

VixDiskLib_Close(srcHandle);

VixDiskLib_Close() returns VIX_0K if successful, otherwise a suitable error code. To obtain a list of possible return codes, see "Finding Error Code Documentation" on page 115.

If a program has worker threads called from a master process, errors might occur in the threads after the master process calls VixDiskLib_Close(). Always wait for worker threads to end before calling close.

Get Information About a Disk

```
vixError = VixDiskLib_GetInfo(srcHandle, diskInfo);
```

VixDiskLib_GetInfo() gets data about an open virtual disk, allocating a filled-in VixDiskLibDiskInfo structure. Some of this information overlaps with metadata (see "Metadata Handling" on page 31).

Free Memory from Get Information

This function deallocates memory allocated by VixDiskLib_GetInfo(). Call it to avoid a memory leak. vixError = VixDiskLib_FreeInfo(diskInfo);

Metadata Handling

VMware provides mechanisms for reading, writing, and repairing virtual disk metadata.

Read Metadata Key from Disk

```
vixError = VixDiskLib_ReadMetadata(disk.Handle(), appGlobals.metaKey, &val[0], requiredLen, NULL);
```

Retrieves the value of a given key from disk metadata. The metadata for a hosted VMDK is not as extensive as for managed disk on an ESXi host. Held in a mapping file, VMFS metadata might also contain information such as disk label, LUN or partition layout, number of links, file attributes, locks, and so forth. Metadata also describes encapsulation of raw disk mapping (RDM) storage, if applicable.

Get Metadata Table from Disk

VixDiskLib_GetMetadataKeys() retrieves all existing keys from the metadata of a virtual disk, but not the key values. Use this in conjunction with VixDiskLib_ReadMetadata().

```
vixError = VixDiskLib_GetMetadataKeys(disk.Handle(), &buf[0], requiredLen, NULL);
```

Here is an example of a simple metadata table. Uuid is the universally unique identifier for the virtual disk.

```
adapterType = buslogic
geometry.sectors = 32
geometry.heads = 64
geometry.cylinders = 100
uuid = 60 00 C2 93 7b a0 3a 03-9f 22 56 c5 29 93 b7 27
```

Write Metadata Table to Disk

VixDiskLib_WriteMetadata() updates virtual disk metadata with the given key-value pair. If the key-value pair is new, it gets added. If the key already exists, its value is updated. A key can be zeroed but not deleted.

```
vixError = VixDiskLib_WriteMetadata(disk.Handle(), appGlobals.metaKey, appGlobals.metaVal);
```

Check and Repair Sparse Disk Metadata

VixDiskLib_CheckRepair() checks the metadata of a sparse disk, specified as a file on a connection, and optionally repairs the metadata if necessary. Sparse disks occupy space on a datastore only when a portion of the disk is used; the metadata tracks which portions are allocated.

```
vixError = VixDiskLib_CheckRepair(appGlobals.connection, appGlobals.diskPath, TRUE);
```

Disk Chaining and Redo Logs

In VMDK terminology, all the following are synonyms: child disk, redo log, and delta link. From the original parent disk, each child constitutes a redo log pointing back from the present state of the virtual disk, one step at a time, to the original. This pseudo equation represents the relative complexity of backups and snapshots:

backup image < child disk = redo log = delta link < snapshot

A backup image is less than a child disk because a backup image is merely a data stream. A snapshot is more than a child disk because it contains virtual machine state, with pointers to associated file states on VMDK.

There exist other types of redo logs, such as those that perform progressive protection in vSphere Replication (VR). For disk chaining, the "redo" terminology is especially appropriate for the snapshot revert operation, when changed blocks in the redo log are applied to the base disk, before deleting the redo log. Afterwards the base disk contains a "redo" of all changes that the virtual machine made while the snapshot was active.

Create Child from Parent Disk

Usually you create the first child disk from the parent and create successive children from the latest one in the chain. The disk tracks, in SPARSE format, any disk sectors changed since inception, as illustrated in Figure 4-1.

Child3 vm.vmdk vm-001.vmdk
Changed Sectors Only
Child2 vm.vmdk vm-002.vmdk

Child1 vm.vmdk vm-001.vmdk

Parent vm.vmdk vm-flat.vmdk

Figure 4-1. Child Disks Created from Parent

VixDiskLib_CreateChild() creates a child disk (or redo log) for a hosted virtual disk. After you create a child, it is generally not necessary to open the parent, or earlier children in the disk chain. The children's vm.vmdk files point to redo logs, not to the parent disk, vm-flat.vmdk in this example. To access the original parent, or earlier children in the chain, you can use VixDiskLib_Attach() on hosted disk.

Attach Child to Parent Disk

VixDiskLib_Attach() attaches the child disk into its parent disk chain. Afterwards, the parent handle is invalid and the child handle represents the combined disk chain of redo logs.

vixError = VixDiskLib_Attach(parent.Handle(), child.Handle());

For example, suppose you want to access the older disk image recorded by Child1. Attach the handle of new Child1a to Child1, which provides Child1a's parent handle, as shown in Figure 4-2. It is now permissible to open, read, and write the Child1a virtual disk.

The parent-child disk chain is efficient in terms of storage space, because the child VMDK records only the sectors that changed since the last VixDiskLib_CreateChild(). The parent-child disk chain also provides a redo mechanism, permitting programmatic access to any generation with VixDiskLib_Attach().

Virtual Machine Writes Here

Child3 vm.vmdk vm-001.vmdk

Child2 vm.vmdk vm-002.vmdk

Child1 vm.vmdk vm-001.vmdk

Parent vm.vmdk

Figure 4-2. Child Disks Created from Parent

Opening in a Chain

With (parent) base disk B and children C0, C1, and C2, opening C2 gives you the contents of B + C0 + C1 + C2 (not really addition linked data sectors), while opening C1 gives you the contents of B + C0 + C1.

Physical Disk

A better solution than recording base disks and which children are descended from which is changed block tracking, QueryChangedDiskAreas in the vSphere API. See "Algorithm for vSphere Backup" on page 40.

Redo Logs and Linked Clone Backup

For managed virtual disk on vSphere, snapshots are used primarily for saving system state and for backup, while linked clones create duplicate images for provisioning of View desktops. A snapshot is usually a single redo log in a parent-child chain, while linked clones are usually multiple redo logs based on the same parent.

In the vSphere 5.5 release and later, handling of linked clone hierarchies was changed to improve the efficiency of backup and restore. The disk object now contain a "disk backing" that contains one or more parent backing objects until the base disk is reached. This allows access anywhere in the parent-child disk chain. With a clean never-used base virtual machine, the linked clone hierarchy or snapshot chain always has the proper number of parent backing objects for the nodes in the chain.

VDDK does not contain any convenience interfaces for backing up and restoring the linked clone hierarchy (or the snapshot chain). Backup applications are responsible for discovering and saving the hierarchy if they want to support this as a feature. Linked clones cannot be restored using SAN transport.

In VMware View (VDI) environments, linked clone backup might not be necessary or advisable, especially for nonpersistent desktops that revert to default after use.

When the base disk or a child disk has an extra snapshot, when redo logs used to create linked clones were never deleted, or when any parent or child in the chain needs disk consolidation or is in a bad snapshot state, it is possible to have extra (too many) parent backing objects. Consequently, restore applications should never assume the correct number of parent backing objects. They must recursively query until the base parent backing object is reached, and make sure when restoring leaf nodes that the correct parent backing object matches the node being restored.

Cloning a Virtual Disk

Compute Space Needed for Clone

This function computes the space required (in bytes) to clone a virtual disk, after possible format conversion.

vixError = VixDiskLib_SpaceNeededForClone(child.Handle(), VIXDISKLIB_DISK_VMFS_FLAT, &spaceReq);



VixDiskLib_SpaceNeededForClone() might not give accurate results, or could return VIX_E_INVALID_ARG, when used with thin provisioned disk type VIXDISKLIB_DISK_VMFS_THIN.

Clone a Disk by Copying Data

This function copies data from one virtual disk to another, converting (disk type, size, hardware) as specified.

vSphere security Due to a vSphere 6.5 security enhancement, VixDiskLib_Clone can no longer clone local disks to remote disks that do not belong to any VM. Previous releases allowed use of a retained username and password, but now enhanced security verification is required. The procedure for VixDiskLib_Clone to make a remote connection is first to get an NFC ticket from the ESXi host based on the MoRef of the VM that manages the cloned-to disks. Then build an authenticated NFC connection to the host using that ticket. The MoRef of the VM must be specified in vmxSpec of the remote connection parameter passed to VixDiskLib_Clone. The VM must be powered off, and the target disks must already exist before cloning, not necessarily with the same names. Afterwards the VM may be removed from the inventory with UnregisterVM, but it must remain on the datastore with its cloned virtual disks.

Error Handling

These functions enhance the usefulness of error messages.

Return Error Description Text

VixDiskLib_GetErrorText() returns the textual description of a numeric error code.

char* msg = VixDiskLib_GetErrorText(errCode, NULL);

Free Error Description Text

VixDiskLib_FreeErrorText() deallocates space associated with the error description text.

VixDiskLib_FreeErrorText(msg);

Administrative Disk Operations

These functions rename, grow, defragment, shrink, and remove virtual disk.

Rename an Existing Disk

VixDiskLib_Rename() changes the name of a virtual disk. Use this function only when the virtual machine is powered off.

vixError = VixDiskLib_Rename(oldGlobals.diskPath, newGlobals.diskPath);

Grow an Existing Local Disk

VixDiskLib_Grow() extends an existing virtual disk by adding sectors. This function supports hosted disk, but not managed disk.

```
vixError =
VixDiskLib_Grow(appGlobals.connection, appGlobals.diskPath, size, FALSE, GrowProgressFunc, NULL);
```

Defragment an Existing Disk

VixDiskLib_Defragment() defragments an existing virtual disk. Defragmentation is effective with SPARSE type files, but might not do anything with FLAT type. In either case, the function returns VIX_OK. This function supports hosted disk, but not managed disk.

```
vixError = VixDiskLib_Defragment(disk.Handle(), DefragProgressFunc, NULL);
```

Defragment consolidates data in the 2GB extents, moving data to lower-numbered extents, and is independent of defragmentation tools in the guest OS, such as **Disk > Properties > Tools > Defragmentation** in Windows, or the **defrag** command for the Linux Ext2 file system.

VMware recommends defragmentation from the inside out: first within the virtual machine, then using this function or a VMware defragmentation tool, and finally within the host operating system.

Shrink an Existing Local Disk

VixDiskLib_Shrink() reclaims unused space in an existing virtual disk, unused space being recognized as blocks of zeroes. This is more effective (gains more space) with SPARSE type files than with pre-allocated FLAT type. On success, the function returns VIX_0K. This function supports hosted disk, but not managed disk.

```
vixError = VixDiskLib_Shrink(disk.Handle(), ShrinkProgressFunc, NULL);
```

In VMware system utilities, "prepare" zeros out unused blocks in the VMDK so "shrink" can reclaim them. In the API, use VixDiskLib_Write() to zero out unused blocks, and VixDiskLib_Shrink() to reclaim space. Shrink does not change the virtual disk capacity, but it makes more space available.

Unlink Extents to Remove Disk

VixDiskLib_Unlink() deletes all extents of the specified virtual disk, which unlinks (removes) the disk data. This is similar to the remove or erase command in a command tool.

```
vixError = VixDiskLib_Unlink(appGlobals.connection, appGlobals.diskPath);
```

Shut Down

All Virtual Disk API applications should call these functions at end of program.

Disconnect from Server

```
VixDiskLib_Disconnect() breaks an existing connection.
```

```
VixDiskLib_Disconnect(srcConnection);
```

Clean Up and Exit

```
VixDiskLib_Exit() cleans up the library before exit.
```

VixDiskLib_Exit();

Advanced Transport APIs

For managed disk, the first release of VDDK required network access ESXi host (LAN or NBDSSL transport). With VDDK 1.1 programs can access virtual disks directly on a storage device, LAN-free. Direct SAN access increases I/O performance. To select the most efficient transport method, a set of APIs is available, including:

- VixDiskLib_InitEx() Initializes the advanced transport library. You must specify the library location. Replaces VixDiskLib_Init() in your application.
- VixDiskLib_ListTransportModes() Lists transport modes that the virtual disk library supports.
- VixDiskLib_ConnectEx() Establishes a connection using the best transport mode available, or one you select, to access a given machine's virtual disk. Currently it does not check validity of transport type. Replaces VixDiskLib_Connect() in your application.

Initialize Virtual Disk API

VixDiskLib_InitEx() replaces VixDiskLib_Init() to initialize new releases of the library. Parameters are similar, except you should specify an actual libDir, and the configFile parameter is new. For multithreaded programming, you should write your own logFunc, because the default logging function is not thread-safe. VixDiskLib loads many libraries (DLLs or shared objects) at start time using the Path or LD_LIBRARY_PATH environment. At run time, libDir specifies were to load advanced transport modes, whitelist, and blacklist. On Windows, *libDir could be C:\Program Files\VMware\VMware Virtual Disk Development Kit. On Linux, *libDir could be /usr/lib/vmware-vix-disklib.

Logged messages appear by default in a temporary folder or log directory, for VDDK and for many other VMware products. See "Location of Log Files" on page 37.

The currently supported entries in the configFile are listed below. The correct way to specify a configuration is name=value. See Example 4-1 for a sample configuration file.

- tmpDirectory = "<TempDirectoryForLogging>"
- vixDiskLib.transport.LogLevel Overrides default logging for vixDiskLib advanced transport functions, not including NFC (see nfc.LogLevel below). The default value is 3, indicating audit. Each level includes all of the messages generated by lower numbered levels. Here are the levels:
 - \bullet 0 = Panic (failure only)
 - 1 = Error
 - 2 = Warning
 - 3 = Audit
 - 4 = Info
 - 5 = Verbose
 - 6 = Trivia
- vixDiskLib.transport.san.blacklist Specifies a device node path, or a comma separated list of device node paths, that VDDK uses as a list of LUNs on which not to attempt VMFS file system discovery. This has the effect of making fewer device nodes available to SAN transport, and avoids I/O on specific device node paths. The special string all indicates that VDDK should use only the device node paths specified by vixDiskLib.transport.san.whitelist.
- vixDiskLib.transport.san.whitelist Specifies a device node path, or a comma separated list of device node paths, that VDDK uses as a list of LUNs on which to attempt VMFS file system discovery. This has the effect of making more device nodes available to SAN transport, encouraging I/O on specific device node paths. Backup applications may create a special device node and whitelist this device node for use in addition to those found by VDDK's device node scanner. Backup applications can also blacklist specific devices found by VDDK's device node scanner to prevent use by SAN transport. Combining whitelist and blacklist, applications can establish a preferred device policy for backup I/O.
- vixDiskLib.disklib.EnableCache Caching by vixDiskLib is off (0) by default. Setting 1 turns it on. Caching increases performance when information is read repeatedly, or accessed randomly. In backup applications, information is usually accessed sequentially, and caching can actually reduce performance. Moreover with caching, backup applications risk getting stale information if a disk sector is rewritten (by another application) before the cache is refreshed.

• vixDiskLib.linuxSSL.verifyCertificates – Whether to check SSL certificates and thumbprints when connecting to a virtual machine. The only possible value is 1 for On, which is the default.

The following NFC related options override the default numbers provided to the various NFC functions. The NFC timeouts shown in Example 4-1 correspond to default values on ESXi 5.x hosts.

- vixDiskLib.nfc.AcceptTimeoutMs Overrides default value (3 minutes) for NFC accept operations.
- vixDiskLib.nfc.RequestTimeoutMs Overrides default value (3 minutes) for NFC request operations.
- vixDiskLib.nfc.ReadTimeoutMs Overrides default value (one minute) for NFC read operations.
- vixDiskLib.nfc.WriteTimeoutMs Overrides default value (ten minutes) for NFC write operations.
- vixDiskLib.nfcFssrvr.TimeoutMs Overrides the default value (default is 0, indefinite waiting) for NFC file system operations. If you specify a value, then a timeout occurs if the file system is idle for the indicated period of time. The hazard of using the default value is that in the rare case of a catastrophic communications failure, the file system will remain locked.
- vixDiskLib.nfcFssrvrWrite.TimeoutMs Overrides the default value (default is no timeout) for NFC file system write operations. The timeout is specified in milliseconds. If you specify a value, it will time out when a write operation fails to complete in the specified time interval.
- vixDiskLib.nfc.LogLevel Overrides the default logging level for NFC operations. The default value is 1, indicating error messages only. The meaning of values is listed here. Each level includes all of the messages generated by lower numbered levels. This is the final NFC setting.
 - \bullet 0 = Quiet (minimal logging)
 - 1 = Error
 - 2 = Warning
 - 3 = Info
 - 4 = Debug

Example 4-1. Sample InitEx configuration file

```
# temporary directory for logs etc.
tmpDirectory="/usr/local/vendorapp/var/vmware/temp"
# log level 0 to 6 for quiet ranging to verbose
vixDiskLib.transport.LogLevel=2
# disable caching to disk
vixDiskLib.disklib.EnableCache=0
# whether to check SSL thumbprint on Linux - has no effect
vixDiskLib.linuxSSL.verifyCertificates=1
# network file copy options
vixDiskLib.nfc.AcceptTimeoutMs=180000
vixDiskLib.nfc.RequestTimeoutMs=180000
vixDiskLib.nfc.ReadTimeoutsMs=60000
vixDiskLib.nfc.WriteTimeoutsMs=600000
vixDiskLib.nfcFssrvr.TimeoutMs=0
vixDiskLib.nfcFssrvrWrite.TimeoutMs=0
# nfc.LogLevel (0 = Quiet, 1 = Error, 2 = Warning, 3 = Info, 4 = Debug)
vixDiskLib.nfc.LogLevel=2
```

Timeout values are stored in a 32-bit field, so the maximum timeout you may specify is 2G (2,147,483,648). Timeouts are specified in milliseconds and apply to each disk handle. NFC settings apply to NBDSSL but not to SAN or HotAdd.

Location of Log Files

On Linux, log messages appear under /var/log by default. On Windows, they appear in a temporary folder, whose location can change from time to time. Early Windows systems used C:\Windows\Temp. Windows XP and Server 2003 use C:\Documents and Settings\cuser>\Local Settings\Temp\vmware-<user>. Vista, Windows 7, and Server 2008 use C:\Users\cuser>AppData\Local\Temp\vmware-<user>.

On all versions of Windows the user's TEMP environment setting overrides the default Temp folder location. Temporary is something of a misnomer because files are never deleted from the Temp folder, unless the user or an application deletes them. If the TEMP or Windows default Temp folder is not found, VDDK (and other VMware software) have a fallback to <localAppDir>\Temp.

Alternatively, your software can set a custom temporary directory, as shown in Example 4-1.

List Available Transport Methods

The VixDiskLib_ListTransportModes() function returns the currently supported transport methods as a colon-separated string value, currently "file:san:hotadd:nbdssl" where nbdssl indicates LAN transport. The default transport priority over the network is san:hotadd:nbdssl assuming all are available.

```
printf("Transport methods: %s\n", VixDiskLib_ListTransportModes());
```

The advanced transport license for VDDK includes all transport types.

Connect to VMware vSphere

VixDiskLib_ConnectEx() connects the library to managed disk on an ESXi host or through vCenter Server. For hosted disk, it works like VixDiskLib_Connect(), but takes three additional parameters:

- Boolean indicating TRUE for read-only access, often faster, or FALSE for read/write access. If connecting read-only, later calls to VixDiskLib_Open() are always read-only regardless of the openFlags setting.
- Managed object reference (MoRef) of the snapshot to access using this connection. This is required for most transport methods (SAN, HotAdd, NBDSSL) and to access a powered-on virtual machine. You must also specify the associated vmxSpec property in connectParams. When connecting to an ESXi host, provide the ESXi MoRef. When connecting by vCenter Server, pass the vSphere MoRef, which differs.
- Preferred transport method, or NULL to accept the defaults. If you specify any advanced transport mode as the only method, and that method is not available, the VixDiskLib_ConnectEx() call does not fail, but the subsequent VixDiskLib_Open() call will fall back to NBDSSL mode.

```
VixDiskLibConnectParams cnxParams = {0};
if (appGlobals.isRemote) {
    cnxParams.vmName = vmxSpec;
    cnxParams.serverName = hostName;
    cnxParams.credType = VIXDISKLIB_CRED_UID;
    cnxParams.creds.uid.userName = userName;
    cnxParams.creds.uid.password = password;
    cnxParams.port = port;
}
VixError vixError = VixDiskLib_ConnectEx(&cnxParams, TRUE, "snapshot-47", NULL, &connection);
```

When a program calls VixDiskLib_ConnectEx() with NULL parameter to accept default transport mode, SAN is selected as the preferred mode, if SAN storage is available from the ESXi host. Then if the program opens a virtual disk on local storage, subsequent writes will fail. In this case, the program should explicitly pass nbdssl as the preferred transport mode.

The port is where vCenter Server listens for API queries. Specifying null allows the library to select the port, usually 443 (HTTPS). By default VADP uses the same port for virtual machine operations as other SOAP-based Web Services. By default VDDK uses port 902 (VIX automation) for NBDSSL data transport.

Connect to ESXi hosts In the connection parameters cnxParams, the vmxSpec managed object reference would be different on an ESXi host than on the vCenter Server. ESXi hosts offer no prepare-for and end access protection. Otherwise ESXi host connections are similar to vCenter Server connections.

```
vmxSpec = "moid=23498";
vmxSpec = "moid=898273";
```

Reuse a vCenter Server session As of vSphere 6.5, you can recycle a vCenter Server session to avoid connection overflow. Set the credential type to VIXDISKLIB_CRED_SESSIONID and supply the value of vmware_soap_session from a still-live vCenter Server session.

```
if (appGlobals.isRemote) {
```

```
cnxParams.vmName = vmxSpec;
cnxParams.serverName = hostName;
cnxParams.credType = VIXDISKLIB_CRED_SESSIONID;
cnxParams.creds.sessionId.cookie = cookie;
cnxParams.creds.sessionId.userName = userName;
cnxParams.creds.sessionId.key = password;
cnxParams.port = port;
}
```

Get Selected Transport Method

The VixDiskLib_GetTransportMode() function returns the transport method selected for diskHandle.

```
printf("Selected transport method: %s\n", VixDiskLib_GetTransportMode(diskHandle));
```

Prepare For Access and End Access

The VixDiskLib_PrepareForAccess() function notifies a vCenter-managed host that a virtual machine's disks are being opened, probably for backup, so the host should postpone virtual machine operations that might interfere with virtual disk access. Call this function before creating a snapshot on a virtual machine. Internally, this function disables the vSphere API method RelocateVM_Task.

```
vixError = VixDiskLib_PrepareForAccess(&cnxParams, "vmName");
```

The connection parameters must indicate one virtual machine only. When opening a managed disk, provide valid credentials for the vCenter Server that manages the ESXi host with the disk. The second parameter is currently just for identity tracking purposes, and is limited to 50 characters. It could be the virtual machine name or the name of your application. If you run VixDiskLib_PrepareForAccess() directly on an ESXi host, the system throws an error saying "VDDK: HostAgent is not a vCenter, cannot disable symotion."

Every VixDiskLib_PrepareForAccess() call should have a matching VixDiskLib_EndAccess() call.

The VixDiskLib_EndAccess() function notifies the host that a virtual machine's disks have been closed, so operations that rely on the virtual disks to be closed, such as vMotion, can now be allowed. Call this function after closing all the virtual disks, and after deleting the virtual machine snapshot. Normally this function is called after previously calling VixDiskLib_PrepareForAccess, but you can call it to clean up after a crash. Internally, this function re-enables the vSphere API method RelocateVM_Task.

```
vixError = VixDiskLib_EndAccess(&cnxParams, "vmName");
```

Here is a code snippet showing use of PrepareForAccess in a backup program that waits up to 10 minutes for Storage vMotion to finish. Regular vMotion would finish much faster than that.

```
/* New sample code accounts for VMODL_TYPE_VIM_FAULT_METHOD_ALREADY_DISABLED_FAULT */
if (appGlobals.vmxSpec != NULL) {
   for (int i = 0; i < 10; i++) {
      vixError = VixDiskLib_PrepareForAccess(&cnxParams, "Sample");
      if (vixError == VIX_OK) {
            break;
      } else {
            Sleep(60000);
      }
   }
}</pre>
```

SAN Mode on Linux Uses Direct Mode

With SAN transport on Linux, read and write operations are performed in "direct" mode (O_DIRECT), meaning that no read or write buffering is done. Direct mode prevents other processes from accessing the latest data, and avoids loss of information if the process dies before committing its write buffers. In direct mode, the most time efficient performance can be achieved if applications follow these guidelines when performing reads and writes. These guidelines are applicable to Windows also.

- The buffer used for data transfer should be aligned on a page boundary.
- The transfer length should be an even multiple of the page size.

Clean Up After Disconnect

If virtual machine state was not cleaned up correctly after connection shut down, VixDiskLib_Cleanup() removes extra state for each virtual machine. Its three parameters specify connection, and pass back the number of virtual machines cleaned up, and the number remaining to be cleaned up.

```
int numCleanedUp, numRemaining;
VixError vixError = VixDiskLib_Cleanup(&cnxParams, &numCleanedUp, &numRemaining);
```

Updating Applications for Advanced Transport

To update your applications for advanced transport with managed disk, follow these steps:

- Find all instances of VixDiskLib_Connect() and change them to VixDiskLib_ConnectEx().
 The vixDiskLib sample program was extended to use VixDiskLib_ConnectEx() with the -mode option.
- 2 Likewise, change VixDiskLib_Init() to VixDiskLib_InitEx() and be sure to call it only once.
- 3 Disable virtual machine relocation with the VixDiskLib_PrepareForAccess() call.
- 4 Add parameters in the middle:
 - TRUE for high performance read-only access, FALSE for read/write access.
 - Snapshot moRef, if applicable.
 - NULL to accept transport method defaults (recommended).
- 5 Re-enable virtual machine relocation with the VixDiskLib_EndAccess() call.
- 6 Find VixDiskLib_Disconnect() near the end of program, and for safety add a VixDiskLib_Cleanup() call immediately afterwards.
- 7 Compile with the new flexible-transport-enabled version of VixDiskLib.

The advanced transport functions are useful for backing up or restoring data on virtual disks managed by VMware vSphere. Backup is based on the snapshot mechanism, which provides a data view at a certain point in time, and allows access to quiescent data on the parent disk while the child disk continues changing.

Algorithm for vSphere Backup

A typical backup application follows this algorithm:

- Preferably through vCenter Server, contact the ESXi host and discover the target virtual machine.
- Ask the ESXi host to take a snapshot of the target virtual machine.
- Using the vSphere API (PropertyCollector), capture configuration (VirtualMachineConfigInfo) and changed block information (with queryChangedDiskAreas). Save these for later.
- Using advanced transport functions and VixDiskLib, access the snapshot and save the data in it. If Changed Block Tracking is enabled, the snapshot contains only incremental backup data.
- Ask the ESXi host to delete the backup snapshot.

A typical back-in-time disaster recovery or file-based restore follows this algorithm:

- Preferably through VMware vCenter, contact the ESXi host containing the target virtual machine.
- Ask the ESXi host to halt and power off the target virtual machine.
- Using advanced transport functions, restore a snapshot from saved backup data.
- For disaster recovery to a previous point in time, have the virtual machine revert to the restored snapshot. For file-based restore, mount the snapshot and restore requested files.

Chapter 7, "Backing Up Virtual Disks in vSphere," on page 57 presents these algorithms in more detail and includes code samples. For best practices in backup and restore, see "Tips and Best Practices" on page 81.

Backup and Recovery Example

The VMware vSphere API method queryChangedDiskArea returns a list of disk sectors that changed between an existing snapshot, and some previous time identified by a change ID.

The queryChangedDiskAreas method takes four arguments, including a snapshot reference and a change ID. It returns a list of disk sectors that changed between the time indicated by the change ID and the time of the snapshot. If you specify change ID as * (star), queryChangedDiskAreas returns a list of allocated disk sectors so your backup can skip the unallocated sectors of sparse virtual disk.

Suppose that you create an initial backup at time *T1*. Later at time *T2* you take an incremental backup, and another incremental backup at time *T3*. (You could use differential backups instead of incremental backups, which would trade off greater backup time and bandwidth for shorter restore time.)

For the full backup at time T1:

- 1 Keep a record of the virtual machine configuration, VirtualMachineConfigInfo.
- 2 Create a snapshot of the virtual machine, naming it snapshot_T1.
- 3 Obtain the change ID for each virtual disk in the snapshot, **changeId_T1** (per VMDK).
- 4 Back up the sectors returned by queryChangedDiskAreas(..."*"), avoiding unallocated disk.
- 5 Delete snapshot_T1, keeping a record of changeId_T1 along with lots of backed-up data.

For the incremental backup at time T2:

- 1 Create a snapshot of the virtual machine, naming it snapshot_T2.
- 2 Obtain the change ID for each virtual disk in the snapshot, changeId_T2 (per VMDK).
- 3 Back up the sectors returned by queryChangedDiskAreas(snapshot_T2,... changeId_T1).
- 4 Delete **snapshot_T2**, keeping a record of **changeId_T2** along with backed-up data.

For the incremental backup at time T3:

- 1 Create a snapshot of the virtual machine, naming it snapshot_T3.
 At time T3 you can no longer obtain a list of changes between T1 and T2.
- 2 Obtain the change ID for each virtual disk in the snapshot, **changeId_T3** (per VMDK).
- Back up the sectors returned by queryChangedDiskAreas(snapshot_T3,... changeId_T2).
 A differential backup could be done with queryChangedDiskAreas(snapshot_T3,... changeId_T1).
- 4 Delete **snapshot_T3**, keeping a record of **changeId_T3** along with backed-up data.

For a disaster recovery at time T4:

- 1 Create a new virtual machine with no guest operating system installed, using configuration parameters you previously saved from VirtualMachineConfigInfo. You do not need to format the virtual disks, because restored data includes formatting information.
- 2 Restore data from the backup at time T3. Keep track of which disk sectors you restore.
- 3 Restore data from the incremental backup at time T2, skipping any sectors already recovered.
 With differential backup, you can skip copying the T2 backup.
- 4 Restore data from the full backup at time T1, skipping any sectors already recovered. The reason for working backwards is to get the newest data while avoiding unnecessary data copying.
- 5 Power on the recovered virtual machine.

When programs open remote disk with SAN transport mode, they can write to the base disk, but they cannot write to a snapshot (redo log). Opening and writing snapshots is supported only for hosted disk.

Multithreading Considerations

In multithreaded programs, disk requests should be serialized by the client program. Disk handles are not bound to a thread and may be used across threads. You can open a disk in one thread and use its handle in another thread, provided you serialize disk access. Alternatively you can use a designated open-close thread, as shown in the workaround below.

Multiple Threads and VixDiskLib

VDDK supports concurrent I/O to multiple virtual disks, with certain limitations:

- VixDiskLib_InitEx() or VixDiskLib_Init() should be called once per process, from the main thread.
- In the VixDiskLib_InitEx() or VixDiskLib_Init() function call, do not specify logging callbacks as NULL. This causes VixDiskLib to provide its default logging functions, which are not thread safe. If you are using VDDK in a multithreaded environment, you must provide your own thread-safe log functions.
- When you call VixDiskLib_Open() and VixDiskLib_Close(), VDDK initializes and uninitializes a number of libraries, some of which do not work if called from multiple threads. For example, this fails:

```
Thread 1: VixDiskLib_Open ..... VixDiskLib_Close
Thread 2: ..... VixDiskLib_Open ..... VixDiskLib_Open ..... VixDiskLib_Close
```

The workaround is to use one designated thread to do all opens and closes, and to have other worker threads doing reads and writes. This diagram shows concurrent reads on two separate disk handles. Concurrent reads on the same disk handles are not allowed.

```
Open/Close Thread:

VixDiskLib_Open ..... VixDiskLib_Open ..... VixDiskLib_Close ..... VixDiskLib_Close ..... (handle1) (handle2) (handle1) (handle2)

I/O Thread 1:
(owns handle1) VixDiskLib_Read ... VixDiskLib_Read ...

I/O Thread 2:
(owns handle2) VixDiskLib_Read ... VixDiskLib_Read ... VixDiskLib_Read ...
```

Capabilities of Library Calls

This section describes limitations, if any.

Support for Managed Disk

Some operations are not supported:

- For VixDiskLib_Connect() to open a managed disk connection, you must provide valid vSphere access credentials. On ESXi hosts, VixDiskLib_Open() cannot open a single link in a disk chain.
- For VixDiskLib_Create() to create a managed disk on an ESXi host, first create a hosted type disk, then use VixDiskLib_Clone() to convert the hosted virtual disk to managed virtual disk.
- VixDiskLib_Defragment() can defragment hosted disks only.
- VixDiskLib_Grow() can grow hosted disks only.
- VixDiskLib_Unlink() can delete hosted disks only.
- Until ESXi 5.1, the HotAdd transport was available only with vSphere Enterprise Edition and higher.

Support for Hosted Disk

Most everything (except advanced transport) is supported, except:

- The VixDiskLib_ConnectEx() extended connect function.
- SAN and HotAdd advanced transports.
- VixDiskLib_PrepareForAccess() and VixDiskLib_EndAccess() to delay Storage VMotion.

Virtual Disk API Sample Code

5

This chapter discusses the VDDK sample program, in the following sections:

- "Compiling the Sample Program" on page 43
- "Usage Message" on page 44
- "Walk-Through of Sample Program" on page 45

Compiling the Sample Program

The sample program is written in C++, although the Virtual Disk API also supports C. For compilation to succeed, the correct DLLs or shared objects must be loaded. You can ensure the success of dynamic loading in a variety of ways.

- Set the path inside the VDDK program.
- Set the path for the shell being used in Linux or in Visual Studio for Windows.
 For a default installation, the Linux path is /usr/share/doc/vmware-vix-disklib/sample.
- In Windows, set the Path element in the System Variables.

To do this in Windows XP, right-click **Computer > Properties > Advanced > Environment Variables**, select **Path** in the **System Variables** lower list, click **Edit**, and add the path of the VDDK bin directory.

In Windows 7, right-click **Computer > Properties > Advanced System Settings > Environment Variables**, select Path in the **System Variables** list, click **Edit**, and add the path of the VDDK bin directory.

C:\Program Files\VMware\VMware Virtual Disk Development Kit\doc\sample\ is the default path.

Note that VDDK loads DLLs by relative path rather than absolute path, so conflicting versions of the DLLs could cause problems.

Visual C++ on Windows

To compile the program, find the sample source vixDiskLibSample.cpp at this location:

C:\Program Files\VMware\VMware Virtual Disk Development Kit\doc\sample\

For VDDK 5.5 and later, make sure that you have the 64-bit debugging tools installed along with Visual Studio. Double-click the vcproj file, possibly convert format to a newer version, and choose **Build > Build Solution**.

To execute the compiled program, choose **Debug > Start Without Debugging**, or type this in a command prompt after changing to the doc\sample location given above:

Debug\vixdisklibsample.exe

SLN and VCPROJ Files

The Visual Studio solution file vixDiskLibSample.sln and project file vixDiskLibSample.vcproj are included in the sample directory.

C++ on Linux Systems

Find the sample source in this directory:

```
/usr/share/doc/vmware-vix-disklib/samples/diskLib
```

You can copy vixDiskLibSample.cpp and its Makefile to a directory where you have write permission, or switch user to root. On some Linux systems you need to add #include statements for <stdio.h> and <string.h> after the #else clause on line 15. Type the make command to compile. Run the application:

```
make
./vix-disklib-sample
```

NOTE If this fails, edit /etc/ld.so.conf and run ldconfig as root or change your LD_LIBRARY_PATH environment to include the library installation path, /usr/lib/vmware-vix-disklib/lib64.

Makefile

The Makefile fetches any packages that are required for compilation but are not installed.

Library Files Required

The virtual disk library comes with dynamic libraries, or shared objects on Linux, to simplify the delivery of third-party and open source components.

Windows requires the lib/vixDiskLib.lib file for linking, and the bin/*.dll files at runtime.

Linux uses .so files for both linking and running.

Usage Message

Running the sample application without arguments produces the following usage message:

```
Usage: vixdisklibsample command [options] diskPath
commands:
-create : creates a sparse virtual disk with capacity specified by -cap
-redo parentPath : creates a redo log 'diskPath' for base disk 'parentPath'
-info : displays information for specified virtual disk
-dump: dumps the contents of specified range of sectors in hexadecimal
-fill : fills specified range of sectors with byte value specified by -val
-wmeta key value : writes (key, value) entry into disk's metadata table
-rmeta key: displays the value of the specified metada entry
-meta : dumps all entries of the disk's metadata
-clone sourcePath : clone source vmdk possibly to a remote site
-readbench blocksize: do read benchmark on a disk using the specified I/O block size in sectors
-writebench blocksize: do write benchmark on disk using the specified I/O block size in sectors
options:
 -adapter [ide|scsi] : bus adapter type for 'create' option (default='scsi')
-start n : start sector for 'dump/fill' options (default=0)
-count n : number of sectors for 'dump/fill' options (default=1)
-val byte: byte value to fill with for 'write' option (default=255)
-cap megabytes : capacity in MB for -create option (default=100)
-single : open file as single disk link (default=open entire chain)
-multithread n: start n threads and copy the file to n new files
-host hostname : hostname / IP addresss (ESXi or vCenter)
-user userid : user name on host (default = root)
-password password : password on host
 -port port : port to use to connect to host (default = 443)
-vm vmPath=/path/to/vm : inventory path to vm that owns the virtual disk
-libdir dir : Directory containing vixDiskLibPlugin library
-initex configfile : Use VixDiskLib_InitEx
-ssmoref moref : Managed object reference of VM snapshot
-mode mode : Mode string to pass into VixDiskLib_ConnectEx
-thumb string : Provides a SSL thumbprint string for validation
```

The -thumb option is a new security-related feature in the VDDK 5.1 release. See "SSL Certificate Thumbprint" on page 48.

The sample program's -single option, which opens a single link instead of the entire disk chain, is supported for (local) hosted disk, but not for (remote) managed disk.

To connect to an ESXi host with the sample program, you must specify the options <code>-host</code>, <code>-user</code>, <code>-password</code>, and provide a <code>diskPath</code> on the ESXi host's datastore. For example:

```
vix-diskLib-sample -info -host esx5 -user root -password secret "[datastore1] <VM>/<VM>.vmdk"
```

To connect to vCenter Server, you must also specify the options —libdir and —vm. Programs need libdir so the DiskLibPlugin can connect with vCenter Server, which must locate the VM. For example:

```
vix-diskLib-sample -info -host vc5 -user Administrator -password secret
    -libdir <pluginDir> -vm vmPath=<path/to/VM> "[<partition>] <VM>/<VM>.vmdk"
```

The vmPath is formulated in vSphere Client by starting at vCenter and inserting /vm/ before the VM name. The diskPath is ascertained by clicking Edit Settings > Hard Disk and copying the Disk File name.

To connect using an advanced transport, for example to virtual machine disk on SAN storage, you must also specify the options <code>-mode</code> and <code>-ssmoref</code>. The transport mode and managed object reference (of a snapshot) are required for <code>VixDiskLib_ConnectEx()</code>. To find the <code>ssmoref</code>, log in to the managed object browser for the vCenter Server, and click <code>content > rootFolder > Datacenter > datastore > vm > snapshot</code>. A snapshot must exist, because it is a bad idea to open the base disk of a powered-on VM.

On Windows, the VDDK package installs diskLibPlugin.dll in the \bin folder, not the \lib folder, so change <pluginDir> accordingly.

Walk-Through of Sample Program

The sample program is the same for Windows as for Linux, with #ifdef blocks for Win32.

Include Files

Windows dynamic link library (DLL) declarations are in process.h, while Linux shared object (.so) declarations are in dlfcn.h. Windows offers the tchar.h extension for Unicode generic text mappings, not readily available in Linux.

Definitions and Structures

The sample program uses twelve bitwise shift operations (1 << 11) to track its available commands and the multithread option. The Virtual Disk API has about 30 library functions, some for initialization and cleanup. The following library functions are not demonstrated in the sample program:

- VixDiskLib_Rename()
- VixDiskLib_Defragment()
- VixDiskLib_Grow()
- VixDiskLib_Shrink()
- VixDiskLib_Unlink()
- VixDiskLib_Attach()

The sample program transmits state in the appGlobals structure.

Dynamic Loading

The #ifdef DYNAMIC_LOADING block is long, starting on line 97 and ending at line 339. This block contains function definitions for dynamic loading. It also contains the LoadOneFunc() procedure to obtain any requested function from the dynamic library and the DynLoadDiskLib() procedure to bind it. This demonstration feature could also be called "runtime loading" to distinguish it from dynamic linking.

To try the program with runtime loading enabled on Linux, add -DDYNAMIC_LOADING after g++ in the Makefile and recompile. On Windows, define DYNAMIC_LOADING in the project.

Wrapper Classes

Below the dynamic loading block are two wrapper classes, one for error codes and descriptive text, and the other for the connection handle to disk.

The error wrapper appears in catch and throw statements to simplify error handling across functions.

Wrapper class VixDisk is a clean way to open and close connections to disk. The only time that library functions VixDiskLib_Open() and VixDiskLib_Close() appear elsewhere, aside from dynamic loading, is in the CopyThread() function near the end of the sample program.

Command Functions

The print-usage message appears next, with output partially shown in "Usage Message" on page 44.

Next comes the main() function, which sets defaults and parses command-line arguments to determine the operation and possibly set options to change defaults. Dynamic loading occurs, if defined. Notice the all-zero initialization of the VixDiskLibConnectParams declared structure:

```
VixDiskLibConnectParams cnxParams = {0};
```

For connections to an ESXi host, credentials including user name and password must be correctly supplied in the <code>-user</code> and <code>-password</code> command-line arguments. Both the <code>-host</code> name of the ESXi host and its <code>-vm</code> inventory path (<code>vmxSpec</code>) must be supplied. When set, these values populate the <code>cnxParams</code> structure. Initialize all parameters, especially <code>vmxSpec</code>, or else the connection might behave unexpectedly.

A call to VixDiskLib_Init() initializes the library. In a production application, you can supply appropriate log, warn, and panic functions as parameters, in place of NULL.

A call to VixDiskLib_Connect() creates a connection to disk. If host cnxParams.serverName is null, as it is without the -host argument, a connection is made to hosted disk on the local host. Otherwise a connection is made to managed disk on the remote host. With -ssmoref argument, advanced transport is used.

Next, an appropriate function is called for the requested operation, followed by error information if applicable. Finally, the main() function closes the library connection to disk and exits.

DoInfo()

This procedure calls VixDiskLib_GetInfo() for information about the virtual disk, displays results, and calls VixDiskLib_FreeInfo() to reclaim memory. The parameter disk.Handle() comes from the VixDisk wrapper class discussed in "Wrapper Classes" on page 46.

In this example, the sample program connects to an ESXi host named esx5 and displays virtual disk information for a Red Hat Enterprise Linux client. For an ESXi host, path to disk is often something like [datastore1] followed by the virtual machine name and the VMDK filename.

```
vix-diskLib-sample -info -host esx5 -user root -password secret "[datastore1] RHEL6/RHEL6.vmdk"
Disk "[datastore1] RHEL6/RHEL6.vmdk" is open using transport mode "nbdssl".
capacity = 4194304 sectors
number of links = 1
adapter type = LsiLogic SCSI
BIOS geometry = 0/0/0
physical geometry = 261/255/63
Transport modes supported by vixDiskLib: file:nbdssl
```

If you multiply physical geometry numbers (261 cylinders * 255 heads per cylinder * 63 sectors per head) the result is a capacity of 4192965 sectors, although the first line says 4194304. A small discrepancy is possible due to rounding. In general, you get at least the capacity that you requested. The number of links specifies the separation of a child from its original parent in the disk chain (redo logs), starting at one. The parent has one link, its child has two links, the grandchild has three links, and so forth.

DoCreate()

This procedure calls VixDiskLib_Create() to allocate virtual disk. Adapter type is SCSI unless specified as IDE on the command line. Size is 100MB, unless set by -cap on the command line. Because the sector size is 512 bytes, the code multiplies appGlobals.mbsize by 2048 instead of 1024. Type is always monolithic sparse and Workstation 5. In a production application, progressFunc and callback data can be defined rather than NULL. Type these commands to create a sample VMDK file (the first line is for Linux only):

```
export LD_LIBRARY_PATH=/usr/lib/vmware-vix-disklib/lib64
vix-disklib-sample -create sample.vmdk
```

As a VMDK file, monolithic sparse (growable in a single file) virtual disk is initially 65536 bytes (2^16) in size, including overhead. The first time you write to this type of virtual disk, as with DoFill() below, the VMDK expands to 131075 bytes (2^17), where it remains until more space is needed. You can verify file contents with the –dump option.

DoRedo()

This procedure calls VixDiskLib_CreateChild() to establish a redo log. A child disk records disk sectors that changed since the parent disk or previous child. Children can be chained as a set of redo logs.

The sample program does not demonstrate use of VixDiskLib_Attach(), which you can use to access a link in the disk chain. VixDiskLib_CreateChild() establishes a redo log, with the child replacing the parent for read/write access. Given a pre-existing disk chain, VixDiskLib_Attach() creates a related child, or a cousin you might say, that is linked into some generation of the disk chain.

For a diagram of the attach operation, see Figure 4-2, "Child Disks Created from Parent," on page 33.

Write by DoFill()

This procedure calls VixDiskLib_Write() to fill a disk sector with ones (byte value FF) unless otherwise specified by -val on the command line. The default is to fill only the first sector, but this can be changed with options -start and -count on the command line.

DoReadMetadata()

This procedure calls VixDiskLib_ReadMetadata() to serve the -rmeta command-line option. For example, type this command to obtain the universally unique identifier:

vix-disklib-sample -rmeta uuid sample.vmdk

DoWriteMetadata()

This procedure calls VixDiskLib_WriteMetadata() to serve the -wmeta command-line option. For example, you can change the tools version from 1 to 2 as follows:

vix-disklib-sample -wmeta toolsVersion 2 sample.vmdk

DoDumpMetadata()

This procedure calls VixDiskLib_GetMetadataKeys() then VixDiskLib_ReadMetadata() to serve the -meta command-line option. Two read-metadata calls are needed for each key: one to determine length of the value string and another to fill in the value. See "Get Metadata Table from Disk" on page 31.

In the following example, the sample program connects to an ESXi host named esx3 and displays the metadata of the Red Hat Enterprise Linux client's virtual disk. For an ESXi host, path to disk might be [storage1] followed by the virtual machine name and the VMDK filename.

```
vix-diskLib-sample -meta -host esx3 -user admin -password secret "[storage1]RHEL5/RHEL5.vmdk"
geometry.sectors = 63
geometry.heads = 255
geometry.cylinders = 522
adapterType = buslogic
toolsVersion = 1
virtualHWVersion = 7
```

Tools version and virtual hardware version appear in the metadata, but not in the disk information retrieved by "DoInfo()" on page 46. Geometry information and adapter type are repeated, but in a different format. Other metadata items not listed above might exist.

DoDump()

This procedure calls VixDiskLib_Read() to retrieve sectors and displays sector contents on the output in hexadecimal. The default is to dump only the first sector numbered zero, but you can change this with the -start and -count options. Here is a sequence of commands to demonstrate:

```
vix-disklib-sample -create sample.vmdk
vix-disklib-sample -fill -val 1 sample.vmdk
vix-disklib-sample -fill -val 2 -start 1 -count 1 sample.vmdk
vix-disklib-sample -dump -start 0 -count 2 sample.vmdk
od -c sample.vmdk
```

On Linux (or Cygwin) you can run the od command to show overhead and metadata at the beginning of file, and the repeated ones and twos in the first two sectors. The –dump option of the sample program shows only data, not overhead.

DoTestMultiThread()

This procedure employs the Windows thread library to make multiple copies of a virtual disk file. Specify the number of copies with the <code>-multithread</code> command-line option. For each copy, the sample program calls the <code>CopyThread()</code> procedure, which in turn calls a sequence of six Virtual Disk API routines.

On Linux the multithread option is unimplemented.

DoClone()

This procedure calls VixDiskLib_Clone() to make a copy of the data on virtual disk. A callback function, supplied as the sixth parameter, displays the percent of cloning completed. For local hosted disk, the adapter type is SCSI unless specified as IDE on the command line, size is 200MB, unless set by -cap option, and type is monolithic sparse, for Workstation 5. For an ESXi host, adapter type is taken from managed disk itself, using the connection parameters established by VixDiskLib_Connect().

The final parameter TRUE means to overwrite if the destination VMDK exists.

The clone option is an excellent backup method. Often the cloned virtual disk is smaller, because it can be organized more efficiently. Moreover, a fully allocated flat file can be converted to a sparse representation.

SSL Certificate Thumbprint

The sample program in the VDDK 5.1 release added the -thumb option to allow an SSL certificate thumbprint to be provided and used. The thumbprint is used for authentication through vCenter Server.

Practical Programming Tasks

6

This chapter presents some practical programming challenges not covered in the sample program, including:

- "Scan VMDK for Virus Signatures" on page 49
- "Creating Virtual Disks" on page 50
- "VMDK File Versions" on page 51
- "Working with Virtual Disk Data" on page 52
- "Managing Child Disks" on page 53
- "RDM Disks and Virtual BIOS" on page 54
- "Interfacing With VMware vSphere" on page 55

Scan VMDK for Virus Signatures

One of the "Use Cases for the Virtual Disk Library" on page 12 is to scan a VMDK for virus signatures. Using our sample program framework, the function in Example 6-1 implements the –virus command-line option, using hypothetical library routine SecureVirusScan(), supplied by an antivirus software vendor. The library routine scans a buffer against the vendor's latest pattern library, returning TRUE if it identifies a virus.

Example 6-1. Function to Scan VMDK for Viruses

```
extern int SecureVirusScan(const uint8 *buf, size_t n);
    DoVirusScan - Scan the content of a virtual disk for virus signatures.
*/
static void DoVirusScan(void)
    VixDisk disk(appGlobals.connection, appGlobals.diskPath, appGlobals.openFlags);
    VixDiskLibDiskInfo info;
    uint8 buf[VIXDISKLIB_SECTOR_SIZE];
    VixDiskLibSectorType sector;
    VixError vixError = VixDiskLib_GetInfo(disk.Handle(), &info);
    CHECK_AND_THROW(vixError);
    cout << "capacity = " << info.capacity << " sectors" << endl;</pre>
    // read all sectors even if not yet populated
    for (sector = 0; sector < info.capacity; sector++) {</pre>
        vixError = VixDiskLib_Read(disk.Handle(), sector, 1, buf);
        CHECK_AND_THROW(vixError);
        if (SecureVirusScan(buf, sizeof buf)) {
            printf("Virus detected in sector %d\n", sector);
        }
    cout << info.capacity << " sectors scanned" << endl;</pre>
}
```

This function calls VixDiskLib_GetInfo() to determine the number of sectors allocated in the virtual disk. The number of sectors is available in the VixDiskLibDiskInfo structure, but normally not in the metadata. With SPARSE type layout, data can occur in any sector, so this function reads all sectors, whether filled or not. VixDiskLib_Read() continues without error when it encounters an empty sector full of zeroes.

The following difference list shows the remaining code changes necessary for adding the -virus option to the vixDiskLibSample.cpp sample program:

Creating Virtual Disks

This section discusses the types of local VMDK files and how to create virtual disk for a remote ESXi host.

Creating Local Disk

The sample program presented in Chapter 5 creates virtual disk of type MONOLITHIC_SPARSE, in other words one big file, not pre-allocated. This is the default because modern file systems, in particular NTFS, support files larger than 2GB, and can hold more than 2GB of total data. This is not true of legacy file systems, such as FAT16 on MS-DOS and early Windows, or the ISO9660 file system for writing files on CD, or NFS version 2, or Linux kernel 2.4. All are limited to 2GB per volume. FAT and FAT32 were extended to 4GB in NT 3.51.

However, a SPLIT virtual disk might be safer than the MONOLITHIC variety, because if something goes wrong with the underlying host file system, some data might be recoverable from uncorrupted 2GB extents. VMware products do their best to repair a damaged VMDK, but having a split VMDK increases the chance of salvaging files during repair. On the downside, SPLIT virtual disk involves higher overhead (more file descriptors) and increases administrative complexity.

When required for a FAT16 or early Linux file system, you can create SPLIT_SPARSE virtual disk. The change is simple: the line highlighted in boldface. The sample program could be extended to have an option for this.

NOTE You can split VMDK files into smaller than 2GB extents, but created filenames still follow the patterns shown in Table 3-1, "VMDK Virtual Disk Files," on page 20.

This one-line change to DoCreate() causes creation of 200MB split VMDK files (200MB being the capacity set on the previous line) unless the -cap command-line argument specifies otherwise.

Creating Remote Disk

As stated in "Support for Managed Disk" on page 42, VixDiskLib_Create() does not support managed disk. To create a managed disk on the remote ESXi host, first create a hosted disk on the local Workstation, then convert the hosted disk into managed disk with VixDiskLib_Clone() over the network.

To create remote managed disk using the sample program, type the following commands:

```
./vix-disklib-sample -create -cap 1000000 virtdisk.vmdk
./vix-disklib-sample -clone virtdisk.vmdk -host esx3i -user root -password secret vmfsdisk.vmdk
You could write a virtual-machine provisioning application to perform the following steps:
```

- 1 Create a hosted disk VMDK with 2GB capacity, using VixDiskLib_Create().
- 2 Write image of the guest OS and application software into the VMDK, using VixDiskLib_Write().
- 3 Clone the hosted disk VMDK onto the VMFS file system of the ESXi host.

In this call, appGlobals.connection and appGolbals.diskPath represent the remote VMDK on the ESXi host, while srcConnection and appGlobals.srcPath represent the local hosted VMDK.

4 Power on the new guest OS to get a new virtual machine.

On Workstation, the VixVMPowerOn() function in the VIX API does this. For ESXi hosts, you must use the PowerOnVM_Task method. As easy way to use this method is in the VMware vSphere Perl Toolkit, which has the PowerOnVM_Task() call (non-blocking), and the PowerOnVM() call (synchronous).

5 Provision and deploy the new virtual machine on the ESXi host.

Special Consideration for ESXi Hosts

No matter what virtual file type you create in Step 1, it becomes type VIXDISKLIB_DISK_VMFS_FLAT in Step 3.

VMDK File Versions

Virtual disk programs must be able to cope with VMDK files up to version three (3).

Version 1 was the initial version of VMDK. All released builds of vixDiskLib can read and write this version.

Version 2 added disk encryption for hosted products (Workstation and Fusion), although encrypted disks were never implemented on ESXi. Version 2 VMDK files can be transferred to and appear on ESXi, where they are treated like version 1 VMDK files.

Version 3 added support for persistent changed block tracking (CBT), and is set when CBT is enabled for a virtual disk. CBT is supported on VMFS datastores. This version 3 first appeared in ESXi 4.0 and continues unchanged in recent vSphere releases. When CBT is enabled, the version number is incremented, and decremented when CBT is disabled.

If you look at the .vmdk descriptor file for a version 3 virtual disk, you can see a pointer to its *-ctk.vmdk ancillary file. For example:

```
version=3
...
# Change Tracking File
changeTrackPath="Windows-2008R2x64-2-ctk.vmdk"
```

The changeTrackPath setting references a file that describes changed areas on the virtual disk.

If you want to back up the changed area information, then your software should copy the *-ctk.vmdk file and preserve the "Change Tracking File" line in the .vmdk descriptor file. If you do not want to back up the changed area information, then you can discard the ancillary file, remove the "Change Tracking File" line, read the VMDK file data as if it were version 1, and roll back the version number on restore.

Working with Virtual Disk Data

The virtual disk library reads and writes sectors of data. It has no interface for character or byte-oriented I/O.

Reading and Writing Local Disk

Demonstrating random I/O, this function reads a sector at a time backwards through a VMDK. If it sees the string "VmWare" it substitutes the string "VMware" in its place and writes the sector back to VMDK.

```
#include <string>
static void DoEdit(void)/
   VixDisk disk(appGlobals.connection, appGlobals.diskPath, appGlobals.openFlags);
   uint8 buf[VIXDISKLIB_SECTOR_SIZE];
   VixDiskLibSectorType i;
   string str;
    for (i = appGlobals.numSectors; i >= 0; i--) {
       VixError vixError;
        vixError = VixDiskLib_Read(disk.Handle(), appGlobals.startSector + i, 1, buf);
       CHECK_AND_THROW(vixError);
        str = buf;
        if (pos = str.find("VmWare", 0)) {
            str.replace(pos, 5, "VMware");
           buf = str;
           vixError = VixDiskLib_Write(disk.Handle(), appGlobals.startSector + i, 1, buf);
            CHECK_AND_THROW(vixError);
       }
   }
}
```

Reading and Writing Remote Disk

The DoEdit() function is similar for remote managed virtual disk on ESXi hosts, but beforehand you must call VixDiskLib_Connect() with authentication credentials instead of passing NULL parameters.

Deleting a Disk (Unlink)

The function to delete virtual disk files is VixDiskLib_Unlink(). It takes two arguments: a connection and a VMDK filename.

```
vixError = VixDiskLib_Unlink(appGlobals.connection, appGlobals.diskPath);
```

Effects of Deleting a Virtual Disk

When you delete a VMDK, you lose all the information it contained. In most cases, the host operating system prevents you from doing this when a virtual machine is running. However, if you delete a VMDK with its virtual machine powered off, that guest OS becomes unbootable.

Renaming a Disk

The function to rename virtual disk files is VixDiskLib_Rename(). It takes two arguments: the old and the new VMDK filenames.

```
vixError = VixDiskLib_Rename(oldGlobals.diskpath, newGlobals.diskpath);
```

Effects of Renaming a Virtual Disk

The server expects VMDK files of its guest OS virtual machines to be in a predictable location. Any file accesses that occur during renaming might cause I/O failure and possibly cause a guest OS to fail.

Managing Child Disks

In the Virtual Disk API, redo logs are managed as a parent-child disk chain, each child being the redo log of disk changes made since its inception. Trying to write on the parent after creating a child results in an error. The library expects you to write on the child instead. See Figure 4-2, "Child Disks Created from Parent," on page 33 for a diagram.

Creating Redo Logs

A redo log is created by taking a virtual machine snapshot, which contains both disk data and virtual machine state. On hosted disk only, VixDiskLib_CreateChild() creates a redo log without virtual machine state.

You could write a simple application to create redo logs, or snapshots on managed disk, at 3:00 AM nightly. (although multiple snapshots have a performance impact). When you create a redo log while the virtual machine is running, the VMware host re-arranges file pointers so the primary VMDK, <vmname>.vmdk for example, keeps track of redo logs in the disk chain. Use the disk chain to re-create data for any given day.

To re-create data for any given day

- 1 Locate the <vmname>-<NNN>.vmdk redo log for the day in question.
 <NNN> is a sequence number. You can identify this redo log or snapshot by its timestamp.
- 2 Initialize the virtual disk library and open the redo log to obtain its parent handle.
- 3 Create a child disk with the VixDiskLib_Create() function, and attach it to the parent: vixError = VixDiskLib_Attach(parent.Handle(), child.Handle());
- 4 Read and write the virtual disk of the attached child.

This is just an example. On managed disk, multiple snapshots are not recommended for performance reasons. Backup software on vSphere usually takes a snapshot, saves data to backup media, then deletes the snapshot.

Virtual Disk in Snapshots

The Virtual Disk API provides the following features to deal with the disk component of snapshots:

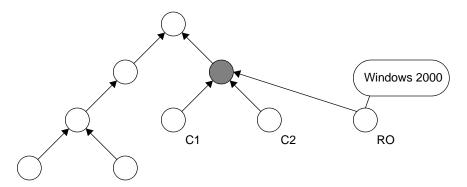
- Attaching an arbitrary child in a disk chain
- Opening read-only virtual disks
- Ability to open snapshot disk on ESXi hosts through VMware vCenter

Windows 2000 Read-Only File System

Another use of parent-child disk chaining is to create read-only access for Windows 2000, which has no option for mounting a read-only file system.

In Figure 6-1, the gray circle represents a virtual disk that must remain read-only because it has children. In this example, you want the Windows 2000 virtual machine to use that virtual disk, rather than the newer ones C1 and C2. Create new child disk RO, attach to the gray virtual disk as parent, and mount RO as the (mostly empty) read-only virtual disk of the Windows 2000 guest OS.

Figure 6-1. Attaching Virtual Read/Write Disk for Windows 2000



RDM Disks and Virtual BIOS

This section outlines low-level procedures for restoring raw device mapping (RDM) disks and NVRAM.

Restoring RDM Disks

Backing up and restoring RDM disks presents unusual challenges. The original backed-up RDM configuration might not apply, and is probably not appropriate, if users restore:

- A virtual machine to a different host or datastore.
- A virtual machine that was deleted, when its originally mapped RDM was also deleted, or the containing LUN was repurposed and rewritten.
- The RDM to a different virtual machine, even if that virtual machine is on the same host and datastore. Users might do this to access files on the disk, or to test a restore.

When performing a proxy backup of an RDM disk, you must present the same LUN ID to both the ESXi host and the proxy server. (This restriction does not apply to VMFS disk because the virtual disk library reads the VMFS header and matching UUID. But for RDM the host and proxy require the same LUN ID.)

Restoring RDM disks is appropriate if the original virtual machine's VMX file and disk mapping is no longer available, but the LUN containing the RDM is still available. In this case, the RDM image on the LUN might still be valid, so it does not need to be restored. If this is true, do not make changes to the RDM configuration during your restore operations. Instead, complete the restore process in two phases:

- Restore the virtual machine configuration (VMX) and system disk. This restores the virtual machine, but does not restore the RDM.
- Add the RDM disk to the virtual machine. After doing so, you can complete normal restore operations on the RDM disk.

Alternatively, it is possible to create a virtual machine that can host the RDM disk and access its contents. After you create the virtual machine, restore its virtual machine configuration (VMX) from backup, and then restore any selected disks.

Restoring the Virtual BIOS or UEFI

The .nvram file stores the BIOS or UEFI customizations of a virtual machine. Usually the only important items in this file are the boot drive setting and the boot order (in the case of multiple virtual disks).

Newer releases of vSphere can change the boot order using extended attribute settings, so boot order no longer must be stored in the .nvram file. However some users want to preserve a virtual machine's serial port settings in the .nvram file, and possibly other items, so applications should back up and restore this information.

To back up and restore NVRAM

- 1 For each virtual machine, make a separate copy of the .nvram file.
- 2 Back up each virtual machine using standard methods.
- 3 If necessary, restore the virtual machine using standard methods.
- 4 Overwrite the virtual machine's .nvram file with the saved copy of the original .nvram file.

IMPORTANT VMware now recommends saving the .nvram file as part of virtual machine backup, a change in recommendation since vSphere 4.1.

Interfacing With VMware vSphere

This section provides pointers to other vSphere programming interfaces.

The VIX API

The VIX API is a popular, easy-to-use developer interface for VMware Workstation, other hosted products, and ESXi. See the VMware developer documentation for information about the VIX API:

http://www.vmware.com/support/developer/vix-api

The *VIX API Reference* guide includes function reference pages for C++, Perl, and COM (component object model) for Microsoft C#, VBScript, and Visual Basic. Most reference pages include helpful code examples. Additionally, the vix-api Web guide includes examples for power on and off, suspending a virtual machine, taking a snapshot, guest operations, virtual machine discovery, and asynchronous callbacks.

Virus Scan all Hosted Disk

Suppose you want to run the antivirus software presented in "Scan VMDK for Virus Signatures" on page 49 for all virtual machines hosted on a VMware Workstation. Here is the high-level algorithm for an VIX-based application that would scan hosted disk on all virtual machines.

To virus scan hosted virtual disk

- 1 Write an application including both the Virtual Disk API and the VIX API.
- 2 Initialize the virtual disk library with VixDiskLib_Init().
- 3 Connect VIX to the Workstation host with VixHost_Connect().
- 4 Call VixHost_FindItems() with item-type (second argument) VIX_FIND_RUNNING_VMS.
 - This provides to a callback routine (fifth argument) the name of each virtual machine, one at a time. To derive the name of each virtual machine's disk, append ".vmdk" to the virtual machine name.
- 5 Write a callback function to open the virtual machine's VMDK.
 - Your callback function must be similar to the VixDiscoveryProc() callback function shown as an example on the VixHost_FindItems() page in the VIX API Reference Guide.
- Instead of printing "Found virtual machine" in the callback function, call the DoVirusScan() function shown in "Scan VMDK for Virus Signatures" on page 49.
- 7 Decontaminate any infected sectors that the virus scanner located.

The vSphere Web Services API

The VMware vSphere Web Services (WS) API is a developer interface for ESXi hosts and vCenter Server. See the VMware developer documentation for information about the vSphere WS API:

http://www.vmware.com/support/developer/vc-sdk

The *Developer's Setup Guide* for the VMware vSphere WS SDK has a chapter describing how to set up your programming environment for Microsoft C# or Java. Some of the information applies to C++ also.

The *Programming Guide* for the vSphere SDK contains some sample code written in Microsoft C# but most examples are written in Java, and based on the JAX-WS development framework.

ESXi hosts and the VMware vSphere WS API use a programming model based on Web services, in which clients generate Web services description language (WSDL) requests that pass over the network as XML messages encapsulated in simple object access protocol (SOAP). On ESXi hosts or vCenter Server, the vSphere layer answers client requests, usually passing back SOAP responses. This is a different programming model than the object-oriented function-call interface of C++ and the VIX API.

Virus Scan All Managed Disk

Suppose you want to run the antivirus software presented in "Scan VMDK for Virus Signatures" on page 49 for all virtual machines hosted on an ESXi host. Here is the high-level algorithm for a VMware vSphere solution that can scan managed disk on all virtual machines.

To virus scan managed virtual disk

- 1 Using the VMware vSphere Perl Toolkit, write a Perl script that connects to a given ESXi host.
- 2 Call Vim::find_entity_views() to find the inventory of every VirtualMachine.
- 3 Call Vim::get_inventory_path() to get the virtual disk name in its appropriate resource.
 The VMDK filename is available as diskPath in the GuestDiskInfo data object.
- 4 Using Perl's system(@cmd) call, run the extended vixDiskLibSample.exe program with -virus option. For ESXi hosts you must specify -host, -user, and -password options.
- 5 Decontaminate any infected sectors that the virus scanner located.

Read and Write VMDK Using vSphere API

The VMware vSphere Web Services API (version 2 and later) offers some methods to manage VMDK files. The VirtualDiskManager managed object defines more than a dozen methods similar to those in the VirtualDisk API documented here.

For more information, navigate to the latest VMware vSphere documentation center on the Web, search for VirtualDiskManager, and follow the specific link. At top of page you can click Local Methods to see a list of all methods defined by the VirtualDiskManager managed object.

VirtualDiskManager methods were not heavily used, and have not been updated to support object oriented file systems such as Virtual SAN (VSAN) or storage profile based management (SPBM).

Back Up First Class Disk (FCD)

First Class Disk (FCD) is a named virtual disk independent of a virtual machine. VDDK supports the backup of FCD in attached mode. This support is independent of transport mode.

VDDK does not support backing up a detached FCD. The workaround is to attach the FCD to a dummy virtual machine, which you then back up. The dummy can be a empty virtual machine without a guest OS.

To attach an FCD using the vSphere Web Client, click **Edit virtual machine settings > New device: Select** and add it as an Existing Hard Disk.

To attach an FCD using the vSphere API, you can call the $AttachDisk_Task$ method.

Backing Up Virtual Disks in vSphere

7

This chapter documents how to write backup and restore software for virtual machines running in vSphere, and contains the following sections about the vSphere Storage APIs – Data Protection (VADP):

- "Design and Implementation Overview" on page 57
- "Low Level Backup Procedures" on page 64
- "Low Level Restore Procedures" on page 73
- "Tips and Best Practices" on page 81
- "Windows Backup Implementations" on page 86

For an overview of backup, and help designing your top-level program structure, read the first section below. For details about implementing low-level backup code, read the remaining sections. You should be familiar with virtual machines, snapshots, ESXi, vCenter, and Java.

Design and Implementation Overview

On vSphere, backups are usually done by taking a snapshot, to efficiency obtain a static image of the virtual machine. Snapshots are a view of a virtual machine at a certain point in time, and enable quick and clean backup operation. Snapshots also provide an incremental backup mechanism called changed block tracking.

To back up virtual machines on vSphere, VMware recommends a two-language solution. First use Java to code the backup program that contacts the host, takes a temporary snapshot, records virtual machine configuration, and (later) deletes the snapshot. Then use C++ or C to code the VDDK program that transfers virtual disk data from the snapshot to backup media.

For restore, VMware recommends a two-language solution. First use Java to code the program that instructs the virtual machine to halt, or re-creates the target virtual machine from recorded configuration. Then use C or C++ to code the VDDK program that transfers saved data from backup media to virtual disk.

The Backup Process

These are the high-level steps to back up a virtual machine running in vSphere:

- Connect to the ESXi host containing the virtual machine targeted for backup.
 A side-effect of this step is determining the arrangement and description of virtual machines on the host.
- 2 Tell the host to take a snapshot of the target virtual machine, using the vSphere API. Use the quiesce flag, but not the memory flag, which is incompatible with quiesce. The virtual machine continues to run, while the snapshot provides a static (quiesced) view.
- 3 Capture the virtual disk data and virtual machine configuration information (vim.vm.ConfigInfo).
- 4 On the ESXi host, use the VDDK (programming in C or C++) to open and read the virtual disk and snapshot files. Copy them to backup media, along with configuration information.
- 5 Tell the host to delete the backup snapshot, using the vSphere API.

Communicating With the Server

In a typical vSphere deployment with many ESXi hosts, an instance of vCenter Server manages the ESXi hosts, and can move virtual machines from host to host (vMotion) to balance load and possibly save electricity by powering off an ESXi host. VMware therefore recommends that backup applications communicate with the vCenter Server instead of with individual ESXi hosts.

The vCenter Server provides location transparency for vSphere Web Services developers. The vCenter Server tracks virtual machines as they move (through vMotion) from one ESXi host to another, and vCenter Server directs SDK operations to the ESXi host that currently runs a virtual machine. Using the vSphere Web Services API, it is possible to back up all the virtual disks associated with a virtual machine.

The handling of the vCenter or an individual ESXi host is essentially equivalent when using the vSphere SDK. With vCenter management, there is no need to contact individual ESXi hosts directly. The remainder of this chapter uses the term vSphere to indicate either a vCenter Server or an ESXi host.

To reduce the resources used by vSphere, VMware recommends that the number of connections (or Sessions) be minimized. It is in the best interests of any program that communicates with vSphere to create one Session and share it with all elements of the program that need to exchange information with vSphere. This means that if your program supports multiple threads, your program should multiplex the use of connection objects by use of access control locks (mutex and the like).

It is also important that all vSphere SDK operations proceed from one instance of the "Session" object that your application requests after logging into vSphere. Using the vSphere API your application can create objects that are "Session specific" and therefore would not be known to other portions of your application that might use a different Session.

Information Containers as Managed Objects

VMware documentation introduces you to the concept of the managed object and its handle, called a managed object reference (moRef). You might be tempted to get configuration and status information of managed objects using a piecemeal approach. This has the severe disadvantage of creating a lot of chatter over the server connection, so it is very slow. A mechanism has been created to provide status information efficiently: the PropertyCollector, discussed in "PropertyCollector Data" on page 59.

More About Managed Objects

The documentation for the vSphere API and object model introduces a large number of managed objects. There are five basic types of managed objects that describe the organization of a server. Other managed objects can be considered as details expanding on these five basic types:

- Folder
- Datacenter
- ComputeResource
- ResourcePool
- VirtualMachine

It is a characteristic of all managed objects that they have a moRef to the managed object that serves as the parent to the managed object. This parent moRef allows you to reconstruct the object hierarchy exposed by the vSphere SDK. In general the hierarchy is a tree-like structure along the lines of:

Root Folder > Datacenter > ComputeResource > ResourcePool > VirtualMachine

There are variations on this theme, depending on whether you connect to vCenter or directly to an ESXi host, but the overall organization is like the structure above. Each managed object also has a Name property.

The virtual machine that you want to back up, and the snapshot you take of it (the extensible managed object VirtualMachineSnapshot) are both designated by their moRef.

Managed Object References

A managed object reference (moRef) is actually a handle and not the managed object itself. While it is certain that a moRef always contains a unique value, the unique value is only relative to the instance of vSphere to which you are connected. For example, if vCenter Server manages a cluster of ESXi hosts, each ESXi host maintains its own managed object reference namespace and the vCenter must maintain a managed object reference namespace representing all of its servers. So when an ESXi host is represented by a vCenter, the vCenter must ensure that the managed object references are unique. The vCenter accomplishes this by creating unique managed object reference names inside its own namespace, which differ from the names that ESXi uses for the same managed objects.

A vSphere instance (vCenter or ESXi) tries to keep the moRef for a virtual machine consistent across sessions, however consistency is not guaranteed. For example, unregistering and reregistering a virtual machine could result in a change to the moRef for the virtual machine. Thus, it is a bad idea to store a moRef and expect it to work correctly in future sessions, or with a different vCenter Server.

Unique ID for a Different vCenter

On one vCenter Server, the moRef uniquely identifies a virtual machine. If you need to track and inventory virtual machine backups across multiple vCenter Servers, you can use moRef together with instanceUuid. You can see the instanceUuid at the following browser path:

https://<vcserver>/mob/?moid=ServiceInstance&doPath=content.about

For direct connections to ESXi, the host address and moRef uniquely identify a virtual machine. However this moRef could be different from the one that vCenter Server returns, hence the fallback to instanceUuid. The instanceUuid was new in VMware vSphere 4.0. In previous releases, the fallback was to Uuid.

Gathering Status and Configuration Information

To save configuration of a virtual machine so you can restore it later, you can use the PropertyCollector to get the virtual machine configuration.

The PropertyCollector is the most efficient mechanism to specify, at the top level, all of the managed objects that are of interest to your application. It has methods for providing updates that indicate only changes to the previous state of these objects. There are two mechanisms for acquiring these updates:

- Polling Check for changes. The result is either "no change" or an object containing the changes. One
 advantage of this mechanism is that it involves no network traffic except for a poll request and reporting.
- Wait for updates "Wait for updates" is basically a blocking call to the PropertyCollector. This is only useful if you dedicate a program thread waiting for the call to unblock. The advantage of this mechanism is that there is no traffic on the communications thread unless something must be reported.

The PropertyCollector is powerful but requires great attention to detail. Backup-related features of the PropertyCollector are covered in "Low Level Backup Procedures" on page 64 of this document. The next section provides some background about PropertyCollector.

PropertyCollector Data

This document assumes that you want to keep up with changes in the configuration of the vCenter Server, and therefore plan to use the update tracking capability of the PropertyCollector.

The PropertyCollector requires two fairly complex arguments: the PropertySpec and the ObjectSpec. The ObjectSpec contains instructions to the PropertyCollector describing where to look for the desired data. Because configuration information in vSphere is organized like a directory tree, the ObjectSpec must describe how to traverse the tree to obtain the desired information. The net result is a complex, nested, and recursive list of instructions. Fortunately, once you have determined the location of all the desired information, the ObjectSpec needed to determine the layout of a vSphere object hierarchy can be a static unvarying object. See the code example in section "Understanding an ObjectSpec" on page 64.

The PropertySpec is a list of desired property information. Formulating a list that includes all of the desired information can take some effort to compile, but once determined, this can be a static object also.

The data returned from the PropertyCollector is a container class called PropertyFilterUpdate, which contains an objectSet with an item-by-item list of changes to object properties. Every item in this container is identified with one of the following keys: enter (add), leave (delete), and modify. On the first data request, every data item is included, and "enter" is marked for every data item.

The PropertyCollector presents its results in what amounts to random order. Since all managed objects have a "parent" property, you can reconstruct the configuration hierarchy by building a tree in memory, using the parent identification to organize. The root folder is identified as the only folder without a parent.

Useful Property Information

In the data returned from PropertyCollector, you can find most of the information that is useful for backup in the Virtual Machine managed object, including the following:

- Virtual Disks names, types, and capacities.
- Virtual Machine Type and Configuration Whatever would be useful in (re)creating a virtual machine. This list might include such information as memory size and number of CPUs.
- Display Names These names appear in VMware products such as the vSphere Client. You should keep track of these names and correlate them for consistency between your product and VMware products.

VMware supports many virtual disk implementations. The disk implementation type is important because:

- On restore, you should re-create virtual disk with the same disk type as the original virtual machine used.
- A disk backed by a pass-through raw device mapping (RDM) mostly bypasses the ESXi storage stack. You cannot make a snapshot of this virtual disk type. Therefore, you cannot back up pass-through RDM disk using the snapshot method described in this document.

For more information about the Java APIs, read the first several chapters of the *VMware vSphere Web Services SDK Programming Guide*, and related pages of the Web-based *VMware vSphere API Reference Documentation*. Both are available at http://www.vmware.com/support/developer/vc-sdk. Examples in this chapter assume that you have set up the vSphere SDK as described in documentation.

Doing a Backup Operation

After your program obtains information about what is available to back up, it can perform a backup. The three steps to the backup process are:

- "Create a Temporary Snapshot on the Target Virtual Machine" on page 60
- "Extract Backup Data from the Target Virtual Machine" on page 61, and save configuration information.
- "Delete the Temporary Snapshot" on page 61

Prerequisites for Backup

To complete a backup, the calling program requires the permissions shown in Table 7-1.

Table 7-1. Required Permissions to Complete a Backup

Privilege Category	Privilege Subcategory	Privilege
Virtual Machine	Provisioning	Allow Virtual Machine Download
	State	Create Snapshot
		Remove Snapshot
	Configuration	Disk Lease

Create a Temporary Snapshot on the Target Virtual Machine

The low-level procedure for creating a snapshot of a virtual machine is documented in the section "Creating a Snapshot" on page 69. Set the quiesce flag True to make the file system quiescent, otherwise the snapshot might represent a transitional system state, with inconsistent data. Restoring such data might be destructive.

Another flag named memory allows you to include in the snapshot a dump of the powered on virtual machine's in-memory state. This is not needed for backup, so set this flag to False.

Changed Block Tracking

This feature, first available in vSphere 4, provides the foundation for incremental (or differential) backup of virtual disks. Your application can back up only changed data as indicated by the QueryChangedDiskAreas method. Virtual machines with virtual hardware version 7 and later support changed block tracking. These virtual machines contain changeTrackingSupported in the capability field of the VirtualMachine managed object. See "Changed Block Tracking on Virtual Disks" on page 70 for details.

Extract Backup Data from the Target Virtual Machine

Associated with the snapshot you just created are "versions" of the virtual disks. To identify these disks, you obtain a moRef to the snapshot you just created. From this snapshot moRef, you can extract the disk names and paths. How to do this is demonstrated in section "Backing Up a Virtual Disk" on page 69.

To read the data in a virtual disk, it is necessary to use the VixDiskLib. This library isolates the programmer from the details of extracting data from a virtual disk and its redo logs. For example, when doing backup you call functions VixDiskLib_Open() and VixDiskLib_Read(), among others. VixDiskLib allows access to disk data on sector boundaries only; the transfer size is some multiple of the disk sector size.

When accessing disks on ESXi hosts, VixDiskLib release 1.0 transferred virtual disk data over the network. Later VixDiskLib releases contain API enhancements so you can request more efficient data paths, such as direct SAN access or HotAdding disks to a virtual backup appliance. These efficient data paths requires minor code changes, such as calling VixDiskLib_ConnectEx() instead of plain connect.

Part of virtual disk information is metadata: a number of key/value pairs describing configuration of the virtual disk. Metadata information can be extracted from a virtual disk using the VixDiskLib functions VixDiskLib_GetMetadataKeys() and VixDiskLib_ReadMetadata(). You should save metadata keys along with the backup, in case you need to re-create the virtual disk.

The VixDiskLib API allows a backup application to perform a full backup of a virtual machine. The newer VixMntapi library can extract information about a guest operating system from its virtual disks, so your backup application can determine the type of operating system that is involved. This allows mounting the volumes to device nodes, so your application can perform file-oriented backups and restores.

Delete the Temporary Snapshot

As the last part of the backup process, you should delete the temporary snapshot. It is no longer needed, worsens virtual machine performance, and takes up storage space that could be put to better use.

The Restore Process

Your software can follow one of two restore scenarios: either revert to a saved state, or disaster recovery:

To bring an existing virtual machine to a previous state

- 1 Connect to the server and command it to halt and power off the target virtual machine.
- 2 Use the server to gain access to the virtual disks. With SAN transport (but not HotAdd or NBDSSL) you must create a snapshot before restoring data.
- 3 Transfer the disk images from backup using VixDiskLib. Revert-to and delete the snapshot, if created.

To completely re-create a virtual machine (disaster recovery)

- 1 Connect to the server.
- 2 Command the server to create a new virtual machine and its virtual disks using the configuration information saved from vim.vm.ConfigInfo during backup.
- 3 Transfer virtual disk data to the newly created virtual disks using VixDiskLib. Virtual disk data includes disk formatting information, so you do not need to build any kind of file system on the virtual disks.

Doing a Restore Operation

The two scenarios of restore operation are described below.

- "Restoring an Existing Virtual Machine to a Previous State" on page 62
- "Creating a New Virtual Machine" on page 63

Prerequisites for Restore

To complete a restore, the calling process requires the permissions in Table 7-2.

Table 7-2. Required permissions to complete a restore

Privilege Category	Privilege Subcategory	Privilege
Virtual Machine	Inventory	Create
		Remove
	Configuration	Settings
		Change Resource
Resource		Assign Virtual Machine to Resource Pool

For security reasons, programs are not granted write access to the disks of a running virtual machine. Before you shut it down, you should determine the run-state of the virtual machine.

Run-state information is available from the PropertyCollector, and if you keep this information up-to-date, your application already knows the run-state of the virtual machine. To change the run-state you must have the moRef of the virtual machine. Use this moRef in a PowerOnVM_Task call through the server connection. For virtual machine shutdown, call the PowerOffVM_Task method.

Restoring an Existing Virtual Machine to a Previous State

The following steps restore a virtual machine to a certain saved state:

- 1 Shut down the virtual machine (if it is not already shut down).
- With SAN advanced transport mode, you must create a snapshot before restoring virtual machine data. The restore snapshot is optional but not necessary with HotAdd and NBDSSL transport modes. With a VVol datastore, if a snapshot was taken, writes must go to the leaf disk only.
- 3 Restore contents of the virtual disk(s). If there were no pre-existing snapshots at backup time, just the snapshot just created, restore only the base disks.

Restoring disk data requires that you obtain the current names of virtual disks. This process is similar to the one described in "Extract Backup Data from the Target Virtual Machine" on page 61, except in this case you obtain this information directly from the virtual machine and not from a snapshot. The target for the saved disk data must be the actual disk name (including any sequence number) because the current incarnation of a virtual machine may be derived from one or more snapshots.

Restoring disk data requires use of VixDiskLib. The VixDiskLib_Write() function opens the virtual disks so your program can write data. VixDiskLib functions transfer data to even-sector boundaries only, and transfer length must be an even multiple of the sector size.

The virtual disk already exists, so it is not necessary to restore the disk configuration information mentioned in "Extract Backup Data from the Target Virtual Machine" on page 61.

4 With SAN transport mode, revert-to and delete the snapshot that you created in Step 2. Failing to perform this step with SAN mode could yield a virtual machine that cannot be powered on.

Creating a New Virtual Machine

The process of building a virtual machine from backup data involves the following steps:

1 Create the virtual machine.

To create a new virtual machine, you use the information about virtual machine configuration that you derived and saved during the backup process. You might allow users of restore software an opportunity to rename the virtual machine during recovery in case they want to clone or move the virtual machine. Also you might consider offering them an opportunity to change virtual machine layout (for instance, storing virtual disks on a different datastore). Creating the virtual disks is also done at the time when you create the virtual machine. This process is fairly complicated. See the section "Low Level Backup Procedures" on page 64 for details.

2 Restore the virtual disk data.

This process is similar to restoring the contents of virtual disks (Step 3 under "Restoring an Existing Virtual Machine to a Previous State" on page 62) with the following exception: you must call the VixDiskLib_WriteMetadata() function to write all the disk configuration key/value data into the virtual disk before restoring any backed-up data to the virtual disk. Then call VixDiskLib_Write() to restore the virtual disk data, as described in Step 3 above.

3 Power on the virtual machine.

Accessing Files on Virtual Disks

It might be necessary for a backup application to access individual files or groups of files on the virtual disks. For example, data protection applications might need to restore individual files on demand.

You can find the interfaces to accomplish this in the VixMntapi library associated with VixDiskLib. The VixMntapi library allows disks or volumes of a virtual machine to be mounted and examined as needed. VixMntapi provides access at the file system level, whereas VixDiskLib provides access at the sector level.

To mount a virtual disk

- 1 Locate the path names of all the virtual disks associated with a snapshot.
- 2 Call VixDiskLib_Open() to open all of these virtual disks. This gives you a number of VixDiskLib handles, which you should store in an array.
- 3 Call VixMntapi_OpenDiskSet() to create a VixDiskSetHandle, passing in the array of VixDiskLib handles that you created in step 2.
- 4 Pass VixDiskSetHandle as a parameter to VixMntapi_GetVolumeHandles() to obtain an array of VixVolumeHandle pointers to all volumes in the disk set.
- 5 Call VixMntapi_GetOsInfo() to determine what kind of operating system is involved, and decide where important pieces of information are to be found.
- 6 For important volumes, call VixMntapi_MountVolume() then VixMntapi_GetVolumeInfo(), which reveals how the volume is set up. (Unimportant volumes include swap partitions.)
- 7 If you need information about how the guest operating system sees the data on this volume, you can look in the data structure VixVolumeInfo returned by VixMntapi_GetVolumeInfo(). For example, VixVolumeInfo::symbolicLink, obtained using VixMntapi_GetVolumeInfo(), is the path on the proxy where you can access the virtual disk's file system using ordinary open, read, and write calls.

Once you are done accessing files in a mounted volume, there are VixMntapi procedures for taking down the abstraction that you created. These calls are:

- VixMntapi_DismountVolume() for each volume handle
- VixMntapi_FreeOsInfo() and VixMntapi_FreeVolumeInfo()
- VixMntapi_CloseDiskSet()

This leaves the VixDiskLib handles that you obtained in the beginning; you must dispose of them properly.

More VADP Details

The preceding sections explained how to contact vSphere and extract information from it, and how to back up or restore virtual disks. The following sections cover the same information at a lower level.

Low Level Backup Procedures

This section describes low level details that may be helpful in coding a backup application. It is not the intent of this material to impose a design, but only to serve as a guideline with examples and exposition. The code samples provided below are not complete. They generally lack error handling and ignore critical details.

Communicating with the Server

Connections to the server machine require credentials: user name, password, and host name (or IP address). The following code connects to the server and extracts information useful for manipulating a service:

1 Create the service instance moRef:

```
ManagedObjectReference svcRef = new ManagedObjectReference();
svcRef.setType("ServiceInstance");
svcRef.setValue("ServiceInstance");
Locate the service:

VimServiceLocator locator = new VimServiceLocator();
locator.setMaintainSession(true);
VimPortType serviceConnection = locator.getVimPort("https://your_server/sdk");
Log in to the session manager:
ServiceInstanceContent serviceContent = serviceConnection.retrieveContent(svcRef);
ManagedObjectReference sessionManager = serviceInstance.getSessionManager();
UserSession us = serviceConnection.login(sessionManager, username, password, null);
```

The PropertyCollector

The PropertyCollector is used in this section to apply the above details to the backup task.

PropertyCollector Arguments

The PropertyCollector uses two relatively complicated argument structures. As was mentioned in "PropertyCollector Data" on page 59, these arguments are PropertySpec and ObjectSpec. PropertySpec is a list of the information desired, and ObjectSpec is a list of instructions indicating where to find the information. In theory, you could directly address an object using its moRef. In that case an ObjectSpec can be very simple. However, getting the initial moRef can be a challenge when a complicated ObjectSpec is required. To formulate a complex ObjectSpec, you need to understand the structure of the available data. This is complicated by the fact that an ObjectSpec can contain recursive elements.

Understanding an ObjectSpec

An ObjectSpec is a list of ObjectSpec elements, each specifying an object type, and giving a "selection spec" for the object. "More About Managed Objects" on page 58 describes five types of managed objects: Folder, Datacenter, ComputeResource, ResourcePool, and VirtualMachine. VirtualApp (vApp) is a sixth type. You can "traverse" objects, because one managed object leads to another.

- Folder One of the items contained in the Folder is called childEntity, which is a list of moRefs that can contain one of the five managed object types. A Folder can be parent to any of these managed objects.
- Datacenter This managed object has two items that lead to other managed objects:
 - hostFolder A moRef to a Folder containing a list of ComputeResources comprising a Datacenter.
 - vmFolder A moRef to a Folder containing the VirtualMachines that are part of the Datacenter. If it is your objective to duplicate the display seen in a vSphere Client GUI, then this Folder is of limited use because it does not describe the ResourcePool that is the parent of a virtual machine.

- ComputeResource A ComputeResource is basically hardware. A ComputeResource can comprise multiple systems. The hardware represents resources that can be used to implement a VirtualMachine object. VirtualMachine is a child of ResourcePool, which controls the sharing of a physical machine's resources among VirtualMachine objects. A ComputeResource contains an item named resourcePool, which is a moRef to a ResourcePool.
- VirtualApp A VirtualApp (vApp) is a collection of VirtualMachines that make up a single application. This is a special form of ResourcePool (defined below). A VirtualApp may have three types of children:
 - VirtualMachine A folder named vm contains a list of moRefs to child VirtualMachines.
 - resourcePool A folder containing a list of moRefs pointing to child ResourcePools or VirtualApps.
 - VirtualApp A VirtualApp can be composed of other VirtualApps.
 - ResourcePool You can segment the resources of a VirtualApp using a ResourcePool.
- ResourcePool This managed object contains two child items:
 - resourcePool A folder containing a list of moRefs pointing to child ResourcePools or VirtualApps.
 - vm A list of moRefs to child VirtualMachines that employ the resources of the parent ResourcPool.
 A VirtualMachine always lists a ResourcePool as its parent.
- VirtualMachine The VirtualMachine is often considered an "end object" so you do not need to describe any traversal for this object.

The ObjectSpec does not have to lead you any farther than the moRef of a target object. You can gather information about the managed object itself using the moRef and the PropertySpec. This is described in detail in the section "Understanding a PropertySpec" on page 66.

A Traversal Spec extends Selection Spec, a property of Object Spec, and contains the following elements:

- Path The element contained in the object that is used to steer traversal.
- SelectSet An array containing either SelectionSpec or TraversalSpec elements.
- Skip Whether or not to filter the object in the Path element.
- Type The type of object being referenced.
- Name Optional name you can use to reference the TraversalSpec, inherited from SelectionSpec.

SelectionSpec is a direct target for traversal, as is TraversalSpec (a class extending SelectionSpec). It is in the SelectSet that recursion can occur.

If you wish to traverse the entire configuration tree for a server, then you need only the "root node" moRef, which is always a Folder. This root folder moRef is available in the property rootFolder of the ObjectSpec service instance content. All of the above goes into this Java code sample.

```
// Traversal objects can use a symbolic name.
// First we define the TraversalSpec objects used to fill in the ObjectSpec.
// This TraversalSpec traverses Datacenter to vmFolder
TraversalSpec dc2vmFolder = new TraversalSpec();
dc2vmFolder.setType("Datacenter"); // Type of object for this spec
dc2vmFolder.setPath("vmFolder"); // Property name defining the next object
dc2vmFolder.setSelectSet(new SelectionSpec[] {"folderTSpec"});
// This TraversalSpec traverses Datacenter to hostFolder
TraversalSpec dc2hostFolder = new TraversalSpec();
dc2hostFolder.setType("Datacenter");
dc2hostFolder.setPath("hostFolder");
// We use the symbolic name "folderTSpec" which will be defined when we create the folderTSpec.
dc2vmFolder.setSelectSet(new SelectionSpec[] {"folderTSpec"});
// This TraversalSpec traverses ComputeResource to resourcePool
TraversalSpec cr2resourcePool = new TraversalSpec();
cr2resourcePool.setType("ComputeResource");
```

```
cr2resourcePool.setPath("resourcePool");
// This TraversalSpec traverses ComputeResource to host
TraversalSpec cr2host = new TraversalSpec();
cr2host.setType("ComputeResource");
cr2host.setPath("host");
// This TraversalSpec traverses ResourcePool to resourcePool
TraversalSpec rp2rp = new TraversalSpec();
rp2rp.setType("ResourcePool");
rp2rp.setPath("resourcePool");
// Finally, we tie it all together with the Folder TraversalSpec
TraversalSpec folderTS = new TraversalSpec();
folderTS.setName{"folderTSpec"); // Used for symbolic reference
folderTS.setType("Folder");
folderTS.setPath("childEntity");
folderTS.setSelectSet(new SelectionSpec[]{ "folderTSpec",
                                            dc2vmFolder.
                                            dc2hostFolder,
                                            cr2resourcePool.
                                            rp2rp});
ObjectSpec ospec = new ObjectSpec();
ospec.setObj(startingPoint); // This is where you supply the starting moRef (usually root folder)
ospec.setSkip(Boolean.FALSE);
ospec.setSelectSet(folderTS); // Attach the TraversalSpec we designed above
```

Understanding a PropertySpec

A PropertySpec is a list of individual properties that can be found at places identified by the ObjectSpec and its TraversalSpec. Once the PropertyCollector has a moRef, it can then return the properties associated with that moRef. This can include "nested" properties. Nested properties are properties that can be found inside of properties identified at the top level of the managed object. Nested properties are identified by a "dot" notation.

An example of nested properties can be drawn from the VirtualMachine managed object. A VirtualMachine has the property identified as summary, which identifies a VirtualMachineSummary data object. The VirtualMachineSummary contains property config, which identifies a VirtualMachineConfigSummary data object. The VirtualMachineConfigSummary has a property called name, which is a string containing the display name of the VirtualMachine. You can access this name property using the summary.config.name string value. To address all the properties of the VirtualMachineConfigSummary object, you would use the summary.config string value.

The PropertyCollector requires an array of PropertySpec elements. Each element includes:

- Type The type of object that contains the enclosed list of properties.
- PathSet An array of strings containing names of properties to be returned, including nested properties.

It is necessary to add an element for each type of object that you wish to query for properties. The following is a code sample of a PropertySpec:

```
// This code demonstrates how to specify a PropertySpec for several types of target objects:
PropertySpec folderSp = new PropertySpec();
folderSp.setType("Folder");
folderSp.setAll(Boolean.FALSE);
folderSp.setPathSet(new String [] {"parent", "name"});
PropertySpec dcSp = new PropertySpec();
dcSp.setType("Datacenter");
dcSp.setAll(Boolean.FALSE):
dcSp.setPathSet(new String [] {"parent","name"});
PropertySpec rpSp = new PropertySpec();
rpSp.setType("ResourcePool");
rpSp.setAll(Boolean.FALSE);
rpSp.setPathSet(new String [] {"parent","name","vm"});
PropertySpec crSp = new PropertySpec();
crSp.setType("ComputeResource");
crSp.setAll(Boolean.FALSE);
```

Getting the Data from the PropertyCollector

Now that we have defined ObjectSpec and PropertySpec (the where and what), we need to put them into a FilterSpec that combines the two. An array of FilterSpec elements is passed to the PropertyCollector (the minimum number of elements is one). Two mechanisms can retrieve data from PropertyCollector:

- RetrieveProperties A one-time request for all of the desired properties. This can involve a lot of data, and has no refresh option. RetrievePropertiesEx has an additional options parameter.
- Update requests PropertyCollector update requests take two forms: polling and waiting (see below).

Requesting Updates

The update method is the way to keep properties up to date. In either Polling or Waiting, it is first necessary to register your FilterSpec array object with the PropertyCollector. You do this using the CreateFilter method, which sends a copy of your FilterSpec to the server. Unlike the RetrieveProperties method, FilterSpec is retained after CreateFilter operation. The following code shows how to set FilterSpec:

```
// We already showed examples of creating pspec and ospec in the examples above.
// The PropertyCollector wants an array of FilterSpec objects, so:
PropertyFilterSpec fs = new PropertyFilterSpec();
fs.setPropSet(pspec);
fs.setObjectSet(ospec);
PropertyFilterSpec [] fsa = new PropertyFilterSpec [] {fs};
ManagedObjectReference pcRef = serviceContent.getPropertyCollector();
// This next statement sends the filter to the server for reference by the PropertyCollector
ManagedObjectReference pFilter = serviceConnection.CreateFilter(pcRef, fsa, Boolean.FALSE);
```

If you wish to begin polling, you may then call the function CheckForUpdates, which on the first try (when it must contain an empty string for the version number) returns a complete dump of all the requested properties from all the eligible objects, along with a version number. Subsequent calls to CheckForUpdates must contain this version number to indicate to the PropertyCollector that you seek any changes that deviate from this version. The result is either a partial list containing only the changes from the previous version (including a new version number), or a return code indicating no data has changed. The following code sample shows how to check for updates:

```
String updateVersion = ""; // Start with no version
UpdateSet changeData = serviceConnection.CheckForUpdates(pcRef, updateVersion);
if (changeData != nil) {
    updateVersion = changeData.getVersion(); // Extract the version of the data set
}
// ...
// Get changes since the last version was sent.
UpdateSet latestData = serviceConnection.CheckForUpdates(pcRef, updateVersion);
```

If instead you wish to wait for updates to occur, you must create a task thread that blocks on the call WaitForUpdates. This task thread would return changes only as they occur and not at any other time. However if the request times out, you must renew it.

NOTE The order of property retrieval is not guaranteed. Multiple update requests may be needed.

Extracting Information from the Change Data

The data returned from CheckForUpdates (or WaitForUpdates) is an array of PropertyFilterUpdate entries. Since a PropertyFilterUpdate entry is very generic, here is some code showing how to extract information from the PropertyFilterUpdate.

```
// Extract the PropertyFilterUpdate set from the changeData
PropertyFilterUpdate [] updateSet = changeData.getFilterSet();
// There is one entry in the updateSet for each filter you registered with the PropertyCollector.
// Since we currently have only one filter, the array length should be one.
PropertyFilterUpdate myUpdate = updateSet[0];
ObjectUpdate [] changes = myUpdate.getObjectSet();
for (a = 0; a < changes.length; a++) {
    ObjectUpdate theObject = changes[a];
    String objName = theObject.getObj().getMoType().getName();
    // Must decide how to handle the value based on the name returned.
    // The only names returned are names found in the PropertySpec lists.
    // Get propertyName and value ...
}</pre>
```

Getting Specific Data

From time to time, you might need to get data that is relevant to a single item. In that case you can create a simple ObjectSpec including the moRef for the item of interest. The PropertySpec can then be set to obtain the properties you want, and you can use RetrieveProperties to get the data. Hopefully you can deduce moRef from a general examination of the properties, by searching for information from the rootFolder.

Identifying Virtual Disks for Backup and Restore

To back up a virtual machine, you first need to create a snapshot. Once the snapshot is created, you then need to identify the virtual disks associated with this snapshot. A virtual machine might have multiple snapshots associated with it. Each snapshot has a virtual "copy" of the virtual disks for the virtual machine. These copies are named with the base name of the disk, and a unique decimal number appended to the name. The format of the number is a hyphen character followed by a 6-digit zero-filled number. An example a disk copy name might be mydisk-NNNNNN.vmdk where NNNNNN would be some number like: 000032.

The vSphere API identifies virtual disk files by prefixing the datastore name onto the file system pathname and the filename: [storageN] myvmname/mydisk-NNNNNN.vmdk. The name in square brackets corresponds to the short name of the datastore that contains this virtual disk, while the remainder of the path string represents the location relative to the root of this datastore.

To get the name and characteristics of a virtual disk file, you use the PropertyCollector to select the property: config.hardware.device from a VirtualMachine managed object. This returns an array of virtual devices associated with a VirtualMachine or Snapshot. You must scan this list of devices to extract the list of virtual disks. All that is necessary is to see if each VirtualDevice entry extends to VirtualDisk. When you find such an entry, examine the BackingInfo property. You must extend the type of the backing property to one of the following, or a VirtualMachineSnapshot managed object:

- VirtualDiskFlatVer1BackingInfo
- VirtualDiskFlatVer2BackingInfo
- VirtualDiskRawDiskMappingVer1BackingInfo
- VirtualDiskSparseVer1BackingInfo
- VirtualDiskSparseVer2BackingInfo

It is important to know which backing type is in use in order to be able to re-create the Virtual Disk.It is also important to know that you cannot snapshot a disk of type VirtualDiskRawDiskMappingVer1BackingInfo, and therefore you cannot back up this type of Virtual Disk.

The properties of interest are the backing fileName and the VirtualDisk capacityInKB. Additionally, when change tracking is in place, you should also save the changeID.

Creating a Snapshot

Before performing a backup operation, you must create a snapshot of the target virtual machine. Both full and incremental backup rely on the snapshot in vSphere.

With SAN transport on VMFS volumes, the virtual machine should not have any pre-existing snapshots, so that reporting of in-use disk sectors will work. For details see "About Changed Block Tracking" on page 84.

As a best practice, you should search for and delete any pre-existing snapshots with the same name that you selected for the temporary snapshot. These snapshots are possibly remnants from failed backup attempts.

Within a specific snapshot, the names of virtual disk files (with extension .vmdk) can be modified with a zero-filled 6-digit decimal sequence number to ensure that the .vmdk files are uniquely named. Depending on whether or not the current virtual machine had a pre-existing snapshot, the disk name for a snapshot could have this format: <diskname>-<NNNNN>.vmdk. This unique name is no longer valid after the snapshot is destroyed, so any data for a snapshot disk should be stored in the backup program under its base disk name.

The following code sample shows how to create a snapshot on a specific virtual machine:

You can use the taskRef return value as a moRef to track progress of the snapshot operation. After successful completion, taskRef.info.result contains the moRef of the snapshot.

Backing Up a Virtual Disk

This section describes how to get data from the Virtual Disk after you have identified it. In order to access a virtual disk, you must use the VixDiskLib. The following code shows how to initialize the VixDiskLib and use it for accessing a virtual disk. All operations require a VixDiskLib connection to access virtual disk data. At the present time VixDiskLib is not implemented for the Java language, so this code is C++ language:

```
VixDiskLibConnectParams connectParams;
VixDiskLibConnection srcConnection;
connectParams.serverName = strdup("TargetServer");
connectParams.creds.uid.userName = strdup("root");
connectParams.creds.uid.password = strdup("yourPasswd");
connectParams.port = 902;
VixError vixError = VixDiskLib_Init(1, 0, &logFunc, &warnFunc, &panicFunc, libDir);
vixError = VixDiskLib_Connect(&connectParams, &srcConnection);
This next section of code shows how to open and read a specific virtual disk:
VixDiskLibHandle diskHandle;
vixError = VixDiskLib_Open(srcConnection, diskPath, flags, &diskHandle);
uint8 mybuffer[some_multiple_of_512];
vixError = VixDiskLib_Read(diskHandle, startSector, numSectors, &mybuffer);
// Also getting the disk metadata:
size_t requiredLength = 1;
char *buf = new char [1];
// This next operation fails, but updates "requiredLength" with the proper buffer size
vixError = VixDiskLib_GetMetadataKeys(diskHandle, buf, requiredLength, &requiredLength);
delete [] buf;
buf = new char[requiredLength]; // Create a large enough buffer
vixError = VixDiskLib_GetMetadataKeys(diskHandle, buf, requiredLength, NULL);
// And finally, close the diskHandle:
vixError = VixDiskLib_Close(diskHandle);
// And if you are completely done with the VixDiskLib
VixDiskLib_Disconnect(srcConnection);
VixDiskLib_Exit();
```

Deleting a Snapshot

When you are done performing a backup, you need to delete the temporary snapshot. You can get the moRef for the snapshot from taskRef.info.result as describe above for the create snapshot operation. The following Java code demonstrates how to delete the snapshot:

```
ManagedObjectReference removeSnapshotTask;
ManagedObjectReference snapshot; // Already initialized.
removeSnapshotTask = serviceConnection.getservice().removeSnapshot_Task(snapshot, Boolean FALSE);
```

Changed Block Tracking on Virtual Disks

On ESXi hosts release 4.0 and later, virtual machines can keep track of disk sectors that have changed. This is called changed block tracking. Its method in the VMware vSphere API is QueryChangedDiskAreas, which takes the following parameters:

- _this Managed object reference to the virtual machine.
- snapshot Managed object reference to a Snapshot of the virtual machine.
- deviceKey Virtual disk for which to compute the changes.
- startOffset Byte offset where to start computing changes to virtual disk. The length of virtual disk sector(s) examined is returned in DiskChangeInfo.
- changeId An identifier for the state of a virtual disk at a specific point in time. A new ChangeId results every time someone creates a snapshot. You should retain this value with the version of change data that you extract (using QueryChangedDiskAreas) from the snapshot's virtual disk.

When you back up a snapshot for the first time, ChangeId should be unset, or unsaved, indicating that a baseline (full) backup is required. If you have a saved ChangeId, it identifies the last time a backup was taken, and tells the changed block tracking logic to identify changes that have occurred since the time indicated by the saved ChangeId.

There are two ways to get this baseline backup:

- 1 Directly save the entire contents of the virtual disk.
- 2 Provide the special ChangeId "*" (star). The star indicates that QueryChangedDiskAreas should return only active portions of the virtual disk. For both thin provisioned (sparse) virtual disks and for ordinary virtual disks, this causes a substantial reduction in the amount of data to save.

To summarize, <code>changeID</code> is an identifier for a time in the past. It can be star "*" to identify all allocated areas of virtual disk, ignoring unallocated areas (of sparse disk), or it could be a <code>changeId</code> string saved at the time when a pre-backup snapshot was taken. It only makes sense to use the <code>specialChangeId = "*"</code> when no previous <code>ChangeId</code> exists. If a previous <code>ChangeId</code> does exist, then <code>QueryChangedDiskAreas</code> returns the disk sectors that changed since the new <code>ChangeId</code> was collected. Table 7-3 shows the algorithm.

Table 7-3. Use of Change ID for Changed Block Tracking

New Change ID	Old Change ID	Used for Query	Result
change 0	none	*	All in-use sectors of the disk.
change 1	change 0	change 0	All sectors altered since change 0.

The following restrictions are imposed on the "*" query when determining allocated areas of a virtual disk:

- The disk must be located on a VMFS volume (backing does not matter).
- The virtual machine must have had **no** (zero) snapshots when changed block tracking was enabled.

Enabling Changed Block Tracking

This feature is disabled by default, because it reduces performance by a small but measurable amount. If you query the virtual machine configuration, you can determine if it is capable of changed block tracking. Use the property collector to retrieve the capability field from the VirtualMachineManagedObject. If the capability field contains the flag changeTrackingSupported, then you can proceed. The virtual machine version must be 7 or higher to support this. If the virtual machine version is lower than 7, upgrade the virtual hardware.

If supported, you enable changed block tracking using an abbreviated form of VirtualMachineConfigSpec, then use the ReconfigVM_Task method to reconfigure the virtual machine with changed block tracking:

```
VirtualMachineConfigSpec configSpec = new VirtualMachineConfigSpec();
configSpec.changeTrackingEnabled = new Boolean(true);
ManagedObjectReference taskMoRef =
    serviceConnection.getService().ReconfiqVm_Task(targetVM_MoRef, configSpec);
```

Powered-on virtual machines must go through a stun-unstun cycle (triggered either by power on, migrate, resume after suspend, or snapshot create/delete/revert) before the virtual machine reconfiguration takes effect.

To enable changed block tracking with the vSphere Client:

- 1 Select the virtual machine and ensure that Summary > VM Version says "7" or higher compatibility.
- 2 In the Summary tab, click **Edit Settings > Options > Advanced > General**.
- 3 In the right side of the dialog box, click **Configuration Parameters...**
- In the new dialog box, locate or create a row with name ctkEnabled, and set its value to true not false. See above concerning the stun-unstun cycle.

To enable changed block tracking and back up with the VMware vSphere API:

- 1 Query change tracking status of the virtual machine. If false, activate changed block tracking. configSpec.changeTrackingEnabled = new Boolean(true);
- 2 Create a snapshot of the virtual machine. The snapshot operation causes a stun-unstun cycle.

```
CreateSnapshot_Task(VMmoRef, SnapshotName, Description, memory_files, quiesce_filesystem);
```

- 3 Starting from the snapshot's ConfigInfo, work your way to the BackingInfo of all virtual disks in the snapshot. This gives you the change IDs for all the disks of the virtual machine.
- 4 Hold onto the change IDs and do a full backup of the snapshot, since this is the first time for backup. VixDiskLib_Read(snapshotDiskHandle, startSector, numSectors, &buffer); /* C not Java */
- 5 Delete the snapshot when your backup has completed. removeSnapshot_Task(SnapshotName, Boolean FALSE);
- 6 Next time you back up this virtual machine, create a snapshot and use QueryChangedDiskAreas with the change IDs from your previous backup to take advantage of changed block tracking.

```
changes = theVM.queryChangedDiskAreas(SnapshotMoRef, diskDeviceKey, startPosition, changeId);
```

Gathering Changed Block Information

Associated with changed block tracking is changeId, an identifier for versions of changed block data. Whenever a virtual machine snapshot is created, associated with that snapshot is a changeId that functions as a landmark to identify changes in virtual disk data. So it follows that when a snapshot is created for the purpose of creating an initial virtual disk backup, the changeId associated with that snapshot can be used to retrieve changes that have occurred since snapshot creation.

To obtain the changeId associated with any disk in a snapshot, you examine the "hardware" array from the snapshot. Any item in the devices table that is of type vim.vm.device.VirtualDevice.VirtualDisk encloses a class describing the "backing storage" (obtained using getBacking) that implements virtual disk. If backing storage is one of the following disk types, you can use the changeId property of the BackingInfo data object to obtain the changeId:

```
vim.vm.device.VirtualDevice.VirtualDiskFlatVer2BackingInfo
vim.vm.device.VirtualDevice.VirtualDiskSparseVer2BackingInfo
vim.vm.device.VirtualDevice.VirtualDiskRawDiskMappingVer1BackingInfo
vim.vm.device.VirtualDevice.VirtualDiskRawDiskVer2BackingInfo
```

Information returned by the QueryChangedDiskAreas method is a DiskChangeInfo data object containing an array of DiskChangeInfo.DiskChangeExtent items that enumerate the start offset and length of various disk areas that changed, and the length and start offset of the entire disk area covered by DiskChangeInfo.

When using QueryChangedDiskAreas to gather information about snapshots, enable change tracking before taking a snapshot. Attempts to collect information about changes that occurred before change tracking was enabled result in a FileFault error. Enabling change tracking provides the additional benefit of saving space because it enables backup of only information that has changed. If change tracking is not enabled, the entire virtual machine must be backed up each time, rather than incrementally.

Changed block tracking is supported whenever the I/O operations are processed by the ESXi storage stack:

- For a virtual disk stored on VMFS, no matter what backs the VMFS volume (SAN or local disk).
- For a virtual disk stored on NFS (though thick vs thin might be an issue).
- For an RDM in virtual compatibility mode.

When I/O operations are not processed by the ESXi storage stack, changed block tracking is not usable:

- For an RDM in physical compatibility mode.
- A disk that is accessed directly from inside a VM. For example if you are running an iSCSI initiator within the virtual machine to access an iSCSI LUN from inside the VM, vSphere cannot track it.

If the guest actually wrote to each block of a virtual disk (long format or secure erase), or if the virtual disk is thick and eager zeroed, or cloned thick disk, then the "*" query reports the entire disk as being in use.

To find change information, you can use the managed object browser at http://<ESXhost>/mob to follow path content > rootFolder > datacenter > datastore > vm > snapshot > config > hardware > virtualDisk > backing. Changed block tracking information (changeId) appears in the BackingInfo.

The following C++ code sample assumes that, in the past, you obtained a complete copy of the virtual disk, and at the time when the changeId associated with the snapshot was collected, you stored it for use at a later time, which is now. A new snapshot has been created, and the appropriate moRef is available:

In the above code, QueryChangedDiskAreas is called repeatedly, as position moves through the virtual disk. This is because the number of entries in the ChangedDiskArea array could occupy a large amount of memory for describing changes to a large virtual disk. Some disk areas may have no changes for a given changeId.

The changeId (changed block ID) contains a sequence number in the form <UUID>/<nnn>. If <UUID> changes, it indicates that tracking information has become invalid, necessitating a full backup. Otherwise incremental backups can continue in the usual pattern.

Troubleshooting

If you reconfigure a virtual machine to set changeTrackingEnabled, but the property remains false, check that you have queried the virtual machine status with VirtualMachine->config() after reconfiguration with VirtualMachine->reconfigure() and not before. Also make sure that virtual machine compatibility is hardware version 7 or higher, and that it has undergone a stun-unstun cycle since reconfiguration.

Limitations on Changed Block Tracking

Changed block tracking does not work if the virtual hardware version is 6 or earlier, in physical compatibility RDM mode, or when the virtual disk is attached to a shared virtual SCSI bus. ESXi 3.5 supported only up to virtual hardware version 4.

Changed block tracking can be enabled on virtual machines that have disks like these, but when queried for their change ID, these disks always return an empty string. So if you have a virtual machine with a regular system disk and a pass-through RDM as a data disk, you can track changes only on the system disk.

Checking for Namespace

You can avoid using the queryChangedDiskAreas API on ESXi 3.5 based storage by parsing XML files for the namespace. For prepackaged methods that do this, see these SDK code samples:

Axis/java/com/vmware/samples/version/displaynewpropertieshost/DisplayNewPropertiesHostV25.java
Axis/java/com/vmware/samples/version/getvirtualdiskfiles/GetVirtualDiskFilesV25.java
DotNet/cs/DisplayNewProperties/DisplayNewPropertiesV25.cs
DotNet/cs/GetVirtualDiskFiles/GetVirtualDiskFilesV25.cs

Low Level Restore Procedures

The following sections describe how to recover virtual machines and restore virtual disk data.

- "Restoring a Virtual Machine and Disk" on page 73
- "Restore of Incremental Backup Data" on page 81

Restoring a Virtual Machine and Disk

You cannot get write access to a virtual disk that is in active use. For a full restore, you first must ensure that the virtual disk is not in use by halting the parent virtual machine, then performing the "power off" sequence. The following code sample demonstrates how to "power off" a Virtual Machine:

```
// At this point we assume that you have a ManagedObjectReference to the VM - vmMoRef.
// Power on would need a ManagedObjectReference to the host running the VM - hostMoRef.
ManagedObjectReference taskRef = serviceConnection.powerOffVm(vmMoRef);
```

With SAN transport mode, you must create a snapshot of the virtual machine before virtual disk restore. See "Creating a Snapshot" on page 69. If at restore time the virtual machine had a pre-existing snapshot, you must delete it, otherwise SAN mode restore will fail. For other transport modes, the restore snapshot is optional.

With a VVol datastore, the restore snapshot is forbidden when restoring to the parent (backing disk). A null MoRef is allowed as of vSphere 6.0 to minimize code changes when applications restore to a VVol datastore. To get the unique ID of a VVol object in the MOB (managed object browser) click VirtualMachine > Config > Hardware > Device (VirtualDisk) > Backing.

Table 7-4 summarizes the snapshot requirements.

Table 7-4. Summary of restore snapshot requirements

	NBDSSL	HotAdd	SAN
Non-VVol datastore	A snapshot is optional. If a snapshot is used, writing to either the leaf disk or the parent disk is supported. If writing to the parent disk, the snapshot must be reverted and then deleted.		A snapshot is required. Writing to the leaf disk is not supported, and writes must only go to the base disk. The snapshot must be reverted and then deleted.
VVol datastore	A snapshot is optional. If a snapshot is used, writing must be to the leaf disk only.		Not applicable, because SAN transport is not supported on VVol datastores.

In this phase you use VixDiskLib to reload contents of the Virtual Disk, so the following code is C++ not Java:

```
// At this point we assume that you already have a VixDiskLib connection to the server machine.
uint8 mybuffer[some_multiple_of_512];
int mylocalfile = open("localfile", openflags); // Contains backup copy of virtual disk.
read(mylocalfile, mybuffer, sizeof mybuffer);
vixError = VixDiskLib_Open(srcConnection, path, flags, &diskHandle);
VixDiskLib_Write(diskHandle, startsector, (sizeof mybuffer) / 512, mybuffer);
```

With SAN transport mode, you must revert-to and delete the snapshot. If you forget the snapshot revert, snapshot delete will fail due to CID mismatch, so the virtual machine cannot be powered on. If you forget the snapshot delete, the extraneous snapshot will cause restore problems for subsequent backups.

Creating a Virtual Machine

This section shows how to create a VirtualMachine object, which is complicated but necessary so you can restore data into it. Before creating this object, you must create a VirtualMachineConfigSpec describing the virtual machine and all of its supporting virtual devices. Almost all the required information is available from the virtual machine property config.hardware.device, which is a table containing the device configuration information. The relationships between devices are described by the value key, which is a unique identifier for the device. In turn, each device has a controllerKey, which is the key identifier of the controller where the device is connected. Use negative integers as temporary key values in the VirtualMachineConfigSpec to guarantee that temporary key numbers do not conflict with real key numbers when they are assigned by the server. When associating virtual devices with default devices, the controllerKey property should be reset with the key property of the controller. Below are the settings for a sample VirtualMachineConfigSpec used to create a virtual machine.

```
// beginning of VirtualMachineConfigSpec, ends several pages later
  dynamicType = <unset>,
   changeVersion = <unset>,
//This is the display name of the VM
  name = "My New VM",
  version = "vmx-04".
  uuid = <unset>,
  instanceUuid = <unset>,
  npivWorldWideNameType = <unset>,
  npivDesiredNodeWwns = <unset>,
  npivDesiredPortWwns = <unset>.
  npivTemporaryDisabled = <unset>,
  npivOnNonRdmDisks = <unset>,
  npivWorldWideNameOp = <unset>,
  locationId = <unset>,
// This is advisory, the disk determines the O/S
guestId = "winNetStandardGuest",
  alternateGuestName = "Microsoft Windows Server 2008, Enterprise Edition",
  annotation = <unset>,
   files = (vim.vm.FileInfo) {
      dynamicType = <unset>,
      vmPathName = "[plat004-local]",
     snapshotDirectory = "[plat004-local]",
     suspendDirectory = <unset>,
     logDirectory = <unset>,
   tools = (vim.vm.ToolsConfigInfo) {
     dynamicType = <unset>,
      toolsVersion = <unset>,
     afterPowerOn = true,
      afterResume = true,
     beforeGuestStandby = true,
     beforeGuestShutdown = true,
     beforeGuestReboot = true,
      toolsUpgradePolicy = <unset>,
      pendingCustomization = <unset>,
      syncTimeWithHost = <unset>,
  },
```

```
flags = (vim.vm.FlagInfo) {
      dynamicType = <unset>,
      disableAcceleration = <unset>,
      enableLogging = <unset>,
      useToe = <unset>,
      runWithDebugInfo = <unset>,
      monitorType = <unset>,
      htSharing = <unset>,
      snapshotDisabled = <unset>,
      snapshotLocked = <unset>,
      diskUuidEnabled = <unset>,
      virtualMmuUsage = <unset>,
      snapshotPowerOffBehavior = "powerOff",
      recordReplayEnabled = <unset>,
  },
  consolePreferences = (vim.vm.ConsolePreferences) null,
  powerOpInfo = (vim.vm.DefaultPowerOpInfo) {
      dynamicType = <unset>,
      powerOffType = "preset",
      suspendType = "preset",
      resetType = "preset",
      defaultPowerOffType = <unset>,
      defaultSuspendType = <unset>,
      defaultResetType = <unset>,
      standbyAction = "powerOnSuspend",
  },
// the number of CPUs
  numCPUs = 1,
// the number of memory megabytes
  memoryMB = 256,
  memoryHotAddEnabled = <unset>,
   cpuHotAddEnabled = <unset>,
  cpuHotRemoveEnabled = <unset>,
  deviceChange = (vim.vm.device.VirtualDeviceSpec) [
      (vim.vm.device.VirtualDeviceSpec) {
         dynamicType = <unset>,
         operation = "add",
         fileOperation = <unset>,
// CDROM
         device = (vim.vm.device.VirtualCdrom) {
            dynamicType = <unset>,
// key number of CDROM
           key = -42,
            deviceInfo = (vim.Description) null,
            backing = (vim.vm.device.VirtualCdrom.RemotePassthroughBackingInfo) {
               dynamicType = <unset>,
               deviceName = "",
               useAutoDetect = <unset>,
               exclusive = false,
            }.
            connectable = (vim.vm.device.VirtualDevice.ConnectInfo) {
               dynamicType = <unset>,
               startConnected = false,
               allowGuestControl = true,
               connected = false,
// connects to this controller
            controllerKey = 200,
            unitNumber = 0,
        },
      },
      (vim.vm.device.VirtualDeviceSpec) {
         dynamicType = <unset>,
         operation = "add",
         fileOperation = <unset>,
// SCSI controller
         device = (vim.vm.device.VirtualLsiLogicController) {
            dynamicType = <unset>,
```

```
// key number of SCSI controller
            key = -44
            deviceInfo = (vim.Description) null,
            backing = (vim.vm.device.VirtualDevice.BackingInfo) null,
            connectable = (vim.vm.device.VirtualDevice.ConnectInfo) null,
            controllerKey = <unset>,
            unitNumber = <unset>,
            busNumber = 0,
            hotAddRemove = <unset>,
            sharedBus = "noSharing",
            scsiCtlrUnitNumber = <unset>,
         },
      },
      (vim.vm.device.VirtualDeviceSpec) {
         dynamicType = <unset>,
         operation = "add",
         fileOperation = <unset>.
// Network controller
         device = (vim.vm.device.VirtualPCNet32) {
            dynamicType = <unset>,
// key number of Network controller
            key = -48,
            deviceInfo = (vim.Description) null,
            backing = (vim.vm.device.VirtualEthernetCard.NetworkBackingInfo) {
               dynamicType = <unset>,
               deviceName = "Virtual Machine Network",
               useAutoDetect = <unset>,
               network = <unset>,
               inPassthroughMode = <unset>,
            },
            connectable = (vim.vm.device.VirtualDevice.ConnectInfo) {
               dynamicType = <unset>,
               startConnected = true,
               allowGuestControl = true,
               connected = true,
            },
            controllerKey = <unset>,
            unitNumber = <unset>,
            addressType = "generated",
            macAddress = <unset>,
            wakeOnLanEnabled = true,
        },
      },
      (vim.vm.device.VirtualDeviceSpec) {
         dynamicType = <unset>,
         operation = "add",
         fileOperation = "create",
// SCSI disk one
         device = (vim.vm.device.VirtualDisk) {
           dynamicType = <unset>,
// key number for SCSI disk one
            key = -1000000,
            deviceInfo = (vim.Description) null,
            backing = (vim.vm.device.VirtualDisk.FlatVer2BackingInfo) {
               dynamicType = <unset>,
               fileName = "",
               datastore = <unset>,
               diskMode = "persistent",
               split = false,
               writeThrough = false,
               thinProvisioned = <unset>,
               eagerlyScrub = <unset>,
               uuid = <unset>,
               contentId = <unset>,
               changeId = <unset>,
               parent = (vim.vm.device.VirtualDisk.FlatVer2BackingInfo) null,
            },
            connectable = (vim.vm.device.VirtualDevice.ConnectInfo) {
```

```
dynamicType = <unset>,
               startConnected = true,
               allowGuestControl = false,
               connected = true,
            },
// controller for SCSI disk one
           controllerKey = -44,
            unitNumber = 0,
// size in MB SCSI disk one
            capacityInKB = 524288,
            committedSpace = <unset>,
            shares = (vim.SharesInfo) null,
         },
      },
      (vim.vm.device.VirtualDeviceSpec) {
         dynamicType = <unset>,
         operation = "add",
         fileOperation = "create",
// SCSI disk two
         device = (vim.vm.device.VirtualDisk) {
            dynamicType = <unset>,
// key number of SCSI disk two
            key = -100,
            deviceInfo = (vim.Description) null,
            backing = (vim.vm.device.VirtualDisk.FlatVer2BackingInfo) {
               dynamicType = <unset>,
               fileName = "".
               datastore = <unset>,
               diskMode = "persistent",
               split = false,
               writeThrough = false,
               thinProvisioned = <unset>,
               eagerlyScrub = <unset>,
               uuid = <unset>,
               contentId = <unset>,
               changeId = <unset>,
               parent = (vim.vm.device.VirtualDisk.FlatVer2BackingInfo) null,
            connectable = (vim.vm.device.VirtualDevice.ConnectInfo) {
               dynamicType = <unset>,
               startConnected = true,
               allowGuestControl = false,
               connected = true,
            },
// controller for SCSI disk two
            controllerKey = -44,
            unitNumber = 1,
// size in MB SCSI disk two
           capacityInKB = 131072,
            committedSpace = <unset>,
            shares = (vim.SharesInfo) null,
         },
      }
  },
   cpuAllocation = (vim.ResourceAllocationInfo) {
      dynamicType = <unset>,
      reservation = 0,
      expandableReservation = <unset>,
      limit = <unset>,
      shares = (vim.SharesInfo) {
         dynamicType = <unset>,
         shares = 100,
         level = "normal",
      },
      overheadLimit = <unset>,
  },
  memoryAllocation = (vim.ResourceAllocationInfo) {
      dynamicType = <unset>,
```

```
reservation = 0,
     expandableReservation = <unset>,
      limit = <unset>,
      shares = (vim.SharesInfo) {
         dynamicType = <unset>,
         shares = 100.
         level = "normal",
     },
     overheadLimit = <unset>,
  },
  cpuAffinity = (vim.vm.AffinityInfo) null,
  memoryAffinity = (vim.vm.AffinityInfo) null,
  networkShaper = (vim.vm.NetworkShaperInfo) null,
   swapPlacement = <unset>,
   swapDirectory = <unset>,
  preserveSwapOnPowerOff = <unset>,
   bootOptions = (vim.vm.BootOptions) null,
  appliance = (vim.vService.ConfigSpec) null,
  ftInfo = (vim.vm.FaultToleranceConfigInfo) null,
  applianceConfigRemoved = <unset>,
  vAssertsEnabled = <unset>.
  changeTrackingEnabled = <unset>,
}
// end of VirtualMachineConfigSpec
```

The information above is quite complex, but much of the input consists of defaulted values that are assigned by the system. The remainder of the supplied information can be extracted from the output of the config.hardware.device table returned from PropertyCollector. Borrowing heavily from an SDK code example, the following code sets up the configuration specification:

```
// Duplicate virtual machine configuration
VirtualMachineConfigSpec configSpec = new VirtualMachineConfigSpec();
// Set the VM values
configSpec.setName("My New VM");
configSpec.setVersion("vmx-04");
configSpec.setGuestId("winNetStandardGuest");
configSpec.setNumCPUs(1);
configSpec.setMemoryMB(256);
// Set up file storage info
VirtualMachineFileInfo vmfi = new VirtualMachineFileInfo();
vmfi.setVmPathName("[plat004-local]");
configSpec.setFiles(vmfi);
vmfi.setSnapshotDirectory("[plat004-local]");
// Set up tools config info
ToolsConfigInfo tools = new ToolsConfigInfo();
configSpec.setTools(tools);
tools.setAfterPowerOn(new Boolean(true));
tools.setAfterResume(new Boolean(true));
tools.setBeforeGuestStandby(new Boolean(true));
tools.setBeforeGuestShutdown(new Boolean(true));
tools.setBeforeGuestReboot(new Boolean(true));
// Set flags
VirtualMachineFlagInfo flags = new VirtualMachineFlagInfo();
configSpec.setFlags(flags);
flags.setSnapshotPowerOffBehavior("powerOff");
// Set power op info
VirtualMachineDefaultPowerOpInfo powerInfo = new VirtualMachineDefaultPowerOpInfo();
configSpec.setPowerOpInfo(powerInfo);
powerInfo.setPowerOffType("preset");
powerInfo.setSuspendType("preset");
powerInfo.setResetType("preset");
powerInfo.setStandbyAction("powerOnSuspend");
// Now add in the devices
VirtualDeviceConfiqSpec[] deviceConfiqSpec = new VirtualDeviceConfiqSpec [5];
configSpec.setDeviceChange(deviceConfigSpec);
// Formulate the CDROM
deviceConfigSpec[0].setOperation(VirtualDeviceConfigSpecOperation.add);
VirtualCdrom cdrom = new VirtualCdrom();
```

```
VirtualCdromIsoBackingInfo cdDeviceBacking = new VirtualCdromRemotePassthroughBackingInfo();
cdDeviceBacking.setDatastore(datastoreRef);
cdrom.setBacking(cdDeviceBacking);
cdrom.setKey(-42);
cdrom.setControllerKey(new Integer(-200)); // Older Java required type for optional properties
cdrom.setUnitNumber(new Integer(0));
deviceConfigSpec[0].setDevice(cdrom);
// Formulate the SCSI controller
deviceConfigSpec[1].setOperation(VirtualDeviceConfigSpecOperation.add);
VirtualLsiLogicController scsiCtrl = new VirtualLsiLogicController();
scsiCtrl.setBusNumber(0);
deviceConfigSpec[1].setDevice(scsiCtrl);
scsiCtrl.setKey(-44);
scsiCtrl.setSharedBus(VirtualSCSISharing.noSharing);
// Formulate SCSI disk one
deviceConfiqSpec[2].setFileOperation(VirtualDeviceConfiqSpecFileOperation.create);
deviceConfiqSpec[2].setOperation(VirtualDeviceConfiqSpecOperation.add);
VirtualDisk disk = new VirtualDisk();
VirtualDiskFlatVer2BackingInfo diskfileBacking = new VirtualDiskFlatVer2BackingInfo();
diskfileBacking.setDatastore(datastoreRef);
diskfileBacking.setFileName(volumeName);
diskfileBacking.setDiskMode("persistent");
diskfileBacking.setSplit(new Boolean(false));
diskfileBacking.setWriteThrough(new Boolean(false));
disk.setKey(-1000000);
disk.setControllerKey(new Integer(-44));
disk.setUnitNumber(new Integer(0));
disk.setBacking(diskfileBacking);
disk.setCapacityInKB(524288);
deviceConfigSpec[2].setDevice(disk);
// Formulate SCSI disk two
deviceConfigSpec[3].setFileOperation(VirtualDeviceConfigSpecFileOperation.create);
deviceConfigSpec[3].setOperation(VirtualDeviceConfigSpecOperation.add);
VirtualDisk disk2 = new VirtualDisk();
VirtualDiskFlatVer2BackingInfo diskfileBacking2 = new VirtualDiskFlatVer2BackingInfo();
diskfileBacking2.setDatastore(datastoreRef);
diskfileBacking2.setFileName(volumeName);
diskfileBacking2.setDiskMode("persistent");
diskfileBacking2.setSplit(new Boolean(false));
diskfileBacking2.setWriteThrough(new Boolean(false));
disk2.setKey(-100);
disk2.setControllerKey(new Integer(-44));
disk2.setUnitNumber(new Integer(1));
disk2.setBacking(diskfileBacking2);
disk2.setCapacityInKB(131072);
deviceConfigSpec[3].setDevice(disk2);
// Finally, formulate the NIC
deviceConfigSpec[4].setOperation(VirtualDeviceConfigSpecOperation.add);
com.VMware.vim.VirtualEthernetCard nic = new VirtualPCNet32();
VirtualEthernetCardNetworkBackingInfo nicBacking = new VirtualEthernetCardNetworkBackingInfo();
nicBacking.setNetwork(networkRef);
nicBacking.setDeviceName(networkName);
nic.setAddressType("generated");
nic.setBacking(nicBacking);
nic.setKey(-48);
deviceConfigSpec[4].setDevice(nic);
// Now that it is all put together, create the virtual machine.
// Note that folderMo, resourcePool, and hostMo, are moRefs to
// the Folder, ResourcePool, and Host where the VM is to be created.
ManagedObjectReference taskMoRef =
    serviceConnection.getService().createVM_Task(folderMo, configSpec, resourcePool, hostMo);
```

Using the VirtualMachineConfigInfo

A backup application can also use information contained in a VirtualMachineConfigInfo. If at backup time you preserve all the VirtualMachineConfigInfo details that describe the virtual machine, you can transfer much of this information into a VirtualMachineConfigSpec to create a virtual machine at restore time. However, some of the information in VirtualMachineConfigInfo is not needed, and if used in the Spec, virtual machine creation can fail. For example, a VirtualMachineConfigSpec that contains information about so called "Default Devices" usually fails. The list of default devices includes:

```
vim.vm.device.VirtualIDEController
vim.vm.device.VirtualPS2Controller
vim.vm.device.VirtualPCIController
vim.vm.device.VirtualSIOController
vim.vm.device.VirtualKeyboard
vim.vm.device.VirtualVMCIDevice
vim.vm.device.VirtualPointingDevice
```

However, other controllers and devices must be explicitly included in the VirtualMachineConfigSpec.

Some information about devices is unneeded and can cause problems if supplied. Each controller device has its vim.vm.device.VirtualController.device field, which is an array of devices that report to the controller. The server rebuilds this list when a virtual machine is created, using the (negative) device key numbers supplied as a guide. The relationship between controller and device must be preserved using negative key numbers in the same relationship as in the hardware array of VirtualMachineConfigInfo.

The parent property for virtual disk backing information must be set to null. In the sample code for creating a virtual machine, find vim.vm.device.VirtualDisk.FlatVer2BackingInfo on page 76 and page 77. The null setting is required because the pre-backup snapshot causes the parent property to be populated with a reference to the base disk.

One other configuration needs substitution. VirtualMachineConfigInfo contains the cpuFeatureMask, field, which is an array of HostCpuIdInfo. The array entries must be converted to ArrayUpdateSpec entries containing the VirtualMachineCpuIdInfoSpec along with the "operation" field, which must contain the value ArrayUpdateOperation::add. The VirtualMachineCpuIdInfoSpec also contains a HostCpuIdInfo array that you can copy from the cpuFeatureMask array in VirtualMachineConfigInfo. These items are not reflected in the sample code. Everything else can be copied intact from VirtualMachineConfigInfo data.

To summarize: when creating a virtual machine in which to restore virtual disk:

- Exclude default devices, and VirtualController.device, from the VirtualMachineConfigSpec.
- Set the parent virtual disk backing information (VirtualDisk.FlatVer2BackingInfo) to null.
- Convert HostCpuIdInfo array entries to ArrayUpdateSpec, insert ArrayUpdateOperation::add, and copy the HostCpuIdInfo array from cpuFeatureMask into VirtualMachineConfigInfo.

Editing or Deleting a Device

If backup clients want to edit or delete a device, they must use the server-provided key when referring to an existing device. For the definition of key, see "Creating a Virtual Machine" on page 74. For example, see the key and controllerKey for CDROM in the source code on page 75. The key uniquely identifies a device, while the controllerKey uniquely identifies the controller where it is connected.

Restoring Virtual Disk Data

As in the section "Low Level Restore Procedures" on page 73, VixDiskLib functions provide interfaces for writing the data to virtual disk, either locally or remotely.

Raw Device Mapping (RDM) Disks

To create an RDM disk using CreateVM_Task, use a LUN that is not occupied and thus is still available. Developers sometimes use the same LUN uuid that is available in the configInfo object, which can cause errors because the LUN uuid is datastore specific.

Call QueryConfigTarget to fetch the ConfigTarget.ScsiDisk.Disk.CanonicalName property, set in VirtualDiskRawDiskMappingVer1BackInfo.deviceName. Also call QueryConfigTarget to fetch ConfigTarget.ScsiDisk.Disk.uuid, set in VirtualDiskRawDiskMappingVer1BackInfo.lunUuid. When creating the virtual machine, avoid host-specific properties of configInfo, which should be set according to host configuration where the virtual machine is restored.

Restore of Incremental Backup Data

At some point you might need to restore a virtual disk from the backup data that you gathered as described in "Changed Block Tracking on Virtual Disks" on page 70. The essential procedure is as follows:

- 1 Power off the virtual machine, if powered on.
- 2 Using VirtualMachineConfigInfo that corresponds to the last known good state of the guest operating system, re-create the virtual machine as described in "Using the VirtualMachineConfigInfo" on page 80.
- 3 Completely reload the base virtual disk using the full backup that started the most recent series of incremental backups.
- 4 Create a snapshot. This is mandatory for SAN mode restore.
 - A restore snapshot is forbidden when restoring to the parent (backing) disk on a VVol datastore.
- 5 For SAN mode restore, disable changed block tracking. SAN writes are not possible with it enabled
- 6 Sequentially restore the incremental backup data. You can do this either forwards or backwards. If you work forwards, the restore might write some sectors more than once. If you work backwards, you must keep track of which sectors were restored so as to avoid restoring them again from older data.
 - a From your backup records, get the change ID of the incremental backup to be restored. Your software must also store the changed-block information, so it knows which sectors of virtual disk to restore. Once you start restoring virtual disk, the change tracking mechanism will misreport.
 - b Restore only changed areas to the virtual disks referred to by the snapshot. This ensures that you do not write the data to the redo log created by the snapshot. When restoring a thin provisioned (sparse) disk, use the star "*" change ID to avoid writing zeroes to the unallocated blocks.
 - c Repeat Step a and Step b as necessary by applying incremental backup data sets in order.
- 7 If applicable (SAN mode) revert to the base virtual disk, thus eliminating the snapshot.

Restore Fails with Direct Connection to ESXi Host

Sometimes you must restore a virtual machine directly to an ESXi host, for example in disaster recovery when vCenter Server runs on ESXi as a virtual machine. A new vSphere 5 feature tries to prevent this if the ESXi host is managed by vCenter. To circumvent this and restore the virtual machine, you must first disassociate the host from vCenter. In earlier releases, vCenter management had less state but was revocable only from vCenter.

- 1 Using the vSphere Client, connect directly to the ESXi 5.0 or later host.
- 2 In the Inventory left-hand panel, select the host. In the right-hand panel, click Summary.
- 3 In the box titled Host Management, click Disassociate host from vCenter Server. You do not need to put the host in Maintenance Mode.
- 4 After the vCenter Server has been restored and is back in service, use it to reacquire the host.

Currently there is no API to disassociate a host from vCenter Server.

Tips and Best Practices

This section discusses the various advanced transport mechanisms, and other backup issues.

VDDK 5.0 contained two new VixDiskLib calls (PrepareForAccess and EndAccess) to disable and enable Storage vMotion during backup. This prevents stale disk images from being left behind if a virtual machine has its storage moved while a backup is taking place. VMware strongly recommends use of these calls.

When an ESXi host is managed by vCenter Server, vSphere API calls cannot contact the host directly: they must go through vCenter. If necessary, especially during disaster recovery, the administrator must disassociate the ESXi host from vCenter Server before the host can be contacted directly.

Advanced transports allow programs to transfer data in the most efficient manner. SAN transport is available only when the physical-machine host has SAN access. HotAdd works for the appliance model, where backup is done from inside virtual machines. HotAdd requires the virtual machine datastore to be accessible from the backup appliance. NBDSSL is a secure fallback when over-the-network backup is your only choice.

SAN transport is supported only on physical machines, and HotAdd transport is supported only on virtual machines. SAN requires a physical proxy to share a LUN with the ESXi host where a datastore resides, enabling direct access to raw data, and bypassing the host altogether for I/O operations. HotAdd involves attaching a virtual disk to the backup proxy just like attaching the disk to a virtual machine.

Best Practices for SAN Transport

For array-based storage, SAN transport is often the best performing choice for backups when running on a physical proxy. It is disabled inside virtual machines, so use SCSI HotAdd instead on a virtual proxy.

SAN transport is not always the best choice for restores. It offers the best performance on thick disks, but the worst performance on thin disks, because of round trips through two disk manager APIs, AllocateBlock and ClearLazyZero. For thin disk restore, NBDSSL is usually faster. Changed Block Tracking (CBT) must be disabled for SAN restores. SAN transport does not support writing to redo logs (child disks including linked clones and snapshots), only to base disks. SAN transport is not supported on VVol datastores.

Before vSphere 5.5, when writing to SAN during restore, disk size had to be a multiple of the underlying VMFS block size, otherwise the write to the last fraction of a disk would fail. This was fixed in the ESXi 5.5 release.

Programs that open a local virtual disk in SAN mode might be able to read (if the disk is empty) but writing will throw an error. Even if programs call VixDiskLib_ConnextEx() with NULL parameter to accept the default transport mode, SAN is selected as the preferred mode if SAN storage is connected to the ESXi host. VixDiskLib should, but does not, check SAN accessibility on open. With local disk, programs must explicitly request NBDSSL mode.

For a Windows Server 2008 and later proxy, set SAN policy to onlineAll. Set SAN disk to read-only except for restore. You can use the **diskpart** utility to clear the read-only flag. SAN policy varies by Windows Server 2008 edition. For Enterprise and Datacenter editions, the default Windows SAN policy is offline, which is unnecessary when vSphere mediates SAN storage.

For SAN transport, one major factor impacting performance is that the read buffer should be aligned with the sector size, currently 512. You can specify three parameters for VixDiskLib_Read: start sector, number of sectors to read, and the buffer to hold data. The proper read buffer size can be allocated using, for example, _aligned_malloc on Windows or posix_memalign on Linux. SAN mode performs best with about six concurrent streams per ESXi host; more than six streams usually results in slower total throughput.

Best Practices for HotAdd Transport

Deploy the proxy on VMFS-6 or VMFS-5 volumes, or on VMFS-3 volumes capable of large block size (see "About the HotAdd Proxy" on page 25) so that the proxy can back up very large virtual disks. HotAdd is a SCSI feature and does not work for IDE disks. The paravirtual SCSI controller (PVSCSI) is the recommended default for HotAdd, but other controller types work too.

A redo log is created for HotAdded disks, on the same datastore as the base disks. Do not remove the target virtual machine (the one being backed up) while HotAdded disk is still attached. If removed, HotAdd fails to properly clean up redo logs so virtual disks must be removed manually from the backup appliance. Also, do not remove the snapshot until after cleanup. Removing it could result in an unconsolidated redo log.

Removing all disks on a controller with the vSphere Client also removes the controller. You might want to include some checks in your code to detect this in your appliance, and reconfigure to add controllers back in.

HotAdded disks should be released with VixDiskLib_Cleanup() before snapshot delete. Cleanup might cause improper removal of the change tracking (ctk) file. You can fix it by power cycling the virtual machine.

Virtual disk created on Windows by HotAdd backup or restore may have a different disk signature than the original virtual disk. The workaround is to reread or rewrite the first disk sector in NBDSSL mode. Customers running a Windows Server 2008 or later proxy should set SAN policy to onlineAll (see SAN section above).

The HotAdd implementation assumes that proxy and target VMs are on the same datastore and accessible from the same connection, that is, the same vCenter Server. This is so VADP can obtain a list of all disks on the target VM from a connection. In vCloud environments where two vCenter Servers share a single datastore, causing VADP to make two connections, there is no mechanism for informing one connection of the disks available on the other connection. VMware needs to redesign HotAdd to support multiple connections.

Best Practices for NBDSSL Transport

Various versions of ESXi have different defaults for timeouts. Before ESXi 5.0 there were no default network file copy (NFC) timeouts. Default NFC timeout values may change in future releases.

VMware recommends that you specify default NFC timeouts in the VixDiskLib configuration file. If you do not specify a timeout, older versions of ESXi hold the corresponding disk open indefinitely, until vpxa or hostd is restarted. However if you do specify a timeout, you might need to perform some "keepalive" operation to prevent the disk from being closed on the server side. Reading block 0 periodically is a good keepalive operation. As a starting point, recommended settings are three minutes for Accept and Request, one minute for Read, ten minutes for Write, and no timeouts (0) for nfcFssrvr and nfcFssrvrWrite.

General Backup and Restore

With SSL certificate checking in vSphere 5.1 and after, DNS services must be configured in the backup proxy, otherwise SSL_Verify will fail with the "no host found" error.

For incremental backup of virtual disk, always enable changed block tracking (CBT) before the first snapshot. When doing full restores of virtual disk, disable CBT for the duration of the restore. File-based restores affect change tracking, but disabling CBT is optional for partial restore (file level restore), except with SAN transport. CBT must be disabled for SAN writes because of thin-provisioning and clear-lazy-zero operations.

Backup software should ignore independent disks (those not capable of snapshots). These virtual disks are unsuitable for backup. They throw an error if a snapshot is attempted on them.

To back up thick disk, the proxy's datastore must have at least as much free space as the maximum configured disk size for the backed-up virtual machine. Thick disk takes up all its allocated size in the datastore. To save space, you can choose thin-provisioned disk, which consumes only the space actually containing data.

If you do a full backup of lazy-zeroed thick disk with CBT disabled, the software reads all sectors, converting data in empty (lazy-zero) sectors to actual zeros. Upon restore, this full backup data will produce eager-zeroed thick disk. This is one reason why VMware recommends enabling CBT before the first snapshot.

With CBT enabled for backups on an NFS datastore, thin-provisioned virtual disk may be turned thick upon restore, unless the NFS server supports <code>lseek(...SEEK_DATA)</code>, <code>ioctl(...FS_IOC_FIEMAP)</code>, or equivalent function.

Do not make verbatim copies of configuration files, which can change. For example, entries in the .vmx file point to the snapshot, not the base disk. The .vmx file contains virtual-machine specific information about current disks, and attempting to restore this information could fail. Instead use PropertyCollector and keep a record of the ConfigInfo structure.

Backup and Restore of Thin-Provisioned Disk

Thin-provisioned virtual disk is created on first write. So the first-time write to thin-provisioned disk involves extra overhead compared to thick disk, whether using NBDSSL or HotAdd. This is due to block allocation overhead, not VDDK advanced transports. However once thin disk has been created, performance is similar to thick disk, as discussed in the *Performance Study of VMware vStorage Thin Provisioning*.

When applications perform random I/O or write to previously unallocated areas of thin-provisioned disk, subsequent backups can be larger than expected, even with CBT enabled. In some cases, disk defragmentation might help reduce the size of backups.

About Changed Block Tracking

QueryChangedDiskAreas("*") returns information about areas of a virtual disk that are in use (allocated). The current implementation depends on VMFS properties, similar to properties that SAN transport mode uses to locate data on a SCSI LUN. Both rely on unallocated areas (file holes) in virtual disk, and the LazyZero designation for VMFS blocks. Thus, changed block tracking yields meaningful results only on VMFS. On other storage types, it either fails, or returns a single extent covering the entire disk.

You should enable changed block tracking in the order recommended by "Enabling Changed Block Tracking" on page 71. The first time you call QueryChangedDiskAreas ("*"), it should return allocated areas of virtual disk. Subsequent calls return changed areas, instead of allocated areas. If you call QueryChangedDiskAreas 'pfafter a snapshot but before you enable changed block tracking, it also returns unallocated areas of virtual disk. With thin-provisioned virtual disk this could be a large amount of zero data.

The guest operating system has no visibility of changed block tracking. Once a virtual machine has written to a block on virtual disk, the block is considered in use. The information required for the "*" query is computed when changed block tracking is enabled, and the .ctk file is pre-filled with allocated blocks. The mechanism cannot report changes made to virtual disk before changed block tracking was enabled.

HotAdd and SCSI Controller IDs

When using HotAdd backup, always add SCSI controllers to virtual machines in numeric order.

Most systems lack an interface to report which SCSI controller is assigned to which bus ID. HotAdd assumes that the unique ID for a SCSI controller corresponds to its bus ID. This assumption could be false. For instance, if the first SCSI controller on a VM is assigned to bus ID 0, but you add a SCSI controller and assign it to bus ID 3, HotAdd transport may fail because it expects unique ID 1. To avoid problems, when adding SCSI controllers to a VM, the bus assignment for the controller must be the next available bus number in sequence.

Also note that VMware implicitly adds a SCSI controller to a VM if a bus:disk assignment for a newly created virtual disk refers to a controller that does not yet exist. For instance, if disks 0:0 and 0:1 are already in place, adding disk 1:0 is fine, but adding disk 3:0 breaks the bus ID sequence, implicitly creating out-of-sequence SCSI controller 3. To avoid HotAdd problems, you should add virtual disks in numeric sequence.

To handle more disks than fit on one controller, you must add some permanent dummy disks to the proxy VM, one on each additional controller that may be needed. Adding only the controller does not cause the controller to remain attached to the proxy VM. A VMDK must be added to keep the controller attached to its proxy VM.

Encrypted VM Backup and Restore

Starting with vSphere 6.5, customers can take advantage of virtual machine encryption. Encryption protects sensitive parts of a VM and some or all of its virtual disks. The vCenter Server retrieves keys from a key management server and pushes them to ESXi hosts, which use the keys to encrypt virtual disk data, NVRAM, portions of the VMX file, and so forth. See the *vSphere Security* manual for details.

To back up encrypted virtual disks, VDDK obtains the encryption keys and decrypts virtual disk data before copying to backup media, so virtual disk data on backup media are in the clear (not encrypted). VMware recommends that backup agents use their own encryption to protect data on backup media.

Backup of encrypted disks is supported with NBD, NBDSSL and HotAdd transport modes, but not supported with SAN transport. To back up encrypted virtual disks using HotAdd, the backup proxy VM must have been encrypted as well. NBD, NBDSSL, and HotAdd backup modes require **Cryptographer.Access** permission, and HotAdd backup mode requires **Cryptographer.AddDisk** also.

Encrypted virtual disks When customers restore virtual disks that were encrypted at backup time, they likely want them to remain encrypted after restore. If a disk was encrypted when it was backed up, the backup agent can remember the storage policy and apply it to the restored disk. If that storage policy no longer exists in the system, the backup agent could ask the administrator what policy to apply for encrypted disks, or use the sample VM encryption storage policy. Optionally the backup agent can remember a disk's key ID and provide it at restore time, which will result in the restored disk using the same key as the original disk. If not specified, vCenter Server gets a new key from the key server.

Recommendations for VM home When a VM is encrypted, its VMX configuration file contains portions that are encrypted, and the entire NVRAM file is encrypted. To completely recover a VM from backups, the NVRAM file must be restored, and two additional properties from the VirtualMachine object's configuration must be restored: (1) the encryption key identifier, found in the VirtualMachine ConfigInfo.keyId property, and (2) an encrypted blob containing VM secrets, called the encryption.bundle, found in the VirtualMachine ConfigInfo.extraConfig list.

VMware recommends against directly copying the entire VMX configuration file, but instead using the PropertyCollector to keep a record of configuration structures as documented earlier in this chapter. When a VM is saved, the backup agent should include the ConfigInfo.keyId, the encryption.bundle from ConfigInfo.extraConfig, and the current storage policy.

Later when the backup is restored, these values should be provided in the new VirtualMachine ConfigSpec. This will ensure that the recovered VM metadata files are protected with the same key as the original VM. If a VM is restored to a vCenter Server with key server access, the VM will boot. However if the vCenter Server lacks key server access, the VM will not power on after restore, because vCenter Server cannot push encryption keys to its ESXi host.

The NVRAM file can be handled as it was in previous releases, using HTTP download and upload, but without saving the additional ConfigInfo properties described above, the NVRAM will be unusable. This is because it is encrypted with the key that is found in the encryption.bundle, which is in turn sealed with the key identified by ConfigInfo.keyId. The recovered VM must be created in a vCenter Server that has access to the same key server as the original, or a replicated key server instance with the same cluster name.

What to do if something fails If the ConfigInfo.keyId and encryption.bundle were not saved, it will be impossible to encrypt the recovered VM using its original encryption keys. However the VM may be encrypted with new keys from the key server. If the NVRAM file was saved, it is unusable without the original ConfigInfo.keyId and encryption.bundle information. If the NVRAM file was not saved, or unusable, a generic one can be used instead. After NVRAM is lost and regenerated, UEFI enabled VMs may require Secure Boot to be reconfigured, or the boot disk to be reconfigured.

If the vCenter Server has lost access to the original keys from the key server, then a restore operation may remove the encryption.bundle from ConfigInfo.extraConfig and specify only a VM encryption storage policy for the VM and its disks. Again this will cause any data present in the NVRAM to be unusable.

Example 7-1. Encrypted VM backup and restore

```
* For encrypted VM backup:
* 1) Back up keyId and encryption.bundle of source VM from configInfo.
 * 2) Back up nvram using HTTP service.
     See "HTTP Access to vSphere Server Files" in vSphere Web Services SDK Programming Guide.
// get source VM config
VirtualMachineConfigInfo sourceVmConfigInfo = ...;
// save keyId
CryptoKeyId keyId = sourceVmConfigInfo.getKeyId();
// save encryption.bundle, which is in extraConfig
List<OptionValue> extraCfg = sourceVmConfigInfo .getExtraConfig();
* For encrypted VM restore:
 * 1) Configure VM with same keyId and encryption.bundle as source VM (requires same KMS).
 * 2) Create a storage profile for encryption and assign it to the VM.
     See "Create an Encryption Storage Policy" in vSphere Web Services SDK Programming Guide.
 * 3) Restore nvram file using HTTP service.
     Again see "HTTP Access to vSphere Server Files" in Web Services SDK Programming Guide.
//create configSpec for VM to be created
VirtualMachineConfigSpec configSpec = ...;
//set keyId
CryptoSpecEncrypt cryptoSpec = new CryptoSpecEncrypt();
cryptoSpec.setCryptoKeyId(keyId);
configSpec.setCrypto(cryptoSpec);
//set encryption.bundle
configSpec.setExtraConfig(extraCfg);
```

```
//set PbmProfile for encryption
public class CreateVMEncryptionProfile extends ConnectedServiceBase {
   private PbmServiceInstanceContent spbmsc;
   private String profileName;
   ...
   for (PbmCapabilityVendorResourceTypeInfo vendor : vendorInfo)
        for (PbmCapabilityVendorNamespaceInfo vnsi : vendor .getVendorNamespaceInfo())
        if (vnsi.getNamespaceInfo().getNamespace().equals("vmwarevmcrypt")) {
            encryptionCapable = true;
            break;
        }
        // For complete code, see Example: Java program to set storage policy for encryption.
```

Windows Backup Implementations

The following sections discuss issues when backing up Windows virtual machines.

Disabling Automount in Windows Proxy

When using HotAdd transport from a Windows proxy, it can make unwanted changes to HotAdded volumes. To prevent this, backup vendors and customers should disable Windows automount on the backup proxy.

To disable Windows automount

```
    Start DiskPart.
```

```
C:\> diskpart
```

2 Disable automounting.

```
DISKPART> automount disable
```

3 Prevent any previously mounted volumes from being re-mounted the next time.

```
DISKPART> automount scrub
```

4 Exit DiskPart.

DISKPART> exit

Security and Remote Desktop

For security reasons, remote desktop protocol (RDP) should be disabled on a Windows proxy.

Working with Microsoft Shadow Copy

Microsoft Shadow Copy, also called Volume Snapshot Service (VSS), is a Windows Server data backup feature for creating consistent point-in-time copies of data (called shadow copies).

The type of quiescing used varies depending on the operating system of the backed-up virtual machine, as shown in Table 7-5. ESXi 4.1 added support for Windows 2008 guests using application level quiescing.

Table 7-5. Driver Type and Quiescing Mechanisms According to Guest Operating Systems

Guest Operating System	Driver Type Used	Quiescing Type Used
Windows XP 32-bit Windows 2000 32-bit	Sync Driver	File-system consistent quiescing
Windows Vista 32- or 64-bit Windows 7 32- or 64-bit	VMware VSS component	File-system consistent quiescing
Windows 2003 32- or 64-bit	VMware VSS component	Application-consistent quiescing

Table 7-5. Driver Type and Quiescing Mechanisms According to Guest Operating Systems (Continued)

Guest Operating System	Driver Type Used	Quiescing Type Used
Windows 2008 32- or 64-bit Windows Server 2008 R2	VMware VSS component	Application-consistent quiescing. For this to be available, several conditions must be met:
		■ Virtual machine must be running on ESXi 4.1 or later.
		■ The UUID attribute must be enabled. It is enabled by default for virtual machines created on 4.1 or later. For details see "Enable Windows 2008 Virtual Machine Application Consistent Quiescing" on page 87.
		The virtual machine must use SCSI disks only and have as many free SCSI slots as the number of disks. Application-consistent quiescing is not supported for virtual machines with IDE disks.
		■ The virtual machine must not use dynamic disks.
Windows Server 2012	VMware VSS component	Same as above.
Other guest operating system	Not applicable	Crash-consistent quiescing

File-system consistent quiescing prevents file systems from becoming corrupted, for example, journaled file systems are allowed to write out pending transactions. Crash-consistent quiescing is the ability to restore an application as if it suddenly crashed and lost all stateful information. This involves minimal effort because only data already written to disk is guaranteed safe. Application-consistent quiescing is the ability to restore stateful information as well.

Restore must be done using the backup application's guest agent. The vSphere APIs for Data Protection provide no host agent support for this. Applications authenticating with SSPI might not work right because HTTP access will demand a user name and password, unless the session was recently authenticated.

When performing VSS quiescing while creating the snapshot of a Windows virtual machine, VMware Tools generate a vss-manifest.zip file containing the backup components document (BCD) and writer manifests. The host agent stores this manifest file in the snapshotDir of the virtual machine. Backup applications should get the vss-manifest.zip file so they can save it to backup media. There are several ways to get this file:

- Using the datastore access HTTPS protocol, for example by browsing to https://<server-or-host>/folder/ and continuing downward to the snapshot directory until you find the vss-manifest.zip file.
- By calling the CopyDatastoreFile_Task method in the vSphere API. This method accepts the URL formulated above for HTTPS, or a datastore path. (CopyVirtualDisk_Task is for VMDK files).
- With the vifs command in the vMA or vCLI.
- With the Copy—DatastoreItem cmdlet in the PowerCLI (requires PowerShell and VMware snap-in).

Windows 2008 application level quiescing is performed using a hardware snapshot provider. After quiescing the virtual machine, the hardware snapshot provider creates two redo logs per disk: one for the live virtual machine writes and another for the VSS and writers in the guest to modify the disks after the snapshot operation as part of the quiescing operations.

The snapshot configuration information reports this second redo log as part of the snapshot. This redo log represented the quiesced state of all the applications in the guest. This redo log must be opened for backup with VDDK 1.2 or later. The older VDDK 1.1 software cannot open the second redo log for backup.

Application consistent quiescing of Windows 2008 virtual machines is only available when those virtual machines are created in vSphere 4.1 or later. Virtual machines created in vSphere 4.0 can be updated to enable application consistent quiescing by modifying a virtual machine's enableUUID attribute.

For information about VSS, see the Microsoft TechNet article, *How Volume Shadow Copy Service Works*. For information about Security Support Provider Interface (SSPI), see the MSDN Web site.

Enable Windows 2008 Virtual Machine Application Consistent Quiescing

- 1 Start the vSphere Client, and log in to a vCenter Server.
- 2 Select Virtual Machines and Templates and click the Virtual Machines tab.

- 3 Right-click the Windows 2008 virtual machine for which you are enabling the disk UUID attribute, and select **Power > Power Off**. Wait for the virtual machine to power off.
- 4 Right-click the virtual machine, and click Edit Settings.
- 5 Click the Options tab, and select the General entry in the settings column.
- 6 Click Configuration Parameters... The Configuration Parameters window appears.
- 7 Click Add Row.
- 8 In the Name column, enter disk.EnableUUID. In the Value column, enter TRUE.
- 9 Click **OK** and click **Save**.
- 10 Power on the virtual machine.

Application consistent quiescing is available for this virtual machine after the UUID property is enabled.

Application-Consistent Backup and Restore

Here is the approximate procedure for software to performs application-consistent backup and restore:

- 1 Call CreateSnapshot_task with the quiescent flag set true.
- 2 Open the leaf node of the disk with VDDK and read both the base VMDK and the snapshot at once.
- 3 Delete the snapshots created in the first step.
- 4 During restore, create a new virtual machine.
- 5 Write the VMDK to disk with VDDK. It should have both base and quiesced information.

During backup, if the snapshot was created with quiesce flag set to true, and all the quiescing conditions are met, so the snapshot is created involving VSS and the snapshot disks represent application consistent state of the guest OS. You should be able to confirm this by downloading the VSS manifest zip file, unzipping it to check if it has just the backup component document (in which case file system quiescing was performed) or also writer manifests (in which case application quiescing was performed).

Quiescing involves the VSS mechanism designed by Microsoft. So, regarding VSS backup-restore verification, refer to the VSS documentation provided by Microsoft. VMware helps by providing a vss-manifest.zip file that contains Backup/Writers Components details. This is generated by the VSS mechanism after backup. By cross verifying these backup/writers components details according to Microsoft VSS documentation, you can verify if a particular application-consistent quiescing was completed successfully or not.

VMware Tools is responsible for initiating the VSS snapshot process as the VSS requester. Users send a request to hostd for a quiesced snapshot of the virtual machine. The request goes from hostd to the VMware Tools for a VSS snapshot. Once the VSS snapshot is completed (with success or error) it communicates back to the hostd process. The VSS snapshot is created with the vss-manifest file, or without this file in the error case.

The VSS requester sets up the overall configuration for the backup operation, including whether the snapshot should be performed in component mode or not, whether to take a snapshot with a bootable system state, and whether the snapshot should be for a full copy or differential backup. If application-consistent quiescing is performed, then all writers and all components are involved.

VMware Tools initiates VSS quiescing using VSS_CTX_BACKUP context for application quiescing capable guests with backup state set to select components, backup bootable system state with backup type VSS_BT_COPY and no partial file support and VSS_CTX_FILE_SHARE_BACKUP for file system quiescing capable guests. Currently there is no way to control any of these parameters.

New VSS Support Added in vSphere 6.5

The vSphere 6.5 release includes additional volume shadow-copy service (VSS) configurations and features. To support these new configuration and for more granular control over Windows guest OS quiescing, the function CreateSnapshotEx_Task was added to the 6.5 vSphere API, superseding CreateSnapshot_Task. CreateSnapshotEx_Task takes a quiesceSpec parameter, of type VirtualMachineGuestQuiesceSpec or VirtualMachineWindowsQuiesceSpec. The latter type can specify several important fields such as:

- vssBackupType VSS_BT_COPY was previously used as the default for CreateSnapshot_Task but now VSS_BT_FULL, VSS_BT_INCREMENTAL, VSS_BT_DIFFERENTIAL, and VSS_BT_LOG are available also. Log truncation may be triggered according to application settings.
- vssBackupContext this was introduced to enforce application (context VSS_CTX_BACKUP) quiescing or file system (context VSS_CTX_FILE_SHARE_BACKUP) quiescing.
- The timeout (default 15 minutes) for quiescing virtual machines can now be configured anywhere from five minutes to four hours.

CreateSnapshotEx_Task requires VMware Tools 10.1.0 or higher installed on the backed-up virtual machine.

The VMware VSS Implementation

As of Windows Server 2008, disk UUIDs must be enabled for VSS quiesced snapshots. Disk UUIDs might not be enabled if a virtual machine was upgraded from virtual hardware version 4.

VMware VSS does not support virtual machines with IDE disks, nor does it support virtual machines with an insufficient number of free SCSI slots.

Before vSphere 5.1, reverting to a writable snapshot sometimes left orphaned virtual disks that the system never removed. In the vSphere 5.1 release, writable snapshots are correctly accounted for as sibling snapshots. This permits cleaner management, because the disk chain matches the snapshot hierarchy, and it avoids orphaned disks. Linux backup software takes a read-only snapshot so is not affected. On Windows, VSS backup software may create two snapshots, one made writable by calling CreateSnapshot_task with the quiesce flag set true.

To add support for granular application control, specify:

- whether pre-freeze and post-thaw scripts get invoked
- whether quiescing gets invoked
- VSS snapshot context (application, file system quiescing, and so forth)
- VSS backup context (full, differential, incremental)
- writers/components to be involved during quiescing
- whether to fail quiescing or continue if one of the writers fails to quiesce
- retry count

A VSS quiesced snapshot reports as VSS_BT_COPY to VSS, hence no log truncation. The VSS manifest can be downloaded with HTTP. By default, all VSS writers are involved, but a mechanism for excluding writers exists; see the VMware KB article 1031200. For help troubleshooting, see KB article 1007696.

Linux Backup Implementation

On Linux virtual machines, VMware supports only crash-consistent backups, however application-consistent backups are possible through the use of customer pre-freeze and post-thaw scripts.

Two methods are attempted sequentially when a quiesced snapshot is requested of a Linux virtual machine:

1 Using the ioctl(2) mechanisms FIFREEZE and FITHAW built into the Linux kernel.

This option is available in only with kernel versions newer than 2.6.32 on Linux virtual machines running VMware Tools 5.x or higher (vSphere 5.0 and later).

2 Using the sync(2) system call.

The first method allows you define custom scripts that get called by VMware Tools before and after quiesce. Here are two commands that can be called from pre-freeze and post-freeze scripts:

```
# -f means freeze
   fsfreeze -f /
# -u means unfreeze
  fsfreeze -u /
```

The mount point used in the scripts (root in the examples above) must be mounted with the noatime option, meaning do not update inode access times on the file system, for freeze and unfreeze to work.

Backing Up vApps in vCloud Director

8

This chapter introduces developers to the concepts and procedures for creating backup and restore solutions for vCloud Director. This chapter is divided into the following main sections:

- "Introduction to Tenant vApps" on page 91
- "Conceptual Overview" on page 92
- "Use Cases Overview" on page 94
- "vCloud API Operations" on page 96
- "Conclusion" on page 106

Introduction to Tenant vApps

The vApp is a management construct that encapsulates one or more virtual machines running in the vSphere environment. The tenant vApp is a higher-level construct that allows vCloud Director to manage vApps and virtual machines running in a multi-tenant datacenter, or in a multi-tenant cloud, based on vSphere.

Figure 1 shows the objects within a single organization that you can access with the vCloud API.

Organization vDC1 Catalog 3 vApp Catalogitem Catalog 2 Catalogitem Catalog 1 vDC2 Catalogitem Catalogitem Catalogitem Catalogitem Media Network users groups TasksList Network

Figure 1. vCloud API Object Taxonomy

Multi-tenant and self-service capabilities of vCloud Director provide multiple levels of protection for a vApp. A service provider can offer vApp protection at the system level, the tenant level, or the end-user level, managed by the system administrator, Organization administrator, and end user, respectively. This chapter focuses on the protection provided at the system level, where service providers can employ backup solutions from vendors of data protection software.

This chapter describes how to design software to back up and restore the vApps in a vCloud. To back up or restore a vApp, you need to deal with both the vCloud configuration and the virtual machines that belong to the vApp. In vSphere, a virtual machine is represented by configuration files and virtual disk files.

Prerequisites

You should be familiar with programming concepts and techniques. You should also be familiar with vCloud, vCloud API, vCloud SDK for .NET, and vSphere concepts. VMware also provides the vCloud SDK for Java and the vCloud SDK for PHP, but this chapter focuses on .NET for the backup and restore examples.

VMware recommends that you design backup and restore software for the vCloud environment using the following APIs:

Table 1. APIs Used To Back Up vApps

Product	API	Data
vCloud Director	vCloud API or vCloud SDK wrapper	vApp metadata
vSphere	WS API	virtual machine configuration
VDDK	VixDiskLib API or VixMntapi	virtual disk contents

You use the vCloud API or SDK to identify vApp targets for backup and restore operations. The vApp metadata identifies the virtual machines that constitute the vApp. You use the WS API to back up and restore virtual machine configurations. You use the VDDK API to back up and restore virtual disk files.

NOTE This chapter uses the term "metadata" in a general sense to mean all the vApp configuration data, in addition to user-defined data that the vCloud REST API serializes in the <Metadata> element.

You should be familiar with the use of the WS API and the VDDK API for backup and restore of individual virtual machines.

Other Information

This chapter does not, in general, duplicate information available in other documents. In particular, this chapter does not provide details about any storage or data protection API that you need to use for backing up and restoring virtual machines in vSphere. You should consult separate reference documentation for details about specific API calls.

This chapter emphasizes the use of the vCloud API and SDK for the purpose of managing metadata of the virtual machines and related artifacts in vCloud Director. The vCloud SDK for .NET translates your C# code into REST operations using the vCloud API.

To learn about VMware vCloud and vSphere concepts and usage, refer to the vCloud Director documentation available from the VMware Web site, http://www.vmware.com/support/sdk_pubs.html. You can also visit the VMware SDK community forum at http://communities.vmware.com/community/vmtn.

Conceptual Overview

This section summarizes the backup and restore processes for vApps managed by vCloud Director. It explains how to use VMware APIs to collect the metadata needed to control backup and restore operations. The actual backup and restore operations are performed using the VMware vStorage APIs for Data Protection (VADP).

VMware vCloud Director uses one or more vCenter servers to manage virtualized resources. At the same time, it manages the vCloud feature of multi-tenancy by maintaining metadata related to various tenant artifacts such as vApp, users, networks, storage, and so on.

When a system administrator chooses to back up a vApp, certain vApp metadata must be retrieved from vCloud Director. The metadata includes general information about the vApp (name, description, virtual machine descriptions), networking information (organization network connectivity, external network connectivity), user information, lease, and quota. This information becomes particularly important when restoring the vApp, in addition to the names of virtual disk files and .vmx files typically retrieved from vSphere using the VADP.

The Backup Process

The backup process requires the backup/restore software to collect and store information both from vCloud Director and from vSphere. This process assumes that you use vCloud Director system administrator credentials to connect to vCloud Director. System administrator credentials allow the software to access vApps belonging to any Organization, and to access all the necessary information about a vApp and associated vCloud constructs.

A vApp in vCloud Director can comprise one or more virtual machines. When you work with a single vApp in vCloud Director, you might be working with a number of virtual machines in vSphere.

To back up a vApp or set of vApps

- 1 Connect to vCloud Director and access the organization where vApp (or vApps) will be backed up.
- When backing up a vApp for a given Organization or VDC in vCloud Director, access the vCloud Director inventory for a list of all desired vApps.
- 3 Select maintenance mode for each vApp to prevent updates during the backup process.
- 4 Collect all the metadata related to the vApp(s), including any user-defined metadata associated with any given vApp.
- 5 Use the vApp metadata to identify the virtual machines associated with each vApp.
- 6 Connect to vCenter Server as a user with sufficient permissions to access the virtual machines. Use the vSphere inventory to locate the virtual machine configuration and virtual disk files.
- 7 Use the VMware APIs for Data Protection to back up the vSphere virtual machine files:
 - a (optional) Save a snapshot of the virtual machine.
 - b Save the virtual machine configuration, using the WS API.
 - c Save the virtual disks using the VDDK API.
 - d (optional) Delete the virtual machine snapshot, if applicable.
- 8 Store the vApp metadata in an appropriate format along with the associated virtual machine files.
- 9 Deselect maintenance mode for each vApp.

The Restore Process

The restore process offers some options to the administrator.

When you restore a vApp, you can choose to overwrite an existing vApp. For instance, the restore software might need to overwrite a vApp with data corruption. You can also choose to restore a vApp that no longer exists, for instance, a vApp that was accidentally deleted.

You can choose whether to keep the same vApp name and other vApp attributes, or you can choose to change attributes during the restore process. If the attributes of the restored vApp no longer conform to the environment because of changes since the backup was taken, you can select new values for the non-conforming attributes.

You might want to restore an existing vApp to an earlier state, or you might want to replace it because it has become corrupted.

To restore an existing vApp

- 1 Identify the child virtual machines of the vApp, using the metadata stored with the backup.
- 2 Connect to vCenter Server as a user with sufficient permissions to access the virtual machines and restore the virtual machines in the vSphere environment. This step restores the virtual disk files and virtual machine configuration. If you are overwriting an existing vApp, you generally restore the files to the same data store that vCloud Director currently uses for the vApp.
- 3 Connect to vCloud Director and authenticate as an administrator, which gives you backup and restore privileges.
- 4 Locate the corrupted vApp, using the ID retrieved from the metadata in the backup store.
- 5 Select maintenance mode for the vApp, to prevent changes while restoring metadata.
- 6 Edit vApp settings such as network, user privileges, lease, and quota as needed. Make sure to include any user-defined metadata from the backup store. If you restored a virtual machine to a different location from the original, you might need to adjust the vApp settings.
- 7 Deselect maintenance mode for the vApp.

You might want to restore a missing vApp because somebody deleted it, or as part of disaster recovery.

To restore a missing vApp

- 1 Identify the child virtual machines of the vApp, using the metadata stored with the backup.
- 2 Connect to vCenter Server as a user with sufficient permissions to access the virtual machines and restore the virtual machines in the vSphere environment. This step restores the virtual disk files and virtual machine configuration.
- 3 Connect to vCloud Director and authenticate as an administrator, which gives you backup and restore privileges.
- 4 Compose a new vApp or import the virtual machine(s) into vCloud Director to create a new vApp with these characteristics:
 - a It has the same name as the lost vApp.
 - b It belongs to the same Organization as the lost vApp.
 - c It obtains resources from the same provider VDC as the lost vApp.
- 5 Select maintenance mode for the vApp, to prevent changes while restoring metadata.
- 6 Edit vApp settings such as network, user privileges, lease, and quota as needed. Make sure to include any user-defined metadata from the backup store.
- 7 Deselect maintenance mode for the vApp.

NOTE This is a simplified view of the restore process. The exact process you use will depend on the features provided by your software. For instance, if the datastore is full, the software could offer to migrate the vApp to a different datastore.

Use Cases Overview

The following sections give an overview of use cases related to the backup and restore processes.

Managing Credentials

Backup software needs vCloud Director access to manage vApps at the metadata level, and vCenter Server access to manage vApps at the virtual machine and virtual disk level. The backup software must collect and retain authentication credentials for both vCloud Director and vCenter Server.

For information about vCloud Director authentication, see "Getting Access to vCloud Director" on page 96. For information about vSphere authentication, see the vSphere Web Services SDK Programming Guide.

Finding a vApp

There are different ways to locate a vApp managed by vCloud Director. One way is to traverse the vCloud Director inventory. Another way is to use the query service.

Inventory Traversal

Using the vCloud Director inventory to locate a vApp requires navigating a hierarchy of containers based on organizational and resource divisions. The process is explained in "Inventory Access" on page 96.

Using the Query Service

The vCloud SDK for .NET also supports the query service of the vCloud API for finding vApps. Consult the sample programs in the SDK for more information about how to use the query service in the SDK.

Protecting Specified vApps

Backup systems typically identify vApps to be backed up in a given Organization based on their identity, using vApp attributes such as name and ID or user defined metadata. A set of vApps to be backed up can also be created based on their Organization (for example, all vApps in the Human Resources Organization), the VDC where they are deployed, and so forth.

In all these cases you must traverse the given Organization and its contents to locate and make a list of vApps.

Recovering an Older Version of a vApp

If a vApp has become corrupted, or if users need to revert to an older state of the vApp, the administrator can restore a version of the vApp from backup storage even when the vApp still exists in vCloud Director. The backup/restore application in these cases can access vCloud Director to get vApp identity information and metadata before restoring the backup copy.

The backup/restore application has a choice between overwriting the current vApp instance or deleting it and creating a new vApp. The choice to delete the vApp can be convenient when the vApp configuration has changed since the last backup, especially when a virtual machine has been added to or deleted from the vApp.

Recovering a Deleted vApp

When recovering a deleted vApp, the backup/restore application must identify the vApp from user input to locate the vApp metadata and virtual machine files on the backup storage medium. After the virtual machines have been restored using vSphere APIs, the vApp can be recomposed using the vCloud API. The backup software must first create a vApp from one of the virtual machines, then import the remaining virtual machines into the same vApp.

Recovering a Single Virtual Machine

The process of recovering a single virtual machine from the backup storage medium is a special case of recovering a deleted vApp. In the case of a deleted vApp, the backup software must re-create the vApp in vCloud Director, then import the remaining virtual machines. For a single lost virtual machine, the backup software must only import the one virtual machine into the existing vApp.

Backing Up vCloud Director

The vCloud SDK for .NET does not offer any special features for backing up or restoring the vCloud Director application and its data. Users should follow standard industry advice for protecting Tomcat applications and Oracle or SOL Server databases.

vCloud API Operations

The following sections describe commonly used vCloud API operations using vCloud SDK for .NET. The API descriptions in this chapter do not provide complete backup/restore implementation details, but focus instead on identifying a set of vCloud API methods that facilitate certain operations that use vCloud Director.

You should be familiar with vCloud Director and vCloud API concepts. Every resource in vCloud Director can be accessed using either its unique ID or HREF (the reference URL) in the vCloud API. The .NET SDK provides wrapper utility classes for commonly-used resources to make the programming easier.

The operations described in the following sections are:

- "Getting Access to vCloud Director" on page 96 Shows how to connect and authenticate with the vCloud API.
- "Inventory Access" on page 96 Shows how to retrieve data for different Organization types.
- "Retrieving Catalog information" on page 100 Shows how to retrieve Catalog entries for backup.
- "Retrieving vApp Configuration" on page 101 Shows how to list virtual machines and vApp configuration data.
- "Preventing Updates to a vApp During Backup or Restore" on page 102 Shows how to use maintenance mode to quiesce vApp configuration.
- "Associating vCloud Resources with vSphere Entities" on page 103 Shows how to get Managed Object References of virtual machines and storage resources from vCloud Director.
- "Restoring vApps" on page 105 Shows how to import virtual machines into vApps.

Getting Access to vCloud Director

The backup/restore software component must use system administrator privileges to connect to vCloud Director, so that it can access any Organization. The system administrator always logs into the System organization. When Administrator@System is used as the user name for the API, Administrator is the login name and System is the System Organization name.

Using system administrator privileges to connect to vCloud Director also allows the backup/restore software to access additional information relating a vApp to the corresponding resources in vSphere. This is described in "Inventory Access" on page 96.

Example 8-1 shows how to log in using C# with the vCloud SDK for .NET. After logging in, the code shows how to access Organization data.

Example 8-1. vCloud Director login code sample using Administrator@System/<password>

```
using com.vmware.vcloud.sdk;
using com.vmware.vcloud.api.rest.schema;

public static vCloudClient client = null;
client = new vCloudClient(vCloudURL, com.vmware.vcloud.sdk.constants.Version.V1_5);
client.Login(username, password);
// Get references to all Organizations:
Dictionary<string,ReferenceType> organizationsMap = client.GetOrgRefsByName();
// Get reference to a specific Organization:
string orgName = "Org1";
ReferenceType orgRef = client.GetOrgRefByName(orgName);
// Convert Organization reference to Organization object:
Organization org = Organization.GetOrganizationByReference(client, orgRef);
```

Inventory Access

In general, you locate a desired vApp for backup in the context of a given Organization and VDC. To locate a vApp that you want to back up, you first need a reference to its parent Organization.

You use the Organization reference to get the Organization object, which you use to get a list of references to the VDCs that belong to the Organization. You use a VDC reference to get a VDC object, which you then use to get a list of references to the vApps that belong to the Organization. You convert the desired vApp reference to a vApp object, which you use to list the virtual machines that belong to the vApp.

Example 8-1 shows how to get a reference to the user view of an Organization. Example 8-2 shows how to get a reference to the admin view of an Organization and a VDC.

Example 8-2. Get Admin Org and Admin VDC

```
public static vCloudClient client = null;
// Login
...
// Get admin view of Org
VcloudAdmin admin = client.GetVcloudAdmin();
string orgName = "Org1";
ReferenceType orgRef = admin.GetAdminOrgRefByName(orgName);
AdminOrganization adminOrg = Organization.AdminGetOrgByReference(client, orgRef);
// Get admin vDC
string vdcName = "VDC1";
ReferenceType vdcRef = adminOrg.GetAdminVdcRefByName(vdcName);
...
AdminVdc adminVdc = AdminVdc.GetAdminVdcByReference(client, vdcRef);
```

Admin Views

The admin view of resources such as Organization, VDC, and vApp provides extra information that is useful to users with administrative privileges. For example, in the case of a vApp, admin view provides information about vCenter and the virtual machines that belong to the vApp. The admin view provides information such as Managed Object References that vCenter uses for those entities. See "Associating vCloud Resources with vSphere Entities" on page 103 for more information about getting vCenter Managed Object References.

To access admin views, you use a method of the client connection object to create an admin client proxy. The admin proxy has methods similar to those of the client connection object to get references to Organizations and other vCloud objects. However, the objects you get from the admin proxy have additional properties not present in user objects.

Admin Extensions

Similar to the admin views, you can use a different method of the client connection object to create an admin extension client proxy. You use the admin extension proxy to find provider VDC. A provider VDC includes one or more resource pools and allocates resources from those pools to the Org VDCs that it supports.

Example 8-3 shows how to get a Provider VDC.

Example 8-3. Get Provider VDC

```
// Login
...
// Get dictionary of Provider vDCs:
AdminExtension.VcloudAdminExtension adminExt = client.GetVcloudAdminExtension();
string pvdcName = "ProvVDC1";
Dictionary<string, ReferenceType> refs = adminExt.GetVMWProviderVdcRefsByName();
...
// Get reference for pvdcName -> pvdcRef
ReferenceType pvdcRef = refs[pvdcName];
VMWProviderVdc vmwPvdc = VMWProviderVdc.GetVMWProviderVdcByReference(client, pvdcRef);
```

Using the vCloud SDK for .NET allows you to access vCloud Director from a C# development environment. These examples show how to use .NET methods. The vCloud SDK for .NET simplifies access to the vCloud API. For more information about using the SDK, see the vCloud SDK for .NET Developer's Guide.

The vCloud API is REST-based. For more information about the vCloud API, see the *vCloud API Programming Guide*. Example 8-4 shows the REST API calls that accomplish the tasks shown in Example 8-1, Example 8-2, and Example 8-3, after logging in.

Example 8-4. REST API Calls To Get Provider VDC

```
GET https://vCloud/api/admin
GET https://vCloud/api/admin/org/id
GET https://vCloud/api/admin/vdc/id

GET https://vCloud/api/admin/extension
GET https://vCloud/api/admin/extension/providervdc/id
```

In general, if you do not need admin views or provider views, you can use an Organization reference to get a VDC reference, and you can use the VDC reference to get a list of vApps belonging to the VDC. Example 8-5 shows how to list the hierarchy of Organizations, VDCs, and vApps known to vCloud Director. This example assumes you have already logged in to vCloud Director.

Example 8-5. List vApps in a VDC for a Given Organization

```
Dictionary<string, ReferenceType> organizationsMap = client.GetOrgRefsByName();
if (organizationsMap != null)
{
   foreach (string organizationName in organizationsMap.Keys)
      ReferenceType organizationReference = organizationsMap[organizationName];
      Organization org = Organization.GetOrganizationByReference(client, organizationReference);
      string OrgID = org.Resource.id;
      Console.WriteLine("Organization Name:" + organizationName);
      Console.WriteLine("Organization Id :" + OrgID);
   foreach (ReferenceType orgRef in organizationsMap.Values)
      Organization org = Organization.GetOrganizationByReference(client, orgRef);
      foreach (ReferenceType vdcRef in org.GetVdcRefs())
         Vdc vdc = Vdc.GetVdcByReference(client, vdcRef);
         string vdcId = vdc.Resource.id;
         Console.WriteLine("Org vDC Id:" + vdcId);
         Console.WriteLine("Org vDC Name:" + vdc.Reference.name);
         foreach (ReferenceType vAppRef in Vdc.GetVdcByReference(client, vdcRef).GetVappRefs())
            Vapp vapp = Vapp.GetVappByReference(client, vAppRef);
            Console.WriteLine("vApp Id:" + vapp.Resource.id);
            Console.WriteLine("vApp Name:" + vapp.Resource.name);
            List<VM> vms = new List<VM>();
            try
               vms = vapp.GetChildrenVms();
            }
            catch
               // Handle exception here
            }
            foreach (VM vm in vms)
               Console.WriteLine("VM Id : " + vm.Resource.id);
               Console.WriteLine("VM Name : " + vm.Resource.name);
         }
      }
  }
}
```

The .NET SDK code in Example 8-5 translates to the API calls shown in Example 8-6

Example 8-6. REST API Calls To List vApps in a VDC for a Given Organization

```
GET https://vCloud/api/admin
GET https://vCloud/api/admin/org/id
GET https://vCloud/api/admin/vdc/id

GET https://vCloud/api/admin/extension
GET https://vCloud/api/admin/extension/providervdc/id
```

You can use a provider VDC reference to enumerate its associated datastores, as shown in Example 8-7. This example assumes you have already logged in to vCloud Director.

Example 8-7. List Datastores

```
/// <summary>
/// Returns list of Provider vDCs.
/// </summary>
/// <returns>ReferenceType</returns>
public static List<ReferenceType> GetProviderVdc()
   List<ReferenceType> vdcRefList = new List<ReferenceType>();
  foreach (ReferenceType vdcRef1 in
      client.GetVcloudAdminExtension().GetVMWProviderVdcRefsByName().Values)
      vdcRefList.Add(vdcRef1);
  }
   return vdcRefList;
}
/// <summary>
/// Returns the list of DataStores
/// </summary>
/// <returns>ReferenceType</returns>
public static List<ReferenceType> GetDataStore()
   extension = client.GetVcloudAdminExtension();
  List<ReferenceType> vmDatastorelist = new List<ReferenceType>();
   foreach (ReferenceType datastoreRef in extension.GetVMwDatastoreRefs())
      vmDatastorelist.Add(datastoreRef);
   return vmDatastorelist;
}
// Get the datastores for the list of Provider vDCs.
foreach (ReferenceType providerVdcRef in GetProviderVdc())
{
   string providerVdcId = GetId(providerVdcRef.href);
  Console.WriteLine("Provider vDC Id:" + providerVdcId);
  Console.WriteLine("Provider vDC Name:" + providerVdcRef.name);
   foreach (string morefitem in
      VMWProviderVdc.GetResourcePoolsByMoref(client, providerVdcRef).Keys)
   {
      Console.WriteLine("Moref :" + morefitem);
   foreach (VMWProviderVdcResourcePoolType VcResourcePool in
      VMWProviderVdc.GetResourcePoolsByMoref(client, providerVdcRef).Values)
      string VcResourcePoolId = GetId(VcResourcePool.ResourcePoolVimObjectRef.VimServerRef.href);
      Console.WriteLine("VcResourcePoolId :" + VcResourcePoolId);
  }
foreach (ReferenceType item in GetDataStore())
```

```
string DatastoreId = GetId(item.href);
Console.WriteLine("Data Store ID:" + DatastoreId);
Console.WriteLine("DataStore:" + item.name);
}
```

Retrieving Catalog information

Catalogs on vCloud Director store vApp templates and ISO images as Catalog items. Backup solutions can be asked to back up the items in the Catalog for a given Organization. Catalogs can be shared or private. A user can choose to back up all items or only selected items in the given catalog. For this it is necessary to traverse the given Catalog in an Organization to access the contents and extract the various metadata associated with the vApp.

Example 8-8 shows inventory traversal to access the Catalog items in a given Organization, and assumes you have already logged in to vCloud Director and obtained a map of Organizations, as in Example 8-1.

Example 8-8. List Catalogs and Catalog Items for a Given Organization

```
Console.WriteLine();
if (organizationsMap != null && organizationsMap.Count > 0)
   foreach (string organizationName in organizationsMap.Keys)
      ReferenceType organizationReference = organizationSMap[organizationName];
      Console.WriteLine(organizationName);
      Console.WriteLine(organizationReference.href);
      Organization organization = Organization.GetOrganizationByReference(client,
               organizationReference);
      List<ReferenceType> catalogLinks = organization.GetCatalogRefs();
      if (catalogLinks != null && catalogLinks.Count > 0)
      {
         foreach (ReferenceType catalogLink in catalogLinks)
            Catalog catalog = Catalog.GetCatalogByReference(client, catalogLink);
            CatalogType catalogType = catalog.Resource;
            Console.WriteLine("
                                  " + catalogType.name);
                                 " + catalogLink.href);
            Console.WriteLine("
            List<ReferenceType> catalogItemReferences = catalog.GetCatalogItemReferences();
            if (catalogItemReferences != null && catalogItemReferences.Count > 0)
            {
               foreach (ReferenceType catalogItemReference in catalogItemReferences)
               {
                  Console.WriteLine("
                                        " + catalogItemReference.name);
                  Console.WriteLine("
                                        " + catalogItemReference.href);
               Console.WriteLine();
            }
            else
            {
               Console.WriteLine("No CatalogItems Found");
         Console.WriteLine();
      }
      else
         Console.WriteLine("No Catalogs Found");
      }
  }
}
else
   Console.WriteLine("No Organizations");
}
```

Example 8-9 shows the REST API calls that accomplish some of the tasks shown in Example 8-8.

Example 8-9. REST API Calls To List Catalog Items

GET https://vCloud/api/catalog/id
GET https://vCloud/api/catalog/id/catalogItems
GET https://vCloud/api/catalogitem/id

Retrieving vApp Configuration

For a typical user, a vApp is the basic unit of backup specified in vCloud Director. The current generation of backup software maps vApps to their associated virtual machines in vSphere, and thus the virtual machine becomes an actual artifact. Virtual disk and virtual machine configuration files need to be stored in a backup. Along with the associated virtual machine artifacts, the user needs to back up the metadata and properties associated with every vApp to successfully restore it in vCloud Director when needed.

When a vApp is lost or deleted from vCloud Director, backup software can restore the vApp by composing a new vApp using virtual machines restored in vSphere. In such a case it becomes imperative to restore the properties and metadata associated with the vApp in vCloud Director.

The SDK includes a number of methods that you can use to get vApp configuration information. Although some of this information is included in the OVF used to upload the vApp to vCloud Director, the information might have subsequently been modified either by using the vCloud API or through the user interface.

All of these methods apply to an object of type Vapp.

Methods To Retrieve vApp Configuration

■ GetChildrenVms()

Gets a list of all child virtual machines that constitute a given vApp. Returns List<VM>.

■ GetStartupSection()

Get virtual machine startup information. Returns StartupSectionType.

■ GetNetworksByName()

Get mapping of all the network sections using their name. Returns Dictionary<string, NetworkSection_TypeNetwork>.

■ GetNetworkConfigSection()

Get network configuration details for a vApp. The information typically contains IP scope (gateway, netmask, DNS settings, IP range), Parent network, Fence Mode settings, and so on. Returns NetworkConfigSectionType.

■ GetLeaseSettingSection()

Get lease settings information. It includes deployment and storage lease settings for the vApp. Returns LeaseSettingsSectionType.

■ GetOwner()

Get owner information for the vApp. Returns ReferenceType.

■ GetMetadata()

Every resource in vCloud API can be associated with user-defined metadata. This method returns user-defined metadata associated with a vApp. Returns MetadataType.

Example 8-10 shows the REST API calls used to get vApp configuration data.

Example 8-10. REST API Calls To Get vApp Configuration

```
GET https://vCloud/api/vapp/id
GET https://vCloud/api/vapp/id/startupSection
GET https://vCloud/api/vapp/id/networkConnectionSection
GET https://vCloud/api/vapp/id/networkConfigSection
GET https://vCloud/api/vapp/id/leaseSettingsSection
GET https://vCloud/api/vapp/id/owner
GET https://vCloud/api/vapp/id/metadata
```

Virtual Machine Information

vCloud Director also stores virtual machine configuration information uploaded from an OVF file into a vApp template. If you have not modified a virtual machine configuration since uploading, you can use this information to verify the configuration of the virtual machine before restoring it.

The following methods, applied to an object of type VM, retrieve configuration data structures from vCloud Director.

Configuration Data for a Virtual Machine

- GetVirtualHardwareSection()
 - Get hardware requirements of the virtual machine. Returns VirtualhardwareSection_Type.
- GetOperatingSystemSection()
 - Get information about the guest operating system installed on this virtual machine. Returns OperatingSystemSectionType.
- GetNetworkConnectionSection()
 - Get information about virtual network devices used by this virtual machine. Returns NetworkConnectionSectionType.
- GetRuntimeInfoSection()

Get version of VMware Tools installed on the virtual machine. Returns RuntimeInfoSectionType.

Example 8-11 shows the REST API calls corresponding to the virtual machine configuration sections available from the SDK for NET

Example 8-11. REST API Calls To Get Virtual Machine Configuration Data

```
GET https://vCloud/api/vapp/id/virtualhardwaresection
GET https://vCloud/api/vapp/id/operatingSystemSection
GET https://vCloud/api/vapp/id/networkConnectionSection
GET https://vCloud/api/vapp/id/runtimeInfoSection
```

Preventing Updates to a vApp During Backup or Restore

While you are backing up or restoring a vApp, you need to prevent updates to the vApp configuration and metadata so that the vApp remains internally consistent. To prevent updates during the backup/restore process, the vCloud API allows the vApp to be placed in maintenance mode, which rejects any new updates to the configuration and metadata.

The backup software must select maintenance mode for the vApp before starting backup or restore operations, and deselect maintenance mode for the vApp after the operations are completed. Example 8-12 shows how to select and deselect maintenance mode for a vApp.

Example 8-12. Protecting a vApp with Maintenance Mode

```
using com.vmware.vcloud.sdk;
using com.vmware.vcloud.api.rest.schema;
...
VApp vapp; // VApp utility class from vCloud SDK
// Identify vApp

vapp.EnableMaintenance(); // Enter maintenance mode

// Perform backup/restore here

vapp.DisableMaintenance(); // Exit maintenance mode
```

Example 8-13 shows the corresponding REST API calls to select and deselect maintenance mode for a vApp.

Example 8-13. REST API Calls To Protect a vApp with Maintenance Mode

```
POST https://vCloud/api/vapp/id/action/enterMaintenancemode
POST https://vCloud/api/vapp/id/action/exitMaintenanceMode
```

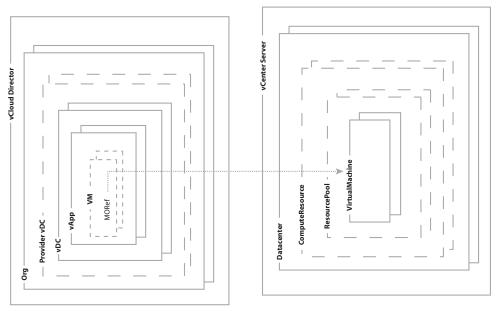
NOTE Selecting maintenance mode does not affect current or pending tasks associated with the vApp. Current or pending tasks will run to completion concurrent with the backup or restore operation. If these tasks involve configuration changes, they could result in an inconsistent vApp configuration. The backup system must ensure that such tasks are complete before storing the vApp properties and metadata.

Associating vCloud Resources with vSphere Entities

The admin view of vCloud Director resources provides additional information about the corresponding entities relevant to the vSphere platform. This information is available only when administrative credentials are used to log in to vCloud Director. The additional information does not replace the use of the vSphere API to provide comprehensive information about the entities. It merely provides the bridge between the vCloud and vSphere by mapping the IDs known to the respective systems.

For example, any given virtual machine is known in vCloud Director by a URN that contains the UUID and resource type. The same resource is identified in vSphere using its native identification, a MoRef (Managed object reference). Additional information provided in the vCloud API makes the necessary link between the two entities by mapping their ID in the two systems. The mapping context is shown in Figure 8-1.

Figure 8-1. Mapping to a Virtual Machine from a vApp



</Vm>

The vCloud API describes the mapping in terms of XML elements, shown in Example 8-14. The box in the example highlights XML data that maps a virtual machine from vCloud Director to vSphere. The MoRef of the virtual machine is in bold type. The object type is shown as VIRTUAL_MACHINE.

Example 8-14. XML Mapping a Virtual Machine URL to a MoRef

```
<Vm needsCustomization="false" deployed="false" status="3" name="RedHat6"</pre>
               id="urn:vcloud:vm:f487ba71-058a-47a9-9e9a-def458c63fd5"
               type="application/vnd.vmware.vcloud.vm+xml"
               href="https://10.20.140.167/api/vApp/vm-f487ba71-058a-47a9-9e9a-def458c63fd5">
   <VCloudExtension required="false">
      <vmext:VmVimInfo>
         <vmext:VmVimObjectRef>
            <vmext:VimServerRef type="application/vnd.vmware.admin.vmwvirtualcenter+xml"</pre>
               name="dao_w2k8_vc"
               href="https://10.20.140.167/api/admin/extension/vimServer/e7026985-19f6-4b9a
               -9d0d-588629e63347"/>
               <vmext:MoRef>vm-63/vmext:MoRef>
               <vmext:VimObjectType>VIRTUAL_MACHINE</vmext:VimObjectType>
         </vmext:VmVimObjectRef>
         <vmext:DatastoreVimObjectRef>
            <vmext:VimServerRef type="application/vnd.vmware.admin.vmwvirtualcenter+xml"</pre>
               name="dao_w2k8_vc"
               href="https://10.20.140.167/api/admin/extension/vimServer/e7026985-19f6-4b9a-9d0d-
               588629e63347"/>
            <vmext:MoRef>datastore-29/vmext:MoRef>
            <vmext:VimObjectType>DATASTORE</vmext:VimObjectType>
         </re></vmext:DatastoreVimObjectRef>
         <vmext:HostVimObjectRef>
            <vmext:VimServerRef type="application/vnd.vmware.admin.vmwvirtualcenter+xml"</pre>
               name="dao_w2k8_vc'
               href="https://10.20.140.167/api/admin/extension/vimServer/e7026985-19f6-4b9a-9d0d-
               588629e63347"/>
            <vmext:MoRef>host-28/vmext:MoRef>
            <vmext:VimObjectType>HOST</vmext:VimObjectType>
         </rd></vmext:HostVimObjectRef>
         <vmext:VirtualDisksMaxChainLength>1/vmext:VirtualDisksMaxChainLength>
      </vmext:VmVimInfo>
   </VCloudExtension>
```

Besides the virtual machine object itself, the VmVIMInfo element encapsulated in the VCloudExtension element of Example 8-14 lists a datastore object and a host object. Each section provides the vSphere entity reference (MoRef) for the corresponding entity, along with its type. The types are DATASTORE and HOST, respectively. In vCloud Director, the virtual machine can be described as virtual machine vm-63 stored in datastore datastore-29 and managed by vCenter Server dao_w2k8_vc.

In a similar way, Example 8-15 shows the administrative view of a VDC wherein the VCloudExtension element provides additional information about the corresponding entities in vSphere. In this particular case, the VDC in the example is based on a resource pool configured in vCenter Server, named dao_w2k8_vc. More information on this server can be obtained by using the vCloud API and its reference URL, which is available as the href property. The MoRef element provides the ID of the resource pool that backs the given VDC, as known to vSphere. Since a MoRef is treated as an opaque value, the VimObjectType element specifies the type of object that the MoRef points to. Combining these elements enables you to use the vSphere API and to locate the Resource Pool served by the specified vCenter Server.

Example 8-15. XML Mapping a Datacenter URL to a MoRef

Example 8-16 shows how to use SDK helper methods to access the vSphere specific information for the virtual machines of a given vApp.

The return value of the methods has type VimObjectRefType, which provides a reference to a vCenter Server, a MoRef to the vSphere entity, and the type of the entity it is referring to.

Example 8-16. Using the SDK for .NET To Access MoRefs

```
using com.vmware.vcloud.sdk;
using com.vmware.vcloud.api.rest.schema;
// Log in with admin privileges and get admin view of vDC containing the vApp.
VApp vapp; // VApp utility class from vCloud SDK
// Identify vApp.
List<VM> Vms:
// Get list of children VM(s)
Vms = vapp.GetChildrenVms();
foreach (VM vm in Vms)
 Console.WriteLine();
  // Access vSphere information for VM
  // VM Info from vSphere
 VimObjectRefType vmRef = vm.GetVMVimRef();
 Console.WriteLine("VirtualMachine: " + vmRef.moRefField);
 // Datastore Info from vSphere for VM
 VimObjectRefType datastoreRef = vm.GetVMDatastoreVimRef();
 Console.WriteLine("Datastore: " + datastoreRef.moRefField);
  // Host info form vSphere for VM
 VimObjectRefType hostRef = vm.GetVMHostVimRef();
 Console.WriteLine("Host: " + hostRef.moRefField);
}
```

Restoring vApps

During the restore process, the backup software typically restores a virtual machine in vSphere using the virtual machine configuration and disk files. In situations where the vApp has been lost from the vCloud Director inventory, the backup software needs to first restore the virtual machine in vSphere, and then import the virtual machine into vCloud Director.

Although the vApp may contain multiple virtual machines in the view of vCloud Director, the virtual machines are known individually to vSphere. To complete the restore operation, the backup software needs to re-create the restored vApp so that all the member virtual machines are created as child virtual machines of the vApp.

To re-create the vApp using the vCloud SDK for .NET, the backup software must use two method calls: ImportVmAsVapp and ImportVmIntoVapp. Use the ImportVmAsVapp method to create a vApp from any one of the child virtual machines. Then call the ImportVmIntoVapp method once for each remaining child virtual machine.

Example 8-17 shows how to use both methods to create a vApp using the vCloud SDK.

Example 8-17. Importing Virtual Machines into vApps

```
/// <summary>
/// Reference to hold the vCloud Client reference
/// </summary>
private static VcloudAdminExtension extension = null;
vcloudClient.login(user, password);
extension = vcloudClient.getVcloudAdminExtension();
// Get references for known VIM Servers
Dictionary<string, ReferenceType> vimServerRefsByName = extension.GetVMWVimServerRefsByName();
// Select VIM Server Reference
VMWVimServer vimServer = VMWVimServer.GetVMWVimServerByReference(
                            vcloudClient, vimServerRefsByName[vimServerName]);
// Import first VM from VIM server as vApp:
ImportVmIntoVAppParamsType importVmIntoVAppParamsType = new ImportVmIntoVAppParamsType();
importVmIntoVAppParamsType.vmMoRefField = moref; // vSphere ID from backup data.
importVmIntoVAppParamsType.vdcField = vdcRef; // vDC where the new vApp will be created.
Vapp vapp = vimServer.ImportVmAsVApp(importVmAsVAppParamsType); // Task is embedded in vapp.
foreach (VM vm in vms)
  // Import remaining VMs from VIM Server into existing vApp:
  importVmIntoVAppParamsType.vmMoRefField = moref; // vSphere ID from backup data.
  importVmIntoVAppParamsType.vAppField = vapp; // vApp to hold restored VMs.
  Task task = vimServer.ImportVmIntoVApp(importVmIntoVAppParamsType);
};
```

Example 8-18 shows the corresponding REST API calls used to rebuild a vApp in vCloud Director.

Example 8-18. REST API Calls To Restore a vApp

```
POST https://vCloud/api/admin/extension/vimServer/id/importVmAsVApp
POST https://vCloud/api/admin/extension/vimServer/id/importVmIntoExistingVApp
```

Conclusion

This chapter provided an overview of how to use the vCloud SDK for .NET to back up and restore vApps in vCloud Director. This information serves as a guide to using the vCloud SDK for writing backup and restore software. Other documentation is require to supplement aspects not described in this chapter.

The examples in this chapter are not intended to be complete. They are intended only to illustrate the method calls you would use during backup and restore operations with vCloud Director and vCenter Server. For more detail about the SDK methods and examples of their use, see the vCloud SDK for .NET Developer's Guide.

Virtual Disk Mount API



To perform file based restore, you can use the disk mount API (vixMntapi) for local and remote mounting of virtual disks. VixMntapi involves a separate library for programs to load.

- "The VixMntapi Library" on page 107
- "Programming with VixMntapi" on page 112
- "Sample VixMntapi Code" on page 113
- "Restrictions on Virtual Disk Mount" on page 113



CAUTION The vixMntapi library for Windows supports HotAdd and SAN transport, but the vixMntapi library for Linux supports only file and NBDSSL transport when libdiskLibPlugin.so is not loaded.

The VixMntapi Library

The VixMntapi library supports guest operating systems on multiple platforms. On POSIX systems it requires FUSE mount, available on recent Linux systems, and freely available on the SourceForge Web site. Definitions are contained in the following header file, installed in the same directory as vixDiskLib.h:

#include "vixMntapi.h"

Types and Structures

This section summarizes the important types and structures.

Operating System Information

The VixOsInfo structure encapsulates the following information:

- Family of the guest operating system, VixOsFamily, one of the following:
 - Windows (NT-based)
 - Linux
 - Netware
 - Solaris
 - FreeBSD
 - OS/2
 - Mac OS X (Darwin)
- Major version and minor version of the operating system
- Whether it is 64-bit or 32-bit
- Vendor and edition of the operating system
- Location where the operating system is installed

Disk Volume Information

The VixVolumeInfo structure encapsulates the following information:

- Type of the volume, VixVolumeType, one of the following:
 - Basic partition.
 - GPT GUID Partition Table.
 - Dynamic volume, including Logical Disk Manager (LDM).
 - LVM Logical Volume Manager disk storage. Not supported with Linux.
- Whether the guest volume is mounted on the proxy.
- Path to the volume mount point on the proxy, or NULL if the volume is not mounted.
- On Windows, numGuestMountPoints is the number of times a basic volume is mapped to a drive letter, or 0 if the volume is not mounted. IDE and boot disk come first. Unimplemented on Linux.
- Mount points for the volume in the guest.

Function Calls

To obtain these functions, load the vixMntapi library separately from the vixDiskLib library. On Windows, compile with the vixMntapi.lib library so your program can load the vixMntapi.dll runtime.

These calls can be used to mount and read Windows virtual disks on Windows hosts (with at least one NTFS volume) or Linux virtual disks on Linux hosts. Cross-mounting is restricted, though it is possible to mount a virtual disk with a mix of formats, if the mounted partition was formatted with Windows.

You should run only one vixMntapi program at a time on a virtual machine, to avoid conflict between registry hives. See "Multithreading Considerations" on page 42 for advice on worker threads.

Available functions in the library are listed below. Under parameters, [in] indicates input, and [out] indicates output parameters. Functions that return vixError return VIX_OK on success, otherwise a VIX error code.

VixMntapi_Init()

Initializes the library. This is similar to VixDiskLib_InitEx() – see "Initialize Virtual Disk API" on page 36. You should call VixMntapi_Init() only once per process.

Parameters:

- majorVersion [in] VixMntapi major version number, currently must be 1 (one).
- minorVersion [in] VixMntapi minor version number, currently must be 0 (zero).
- log [in] Callback function to write log messages.
- warn [in] Callback function to write warning messages.
- panic [in] Callback function to report fatal errors.
- libDir [in] and configFile [in] as for VixDiskLib_InitEx(), allowing you to tmpDirectory.

VixMntapi_Exit()

Cleans up the VixMntapi library. You should call VixMntapi_Exit() only once per process void VixMntapi_Exit();

VixMntapi OpenDiskSet()

Opens the set of disks for mounting on a Windows virtual machine. All the disks for a dynamic volume or Logical Disk Manager (LDM) must be opened together.

The VixDiskLibHandle type, defined in vixDiskLib.h, is the same as for the diskHandle parameter in the VixDiskLib_Open() function, but here it is an array instead of a single value.

Parameters:

- diskHandles [in] Array of handles to open disks.
- numberOfDisks [in] Number of disk handles in the array.
- openMode [in] Must be 0 (zero).
- diskSet [out] Disk set handle to be filled in.

If you want to mount disks on a Windows system, first call VixDiskLib_Open() for every disk, then use the returned disk handle array to call VixMntapi_OpenDiskSet(), which returns a disk set handle.

If you want to mount disks on a Linux system, call the function VixMntapi_OpenDisks(), which opens and creates the disk set handle, all in one function.

VixMntapi_OpenDisks()

Opens disks for mounting on a Linux virtual machine, or disk sets on a Windows virtual machine. VixMntapi supports whatever file systems the proxy supports. On Linux, VixMntapi does not support Logical Volume Manager (LVM), although vendors can write their own scripts to support LVM disks.

VixError

Parameters:

- connection [in] The VixDiskLibConnection to use for opening the disks. Calls VixDiskLib_Open() with the specified flags for each disk to open.
- diskNames [in] Array of disk names to open.
- numberOfDisks [in] Number of disk handles in the array. Must be 1 for Linux.
- flags [in] Flags to open the disk.
- handle [out] Disk set handle to be filled in.

VixMntapi_GetDiskSetInfo()

Retrieves information about the disk set.

Parameters:

- handle [in] Handle to an open disk set.
- diskSetInfo [out] Disk set information to be filled in.

VixMntapi_FreeDiskSetInfo()

Frees memory allocated by VixMntapi_GetDiskSetInfo().

void VixMntapi_FreeDiskSetInfo(VixDiskSetInfo *diskSetInfo);

Parameter:

diskSetInfo [in] OS info to be freed.

VixMntapi_CloseDiskSet()

Closes the disk set. Note, vixDiskLib_Disconnect() invalidates open file handles, so call this function first.

VixError

VixMntapi_CloseDiskSet(VixDiskSetHandle diskSet);

Parameter:

■ diskSet [in] Handle to an open disk set.

VixMntapi_GetVolumeHandles()

Retrieves handles to volumes in the disk set. The third parameter VixVolumeHandle can be a volume handle or an array of volume handles. If you pass an array this function returns the volume handle for the first volume only. If you pass a pointer (such as VixVolumeHandle *volumeHandles) it returns all the volume handles.

VixError

Parameters:

- diskSet [in] Handle to an open disk set.
- numberOfVolumes [out] Number of volume handles.
- volumeHandles [out] Volume handles to be filled in.

VixMntapi_FreeVolumeHandles()

Frees memory allocated by VixMntapi_GetVolumeHandles().

void VixMntapi_FreeVolumeHandles(VixVolumeHandle *volumeHandles);

Parameter:

■ volumeHandles [in] Volume handle to be freed.

VixMntapi_GetOsInfo()

Retrieves information about the default operating system in the disk set.

To get operating system information, VixMntapi_GetOsInfo() requires the system and boot volumes to be already mounted. It does not dismount the system volume at the end of this function. Your application should be prepared to handle the "volume already mounted" error gracefully.

This function is effective only when used with operating systems of the same type. For instance, a VixMntapi program running on Windows can provide information about the VMDK of a Windows virtual machine, but not about the VMDK of a Linux virtual machine.

VixError

Parameters:

- diskSet [in] Handle to an open disk set.
- info [out] OS information to be filled in.

VixMntapi_FreeOsInfo()

Frees memory allocated by VixMntapi_GetOsInfo().

void VixMntapi_FreeOsInfo(VixOsInfo* info);

Parameter:

■ info [in] OS info to be freed.

VixMntapi_MountVolume()

Mounts the volume. After mounting the volume, use VixMntapi_GetVolumeInfo() to obtain the path to the mounted volume. This mount call locks the source disks until you call VixMntapi_DismountVolume(). The VixMntapi_MountVolume() function cannot mount Linux swap or extended partitions.

VixError

Parameters:

- volumeHandle [in] Handle to a volume.
- isReadOnly [in] Whether to mount the volume in read-only mode. Does not override openMode.

VixMntapi DismountVolume()

Unmounts the volume.

VixError

Parameters:

- volumeHandle [in] Handle to a volume.
- force [in] Force unmount even if files are open on the volume.

VixMntapi_GetVolumeInfo()

Retrieves information about a disk volume. Some information, such as the number of mount points, requires you to set the open read-only flag. Some information is available only if a volume was previously mounted by VixMntapi_MountVolume(). The Windows registry returns volume information only for mounted disks. On Windows the VixMntapi_GetVolumeInfo() call returns a symbolic link from the VixVolumeInfo structure in the form \\.\vstor2-mntapi10-shared-<longhexnum>\. You can transform this symbolic link into a target path by replacing \\. with \Device and deleting the final backslash, then map a drive letter with DefineDosDevice(DDD_RAW_TARGET_PATH,...) and proceed as if you have a local drive. Alternatively on Windows, you can open a volume with CreateFile() and traverse the file system with FindFirstFile().

VixError

Parameters:

- volumeHandle [in] Handle to a volume.
- info [out] Volume information to be filled in.

VixMntapi_FreeVolumeInfo()

Frees memory allocated in VixMntapi_GetVolumeInfo().

void VixMntapi_FreeVolumeInfo(VixVolumeInfo *info);

Parameter:

■ info [in] Volume info to be freed.

Programming with VixMntapi

At the top of your program, include vixMntapi.h along with any other header files you need. Structures and type definitions are declared in the include file, so you do not need to create them or allocate memory for them.

Call VixMntapi_Init() to initialize the library in your application. This function takes major and minor version number to account for future extensions. You can provide your own logging, warning, or panic functions to substitute for the default VixMntapi handlers, and custom library and temporary directories.

Call VixMntapi_OpenDiskSet() to open a set of virtual disks for mounting. Pass a set of disk handles obtained from the VixDiskLib_Open() call. The VixMntapi_OpenDiskSet() function also expects number of disks to open, an optional open mode, and a parameter to pass back the resulting disk-set handle.

File System Support

Traditional Windows file systems (including FAT, FAT32, and NTFS) are supported. ReFS is supported but read-only for VixMntapi. Linux file systems (including ext2, ext3, ext4) are supported if the proxy machine supports them. VixMntapi depends on an operating system for file attributes such as compression, encryption, hidden files, ACL, and alternate streams. If a vixMntapi-linked application runs on a virtual machine that supports these attributes, vixMntapi can support them. Windows basic volumes and simple dynamic volumes are supported for writing, but spanned, striped (RAID-0), mirrored (RAID-1) and parity (RAID-5) volumes are supported read-only for VixMntapi.

You must open a disk set read/write to obtain the OS information for dynamic volume types including LDM and LVM. If you cannot open a base disk read/write, create a child disk in front, and open it read/write. In a multi-boot setup, only the first entry #0 is opened.

The order of mounting is important. For instance, mount top-level directories before subdirectories, and drives with dependencies after drives that they depend on. Mount points are not enumerated, nor are they restored. When you mount one volume, the other volumes are not implicitly mounted also.

Read-Only Mount on Linux

Linux vixMntapi does not support read-only access. It is explicitly disabled in the code due to journal replay requirements when mounting ext3 and later file systems. To mount a disk read-only, you must either mount the virtual disk of a powered off virtual machine, or mount the snapshot of a powered on virtual machine. Here are possible use cases for read-only virtual disk mounts:

- Avoid accidentally writing data to the disk, which is opened read/write mode.
- Retain current disk data after the VM is powered off.
- Use previous VM as a restore point by reverting to its original snapshot.

The VixMntapi library can emulate mounting disks read-only, as in the Linux procedure below. A similar procedure could be used on Windows.

To mount disks while saving a read-only copy

- 1 Power off the VM if it is on.
- 2 Create a snapshot. Optional: revert to a previous snapshot if desired.
- 3 Connect to the VM with VixDiskLib_Connect or VixDiskLib_ConnectEx for read/write access.
- 4 Call VixMntapi_OpenDisks to open the current (not the snapshot) VM disk in read/write mode.
- Mount volumes to access files. Write operations can be performed, but they write to the redo log of the current VM disk, not to the snapshot.
- 6 Close the disk and disconnect.
- 7 Revert to the snapshot created in step 2, to eliminate any write changes. Delete the snapshot.
- 8 Power on the VM if it was on before.

Sample VixMntapi Code

You call the VixMntapi functions after initializing VixDiskLib, connecting to a virtual machine, and opening a disk handle. Example A-1 shows test code for Windows with the correct order of function calls.

Example A-1. Test source code for VixMntapi functions

```
MountTest() {
   vixError = VIX_ERR_CODE(VixDiskLib_Init() );
    vixError = VIX_ERR_CODE(VixMntapi_Init() );
    VixDiskLib_ConnectEx(&connectParams, TRUE, NULL, NULL, &connection));
    diskHandles = GetMyDiskHandles(diskPaths, connection, &connectParams, flags, &numberOfDisks);
    vixError = VIX_ERR_CODE(VixMntapi_OpenDiskSet(diskHandles, numberOfDisks, flags, &diskSet));
    GetOsInfo(diskSet);
   vixError = VIX_ERR_CODE(VixMntapi_GetVolumeHandles(diskSet, &numberOfVolumes,
                   &volumeHandles));
    for(size_t i = 0; i < numberOfVolumes; i++) {</pre>
        VixVolumeHandle volumeHandle = volumeHandles[i];
        VixVolumeInfo *volumeInfo;
        vixError = VIX_ERR_CODE( VixMntapi_MountVolume(volumeHandle, TRUE) );
        vixError = VIX_ERR_CODE( VixMntapi_GetVolumeInfo(volumeHandle, &volumeInfo) );
        VixMntapi_FreeVolumeInfo(volumeInfo);
        VerifyMountedVolume();
        CleanUpMountedVolume(volumeHandle, volumeInfo);
   VixMntapi_FreeVolumeHandles(volumeHandles);
    vixError = VIX_ERR_CODE( VixMntapi_CloseDiskSet(diskSet) );
    FreeMyDiskHandles(diskHandles, numberOfDisks);
   VixMntapi_Exit();
   VixDiskLib_Exit();
}
```

Restrictions on Virtual Disk Mount

The following limitations apply when mounting virtual disks:

- You cannot mount virtual disks that are in use by a running or suspended virtual machine. You can mount virtual disks from any powered off virtual machine, or base disks when a Windows virtual machine is running read-only off a snapshot.
- You can mount the last snapshot in a chain read/write, but you must mount previous snapshots read-only.
- On Linux virtual machines before VDDK 5.5, you could not mount previous snapshots in the chain.
- If you specify a virtual disk with snapshots on a powered off virtual machine, VixMntapi locates and mounts the last snapshot in the disk chain. While a disk is mounted, do not revert to a previous snapshot using another VMware interface this would make it impossible to unmount the partition.
- You cannot mount virtual disk if any of its .vmdk files are encrypted, compressed, or read-only. However you can change these attributes and then mount the virtual disk.
- With Windows, you must mount virtual disks on drive D: or greater, and choose a drive letter not in use.
- With Linux, kernel version 2.6 or higher is required to run the FUSE (file system in user space) module. You cannot mount Linux swap or extended partitions. Logical Volume Manager (LVM) is not supported.
- You can mount Windows virtual disks on Windows hosts (with an NTFS volume) or Linux virtual disks on Linux hosts. Cross-mounting is restricted but may be allowed for cross-formatted file systems.
- The C: boot driver should be on scsi:0:0, and all disks should be opened in SCSI order (0:0, 0:1, 0:2, 1:0, 1:1, 1:2, etc.) before mounting any of them.

Virtual Disk Development Kit Programming Guide

Errors Codes and Open Source



This appendix contains the following sections:

- "Recent Changes" on page 115
- "Finding Error Code Documentation" on page 115
- "Troubleshooting Dynamic Libraries" on page 116
- "Open Source Components" on page 116

Recent Changes

In vSphere 6.0, the VDDK libraries were changed to return VIX_E_OUT_OF_MEMORY when an ESXi host runs out of memory for network file copy (NFC). Previously, VDDK would hang during I/O operations.

Error 2 - VIX_E_OUT_OF_MEMORY

Memory allocation failed. Out of memory.

Finding Error Code Documentation

For a list of Virtual Disk API error codes, see the online reference guide Introduction to the VixDiskLib API:

- Windows C:\Program Files\VMware\VMware Virtual Disk Development Kit\doc\intro.html
- Linux /usr/share/doc/vmware-vix-disklib/intro.html

In a Web browser, click the **Error Codes** link in the upper left frame, and click any link in the lower left frame. The right-hand frame displays an alphabetized list of error codes, with explanations.

Association With VIX API Errors

The Virtual Disk API shares many errors with the VIX API, which explains the VIX prefix. The error codes for the VIX API are likely to be the same, or almost the same as, a comparable release of the VDDK.

For information about the VIX API, including its online reference guide to functions and error codes, see the developer support section of the VMware Web site:

https://www.vmware.com/support/developer/vix-api/index.html

Interpreting Errors Codes

A VIX error is a 64-bit value. A value of VIX_OK indicates success, but otherwise (if there is an error), several bit regions in the 64-bit value might be set. The least significant 16 bits are set to the error code described for VIX errors. More significant bit fields might be set to other values.

As for the VIX API, use the macro VIX_ERROR_CODE(err) to mask off bit fields not used by the VDDK.

Troubleshooting Dynamic Libraries

On Windows, the SSL library is placed in the same directory as other vixDiskLib dynamically loaded libraries. On Linux, functions that load the libraries libssl.so.0.9.8 and libraryeto.so.0.9.8 do the following:

- 1 Attempt to load them from the environment's LD_LIBRARY_PATH location.
- 2 Next, attempt to load them from the directory where libvixDiskLib.so is located.
- 3 Next, attempt to load them from the directory where the executable is located.
- 4 Failing that, exit with an error.

On install, VDDK creates the directory /usr/lib/vimware-vix-disklib, populated with 64-bit executables and libraries placed into subdirectories bin64 and lib64. On determining the OS type, VDDK copies the vixDiskLib and vixMntapi libraries into /usr/lib. It does not copy libssl.so.0.9.8 or libcrypto.so.0.9.8 into /usr/lib.

On execution, the root user normally has no LD_LIBRARY_PATH, and /usr/lib is ahead of /opt/vmware/lib in the path. Running the ldd command can help diagnose where a program is getting libvixDiskLib.so and other libraries. The /opt/vmware/lib directory is neither created nor updated by the VDDK install script.

If you see the error "Failed to load library librrypto.so.0.9.8" there are several solutions:

- Set or reset the LD_LIBRARY_PATH environment so it contains one of the directories above, /lib64 and possibly /bin64, before it contains /usr/lib.
- Change the symbolic link in /opt/vmware/lib (or elsewhere) so it points to the directory above, /lib64.
- Copy the libssl and libcrypto libraries from /usr/lib/vmware-vix-disklib/lib64 into /usr/lib.

Open Source Components

VDDK contains the following open source components, with license types indicated:

- Boost (BSD style license)
- Curl (MIT/X derivative license)
- Expat (BSD style license)
- FreeBSD (BSD license)
- ICU, International Components for Unicode (BSD style license)
- LibXML2 (MIT style license)
- OpenLDAP (OpenLDAP v 2.8 license)
- OpenSSL (OpenSSL license)
- Zlib (BSD license)

These open source components have the GNU library general public license:

- GetText (LGPL2.0)
- Glib (LGPL 2.0)
- LibFuse (LGPL2.0)
- LibIconv (LGPL2.0)

Glossary

C changed block tracking (CBT)

A VMware feature that keeps track of which blocks in a virtual disk changed since the time of inception, or when CBT was last enabled. Developers of backup and recovery software can use the CBT feature to help reduce data size for differential or incremental backup.

D differential backup

Saving system data changed since the last full backup, so only two restore steps are necessary.

E extent

In the context of VMDK, a split portion of virtual disk, usually 2GB.

F flat

Space in a VMDK is fully allocated at creation time (pre-allocated). Contrast with sparse.

H hosted disk

A virtual disk stored on a hosted product, such as VMware Workstation, for its guest operating system.

I incremental backup

Saving system data changed since the last backup of any type.

M managed disk

A virtual disk managed by an ESXi host or VMware vCenter, contained within a VMFS volume.

monolithic

The virtual disk is a single VMDK file, rather than a collection of 2GB extents. Contrast with split.

N NBDSSL

Network block device, a standard method for over-the-network access, with SSL encryption.

P proxy

A physical or virtual machine running an operating system with third-party backup software. The proxy is used to perform file-level and image-level virtual machine backups.

Q quiescing

A method of bringing the on-disk data of a physical or virtual computer into a state suitable for backups. Quiescing may include flushing disk buffers from the operating system's in-memory cache or other tasks.

R RDM (Raw Device Mapping)

Enables a virtual machine to directly access a LUN on the physical storage subsystem (SAN connected by Fibre Channel, iSCSI, or SAS). At the same time, the virtual machine has access to the disk that is using a mapping file in the VMFS name space. Performance is similar to VMDK.

S sparse

Space in a VMDK is allocated only when needed to store data. Contrast with flat.

split

The virtual disk is a collection of VMDK files containing 2GB extents. Contrast with monolithic.

V VMDK (Virtual Machine DisK)

The virtual counterpart to a guest operating system's physical disk. A file or group of files that can reside on the host machine or on a remote file system.

VMFS (Virtual Machine File System)

A file system optimized for storing virtual machines. One VMFS partition is supported for each SCSI storage device or LUN.

VMX (virtual machine configuration file)

A file containing a virtual machine's configuration. This .vmx file is created with its virtual machine and is used to identify and run a specific virtual machine.

Index

Numerics 32-bit and 64-bit 12 iconv_open() initialize codeset conversion 21 iconv() convert UTF-8 to and from UTF-16 21 Α incremental backup 41, 117 access and credentials 22 incremental restore 81 application consistent quiescing, Windows 88 installation on Linux 16 installation on Windows 16 В internationalization (i18n) 21 backing information for virtual disk 68 backup algorithms 40 L backup process, overview of 57 lazy zeroed thick disk 20, 83 best practices for SAN transport 82 licensing for advanced transports 38 BIOS customizations, saving 54 limitations of vixMntapi library 113 linked clone backup 33 C Linux installation 16 change ID 41 localization (I10n) 21 changed block tracking (CBT) 51, 70 code sample walk-through 45 М configFile settings for VixDiskLib_InitEx 36 managed disk 13, 19, 28, 42, 48, 51, 117 CopyThread 46, 48 managed object reference (moRef) 59 credentials and access 22 Microsoft Shadow Copy 86 monolithic VMDK 19, 20, 32, 47, 48, 50, 117 MONOLITHIC_FLAT 19, 20 development platforms 15 MONOLITHIC_SPARSE 19, 20 differential backup 41 moref and unique ID 59 disassociate host from vCenter Server 81 mounting virtual disk for backup 63 disaster recovery 40 disk mount (vmware-mount) 12 Ν nonpersistent disk mode 20 Е NVRAM backup and restore 54 eager zeroed thick disk 20, 83 error codes, finding explanations for 115 0 ESXi and vCenter Server 22 ObjectSpec and PropertySpec 59 ESXi and VMware vCenter 11 open read-only disk 112 EUC (ISO-2022) for Chinese, Japanese, Korean 21 open source components in VDDK 116 extent 19, 35, 50, 117 P packaging of Virtual Disk API 16 flat VMDK 19, 20, 34, 35, 117 persistent disk mode 20 platforms supported for development 15 privileges needed in vCenter Server 22 gcc (GNU C compiler) 15 products from VMware that are supported 15 property collector, use of 64 PropertySpec and ObjectSpec 59 hosted disk 13, 19, 28, 30, 42, 46, 48, 51, 55, 117

VMware, Inc. 119

Q

queryChangedDiskAreas 41

R	Visual Studio 15
RDM (raw device mapping) 31, 54, 60, 72, 73, 81	VixDiscoveryProc 55
read-only disk mount 112	VIXDISKLIB_ADAPTER_IDE 23, 50
re-creating a virtual machine from backup 63	VIXDISKLIB_ADAPTER_SCSI_BUSLOGIC 23, 50
redo logs and snapshots 20, 22, 32, 53	VIXDISKLIB_ADAPTER_SCSI_LSILOGIC 23
restore process, overview of 61	VixDiskLib_Attach 28, 32, 47, 53
restoring a virtual machine to previous state 62	VixDiskLib_CheckRepair 31
restrictions on vixMntapi library 113	VixDiskLib_Clone 28, 34, 42, 48, 51
role and privileges in vCenter Server 22	VixDiskLib_Close 28, 30, 46
•	VixDiskLib_Connect 28, 29, 42, 46, 48, 52
S	VixDiskLib_Create 28, 30, 42, 47, 50, 51, 53
sample program walk-through 45	VixDiskLib_CreateChild 28, 32, 33, 47
SAN and the Virtual Disk API 13, 27	VIXDISKLIB_CRED_UID 52
SAN transport on VVol datastores 24, 82	VixDiskLib_Defragment 28, 35, 42
SCSI controller and Hot Add 84	VixDiskLib_Disconnect 28, 35
session object for server communication 58	VIXDISKLIB_DISK_SPLIT_SPARSE 50
snapshot	VixDiskLib_EndAccess 28, 39
backing up a virtual disk 69	VixDiskLib_Exit 28, 35
creating 69	VixDiskLib_FreeErrorText 28, 34
deleting 70	VixDiskLib_FreeInfo 28, 31, 46
restoring virtual disk 73	VixDiskLib_GetErrorText 28, 34
temporary 60	VixDiskLib_GetInfo 28, 31, 46, 49, 50
snapshot management 53	VixDiskLib_GetMetadataKeys 28, 31, 47
snapshots and redo logs 20, 22, 32, 53	VixDiskLib_Grow 28, 35, 42
sparse VMDK 19, 20, 32, 35, 44, 47, 48, 50, 118	VixDiskLib_Init 28, 29, 36, 46, 52, 55
split VMDK 19, 20, 50, 118	VixDiskLib_Open 28, 30, 46
SPLIT_FLAT 19, 20	VixDiskLib_PrepareForAccess 39
SPLIT_SPARSE 19, 20	VixDiskLib_Read 28, 30, 48, 49, 50, 52
SSL certificate 26, 37, 48	VixDiskLib_ReadMetadata 28, 31, 47
STREAM_OPTIMIZED 19	VixDiskLib_Rename 28, 34, 52
supported platforms for development 15	VIXDISKLIB_SECTOR_SIZE 30, 49, 52
supported VMware products 15	VixDiskLib_Shrink 28, 35
T	VixDiskLib_SpaceNeededForClone 28, 34
T	VixDiskLib_Unlink 28, 35, 42, 52
thin provisioned disk 21	VixDiskLib_Write 28, 30, 35, 47, 51, 52
thumbprint, SSL 26, 37, 48	VixDiskLib_WriteMetadata 28, 31, 47
U	VixHost_Connect 55
	VixHost_FindItems 55
UEFI (unified extensible firmware interface) 54	VM configuration information 40
Unicode UTF-8 support 21 unique ID and moRef 59	VMDK (virtual machine disk) file 11, 12, 19, 21, 22, 31, 46, 47, 49, 50, 51, 52, 56
V	VMFS_FLAT 19, 20, 51
	VMFS_SPARSE 19
vCenter Server and ESXi 22	VMFS_THIN 19, 20
vCloud and vCloud Director 12	VMware Consolidated Backup (VCB) 11
verify SSL certificate 26, 37, 48	VMX specification (vmxSpec) 29
versions of VMDK files 51	Volume Snapshot Service (VSS) 86
VHD from Microsoft 22	vSAN with direct attached storage 24
Vim find_entity_views in Perl toolkit 56	vSphere APIs for Data Protection (VADP) 57
Vim get_inventory_path in Perl toolkit 56	VVol datastores 62, 73, 81
virtual disk manager (vmware-vdiskmanager) 12	VVol datastores and SAN transport 24, 82
VirtualMachineConfigInfo 40 VirtualMachineConfigSpec 74	
virtualiviaci ili lecci ili gopec 14	

W

walk-through of sample program 45
Windows 2008 application consistent quiescing 88
Windows installation 16
Windows On Windows 64 12

Virtual Disk Development Kit Programming Guide