Contents

B

C

D

E

F

G

H

I

L

N

O

P

R

S

T

V

W

Volume Shadow Copy Service

6/3/2021 • 2 minutes to read • Edit Online

Purpose

The Volume Shadow Copy Service (VSS) is a set of COM interfaces that implements a framework to allow volume backups to be performed while applications on a system continue to write to the volumes.

For an introduction to VSS for system administrators, see Volume Shadow Copy Service in the TechNet Library.

Run-time requirements

VSS is supported on Microsoft Windows XP and later. For information about run-time requirements for a particular programming element, see the Requirements section of the documentation for that element.

All 32-bit VSS applications (requesters, providers, and writers) must run as native 32-bit or 64-bit applications. Running them under WOW64 is not supported. For more information, see Supported Configurations and Restrictions.

Windows Server 2003 and Windows XP: Running 32-bit VSS requesters under WOW64 is supported, but not for system-state backups. Running 32-bit VSS providers and writers under WOW64 is not supported. Support for running 32-bit requesters under WOW64 was removed in Windows Vista and subsequent versions.

> **NOTE**
> A shadow copy that was created on Windows Server 2003 R2 or Windows Server 2003 cannot be used on a computer that is running Windows Server 2008 R2 or Windows Server 2008. A shadow copy that was created on Windows Server 2008 R2 or Windows Server 2008 cannot be used on a computer that is running Windows Server 2003. However, a shadow copy that was created on Windows Server 2008 can be used on a computer that is running Windows Server 2008 R2, and vice versa.

In this section

TOPIC	DESCRIPTION
Overview	Describes the VSS object model, backup and restore strategies, and how to create VSS providers, requesters, and writers.
Reference	Describes VSS classes, data types, enumerations, functions, interfaces, and structures.

Additional resources

RESOURCE	DESCRIPTION
Windows Vista and later	VSS is available in the Microsoft Windows Software Development Kit (SDK). You can install the SDK for Windows 7 and Windows Server 2008 R2 from the Windows Download Center. You can also download the ISO version of the SDK from the Windows Download Center. Previous versions of the SDK can be downloaded from the Windows SDK Download Page.
Windows Server 2003 and Windows XP	VSS is available in the Volume Shadow Copy Service 7.2 SDK, which you can download from the Windows Download Center. Note that the 64-bit vssapi.lib files in the directories under the Win2003\Obj directory can be used for the 64-bit versions of Windows Server 2003 and Windows XP.

Volume Shadow Copy Service Overview

3/5/2021 • 2 minutes to read • Edit Online

The Volume Shadow Copy Service (VSS) is a set of COM APIs that implements a framework to allow volume backups to be performed while applications on a system continue to write to the volumes.

VSS provides a consistent interface that allows coordination between user applications that update data on disk (*writers*) and those that back up applications (*requesters*).

- What's New in VSS
- About the Volume Shadow Copy Service
- Using the Volume Shadow Copy Service
- VSS Implementation Details
- Troubleshooting VSS Applications

What's New in VSS

3/5/2021 • 2 minutes to read • Edit Online

The following topics describe the changes in each release of the VSS API:

- What's New in VSS in Windows Server 2008 R2 and Windows 7
- What's New in VSS in Windows Server 2008 and Windows Vista SP1
- What's New in VSS in Windows Vista
- What's New in VSS in Windows Server 2003 SP1
- What's New in VSS in Windows Server 2003

Note that all changes for Windows Vista also apply to Windows Server 2008 and Windows Vista with Service Pack 1 (SP1).

What's New: VSS in Windows Server 2008 R2 & Windows 7

3/5/2021 • 2 minutes to read • Edit Online

Windows Server 2008 R2 and Windows 7 introduce the following changes to the Volume Shadow Copy Service.

New VSS Interfaces

IVssBackupComponentsEx3
IVssComponentEx2
IVssCreateExpressWriterMetadata
IVssExpressWriter

New VSS Classes

CVssWriterEx2

New VSS Enumerations

VSS_RECOVERY_OPTIONS

New VSS Functions

CreateVssExpressWriter

Existing VSS Interface Modifications

IVssHardwareSnapshotProviderEx interface

Added method: **ResyncLuns**

VSS Event Tracing and Logging

Beginning with Windows Server 2008 R2 and Windows 7, you can use the VssTrace tool, the Logman tool, or the Tracelog tool to collect tracing information for the VSS infrastructure. For more information, see Using Tracing Tools with VSS.

LUN Resynchronization

In Windows Server 2008 R2 and Windows 7, VSS requesters can use a hardware shadow copy provider feature called LUN resynchronization (or "LUN resync"). This is a fast-recovery scheme that allows an application administrator to restore data from a shadow copy to the original logical unit number (LUN) or to a new LUN. The shadow copy can be a full clone or a differential shadow copy. In either case, at the end of the resync operation, the destination LUN will have the same contents as the shadow copy LUN. During the resync operation, the array performs a block-level copy from the shadow copy to the destination LUN. For more information, see the following:

- LUN Resync - A fast recovery scenario in Server 2008 R2
- "How Shadow Copies Are Used" in Volume Shadow Copy Service

- Common LUN Resync gotchas

Express Writers

In Windows Server 2008 R2 and Windows 7, the VSS express interface allows an application to register the location of its data files without implementing a VSS writer. For more information, see Volume Shadow Copy Service (VSS) Express Writers.

New VSS Writers

Windows Server 2008 R2 and Windows 7 introduce the following VSS writers:

- Performance Counters Writer
- Task Scheduler Writer
- VSS Metadata Store Writer

These new writers are documented in In-Box VSS Writers.

What's New in VSS in Windows Server 2008 and Windows Vista SP1

3/5/2021 • 2 minutes to read • Edit Online

Windows Server 2008 and Windows Vista with Service Pack 1 (SP1) introduce the following changes to the Volume Shadow Copy Service.

> **NOTE**
> All changes for Windows Vista apply to Windows Server 2008 and Windows Vista with SP1. For details on those changes, see What's New in VSS in Windows Vista.

New VSS Interfaces

IVssDifferentialSoftwareSnapshotMgmt3
IVssHardwareSnapshotProviderEx

New VSS Enumerations

_VSS_HARDWARE_OPTIONS
VSS_PROTECTION_FAULT
VSS_PROTECTION_LEVEL

New VSS Structures

VSS_VOLUME_PROTECTION_INFO

Existing VSS Enumeration Modifications

VSS_BACKUP_SCHEMA enumeration

Added value:

VSS_BS_WRITER_SUPPORTS_PARALLEL_RESTORES

_VSS_VOLUME_SNAPSHOT_ATTRIBUTES enumeration

Added values:

VSS_VOLSNAP_ATTR_DELAYED_POSTSNAPSHOT

VSS_VOLSNAP_ATTR_TXF_RECOVERY

Existing VSS Interface Modifications

IVssBackupComponentsEx2 interface

Added method:

BreakSnapshotSetEx

New VSS Writers

Windows Server 2008 and Windows Vista with SP1 introduce the following VSS writers:

- IIS Configuration Writer
- IIS Metabase Writer
- NPS VSS Writer
- Remote Desktop Services (Terminal Services) Gateway VSS Writer
- Remote Desktop Services (Terminal Services) Licensing VSS Writer

These writers are documented in In-Box VSS Writers.

What's New in VSS in Windows Vista

3/5/2021 • 2 minutes to read • Edit Online

Windows Vista introduces the following changes to the Volume Shadow Copy Service.

Note that all changes for Windows Vista also apply to Windows Server 2008 and Windows Vista with Service Pack 1 (SP1).

New VSS Interfaces

IVssBackupComponentsEx2

IVssComponentEx

IVssCreateWriterMetadataEx

IVssDifferentialSoftwareSnapshotMgmt2

IVssExamineWriterMetadataEx2

New VSS Classes

CVssWriterEx

New VSS Enumerations

VSS_ROLLFORWARD_TYPE

Existing VSS Enumeration Modifications

VSS_BACKUP_SCHEMA enumeration

Added values:

VSS_BS_AUTHORITATIVE_RESTORE

VSS_BS_INDEPENDENT_SYSTEM_STATE

VSS_BS_RESTORE_RENAME

VSS_BS_ROLLFORWARD_RESTORE

VSS_COMPONENT_FLAGS enumeration

Added values:

VSS_CF_NOT_SYSTEM_STATE

_VSS_VOLUME_SNAPSHOT_ATTRIBUTES enumeration

Added values:

VSS_VOLSNAP_ATTR_NO_AUTORECOVERY

VSS_VOLSNAP_ATTR_NOT_TRANSACTED

VSS Event Tracing and Logging

- The VSS trace file can now be located on any local volume. On versions of Windows prior to Windows Vista, the VSS trace file could not be located on a volume that was in the shadow copy set.
- Many event log entries have been reworded to make them clearer.
- All VSS event log entries now contain context information.

VSS Error Reporting

- Descriptions of all VSS error codes can now be retrieved by calling the **FormatMessage** function with the FORMAT_MESSAGE_FROM_HMODULE flag specified in the *dwFlags* parameter.
- The VSS error code messages are contained in vsstrace.dll. A handle to this module must be specified in the *lpSource* parameter.

Excluding Files from Shadow Copies

Applications or services can use the FilesNotToSnapshot registry key to specify files to be deleted from newly created shadow copies. For more information, see Excluding Files from Shadow Copies.

Backup and Restore Application Compatibility

Developers of backup and restore applications need to be aware of certain new features in Windows Vista and Windows Server 2008. For an application compatibility checklist, see Application Compatibility for Backup and Restore.

Application Compatibility for Backup and Restore

3/5/2021 • 2 minutes to read • Edit Online

Windows Vista and Windows Server 2008 contain new and changed features that a backup application developer must be aware of. These features are described in the following topics.

- Porting Backup Applications
- Backing Up and Restoring System State
- Working with File System and Security Features
- Junction Points

Porting Backup Applications

3/5/2021 • 2 minutes to read • Edit Online

Binaries from Windows Server 2003 with Service Pack 1 (SP1) are compatible with Windows Vista and Windows Server 2008. However, binaries from earlier versions of Windows must be recompiled.

Porting Windows XP Backup Applications to Windows Vista

Several VSS interfaces have changed since Windows XP. At a minimum, Windows XP applications must be recompiled using header files and libraries from the VSS 7.2 SDK or the Windows Vista or Windows Server 2008 SDK.

Porting Windows Server 2003 SP1 Backup Applications to Windows Vista

Binaries from Windows Server 2003 with SP1 are compatible with Windows Vista and Windows Server 2008. However, most Windows Server 2003 with SP1 backup applications must be updated to account for new features, which are described in the following sections.

Compiling Backup Applications for Windows Vista

Applications that use interfaces available in Windows Server 2003 can be compiled using header files and libraries provided in the VSS 7.2 SDK. To use new interfaces, applications must be recompiled using the header files and libraries included in the Windows Vista SDK.

> **NOTE**
>
> Binaries compiled using Windows Vista or Windows Server 2008 are not compatible with Windows Server 2003 with SP1 or earlier versions of Windows.

Backing Up and Restoring System State

3/5/2021 • 3 minutes to read • Edit Online

> **NOTE**
> This topic applies to Windows Vista, Windows Server 2008, and later. For information about Windows Server 2003, see Backing Up and Restoring System State in Windows Server 2003 R2 and Windows Server 2003 SP1

When performing a VSS backup or restore, the Windows system state is defined as being a collection of several key operating system elements and their files. These elements should always be treated as a unit by backup and restore operations.

> **NOTE**
> Microsoft does not provide developer or IT professional technical support for implementing online system state restores on Windows (all releases).

When backing up and recovering system state, the recommended strategy is to back up and recover the system and boot volumes in addition to the files enumerated by the system state writers. System state writers are writers that have the VSS_USAGE_TYPE attribute set to either VSS_UT_BOOTABLESYSTEMSTATE or VSS_UT_SYSTEMSERVICE.

> **IMPORTANT**
> If a VSS Writer is identified by its VSS_USAGE_TYPE as a system state writer it must be included in a system state backup even if it is selectable.

In addition to the enumerated operating system and driver binary files that are enumerated by the system state writers, there are certain other files that must be backed up as part of system state.

All the components reported by a VSS system state writer are part of system state except those for which the VSS_CF_NOT_SYSTEM_STATE flag is set.

Backup programs should also set the **LastRestoreId** registry key. For more information, see Registry Keys and Values for Backup and Restore.

> **NOTE**
> In Windows Vista, Windows Server 2008, and later, the names and locations of some system files have been changed as follows.

System State

For Windows Server 2012 and later, in addition to the files reported by the various VSS system-state writers, only the following licensing files need to be included explicitly, and the following DRM files need to be excluded explicitly.

Windows Media Digital Rights Management Files

In Windows Server 2008 and later, the following files, including all subdirectories under the following path, are excluded from system state and must not be backed up:

- %ProgramData%\Microsoft\Windows\DRM\

This supersedes the information in the Windows Media Digital Rights Management section of Working with File System and Security Features.

Performance Counter Configuration Files

The performance counter configuration files are located in the %SystemRoot%\System32\ directory and have the following names:

Perf?00?.dat Perfc0??.dat Perfd0??.dat Perfh0??.dat Perfi0??.dat Prfc0???.dat Prfd0???.dat Prfh0???.dat Prfi0???.dat

These files are only modified during application installation and should be backed up and restored during system state backups and restores.

IIS Configuration Files

> **NOTE**
> In Windows Vista with Service Pack 1 (SP1) and later, you should not back up these files. Instead, use the in-box IIS configuration writer. For more information about this writer, see In-Box VSS Writers.

The relevant IIS configuration files and their locations are listed below:

- The .NET FX machine.config file is located in the framework version directory.
- The ASP.NET root web.config file is located in the framework version directory.

  > **NOTE**
  > The configuration files for both .NET FX and ASP.NET are in the framework version directory. If multiple versions of the framework are installed on the computer, this directory will contain one configuration file for each installed version.

- The IIS applicationHost.config central configuration file is located in the %windir%\system32\inetsrv\config directory. For the server to understand this configuration file, there are schema files that determine its grammar and structure. These files are located in the %windir%\system32\inetsrv\config\schema directory.

The framework version directory path is stored in the following registry key:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework\InstallRoot

In addition, the following cryptography keys must be backed up:

%ProgramData%\Microsoft\Crypto\RSA\MachineKeys*
%SystemRoot%\System32\Microsoft\Protect*

Framework Files

All versions of the .NET framework must be backed up. The files are located in one or both of the following directories:

%windir%\\Microsoft.Net\\Framework %windir%\\Microsoft.Net\\Framework64

In addition, the assembly files must be backed up. These files are located in the following directory:

%windir%\assembly

Task Scheduler Task Files

The task scheduler's task files must be backed up. The files are located in one or both of the following locations:

%windir%\\system32\\tasks and any subdirectories (recursively) %windir%\\tasks (no subdirectories)

Working with File System and Security Features

3/5/2021 • 2 minutes to read • Edit Online

The following are hints for interoperating correctly with various file system and security features that were introduced in Windows Vista and Windows Server 2008.

Interoperability with Transactions

To support transactions, VSS ensures that both the Kernel Transaction Manager (KTM) and the Distributed Transaction Coordinator (DTC) are frozen prior to the creation of volume shadow copies. If the system fails to freeze or thaw KTM or DTC, the following timeout errors may be returned by the **IVssBackupComponents::DoSnapshotSet** method:

VSS_E_TRANSACTION_FREEZE_TIMEOUT VSS_E_TRANSACTION_THAW_TIMEOUT

If the requester receives one of these error codes, it must retry the shadow copy creation.

Registry and file system operations can also be transacted. In the case of the registry, the registry writer is responsible for ensuring transactional consistency. In the case of Transactional NTFS (TxF) file system operations, the system provider is responsible for ensuring transactional consistency. This is accomplished by mounting the shadow copy as read/write after it is created to allow for auto-recovery. During the auto-recovery phase, KTM rolls back any incomplete transactions.

Identifying and Creating Hard Links

When backing up files, a backup application must identify all hard links to each file to avoid backing up the same file more than once. Two new functions are available for enumerating hard links: **FindFirstFileNameW** and **FindNextFileNameW**.

Similarly, when restoring files, the backup application must restore hard links using the **CreateHardLink** function.

Backup applications must also assert the SE_BACKUP_NAME privilege during the backup phase and the SE_RESTORE_NAME during the restore phase.

Permissions and Privileges Required by Backup Applications

Backup applications that restore system files require the following privileges:

SE_BACKUP_NAME SE_RESTORE_NAME SE_SECURITY_NAME SE_TAKE_OWNERSHIP_NAME

Backup applications must also request WRITE_OWNER access rights during the restore phase.

Windows Media Digital Rights Management

Windows Media Digital Rights Management (DRM) client maintains a directory of sensitive state and license files in the %ProgramData%\Microsoft\Windows\DRM directory. The files in this directory should be purged at the same time as temporary and cache files. To ensure that the Windows DRM client remains in a consistent state, you must avoid backing up or restoring these files. This directory is listed in the FilesNotToBackup registry key. For more information about the FilesNotToBackup key, see Generating a Backup Set.

User Account Control

The introduction of User Account Control (UAC) in Windows Vista means that unless specified otherwise, applications must run under a standard user account. Additionally, the file and registry virtualization feature of UAC alters the locations where user data is stored. For more information about how to work with UAC and file and registry virtualization, see the following references:

The Windows Vista and Windows Server 2008 Developer Story: Windows Vista Application Development Requirements for User Account Control (UAC)
Windows Vista Application Development Requirements for User Account Control Compatibility
User Account Control (UAC) Patching

BitLocker Drive Encryption

BitLocker Drive Encryption is a new feature in Windows Vista Enterprise, Windows Vista Ultimate, and Windows Server 2008 that offers secure startup and full volume encryption. Understanding this feature is important for developers of backup applications that perform offline restores where data may need to be restored onto an encrypted drive.

To restore data onto an encrypted drive, perform the following steps:

1. Unlock the drive.
2. Turn off BitLocker Drive Encryption.
3. Perform the restore.
4. Boot into the restored operating system and turn on BitLocker Drive Encryption.

For detailed information about BitLocker Drive Encryption, including a step-by-step guide, see BitLocker Drive Encryption on the Microsoft TechNet Windows Vista website. For information about the BitLocker Drive Encryption WMI provider, see BitLocker Drive Encryption Provider.

Junction Points

3/5/2021 • 2 minutes to read • Edit Online

In Windows Vista and Windows Server 2008, the default locations for user data and system data have changed. For example, user data that was previously stored in the %SystemDrive%\Documents and Settings directory is now stored in the %SystemDrive%\Users directory. For backward compatibility, the old locations have junction points that point to the new locations. For example, C:\Documents and Settings is now a junction point that points to C:\Users. Backup applications must be capable of backing up and restoring junction points.

These junction points can be identified as follows:

- They have the FILE_ATTRIBUTE_REPARSE_POINT, FILE_ATTRIBUTE_HIDDEN, and FILE_ATTRIBUTE_SYSTEM file attributes set.
- They also have their access control lists (ACLs) set to deny read access to everyone.

Applications that call out a specific path can traverse these junction points if they have the required permissions. However, attempts to enumerate the contents of the junction points will result in failures. It is important that backup applications do not traverse these junction points, or attempt to backup data under them, for two reasons:

- Doing so can cause the backup application to back up the same data more than once.
- It can also lead to cycles (circular references).

Per-User Junctions and System Junctions

The junction points that are used to provide file and registry virtualization in Windows Vista and Windows Server 2008 can be divided into two classes: per-user junctions and system junctions.

Per-user junctions are created inside each individual user's profile to provide backward compatibility for user applications. The junction point at C:\Users*username*\My Documents that points to C:\Users*username*\Documents is an example of a per-user junction. Per-user junctions are created by the Profile service when the user's profile is created.

The other junctions are system junctions that do not reside under the Users*username* directory. Examples of system junctions include:

- Documents and Settings
- Junctions within the All Users, Public, and Default User profiles

System junctions are created by userenv.dll when it is invoked by Windows Welcome (also called the machine out-of-box-experience, or mOOBE).

> **NOTE**
> If the user changes the system language to a language other than English, the per-user and system junction points will be created with localized names.

What's New in VSS in Windows Server 2003 SP1

3/5/2021 • 2 minutes to read • Edit Online

The following list indicates additions and changes to the Volume Shadow Copy Service interface in Windows Server 2003 with Service Pack 1 (SP1):

Auto-recovery

Auto-recovery allows writers a time to update components in a shadow copy before the shadow copy is permanently changed to read-only. For example, a database may need to rollback any incomplete transactions for all shadow copies (the database writer would set the **VSS_CF_BACKUP_RECOVERY** flag for the database components). Auto-recovery initiated by the requester allows both fine-grained restore (for example restoring one table in a database or one folder on a mail server), or the rollback to support data mining to perform analysis that would be too slow for a production server (the requester would add **VSS_VOLSNAP_ATTR_ROLLBACK_RECOVERY** to the shadow copy context.) Auto-recovery is not compatible with transportable shadow copies, but the same functionality is supported by using Fast Recovery Using Transportable Shadow Copied Volumes.

The following programming elements have changes to support auto-recovery:

Class methods

- **CVssWriter::GetSnapshotDeviceName**
- **CVssWriter::OnPostSnapshot**

Enumerations

- **VSS_COMPONENT_FLAGS**
- **_VSS_VOLUME_SNAPSHOT_ATTRIBUTES**

Interface methods

- **IVssProviderCreateSnapshotSet::PreFinalCommitSnapshots**
- **IVssProviderCreateSnapshotSet::PostFinalCommitSnapshots**

Full Support for Transportable Shadow Copies

Transportable shadow copies are supported in all editions of Windows Server 2003 with SP1. For more information, see Importing Transportable Shadow Copied Volumes.

Fast Recovery Using Transportable Shadow Copied Volumes

Fast Recovery Using Transportable Shadow Copied Volumes

Shadow copy storage area management

IVssSnapshotMgmt2

What's New in VSS in Windows Server 2003

3/5/2021 • 2 minutes to read • Edit Online

The following list indicates additions and changes to the Volume Shadow Copy Service interface in Windows Server 2003:

Summary of VSS API Changes in Windows Server 2003

3/5/2021 • 2 minutes to read • Edit Online

Changes in the VSS Service

Events added:

BackupShutdown

Changes in VSS Functionality

Additional functionality:

partial file support

directed targeting

New VSS Interfaces

IVssWMDependency

Existing VSS Interface Modifications

IVssAsync interface

Methods modified:

IVssAsync::Wait

IVssBackupComponents interface

Methods added:

IVssBackupComponents::AddNewTarget

IVssBackupComponents::QueryRevertStatus

IVssBackupComponents::RevertToSnapshot

IVssBackupComponents::SetRangesFilePath

IVssBackupComponents::SetRestoreState

IVssCreateWriterMetadata interface

Methods added:

IVssCreateWriterMetadata::AddComponentDependency

IVssCreateWriterMetadata::SetBackupSchema

Methods modified:

IVssCreateWriterMetadata::AddComponent

IVssCreateWriterMetadata::AddDatabaseFiles

IVssCreateWriterMetadata::AddDatabaseLogFiles

IVssCreateWriterMetadata::AddFilesToFileGroup

IVssExamineWriterMetadata interface

Methods added:

IVssExamineWriterMetadata::GetBackupSchema

IVssComponent interface

Methods removed:

IVssComponent::AddNewTarget

Methods added:

IVssComponent::AddDifferencedFilesByLastModifyTime

IVssComponent::GetDifferencedFile

IVssComponent::GetDifferencedFilesCount

Methods no longer reserved:

IVssComponent::AddDirectedTarget

IVssComponent::GetDirectedTarget

IVssWMComponent interface

Methods added:

IVssWMComponent::GetDependency

IVssWMFiledesc interface

Methods added:

IVssWMFiledesc::GetBackupTypeMask

Existing VSS Class Modifications

CVssWriter class

Methods modified:

CVssWriter::Initialize

Methods added:

CVssWriter::GetContext

CVssWriter::GetRestoreType

CVssWriter::GetSnapshotDeviceName

CVssWriter::OnBackupShutdown

New VSS Enumerations

VSS_BACKUP_SCHEMA

VSS_COMPONENT_FLAGS

VSS_FILE_SPEC_BACKUP_TYPE

VSS_RESTORE_TYPE

Existing VSS Enumeration Modifications

VSS_BACKUP_TYPE enumeration

Added values:

VSS_BT_COPY

VSS_RESTORE_TARGET enumeration

Removed values:

VSS_RT_NEW

VSS_RESTOREMETHOD_ENUM enumeration

Added values:

VSS_RME_RESTORE_AT_REBOOT_IF_CANNOT_REPLACE

VSS_SNAPSHOT_STATE enumeration

Added values:

VSS_SS_PROCESSING_POSTCOMMIT

VSS_SS_PROCESSING_PREFINALCOMMIT

VSS_SS_PREFINALCOMMITTED

VSS_SS_PROCESSING_POSTFINALCOMMIT

_VSS_VOLUME_SNAPSHOT_ATTRIBUTES enumeration

Added values:

VSS_VOLSNAP_ATTR_AUTORECOVER

Reserved values now support:

VSS_VOLSNAP_ATTR_HARDWARE_ASSISTED

VSS_VOLSNAP_ATTR_IMPORTED

VSS_VOLSNAP_ATTR_EXPOSED_LOCALLY

VSS_VOLSNAP_ATTR_EXPOSED_REMOTELY

VSS_WRITER_STATE enumeration

Added values:

VSS_WS_FAILED_AT_BACKUPSHUTDOWN

Changes to VSS Structures

VSS_COMPONENTINFO structure

Added members:

bSelectableForRestore

dwComponentFlags

cDependencies

New Functionality Available in Windows Server 2003

3/5/2021 • 2 minutes to read • Edit Online

The Windows Server 2003 release of the Volume Shadow Copy Service contains the following new functionality:

- Support for *selectability for restore* in addition to and on an equal basis with *selectability for backup* (previously just referred to as selectability). See Working with Selectability and Logical Paths for more information.
- Introduction of a *BackupShutdown* event indicating that a VSS-compliant backup application (requester) has shut down, whether the backup attempt has properly completed or not. See Handling BackupShutdown Events for more information.
- Support for explicit writer-component dependencies. A means for describing and supporting the situations where a component (and the component set it defines) managed by one writer cannot be backed up or restore independently of components managed by other writers. See Dependencies between Components Managed by Different Writers for more information.
- Full support for *partial files* operations. The backup and restore of segments of files by writers and requesters is now fully supported. See Working with Partial Files for more information on partial file operations.
- Introduction of *differenced files* to support incremental backup and restore operations. See Incremental and Differential Backups for more information.
- Introduction of backup schemas as a concept indicating the level of writer support for types of backup operations. See **VSS_BACKUP_SCHEMA** for more information.
- Support for requester specification of new file restore destinations (*new target locations*) during restore operations. See Working with New Targets during Restore for more information.
- Support for conditional restore at reboot time. See VSS_RME_RESTORE_AT_REBOOT_IF_CANNOT_REPLACE (**VSE_RESTOREMETHOD_ENUM**) for more information.
- Definition of a restore state and restore type. See **VSS_RESTORE_TYPE** and VSS Restore State for more information.
- Support for writer queries about shadow copy state. See **CVssWriter::GetContext** and **CVssWriter::GetSnapshotDeviceName** for more information.
- Support for transportable shadow copies in Windows Server 2003, Enterprise Edition and Windows Server 2003, Datacenter Edition. For more information, see Importing Transportable Shadow Copied Volumes.

Functionality Removed From Windows Server 2003

3/5/2021 • 2 minutes to read • Edit Online

The Windows Server 2003 release of VSS no longer supports the following:

- Writers cannot specify new file restore destinations (*new target locations*) at restore operations.
- Remounting an exposed shadow copy as a writable volume is no longer supported. The **IVssBackupComponents::RemountReadWrite** method is no longer available.
- Writers cannot specify explicitly included files (using **IVssCreateWriterMetadata::AddIncludeFiles**). Files can be added to a component only as part of a file set by using **IVssCreateWriterMetadata::AddFilesToFileGroup**, **IVssCreateWriterMetadata::AddDatabaseFiles**, or **IVssCreateWriterMetadata::AddDatabaseLogFiles**.

  Files can still be explicitly excluded from a component by using **IVssCreateWriterMetadata::AddExcludeFiles**.

Semantic Changes to Windows Server 2003

3/5/2021 • 2 minutes to read • Edit Online

The combination of path, file specification, and recursion flag (*wszPath*, *wszFileSpec*, and *bRecursive*, respectively) must match that of one of the file sets added to one of the writer's components using **IVssCreateWriterMetadata::AddFilesToFileGroup**, **IVssCreateWriterMetadata::AddDatabaseFiles**, or **IVssCreateWriterMetadata::AddDatabaseLogFiles** when:

- A requester adds a new target using **IVssBackupComponents::AddNewTarget**.
- A writer adds an alternate location mapping using **IVssCreateWriterMetadata::AddAlternateLocationMapping**.
- Existing files are added as differenced files using **IVssComponent::AddDifferencedFilesByLastModifyTime**.
- A requester indicates that an alternate location mapping was used to restore a file set using **IVssBackupComponents::AddAlternativeLocationMapping**.

Restrictions in Windows Server 2003

3/5/2021 • 2 minutes to read • Edit Online

Certain functionality planned for the Windows Server 2003 release of VSS is not fully supported:

- During backup operations, multiple instances of a given writer class are allowed only if only one of those instances has a writer restore state (see **VSS_WRITERRESTORE_ENUM**) other than VSS_WRE_NEVER. If this condition is met, all writer instances will participate fully during the backup, generating Writer Metadata Documents and participating in event handling.
- During restore operations, only one writer will receive restore events generated by the requester. If more than one instance of a writer class has a writer restore state set to something other than VSS_WRE_NEVER, only one instance will receive restore events. Which instance will receive them is indeterminate.

About the Volume Shadow Copy Service

3/5/2021 • 2 minutes to read • Edit Online

The Volume Shadow Copy Service (VSS) captures and copies stable images for backup on running systems, particularly servers, without unduly degrading the performance and stability of the services they provide.

The VSS solution is designed to enable developers to create services (*writers*) that can be effectively backed up by any vendor's backup application using VSS (*requesters*).

Common Volume Backup Issues

3/5/2021 • 2 minutes to read • Edit Online

Any backup operation that attempts to copy a full and stable image of a system must deal with the following concerns:

- Inaccessible files during a backup. Running applications frequently need to keep files open in exclusive mode during a backup, preventing backup programs from copying them.
- Inconsistent file state. Even if an application does not have its files open in exclusive mode, it is possible—because of the finite time needed to open, back up, and close a file—that files copied to storage media may not all reflect the same application state.
- Need to minimize service interruptions. To ensure file accessibility and the integrity of the data being backed up can require the suspension and/or termination of all running programs during a volume backup. For large disk systems, this could be hours in duration.

  Recently, some storage vendors have attempted to address these problems by providing a volume capture mechanism—a means to capture an image of the files on disk at a given instant in time—using either a copy-on-write or "split mirror" mechanism. However, these solutions entail difficulties of their own:

  - Incompatible vendor implementations of volume capture. Many providers of RAID devices provide volume capture mechanisms. However, each vendor has its own interface and each must get support from the backup vendors for their volume capture interfaces. This means that backup application vendors must support multiple volume capture implementations, which is undesirable.
  - Lack of application coordination. Many devices that support a volume capture do not support the coordination of running applications with the freeze of data on disk. For those devices that do, as with the backup applications, each vendor has a different interface.
  - Limited support for non-RAID devices. Few if any conventional disk vendors provide support for any sort of volume capture in their device drivers. This means that capture mechanisms are limited to certain disk systems, and cannot typically support the backup of system areas.
  - Need to handle updates to disk during volume capture. Although storage-vendor-provided volume capture mechanisms can freeze the state of data on disk, they do not always interoperate with running applications. This frequently means that data sent to the volume while a storage device is undergoing volume capture may be lost.
  - Consistent multivolume backup. The storage device executes these volume captures, so there is generally no mechanism for coordinating the timing of the data freeze. This is particularly true if the devices come from separate vendors. Therefore, if several storage volumes are involved in a backup with a volume capture, the time image preserved for each volume may not be consistent.

The VSS Model

3/5/2021 • 2 minutes to read • Edit Online

VSS is designed to address the problems described in Common Volume Backup Issues.

The VSS model includes the following:

- The shadow copy mechanism. VSS provides fast volume capture of the state of a disk at one instant in time—a *shadow copy* of the volume. This volume copy exists side by side with the live volume, and contains copies of all files on disk effectively saved and available as a separate device.
- Consistent file state via application coordination. VSS provides a COM-based, event-driven interprocess communication mechanism that participating processes can use to determine system state with respect to backup, restore, and shadow copy (volume capture) operations. These events define stages by which applications modifying data on disk (*writers*) can bring all their files into a consistent state prior to the creation of the shadow copy.
- Minimizing application downtime. The VSS shadow copy exists in parallel with a live copy of the volume to be backed up, so except for the brief period of the shadow copy's preparation and creation, an application can continue its work. The time needed to actually create a shadow copy, which occurs between *Freeze* and *Thaw* events, typically takes about one minute.

  While a writer's preparation for a shadow copy, including flushing I/O and saving state (see Overview of Pre-Backup Tasks), may be nontrivial, it is significantly shorter than the time required to actually back up a volume—which for large volumes may take hours.
- Unified interface to VSS. VSS abstracts the shadow copy mechanisms within a common interface while enabling a hardware vendor to add and manage the unique features of its own providers. Any backup application (*requester*) and any writer should be able to run on any disk storage system that supports the VSS interface.
- Multivolume backup. VSS supports *shadow copy sets*, which are collections of shadow copies, across multiple types of disk volumes from multiple vendors. All shadow copies in a shadow copy set will be created with the same time stamp and will present the same disk state for a multivolume disk state.
- Native shadow copy support. Beginning with Windows XP, shadow copy support is available through VSS as a native part of the Windows operating system. As long as at least one NTFS disk is present on a system, these systems can be configured to support shadow copies of all disk systems mounted on them.

Basic VSS Concepts

3/5/2021 • 2 minutes to read • Edit Online

To understand how to develop projects using VSS requires a basic understanding of:

- The Volume Shadow Copy Service
- Shadow Copies and Shadow Copy Sets
- Providers
- Requesters
- Writers

The Volume Shadow Copy Service

3/5/2021 • 2 minutes to read • Edit Online

The Volume Shadow Copy Service (VSS) provides the system infrastructure for running VSS applications on Windows-based systems.

Though largely transparent to user and developer, VSS does the following:

- Coordinates activities of providers, writers, and requesters in the creation and use of shadow copies
- Furnishes the default system provider
- Implements low-level driver functionality necessary for any provider to work

The VSS service starts on demand; therefore, for VSS operations to be successful, this service must be enabled.

Shadow Copies and Shadow Copy Sets

3/5/2021 • 3 minutes to read • Edit Online

A *shadow copy* is a snapshot of a volume that duplicates all of the data that is held on that volume at one well-defined instant in time. VSS identifies each shadow copy by a persistent GUID.

A *shadow copy set* is a collection of shadow copies of various volumes all taken at the same time. VSS identifies each shadow copy set by a persistent GUID.

How a particular hardware or software vendor chooses to implement shadow copies is completely at its discretion. Once a shadow copy is created, there are effectively two images of the shadow-copied volume available to the system: the original volume, which can be accessed conventionally; and the copied data, which can be accessed through the VSS API.

This allows two sets of activities to take place at the same time:

- Ordinary applications on the system can quickly continue or resume using the original volume, updating data on the disk.
- Applications that are using the VSS requester API to access the shadow-copied volume can perform backups or similar operations.

Shadow copies need not be implemented in the same way for every file, directory, or volume. Different implementations of the shadow copy mechanism (*providers*) may use different approaches to creating a shadow copy. However, to all applications that are using the VSS API, all shadow copies should appear the same.

For information on the default Windows provider implementation, see System Provider.

Default Shadow Copy State

Even though the file system flushes all I/O buffers prior to creating a shadow copy, this will not ensure that incomplete I/O is properly handled.

Therefore, assuming that the system has no VSS-enabled applications, the data in a shadow copy is said to be in a *crash-consistent state*. A shadow copy in a crash-consistent state contains an image of the disk that is the same as that which would exist following a catastrophic system shutdown. All files that were open will still exist on the volume, but they are not guaranteed to be free of incomplete I/O operations or data corruption.

While the crash-consistent state does not fully deal with all the issues associated with defining a stable backup set (see Common Volume Backup Issues), it has several advantages over the backup set that conventional backup operations would have to use:

- A volume contained in a shadow copy, even in a crash-consistent state, still contains all files. A backup set created without a shadow copy would not contain all files open at the time of the backup. Files held open at the time of the backup operation are excluded from the backup.
- The shadow copy of the volume is created at one instant in time, and not by traversing an active file system, which typically requires much more time.

Applications on a system that are not VSS-aware—word processors, editors, and so on—will likely have their files left in a crash-consistent state. However, VSS-aware applications (writers) can coordinate their actions so that the state of their files in the shadow copy is well defined and consistent.

Shadow Copy Freeze and Thaw

The creation of every VSS shadow copy operation is bracketed by *Freeze* and *Thaw* events, which writers use to put their files in a stable state prior to shadow copy.

Having Freeze and Thaw events as part of the VSS model means:

- Handling the Freeze event means that those who are developing writers must have a clearly delineated point in the backup cycle where they ensure that all write operations to the disk are stopped and that files are in a well-defined state for backup.
- Handling the Thaw event provides the mechanism for writers to resume writes to the disk and clean up any temporary files or other temporary state information that were created in association with the shadow copy.
- The default window between the Freeze and Thaw events is short (typically 60 seconds); therefore, actual interruption of any service that a writer provides can be minimized.
- Handling of other events (such as PrepareForSnapshot) preceding and following the Freeze and Thaw events, respectively, provides the necessary flexibility to allow writers to complete complicated operations to support shadow copies.

Providers

3/5/2021 • 4 minutes to read • Edit Online

Providers manage running volumes and create the shadow copies of them on demand.

In response to a request from a requester, a provider generates COM events to signal applications of a coming shadow copy, then creates and maintains that copy until it is no longer needed.

While a shadow copy is in existence, the provider creates an environment where there are effectively two independent copies of any volume that has been shadow copied: one the running disk being used and updated as normal, the other a copy that is disk fixed and stable for backup.

While a default provider is supplied as part of Windows, other vendors are free to supply their own implementations that are optimized for their own storage hardware and software offerings.

From the point of view of an end-user or backup/restore application developer, all providers will have the same interface (see Selecting Providers).

All providers must be able to do the following:

- Intercept I/O requests between the file system and the underlying mass storage system.
- Capture and retrieve the status of a volume at the time of shadow copy, maintaining a "point in time" view of the files on the disk with no partial I/O operations reflected in its state.
- Use this "point in time" view to expose (minimally to VSS-enabled applications) a virtual volume containing the shadow copied data.

Depending on how this is done, a provider can be one of three types:

- System Provider
- Software Providers
- Hardware Providers

System Provider

One shadow copy provider, the *system provider*, is supplied as a default part of a Windows operating system installation. Currently, the system provider is a particular instance of a software provider. However, this may change in the future.

To maintain a "point in time" view of a volume contained in the shadow copy, the system provider uses a copy-on-write technique. Copies of the sectors on disk that have been modified (called "diffs") since the beginning of the shadow copy creation are stored in a shadow copy storage area.

Therefore, the system provider can expose the live volume, which can be written to and read from normally, and apply the "diffs" to the live volume's data to effectively expose the frozen data of the shadow copy.

For the system provider, the shadow copy storage area must be on an NTFS volume. The volume to be shadow copied does not need to be an NTFS volume, but at least one volume mounted on the system must be an NTFS volume.

Software Providers

Software shadow copy providers intercept and process I/O requests in a software layer between the file system and the volume manager software. These providers are implemented as a user-mode DLL component and at

least one kernel-mode device driver, typically (but not necessarily) a storage filter driver. The work of creating these shadow copies is done in software.

A software shadow copy provider must maintain a "point-in-time" view of a volume by having access to a set of files that can be used to accurately re-create volume status prior to the shadow copy. An example of this is the copy-on-write technique of the system provider.

However, VSS places no restrictions on what technique that software providers use to create and maintain shadow copies, and third-party vendors are free to implement their software providers as they see fit.

In addition, VSS provides support for much of the functionality of software shadow copy providers, such as defining the point-in-time, data synchronization and flushing, providing a common interface for backup applications, and management of the shadow copy.

A software provider will, by definition, be applicable to a wider range of storage platforms than a hardware provider, and should be able to work with basic disks or logical volumes equally well. This generality sacrifices the performance that may be available by implementing shadow copies in hardware and does not make use of any vendor-specific volume capture or file mirroring features.

Hardware Providers

Hardware shadow copy providers intercept I/O requests from the file system at the hardware level by working in conjunction with a hardware storage adapter or controller. The work of creating the shadow copy is performed by a host adapter, storage appliance, or RAID controller outside of the operating system.

These providers are implemented as a user-mode DLL component communicating with the hardware that will expose the shadow copy data: therefore, hardware shadow copy providers may need to call or create other kernel-mode components.

Hardware providers expose to VSS shadow copies of entire disks or logical units (LUNs). Requesters still deal with shadow copies of volumes; all the volume-to-disk mapping is handled internally by VSS. Shadow copies created by hardware providers of volumes that reside on dynamic disks have a specific requirement: They cannot be imported onto the same system. They must be created transportable and imported on a second system.

While a hardware shadow copy provider makes use of VSS functionality that defines the point in time, allows data synchronization, manages the shadow copy, and provides a common interface with backup applications, VSS does not specify the underlying mechanism by which the hardware provider produces and maintains shadow copies.

Requesters

3/5/2021 • 3 minutes to read • Edit Online

A *requester* is any application that uses the VSS API (specifically the **IVssBackupComponents** interface) to request the services of the Volume Shadow Copy Service to create and manage shadow copies and shadow copy sets of one or more volumes.

The most typical example of a requester (and the only one addressed in this documentation) is a VSS-aware backup/restore application, which uses shadow-copied data as a stable source for its backup operations.

In addition to initiating shadow copies, backup/recover requester applications communicate with data producers (writers) to gather information on the system and to signal writers to prepare their data for backup.

Requester State

A requester maintains its state information in an XML-based metadata object called the Backup Components Document. The requester metadata is necessary, but not sufficient to allow a requester to back up and then restore a file system. The reasons for this are the following:

- During a backup operation, only a subset of all the components involved in the backup—*nonselectable for backup* components without selectable for backup ancestors and selectable for backup components that have been *included explicitly* in the backup—have had their information added to the requester's Backup Components Document.
- The information even for those components added to the Backup Components Document is incomplete—file and path specifications are not included.
- During restore operations, a component *implicitly included* in the backup may be *selectable for restore* and therefore can be explicitly included in the restore. This will require the updating of the requester's Backup Components Document with information from stored copies of a writer's Writer Metadata Document.

To allow for full specification of a backup or restore operation, the VSS API allows the requester to query running writers' metadata (during backups) or examine stored writer metadata (during restores). In addition, a writer can modify component information in the Backup Components Document in the course of a backup or restore operation.

Using the information about which components have been selected for backup and restore and the rules regarding component selection (for more information, see Setting up Component Organization and Working with Selectability and Logical Paths), a requester can determine which files of which writer it needs to back up or restore, and where it can find those files.

As part of a backup, both requester and writer metadata must be stored so that it can be used in the restore. Conversely, restore operations require the retrieval of the old Backup Components and Writer Metadata Documents to obtain full instructions on restoring files.

Requester Interprocess Communication

The requester maintains control over VSS backup and restore operations by generating COM events through various calls in the requester API. These calls can do the following:

- Make requests of the providers, for example, **IVssBackupComponents::DoSnapshotSet** causes the provider to create a shadow copy of the selected volume.
- Trigger the writers to return information, for example, **IVssBackupComponents::GatherWriterMetadata**

enables the requester to obtain each writer's Writer Metadata Document.

- Require writers to prepare for or handle various phases of the shadow copy and backup operations, for example, **IVssBackupComponents::PrepareForBackup** signals writers to set up for the I/O freeze.

A requester receives information from the writers through live or stored Writer Metadata Documents and through the use of the **IVssComponent** interface, which the writer can update.

Life Cycle of a Requester during Backup

The following is a summary of the requester life cycle for backup:

1. Instantiate and initialize VSS API interfaces.
2. Contact writers and retrieve their information.
3. Choose data to back up.
4. Request a shadow copy of the provider.
5. Back up the data.
6. Release the interface and the shadow copy.

Life Cycle of a Requester during Restore

The restore life cycle does not require a shadow copy, but still requires writer cooperation:

1. Instantiate VSS API interfaces.
2. Initialize the requester for the restore operation by loading a stored Backup Components Document.
3. Retrieve stored Writer Metadata and Backup Components Documents.
4. Contact the writers to initialize cooperation.
5. Check for writer updates to the Backup Components Document.
6. Restore the data.

Writers

3/5/2021 • 3 minutes to read • Edit Online

Writers are applications or services that store persistent information in files on disk and that provide the names and locations of these files to requesters by using the shadow copy interface.

During backup operations, writers ensure that their data is quiescent and stable—suitable for shadow copy and backup. Writers collaborate with restores by unlocking files when possible and indicating alternate locations when necessary.

If no writers are present during a VSS backup operation, a shadow copy can still be created. In this case, all data on the shadow-copied volume will be in the *crash-consistent state*.

Writer State

Writers maintain their state in an XML-based metadata object, the *Writer Metadata Document*.

This writer metadata is the only data structure that contains the *file set*—path, file specification, and recursion flag—of the data to be backed up and restored.

The Writer Metadata Document organizes the writer's file sets into groups or *components*. The relationship of one of these components during backup and restore operations to the other components managed by the writer is described in the Writer Metadata Document by the component's *selectability for backup*, its *selectability for restore*, and its *logical paths*. (For more information, see Setting up Component Organization and Working with Selectability and Logical Paths.)

Additional information that governs file restoration and other issues is also contained in this document.

The requester needs the writer metadata, in conjunction with its own Backup Components Document, to process a backup or a restore.

Unlike the Backup Components Document, the Writer Metadata Document should be thought of as a read-only structure. Once a writer creates it, the document is not altered.

Writer Event Handling

A writer's VSS operations are initiated through the receipt of COM events.

When no events are present, a writer does not perform VSS operations (such as a VSS backup or restore). Instead, it performs its normal work, such as responding to database queries, managing user data, or providing other services.

To ensure that error handling for multiple parallel backup and restore sessions is performed correctly, and to ensure that one backup or restore session does not corrupt another, you must do the following:

- If a writer's event handler (such as **CVssWriter::OnFreeze**) calls the **CVssWriterEx2::GetSessionId**, **CVssWriter::SetWriterFailure**, or **CVssWriterEx2::SetWriterFailureEx** method, the event handler must call the method in the same thread that called the event handler.
- Your writer's implementation of an event handler such as **OnFreeze** can offload work to worker threads if desired, as long as each worker thread marshals any needed error reporting back to the original event handler thread.

Handling Identify Events

With the exception of the Identify event, the type and order of the events a writer receives depends uniquely on the type of VSS operations that are currently ongoing.

The Identify event requires writers to provide the system information about their configuration and the files they manage through their *Writer Metadata Document*. An Identify event is generated in support of almost any VSS operation, including system queries as well as shadow copy and backup and restore operations. Therefore, any writer's implementation of the Identify event handler **CVssWriter::OnIdentify** must be able to handle an Identify event at any time—including in the middle of processing another VSS operation, such as a backup or restore. An Identify event should never be thought of as part of the life cycle of a VSS operation, even though its generation may be expected and required prior to the start of that operation.

It is particularly important that state information about a VSS operation not be modified in **CVssWriter::OnIdentify**, because receipt of an out-of-order event would reset that information.

Backup and Restore Events

Depending on whether it is participating in a backup or restore, a writer will receive between two and seven events, in addition to an initial Identify event.

Handling these events constitutes (from the point of view of a writer) the life cycle of a backup or restore operation.

In a typical backup operation (see Overview of Processing a Backup Under VSS), a writer would handle the following events (in addition to an initial Identify event):

- PrepareForBackup
- PrepareForSnapshot
- Freeze
- Thaw
- PostSnapshot
- BackupComplete
- BackupShutdown

In a typical restore operation (see Overview of Processing a Restore Under VSS), a writer would handle the following events:

- PreRestore
- PostRestore

In-Box VSS Writers

3/5/2021 • 15 minutes to read • Edit Online

The Windows operating system includes a set of VSS writers that are responsible for enumerating the data that is required by various Windows features. These are referred to as "in-box" writers.

> **NOTE**
> The MSDE in-box writer is not available in Windows Vista, Windows Server 2008, and later. Instead, the SQL Writer should be used to back up SQL Server databases. Only SQL Server 2005 SP2 and later are supported on Windows Vista, Windows Server 2008, and later.

- Active Directory Domain Services (NTDS) VSS Writer
- Active Directory Federation Services Writer
- Active Directory Lightweight Directory Services (LDS) VSS Writer
- Active Directory Rights Management Services (AD RMS) Writer
- Automated System Recovery (ASR) Writer
- Background Intelligent Transfer Service (BITS) Writer
- Certificate Authority Writer
- Cluster Service Writer
- Cluster Shared Volume (CSV) VSS Writer
- COM+ Class Registration Database Writer
- Data Deduplication Writer
- Distributed File System Replication (DFSR)
- Dynamic Host Configuration Protocol (DHCP) Writer
- File Replication Service (FRS)
- File Server Resource Manager (FSRM) Writer
- Hyper-V Writer
- IIS Configuration Writer
- IIS Metabase Writer
- Microsoft Message Queuing (MSMQ) Writer
- MSSearch Service Writer
- NPS VSS Writer
- Performance Counters Writer
- Registry Writer
- Remote Desktop Services (Terminal Services) Gateway VSS Writer
- Remote Desktop Services (Terminal Services) Licensing VSS Writer
- Shadow Copy Optimization Writer
- Sync Share Service Writer
- System Writer
- Task Scheduler Writer
- VSS Metadata Store Writer
- Windows Deployment Services (WDS) Writer

- Windows Internal Database (WID) Writer
- Windows Internet Name Service (WINS) Writer
- WMI Writer

Active Directory Domain Services (NTDS) VSS Writer

This writer reports the NTDS database file (ntds.dit) and the associated log files. These files are required to restore the Active Directory correctly.

There is only one ntds.dit file per domain controller, and it is reported in the writer metadata as in the following example:

```
<DATABASE_FILES path="C:\Windows\NTDS"
                filespec="ntds.dit"
                filespecBackupType="3855"/>
```

Here is an example that shows how to list components in the writer's metadata:

```
<BACKUP_LOCATIONS>
    <DATABASE logicalPath="C:_Windows_NTDS"
                componentName="ntds"
                caption="" restoreMetadata="no"
                notifyOnBackupComplete="no"
                selectable="no"
                selectableForRestore="no"
                componentFlags="3">
    <DATABASE_FILES path="C:\Windows\NTDS"
                filespec="ntds.dit"
                filespecBackupType="3855"/>
    <DATABASE_LOGFILES path="C:\Windows\NTDS"
                filespec="edb*.log"
                filespecBackupType="3855"/>
    <DATABASE_LOGFILES path="C:\Windows\NTDS"
                filespec="edb.chk"
                filespecBackupType="3855"/>
    </DATABASE>
</BACKUP_LOCATIONS>
```

At backup time, the writer sets the backup expiration time in the writer's backup metadata. Requesters should retrieve this metadata by using **IVssComponent::GetBackupMetadata** to determine whether the database has expired. Expired databases cannot be restored.

If the computer that contains the NTDS database is a domain controller, the backup application should always perform a system state backup across all volumes containing critical system state information. At restore time, the application should first restart the computer in Directory Services Restore Mode and then perform a system state restore.

The writer name string for this writer is "NTDS".

The writer ID for this writer is B2014C9E-8711-4C5C-A5A9-3CF384484757.

Active Directory Federation Services Writer

This writer reports the Active Directory Federation Services (ADFS) data files.

Windows Server 2008 R2, Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: This writer is not supported.

The writer name string for this writer is "ADFS VSS Writer".

The writer ID for this writer is 772C45F8-AE01-4F94-940C-94961864ACAD.

Active Directory Lightweight Directory Services (LDS) VSS Writer

This writer reports the ADAM database file (adamntds.dit) and the associated log files for each instance in %program files%\Microsoft ADAM\instance*N*\data, where *N* is the ADAM instance number. These database log files are required to restore ADAM instances.

Windows XP: This writer is not supported.

Here is an example that shows how to list components in the writer's metadata:

```
<BACKUP_LOCATIONS>
    <DATABASE logicalPath="C:_Program Files_Microsoft ADAM_instance1_data"
                componentName="adamntds"
                caption=""
                restoreMetadata="no"
                notifyOnBackupComplete="no"
                selectable="no"
                selectableForRestore="no"
                componentFlags="3">
    <DATABASE_FILES path="C:\Program Files\Microsoft ADAM\instance1\data"
                filespec="adamntds.dit"
                filespecBackupType="3855"/>
    <DATABASE_LOGFILES path="C:\Program Files\Microsoft ADAM\instance1\data"
                filespec="edb*.log"
                filespecBackupType="3855"/>
    <DATABASE_LOGFILES path="C:\Program Files\Microsoft ADAM\instance1\data"
                filespec="edb.chk"
                filespecBackupType="3855"/>
    </DATABASE>
</BACKUP_LOCATIONS>
```

At backup time, the writer sets the backup expiration time in the backup metadata. Backup applications should retrieve this metadata by using the **IVssComponent::GetBackupMetadata** method to determine whether the database has expired. Expired databases cannot be restored.

The writer name string for this writer is "ADAM (instance*N*) Writer", where *N* is the ADAM instance number, for example, "ADAM (instance1) Writer", "ADAM (instance2) Writer", and so on.

The writer ID for this writer is DD846AAA-A1B6-42A8-AAF8-03DCB6114BFD. This writer ID is the same for all instances.

Active Directory Rights Management Services (AD RMS) Writer

This writer reports the Active Directory Rights Management Service (AD RMS) data files.

Windows Server 2008 R2, Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: This writer is not supported.

The writer name string for this writer is "AD RMS Writer".

The writer ID for this writer is 886C43B1-D455-4428-A37F-4D6B9E43F50F.

Automated System Recovery (ASR) Writer

The ASR writer stores the configuration of disks on the system. This writer reports the Boot Configuration Database (BCD) and is also responsible for dismounting the registry hive that represents the BCD during shadow copy creation. The ASR writer must be included in any backups required for bare-metal recovery. For more information about ASR, see Using VSS Automated System Recovery for Disaster Recovery.

Windows Server 2003 and Windows XP: This writer is not supported.

The writer name string for this writer is "ASR Writer".

The writer ID for the ASR writer is BE000CBE-11FE-4426-9C58-531AA6355FC4.

Background Intelligent Transfer Service (BITS) Writer

The BITS writer uses the **FilesNotToBackup** registry key to exclude files from the BITS cache folder. The default cache location is %AllUsersProfile%\Microsoft\Network\Downloader\Cache.

Windows Server 2003 and Windows XP: This writer is not supported.

In addition, the BITS writer excludes the following files from backup: the BITS state files (qmgr0.dat and qmgr1.dat), the BITS debug log file, and BITS partially downloaded files.

If the BITS download destination file is an SMB file, the client account must have a trust relationship to the server, or else backups may fail.

The writer name string for this writer is "BITS Writer".

The writer ID for the BITS writer is 4969D978-BE47-48B0-B100-F328F07AC1E0.

Certificate Authority Writer

This writer is responsible for enumerating the data files for the Certificate Server.

The writer name string for this writer is "Certificate Authority".

Windows XP: The writer name string for this writer is "Certificate Server Writer".

The writer ID for the writer is 6F5B15B5-DA24-4D88-B737-63063E3A1F86.

Cluster Service Writer

The Cluster Service VSS writer is documented in the Cluster Service API documentation.

Windows Vista, Windows Server 2003 and Windows XP: This writer is not supported until Windows Vista with Service Pack 1 (SP1) and Windows Server 2008.

Cluster Shared Volume (CSV) VSS Writer

This writer reports the Cluster Shared Volume (CSV) data files. This writer is an in-box writer for Windows Server operating system versions; it does not ship in Windows Client.

Windows Server 2008 R2, Windows Server 2008 and Windows Server 2003: This writer is not supported.

The writer name string for this writer is "Cluster Shared Volume VSS Writer".

The writer ID for the writer is 1072AE1C-E5A7-4EA1-9E4A-6F7964656570.

COM+ Class Registration Database Writer

This writer is responsible for the contents of the %SystemRoot%\Registration directory. The COM+ catalog is a file in %SystemRoot%\Registration with a name that follows the pattern R*xxxxxxxxxxxx*.clb, where *xxxxxxxxxxxx* is a 12-digit hexadecimal number. There can potentially be multiple such files if COM+ applications have been configured on the computer. The file that contains the current COM+ catalog is the file with the largest value of *xxxxxxxxxxxx*.

Windows Server 2003 and Windows XP: This writer is not supported.

The COM+ class registration database depends on a registry key being backed up and hence needs to be backed up and restored together with the registry.

To restore the COM+ Registration Database, a backup application (requester) must call the **ICOMAdminCatalog::RestoreREGDB** method.

The writer name string for this writer is "COM+ REGDB Writer".

The writer ID for the COM+ class registration database writer is 542DA469-D3E1-473C-9F4F-7847F01FC64F.

Data Deduplication Writer

The Data Deduplication VSS writer is documented in the Data Deduplication API documentation. This writer is an in-box writer for Windows Server operating system versions; it does not ship in Windows Client.

Windows Server 2008 R2, Windows Server 2008 and Windows Server 2003: This writer is not supported.

Distributed File System Replication (DFSR)

The following component includes a VSS writer: Distributed File System Replication (DFSR)

Windows Vista, Windows Server 2003 and Windows XP: This writer is not supported until Windows Vista with SP1 and Windows Server 2008.

Dynamic Host Configuration Protocol (DHCP) Writer

This writer is responsible for enumerating files required for the DHCP server role. This writer is an in-box writer for Windows Server operating system versions; it does not ship in Windows Client.

The writer name string for this writer is "Dhcp Jet Writer".

The writer ID for this writer is BE9AC81E-3619-421F-920F-4C6FEA9E93AD.

File Replication Service (FRS)

The File Replication Service writer is documented in Backing Up and Restoring an FRS-Replicated SYSVOL Folder.

Windows Vista, Windows Server 2003 and Windows XP: This writer is not supported until Windows Vista with SP1 and Windows Server 2008.

File Server Resource Manager (FSRM) Writer

This writer enumerates the FSRM configuration files that are used for system state backup. During restore operations it prevents changes in FSRM configuration and temporarily halts enforcement of quotas and file screens. After the restore is complete, the writer refreshes FSRM with the configuration that was restored. This writer is only present when FSRM (part of File Services role) is both installed and running. This writer is an in-box writer for Windows Server operating system versions; it does not ship in Windows Client.

Windows Server 2003: This writer is not supported until Windows Server 2003 R2.

The writer name string for this writer is "FSRM Writer".

The writer ID for this writer is 12CE4370-5BB7-4C58-A76A-E5D5097E3674.

Hyper-V Writer

The Hyper-V VSS writer is documented in the Hyper-V API documentation. This writer is an in-box writer for Windows Server operating system versions; it does not ship in Windows Client.

Windows Server 2003: This writer is not supported until Windows Server 2008.

IIS Configuration Writer

The IIS configuration writer is responsible for enumerating the IIS configuration files.

Windows Vista, Windows Server 2003 and Windows XP: This writer is not supported until Windows Vista with SP1 and Windows Server 2008. Requesters are required to back up the IIS configuration files, including the .NET FX machine.config file or the ASP.NET root web.config file.

This writer does not back up the .NET FX machine.config file or the ASP.NET root web.config file because these files are enumerated by the System Writer.

This writer backs up all files that are in the %windir%\system32\inetsrv\config\schema and %windir%\system32\inetsrv\config directories, except for files that are enumerated by the System Writer.

The writer ID for the IIS configuration writer is 2A40FD15-DFCA-4aa8-A654-1F8C654603F6.

IIS Metabase Writer

The Internet Information Server (IIS) metabase writer is responsible for enumerating the IIS metabase file.

Windows Vista, Windows Server 2003 and Windows XP: This writer is not supported until Windows Vista with SP1 and Windows Server 2008. Requesters are required to back up the IIS metabase file.

The writer ID for the IIS metabase writer is 59B1f0CF-90EF-465F-9609-6CA8B2938366.

Microsoft Message Queuing (MSMQ) Writer

This writer reports the Microsoft Message Queuing (MSMQ) data files.

Windows Server 2008 R2, Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: This writer is not supported.

The writer name string for this writer is "MSMQ Writer (*SvcName*)", where *SvcName* is the name of the MSMQ service the writer is associated with. For default installation, the service name is "MSMQ". For clustered instances, the service name is MSMQ$*ResourceName*, where *ResourceName* is the clustered MSMQ resource name.

The writer ID for this writer is 7E47B561-971A-46E6-96B9-696EEAA53B2A.

MSSearch Service Writer

This writer exists to delete search index files from shadow copies after creation. This is done to minimize the impact of Copy-on-Write I/O during regular I/O on these files on the shadow-copied volume.

Windows Server 2003: This writer is not supported.

To install this writer on the server, you must install the File Services role and enable the Windows Search Service.

The writer name string for this writer is "MSSearch Service Writer".

The writer ID for the MSSearch service writer is CD3F2362-8BEF-46C7-9181-D62844CDC0B2.

NPS VSS Writer

The NPS writer is responsible for enumerating the Network Policy Server (NPS) configuration files for servers that have the Network Policy and Access Services role installed.

Windows Vista, Windows Server 2003 and Windows XP: This writer is not supported until Windows Vista with SP1 and Windows Server 2008.

The writer name string for this writer is "NPS VSS Writer".

The writer ID for the NPS VSS writer is 0x35E81631-13E1-48DB-97FC-D5BC721BB18A.

Performance Counters Writer

This writer reports the performance counter configuration files. These files are only modified during application installation and should be backed up and restored during system state backups and restores.

Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: This writer is not supported. Requesters are required to back up these files manually. (See Backing Up and Restoring System State.)

The writer name string for this writer is "Performance Counters Writer".

The writer ID for the Performance Counters Writer is 0BADA1DE-01A9-4625-8278-69E735F39DD2.

Registry Writer

The registry writer is reports the Windows registry files to enable in-place backups and restores of the registry. It does not report user hives.

Windows Server 2003: In Windows Server 2003, this writer uses an intermediate repository file (sometimes called a "spit file") to store registry data. (See Registry Backup and Restore Operations Under VSS.)

Windows XP: This writer is not supported. (See Registry Backup and Restore Operations Under VSS.)

The writer name string for this writer is "Registry Writer".

The writer ID for the registry writer is AFBAB4A2-367D-4D15-A586-71DBB18F8485.

Remote Desktop Services (Terminal Services) Gateway VSS Writer

This writer is responsible for enumerating the Remote Desktop Services (Terminal Services) Gateway files for servers that have Remote Desktop Gateway, a subrole of Remote Desktop Services role, installed.

Windows Server 2003: This writer is not supported.

This writer is an in-box writer for Windows Server operating system versions; it does not ship in Windows Client.

The Remote Desktop Services Gateway depends on several registry keys being backed up and therefore needs to be backed up and restored together with the registry.

The writer name string for this writer is "TS Gateway Writer".

The writer ID for this writer is 368753EC-572E-4FC7-B4B9-CCD9BDC624CB.

Remote Desktop Services (Terminal Services) Licensing VSS Writer

This writer is responsible for enumerating the Remote Desktop Services Licensing (RD Licensing) or Terminal

Services Licensing (TS Licensing) files for servers that have Remote Desktop Licensing, a subrole of Remote Desktop Services role, installed.

Windows Server 2003: This writer is not supported.

This writer is an in-box writer for Windows Server operating system versions; it does not ship in Windows Client.

Remote Desktop Services Licensing depends on several registry keys being backed up and therefore needs to be backed up and restored together with the registry.

The writer name string for this writer is "TermServLicensing".

The writer ID for this writer is 5382579C-98DF-47A7-AC6C-98A6D7106E09.

Shadow Copy Optimization Writer

This writer deletes certain files from volume shadow copies. This is done to minimize the impact of Copy-on-Write I/O during regular I/O on these files on the shadow-copied volume. The files that are deleted are typically temporary files or files that do not contain user or system state.

Windows Server 2003 and Windows XP: This writer is not supported.

The writer name string for this writer is "Shadow Copy Optimization Writer".

The writer ID for the shadow copy optimization writer is 4DC3BDD4-AB48-4D07-ADB0-3BEE2926FD7F.

Sync Share Service Writer

Windows Server 2012 R2: This writer is included with Windows Server 2012 R2 and newer server versions. It is not compatible with desktop versions.

This writer is responsible for enumerating the sync shares on servers that have the Sync Share Service installed, and for ensuring that their metadata and data remain consistent during backup and restore.

This writer is only present when Sync Share Service is both installed and running.

There will be one VSS writer component for each sync share. This includes the metadata and data paths. These must be backed up together for consistency.

The writer name string is "Sync Share Service VSS writer".

The writer ID is D46BF321-FDBA-4A35-8EC3-454DF03BC86A.

System Writer

The system writer enumerates all operating system and driver binaries and it is required for a system state backup.

Windows Server 2003 and Windows XP: This writer is not supported.

This writer runs as part of the Cryptographic Services (CryptSvc) service. The system writer generates a file list that contains the following files:

- All static files that have been installed. A static file is a file that is listed in the component manifest with the **writeableType** file attribute set to "static" or "". Static files include all files that are protected by Windows Resource Protection (WRP). However, not all static files are WRP-protected files. For example, game files are static but not WRP-protected so that administrators can change parental control settings.
- The contents of the Windows Side-by-Side (WinSxS) directory, including all manifests, optional

components, and third-party Win32 files.

> **NOTE**
> Many of the files in the %windir%\system32 directory are hard links to files in the WinSxS directory.

- All PnP files for installed drivers (owned by PnP).
- All user-mode services and non-PnP drivers.
- All catalogs owned by CryptSvc.

The restore application is responsible for laying down the files and registry and setting ACLs to match the system shadow copy. The appropriate hard links must also be created for a system state restore to succeed.

The writer name string for this writer is "System Writer".

The writer ID for the system writer is E8132975-6F93-4464-A53E-1050253AE220.

Task Scheduler Writer

This writer reports the task scheduler's task files.

Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: This writer is not supported. Requesters are required to back up these files manually. (See Backing Up and Restoring System State.)

The writer name string for this writer is "Task Scheduler Writer".

The writer ID for the writer is D61D61C8-D73A-4EEE-8CDD-F6F9786B7124.

VSS Metadata Store Writer

This writer reports the writer metadata files for all VSS express writers.

Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: This writer is not supported.

The writer name string for this writer is "VSS Express Writer Metadata Store Writer".

The writer ID for the writer is 75DFB225-E2E4-4D39-9AC9-FFAFF65DDF06.

Windows Deployment Services (WDS) Writer

This writer reports the Windows Deployment Services (WDS) data files.

Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: This writer is not supported.

The writer name string for this writer is "WDS VSS Writer".

The writer ID for this writer is 82CB5521-68DB-4626-83A4-7FC6F88853E9.

Windows Internal Database (WID) Writer

This writer reports the Windows Internal Database (WID) data files.

Windows Server 2008 R2, Windows 7, Windows Server 2008, Windows Vista, Windows

Server 2003 and Windows XP: This writer is not supported.

The writer name string for this writer is "WIDWriter".

The writer ID for this writer is 8D5194E1-E455-434A-B2E5-51296CCE67DF.

Windows Internet Name Service (WINS) Writer

This writer is responsible for enumerating files required for WINS.

Windows XP: This writer is not supported.

The writer name string for this writer is "WINS Jet Writer".

The writer ID for this writer is F08C1483-8407-4A26-8C26-6C267A629741.

WMI Writer

This writer is used for identifying WMI-specific state and data during backup operations. The data includes files from the WBEM repository.

Windows Server 2003 and Windows XP: This writer is not supported.

The writer name string for this writer is "WMI Writer".

The writer ID for this writer is A6AD56C2-B509-4E6C-BB19-49D8F43532F0.

Using the Volume Shadow Copy Service

3/5/2021 • 2 minutes to read • Edit Online

VSS backup and restore operations each use a protocol for the interaction of the systems that use mass storage (writers) and those that back it up (requesters).

Both backup and restore operations are primarily driven by the requester, which controls writer and provider behavior by generating systemwide COM events for the writer to process. Because event-generating methods are asynchronous, writers do have limited control into the requester's state through the asynchronous handlers available to VSS (see **IVssAsync**).

This interaction requires basic data structures describing files and groups of files (*components*), as well as a metadata model to allow the storage of backup information and to permit writer/requester communication.

- VSS Application Compatibility
- Configuring VSS
- VSS Metadata
- Working with Selectability and Logical Paths
- Overview of Processing a Backup Under VSS
- Overview of Processing a Restore Under VSS

VSS Application Compatibility

3/5/2021 • 4 minutes to read • Edit Online

When developing your own VSS application, you should observe the following guidelines and restrictions. You may find it helpful to refer to the sample code for VSS requesters, providers, and writers that is provided in the Microsoft Windows Software Development Kit (SDK).

> **NOTE**
>
> The Windows SDK can be used to develop VSS applications only for Windows Vista and later Windows operating system versions. It cannot be used to develop VSS requesters, providers, or writers for Windows Server 2003 R2, Windows Server 2003, or Windows XP.

Windows Server 2003 R2, Windows Server 2003 and Windows XP: VSS is available in the Volume Shadow Copy Service 7.2 SDK, which you can download from https://www.microsoft.com/download/details.aspx?id=23490. Note that the 64-bit vssapi.lib files in the directories under the Win2003\Obj directory can be used for the 64-bit versions of Windows Server 2003 R2, Windows Server 2003, and Windows XP. This SDK also provides sample code for VSS requesters, providers, and writers.

Compiling VSS Applications

When developing a requester, such as a backup application:

- Include the following headers:

  Vss.h
  VsWriter.h
  VsBackup.h

- Link the following library:

  VssApi.Lib

When developing a writer:

- Include the following headers:

  Vss.h
  VsWriter.h

- Link the following library:

  VssApi.lib

Supported Configurations and Restrictions

The following list describes supported configurations and restrictions:

- VSS is provided and supported on versions of the Windows operating system beginning with Windows XP.

- The following table summarizes compatibility information across Windows versions. Note that if a VSS application is "compiled for" a specified Windows version, this means that the application was compiled using the header files and libraries that are specific to that version.

> **NOTE**
> Hardware providers will run only on Windows server operating system versions. They will not run on Windows client operating system versions.

> **NOTE**
> In the following tables, Windows Server 2008 with Service Pack 2 (SP2) should be considered the same as Windows Server 2008. For more information about Windows Server 2008 with SP2, see https://go.microsoft.com/fwlink/p/?linkid=178730. Windows Server 2003 R2 should be considered the same as Windows Server 2003.

> **NOTE**
> If a VSS application is compiled for Windows Server 2003 or later, it will also run on later versions of Windows.

VSS REQUESTERS, WRITERS, AND PROVIDERS COMPILED FOR	WILL RUN ON
Windows Server 2008 R2 (64-bit), Windows 7 (64-bit), Windows Server 2008 (64-bit) and Windows Vista (64-bit)	Windows Server 2008 R2 (64-bit), Windows 7 (64-bit), Windows Server 2008 (64-bit) and Windows Vista (64-bit)
Windows Server 2008 R2 (32-bit), Windows 7 (32-bit), Windows Server 2008 (32-bit) and Windows Vista (32-bit)	Windows Server 2008 R2 (32-bit), Windows 7 (32-bit), Windows Server 2008 (32-bit) and Windows Vista (32-bit)
Windows Server 2003 (64-bit)	Windows Server 2008 R2 (64-bit), Windows 7 (64-bit), Windows Server 2008 (64-bit), Windows Vista (64-bit), and Windows Server 2003 (64-bit)
Windows Server 2003 (32-bit)	Windows Server 2008 R2 (32-bit), Windows 7 (32-bit), Windows Server 2008 (32-bit), Windows Vista (32-bit), and Windows Server 2003 (32-bit) [!Note] Requesters will also run on Windows Server 2003 (64-bit).
Windows XP 64-Bit Edition	Windows Server 2003 (64-bit) and Windows XP 64-Bit Edition
Windows XP (32-bit)	Windows XP (32-bit)

TO COMPILE A VSS REQUESTER, WRITER, OR PROVIDER FOR	USE
Windows Server 2008 R2 or Windows 7	Windows SDK for Windows 7 (Available from the Windows Download Center.)
Windows Server 2008 or Windows Vista	Windows SDK for Windows Server 2008 (Available from the Windows SDK Developer Center.)
Windows Server 2003 R2, Windows Server 2003, or Windows XP	Volume Shadow Copy Service 7.2 SDK

- All 32-bit VSS applications (requesters, providers, and writers) must run as native 32-bit or 64-bit applications. Running them under WOW64 is not supported.

  Windows Server 2003 and Windows XP: Running 32-bit VSS requesters under WOW64 is supported, but not for system-state backups. Running 32-bit VSS providers and writers under WOW64 is not supported. Support for running 32- bit requesters under WOW64 was removed in Windows Vista and subsequent versions.

- A shadow copy that was created on Windows Server 2003 R2 or Windows Server 2003 cannot be used on a computer that is running Windows Server 2008 R2 or Windows Server 2008. A shadow copy that was created on Windows Server 2008 R2 or Windows Server 2008 cannot be used on a computer that is running Windows Server 2003. However, a shadow copy that was created on Windows Server 2008 can be used on a computer that is running Windows Server 2008 R2, and vice versa.
- To support shadow copies, a system running VSS must have at least one NTFS file system. This file system will host the shadow copy's "diff area." For more information, see System Provider.
- Given the presence of one NTFS file system, and given the appropriate choice of context (see Shadow Copy Context Configurations), any supported local file system can be shadow copied.
- You can make shadow copies only for locally mounted file systems. Remote shares and other cross-mounted file systems cannot be shadow copied by the system that mounts them. These file systems can be shadow copied only by the systems that serve out the file systems.
- Writers and requesters should specify only local resources. Local resources are sets of files whose absolute path starts with a drive letter, and the drive letter cannot be associated with a mounted folder on a remote share.
- The maximum number is of software shadow copies for each volume is 512. However, by default you can only maintain 64 shadow copies that are used by the Shadow Copies of Shared Folders feature. To change the limit for the Shadow Copies of Shared Folders feature, use the MaxShadowCopies registry key.
- The Backup Components infrastructure does not support backing up cluster resources as writer components. To back up cluster resources, applications should assume that the path is local to a specified particular cluster node.
- [!Note]

  > Microsoft does not provide developer or IT professional technical support for implementing online system state restores on Windows (all releases).

When backing up and recovering system state, the recommended strategy is to back up and recover the system and boot volumes in addition to the files enumerated by the system state writers.

> **NOTE**
>
> System state writers are writers that have the **VSS_USAGE_TYPE** attribute set to either VSS_UT_BOOTABLESYSTEMSTATE or VSS_UT_SYSTEMSERVICE.

Configuring VSS

3/5/2021 • 2 minutes to read • Edit Online

By using VSS configuration tools—including setting the shadow copy context, choosing a restore method, and changing a restore target—writers and requesters control how a backup or restore operation takes place:

- Shadow Copy Context Configurations
- VSS Backup Configurations
- VSS Restore Configurations

Shadow Copy Context Configurations

3/5/2021 • 2 minutes to read • Edit Online

Requesters control a shadow copy's features by setting its context. This context indicates whether the shadow copy will survive the current operation, and the degree of writer/provider coordination.

Persistence and Shadow Copy Context

A shadow copy may be *persistent*—that is, the shadow copy is not deleted following the termination of a backup operation or the release of an **IVssBackupComponents** object.

Persistent shadow copies require **_VSS_SNAPSHOT_CONTEXT** contexts of **VSS_CTX_CLIENT_ACCESSIBLE**, **VSS_CTX_APP_ROLLBACK**, or **VSS_CTX_NAS_ROLLBACK**. Persistent shadow copies can be made only for NTFS volumes.

Nonpersistent shadow copies are created with contexts of **VSS_CTX_BACKUP** or **VSS_CTX_FILE_SHARE_BACKUP**. Nonpersistent shadow copies can be made for NTFS and non-NTFS volumes.

Writer Participation and Shadow Copies

A shadow copy context can be classified as either involving writers or not involving writers.

Shadow copy contexts that involve writers in their creation include:

- **VSS_CTX_APP_ROLLBACK**
- **VSS_CTX_BACKUP**
- **VSS_CTX_CLIENT_ACCESSIBLE_WRITERS**

Those that do not involve writers in their creation include:

- **VSS_CTX_CLIENT_ACCESSIBLE**
- **VSS_CTX_FILE_SHARE_BACKUP**
- **VSS_CTX_NAS_ROLLBACK**

One context can be used with both types of shadow copies, but cannot be used in creating a shadow copy:

- **VSS_CTX_ALL**

Creating a shadow copy with a context of **VSS_CTX_ALL** (using **IVssBackupComponents::StartSnapshotSet** and **IVssBackupComponents::DoSnapshotSet**) is not supported.

Operations that support a context of **VSS_CTX_ALL** are the administrative operations **IVssBackupComponents::Query**, **IVssBackupComponents::DeleteSnapshots**, **IVssBackupComponents::BreakSnapshotSet**, and **IVssBackupComponents::ExposeSnapshot**.

Obtaining Shadow Copy Information

If a requester knows the identifying GUID of a shadow copy (its **VSS_ID**), it can obtain information about the context of a specific shadow copy (identified by its **VSS_ID**) by unpacking the **VSS_SNAPSHOT_PROP** structure returned by a call to **IVssBackupComponents::GetSnapshotProperties**.

To obtain context information about all shadow copies on a system, a requester examines the

m_lSnapshotAttributes member of the **Obj.Snap** member of the **VSS_OBJECT_PROP** (which is a **VSS_SNAPSHOT_PROP** structure) structure obtained by using **IVssEnumObject** to iterate over the list of objects returned by a call to **IVssBackupComponents::Query**.

VSS Backup Configurations

3/5/2021 • 2 minutes to read • Edit Online

There are a number of conventionally supported backup types—incremental, differential, and full—that VSS is aware of, as well as a backup configuration peculiar to VSS.

In defining a backup configuration, a requester and the writers on a system indicate how data will be written to a storage device, how the shadow copy mechanism will be deployed, and what information needs to be saved. Interaction between writers and requesters is determined by the type of backup a requester wants to perform and the kinds (or schemas) that each writer can support:

- VSS Backup State
- Writer Backup Schema Support
- File Level Schema Support

VSS Backup State

3/5/2021 • 3 minutes to read • Edit Online

During a backup operation, the requester uses **IVssBackupComponents::SetBackupState** to define the type of operation in progress.

This information is not included in an easily retrievable form in the Backup Components Document, so requester developers should store this information independently on any backup media.

The backup state contains the following:

Backup Type

The backup type specifies criteria for identifying files to be backed up. The evaluation of these criteria must be made using the VSS API.

When deciding on the type of backup to pursue and which writers to work with, requesters should examine the kinds (or schemas—see Writer Backup Schema Support) of backup operations that each of the system's writers supports. Backups under VSS can be the following types:

- Full (VSS_BT_FULL)—files will be backed up regardless of their last backup date. The backup history of each file will be updated, and this type of backup can be used as the basis of an incremental or differential backup. Restoring a full backup requires only a single backup image.
- Copy Backup (VSS_BT_COPY)—like the VSS_BT_FULL backup type, files will be backed up regardless of their last backup date. However, the backup history of each file will not be updated, and this sort of backup cannot be used as the basis of an incremental or differential backup.
- Incremental (VSS_BT_INCREMENTAL)—the VSS API is used to ensure that only files that have been changed or added since the last full or incremental backup are to be copied to a storage medium. Restoring an incremental backup requires the original backup image and all incremental backup images made since the initial backup.
- Differential (VSS_BT_DIFFERENTIAL)—the VSS API is used to ensure that only files that have been changed or added since the last full backup are to be copied to a storage media; all intermediate backup information is ignored. Restoring a differential backup requires the original backup image and the most recent differential backup image made since the last full backup.
- Log File (VSS_BT_LOG)—only a writer's log files (files added to a component with the **IVssCreateWriterMetadata::AddDataBaseLogFiles** method, and retrieved by a call to **IVssWMComponent::GetDatabaseLogFile**) will be backed up. This backup type is specific to VSS.

It is possible for requesters to implement these backups using information and methods outside of VSS. Only when a requester implements a backup using the VSS API should it be said to have one of the listed backup types. For instance, a requester participates in a VSS_BT_LOG type of backup only if it used the information returned by **IVssWMComponent::GetDatabaseLogFile** to identify log files. Similarly, the VSS_BT_INCREMENTAL and VSS_BT_DIFFERENTIAL types apply only to incremental or differential operations, as described in Incremental and Differential Backups.

Specification about Selectability

A VSS backup may choose to respect VSS notions of component selectability—this is referred to as running in *component mode*—or to ignore them.

An example of not running in component mode would be performing a system image backup, where the backup application would need writer cooperation to ensure data stability but where selection of components

would be irrelevant.

Saving Bootable State

VSS supports saving the running system state in a fully bootable configuration. However, this is not always necessary, and writer preparation to save a bootable state can sometimes degrade real-time performance of a running system.

Therefore, requesters indicate whether a backup will include a bootable system state as an argument to **IVssBackupComponents::SetBackupState**. Writers determine whether they have to support saving the bootable system state by calling **CVssWriter::IsBootableStateBackedUp**.

Even if bootable system state is not selected, shadow copies of the system files will be made and the files may be backed up.

However, great care should be exercised in restoring system files if the backup did not save the bootable system state (see Backing Up and Restoring System State in Windows Server 2003 R2 and Windows Server 2003 SP1).

It is not possible to recover this information from a retrieved Backup Components Document, so requester authors should store whether the system was backed up with a bootable system state or not.

Partial File Support

Some writers support file restoration through the overwriting of parts of the files they manage. A requester may be designed to take advantage of this, and if so it indicates this by setting the information in **IVssBackupComponents::SetBackupState**.

Writer Backup Schema Support

3/5/2021 • 3 minutes to read • Edit Online

To fully implement a backup requires participation of the system's writers. Different types of supported backups are referred to as schemas and are indicated by a bitmask (or bitwise OR) of members of the VSS_BACKUP_SCHEMA enumeration. The currently supported valid schemas include the following:

- Default Schema: Full (VSS_BS_UNDEFINED)—indicates that a writer supports a backup where all files will be backed up regardless of their last backup date. The backup history of each file can be updated by the requester, and writers supporting the VSS_BS_TIMESTAMPED enumeration value, it will store an updated time stamp with the requester. This backup scheme can be used as the basis of an incremental or differential backup.

  Restoring a full backup requires only a single backup image.

- Copy Backup (VSS_BS_COPY)—like the VSS_BS_FULL backup schema, indicates that a writer supports a backup where all files will be backed up regardless of their last backup date. However, the backup history of each file will not be updated by either requester or writer, and this sort of backup cannot be used as the basis of an incremental or differential backup.

- Log File (VSS_BS_LOG)—only a writer's log files are to be backed up. This requires a writer to have added at least one file to at least one component using the IVssCreateWriterMetadata::AddDatabaseLogFiles method. This backup type is specific to VSS.

- Custom Restore Locations (VSS_BS_WRITER_SUPPORTS_NEW_TARGET)—indicates writer support for a requester changing, at restore time, where its files are restored. This means that a writer has been coded to check for such relocation (using IVssComponent::GetNewTarget), and has the capacity for working with relocated files.

- Restore with Move (VSS_BS_WRITER_SUPPORTS_RESTORE_WITH_MOVE)—indicates that the writer supports running multiple writer instances with the same class ID, and it supports a requester moving a component to a different writer instance at restore time. The writer instance is specified by using a persistent writer instance name that was passed as the *wszWriterInstanceName* parameter to the CVssWriter::Initialize method. A requester can obtain the writer instance name using IVssExamineWriterMetadataEx::GetIdentityEx and move components during restore using IVssBackupComponentsEx::SetSelectedForRestoreEx.

  Windows Server 2003 and Windows XP: This value is not supported until Windows Server 2003 with Service Pack 1 (SP1).

- Incremental (VSS_BS_INCREMENTAL)—indicates that the writer uses the VSS API to help the requester, ensuring that only files that have been changed or added since the last full or incremental backup are to be copied to a storage medium.

  Restoring an incremental backup requires the original backup image and all incremental backup images made since the initial backup.

- Differential (VSS_BS_DIFFERENTIAL)—indicates that the writer uses the VSS API to help the requester ensure that only files that have been changed or added since the last full backup are to be copied to a storage medium; all intermediate backup information is ignored.

  Restoring a differential backup requires the original backup image and the most recent differential backup image made since the last full backup.

- Incremental/Differential: Time Stamping Support (VSS_BS_TIMESTAMPED)—indicates that a writer supports using the VSS time-stamp mechanism to include files in incremental or differential operations. At backup, the writer must store a *file set's* backup stamp with the **IVssComponent::SetBackupStamp** method, and at restore retrieve it with **IVssComponent::GetPreviousBackupStamp**.
- Incremental/Differential: Time of Last Modification Support (VSS_BS_LAST_MODIFY)—indicates that when implementing incremental or differential backups with differenced files, a writer can provide last modification time information independently. This information can be provided to a requester via the **IVssComponent::AddDifferencedFilesByLastModifyTime** method.
- Incremental/Differential: Support Limitation (VSS_BS_EXCLUSIVE_INCREMENTAL_DIFFERENTIAL)—indicates writer support of both differential or increment backup schemas, but only exclusively: for example, you cannot follow an incremental backup with a differential backup.
- Independent System State (VSS_BS_INDEPENDENT_SYSTEM_STATE)—indicates that the writer supports backing up data that is part of the system state, but that can also be backed up independently of the system state.

  Windows Server 2003 and Windows XP: This value is not supported until Windows Vista.
- Roll-Forward Restore (VSS_BS_ROLLFORWARD_RESTORE)—indicates that the writer supports a requester setting a roll-forward restore point using **IVssBackupComponentsEx2::SetRollForward**.

  Windows Server 2003 and Windows XP: This value is not supported until Windows Vista.
- Restore Rename (VSS_BS_RESTORE_RENAME)—indicates that the writer supports a requester setting a restore name using **IVssBackupComponentsEx2::SetRestoreName**.

  Windows Server 2003 and Windows XP: This value is not supported until Windows Vista.
- Authoritative Restore (VSS_BS_AUTHORITATIVE_RESTORE)—indicates that the writer supports a requester setting authoritative restore using **IVssBackupComponentsEx2::SetAuthoritativeRestore**.

Writers set their schemas using the **IVssCreateWriterMetadata::SetBackupSchema** method, and a requester obtains each writer's schema by calling **IVssExamineWriterMetadata::GetBackupSchema**.

File Level Schema Support

3/5/2021 • 2 minutes to read • Edit Online

Writers can fine-tune the performance of various types of backup at the *file set* level by indicating through a file specification backup type, indicated by a bit mask or bitwise OR of the members of the **VSS_FILE_SPEC_BACKUP_TYPE** enumeration.

For specified types of backup, the writer indicates the following:

- Whether it will be necessary to shadow copy the volume to properly back up a file set. For instance, if the files in a file set are not exclusively opened by the writer and it can ensure that it is stable, a shadow copy is not needed.
- Whether the requester has to perform the backup in such a way that, regardless of last modification time or other issues, the version of the file set's files current at backup will be restored. Typically, this means that the file set is copied as part of the backup.

The values of **VSS_FILE_SPEC_BACKUP_TYPE** that indicate shadow copy requirement are:

- VSS_FSBT_ALL_SNAPSHOT_REQUIRED
- VSS_FSBT_FULL_SNAPSHOT_REQUIRED
- VSS_FSBT_DIFFERENTIAL_SNAPSHOT_REQUIRED
- VSS_FSBT_INCREMENTAL_SNAPSHOT_REQUIRED
- VSS_FSBT_LOG_SNAPSHOT_REQUIRED

The values of **VSS_FILE_SPEC_BACKUP_TYPE** that indicate the requirement to back up a file are:

- VSS_FSBT_ALL_BACKUP_REQUIRED
- VSS_FSBT_FULL_BACKUP_REQUIRED
- VSS_FSBT_DIFFERENTIAL_BACKUP_REQUIRED
- VSS_FSBT_INCREMENTAL_BACKUP_REQUIRED
- VSS_FSBT_LOG_BACKUP_REQUIRED

By default, file sets are tagged with a file specification backup type of VSS_FSBT_ALL_BACKUP_REQUIRED | VSS_FSBT_ALL_SNAPSHOT_REQUIRED, meaning that a shadow copy is always required in handling the file set's files, and that the files should be copied in all types of backup.

VSS Restore Configurations

3/5/2021 • 2 minutes to read • Edit Online

File restoration on a running system can be problematic. It is important that running applications (writers) indicate what to do when difficulties arise during restores, for instance, if the file being restored is currently in use.

Under VSS, writers have two complementary ways of managing restores—*restore methods* and *restore targets*.

In addition, requesters can choose to restore files to previously unspecified locations and notify writers (see **IVssBackupComponents::AddNewTarget**).

The restore method (also call the original restore target) is specified by a writer at a backup time, and sets a writer-wide definition of the method to be used to restore all its components in the future. All files and components managed by a writer share the same restore method.

Restore targets allow writers to change how specific components are to be restored at restore time. Unlike restore methods, restore targets are defined for a component set.

A detailed discussion of the usage of restore methods and restore targets is found in the topics listed below:

- VSS Restore State
- Setting VSS Restore Methods
- Setting VSS Restore Targets
- Setting VSS Restore Options

(For information on restores that do not use these mechanisms, see Restores without Writer Participation.)

VSS Restore State

3/5/2021 • 2 minutes to read • Edit Online

During a restore operation, the requester can use the **IVssBackupComponents::SetRestoreState** method to define the type of restore operation in progress. However, most restore operations will not need to override the default restore type (VSS_RTYPE_UNDEFINED). Writers should treat this restore type as if it were VSS_RTYPE_BY_COPY. For more information about restore type values, see the **VSS_RESTORE_TYPE** enumeration.

Setting VSS Restore Methods

3/5/2021 • 2 minutes to read • Edit Online

The configuration of restore operations actually begins during data backup, when writers specify, in their Writer Metadata Documents, how their data should be restored.

These specifications, referred to either as *restore methods* or original *restore targets*, can be modified during restore by writers setting new restore targets or by requesters restoring to new locations (see Non-Default Backup and Restore Locations).

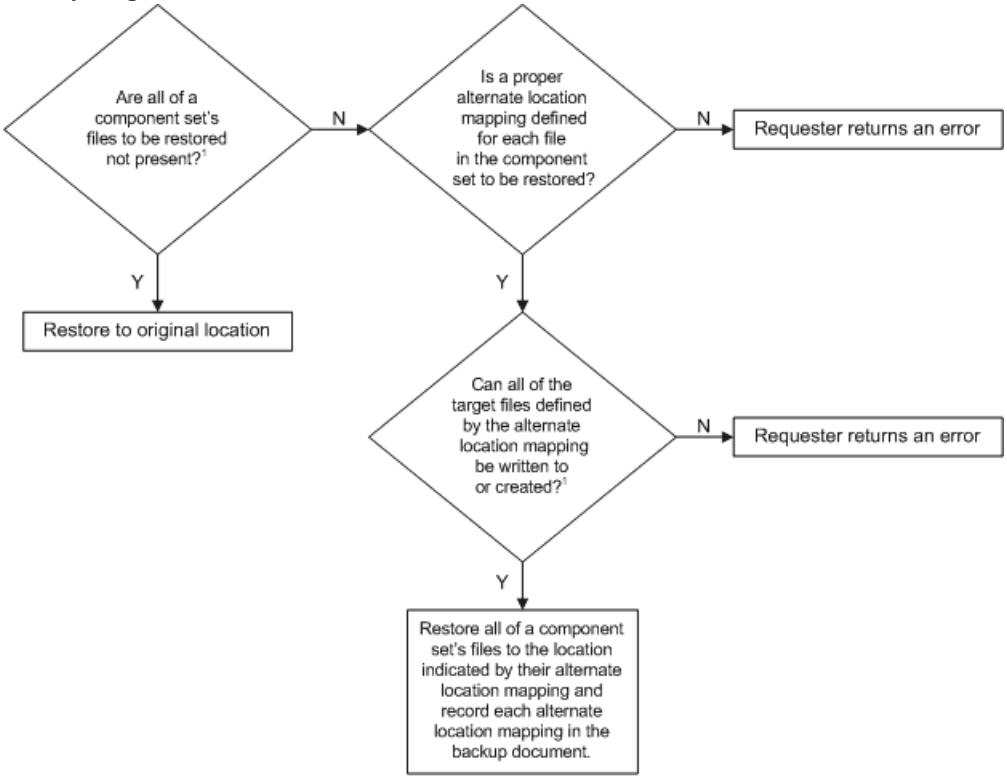
By calling **IVssCreateWriterMetadata::SetRestoreMethod**, a writer indicates which restore method should be used in its Writer Metadata Document. The restore method is set writer wide and applied to all files in all components that a writer manages.

A requester obtains (and must respect) this information by calling **IVssExamineWriterMetadata::GetRestoreMethod**.

The restore method is defined by a **VSS_RESTOREMETHOD_ENUM** enumeration, which is passed to **IVssCreateWriterMetadata::SetRestoreMethod** and returned from **IVssExamineWriterMetadata::GetRestoreMethod**.

The Writer Metadata Document supports the following valid restore methods (a restore method of VSS_RME_UNDEFINED indicates a writer error). The figures summarize how the various supported and defined restore methods should be implemented (VSS_RME_CUSTOM has no figure associated with it, because by definition it is specific to the writer and must follow the specific writer APIs and documentation):

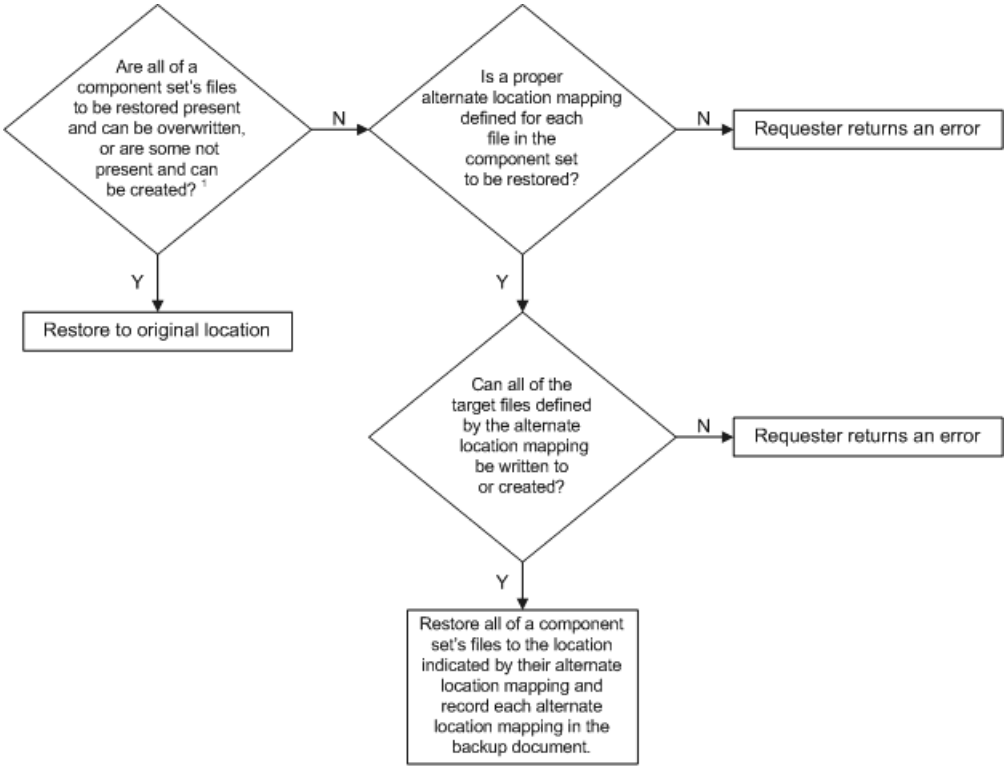
- VSS_RME_RESTORE_IF_NOT_THERE. Restore component files to disk if none of the files are on the disk already. Target file status should be checked after a **PreRestore** event.



[1] This check must be performed after the PreRestore event.

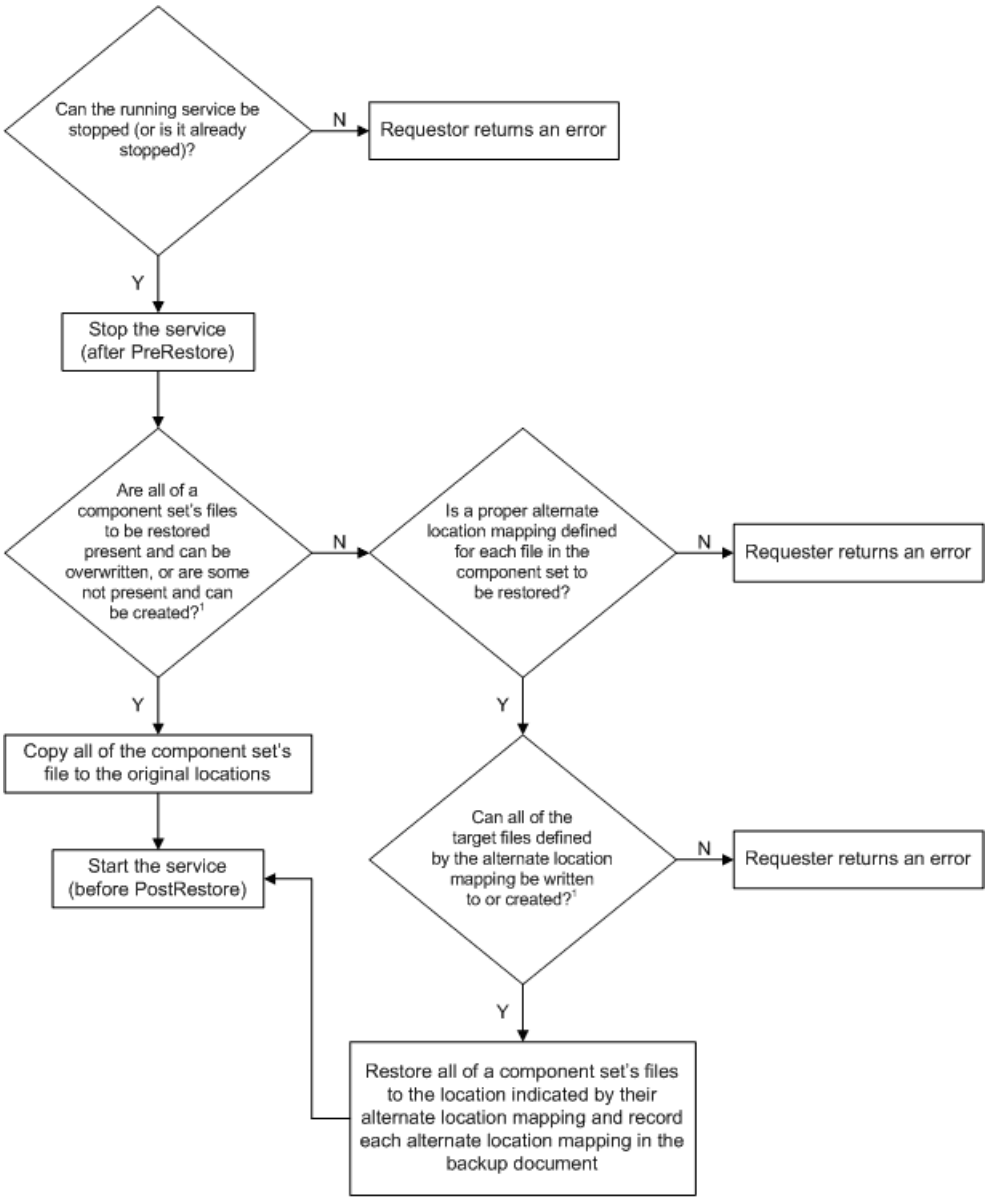
- VSS_RME_RESTORE_IF_CAN_REPLACE. Restore files to disk if all the files can be replaced. Target file status

should be checked after a PreRestore event.



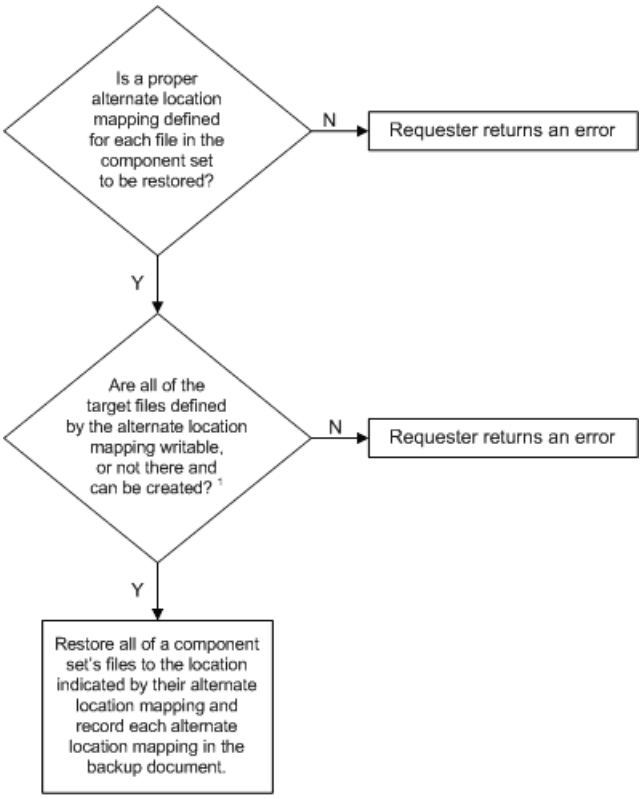
[1] This check must be performed after the PreRestore event.

- VSS_RME_STOP_RESTORE_START. A service will be stopped prior to restoring the files.



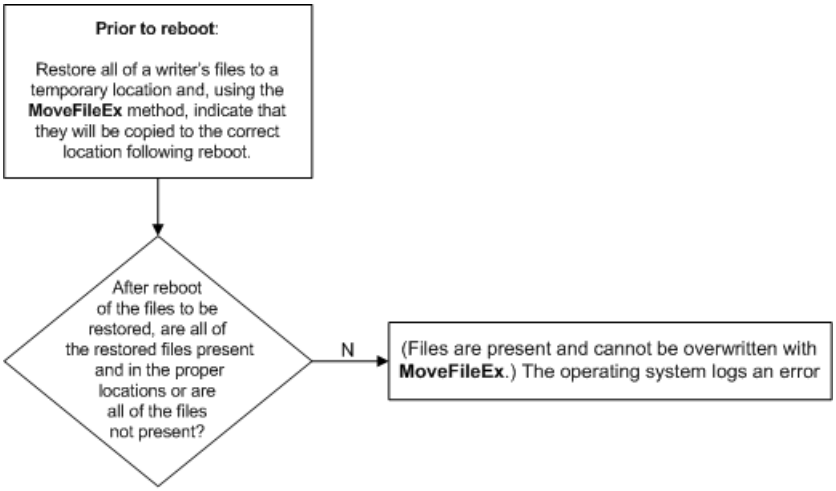
[1] This check must be performed after the PreRestore event.

- VSS_RME_RESTORE_TO_ALTERNATE_LOCATION. Restore files to disk in an alternate location. The alternate location mappings are specified in the Writer Metadata Document.



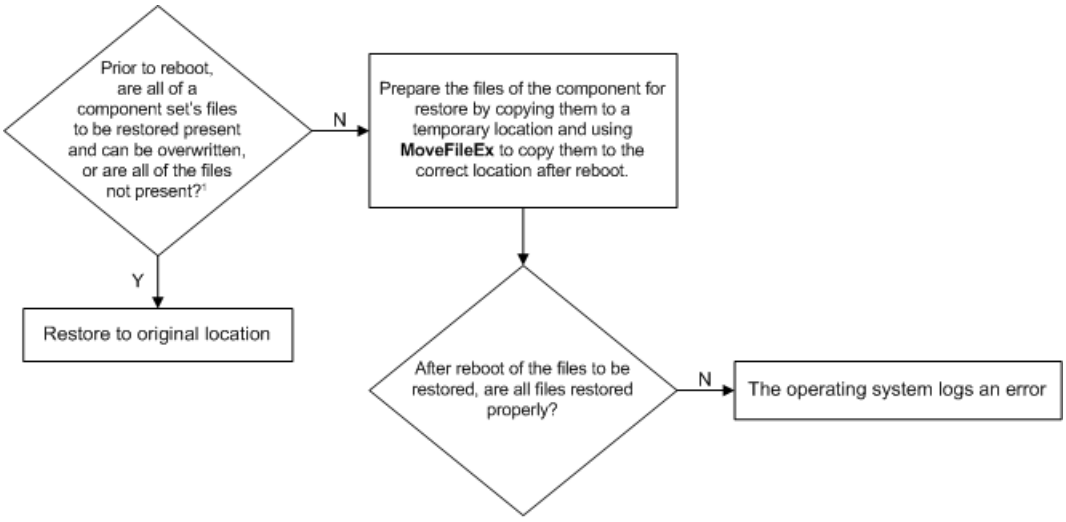
[1] This effectively applies VSS_RME_RESTORE_IF_NOT_THERE to the alternate location.

- VSS_RME_RESTORE_AT_REBOOT. Cause files to be restored (overwritten) when the computer is rebooted.



Note: No use of alternate location mappings can be made when files are restored at reboot.

- VSS_RME_RESTORE_AT_REBOOT_IF_CANNOT_REPLACE. If a file could not be restored to disk on a running system, then it is restored (overwritten) when the computer is rebooted.



Note: No use of alternate location mappings can be made when files are restored at reboot.

[1] This check must be performed after the PreRestore event.

- VSS_RME_CUSTOM. None of the predefined methods will work; the backup application must use specialized APIs to perform the restore operation. For this backup method, the requester must completely understand the writer in question. See Special VSS Usage Cases for currently supported instances.

Setting VSS Restore Targets

3/5/2021 • 2 minutes to read • Edit Online

The **IVssComponent** interface enables a writer to fine-tune exactly how files are restored on a component-by-component basis.

Because it is possible that the system configuration during restore is something other than that anticipated during backup, the restore target mechanism is provided.

It allows writers to call **IVssComponent::SetRestoreTarget** to change how those components that are *explicitly included* in the Backup Components Document are restored. This also changes the restoration mechanism used on those components that are *implicitly included*.

File restoration that takes place during a system reboot (under the **VSS_RESTOREMETHOD_ENUM** enumeration values VSS_RME_RESTORE_AT_REBOOT and VSS_RME_RESTORE_AT_REBOOT_IF_CANNOT_REPLACE) cannot be affected by restore targets because there are no running VSS services when **MoveFileEx** copies files to their final location.

Similarly, VSS_RME_CUSTOM restores may or may not be affected, because each custom restore is specific to a given writer and can choose to respect or ignore restore targets.

Requesters and writers can use **IVssComponent::GetRestoreTarget** to check the restore target of a component set.

IVssComponent supports the following restore targets, which can be set on a component set by component set basis:

- VSS_RT_ORIGINAL. The restore method specified by the **VSS_RESTOREMETHOD_ENUM** enumeration will be respected.
- VSS_RT_ALTERNATE. The files are restored to a location determined from an existing alternate location mapping. If an alternate location mapping matching a path in a component set subcomponent exists, restore to the alternate location if possible; otherwise, return an error.

Setting VSS Restore Options

3/5/2021 • 2 minutes to read • Edit Online

Restore options allow requesters to communicate customized restore options to writers.

Restore Options

Standardizing the format of the restore options allow writers and requesters to handle common custom requests. Restore options are set by the requester by calling the **IVssBackupComponents::SetRestoreOptions** method up to once per selected-for-backup component before calling the **IVssBackupComponents::PreRestore** method. The string passed in the *wszRestoreOptions* parameter to the **SetRestoreOptions** method can contain multiple values, as described below.

Format

The format of restore options, is one or more comma-separated name/value pairs, and the name is optionally prefixed with the name of the subcomponent to which it applies. The component names and option names are case-insensitive. The case-sensitivity of the values is determined by the writer. For example:

```
"Child1":"Option1"="Value1","Option2"="Value2","Child2\Grandchild3":"Option3"="Value3"
```

In this example, "Option1" only applies to the "Child1" subcomponent and its descendants, "Option2" applies to all components and their descendants, and "Option3" applies only to the "Child2\Grandchild3" subcomponents and its descendants.

The **SetRestoreOptions** method can only be called on components that are selectable for backup, while descendant nodes may not be selectable for backup, they may be selectable for restore.

Common Restore Options

These common restore options have been defined to increase interoperability between writers and requesters.

- Authoritative

  The "Authoritative" option supports multiple "Item" values, but only one "All" value.

  This entire component is authoritative.

  ```
  "Authoritative"="All"
  ```

  Only the specified item is authoritative. The format of the named item is defined by the writer. Common designations are "*" to indicate all files, "..." to indicate all files and subdirectories of the specified component.

  ```
  "Authoritative"="Item:XXX"
  ```

- Roll Forward

  After a database is restored, writers usually roll forward through logs to bring the database up to date. In the case of incremental or differential restores, the requester uses the

IVssBackupComponents::SetAdditionalRestores method to partially control the log-handling behavior - this restore option allows more granular control.

Do not roll through logs.

```
"Roll Forward"="None"
```

Roll through all logs.

```
"Roll Forward"="All"
```

Roll through logs up to specified point. The format of the specified point is defined by the writer.

```
"Roll Forward"="Partial:XXX"
```

- New Component Name

  A writer may want to restore a component to a new name. For example, restoring a database to a different name to restore an individual item; restoring to the same name would please all data We recommend that writers accept a valid logical path and component name as this options' value. This will often be used with a *directed target*.

  ```
  "New Component Name"="Logical Path\Component Name"
  ```

VSS Metadata

3/5/2021 • 2 minutes to read • Edit Online

Both writers and requesters maintain information necessary to a backup or restore operation in their own metadata documents.

These documents, the *Writer Metadata Document* and the *Backup Components Document*, respectively, are used as the basis for writer and requester communication and coordination during backup and restore activities.

The metadata is represented in XML format using a proprietary schema. Metadata can be copied to disk, tape, or any other mass storage device. It should always be saved to the backup media during a backup operation, and will need to be retrieved from that media in the course of a restore operation.

Caution: The specific details of the format and schema are for system use only. Developers should not attempt to modify or directly use the XML representation of the metadata.

Writers and requesters are provided with a variety of query and set methods to access and modify the Backup Components and Writer Metadata documents:

- Working with the Writer Metadata Document
- Working with the Backup Components Document
- VSS Metadata Components

Working with the Writer Metadata Document

3/5/2021 • 2 minutes to read • Edit Online

Each writer produces a Writer Metadata Document. The backup application uses this document to get information about that writer, the data it owns, and how to restore that data. Once the writer produces it, the Writer Metadata Document is a read-only document to the backup application.

To use the Writer Metadata Document requires that you understand the following:

- Writer Metadata Document Life Cycle
- Writer Metadata Document Contents

Writer Metadata Document Life Cycle

3/5/2021 • 2 minutes to read • Edit Online

In response to an *Identify event*, each writer present on the system constructs its own Writer Metadata Document using **IVssCreateWriterMetadata**. An *Identify event* is typically generated by a requester calling **IVssBackupComponents::GatherWriterMetadata**.

When creating a Writer Metadata Document, either through the **IVssCreateWriterMetadata** interface or through writer initialization (**CVssWriter::Initialize**), a writer must explicitly specify the following:

- *Restore method*
- Writer name
- *Writer class ID*
- Data usage (see **VSS_USAGE_TYPE**)
- Date source type (see **VSS_SOURCE_TYPE**)

In addition, it may also specify the following:

- Components (which may or may not contain file sets)
- Add alternate mappings
- Exclude file lists

An overview of Writer Metadata Document creation is found in Writer Actions during Backup Initialization.

Requesters typically make use of one of two methods to obtain access to writer metadata:

- During most backup operations, a requester uses **IVssBackupComponents::GetWriterMetadata** to obtain an instance of the **IVssExamineWriterMetadata** interface to allow access to the metadata of a currently executing writer.
- For restore operations, or backups using imported shadow copies (see Importing Transportable Shadow Copied Volumes for more information on importing shadow copies), a requester retrieves an XML document containing the metadata and uses **CreateVssExamineWriterMetadata** to obtain an **IVssExamineWriterMetadata** interface, which it uses to read the restore metadata.

Writer Metadata Documents enable the requester performing a backup to learn about currently executing writers during the discovery phase of a backup.

For those writers chosen to participate in a backup, a requester imports much, but not all, of the information in the Writer Metadata Document into its own Backup Components Document during the discovery phase of a backup.

However, only Writer Metadata Documents and not the Backup Components Document contain the file and path specifications.

For more information on how the discovery phase of a backup operation is conducted, see Overview of the Backup Discovery Phase.

In addition, only *explicitly included* components have their information stored in the Backup Components Document during a backup operation. Information on *implicitly included* components is not included in the Backup Components Document during a backup operation, and must be interpolated using information about the explicitly included components and the available Writer Metadata Documents.

Implicitly included components may still be *selectable for restore* and may need to be explicitly included in the

Backup Components Document at restore time. In this case, just as adding a component during a backup operation required access to the component's writer's Writer Metadata Document (then retrieved from the writer), a requester will require access to a copy of that writer's Writer Metadata Documents stored at backup time.

Therefore, the only way to obtain all the information about all the files and components involved in a backup or restore is to have each Writer Metadata Document for each writer participating in a backup stored along with the Backup Components Document. (For more information, see Overview of Actual File Restoration.)

Writer Metadata Document Contents

3/5/2021 • 4 minutes to read • Edit Online

The Writer Metadata Document contains three sets of data: writer identification and classification information, writer-level specifications, and component data.

Writer Identification Information

The writer identification and classification information includes the following:

- Writer name
- *Writer class ID*
- *Writer instance*
- How the data managed by the writer is used on the host system (see **VSS_USAGE_TYPE**)
- The type of data managed by the writer (see **VSS_SOURCE_TYPE**)

With the exception of the writer instance, which is unique and is generated by the system when a **CVssWriter** object is initialized, all these values are set by a writer when it calls **CVssWriter::Initialize** and are available to a requester by calling **IVssExamineWriterMetadata::GetIdentity**.

Because the writer instance is uniquely generated, a stored writer instance retrieved from a stored Writer Metadata Document is not likely to be useful.

By checking **VSS_USAGE_TYPE**, an application can determine if a writer is managing general application data, or if the files it works with are part of the system's boot state or are used by a system service. Backup and restore applications need to respect usage types to help maintain system stability.

The **VSS_SOURCE_TYPE** flag indicates what type of application the writer managing the data to be backed up performs during normal operation.

Currently, the distinction is limited to specifying whether the writer produces files as part of transactional or nontransactional database operations, or if the files are the result of a more general type of activity. This list may grow over time. This information can be useful in determining the ordinary level of activity expected in the files of a writer.

Writer-Level Specification

Writer-level specifications contain information that is writer wide in its scope, applying to all data independent of which one component manages it.

A writer must always specify *restore methods*.

It may optionally specify the following:

- Exclude file list
- *Alternate location mappings* for restore

The include and exclude file lists contain file information beyond that in the components, and their specification supersedes component specification.

Restore Method Specification

The *restore method* is set in the Writer Metadata Document by

IVssCreateWriterMetadata::SetRestoreMethod and retrieved by a requester with **IVssExamineWriterMetadata::GetRestoreMethod**.

In setting a restore method, a writer indicates the preferred manner of file restoration, also known as the original restore target, for all files managed by a writer. For instance, the restore method specifies if all files managed by a writer should be allowed to overwrite files currently on disk. (See VSS Restore Configurations and **VSS_RESTOREMETHOD_ENUM** for more information.)

Exclude File List Specification

The exclude list allows fine-tuning of wildcard specifications in components by explicitly preventing certain files from being included in a backup set.

For instance, a component might have a *file set* containing a file specification of c:\Database*.*. While this is a convenient definition, there may occasionally be temporary files generated (perhaps of the form *.tmp), and the writer always wants to prevent their backup.

In this case, a writer would add *.tmp to its exclude list using **IVssCreateWriterMetadata::AddExcludeFiles**. This specification could be recursive.

A requester would query this information by using **IVssExamineWriterMetadata::GetExcludeFile**.

The exclude file list takes precedence over component files lists.

Thus, the list of files specified for backup in a Writer Metadata Document would consist of all the files specified in the *explicitly included* components and the *implicitly included* components, less all excluded files.

Alternate Location Mappings Specification

Alternate location mappings are initially set during the creation of a Writer Metadata Document and indicate a location on disk to which files can be restored if restoration of a file to the original location is not possible.

The information is added as a null-terminated wide character string with **IVssCreateWriterMetadata::AddAlternateLocationMapping** and retrieved as an **IVssWMFiledesc** object by **IVssExamineWriterMetadata::GetAlternateLocationMapping**.

Despite the fact that alternate location mappings are specified and examined using the writer-level interfaces (**IVssCreateWriterMetadata** and **IVssExamineWriterMetadata**), they are specified in terms of *file sets*. The file set used in specifying an alternate location mapping (path, file specification, and recursion flag) must match one of the file sets already added to one of the writer's components (see Adding Files to Components).

For more information, see Non-Default Backup and Restore Locations.

Component-Level Information

Components are collections of files that form a logical unit for purposes of backup and restore. All files in a component (except those explicitly excluded) must be backed up and restored as a unit.

Writers add components using **IVssCreateWriterMetadata::AddComponent**, specifying the following component information:

- Type
- Name
- Logical path (if any)
- Supported feature
- *Selectability*
- Metadata to be used by the writer during restore

- Display information
- Notification information

Selectability for backup and *selectability for restore* are completely independent of each other, and a writer uses them in conjunction with logical paths to indicate relationships between the various components it manages. Writers can indicate which components are required for *explicitly included* (those that may be explicitly included at the discretion of a requester), and those that can only be *implicitly included*. (See Working with Selectability and Logical Paths.)

Files are added to a given component using either **IVssCreateWriterMetadata::AddFilesToFileGroup**, **IVssCreateWriterMetadata::AddDatabaseFiles**, or **IVssCreateWriterMetadata::AddDatabaseLogFiles**. (See Adding Files To Components.)

When adding files to a component during backup, a writer must specify a file set (a path, file specification, and recursion flag) that defines the files to be backed up.

Writers can also specify an *alternate path* for backup, which should not be confused with *alternate location mappings* mentioned previously. This alternate path indicates a non-default location from which files are to be copied when a volume is backed up.

Information about a given component in the Writer Metadata Document can be obtained through an **IVssWMComponent** interface returned by **IVssExamineWriterMetadata::GetComponent**.

The files and paths are returned in **IVssWMComponent** as **IVssWMFiledesc** objects.

A writer's component information is discussed in detail in Definition of Components by Writers.

Working with the Backup Components Document

3/5/2021 • 2 minutes to read • Edit Online

A requester creates a Backup Components Document at the start of performing a backup. The document initially contains only a description of the state of the backup operation. During restore, the document should provide instructions on how a requester should proceed in copying files back to disk. In the course of the restore operation, the Backup Components Document is modified and contains the state of that operation.

Unlike the Writer Metadata Document, which is read-only, there is a window in which the Backup Components Document can be modified by both requesters and writers. The document can be updated until the generation of a *BackupComplete* or *BackupShutdown* event during backup operations, and until a *PostRestore* event during restores.

To use a requester's Backup Components Document requires an understanding of the following topics:

- Backup Components Document Life Cycle
- Backup Components Document Contents

Backup Components Document Life Cycle

3/5/2021 • 3 minutes to read • Edit Online

Requesters have primary responsibility for the life cycle of a Backup Components Document.

This control is exercised by an instance of the **IVssBackupComponents** interface object returned by **CreateVssBackupComponents**.

A requester must initialize a Backup Components Document prior to a backup or restore by calling **IVssBackupComponents::InitializeForBackup** or **IVssBackupComponents::InitializeForRestore**. The requester can initialize the document as empty, or it can load a previously stored copy of the document.

For backup operations, a Backup Components Document is typically initialized as empty. Its data will be filled in with cooperation from the system's writers in the course of processing the backup.

For restore operations, a Backup Components Document is typically initialized from a stored document generated during the initial backup. This allows the restore (in conjunction with examination of stored Writer Metadata Documents) to determine what data was initially backed up and how it should be restored.

Backing up *transportable shadow copies* is an exception to this rule. In this case, a shadow copy could have been moved from one system (where it was created along with the initial Backup Components Document) to another by means of reassigning a shared storage device's logical unit. To back up under these circumstances, a requester loads the stored backup state and proceeds from where the initial system left off. (For more information, see Importing Transportable Shadow Copied Volumes.)

In the course of processing a backup, the requester decides which components to actually copy on the basis of which components are marked as *selectable for backup*, the component's *logical paths*, and its own internal logic.

Some of the components will be *explicitly included* in the backup operation; information about the component will be added to the Backup Components Document. Others will be *implicitly included* in the backup; information about the added components will not be added to the Backup Components Document.

All of a writer's nonselectable for backup components without a selectable ancestor in their logical path, and those selectable for backup components the requester chooses, will be added explicitly.

Both nonselectable and selectable for backup components can be added implicitly if they have a selectable ancestor in their logical path, which is explicitly included in the backup. These components (*subcomponents*) are members of *component sets* defined by their selectable ancestor.

When handling restore operations, the requester uses *selectability for restore* instead of selectability for backup in conjunction with logical path information and its own internal logic to decide which files to restore.

If a component that had been implicitly added to the backup is now to be explicitly added to the restore, the requester will update the Backup Components Document with that component's information.

Information about the stored components is available both to requesters and writers through instances of the **IVssComponent** interface.

It is through **IVssComponent** interfaces that writers can query and (until the end of **PostSnapshot** and **PostRestore** events) modify information in the Backup Components Document.

When the **CVssWriter::OnPrepareBackup**, **CVssWriter::OnPreRestore**, **CVssWriter::OnPostSnapshot**, **CVssWriter::OnBackupComplete**, or **CVssWriter::OnPostRestore** event handler is called, a writer receives an instance of an **IVssWriterComponents** interface.

Note that upon generation of the **BackupComplete** event, the Backup Components Document is made read-only, and therefore **CVssWriter::OnBackupComplete** cannot use the **IVssComponent** interface to modify it.

From the **IVSSWriterComponents** interface, the writer can retrieve instances of the **IVssComponent** interface that will allow it to access all of its components explicitly added to the Backup Components Document and to alter their state. For more information, see Overview of Processing a Backup Under VSS and Overview of Processing a Restore Under VSS.

Backup Components Documents are removed from memory when the **IVssBackupComponents** interface is released, and must be stored using **IVssBackupComponents::SaveAsXML**, or all their information will be lost.

In addition, when an **IVssBackupComponents** document is properly released, a **BackupShutdown** event is generated and *auto-release shadow copies* are deleted.

Backup Components Document Contents

3/5/2021 • 5 minutes to read • Edit Online

The Backup Components Document is maintained by instances of the **IVssBackupComponents** interface. This interface also contains numerous methods for controlling backup operations, manipulating shadow copies, and querying the system state. However, not all of the document's information is directly accessible through this interface.

The Backup Components Document consists of several sets of data:

- Information about which components were explicitly included in a backup or restore operation
- A representation of the stored component and writer information
- State information about the backup/recover operation

While the component information is available to both the requester and the writer, only the writer has access to the state information.

Component Inclusion Information

The Backup Components Document contains a list of those components explicitly included in backup and restore by the requester. The list contains the following:

- Explicitly included *selectable components*.

  The inclusion of these files in backup operations is indicated by **IVssBackupComponents::AddComponent** and in restore operations by **IVssBackupComponents::SetSelectedForRestore**.

- Nonselectable for backup subcomponents without a selectable for backup component ancestor.

  All of these components must be included if any components of the writer are to be included in the operation. The inclusion of these files in backup operations is indicated by **IVssBackupComponents::AddComponent** and in restore operations by **IVssBackupComponents::SetSelectedForRestore**.

- Components implicitly added to the backup (*subcomponents*) that are *selectable for restore* and are explicitly added to the restore.

  These components may be either selectable or nonselectable, but have a selectable ancestor that was used to implicitly select them for backup. They are added to the Backup Components Document by **IVssBackupComponents::AddRestoreSubcomponent**.

The identities of components implicitly included in the restore are not stored in the Backup Components Document.

VSS has access to information about component inclusion: writers with no components explicitly included in a restore or backup receive no VSS events following the generation of the *PrepareForBackup* or *PreRestore* events.

Writers cannot directly query this information. This is not a significant limitation because by design, individual VSS writers should not require detailed information about the status of other writers on the system and, as noted above, those with no included components will not have to participate in the VSS operation.

A requester can determine which components have been explicitly included in an operation.

The IVssBackupComponents::GetWriterComponentsCount method returns the number of writers with component information stored in the Backup Components Document (and not the number of components in the document).

The requester indexes through the stored writer information using IVssBackupComponents::GetWriterComponents, which returns instances of the IVssWriterComponentsExt interface. The **IVssWriterComponentsExt** interface allows the requester to obtain the *writer class* and *writer instance* of participating writers, as well as to access information about those of its components stored in the Backup Components Document.

Information on Included Components

The Backup Components Document's representation of the component data (which does not include path and file specification information), which is accessed through instances of the IVssComponent interface.

Requesters and writers obtain access to instances of the IVssComponent interface in different ways.

A requester examines component data on a writer by writer basis by using instances of the IVssWriterComponentsExt interface returned by IVssBackupComponents::GetWriterComponents.

In addition to the writer identification information, the IVssWriterComponentsExt interface provides an array of instances of the IVssComponent interface—one for each of the selected writers included components.

As noted in Backup Components Document Life Cycle, the writers can gain access to the same information through the IVssWriterComponents interface when handling the PrepareForBackup, PrepareForSnapshot, PostSnapshot, BackupComplete, PreRestore, or PostRestore event.

IVssComponent allows both writer and requesters to get the following information:

- A component's name, type, and *logical path* (GetComponentName, GetComponentType, GetLogicalPath)
- How a component should be restored as indicated by the *restore target* (IVssComponent::GetRestoreTarget)
- If an alternate location was used in restoring a file (GetAlternateLocationMapping, GetAlternateLocationMappingCount)
- New target information, if any (GetNewTarget, GetNewTargetCount)
- Pre-and post-restore error messages (GetPreRestoreFailureMsg, GetPostRestoreFailureMsg)
- If a *selectable for backup* component defining a component set has been selected for restore (IsSelectedForRestore)
- Whether a backup or restore succeeded (GetBackupSucceeded, GetFileRestoreStatus)
- Any writer-specific backup or restore options that may have been set by IVssBackupComponents::SetBackupOptions or IVssBackupComponents::SetRestoreOptions (GetBackupOptions, GetRestoreOptions)
- Any writer-specific metadata backup or restore metadata (GetBackupMetadata), GetRestoreMetadata)
- Time-stamp information (GetBackupStamp, GetPreviousBackupStamp)
- Information about backup subcomponents explicitly included in a restore (GetRestoreSubcomponent, GetRestoreSubcomponentCount)

Unlike requesters, writers can change certain information in the Backup Components Document via the IVssComponent interface:

- How a component should be restored as indicated by the restore target (IVssComponent::SetRestoreTarget)
- Writer-specific backup and restore metadata (IVssComponent::SetBackupMetadata, IVssComponent::SetRestoreMetadata)

- Time-stamp information (SetBackupStamp)
- Pre- and post-restore error messages (SetPreRestoreFailureMsg, SetPostRestoreFailureMsg)

Requester State Information

Requesters insert information about the state of a backup or restore operation into the Backup Components Document using the IVssBackupComponents interface. Writer applications are able to query for this information through the CVssWriter class.

State information stored in the Backup Components Document includes the following:

General Information about the Backup

- The overall backup type (incremental, differential, or full)

Set by requesters using IVssBackupComponents::SetBackupState

Retrieved by writers using CVssWriter::GetBackupType

- Whether component operations are supported

Set by requesters using IVssBackupComponents::SetBackupState

Retrieved by writers using CVssWriter::AreComponentsSelected

- Whether the bootable system state is backed up

Set by requesters using IVssBackupComponents::SetBackupState

Retrieved by writers using CVssWriter::IsBootableStateBackedUp

- Whether partial file operations are supported

Set by requesters using IVssBackupComponents::SetBackupState

Retrieved by writers using CVssWriter::IsPartialFileSupportEnabled

General Information about the Restore

- The overall restore type (whether restore is by copy or import)

Set by requesters using IVssBackupComponents::SetRestoreState

Retrieved by writers using CVssWriter::GetRestoreType

Information about Supporting Files

- The location of ranges files used by a specific component in partial file operations

Set by requesters using IVssBackupComponents::SetRangesFilePath

Retrieved by writers (or requesters) using IVssComponent::GetPartialFile

Status of Information

- Indicate whether one of a given writer's components was successfully backed up

Set by requesters using IVssBackupComponents::SetBackupSucceeded

Retrieved by writers and requesters using IVssComponent::GetBackupSucceeded

- Indicate whether one of a given writer's components was successfully restored

Set by requesters using **IVssBackupComponents::SetFileRestoreStatus**

Retrieved by writers and requester using **IVssComponent::GetFileRestoreStatus**

Writer-Settable Information

- Additional backup specification for one of a given writer's components

Set by writers using **IVssComponent::SetBackupMetadata**

Retrieved by writers and requesters using **IVssComponent::GetBackupMetadata**

- Additional restore specification for one of a given writer's components

Set by writers using **IVssComponent::SetRestoreMetadata**

Retrieved by writers and requesters using **IVssComponent::GetRestoreMetadata**

- A backup stamp that indicates, in a writer's own specific format, the time of the current backup of one of its component's backups

Set by writers using **IVssComponent::SetBackupStamp**

Retrieved by writers and requesters using **IVssComponent::GetBackupStamp**

- A backup stamp that indicates, in a writer's own specific format, the time of the last backup of one of its components' backups using a backup stamp initially set by IVssComponent::SetBackupStamp

Stored and set by requesters for a specific component using **IVssBackupComponents::SetPreviousBackupStamp**

Retrieved by writers and requesters using **IVssComponent::GetPreviousBackupStamp**

- Error messages for failure before and after restore operations

Set by writers using **IVssComponent::SetPreRestoreFailureMsg** or **IVssComponent::SetPostRestoreFailureMsg**

Retrieved by writers and requesters using **IVssComponent::GetPreRestoreFailureMsg** or **IVssComponent::GetPostRestoreFailureMsg**

VSS Metadata Components

3/5/2021 • 2 minutes to read • Edit Online

Critical to organizing which files of which writer are to be backed up or restored is the concept of a *component*.

Components allow a writer to indicate to a backup engine how its files are to be organized, dependencies between files, and what type of data those files can contain. This allows a backup engine to decide how to store files for maximum efficiency.

In addition, VSS-based applications use components as the building blocks for their metadata and the medium for writer/requester communication.

Writers and requesters store information about components separately—in the Writer Metadata Document and the Backup Components Document, respectively—and the information differs in each representation.

Component information in Writer Metadata Documents includes the following:

- Information from only one writer in each document
- All of that writer's components, whether they can be *explicitly included* or must be *implicitly included* in a backup or restore operation
- *Logical path* information used to associate a selectable for backup component with particular nonselectable for backup components, thus forming a component set
- The *file set* information—path, file specification, and recursion flag—managed for each component

Writer Metadata Documents also contain writer-level metadata information, such as restore methods and alternate location mappings for restore. Writer Metadata Documents are read-only. The interface for examining this information is **IVssWMComponent**.

Component information in Backup Components Documents includes the following:

- Only information on explicitly included components
- Writer-level metadata information, such as alternate location mappings and restore
- State information describing a backup or restore operation

Backup Component Documents do not contain information on components' *file sets*. Backup Component Documents are not read-only and can be modified by the writer. The interface for accessing this information is **IVssComponent**.

The life cycle and relationship between the two expressions of a component can be understood as follows:

- Writers are responsible for the initial definitions of components.
- A requester examines the metadata of all writers and their components.
- From components' selectability and logical path information, a requester determines which components must be explicitly included, which may be explicitly included, which define component sets, and which are members of component sets.
- A requester adds those components that require explicit inclusion, and implicitly includes subcomponents in *component sets* (whose information is not in the Backup Components Document).
- When handling events, writers and requesters may modify and examine the component information stored in the Backup Components Document to coordinate their activity.

Both the writer and the requester versions component information are required to properly execute backup and restore operations, and both must be stored with any backed-up data:

Component Types

3/5/2021 • 2 minutes to read • Edit Online

Components indicate the sort of data they represent through a type.

Currently, component types (see **VSS_COMPONENT_TYPE**) are limited to the following:

- Database components
- File groups

For implementation information about setting component types, see Definition of Components by Writers.

Writers have a data typing that indicates their usage (see **VSS_SOURCE_TYPE**), which can be the following:

- A transactional database (such as an SQL server)
- A nontransactional database (such as a spreadsheet client)
- File group (other)

Specifying a component type as database allows for easier identification of its content, allows for separate handling of log and data files (see **IVssCreateWriterMetadata** and **IVssExamineWriterMetadata** for details), and enforces greater rigor in file selection by not allowing either recursive file selection or using an *alternate path* (see **IVssCreateWriterMetadata::AddDatabaseFiles** and **IVssCreateWriterMetadata::AddDatabaseLogFiles**).

With a file group component, on the other hand, at the price of not knowing what data it contains, you have greater freedom about how files are inserted, because you can use recursive specification and alternate paths.

Additional component types may be added in the future.

Definition of Components by Writers

3/5/2021 • 6 minutes to read • Edit Online

Components are defined by and instantiated by writers in their Writer Metadata Document in response to an *Identify event* at the start of a backup operation (see Overview of Backup Initialization) when the Writer Metadata Document is populated.

When creating a component in its Writer Metadata Document, using **IVssCreateWriterMetadata** and **IVssCreateWriterMetadata::AddComponent**, a writer must specify:

- Whether the component is *selectable for backup*
- A component type
- A component name (which must be unique not only within a given *writer instance* but across all writer instances)
- Whether the component has any writer-specific metadata associated with it
- Whether the writer requires notification following a successful backup

Writers can optionally specify:

- A component's *logical path* (which must be unique not only within a given writer instance but across all writer instances)
- A component description (or caption)
- An icon to be used with GUIs to indicate the component

It is not necessary for a component to actually contain any files. This sort of empty or "dummy" component can be useful in organizing components. Such a component can be used to define a *component set* and a writer's component (see Logical Pathing of Components).

Setting Up Component Organization

Setting a component's *selectability* (its *selectability for backup*, and its *selectability for restore*) and its *logical paths* allows a writer to mandate or make optional the inclusion of certain components in a backup or restore operation, and to group components into *component sets* with one selectable component acting as an entry point to the whole group.

Membership in these groupings determines which components will be used during backup and restore operations. Using "selectable" to mean selectable for back for backup operation, and selectable for restore for restore operation, developers should understand that:

- If any components managed by a given writer are backed up, then a requester must *explicitly include* all nonselectable *components* without selectable ancestors in their *logical path* to the Backup Components Document and back up and restore those components as a group.
- A requester has the option of explicitly adding selectable components to the Backup Components Document during backup and restore operations; once added, the component must be backed up or restored.
- If a component is selectable, the component and all of its *subcomponents* (as defined by logical paths) form a component set, which can be treated as a single unit that may optionally participate in backup and restore operations.
- A requester never explicitly adds a nonselectable component with selectable ancestors, a subcomponent in a component set, to its Backup Components Document during backup and restore operations. These components must be *implicitly included* if their selectable ancestor is explicitly added, in which case they

must be backed up or restored (see Use of Components by the Requester).

- A selectable component with a selectable ancestor is still a *subcomponent* (a member of a component set) and may be implicitly included if its selectable ancestor is explicitly included in the operation. In this case, its information is not added to the Backup Components Document. If its selectable ancestor is not included in the operation, the component can be explicitly selected for inclusion in the operation, in which case its information is included in the Backup Components Document.
- A subcomponent implicitly included in a backup can be explicitly included in a restore operation, regardless of the status of any selectable ancestor, if it is selectable for restore. Any selectable for restore subcomponent included during a restore operation must have its information added to the Backup Components Document.
- A writer that has had no component explicitly added to the Backup Components Document prior to the generation of *PrepareForBackup* and *PreRestore* events will not receive any further VSS events.

For more information, see Working with Selectability and Logical Paths.

Adding Files to a Component

A component contains file information in the form of a *file set* that contains:

- A root directory of the files in the component.
- A file specification for the files in the component.
- A flag that indicates whether the component's specification is recursive.

Depending on component type, which can be a database or a file group, and (in the case of database components) whether the files to be loaded are data or log files, a writer calls **IVssCreateWriterMetadata::AddFilesToFileGroup**, **IVssCreateWriterMetadata::AddDatabaseFiles**, or **IVssCreateWriterMetadata::AddDatabaseLogFiles** to add a file set.

When using these functions, you should specify the files to be added to the file set as follows:

- *wszPath*: This is the path to the directory that contains the files to be added to the file set. If the *bRecursive* parameter is set to **true**, the *wszPath* parameter specifies a hierarchy of directories to be traversed recursively, and all directories should be recreated, including empty directories.
- *wszFilespec*: This string specifies the files in each directory to be added to the file set.

For example, suppose the following directory structure exists:

C:\\Directory1\\File1.txt C:\\Directory1\\File2.txt C:\\Directory1\\Directory2\\File1.txt C:\\Directory1\\Directory2\\File2.txt C:\\Directory1\\Directory3\\

If the writer specifies "C:\Directory1" for *wszPath*, "File1.*" for *wszFilespec*, and **true** for *bRecursive*, the requester should include these files:

C:\\Directory1\\File1.txt C:\\Directory1\\Directory2\\File1.txt

If the writer instead specifies "C:\Directory1" for *wszPath*, "*" for *wszFilespec*, and **true** for *bRecursive*, the requester should include these files:

C:\\Directory1\\File1.txt C:\\Directory1\\File2.txt C:\\Directory1\\Directory2\\File1.txt C:\\Directory1\\Directory2\\File2.txt

If the writer specifies "C:\Directory1" for *wszPath*, "*" for *wszFilespec*, and **false** for *bRecursive*, the requester should include these files:

C:\\Directory1\\File1.txt C:\\Directory1\\File2.txt

In all of the preceding examples, whenever the writer specifies **true** for *bRecursive*, the empty directory C:\Directory1\Directory3\ should be recreated.

For a file set added to a file group component, in cases where files currently on disk are not in what the writer would consider the appropriate or default location, a writer has the option of adding an alternate path. In these cases, the file set's definition of the path contains the normal location of the files and to where files should be restored, while the alternate path contains the current location of files to be backed up.

All files in the file set must exist at the time of the backup. Requesters must assume that all files listed in the file set are required for the backup and will fail the backup if any files are missing. Note that when "*" is specified for the *wszFilespec* parameter, it can match zero or more files.

Note that such Writer Metadata Document attributes as alternate location mappings, explicitly included and excluded files, and restore methods are set at the writer level, not the component level. (For more information, see Working with the Writer Metadata Document.)

Component Definition for Backup and Restore Operations

Both restore and backup operations necessarily generate an *Identify event*, and for both backups and restores it will be handled by the same **CVssWriter::OnIdentify** method.

During backup operations, requesters use the information returned by a writer's **CVssWriter::OnIdentify** methods to determine which writers are present on the system and then to determine which of their files to back up.

During restore operations, the information returned by a writer's **CVssWriter::OnIdentify** event is used only to establish the identity and status of writers currently present on the system; the file specification information generated during a restore is not used. Instead, Writer Metadata Documents stored at backup time are used to obtain this data.

Once generated during a backup operation, the writer component information, along with the rest of the writer information, is saved to be retrieved to support restore operations. It is typically the responsibility of the requester to store this information.

Use of Components by the Requester

3/5/2021 • 5 minutes to read • Edit Online

In addition to performing a backup or restore, and supervising shadow copies, a requester must manage information about the components of the writers it interacts with. Component selectability and logical path are used to include or exclude data from a backup, and to decide what component information is included in the Backup Components Document.

Requester Component Selection during Backup

During backup operations, a requester imports writer metadata component data using the **IVssBackupComponents::GatherWriterMetadata** and **IVssBackupComponents::GetWriterMetadata** methods (see Overview of Backup Initialization for more information).

After examining writer information with the **IVssExamineWriterMetadata** interface, a requester decides which writers it will back up and, to a limited extent, which of a given writer's components it will back up.

When backing up a writer, a requester:

- Must *explicitly include* all of a writer's nonselectable for backup components without selectable for backup ancestors using **IVssBackupComponents::AddComponent** to add the component to the Backup Components Document
- May explicitly include any of a writer's selectable for backup components using **IVssBackupComponents::AddComponent** to add the component to the Backup Components Document
- If a selectable for backup component defines a *component set*, its explicit inclusion *implicitly includes* all the component set's members—whether selectable for backup or not. These components are not added to the Backup Components Document.

In adding a selectable for backup component or a nonselectable for backup components without selectable for backup ancestors to its Backup Components Document, a requester specifies the following:

- The instance of the writer managing the component
- The writer's class identifier
- The *logical path* of the component (which may be **NULL**)
- The component's name

If a component does not match the specification, an error will be returned.

If such a component exists, VSS creates an **IVssComponent** interface for the component in the Backup Components Document. This information will be accessible and modifiable by the writer and requester. For a selectable component that defines a *component set*, it describes not only properties of the component but also all the subcomponents it contains.

Information about implicitly added components is not available in the Backup Components Document. In addition, no file information is available in the Backup Components Document. To obtain that information, the requester will have to examine the Writer Metadata Documents (which will have already been read) in the context of the selected stored components in the Backup Components Document.

Requester Component Selection during Restore

During restore operations, a requester should not import component information from the writers currently active on the system via **IVssBackupComponents::GatherWriterMetadata**, because the state of currently

executing processes will not necessarily reflect the state of processes when a backup was made.

It should still generate an *Identify event* via **IVssBackupComponents::GatherWriterMetadata**, both to create an *Identify event* and to determine which writers are currently on the system and their status.

The requester retrieves the stored Backup Components Document during its initialization as well as stored Writer Metadata Documents (see Overview of Restore Initialization for more information) .

Inclusion of components during backup is largely the same as that for restore, except that you must consider *selectable for restore* along with *logical path*—not *selectable for backup*.

There are, however, some differences:

- If a component has already been *explicitly included* to the Backup Components Document during backup, if it is included for restore (either explicitly or implicitly), **IVssBackupComponents::SetSelectedForRestore** is used to explicitly add it to the Backup Components Document for restore.
- If a component was *implicitly included* to the backup, and is nonselectable for restore with no selectable for restore ancestors—which in the backup case would imply the need for explicit inclusion—the component is not explicitly included (that is, it is not added to the Backup Components Document using **IVssBackupComponents::SetSelectedForRestore**). Such a component should be considered implicitly selected for restore.
- Of those components implicitly selected for backup (whether that component was selectable for backup or not), only those that are selectable for restore can be added to the Backup Components Document using **IVssBackupComponents::AddRestoreSubcomponent**.
- Selectable for restore components may define a *component set* for restore—just as selectable for backup components do. This selectable for restore component then defines this component set for the restore operation.

A writer with no components explicitly selected for restore prior to the generation of a *PreRestore* event will not receive any VSS events.

Requesters and writers can access stored component information using the **IVssComponent** interface. Through the **IVssComponent** interface, writers can modify some of the settings of those of its components explicitly included in the Backup Components Document to support a restore (such as the *restore target*). If it defines a component set, writer modifications of an explicitly included component will propagate to its *subcomponents*. In addition, the interface provides a mechanism for passing information about restore success and failure between writer and requester.

As during backup, there is insufficient information in the Backup Components Document itself to implement the restore. Again, the Writer Metadata Documents will be required to supply information about the actual paths of files to be restored and to discover what nonselectable components are part of the selectable components component set and therefore need to be restored.

See Working with Selectability and Logical Paths for information about the types of selectability and their usage.

Use of Writer Component Document Information by the Requester

Each component is uniquely identified by the *Writer Class ID* of its parent writer, its name, and its *logical path*.

The requester can use the **IVssWriterComponentsExt** interface returned by the **IVssBackupComponents::GetWriterComponents** method to obtain information about each stored component.

The component's name and logical path (among other items) can be found via the **IVssComponent** interface returned by **IVssWriterComponentsExt::GetComponent**.

NOTE

During the restore phase, the requester should call **IVssWriterComponentsExt::GetComponent** or **IVssWriterComponentsExt::GetComponentCount** only after the call to **IVssBackupComponents::PreRestore** has returned, to allow time for the writer to update the Backup Components Document. One example of such an update would be to change the restore target.

Information about each stored selectable component's parent writer can be found using **IVssWriterComponentsExt::GetWriterInfo**.

With this information, the Writer Metadata Documents can be queried and the matching document identified. Then, by using the *logical path*, the requester can identify the dependent nonselectable components for each selectable component—that is, identify all members of the selectable component's *component set*.

Using the **IVssExamineWriterMetadata** interface, the requester now has full component information—including path specification (from the **IVssWMComponent** interface)—for both selectable and nonselectable components it needs to back up or restore.

This is one reason why it is vital for a requester to save both the state of its own Backup Components Document and the Writer Metadata Documents of the writers it is backing up.

See Working with Selectability and Logical Paths for more detailed information.

Working with Selectability and Logical Paths

3/5/2021 • 2 minutes to read • Edit Online

A writer's participation in a backup or restore operation, and which of its data are saved, depends on which of its components are *explicitly included* as part of a requester's Backup Components Document and the relationship between those components and the logical path of other components within the Writer Metadata Document.

Writers with no components added to a requester's Backup Component Document prior to the generation of a *PrepareForBackup* event (in the case of backup operations) or of a **PreRestore** event (in the case of restore operations) receive no events after this point and will not participate in the backup or restore operation.

However, a requester's freedom to include or exclude any given component in a backup or restore is governed by the following:

- Any hierarchy that exists between the components managed by a writer and expressed by *logical paths*
- Whether the component is designated as being *selectable for backup*
- Whether it is designated as being *selectable for restore*
- Whether an explicit dependency exists between a given component in a given writer and other components in other writers

More information on these issues is in the following topics:

- Logical Pathing of Components
- Working with Selectability for Backup
- Working with Selectability For Restore and Subcomponents
- Selectability and Working with Component Properties
- Dependencies between Components Managed by Different Writers

Logical Pathing of Components

3/5/2021 • 3 minutes to read • Edit Online

Logical pathing is used to organize components managed by a writer into well-defined groups.

The logical path is analogous in structure to traditional file pathing, using the backslash "\" to separate elements in the path. Unlike file paths, the root of a logical path is **NULL**, instead of "\".

The logical path is expressed as a **NULL**-terminated string, and there are no other restrictions on the characters the path can contain.

The most important use of logical pathing is in defining *component sets*, where the *explicit component inclusion* in a backup or restore operation of one selectable component requires the inclusion of a number of other components (*subcomponent*). The logical path of the defining component of a component set is a parent to the logical paths of its subcomponents and:

- Subcomponents must share as a root path the logical path of the selectable component that defines the component set.
- A root path of **NULL** is valid.
- The name of the defining selectable component must be the first logical path element after the root path for every nonselectable subcomponent of the component set.
- Component sets can be nested.
- The combination of logical path and component name must be unique across all instances of a *writer class*.

The hypothetical example of a writer *MyWriter* with a logical path structure defined below illustrates logical pathing.

COMPONENT NAME	LOGICAL PATH	SELECTABLE FOR BACKUP
"Executables"	""	N
"ConfigFiles"	"Executables"	N
"LicenseInfo"	""	Y
"Security"	""	Y
"UserInfo"	"Security"	N
"Certificates"	"Security"	N
"writerData"	""	Y
"Set1"	"writerData"	N
"Jan"	"writerData\Set1"	N
"Dec"	"writerData\Set1"	N
"Set2"	"writerData"	N

COMPONENT NAME	LOGICAL PATH	SELECTABLE FOR BACKUP
"Jan"	"writerData\Set2"	N
"Dec"	"writerData\Set2"	N
"Query"	"writerData\QueryLogs"	N
"Usage"	"writerData"	Y
"Jan"	"writerData\Usage"	N
"Dec"	"writerData\Usage"	N

Note that the components "Executables" and "ConfigFile" have a parent-child relationship, but both are nonselectable. Therefore, they do not form a component set. Whenever the writer *MyWriter* is backed up or restored, these two components will have to be *explicitly included* in the operation.

The component "LicenseInfo" is selectable with neither ancestor nor descendant. It can be explicitly included, or not, in a backup or restore operation at the discretion of the requester.

The component "Security" defines a simple component set, containing no component sets beneath it.

The component "writerData" defines a component set, which contains a complex collection of components with several well-defined logical path hierarchies beneath it.

One subcomponent, "Usage", is selectable and defines a component set.

Several components have the same name and are distinguished by their logical paths. Instances of the nonselectable components "Dec" and "Jan" exist under the components nonselectable components "Set1" and "Set2" and under the selectable subcomponent "Usage".

If the component "writerData" is explicitly included in a backup or restore, then all of its subcomponents—even those in the nested component set defined by "Usage"—will be *implicitly includedimplicitly included* in the operation.

If the component set defined by "writerData" is not explicitly included in a backup or restore, components "Set1", "Set2", and "QueryLogs" (and their instances of subcomponents "Dec" and "Jan") will not be included implicitly in the backup or restore operation.

However, even if "writerData" is not included in the operation, the component "Usage" is still selectable, and it can still be explicitly included in a backup or restore operation. If it is, then its subcomponents "Jan" and "Dec" will be implicitly included.

Other points worthy of note:

- The selectable components "LicenseInfo" and "writerData" and the nonselectable component "Executables" are all at the same level in *MyWriter*'s logical path hierarchy: all have the same logical path of **NULL** or "", the root logical path.
- The selectable component "Usage" should never be explicitly included in a backup, if its selectable parent ("writerData") is explicitly included in a backup or restore operation.
- Components that define component sets may exist simply for organizational reasons. For instance, either the "writerData" or the "Usage" component, or both, might be empty; that is, no *file sets* were added to them using the **IVssCreateWriterMetadata::AddFilesToFileGroup**,

IVssCreateWriterMetadata::AddDatabaseFiles or **IVssCreateWriterMetadata::AddDatabaseLogFiles** method. The components still define a component set.

Working with Selectability for Backup

3/5/2021 • 3 minutes to read • Edit Online

The following table describes the four types of components that can be involved with a backup operation.

COMPONENT TYPE	DESCRIPTION
Nonselectable-for-backup components	No selectable-for-backup ancestors in their logical paths.
Selectable-for-backup components	No selectable-for-backup ancestors in their logical paths.
Nonselectable-for-backup subcomponents	Nonselectable-for-backup components with selectable-for-backup ancestors in their path.
Selectable-for-backup subcomponents	Selectable-for-backup components with selectable-for-backup ancestors in their path.

In addition, any selectable-for-backup component—regardless of whether it has selectable-for-backup ancestors or not—defines a *component set* if other components have it as an ancestor in their logical paths.

The rules governing the selection of components for backup can be summarized as follows:

- When any component without a selectable-for-backup ancestor in its logical path—whether the component is selectable-for-backup or nonselectable-for-backup—is included in a backup, it must be *included explicitly*. This means that metadata for these components is added to the Backup Components Document.

  Requesters explicitly add these components using the **IVssBackupComponents::AddComponent** method.

- Nonselectable-for-backup subcomponents are always *included implicitly* in the backup. This means that metadata for these components is not part of the Backup Components Document.

- Selectable-for-backup subcomponents are implicitly included if that ancestor is explicitly included in the backup. In this case, metadata for these components is not added to the Backup Components Document. If an implicitly selectable for backup subcomponent defines a component set, the members of that component set are also implicitly selected.

- Selectable-for-backup subcomponents whose selectable-for-backup ancestor is not explicitly included in the backup can still be included explicitly by the requester using the **IVssBackupComponents::AddComponent** method. The metadata for the component will then be added to the Backup Components Document. In addition, if a selectable-for-backup subcomponent defines a component set, the members of that component set are implicitly included in the backup.

The "MyWriter" case discussed in Logical Pathing of Components can be used as an example to illustrate selectability for backup.

COMPONENT NAME	LOGICAL PATH	SELECTABLE FOR BACKUP
"Executables"	""	N

COMPONENT NAME	LOGICAL PATH	SELECTABLE FOR BACKUP
"ConfigFiles"	"Executables"	N
"LicenseInfo"	""	Y
"Security"	""	Y
"UserInfo"	"Security"	N
"Certificates"	"Security"	N
"writerData"	""	Y
"Set1"	"writerData"	N
"Jan"	"writerData\Set1"	N
"Dec"	"writerData\Set1"	N
"Set2"	"writerData"	N
"Jan"	"writerData\Set2"	N
"Dec"	"writerData\Set2"	N
"Query"	"writerData\QueryLogs"	N
"Usage"	"writerData"	Y
"Jan"	"writerData\Usage"	N
"Dec"	"writerData\Usage"	N

Whenever "MyWriter" is backed up, explicitly including the "Executables" component using the **IVssBackupComponents::AddComponent** method will implicitly include the "ConfigFiles" component.

The component "LicenseInfo" is a stand-alone selectable-for-backup component. It may be selected using the **IVssBackupComponents::AddComponent** method at the discretion of the requester, but its selection will select no other components.

The selectable-for-backup component "Security" defines a simple component set containing two nonselectable-for-backup subcomponents, "UserInfo" and "Certificates". If "Security" is explicitly included for backup, then "UserInfo" and "Certificates" are always implicitly included as well. There is no way to include the subcomponents "UserInfo" or "Certificates" in a backup operation unless "Security" is included.

If the component "writerData" is selected, then the nonselectable-for-backup components "Set1", "Set2", and "Query" as well as the selectable-for-backup component "Usage" are implicitly selected. Each of these components has subcomponents that are implicitly selected for backup. None of their metadata will be added to the Backup Components Document.

If the component "writerData" is not selected, the nonselectable-for-backup components "Set1", "Set2", and "Query" are not included for backup.

However, requesters may choose to explicitly include the selectable for backup component "Usage". Metadata for this component will be added to the Backup Components Document. "Usage"'s subcomponents "Jan" and "Dec" will be implicitly added to the backup, but will not have their information added to the Backup Components Document.

Explicitly including a component for backup will create a corresponding **IVssComponent** instance in the Backup Components Document.

A requester will retrieve information about explicitly included components from its Backup Components Document by examining those writers (using **IVssBackupComponents::GetWriterComponents**) included in its document and retrieving the stored **IVssComponent** objects.

As neither the file set information (file specification, path, and recursion flag) of the components present in the Backup Components Document, nor any information about implicitly added components will be present, requesters will have to query Writer Metadata Documents to obtain full information about all components included in the Backup Components Document.

Working with Selectability For Restore and Subcomponents

3/5/2021 • 2 minutes to read • Edit Online

Selectability for restore allows the requester to determine when a component can be individually restored. A component that has been included for backup can appear in one of two ways:

- A component may have been *explicitly included* in the backup. These components have a corresponding **IVssComponent** instance in the Backup Components Document. These components are included in a restore using **IVssBackupComponents::SetSelectedForRestore**.
- A component may have been *implicitly included* in the backup. These components do not have a corresponding **IVssComponent** instance in the Backup Components Document; however, there will always be an **IVssComponent** instance for some ancestor component in the document. These components are included in a restore using **IVssBackupComponents::AddRestoreSubcomponent**.

Any component that has been explicitly included in the backup can always be individually selected for restore, regardless of its selectability-for-restore value. The requester calls **IVssBackupComponents::SetSelectedForRestore**, passing in the writer ID, logical path, and name of the specific component. Components that have been implicitly included in the backup will be restored when an explicitly included ancestor is restored. Implicitly included components can be individually selected for restore only if they are marked as selectable for restore. The requester first calls **IVssBackupComponents::SetSelectedForRestore** on the closest explicitly included ancestor component, and then calls **IVssBackupComponents::AddRestoreSubcomponent** on the ancestor component to select the implicitly included component for restore. After this is done, only the implicitly selected component will be restored; all other components in the component set will not be restored.

Unlike selectability for backup, which must always be explicitly set when a component is added with **IVssCreateWriterMetadata::AddComponent**, selectability for restore has a default value of false, which can be overridden.

Because top-level components (components with an empty logical path) can only be explicitly included in a backup, selectability for restore has no meaning for these components.

Selectability and Working with Component Properties

3/5/2021 • 6 minutes to read • Edit Online

Working with implicitly selected components requires access to both the Backup Components Document and Writer Metadata Documents.

There are two reasons for this:

- The component data stored in the Backup Components Document (represented by the **IVssComponent** interface) lacks access to component file set information—file specification, path, and recursion flag. (See Working with the Backup Components Document.)
- Only components that are *included explicitly* in the Backup Components Document during backup have their information directly stored in the Backup Components Document. Requesters and writers must use the information available through the **IVssComponent** interface, in conjunction with *logical path* information and Writer Metadata Documents to obtain information about, and set properties of, *implicitly included* components.

The "MyWriter" case discussed in Logical Pathing of Components can be used to illustrate selectability for backup.

COMPONENT NAME	LOGICAL PATH	SELECTABLE FOR BACKUP	SELECTABLE FOR RESTORE	EXPLICITLY INCLUDED
"Executables"	""	N	N	Y
"ConfigFiles"	"Executables"	N	N	Y
"LicenseInfo"	""	Y	N	Y
"Security"	""	Y	N	Y
"UserInfo"	"Security"	N	N	N
"Certificates"	"Security"	N	N	N
"writerData"	""	Y	Y	Y
"Set1"	"writerData"	N	Y	N
"Jan"	"writerData\Set1"	N	N	N
"Dec"	"writerData\Set1"	N	N	N
"Set2"	"writerData"	N	Y	N
"Jan"	"writerData\Set2"	N	N	N
"Dec"	"writerData\Set2"	N	N	N

COMPONENT NAME	LOGICAL PATH	SELECTABLE FOR BACKUP	SELECTABLE FOR RESTORE	EXPLICITLY INCLUDED
"Query"	"writerData\QueryLogs"	N	N	N
"Usage"	"writerData"	Y	Y	N
"Jan"	"writerData\Usage"	N	N	N
"Dec"	"writerData\Usage"	N	N	N

Implicitly Included Components in the Backup Set

While examining a writer's Writer Metadata Document (see **IVssBackupComponents::GetWriterMetadata**) during backup, a requester should be storing a list of all components, their *logical paths*, and their file set information.

File set and excluded file information will be needed to determine a list of files for any (explicitly or implicitly) included component.

For nonselectable for backup components with no selectable for backup ancestors and selectable for backup components that do not define a *component set*, only file set and excluded file information will be needed to identify all the component's candidates for backup, because these components do not define subcomponents.

For explicitly included selectable for backup components that define a component set, the file sets and exclude file information both for the defining component and all *subcomponents* needs to be used to select files for backup.

This means that backup sets for the components "Executables", "ConfigFiles", and "LicenseInfo" can be found only by examining the writer metadata for just these components using their instances of the **IVssWMComponent** interface.

However, if writerData is explicitly included in the backup, you must examine its instance of the **IVssWMComponent** interface and those for "Set1", "Jan" (with logical path "writerData\Set1"), "Dec" (with logical path "writerData\Set1"), "Set2", "Jan" (with logical path "writerData\Set2"), "Dec" (with logical path "writerData\Set2"), "Query", "Usage", "Jan" (with logical path "writerData\Usage"), and "Dec" (with logical path "writerData\Usage").

To do this, a requester will have to first identify that the component "writerData" (logical path "") is selectable. It will then have to scan all the other components managed by the writer to determine whether the first element in their logical path is "writerData". Those components that have "writerData" as the leading members of their logical path are identified as being subcomponents of "writerData" and are implicitly selected when it is explicitly selected.

In fact, a similar scan will need to be made to determine that no component has "LicenseInfo" as the leading member of its logical path, and thus that "LicenseInfo" has no subcomponents.

Because of the complexity of this mechanism in VSS, many requester writers may find it useful to create their own structures for storing component and backup set information for both explicitly and implicitly added components.

Properties of Implicitly Included Components

During restore and backup operations, instances of the IVssComponent and IVssBackupComponents interfaces are used both to retrieve information about components, and to set or change component properties. However, only components explicitly included will have instances of the **IVssComponent** interface or be accessible to the **IVssBackupComponents** interface.

Some properties are component-set-wide in scope. These properties include the following:

- Backup and restore status:

IVssBackupComponents::SetBackupSucceeded
IVssComponent::GetBackupSucceeded
IVssBackupComponents::SetFileRestoreStatus
IVssComponent::GetFileRestoreStatus

- Backup and restore options:

IVssBackupComponents::SetBackupOptions
IVssComponent::GetBackupOptions
IVssBackupComponents::SetRestoreOptions
IVssComponent::GetRestoreOptions

- Failure messages:

IVssComponent::SetPostRestoreFailureMsg
IVssComponent::SetPreRestoreFailureMsg
IVssComponent::SetPostRestoreFailureMsg
IVssComponent::SetPreRestoreFailureMsg

- Restore targets:

IVssComponent::SetRestoreTarget
IVssComponent::GetRestoreTarget

- Backup stamps:

IVssComponent::SetBackupStamp
IVssComponent::GetBackupStamp

- Additional metadata:

IVssComponent::SetRestoreMetadata
IVssComponent::GetRestoreMetadata
IVssComponent::SetBackupMetadata
IVssComponent::GetBackupMetadata

Therefore, you use the instance of the IVssComponent interface of a component set's defining member or use the defining member's name, type, and logical path with an IVssBackupComponents method to set or retrieve properties for all the component set's members.

For this reason, component sets are treated as units. For instance, a backup of a component set is successful only if the backup of all of the file sets of all of its components is successful.

In the preceding example, suppose one file in the component "Jan" (with logical path "writerData\Set2") is a member of the component set defined by "writerData". If one of "Jan"'s files failed to back up, a requester would use "writerData"'s information (its name "writerData", its path "", and its component type) as arguments when setting IVssBackupComponents::SetBackupSucceeded with **false** to indicate the component set's failure.

Similarly, the state returned by IVssComponent::GetBackupSucceeded for "writerData"'s instance of the IVssComponent interface applies not just to "writerData" but to all its subcomponents as well.

In addition, if a writer chose to change the restore target using **IVssComponent::SetRestoreTarget** of "writerData"'s instance of **IVssComponent**, that would change the restore target for all the file sets of all of "writerData"'s subcomponents.

The following properties apply not component-wide, but to particular files or file sets:

- Alternate location mappings:

IVssBackupComponents::AddAlternativeLocationMapping
IVssComponent::GetAlternateLocationMapping
IVssComponent::GetAlternateLocationMappingCount

- Differenced files:

IVssComponent::AddDifferencedFilesByLastModifyTime
IVssComponent::GetDifferencedFile
IVssComponent::GetDifferencedFilesCount

- Partial files:

IVssComponent::AddPartialFile
IVssComponent::GetPartialFile
IVssComponent::GetPartialFileCount

- Directed targets:

IVssComponent::AddDirectedTarget
IVssComponent::GetDirectedTarget
IVssComponent::GetDirectedTargetCount

- New targets:

IVssBackupComponents::AddNewTarget
IVssComponent::GetNewTarget
IVssComponent::GetNewTargetCount

When a requester accesses these features for a subcomponent using the **IVssBackupComponents** interface, it uses the component information for the component set's defining component, but the file or file set information for the subcomponent.

Likewise, if the property is accessible through the **IVssComponent** interface, the instance corresponding to the defining subcomponent is used, but the file or file set arguments are taken from the subcomponent.

For instance, suppose the subcomponent "Jan" (with logical path "writerData\Set2") had a file set with a path of "c:\fred", a file specification of "*.dat", and a recursive flag of **true** might have to be restored to an alternate location.

If this was the case, a requester would call **IVssBackupComponents::AddAlternativeLocationMapping**, using "writerData"'s information (component type, a component name of "writeData", and a logical path of "") along with "Jan"'s file set information (path "c:\fred", file specification "*.dat", and recursion equals **true**).

Note that in this case the file set information must exactly match the file set information used by **IVssCreateWriterMetadata::AddFilesToFileGroup**, **IVssCreateWriterMetadata::AddDatabaseFiles**, or **IVssCreateWriterMetadata::AddDatabaseLogFiles** to add files to Jan.

Similarly, if a writer wanted to add a directed target to a file with a path of "c:\ethel" and name "lucy.dat" managed by "Jan" (with logical path "writerData\Set2"), it would use the **IVssComponent** instance corresponding to "writerData", but "Jan"'s file information.

Implicitly Included Components in the Restore Set

Components that were implicitly included in a backup can be explicitly included in a restore if they are selectable for restore. As noted in Working with Selectability for Restore and Subcomponents, such components are added to the Backup Components Document using the **IVssBackupComponents::AddRestoreSubcomponent** method.

However, this does not create a new instance of the **IVssComponent** interface, nor is the component directly accessible through the **IVssBackupComponents** interface.

Instead, a component explicitly included for restore, but implicitly included for backup, must be accessed through an instance of the **IVssComponent** interface corresponding to the component that defined the component set of which it was a member upon backup.

For example, to explicitly include for restore "Set1", a subcomponent of the selectable for backup component "writerData", you would obtain information about it by calling the **IVssComponent::GetRestoreSubcomponent** method of "writerData"'s instance of the **IVssComponent** interface.

Dependencies between Components Managed by Different Writers

3/5/2021 • 8 minutes to read • Edit Online

There are situations where data from one writer depends on data managed by another writer. In these cases, you should back up or restore data from both writers.

VSS handles this problem through the notion of an explicit writer-component dependency and the **IVssWMDependency** interface.

A writer adds one or more dependencies while creating the Writer Metadata Document using the **IVssCreateWriterMetadata::AddComponentDependency** method. The writer passes the method the name and logical path of the dependent component (which it manages), as well as the name and logical path and the *writer class ID* (the GUID identifying the class) of the component upon which it depends.

Once established, this dependency informs the requester that during any backup or restore operation both the dependent component and the targets of its dependencies must participate.

A given component can have multiple dependencies, which requires that it and all its dependent targets participate in the backup and restore together.

The dependent component and/or the target(s) of its dependencies can be included either *explicitly* or *implicitly* in a backup or restore operations.

The explicit writer component dependency mechanism should not be used to create a dependency between two components on the same writer. The selection rules can supply the same functionality more efficiently without risk of circular dependencies.

As an example, **IVssCreateWriterMetadata::AddComponentDependency** could be used to define the dependency of the component writerData (with logical path "") of the writer MyWriter on component internetData (with a logical path of "Connections") of a writer named InternetConnector with a writer Class ID X. (While it is possible for multiple writers with the same class ID to be on the system simultaneously, confusion will be avoided because the combination of logical path and component name is unique on the system under VSS.)

A writer adds multiple dependencies to a given component simply by calling **IVssCreateWriterMetadata::AddComponentDependency** repeated with different components from different writers. The number of other components a given component depends on can be found by examining the **cDependencies** member of the **VSS_COMPONENTINFO** structure.

A writer or requester retrieves instances of the **IVssWMDependency** interface with **IVssWMComponent::GetDependency**. The **IVssWMDependency** returns the component name, logical path, and class ID of the writer managing the component that is the target of the dependency.

The dependency mechanism does not prescribe any particular order of preference between the dependent component and the targets of its dependencies. As noted above, all a dependency indicates is that whenever a given component is either backed up or restored, the component or components it depends on must be backed up or restored as well. The exact implementation of dependency is at the discretion of the backup application.

For example, in the case above, the component writerData (logical path "") depends on component InternetConnector (logical path "Connections"). A requester is free to interpret this in either of the following ways:

- If the dependent component, writerData, is selected (implicitly or explicitly) for backup or restore, the requester must select (either implicitly or explicitly) the target of its dependency, internetData
- If the target of its dependency, internetData, is not selected for backup, then the dependent component, writerData, should not be selected.

However, when developing support for dependencies, requester developers should be aware that there is no way a writer can determine if one of its components is a target of a dependency.

Declaring Remote Dependencies

A distributed application is an application that can be configured to use one or more computers at a time. Typically the application runs on one or more application server computers and communicates with (but may or may not actually run on) one or more database server computers. This configuration is sometimes referred to as a multi-system deployment. Often the same application can also be configured to run on a single computer that runs both an application server and a database server. Such a configuration is called a single-system deployment. In both configurations, the application server and database server each have their own independent VSS writers.

In a multi-system deployment, if a component managed by the application's writer depends on a remote component managed by the database server's writer, this is called a remote dependency. (A single-system deployment, in contrast, has only local dependencies.)

As an example of a multi-system deployment, consider an application server that uses a SQL Server database server as a data store. The application-specific data, which includes the web parts, web content files, and the IIS metabase, resides on one or more computers, called front-end web servers. The actual SQL data store, which includes the Config database and multiple Content databases, resides on one or more other computers, called back-end database servers. Each of the front-end web servers contains the same application-specific content and configuration. Each of the back-end database servers can host any of the Content databases or the Config database. The application software runs only on the front-end web servers, not on the database servers. In this configuration, the application's VSS writer has remote dependencies on the components managed by the SQL writer.

A writer can declare a remote dependency by calling the **AddComponentDependency** method, prepending "*RemoteComputerName*\", where *RemoteComputerName* is the name of the computer where the remote component resides, to the logical path in the *wszOnLogicalPath* parameter. The value of *RemoteComputerName* can be an IP address or a computer name returned by the **GetComputerNameEx** function.

Windows Server 2003: A writer cannot declare remote dependencies until Windows Server 2003 with Service Pack 1 (SP1).

To identify a dependency, a requester calls the **GetWriterId**, **GetLogicalPath**, and **GetComponentName** methods of the **IVssWMDependency** interface. The requester must examine the component name that **GetComponentName** returns in the *pbstrComponentName* parameter. If the component name begins with "\\", the requester must assume that it specifies a remote dependency and that the first component following "\\" is the *RemoteComputerName* that was specified when the writer called **AddComponentDependency**. If the component name does not begin with "\\", the requester should assume that it specifies a local dependency.

If there is a remote dependency, the requester must back up the remote component when it backs up the local component. To back up the remote component, the requester should have an agent on the remote computer and should initiate the backup on the remote computer.

Structuring Remote Dependencies

It is important to understand that a dependency is not a component in and of itself. A component is necessary to hold the dependency.

The following examples show two ways to structure a set of dependencies.

```
Example 1:
    Writer 1
        Component A
            File A
            File B
            Dependency (SQL/MSDE Writer, Component X, "\")
            Dependency (SQL/MSDE Writer, Component Y, "\")

Example 2:
    Writer 2
        Component A
            File A
            File B
        Component B
            Dependency (SQL/MSDE Writer, Component X, "\")
            Dependency (SQL/MSDE Writer, Component Y, "\")
```

In example 1, the dependencies are held by Component A. Because only components can be selected, not individual files, structuring Component A's dependencies this way would require that the entire component, both the files and the dependencies, must always be backed up and restored together. They cannot be backed up or restored individually.

In example 2, separate components (Components A and B) hold each of the dependencies. In this case, the two components can be selected independently, and therefore backed up and restored independently. Structuring the dependencies this way gives a distributed application much more flexibility in managing its remote dependencies.

Supporting Remote Dependencies

A requester can provide full or partial support for remote dependencies.

To provide full support, the requester should do the following at backup and restore time.

At backup time, the requester must start the backup on the front-end (local) computer, determine the dependencies that exist, and spool additional backup jobs to capture the back-end databases. The requester must wait until after the back-end backup jobs on the remote computer have completed before calling the **IVssBackupComponents::SetBackupSucceeded** and **IVssBackupComponents::BackupComplete** methods. If the requester waits until the backup of back-end components is complete before calling **BackupComplete**, this will produce a more easily recoverable backup for a writer that implements additional enhancements—such as topology locking, for example—during the backup. The following procedure outlines what the requester should do:

1. On the local computer, the requester calls the **IVssBackupComponents::InitializeForBackup**, **IVssBackupComponents::GatherWriterMetadata**, **IVssBackupComponents::PrepareForBackup**, and **IVssBackupComponents::DoSnapshotSet** methods.
2. After the local shadow copy is completed, but before the backup is completed, the requester spools additional backup jobs by sending a request to its agent on the remote computer.
3. On the remote computer, the requester's agent performs the spooled backup sequence by calling **InitializeForBackup**, **GatherWriterMetadata**, **PrepareForBackup**, **DoSnapshotSet**, **SetBackupSucceeded**, and **BackupComplete**.
4. When the requester's agent has completed the spooled jobs on the remote computer, the requester completes the backup sequence by calling **SetBackupSucceeded** and **BackupComplete**.

At restore time, the requester must start the restore involving the front-end (local) computer, select the components and their dependencies to be restored, and then send the **PreRestore** event by calling the

IVssBackupComponents::PreRestore method. The requester should then spool the back-end restore jobs on the remote computer and call the **IVssBackupComponents::PostRestore** method when the back-end restores are complete. This requirement gives the front-end writer more control over the restore experience and a better administrator user experience overall. Because the backups across each of the systems do not occur at the same point in time, the front-end writer will need to perform some cleanup of the back-end data. In the example application discussed in the preceding "Declaring Remote Dependencies", the writer should initiate a site remapping or reindexing after a restore of one of the back-end databases has completed. To do this, the writer must receive events on the front-end server. The following procedure outlines what the requester should do:

1. On the local computer, the requester calls **IVssBackupComponents::InitializeForRestore**, **GatherWriterMetadata**, **IVssBackupComponents::SetSelectedForRestore** (or **IVssBackupComponentsEx::SetSelectedForRestoreEx**), and **PreRestore**.
2. After the **PreRestore** phase is completed, but before the **PostRestore** phase begins, the requester spools additional restore jobs by sending a request to its agent on the remote computer.
3. On the remote computer, the requester's agent performs the spooled restore jobs by calling **InitializeForRestore**, **GatherWriterMetadata**, **SetSelectedForRestore**, **PreRestore**, **SetFileRestoreStatus** (or **SetSelectedForRestoreEx**), and **PostRestore**.
4. When the requester's agent has completed the spooled jobs on the remote computer, the requester completes the restore sequence by calling **IVssBackupComponents::SetFileRestoreStatus** and **PostRestore**.

To provide partial support for remote dependencies, the requester must follow remote dependencies and include them as part of the backup, but the ordering of events on front-end and back-end systems as detailed in the previous two procedures would not be required. For a requester that implements only partial support, the requester should refer to the writer application's backup/restore documentation to understand what scenarios can be supported.

Overview of Processing a Backup Under VSS

3/5/2021 • 2 minutes to read • Edit Online

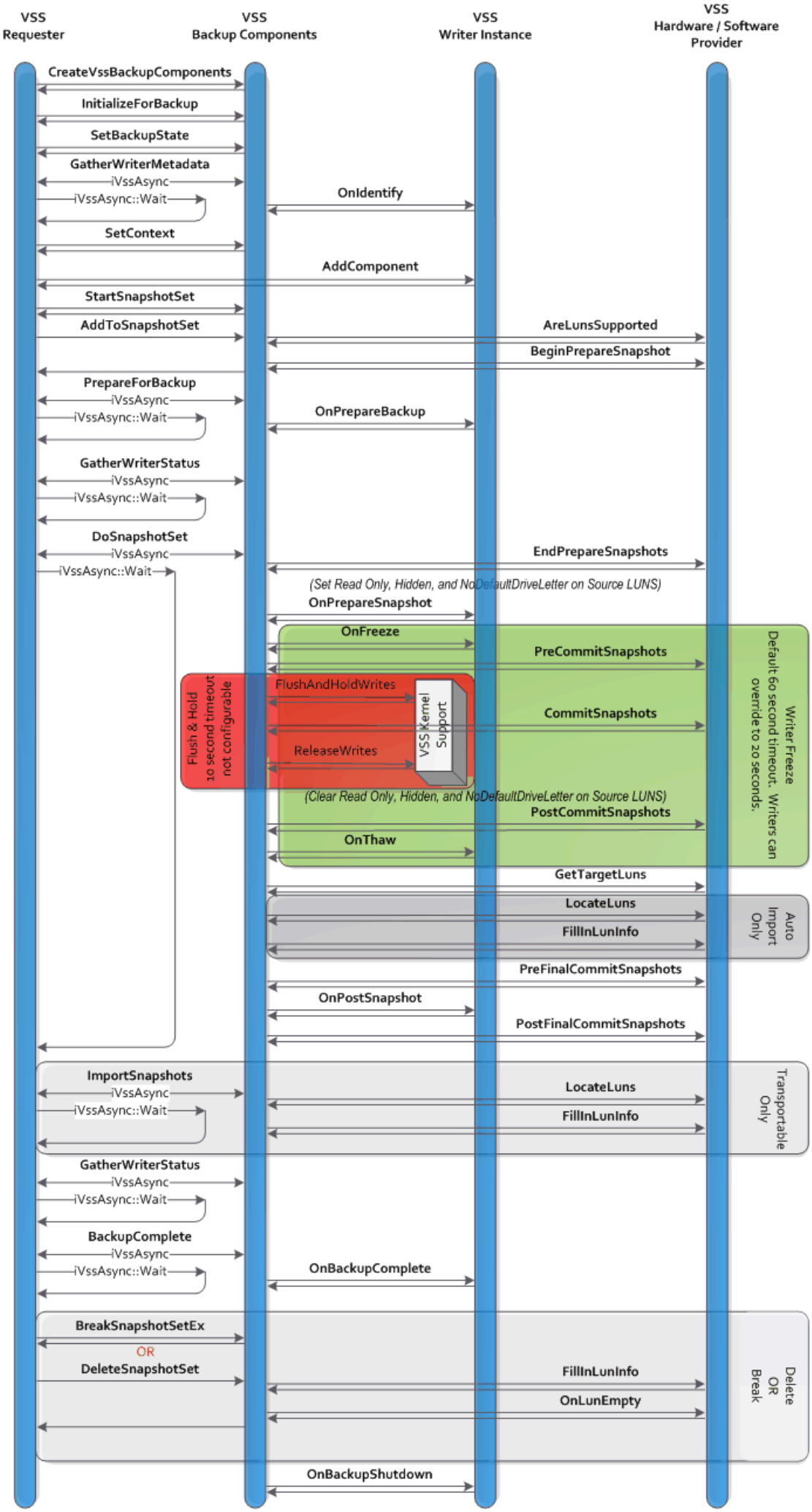
In processing a backup, requester and writers coordinate to provide a stable system image from which to back up data (the shadow copied volume), to group files together on the basis of their usage, and to store information on the saved data. This must all be done while creating only minimal interruption to the writer's normal work flow.

A requester queries writers for their metadata, processes this data, notifies the writers prior to the beginning of the shadow copy and of the backup operations, and then notifies the writers again after the shadow copy and backup operations end.

In response to these notifications, the writer provides information about files to be backed up—including specifying groups of files to coordinate (*components*)—pauses in its I/O operations prior to a shadow copy, and then returns to normal operation following the completion of a shadow copy or at the end of the backup.

In the course of processing the backup, a writer specifies the files it is responsible for through its read-only metadata—the Writer Metadata Document (see VSS Metadata: Working with the Writer Metadata Document). The requester then interprets this metadata, chooses what to back up, and stores these decisions in its own metadata object, the Backup Components Document (see VSS Metadata: Working with the Backup Components Document). This Backup Components Document is available for writer inspection and modification during both the backup and restore operations.

This diagram shows the interactions between the requester, the VSS service, the VSS kernel support, any VSS writers involved, and any VSS hardware providers.


VSS Requester
VSS Backup Components
VSS Writer Instance
VSS Hardware / Software Provider
CreateVssBackupComponents
InitializeForBackup
SetBackupState
GatherWriterMetadata
iVssAsync
iVssAsync::Wait
OnIdentify
SetContext
AddComponent
StartSnapshotSet
AddToSnapshotSet
AreLunsSupported
BeginPrepareSnapshot
PrepareForBackup
iVssAsync
iVssAsync::Wait
OnPrepareBackup
GatherWriterStatus
iVssAsync
iVssAsync::Wait
DoSnapshotSet
iVssAsync
iVssAsync::Wait
EndPrepareSnapshots
(Set Read Only, Hidden, and NoDefaultDriveLetter on Source LUNS)
OnPrepareSnapshot
OnFreeze
PreCommitSnapshots
FlushAndHoldWrites
VSS Kernel Support
CommitSnapshots
ReleaseWrites
Flush & Hold 10 second timeout not configurable
(Clear Read Only, Hidden, and NoDefaultDriveLetter on Source LUNS)
PostCommitSnapshots
OnThaw
Writer Freeze Default 60 second timeout. Writers can override to 20 seconds.
GetTargetLuns
LocateLuns
FillInLunInfo
Auto Import Only
PreFinalCommitSnapshots
OnPostSnapshot
PostFinalCommitSnapshots
ImportSnapshots
iVssAsync
iVssAsync::Wait
LocateLuns
FillInLunInfo
Transportable Only
GatherWriterStatus
iVssAsync
iVssAsync::Wait
BackupComplete
iVssAsync
iVssAsync::Wait
OnBackupComplete
BreakSnapshotSetEx
OR
DeleteSnapshotSet
FillInLunInfo
OnLunEmpty
Delete OR Break
OnBackupShutdown

To more fully understand the basic tasks involved in performing a backup, it is useful to break down this overview into the following topics:

- Overview of Backup Initialization
- Overview of the Backup Discovery Phase
- Overview of Pre-Backup Tasks
- Overview of Actual Backup Of Files
- Overview of Backup Termination

Overview of Backup Initialization

3/5/2021 • 5 minutes to read • Edit Online

This stage of the backup initializes both the writer and the requester, filling in their internal data structures, specifying the backup and establishes writer/requester communication through the required call to **IVssBackupComponents::GatherWriterMetadata**. For more information, see Overview of Processing a Backup Under VSS.

The following table shows the sequence of actions and events that are required for backup initialization.

REQUESTER ACTION	EVENT	WRITER ACTION
Creates an **IVssBackupComponents** interface and initializes it to manage a backup (see **CreateVssBackupComponents**, **IVssBackupComponents::InitializeForBackup**) and optionally enable or disable writers on the system.	None	None
Optionally set the context for shadow copy operations and optionally query the system about the providers and shadow copies it supports (see **IVssBackupComponents::SetContext**, **IVssBackupComponents::Query**).	None	None
The requester can provide additional information on handling backup and restore operations (see **IVssBackupComponents::SetBackupState**)	None	None
Initiates asynchronous contact with writers (see **IVssBackupComponents::GatherWriterMetadata**)	*Identify*	Creates a Writer Metadata Document (see Working with the Writer Metadata Document, **CVssWriter::OnIdentify**, **IVssCreateWriterMetadata**)

Requester Actions during Backup Initialization

An **IVssBackupComponents** object can be used for only one backup. Therefore, a requester must proceed through to the end of the backup, including releasing the **IVssBackupComponents** interface. If the backup needs to terminate prematurely, the requester needs to call **IVssBackupComponents::AbortBackup** and then release the **IVssBackupComponents** object (see Aborting VSS Operations for more information). Do not attempt to resume the **IVssBackupComponents** interface.

Typically, a requester's Backup Components Document is initialized as empty. A stored Backup Components Document can be loaded when **IVssBackupComponents::InitializeForBackup** is called, typically in support of transportable shadow copied volumes. In this case, the writer-requester communication will be somewhat different than what is described below. (See Importing Transportable Shadow Copied Volumes for more information.)

To add volumes to the shadow copy set, a requester must first set the context for the shadow copy operation by calling **IVssBackupComponents::SetContext**. If this method is not called, the default context for shadow copies, VSS_CTX_BACKUP, is used. For information about setting the shadow copy context, see Shadow Copy Context Configurations.

To begin the completion of its setup prior to backup, a requester must call **IVssBackupComponents::SetBackupState**. By doing this, a requester indicates to the writers:

- The type of backup (as defined in **VSS_BACKUP_TYPE**)
- Whether the backup includes a bootable system state
- Whether the requester supports the selection of individual components or backs up entire volumes.

All requesters participating in backup and restore operations must always call **IVssBackupComponents::GatherWriterMetadata**. This method initiates writer-requester communication by generating a VSS *Identify* event, in response to which a writer creates its metadata document.

Prior to calling **IVssBackupComponents::GatherWriterMetadata**, a requester has an opportunity to explicitly enable or disable certain specific writers and writer classes using **IVssBackupComponents::EnableWriterClasses**, **IVssBackupComponents::DisableWriterInstances**, and **IVssBackupComponents::DisableWriterClasses** (by default, all classes are enabled). After **IVssBackupComponents::GatherWriterMetadata** is called, these calls have no effect.

Because there is no way to obtain a list of writers on the system prior to calling **IVssBackupComponents::GatherWriterMetadata**, requesters may consider creating and then deleting a second instance of **IVssBackupComponents** to obtain the list.

It is not necessary to call **IVssBackupComponents::GatherWriterStatus** following the completion of **IVssBackupComponents::GatherWriterMetadata**. Writers that fail to process the *Identify* event generated by the calls will not be part of the list of writers providing metadata found by **IVssBackupComponents::GetWriterMetadataCount** and **IVssBackupComponents::GetWriterMetadata** (see Determining Writer Status).

Writer Actions during Backup Initialization

In response to the Identify event, VSS calls each writer's virtual handler method, **CVssWriter::OnIdentify**. A writer creates its Writer Metadata Document by overriding the default implementation of **CVssWriter::OnIdentify** and using the **IVssCreateWriterMetadata** interface.

Note that applications other than the current requester (for instance, system applications) can generate Identify events that must be handled by the writer. In addition, there is no way for a writer to determine from within **CVssWriter::OnIdentify** which application has generated the Identify event.

This being the case, given that a writer may receive several Identify events while processing a backup operation, a writer should never set state information in the **CVssWriter::OnIdentify** handler.

Instead, **CVssWriter::OnIdentify** should perform a consistent algorithm to create the writer's Writer Metadata Document, particularly because after a writer creates the document, neither the requester nor the writer can modify it. From this point forward, it is a read-only document.

This means that the number and type of *components* associated with a writer, which files are part of each component, and the explicit exclusion of files from backup or restore operations cannot be changed after a writer returns from processing the Identify event.

All writers participating with VSS are required to do the following:

1. Indicate a restore method for all components managed by the writer using **IVssCreateWriterMetadata::SetRestoreMethod**.

2. Add at least one component using **IVssCreateWriterMetadata::AddComponent** (see Definition of Components by Writers for more information on component specification).

A writer indicates the files to participate in a backup or restore operation by adding *file sets*—a combination of a path, file specification, and a recursion flag—to a given component using **IVssCreateWriterMetadata::AddFilesToFileGroup**, **IVssCreateWriterMetadata::AddDatabaseFiles**, or **IVssCreateWriterMetadata::AddDatabaseLogFiles**, depending on type (See Adding Files to Components.)

A writer may also have one or more empty components, components to which no files have been added. These are very useful in organizing the writer's components. (See Logical Pathing of Components.)

A writer uses **IVssCreateWriterMetadata::AddExcludeFiles** to explicitly prevent files from being included in the backup. This explicit exclusion is useful because wildcard characters can be used to specify files for inclusion (see Exclude File List Specification). Note that the exclude file list takes precedence over component files lists.

IVssCreateWriterMetadata::AddAlternateLocationMapping is used to create *alternate location mappings* for specified file sets that have been added to one of the writer's components. These mappings are used during file restore when restoring to a file's original location is not possible or desirable. (See Overview of Actual File Restoration and Non-Default Backup and Restore Locations.)

Because the backup file set is specified in the Writer Metadata Document, it cannot later be modified. Therefore, a writer should be coded so that the file set's definition will include all files needed in the backup, either by name or through wildcard characters. Conceivably, this may include some files that might be created after the Identify event.

Overview of the Backup Discovery Phase

3/5/2021 • 4 minutes to read • Edit Online

After calling **IVssBackupComponents::GatherWriterMetadata**, a requester uses the instance of the **IVssAsync** interface returned from this call to determine when all writers on the system have finished constructing their Writer Metadata Documents. For more information, see Overview of Processing a Backup Under VSS.

At this point, the requester can begin a discovery phase, examining metadata to determine what applications are running and which volumes must be shadow copied to get a complete backup. The following table shows the sequence of actions and events that are required for the backup discovery phase.

REQUESTER ACTION	EVENT	WRITER ACTION
Retrieve Writer Metadata Documents (see **IVssBackupComponents::GetWriterMetadata**, **IVssExamineWriterMetadata**).	None	During this period, writers may be able to continue with their normal operations.
Use the list of components and their *file sets*, as well as excluded files, to obtain a list of volumes and files involved in the backup (see **IVssWMComponent**, **IVssWMFiledesc**).	None	None
Choose which components in the writer's Writer Metadata Document to back up. Call **IVssBackupComponents::AddComponent** for each component to add it to the Backup Components Document. (See Working with Selectability for Backup and Working with the Backup Components Document.)	None	None
Initialize the shadow copy set, context and check for supported volumes (see **IVssBackupComponents::StartSnapshotSet**, **IVssBackupComponents::IsVolumeSupported**).	None	None

REQUESTER ACTION	EVENT	WRITER ACTION
If performing a non-component backup, add the desired target volumes from the Writer Metadata Document to the shadow copy set by calling **IVssBackupComponents::AddToSnapshotSet** for each volume. Otherwise, for the components in the Writer Metadata Document that were already added to the Backup Components Document (by calling **AddComponent**), the requester must also call **IVssBackupComponents::AddToSnapshotSet** for each affected volume.	None	None

Writer Actions during the Discovery Phase

Because the discovery phase of a backup consists primarily of a requester processing the information it has retrieved from Writer Metadata Documents, there are few if any requirements on a writer.

In theory, a writer could continue to run normally at this point. However, it may be desirable for writers to begin preparations for the coming shadow copy and backup operations.

Requester Actions during the Discovery Phase

A requester uses the **IVssExamineWriterMetadata** objects obtained through **IVssBackupComponents::GetWriterMetadata** to iterate over all the writers' metadata and selecting those writers whose data it intends to back up.

At this point, the requester will need to generate an initial list of each writer's backup candidates by iterating over the writer's components using **IVssExamineWriterMetadata::GetComponent**. This provides the requester with **IVssWMComponent** objects, from which you can get the specifications for the files to be backed up using **IVssWMComponent::GetFile**, **IVssWMComponent::GetDatabaseFile**, and **IVssWMComponent::GetDatabaseLogFile**.

Because the **IVssWMFiledesc** object can use wildcard characters to hold file location information, it may be necessary to use functions such as **FindFirstFile**, **FindFirstFileEx**, and **FindNextFile**.

Until the shadow copy has been completed, it is still possible for writers to add or remove files from disk in the normal course of their work, so you should not generate the actual list of files to be backed up at this time.

Instead, the initial list of files and volumes to be backed up is found at this point by doing the following:

1. Examining all selectable for backup and nonselectable components in each writer's Writer Metadata Document (using **IVssExamineWriterMetadata**) and organizing them into *component sets* use *logical path* (see Working with Selectability and Logical Paths)
2. *Including explicitly* all required components (nonselectable for backup components without selectable for backup ancestors) in the Backup Components Document using **IVssBackupComponents::AddComponent**
3. Choosing to explicitly include optional selectable for backup components that do not define a component set (using **IVssBackupComponents::AddComponent**)
4. Selecting *component sets* for participation in a backup by explicitly adding their defining selectable for backup component (using **IVssBackupComponents::AddComponent**), which *implicitly includes* the

component set's *subcomponents*.

5. Using *file set* information in the selected writers' Writer Metadata Document and volume management functions, a requester determines the paths of files to be backed up and the volumes that will need to be shadow copied

Note that only the components explicitly included (using **IVssBackupComponents::AddComponent**) in the backup and in the Backup Components Document will have instances of the **IVssComponent** interface added to that document. These instances will be used to examine and modify component settings for both explicitly included components and any of their implicitly included subcomponents (see Selectability and Working with Component Properties).

If a writer includes any of a writer's components, it must add all of its required components. However, a requester is also free to entirely skip all of a writer's component sets. If none of a writer's components are explicitly selected, the writer is not selected, and VSS inhibits that writer from participating in the rest of the backup operation.

The requester initiates the shadow copy set that will contain the selected volumes by calling **IVssBackupComponents::StartSnapshotSet**.

If the volume can participate in a shadow copy (which can be checked with **IVssBackupComponents::IsVolumeSupported**), the requester can add that volume to the shadow copy set using **IVssBackupComponents::AddToSnapshotSet**.

Although it is not generally useful, a requester can sometimes also choose which *provider* will maintain the shadow copy for a given volume (see Selecting Providers for details).

Care should be given to the handling of a volume containing mounted folders or reparse points. A mounted folder or reparse point can appear in a shadow copy and can be backed up. However, a mounted folder or reparse point cannot be traversed on the shadow copied volume (see Working with Mounted Folders and Reparse Points).

At this point in the backup, the Backup Components Document is initialized and filled. In future operations, writers and requesters can use the **IVssComponent** interface to communicate with each other.

Writers are given access to the **IVssComponent** interface when handling the *PrepareForBackup*, *PostSnapshot*, and *BackupComplete* events.

Overview of Pre-Backup Tasks

3/5/2021 • 6 minutes to read • Edit Online

Pre-backup tasks under VSS are focused on creating a shadow copy of the volumes containing data for backup. The backup application will save data from the shadow copy, not the actual volume. For more information, see Overview of Processing a Backup Under VSS.

Requesters typically wait for writers to prepare for backup and creating the shadow copy. The writer must determine if it is to participate in the backup and, if it is, configure its files and itself to be ready for backup and shadow copy. The following table shows the sequence of actions and events that are required to prepare for a backup operation.

REQUESTER ACTION	EVENT	WRITER ACTION
The requester can set backup options (see **IVssBackupComponents::SetBackupOptions**)	None	None
Support incremental and differential backup operations by examining any stored backup stamps (see **IVssComponent::GetBackupStamp**, **IVssBackupComponents::SetPreviousBackupStamp**)	None	None
Notify writers to prepare for a backup operation using **IVssBackupComponents::PrepareForBackup**	*PrepareForBackup*	Writer preparations include determining if files are to be backed up, whether the writer will participate in the shadow copy freeze, as well as creating writer-specific metadata (see **CVssWriter::OnPrepareBackup**, **CVssWriter::IsPathAffected**, **IVssWriterComponents**, **IVssComponent**, **IVssComponent::GetBackupOptions**, **CVssWriter::AreComponentsSelected**, **IVssComponent::SetBackupMetadata**, and **IVssComponent::GetPreviousBackupStamp**.
The requester waits for writers to set up for backup using **IVssAsync**. It should also verify writer status (see **IVssBackupComponents::GatherWriterStatus**, **IVssBackupComponents::GetWriterStatus**)	None	None

REQUESTER ACTION	EVENT	WRITER ACTION
Requester requests a shadow copy using **IVssBackupComponents::DoSnapshotSet**	None	None
None	*PrepareForSnapshot*	**CVssWriter::OnPrepareSnapshot**: Put the writer into a shadow copy ready state.
None	*Freeze*	**CVssWriter::OnFreeze**: Final setup before the shadow copy.
None	*Thaw*	**CVssWriter::OnThaw**: Normal functioning (including I/O) can resume.
None	*PostSnapshot*	**CVssWriter::OnPostSnapshot**: Final clean ups of shadow copy preparations. See **IVssComponent::AddDifferencedFilesByLastModifyTime** and **IVssComponent::SetBackupStamp**.
Requester waits for completion of shadow copy using: **IVssAsync**, it should also verify writer status (see **IVssBackupComponents::GatherWriterStatus**, **IVssBackupComponents::GetWriterStatus**	None	None

Requester Pre-Backup Tasks

In addition, before creating a **IVssBackupComponents::PrepareForBackup** event, a requester may also set backup options for individual writers using **IVssBackupComponents::SetBackupOptions** depending on the specifics of each writer and whether a requester is aware of them.

To support incremental and differential operations, requesters may at this point choose to examine components for time stamps of earlier backup operation (using **IVssComponent::GetBackupStamp**) and use that information to set a previous time stamp for a writer to process (using **IVssBackupComponents::SetPreviousBackupStamp**). See Incremental and Differential Backups for more information.

A requester can now direct the system's writers to complete pre-backup preparations and to handle the creation of a shadow copy.

First, the requester generates a **PrepareForBackup** event by calling **IVssBackupComponents::PrepareForBackup**.

After all participating writers return from handling the **PrepareForBackup** event (which a requester determines by using the instance of the **IVssAsync** interface returned by **PrepareForBackup**), the requester can initiate the shadow copy by calling **IVssBackupComponents::DoSnapshotSet**, which, as it progresses, will generate *PrepareForSnapshot*, *Freeze*, *Thaw*, and *PostSnapshot* events for the writers to handle.

There are some cases where a requester may not need to create a shadow copy. Specifically, each *file set*

managed by one of a given writer's components has a File Specification Backup mask (indicated by a bitwise OR of **VSS_FILE_SPEC_BACKUP_TYPE** values) set during the *Identify* event. This mask specifies, among other things, whether a file set requires the system to be shadow copied before its backup is performed.

If no file sets to be backed up on any volumes require a shadow copy, then **IVssBackupComponents::DoSnapshotSet** need not be called.

Writer Pre-Backup Tasks

When handling the **PrepareForBackup** event, VSS will call each writer's **CVssWriter::OnPrepareBackup** method, a virtual method, which by default simply returns true.

Writers can override this default implementation and use the handling to find information about the upcoming backup and take action.

A writer can determine information about the sort of backup operation contemplated by using the following methods:

1. **CVssWriter::GetBackupType**
2. **CVssWriter::IsBootableStateBackedUp**
3. **CVssWriter::AreComponentsSelected**

A writer determines if the files it manages will be involved in the shadow copy by using **CVssWriter::IsPathAffected**.

More important, when VSS calls the **CVssWriter::OnPrepareBackup** method, it passes in an instance of the **IVssWriterComponents** interface, which allows direct access through the **IVssComponent** interface to those of its components *explicitly included* in the requester's Backup Components Document. The writer used the instances of the **IVssComponent** interface that define component sets to gain access to its *implicitly included* component (see Selectability and working with Component Properties).

During the handling of the **PrepareForBackup** event, writers use the **IVssComponent** interface to perform component-by-component (or component set by component set) operations, including:

1. Adding *partial files* (if supported) by calling **IVssComponent::AddPartialFile**.
2. Setting any private metadata that the writer will need to handle the restore.
3. If the writer supports incremental and differential backups (see Incremental and Differential Backups), doing the following:
   - Checking for previous backup time stamps by calling **IVssComponent::GetPreviousBackupStamp**.
   - Adding any required *differenced files* by calling **IVssComponent::AddDifferencedFilesByLastModifyTime**.
   - If the writer supports the **VSS_BS_TIMESTAMPED** schema, adding backup time-stamp strings in the writer's own format using **IVssComponent::SetBackupStamp**.
4. Initiating very time-consuming asynchronous operations, such as synchronizing data across multiple disks. This will allow the writer to continue working while the operation is processed, including handling other VSS events. These operations must terminate before the *Freeze* event.

The requester's call to **IVssBackupComponents::DoSnapshotSet** initiates the shadow copy and generates the following events for the writers to handle:

- *PrepareForSnapshot*
- *Freeze*
- *Thaw*
- *PostSnapshot*

Three of the writer's handlers—**CVssWriter::OnPrepareSnapshot**, **CVssWriter::OnFreeze**, and **CVssWriter::OnThaw**—are pure virtual methods, and each writer must implement them rather than relying on defaults. Depending on a writer's needs, they may be coded as dummy methods, simply returning **TRUE**.

Because there is typically a narrow time window between the issuing of a *Freeze* event and the issuing of a *Thaw* event, most of the major work in preparing for the shadow copy—such as shutting down processes, creating temporary files, or draining I/O queues—would be handled in **CVssWriter::OnPrepareSnapshot**.

How a writer might use **CVssWriter::OnPrepareSnapshot** to handle its I/O before the creation of a shadow copy is highly dependent on the writer's own architecture.

Writers that can afford to hold all writes and keep the data in an absolute consistent state before *Freeze*, should do so.

If the writer cannot freeze its I/O, then it should take actions to create a stable source for backup and to reduce the recovery time for a shadow copy. Examples of this might include queuing incoming I/O requests or generating a duplicate set of files in an *alternate path* to be used as the source of a backup.

The **CVssWriter::OnFreeze** method performs simple, short tasks such as verifying that the **CVssWriter::OnPrepareSnapshot** left I/O in the correct state, and that any asynchronous tasks started by **CVssWriter::OnPrepareBackup** completed. This method is a writer's last chance to veto a shadow copy if there are problems (see Writer Errors and Vetoes).

It is generally possible for a writer to resume normal operation following a *Thaw* event: a shadow copy may not be immediately ready for backup after the Thaw, but a writer should be able to resume normal operation. Therefore, typically **CVssWriter::OnThaw** is used by writers to return to a pre-freeze state. However, any temporary files created to support the shadow copy should be left in place until the *PostSnapshot* event. Typically, you would use **CVssWriter::OnPostSnapshot** for this sort of cleanup. Because many applications do not require this sort of cleanup, **CVssWriter::OnPostSnapshot** is a virtual method with a default implementation that simply returns **TRUE**. If an incremental or differential backup is being performed, the writer may call **IVssComponent::GetPreviousBackupStamp** and **IVssComponent::SetBackupStamp**. For more information, see Writer Role in Backing Up Complex Stores. Another method that can be called at this time is **IVssComponent::AddDifferencedFilesByLastModifyTime**.

Overview of Actual Backup Of Files

3/5/2021 • 6 minutes to read • Edit Online

VSS enables a requester to access the shadow copy of volumes containing data for backup and to copy data to backup media. Writers generally proceed with normal operation during this process. For more information, see Overview of Processing a Backup Under VSS.

The following table shows the sequence of actions and events that are required for files to be backed up.

REQUESTER ACTION	EVENT	WRITER ACTION
Access files on the shadow-copied volume (see **IVssBackupComponents::GetSnapshotProperties**, **VSS_SNAPSHOT_PROP**)	None	None
Generate the list of files to back up, and copy file data to backup media.	None	None
Indicate the success or failure of the backup with **IVssBackupComponents::SetBackupSucceeded**.	None	None
The requester indicates that the backup has completed by calling **IVssBackupComponents::BackupComplete**.	*BackupComplete*	Perform any post-backup cleanup (see **CVssWriter::OnBackupComplete**, **IVssWriterComponents**, **IVssComponent**).
The requester waits for all writers to acknowledge receipt of the **IVssBackupComponents::BackupComplete** event using **IVssAsync**. It should also verify writer status (see **IVssBackupComponents::GatherWriterStatus**, **IVssBackupComponents::GetWriterStatus**). The requester must call **GatherWriterStatus** at this time to cause the writer session to be set to a completed state. [!Note] This is only necessary on Windows Server 2008 with Service Pack 2 (SP2) and earlier.	None	None

REQUESTER ACTION	EVENT	WRITER ACTION
Save the Backup Components Document and each Writer Metadata Document to XML documents, which can be written to the backup media (see **IVssBackupComponents::SaveAsXML** and **IVssExamineWriterMetadata::SaveAsXML**).	None	None

Writer Actions during Backup of Files

After the shadow copy has been completed, all I/O operations on the system that is being backed up should be able to resume without disrupting the integrity of the backup. This is one of the prime motivations for having the shadow copy functionality.

Therefore, as in the discovery phase (see Overview of the Backup Discovery Phase), there are few demands placed on the writers while files are actually being backed up.

After a backup has completed, and a requester has generated a *BackupComplete* event, VSS will call each writer's **CVssWriter::OnBackupComplete** method, a virtual method that by default simply returns **TRUE**. However, writers can override the default implementation and take such actions as removing remaining temporary files, or use the **IVssWriterComponents** interface it is called with to check the state of the backup of each of its *included explicitly* components (and any *component set* they might define) by retrieving the corresponding **IVssComponent** object. The writer can then determine, and act on, the success or failure of the backup by calling **IVssComponent:GetBackupSucceeded**. The value returned by **IVssComponent:GetBackupSucceeded** will be **TRUE** only if all explicitly included files in the component and all *implicitly included* of any of its *subcomponents* have been successfully backed up.

When the call to **CVssWriter::OnBackupComplete** has completed, the requester should call **IVssBackupComponents::GatherWriterStatus** and **IVssBackupComponents::GetWriterStatus** (for each writer) one last time. The writer session state memory is a limited resource and writers must eventually reuse session states. This step marks the writer's backup session state as completed and notifies VSS that this backup session slot can be reused by a subsequent backup operation.

Requester Actions during Backup of Files

As noted in Overview of the Backup Discovery Phase, you should not generate the actual list of files to be backed up until the shadow copy has completed.

The *device object* corresponding to the shadow copy of a given volume is used to get access to files on the shadow copied volume once the shadow copy has completed. The device object is obtained from the **VSS_SNAPSHOT_PROP** object returned by **IVssBackupComponents::GetSnapshotProperties**. Each shadow copy of a shadow copy set will have its own device object.

The device object and the paths obtained from the Writer Metadata Document specifications of components are then used to select files for backup. See Requester Access to Shadow Copied Data for more information.

Which files will be included in the backup list depends on the nature of the particular backup, upon component *selectability for backup*, and the logical path structure of the writer (see Working with Selectability for Backup).

In addition to files specified in the components, a given writer may also have explicitly excluded files. File exclusion must always be respected, regardless of which components are selected.

Also as noted in Overview of the Backup Discovery Phase, a mounted folder or reparse point can appear in a shadow copy and can be backed up. However, a mounted folder or reparse point cannot be traversed on the shadow-copied volume (see Working with Mounted Folders and Reparse Points).

Care should also be taken during a backup operation, if the *alternate path* returned by **IVssWMFiledesc::GetAlternateLocation** is not blank. An alternate path differs from an *alternate location mapping* in that it is used only during backups, while an alternate location mapping is used only during restores.

In this case, the data is not to be backed up from its normal location (indicated by **IVssWMFiledesc::GetPath**), but from the location returned by **IVssWMFiledesc::GetAlternateLocation**. On restore, the file should be returned to its normal location. See Non-Default Backup and Restore Locations for more information.

VSS places no restrictions on the actual mechanism of backing up data to a storage medium or the choice of that medium. However, it is recommended that the files of each component of each *writer instance* be processed as a unit. See Generating A Backup Set for a discussion of best practices in generating the backup file list.

The success or failure of backing up any of the files managed by a given component and (if it defines a *component set*) its *subcomponents* for a given writer instance should be preserved in the Backup Components Document by calling **IVssBackupComponents::SetBackupSucceeded**. If any file managed by a component or component set fails to back up, the entire component is said to fail. Exact information about which file failed to back up should always be logged.

Developers may find it helpful to store a record on the backup media of which files are backed up, the components and component set they were a member of, their specification, and what their original paths were. It may also be useful to store information such as each writer's component definition. Doing this may make the retrieval operation simpler. However, such details are left to the requester developer.

Because writers can modify the Backup Components Document while handling the **PostSnapshot** event generated by the requester's call to **IVssBackupComponents::DoSnapshotSet**, the Backup Components Document should not be saved until after that asynchronous call has completed.

Although it may take place earlier, this is also a convenient time to save the Writer Metadata Document.

It is very important that both the Backup Components Document and the Writer Metadata Documents be preserved using **IVssBackupComponents::SaveAsXML** and **IVssExamineWriterMetadata::SaveAsXML**. If they are not, it will not be possible to make use of VSS during file restoration.

In addition to storing the original metadata, some backup applications may find it useful to save a copy of their own list of the files (in their own optimized format)—and their associated writer, component, restore procedure, and location information—to the backup media for later retrieval. Such a list can be used to avoid some of the parsing and comparing of Writer Metadata Documents and the Backup Component Documents during restore.

Volumes being backed up may have data that is not managed by VSS writers. This data can and should be backed up from the shadow copied volume, where it will be in a *crash-consistent state*. See Backups without Writer Participation for more information.

Overview of Backup Termination

3/5/2021 • 2 minutes to read • Edit Online

The following table shows the sequence of actions and events that are required for a backup operation to be terminated. For more information, see Overview of Processing a Backup Under VSS.

REQUESTER ACTION	EVENT	WRITER ACTION
The requester terminates the shadow copy by releasing the **IVssBackupComponents** interface or by calling **IVssBackupComponents::DeleteSnapshots**.	None	None
IVssBackupComponents is released by calling **IUnknown::Release**.	*BackupShutdown*	The writer handles the event with **CVssWriter::OnBackupShutdown**, which allows it to clean up any state related to the shadow copy set. If the backup operation failed—that is, it did not generate a **BackupComplete** event—the writer may also have to perform error handling. See Handling BackupShutdown Events for more information.

Because an **IVssBackupComponents** interface cannot be reused, and the destructor of the interface terminates the shadow copies, there is typically no reason to call **IVssBackupComponents::DeleteSnapshots**. This method is designed to be used in conjunction with error handling and aborting backups (see Aborting VSS Operations).

Overview of Processing a Restore Under VSS

3/5/2021 • 2 minutes to read • Edit Online

In processing a backup under VSS, requesters and writers worked together to produce a stable system image from which to back up data, which required coordination and the creation of a shadow copy of the current system.

In contrast to a VSS backup, where the purpose of requester and writers coordination is to produce a stable system image from which to copy data, the objective of a VSS-based restore is to return files to disk in a manner most useful to applications (writers) currently running on the system. This does not require a stable system image, so no shadow copy is necessary.

Instead, attention is focused on avoiding disruption of writer activity, preventing the creation of inconsistent data sets on disk, and allowing writers the necessary scope to evaluate and merge data when necessary.

Like a backup operation, a VSS restore requires access to both a Backup Components Document and to Writer Metadata Documents. These documents are not generally obtained by querying running applications, but instead are retrieved from versions stored as XML documents on backup media.

As is the case during backup operations, the Writer Metadata Documents remain read-only objects, and (once retrieved) the Backup Components Document can still be modified. Modifications of the Backup Components Document allow writers and requester to communicate about the following:

- Overriding *restore methods* with *restore targets*
- Using *alternate location mappings*
- Restoring files to new locations on disk
- Using different selection rules from those used during backup

To more fully understand the basic tasks involved in performing a restore, it is useful to break down this overview into the following topics:

- Overview of Restore Initialization
- Overview of Preparing for Restore
- Overview of Actual File Restoration
- Overview of Restore Clean up and Termination

Overview of Restore Initialization

3/5/2021 • 3 minutes to read • Edit Online

In initializing a VSS restore operation, a requester needs to retrieve the Backup Component Document and each relevant Writer Metadata Document created and saved during the backup operation. The writer will have its current state queried in handling the *Identify event* that the requester generates. For more information, see Overview of Processing a Restore Under VSS.

The following table shows the sequence of actions and events that are required to initialize a restore operation.

REQUESTER ACTION	EVENT	WRITER ACTION
Create an **IVssBackupComponents** interface, initialize it to manage a restore, and load stored requester metadata (see **CreateVssBackupComponents**, **IVssBackupComponents::InitializeForRestore**).	None	None
Call **CreateVssExamineWriterMetadata** to create **IVssExamineWriterMetadata** interfaces and load them with stored writer metadata.	None	None
Initiate asynchronous contact with writers (see **IVssBackupComponents::GatherWriterMetadata**.)	*Identify*	The writer begins event handling in support of the restore. Creates the Writer Metadata Document (see Working with the Writer Metadata Document, **CVssWriter::OnIdentify**, **IVssCreateWriterMetadata**).
The requester waits for writers to initialize by calling **IVssAsync**.	None	None

Requester Actions during Restore Initialization

During the initialization phase of a restore, the requester needs to have access to the stored Backup Components Document and all the Writer Metadata Documents.

Depending on the implementation, this will mean either that the requester will require that backup media be mounted and readable, or that some other mechanism for accessing the stored metadata be available.

The requester uses the stored XML document containing the Backup Components Document of the requester that performed the backup to initialize its Backup Components Document using **IVssBackupComponents::InitializeForRestore** can access the information.

As was the case during the backup, the Backup Components Document has insufficient information to support a restore; therefore, the requester needs access to those Writer Metadata Documents stored during backup (see Use of Components by the Requester).

The requester retrieves the stored writer metadata by calling **CreateVssExamineWriterMetadata** for each writer whose data was backed up and now is to be restored. This function creates an **IVssExamineWriterMetadata** object for each writer and loads the writer's Writer Metadata Document into the object.

As was the case during the backup, to initiate cooperation between itself and the system's writers, a requester must generate an *Identify* event by calling **IVssBackupComponents::GatherWriterMetadata**. It is not necessary to call **IVssBackupComponents::GatherWriterStatus** following the completion of **GatherWriterMetadata**. Writers that fail to process the *Identify* event will not be included in the list of writers providing the metadata to be returned by **IVssBackupComponents::GetWriterMetadataCount** and **IVssBackupComponents::GetWriterMetadata** (see Determining Writer Status).

As with the backup operation, a requester will need to query and parse the information in the Backup Components Document and compare it to data in the Writer Metadata Documents to determine which components were backed up and to choose those to be restored (see Overview of Preparing for Restore). In addition, the requester will need to generate a detailed list containing information about the files on the backup media selected for restore, as well as how and where they are to be restored. (See Generating a Restore Set.)

Therefore, some backup applications may find it useful to have stored on the backup media their own list (in their own optimized format) of the files and their associated writer, component, restore procedure, and location information. This list can be used to minimize the amount of parsing and comparing of Writer Metadata Documents and the Backup Component Documents required to support a restore.

Writer Actions during Restore Initialization

Just as is done during a restore operation, in response to the Identify event, VSS calls each writer's virtual handler method **CVssWriter::OnIdentify**.

Note that applications other than the current requester (for instance, system applications) can generate Identify events, which must be handled by the writer. In addition, there is no way for a writer to determine from within **CVssWriter::OnIdentify** which application has generated the Identify event.

Given that a writer may receive several Identify events while processing a restore operation, writers should never set state information in the **CVssWriter::OnIdentify** handler. Instead, they must use the same algorithm for creating their Writer Metadata Document as was done during backup operations (see Writer Actions during Backup Initialization for more information).

Overview of Preparing for Restore

3/5/2021 • 5 minutes to read • Edit Online

In preparing for a restore, a requester uses the stored Writer Metadata Documents in conjunction with its own retrieved Backup Components Document to determine what is to be restored and how. For more information, see Overview of Processing a Restore Under VSS.

Following the selection of restore candidate components, writers currently running on the system access the requester's Backup Components Document. Writers use this access to indicate how to cause minimum difficulty for running services due to the restore.

After this is completed, the requester has enough information to determine which files will need to be restored, as well as where and how they should be restored. (For more information, see Generating a Restore Set.)

The following table shows the sequence of actions and events that are required to prepare for a restore operation.

REQUESTER ACTION	EVENT	WRITER ACTION
Retrieve information from the Backup Components Document about the components *explicitly included* in the backup operation (see **IVssBackupComponents::GetWriterComponents**) Examine retrieved Writer Metadata Documents to get details about those components explicitly included in the backup, and any find *implicitly included* subcomponents. (See **IVssExamineWriterMetadata**, **IVssWMComponent**.)	None	None
Select components and component sets to be restored (see **IVssBackupComponents::SetSelectedForRestore** and **IVssBackupComponents::AddRestoreSubcomponent**.)	None	None
The requester allows the writer to update the Backup Components Document and may optionally communicate any special restore options to the writer. (See **IVssBackupComponents::SetRestoreOptions**, **IVssBackupComponents::AddNewTarget**, and **IVssBackupComponents::PreRestore**.)	*PreRestore*	The writer determines participation in the restore, prepares files to restore, and optionally modifies Backup Components Document if necessary. (See **CVssWriter::OnPreRestore**, **IVssComponent**, **IVssComponent::IsSelectedForRestore**, **IVssComponent::GetRestoreOptions**, **IVssComponent::SetRestoreTarget**, **IVssComponent::SetRestoreMetadata**, **IVssComponent::AddDirectedTarget**.)

REQUESTER ACTION	EVENT	WRITER ACTION
The requester waits on writers to handle the **PreRestore** event with **IVssAsync**. It should also verify writer status. (See **IVssBackupComponents::GatherWriterStatus**, **IVssBackupComponents::GetWriterStatus**.)	None	None

Requester Actions during Restore Preparations

To determine which components are candidates for restore, the requester must do the following:

- Establish the component and the *component set* structure used to make the backup.
- Examine the components' *selectability for restore*.
- Use selectability guidelines (Working with Selectability for Restore and Subcomponents) to choose components to include.
- Use component *file set* information to determine which files on the backup media must be restored.

To do this, the requester needs to examine *explicitly included* components in the stored Backup Components Document. This component information is available on a writer-by-writer basis using **IVssBackupComponents::GetWriterComponents**, which returns instances of the **IVssWriterComponentsExt** interface, from which both writer information and instances of the **IVssComponent** interface can be retrieved.

As noted elsewhere (Use of Components by the Requester), the Backup Components Document and the **IVssComponent** interface do not contain enough information to support the backup. Therefore, the requester must examine the corresponding stored Writer Metadata Document by using **IVssExamineWriterMetadata** (see Writer Identification Information).

The number of components each writer manages is returned by **IVssExamineWriterMetadata::GetFileCounts**. The requester can then use **IVssExamineWriterMetadata::GetComponent** to get an **IVssWMComponent** interface for each component a writer manages.

By examining the components' *selectability for backup* and *logical paths* (see Working with Selectability and Logical Paths), a requester is able to identify the components that defined backup-time component sets (explicitly included components), and the subcomponents members of those sets (implicitly included components).

Requesters indicate through the Backup Components Document if a component is to be explicitly restored, using either **IVssBackupComponents::SetSelectedForRestore** or **IVssBackupComponents::AddRestoreSubcomponent**. The choice of method will depend on how the component was originally backed up and its *selectability for restore*. These components explicitly included for restore designate other components which are implicitly included (see Working with Selectability for Restore and Subcomponents for details).

A requester may explicitly include none of a currently executing writer's components for restore using **IVssBackupComponents::SetSelectedForRestore** or **IVssBackupComponents::AddRestoreSubcomponent**. In this case, that writer will not receive any VSS events for the remainder of the restore operation.

Explicitly using either **IVssBackupComponents::SetSelectedForRestore** or **IVssBackupComponents::AddRestoreSubcomponent** to select a component of a writer that is not currently running returns a VSS_E_OBJECT_NOT_FOUND error. See Restores without Writer Participation for information on restoring the data of missing writers.

To enable a writer to have complete information upon which to act, writer-specific restore options and indication of an incremental restore can be sent to the writers by requester calls to **IVssBackupComponents::SetRestoreOptions** and **IVssBackupComponents::SetAdditionalRestores**, respectively.

At this point, a requester has finished its preparation, and it generates a **PreRestore** event by calling **IVssBackupComponents::PreRestore**, allowing writers to prepare for the actual restore.

Writer Actions during Restore Preparations

Writer preparation for the restore operation occurs when handling the **PreRestore** event with the virtual method **CVssWriter::OnPreRestore**. The default implementation simply returns without taking any action. Writers may choose to override the default implementation to exercise more control by:

- Overriding *restore methods* with *restore targets*
- Defining *directed targets*
- Creating error messages and additional data
- Supplying backup stamp information

The event handler **CVssWriter::OnPreRestore** receives an instance of the **IVssWriterComponents**, from which it can obtain **IVssComponent** interfaces for those of its components explicitly included in the Backup Components Document during backup.

Information about subcomponents implicitly included in backup operations and explicitly included in restores by using an instance of the **IVssComponent** corresponding to the component that defined its backup *component set*.

The **IVssComponent::IsSelectedForRestore** method is used to determine whether an explicitly included for backup component is to be restored.

To determine if a backup subcomponent was explicitly included in the restore, writers use **IVssComponent::GetRestoreSubcomponent**.

The writer should examine the *file set* in each component and determine if it needs to take actions to support the restore. The writer will need to evaluate if it wants its current files overwritten, or if it will require restoration to new locations. Actions can include the following:

- Getting and acting on any writer- or requester-specific options governing restore operations (see **IVssComponent::GetRestoreOptions**)
- Closing and making writable any currently open files
- Updating the restore target (for instance, to force restoration to an alternate location mapping). See **IVssComponent::SetRestoreTarget**.
- Communicating with the requester via private metadata (see **IVssComponent::SetRestoreMetadata**)
- Indicating that a file should be restored by remapping through the definition of *directed targets* (see **IVssComponent::AddDirectedTarget**)

The instance of **IVssComponent** used will either be that created by the component's explicit inclusion in the Backup Components Document during backup, or that of the component defining the backup component set of which it was a member (see Working with Selectability For Restore and Subcomponents).

Overview of Actual File Restoration

3/5/2021 • 2 minutes to read • Edit Online

After performing the actions described in Overview of Restore Initialization and Overview of Preparing for Restore, the requester has sufficient information to begin restoring files. File restoration does not involve writer interactions or the generation of events. For more information, see Overview of Processing a Restore Under VSS.

The following table shows the sequence of actions and events that are required to restore files.

REQUESTER ACTION	EVENT	WRITER ACTION
Generate a restore set listing for files on backup media.	None	None
Handle *directed targets* or *partial file* restoration (see **IVssComponent::GetDirectedTarget**, **IVssComponent::GetPartialFile**).	None	None
If necessary, ignore all specified restore locations and restore to a new location specified in an earlier call to **IVssBackupComponents::AddNewTarget**.	None	None
If the restore is incremental and further restores are needed, indicate (see **IVssBackupComponents::SetAdditionalRestores** and Incremental and Differential Backups).	None	None
To learn whether a writer has modified the contents of the Backup Components Document, call **IVssBackupComponents::GetWriterComponents**. For example, the writer might have changed the restore target.	None	None

Requester Actions during Restoring Files

For most files on the backup media, the requester needs to determine their original locations and any new locations or alternate location mappings that apply to them. (See Generating a Restore Set for a discussion of best practices in determining which files to restore and where to restore them.)

In addition, some files may have *directed targets* or support *partial file* restoration. The number of such files can be found by calling **IVssComponent::GetDirectedTargetCount** and **IVssComponent::GetPartialFileCount**, and information on detailed restoration instructions can be found by calling **IVssComponent::AddDirectedTarget** and **IVssComponent::GetPartialFile**. (Partial and directed files can be

part of components added implicitly or explicitly to the original backup, see Working with Selectability For Restore and Subcomponents for more information.)

Success or failure of a restore is indicated on a component-by-component basis using **IVssBackupComponents::SetFileRestoreStatus**. The need for further restore operations (in the case of incremental or differential restores) is also indicated on a component-by-component basis using **IVssBackupComponents::SetAdditionalRestores**.

In general, VSS does not specify a mechanism for retrieving data from a storage media, a choice of storage medium, or how to determine which files should be restored where.

However, for certain writers, restoring files may involve the use of a documented custom interface and procedure. Windows system writers, which currently require such support, are documented in Special VSS Usage Cases.

In general, it is recommended that the files of each component of each *writer instance* be processed as a unit. This requires the following:

- Associating each file to be restored with the component that managed it. This requires the use of Writer Metadata Documents.
- Obtaining correct restoration target information. This requires information from the Backup Components Document.

Overview of Restore Clean up and Termination

3/5/2021 • 2 minutes to read • Edit Online

Following a restore, writers check the status of the operation so that they can make use of the restored data and deal with errors. The requester must wait for the completion of this activity. For more information, see Overview of Processing a Restore Under VSS.

The following table shows the sequence of actions and events that are required after a restore operation has taken place.

REQUESTER ACTION	EVENT	WRITER ACTION
The requester indicates the end of the restore (see **IVssBackupComponents::PostRestore**).	*PostRestore*	The writer conducts post-restore cleanup, and handles restoration failures and files that have been restored to nonstandard locations (see **CVssWriter::OnPostRestore**, **IVssComponent**).
The requester waits on writers to handle the **PostRestore** event with **IVssAsync**. It should also verify writer status (see **IVssBackupComponents::GatherWriterStatus**, **IVssBackupComponents::GetWriterStatus**).	None	None
The requester releases the **IVssBackupComponents** interface.	None	None

Requester Actions during Cleanup and Termination

At this point, a requester indicates the end of its file restoration activities by generating a *PostRestore* event by calling **IVssBackupComponents::PostRestore**.

The requester's actions are limited to waiting on the writers, which may need to perform some final cleanup and handle restore errors, and releasing the **IVssBackupComponents** interface after all writers have returned from handling the *PostRestore* event.

Writer Actions during Cleanup and Termination

The *PostRestore* event is handled by the virtual method **CVssWriter::OnPostRestore**. The default implementation simply returns **true** without taking any action. If a writer needs to exercise more control of the post-restore situation, it can override this method.

In addition to any normal cleanup (such as removing temporary files) that a writer might perform in **CVssWriter::OnPostRestore**, it can handle the success or failure of restore operations.

How it handles restore errors, files restored to an alternate location, and the need for future restores are completely at the writer's discretion. Typical actions might include comparing files in alternate or new locations

with files currently in use, merging data from multiple files, or starting new sessions connected to the new data files. VSS provides the following mechanisms for supporting this on a component-by-component basis:

- Success or failure in restoring any component can be found with **IVssComponent::GetFileRestoreStatus**.
- The use of alternate location mappings in restoring files will be indicated by **IVssComponent::GetAlternateLocationMapping**.
- Determining if a restore is incremental and will require further restores is done by calling **IVssComponent::GetAdditionalRestores**. Writers that need a complete restoration of their data should not restart until this method returns false.
- Writers can determine if the requester has needed to restore files to a previously unspecified location using **IVssComponent::GetNewTargetCount** and **IVssComponent::GetNewTarget**

(For more information on restoring files to non-default locations, see Non-Default Backup and Restore Locations.)

As with any **IVssComponent** method, the information returned by a given instance applies to those components *explicitly included* for backup and any of its *implicitly included* for backup subcomponents, including those subcomponents explicitly included for restore by the requester using **IVssBackupComponents::AddRestoreSubcomponent** (see Working with Selectability For Restore and Subcomponents for details).

Because the writers will require access to the Backup Components Document, it is important that the requester not release the **IVssBackupComponents** interface until writers have finished processing.

Developing VSS Hardware Providers

3/5/2021 • 2 minutes to read • Edit Online

Hardware providers implement the **IVssProviderCreateSnapshotSet** and **IVssHardwareSnapshotProvider** interfaces. **IVssProviderCreateSnapshotSet** implements the shadow copy state sequencing. **IVssHardwareSnapshotProvider** operates on a LUN abstraction. Providers must be implemented as an out-of-process COM server. Providers use provider-specific mechanisms (often private IOCTLs) to communicate with the storage subsystem that instantiates the shadow copy LUNs.

- Shadow Copy Provider Registration and Loading
- Shadow Copy Creation for Providers
- Hardware Provider Interactions and Behaviors

Shadow Copy Provider Registration and Loading

3/5/2021 • 2 minutes to read • Edit Online

Provider Registration

Only VSS calls providers. The provider registers for calls by invoking **IVssAdmin::RegisterProvider**, passing **VSS_PROV_HARDWARE** for the *eProviderType* parameter. Once registered, the provider will be involved in shadow copy management until unregistering using **IVssAdmin::UnregisterProvider**.

Provider Loading and Unloading

Providers can be frequently loaded and unloaded. Providers can implement **IVssProviderNotifications** to determine when VSS is loading or unloading the provider.

Shadow Copy Creation for Providers

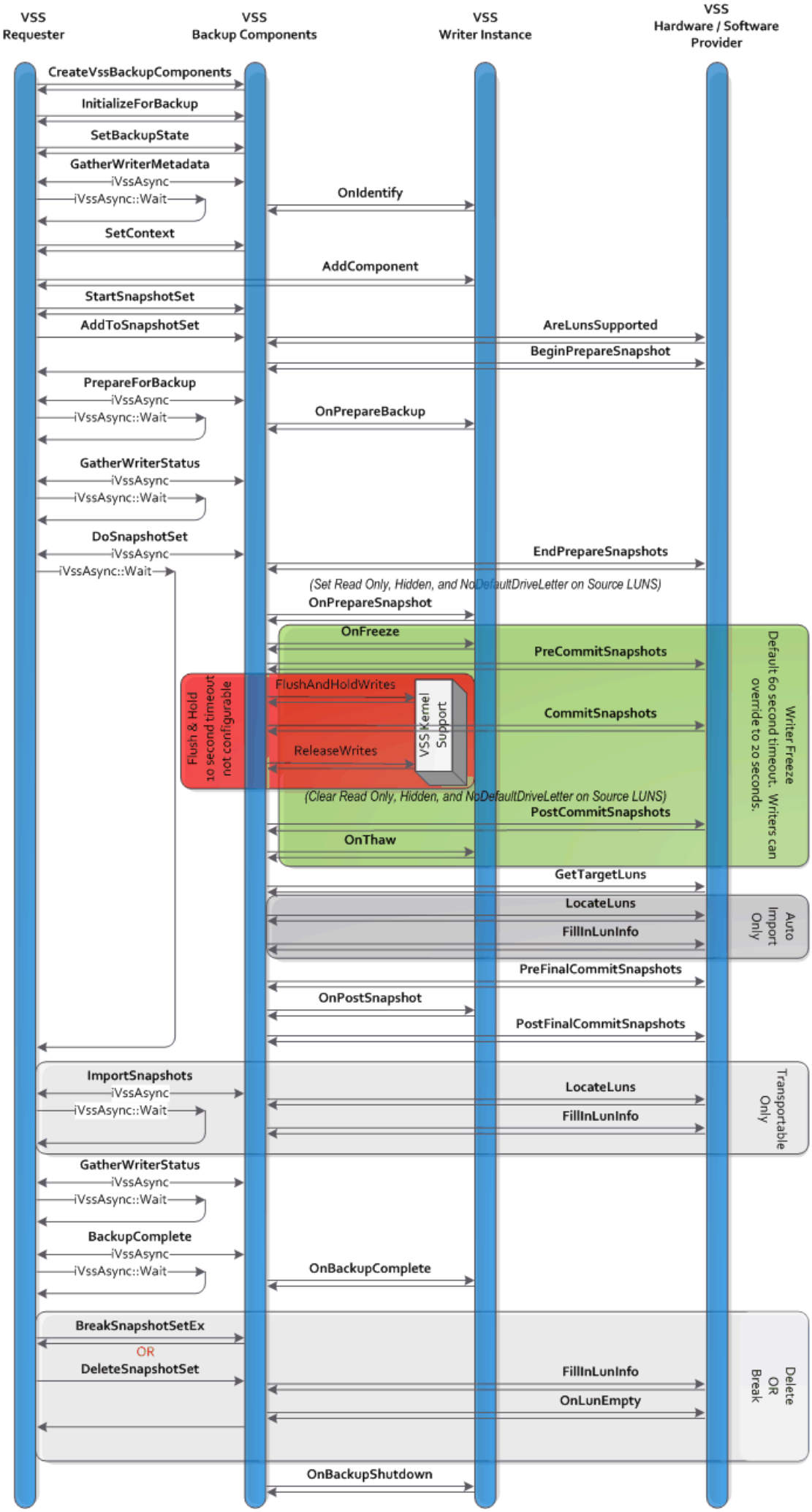
3/5/2021 • 6 minutes to read • Edit Online

The Shadow Copy Creation Process

A requester is the application that initiates the request to create a shadow copy. Typically the requester is a backup application. As necessary, VSS will call the providers involved. Most providers are interested in three specific requests from the requester.

1. The requester begins the shadow copy creation activity with a call to **IVssBackupComponents::StartSnapshotSet**. This generates a **GUID** of type **VSS_ID** that uniquely identifies this specific shadow copy set - the SnapshotSetId. The provider is not involved in this step, but the SnapshotSetId is used extensively in all subsequent steps.
2. For each volume it wishes to include in this shadow copy set, the requester calls **IVssBackupComponents::AddToSnapshotSet**. VSS determines which provider will be used to shadow copy the volume.
   - Multiple providers may participate in a shadow copy set. For example, if the system volume and a data volume are part of the same shadow copy set, the system provider may serve as the shadow copy provider for the system volume while a hardware provider may serve as the shadow copy provider for the data volume. Both providers would be part of the same shadow copy set and the user would expect the same point-in-time consistency across both volumes.
   - For a hardware provider to be selected, the hardware provider must be able to support all LUNs contributing to the specified volume.
   - All registered providers are given the opportunity to indicate support for a given volume during shadow copy creation. If more than one provider indicates support, VSS will first default to hardware providers, then software providers, and finally the system provider (if no other provider indicates support for that volume).
   - A requester may override this default order by explicitly indicating the provider it requires to create the shadow copy.
   - If there are multiple hardware providers that support a given volume, there is no guarantee to the order in which the hardware providers will be called.
3. After one or more calls to **AddToSnapshotSet**, the requester can ask for the shadow copy to be created by using the **IVssBackupComponents::DoSnapshotSet** method. VSS then works with the system to create the shadow copy. The **DoSnapshotSet** method performs this work asynchronously, and the requester can either poll or wait for the shadow copy creation process to complete.

This diagram shows the interactions between the requester, the VSS service, the VSS kernel support, any VSS writers involved, and any VSS hardware providers. See The Shadow Copy Creation Process for a detailed description of these interactions.


VSS Requester
VSS Backup Components
VSS Writer Instance
VSS Hardware / Software Provider
CreateVssBackupComponents
InitializeForBackup
SetBackupState
GatherWriterMetadata
iVssAsync
iVssAsync::Wait
OnIdentify
SetContext
AddComponent
StartSnapshotSet
AddToSnapshotSet
AreLunsSupported
BeginPrepareSnapshot
PrepareForBackup
iVssAsync
iVssAsync::Wait
OnPrepareBackup
GatherWriterStatus
iVssAsync
iVssAsync::Wait
DoSnapshotSet
iVssAsync
iVssAsync::Wait
EndPrepareSnapshots
(Set Read Only, Hidden, and NoDefaultDriveLetter on Source LUNS)
OnPrepareSnapshot
OnFreeze
PreCommitSnapshots
FlushAndHoldWrites
VSS Kernel Support
CommitSnapshots
ReleaseWrites
Flush & Hold 10 second timeout not configurable
(Clear Read Only, Hidden, and NoDefaultDriveLetter on Source LUNS)
PostCommitSnapshots
OnThaw
Writer Freeze Default 60 second timeout. Writers can override to 20 seconds.
GetTargetLuns
LocateLuns
FillInLunInfo
Auto Import Only
PreFinalCommitSnapshots
OnPostSnapshot
PostFinalCommitSnapshots
ImportSnapshots
iVssAsync
iVssAsync::Wait
LocateLuns
FillInLunInfo
Transportable Only
GatherWriterStatus
iVssAsync
iVssAsync::Wait
BackupComplete
iVssAsync
iVssAsync::Wait
OnBackupComplete
BreakSnapshotSetEx
OR
DeleteSnapshotSet
FillInLunInfo
OnLunEmpty
Delete OR Break
OnBackupShutdown

When the shadow copy creation process is complete, the requester can determine if the shadow copy creation was successful, and if not, determine the source of the failure.

The time interval between the freeze and thaw of the writer applications must be minimized. Provider must asynchronously start all preparation work related to the shadow copy (such as a hardware provider that uses plexes starting the synchronization) in the **IVssHardwareSnapshotProvider::BeginPrepareSnapshot** method, and then wait for the completions in the **IVssProviderCreateSnapshotSet::EndPrepareSnapshots** method.

There are multiple timing limit windows that providers must follow. As a result, well-behaved providers will perform all unnecessary processing before **IVssProviderCreateSnapshotSet::PreCommitSnapshots** and after **IVssProviderCreateSnapshotSet::PostCommitSnapshots**.

The shadow copy set is fixed when **DoSnapshotSet** is called. Additional volumes cannot be added later because the additional volumes would not share the same point-in-time.

There is a limit of 64 volumes in the shadow copy set. A specific volume may map to an entire LUN, a portion of a LUN, or portions of multiple LUNs. Most configurations will have one volume per LUN, although arbitrary mappings are possible.

There is no limit on the number of shadow copy sets or the number of shadow copy sets of an original volume. A provider may define specific limits, or dynamically limit based on hardware resources available.

Point-in-time for Writerless Applications

VSS includes special support that defines the point-in-time that is common for all volumes in a shadow copy set. Hardware providers do not need to directly interface with these kernel technologies, since they are invoked as part of the normal shadow copy commit processing. However, it is useful to understand the mechanisms used because it explains the definition of 'point-in-time' for writerless applications (applications that have not exposed a VSS Writer interface and therefore do not participate in the volume shadow copy creation process.)

This VSS kernel support for common point-in-time is distributed between the VolSnap.sys driver, the file systems, and VSS.

1. Before the VSS kernel support is invoked, VSS has already:
    a. Determined which volumes are to be involved in the shadow copy.
    b. Determined which provider is to be used on each volume.
    c. Frozen applications that are accepting freeze/thaw messages.
    d. Prepared the providers for the shadow copy by calling the **PreCommitSnapshots** methods. All providers are now waiting to do the actual shadow copy creation.
2. The point-in-time is then created. VSS concurrently flushes the file systems on all of the volumes that are to be shadow copied.
    a. VSS issues an IOCTL_VOLSNAP_FLUSH_AND_HOLD_WRITES control command on each volume that flushes the file systems. That IOCTL is passed down the storage stack to VolSnap.sys. VolSnap.sys then holds all write IRPs until step 4 below. Any file system (such as RAW) without support for this new IOCTL passes the unknown IOCTL down—where it is again held by VolSnap.sys. On NTFS volumes, the flush also commits the NTFS log.
    b. This suspends all NTFS/FAT metadata activity; the file system metadata is cleanly committed.
    c. The shadow copy instant: VolSnap.sys causes all subsequent write IRPs to be queued on all of the volumes that are to be shadow copied.
    d. VolSnap.sys waits for all pending writes on the shadow copied volumes to complete. The volumes are now quiescent with respect to writes, and were quiescent at exactly the same moment on each volume. There are no guarantees about writes to user mapped sections or writes issued between (a) and (b) on file systems that do not implement the flush IOCTL (e.g. RAW).

3. VSS instructs each provider to take in the shadow copy by calling the **IVssProviderCreateSnapshotSet::CommitSnapshots** methods. The providers should have all preparation done so that this is a quick operation.

   Note that the I/O system is quiescent only while these **CommitSnapshots** methods are executing. If a provider performs any synchronization of the source and shadow copy LUNs, this synchronization must be completed before the provider's **CommitSnapshots** method returns. It cannot be performed asynchronously.

4. Immediately after the last provider's **CommitSnapshots** method returns, VSS releases all pending write IRPs (including the IRPs that were blocking the file systems at the conclusion of their commit paths) by invoking another IRP passed to VolSnap.sys.

5. If the shadow copy process was successful, then VSS now:

   a. Calls **PostCommitSnapshots** for the providers involved.
   b. Calls **CVssWriter::OnThaw** for the writers involved.
   c. Informs the requester that the shadow copy process has completed.

PreCommitSnapshots, **CommitSnapshots**, to **PostCommitSnapshots** are all time critical. All I/O from applications with writers is frozen from **PreCommitSnapshots** to **PostCommitSnapshots**; any delays affect application availability. All file I/O, including writerless application I/O, is suspended during **CommitSnapshots**.

Providers should complete all time-critical work prior to returning from **EndPrepareSnapshots**.

- **CommitSnapshots** should be returned within seconds. The **CommitSnapshots** phase is located within the Flush and Hold window. VSS kernel support will cancel the Flush and Hold that is holding the I/O if the subsequent release is not received within 10 seconds, and VSS will fail the shadow copy creation process. Other activities will be happening on the system, so a provider should not rely on having the full 10 seconds. The provider should not call Win32 APIs during commit as many will result in unexpected writes and block. If the provider takes more than a few seconds to complete the call, there is a high probability that this will fail.
- The full sequence from **PreCommitSnapshots** to the return of **PostCommitSnapshots** maps to the window between writers receiving the Freeze and Thaw events. The writer default for this window is 60 seconds, but a writer may override this value with a smaller timeout. For example, the Microsoft Exchange Server writer changes the timeout to 20 seconds. Providers should not spend more than a second or two in this method.

During **CommitSnapshots** the provider must avoid any non-paging file I/O; such I/O has a very high probability of deadlocking. In particular, the provider should not synchronously write any debug or trace logs.

Hardware Provider Interactions and Behaviors

3/5/2021 • 2 minutes to read • Edit Online

Hardware providers may support copy-on-write and/or full copy mirror (differential and/or plex) shadow copies. Resource allocation for shadow copies should follow the context specified by the requester in **IVssBackupComponents::SetContext**, enumerated by **_VSS_SNAPSHOT_CONTEXT**, for the shadow copy set.

- If the requester specifies a shadow copy context that is supported by the provider, the provider should create a shadow copy using that method.
- If the requester specifies a shadow copy context that is not supported by the provider, then the provider should not attempt to create the shadow copy and return **FALSE** through the *pbIsSupported* parameter in **IVssHardwareSnapshotProvider::AreLunsSupported**.
- If the requester does not explicitly specify a shadow copy context, then the provider should use reasonable default behavior for shadow copy creation.

Hardware providers associated with SAN RAID subsystems should support transportable shadow copies to allow movement between hosts on a SAN.

Hardware providers running on multiple machines on a SAN yet managing the same RAID subsystem do not need to coordinate between providers on a SAN. The coordinator maintains any necessary state. Two kinds of state are recognized:

- State necessary to support data access to the volumes contained on a hardware shadow copy. This includes any tagging of a volume as read-only or hidden. This state must be on the hardware LUN and travel with the LUN. This state is preserved across boot epochs and/or device discovery. VSS manages this state for the lifetime of the shadow copy.
- State necessary to recognize a specific volume as part of a shadow copy set. This state is persisted by VSS in conjunction with the requester that originally created the shadow copy set.

For more information, see the following topics:

- The Shadow Copy Creation Process
- Required Behaviors for Shadow Copy Providers
- State Transitions in shadow copy providers
- Error Handling in shadow copy providers

The Shadow Copy Creation Process

3/5/2021 • 5 minutes to read • Edit Online

The following is a detailed description of the hardware provider's role in the creation of a shadow copy set. For a diagram of this process, see Shadow Copy Creation for Providers.

1. As the requester adds each volume to the shadow copy set, VSS determines the associated LUNs. For example, a spanned volume could be composed of multiple LUNs. VSS creates an initial **VDS_LUN_INFORMATION** using physical device information for each LUN.

2. VSS calls **AreLunsSupported** to determine if the provider supports a shadow copy for that LUN. The hardware provider uses the **VDS_LUN_INFORMATION** to determine whether the LUNs are supported. This determination must be all or nothing—the same hardware provider must support all LUNs contributing to a specific volume. In the spanned volume example, the provider cannot indicate that it supports only one of the LUNs that comprise the volume.

3. For each volume in the shadow copy set, VSS calls **IVssHardwareSnapshotProvider::BeginPrepareSnapshot** with the same array of **VDS_LUN_INFORMATION** structures. Within a single call, all LUNs in the array are unique, but the same LUN may appear in a subsequent call whenever the same LUN contributes to multiple volumes. The provider must handle that case correctly.

4. Immediately after **IVssProviderCreateSnapshotSet::EndPrepareSnapshots**, VSS will set the read-only, hidden, no drive letter, and shadow copy flags on all affected LUNs. The flags are implemented using hidden sectors on MBR disks or operating-system-specific bits in the partition header entry on GPT disks. The flags will cause all volumes on the affected disks to be surfaced on the receiving machines as read-only and hidden. Note that the shadow-copied LUNs have not been committed at this stage, so that in the event of a system crash, the flags will be cleared and the original LUN will not be affected. That is, all volumes on the original LUN will be treated as normal—and keep their original drive-letter assignments, mounted folders, and read/write status.

   > **NOTE**
   > Providers should not attempt to modify any of the LUN flags.

5. At **CommitSnapshots**, the shadow copy LUNs are committed. **CommitSnapshots** should be returned within a few seconds or the whole shadow copy creation process will probably fail.

6. After **CommitSnapshots**, VSS clears the flags on all original LUNs involved in the shadow copy. Since the shadow copy LUNs have been committed at this point, the shadow copy LUNs retain these flags.

7. After **IVssProviderCreateSnapshotSet::PostCommitSnapshots**, VSS clears the LUN flags (that it set after **IVssProviderCreateSnapshotSet::EndPrepareSnapshots**) and notifies the writers to thaw. VSS then calls the provider's **IVssProviderCreateSnapshotSet::PreFinalCommitSnapshots** and **IVssProviderCreateSnapshotSet::PostFinalCommitSnapshots** methods.

8. VSS retrieves an array of **VDS_LUN_INFORMATION** structures, one for each newly created shadow copy LUN, by calling **IVssHardwareSnapshotProvider::GetTargetLuns**. The provider must provide enough information to allow VSS and the provider to identify the shadow copies at any later time and on any system connected to the subsystem. At this time, the shadow copy LUNs should be inaccessible to

this machine—the provider must use some mechanism other than by simply reading the information from the LUN.

9. For transportable shadow copies, VSS appends the following to the Backup Components Document describing the shadow copy set:
    a. The original LUN VDS_LUN_INFORMATION array.
    b. The new shadow copy LUN VDS_LUN_INFORMATION array.
    c. The disk extents for each volume in the shadow copy set.
10. For auto-import shadow copies, the import process begins. In IVssHardwareSnapshotProvider::LocateLuns, the provider will be given the VDS_LUN_INFORMATION generated in GetTargetLuns at creation time. During **LocateLuns**, the provider should make those shadow copy LUNs visible to the system. The provider should not cause a bus rescan. VSS will cause the rescan and enumeration to allow the LUNs to be discovered by PNP and the volumes to come online.
11. VSS calls IVssHardwareSnapshotProvider::FillInLunInfo for the newly arrived disks in the system. VSS uses the shadow copy LUN VDS_LUN_INFORMATION array from **FillInLunInfo**, comparing it with the **VDS_LUN_INFORMATION** generated during creation in GetTargetLuns and the disk extents for each volume in the shadow copy set to identify the newly arrived shadow copy LUNs.
12. At this point, all volumes contained on the shadow copy LUNs are mounted hidden and read-only, and VSS has a mapping from original volume to shadow copy volume.

Because a single LUN can contain portions of multiple volumes, the set of shadow copy LUNs may include "extra" volumes. Those volumes are not part of the shadow copy set and should not be used as they are not guaranteed to be consistent.

After import, the shadow copy can be accessed via the IVssBackupComponents::Query method. A shadow copy can also be exposed as a drive letter, mounted folder, or share using IVssBackupComponents::ExposeSnapshot. A shadow copy set may also be converted to a regular LUN by using the IVssBackupComponents::BreakSnapshotSet method. From there management applications can use appropriate disk management APIs to further manage the LUN (such as changing the read/write attributes).

Transportable Shadow Copies on a SAN

Beginning with Windows Server 2003, Datacenter Edition and Windows Server 2003, Enterprise Edition, VSS enables transportable shadow copy sets on a SAN. From the point of view of a provider capable of transporting LUNs between hosts, there is little or no difference between a non-transportable and a transportable shadow copy set.

Windows Server 2003, Standard Edition, Windows Server 2003, Web Edition and Windows XP: Transportable shadow copy sets are not supported. All editions of Windows Server 2003 with Service Pack 1 (SP1) support transportable shadow copy sets.

Transportable shadow copy sets must be created with the **VSS_VOLSNAP_ATTR_TRANSPORTABLE** attribute added to the shadow copy context described by the _VSS_VOLUME_SNAPSHOT_ATTRIBUTES enumeration. All volumes in the set must be transportable. By returning success from IVssHardwareSnapshotProvider::AreLunsSupported, the provider is indicating that not only does it support the LUN, but also that it can transport the LUN. The creation of a transportable shadow copy set concludes when VSS records the LUN information in the Backup Components Document. A shadow copy set may only be imported once after initial creation.

On the receiving machine, the requester imports the shadow copy set. The IVssBackupComponents::ImportSnapshots method uses the Backup Components Document describing the shadow copy set as input. Importing proceeds by:

1. VSS constructs an array of shadow copy **VDS_LUN_INFORMATION** structures from the Backup Components Document.
2. When VSS calls **IVssHardwareSnapshotProvider::LocateLuns**, the array of **VDS_LUN_INFORMATION** structures generated in **IVssHardwareSnapshotProvider::GetTargetLuns** will be passed to the provider. While processing this method, the provider should make those shadow copy LUNs visible to the system. The provider should not cause a bus rescan. VSS will trigger the rescan and enumeration to allow the LUNs to be discovered by PNP and the volumes to come online.
3. VSS calls **IVssHardwareSnapshotProvider::FillInLunInfo** for the newly arrived disks in the system. VSS uses the shadow copy LUN array of **VDS_LUN_INFORMATION** structures from **FillInLunInfo**, comparing it with the **VDS_LUN_INFORMATION** generated during shadow copy set creation in **GetTargetLuns** and the disk extents for each volume in the shadow copy set to identify the newly arrived shadow copy LUNs.
4. At this point, all volumes contained on the transported LUNs are mounted hidden and read-only and VSS has a mapping from the original volume to the shadow copy volume.

For both the single machine and transportable case, the sequence is similar starting with the point that **GetTargetLuns** is called. For auto-import, the steps happen directly after creation.

Required Behaviors for Shadow Copy Providers

3/5/2021 • 4 minutes to read • Edit Online

A shadow copy provider must conform to guidelines to ensure requester compatibility. At runtime, a backup application or any other requester that uses shadow copies must function correctly without any knowledge of specific provider implementation details.

Automagic Resource Management

It must be possible for the requester to simply add a volume to a shadow copy set. The requester can specify shadow copy attributes such as **VSS_VOLSNAP_ATTR_TRANSPORTABLE**, **VSS_VOLSNAP_ATTR_DIFFERENTIAL**, and/or **VSS_VOLSNAP_ATTR_PLEX** in the _VSS_SNAPSHOT_CONTEXT. The provider must honor those attributes. If it cannot honor those attributes, then it should indicate that the shadow copy set is unsupported by setting the **pbIsSupported* in IVssHardwareSnapshotProvider::AreLunsSupported to **FALSE**.

The provider must never agree to support a shadow copy that it cannot create. If the requester does not specify a plex or differential attribute, the provider must include automagic logic for allocating subsystem space for the shadow copy and create the shadow copy using the most appropriate method. The hardware provider must not include a provider-specific API for managing required shadow copy properties.

Created Shadow Copy Volumes Are Read-Only and Hidden

VSS sets flags on each affected LUN such that the resulting shadow copy volumes will be hidden and read-only when detected by a computer running Windows Server 2003. Drive letters and mounted folders are not automatically assigned. VSS maintains these flags throughout the life cycle of a shadow copy.

Hardware Shadow Copy LUNs Must Be Read/Write

VSS supports hardware shadow copies only where the underlying LUN is mapped as read/write. This must be done before the shadow copy is created; it cannot be done after the fact. Hardware providers should not modify these flags. For more information about how VSS uses these flags, see The Shadow Copy Creation Process.

Auto-Import Hardware Shadow Copies Are Not Supported on Windows Cluster Service

Windows Cluster service cannot accommodate LUNs with duplicate signatures and partition layout. The shadow copy LUNs must be transported to a host outside the cluster. For more information, see Fast Recovery Using Transportable Shadow Copied Volumes.

Shadow Copies That Contain Dynamic Disks Must Be Transported to a Different Host

Prior to Windows Server 2008: The native support for dynamic disks cannot accommodate LUNs with duplicate signatures and configuration database contents. The shadow copy LUNs must be transported to a different host. VSS enforces this by not allowing auto-import shadow copies of dynamic disks. A requester should not import a transportable shadow copy back to the same host.

Beginning with Windows Server 2008: This limitation is removed.

Providers Are Out-of-Process

All providers must be implemented as out-of-process COM components.

Time-Critical Operations

Long operations, such as syncing plexes, should be initiated in **IVssHardwareSnapshotProvider::BeginPrepareSnapshot**. This function should start the asynchronous preparation work and return quickly. **IVssProviderCreateSnapshotSet::EndPrepareSnapshots** serves as a "rendezvous" function—the provider can wait synchronously in this method for the completion of the preparation work that was started by **BeginPrepareSnapshot**.

Providers must not add delays in **IVssProviderCreateSnapshotSet::PreCommitSnapshots**, **IVssProviderCreateSnapshotSet::CommitSnapshots**, or **IVssProviderCreateSnapshotSet::PostCommitSnapshots** as applications are frozen during these calls. **CommitSnapshots** should return within seconds, because this is called during the Flush and Hold window, and VSS Kernel Support will cancel the Flush and Hold if the release is not received within 10 seconds, which would cause VSS to fail the shadow copy creation process. If the provider takes more than a few seconds to complete this call, there is a high probability that the shadow copy creation will fail.

Careful I/O During CommitSnapshots

Hardware providers should not need to do any I/O to the volumes involved in the shadow copy during **CommitSnapshots**. If any I/O is required, providers must not initiate any I/O to an original volume while handling the **CommitSnapshots** method.

During **CommitSnapshots**, VSS Kernel Support drivers block I/O to the original volumes that are being shadow copied. If the provider uses synchronous I/O to a volume which is in the shadow copy set, then that I/O will be blocked and the shadow copy commit process will be deadlocked. The provider should not call any Win32 APIs during **CommitSnapshots** as they will have a high probability of causing I/O and a deadlock. The VSS Kernel Support timeout will break this deadlock after 10 seconds, but this will cause the shadow copy creation to fail.

If I/O must be attempted, it must be performed asynchronously. The I/O will not complete until after all providers have returned from their **CommitSnapshots** methods. In general, it is better to perform such operations in the **PostCommitSnapshots** call.

Read/Write Shadow Copy LUNs

In this version, VSS supports only read/write shadow copy LUNs even if the volumes are exposed as read-only. The reason is that the system needs to change on-disk data structures such as:

- Disk signature, which changes during the shadow copy process
- Partition-related data structures, including those indicating the partition is read-only, hidden, a shadow copy, and should not be assigned a drive letter.

Unique Page 83 Information

Both the original LUN and the newly created shadow copy LUN must have at least one unique storage identifier in the page 83 data. At least one STORAGE_IDENTIFIER with a type of 1, 2, 3, or 8, and an association of 0 must be unique on the original LUN and the newly created shadow copy LUN.

State Transitions in Shadow Copy Providers

3/5/2021 • 2 minutes to read • Edit Online

The state transition model in a shadow copy provider is simplified by the serialization of shadow copy creation from the time that **IVssBackupComponents::StartSnapshotSet** is called until the call to **IVssBackupComponents::DoSnapshotSet** returns. If another requester tries to create a shadow copy during this time, the call to **StartSnapshotSet** will fail with error **VSS_E_SNAPSHOT_SET_IN_PROGRESS**, indicating that the second requester should wait and try again.

VSS will only call **IVssProviderCreateSnapshotSet::AbortSnapshots** after the requester has called **DoSnapshotSet**, even if the shadow copy fails or is aborted before this point. This means that a provider will not receive a call to **AbortSnapshots** until after **IVssProviderCreateSnapshotSet::EndPrepareSnapshots** has been called. If a shadow copy is aborted or fails before this point, the provider is not given any indication until a new shadow copy is started. For this reason, the provider must be prepared to handle an out-of-sequence call to **IVssHardwareSnapshotProvider::BeginPrepareSnapshot** at any point. This out-of-sequence call represents the start of a new shadow copy sequence and will have a new shadow copy set ID.

Error Handling in Shadow Copy Providers

3/5/2021 • 2 minutes to read • Edit Online

VSS allows many shadow copies to exist at once, but it only allows one shadow copy set creation to be in-progress between the call to **IVssBackupComponents::StartSnapshotSet** and the return from the call to **IVssBackupComponents::DoSnapshotSet**.

No Partial Commit

If any provider fails a shadow copy on any volume or LUN in the shadow copy set, then the creation of the entire shadow copy set fails. This is by design. This design simplifies the behavioral issues associated with partial failure semantics and maintains a consistent point-in-time across all shadow copies in the set.

Reporting Fault Conditions

If a provider error or fault condition occurs, the provider must log an error to the Application event log. This includes, but is not limited to, any provider-specific errors when creating or importing a Shadow Copy Set, or resource allocation failure for copy-on-write shadow copy after creation. This logging must not take place during the time the volumes in the shadow copy set are in a frozen state.

Valid Provider Return Values

The following table lists the valid return codes for provider methods and their meanings.

VALUE	DESCRIPTION
S_OK	The method was successful.
E_OUTOFMEMORY	The provider is out of memory or other system resources.
E_INVALIDARG	One of the parameter values is not valid.
E_ACCESSDENIED	A non-privileged client has tried to call into the provider.
VSS_E_BAD_STATE	Either no provider supports the requested operation, or the operation cannot be performed on the object because the object is not in the correct state.
VSS_E_OBJECT_NOT_FOUND	A parameter refers to an object that was not found (such as a shadow copy ID, shadow copy set ID, or volume.)
VSS_E_INSUFFICIENT_STORAGE	The provider cannot perform the operation because there is not enough disk space.
VSS_E_VOLUME_NOT_SUPPORTED	No provider on this computer supports the requested operation on the volume.
VSS_E_VOLUME_NOT_SUPPORTED_BY_PROVIDER	The provider does not support the volume.

VALUE	DESCRIPTION
VSS_E_MAXIMUM_NUMBER_OF_SNAPSHOTS_REACHED	The provider has reached the maximum number of shadow copies that it can support.
VSS_E_PROVIDER_VETO	The provider has encountered an internal run-time error that does not map to another return value. The provider must also write an event into the Application event log providing the user with information on how to resolve this problem.
VSS_E_CANNOT_REVERT_DISKID	The MBR signature or GPT ID for one or more disks could not be set to the requested value. Check the Application event log for more information.

The provider should not attempt to return any other error codes.

If the provider returns an error code that is not expected (for example **S_FALSE**, **E_FAIL**, **E_UNEXPECTED**, or **E_ABORT**), VSS writes an event into the event log mentioning the provider and failed method and translates the error to **VSS_E_UNEXPECTED_PROVIDER_ERROR** before returning to the requester. This is not done for any returns from **IVssProviderCreateSnapshotSet::AbortSnapshots** or **IVssProviderNotifications::OnUnload**.

VSS Implementation Details

3/5/2021 • 2 minutes to read • Edit Online

The following topics describe VSS implementation details:

- Implementation Details for VSS Backups
- Implementation Details for VSS Restores
- Non-Default Backup and Restore Locations
- Implementation Details for Creating Shadow Copies
- Implementation Details for Using Shadow Copies

Implementation Details for VSS Backups

3/5/2021 • 2 minutes to read • Edit Online

The following topics describe implementation details for VSS backups:

- Generating a Backup Set
- Incremental and Differential Backups
- Backups without Writer Participation
- Working with Mounted Folders and Reparse Points
- Working with Alternate Paths during Backup

Generating A Backup Set

3/5/2021 • 2 minutes to read • Edit Online

A backup set is a list of all files to be backed up, their locations, and how to back them up.

A requester must make use of the files contained on the shadow-copied volumes after the **IVssBackupComponents::DoSnapshotSet** returns successfully to generate the full list of files to be backed up.

In addition, a requester must deal with the possibility that some files have *alternate paths* and that some files have been excluded.

An algorithm for selecting files to back up should go on a *writer instance* by writer instance, component-by-component basis (as will be the case during restore; see Generating a Restore Set) and might proceed by doing the following:

1. Determining the volumes that contain the writer's files and the corresponding device objects
2. Using the *file set* information (contained in **IVssWMFiledesc** objects returned by **IVssExamineWriterMetadata::GetExcludeFile**) to create a list of the explicitly excluded files, if necessary using **FindFileFirst**, **FindFileFirstEx**, and **FindNextFile**.
3. Iterating over all of a writer's components, using **IVssExamineWriterMetadata::GetComponent**. If a selectable component is selected, use *logical path* to obtain those nonselectable components associated with it in a component set. (See Working with Selectability and Logical Paths.)
4. Obtaining the *file sets* contained in each selected component by using the **IVssWMComponent** interface corresponding to each component it contains.
5. Generating a list of files from the specifications—if necessary using **FindFileFirst**, **FindFileFirstEx**, and **FindNextFile**.
6. Checking each file in the list generated from component information against the list of excluded files generated above. This should be done using the default path for the file (returned by **IVssWMFiledesc::GetPath**), not by the alternate path returned by **IVssWMFiledesc::GetAlternateLocation**). If the file matches the excluded list, it will not be backed up.
7. Choosing the actual location from which to back up (using the alternate path if it was set)
8. At this point, a full list of files and their locations is available and a backup can begin.

After an initial backup set has been generated for all writers that are present on the system, the requester then checks the following registry key:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\BackupRestore\FilesNotToBackup

The requester uses the subkeys under this key as follows:

- If a writer is present on the system, and there is a subkey whose name matches the writer's name, that subkey must be ignored.
- If a writer was present on the system but is currently absent from the backup set, and there is a matching subkey, any files specified in the subkey data are excluded and must be removed from the backup set.
- The backup application adds files to the subkey data by creating a MULTI_SZ value containing a list of file specifications for the files that must not be backed up. Each string in the MULTI_SZ value should contain one file specification.
- File specifications can contain the ? and * wildcard characters. A specification can be made recursive by appending /s to the end. For example, specifying "%TEMP%* /s" causes all files in the %TEMP% directory and all of its subdirectories not to be backed up.

Incremental and Differential Backups

6/16/2021 • 2 minutes to read • Edit Online

As with all important operations under VSS, *incremental* and *differential* backups require close cooperation between requesters and writers.

- Writer Role in VSS Incremental and Differential Backups
- Requester Role in VSS Incremental and Differential Backups
- Writer Role in Backing Up Complex Stores
- Requester Role in Backing Up Complex Stores

Writer Role in VSS Incremental and Differential Backups

3/5/2021 • 5 minutes to read • Edit Online

A writer's participation in incremental and differential backups typically takes place while handling *Identify* (**CVssWriter::OnIdentify**), *PrepareForBackup* (**CVssWriter:OnPrepareBackup**), and *PostSnapshot* (**CVssWriter:OnPostSnapshot**) events. How a writer participates is shaped by whether it supports backup stamps and last modification times, and whether the requester running the backup supports *partial file operations*.

Handling Identify Events during Incremental and Differential Backups

While handling the Identify event, writers establish their basic architecture for incremental and differential backup operation through the backup schema (**VSS_BACKUP_SCHEMA**) and file specification backup type (**VSS_FILE_SPEC_BACKUP_TYPE**) masks.

A writer indicates which operations it supports in its Writer Metadata Document by creating a bit mask of **VSS_BACKUP_SCHEMA** values and passing it to the **IVssCreateWriterMetadata::SetBackupSchema** method. With this, a writer can indicate whether it supports the following:

- Incremental backups (**VSS_BS_INCREMENTAL**)
- Differential backups (**VSS_BS_DIFFERENTIAL**)
- Incremental and differential backups cannot be mixed (**VSS_BS_EXCLUSIVE_INCREMENTAL_DIFFERENTIAL**)
- Incremental and differential backups using backup stamps (**VSS_BS_TIMESTAMPED**)
- Incremental and differential backups on the basis of information about a file's last modification (**VSS_BS_LAST_MODIFY**)

Writers use the file specification backup type mask to provide file set level information to requesters on how to include files in an incremental or differential backup.

A writer can set a file set's file specification backup type mask when it adds the file set to a component by creating a bit mask of **VSS_FILE_SPEC_BACKUP_TYPE** values and passing it to **IVssCreateWriterMetadata::AddDatabaseFiles**, **IVssCreateWriterMetadata::AddDatabaseLogFiles**, or **IVssCreateWriterMetadata::AddFilesToFileGroup**.

There are three "backup required" values of the **VSS_FILE_SPEC_BACKUP_TYPE** enumeration that affect differential and incremental backups:

- **VSS_FSBT_ALL_BACKUP_REQUIRED**
- **VSS_FSBT_INCREMENTAL_BACKUP_REQUIRED**
- **VSS_FSBT_DIFFERENTIAL_BACKUP_REQUIRED**

There are three "shadow copy required" values:

- **VSS_FSBT_ALL_SNAPSHOT_REQUIRED**
- **VSS_FSBT_INCREMENTAL_SNAPSHOT_REQUIRED**
- **VSS_FSBT_DIFFERENTIAL_SNAPSHOT_REQUIRED**

File sets tagged with a file specification backup type of "shadow copy required" indicate whether a requester needs to copy data from a shadow copy when performing INCREMENTAL, DIFFERENTIAL, or ALL (which includes

both incremental and differential operations) backup operations.

The "backup required" flag, applied to INCREMENTAL, DIFFERENTIAL, or ALL backup operations, indicates that the writer expects a copy of the current version of the file set to be available following the restore of any backup operation. Typically, this means that if a file set is tagged with "backup required," all its members will be copied to backup media during an incremental or differential backup, regardless of when their backup or modification last occurred.

By default, file sets are added to components with a file specification backup type of **VSS_FSBT_ALL_BACKUP_REQUIRED | VSS_FSBT_ALL_SNAPSHOT_REQUIRED**. Therefore, unless a writer developer decides not to use the default (by choosing another file specification backup type, using partial file operations, or using differenced files), the files in most file sets will typically be copied in their entirety to backup media.

At this point, the writer's Writer Metadata Document is fully populated with most of information a requester will need to start a differential or incremental backup. Addition specification of file set or file level information to support the backup will be handled during the **PrepareForBackup** event.

Handling PrepareForBackup Events during Incremental and Differential Backups

Before the requester proceeds with the actual backup operation, writers can modify the specification of an *incremental* or *differential* backup by altering the requester's Backup Components Document through the **IVssComponent** interface.

Because the writers use the **IVssComponent** interface, they typically perform these preparations while handling the **PrepareForBackup** event.

In **CVssWriter:OnPrepareBackup**, writers can more precisely specify how some files are to be evaluated for backup, specify what mechanisms should be used in backing them up, and possibly set backup stamps.

Partial Files

If a requester supports them, a writer can choose to have an incremental or differential backup implemented by using partial file operations. (Writers can determine if a requester supports partial file operations by calling **CVssWriter::IsPartialFileSupportEnabled**.)

Writers use **IVssComponent::AddPartialFile** to indicate those portions of the selected files to be backed up during the incremental or differential operation. Requesters must respect this specification, and must always back up the specified sections of the files. (See Working with Partial Files for more information on partial file operations.)

Using **IVssComponent::AddPartialFile**, a writer can add a file to the backup that was not previously added to one of its component sets (by **IVssCreateWriterMetadata::AddDatabaseFiles**, **IVssCreateWriterMetadata::AddDatabaseLogFiles**, or **IVssCreateWriterMetadata::AddFilesToFileGroup**) as a partial file. Any new files added to the backup in this manner must be on a volume that already is being shadow copied for this backup.

If a file is involved with partial file operations, this supersedes any file specification backup type, which is ignored.

Differenced Files

Writers that support a last modified backup schema (**VSS_BS_SCHEMA**) can add *differenced files* to an incremental or differential backup.

In specifying a differenced file, a writer uses **IVssComponent::AddDifferencedFileByLastModifyTime** and

specifies a path, a file name, and a recursive flag—however, these do not have to match a file set included in any component.

In fact, a writer can add a file not previously added to one of its component sets (by **IVssCreateWriterMetadata::AddDatabaseFiles**, **IVssCreateWriterMetadata::AddDatabaseLogFiles**, or **IVssCreateWriterMetadata::AddFilesToFileGroup**) to the backup as a differenced file. Any new files added to the backup in this manner must be on a volume that already is being shadow copied for this backup.

Typically, a writer will also specify a time of last modification when adding a differenced file—based on the writer's own history mechanism. This last modification time, if specified, must always be used by requesters in determining if a file needs to be included in an incremental or differential backup.

A writer may choose not to specify a last modification time when adding a differenced file to an incremental or differential backup set. If this is the case, requesters are free to use their own mechanisms—for instance, logs of previous backups or file system information—to determine whether the differenced file should be included in an incremental or differential backup.

The file specification backup type of any differenced file is ignored.

Backup Stamps

Writers that support backup stamps (**VSS_BS_TIMESTAMP**) have their own private format for storing information about when a backup last occurred. This information is generated by the writer during backup.

The backup stamp is stored in the Backup Components Document as a string by the **IVssComponent::SetBackupStamp** method. The backup stamp applies to all file sets in the component (or component set) corresponding to the instance of the **IVssComponent**.

If a requester has access to the backup stamp of a previous backup, it will have made it available to the writer by calling **IVssBackupComponents::SetPreviousBackupStamp**.

A writer then can examine this time stamp using **IVssComponent::GetPreviousBackupStamp**.

Note that the requester merely stores and returns the string containing the backup stamp. It does not know anything about the string's format or how to use it; only the writer has that information.

A writer may choose to update the current backup stamp using **IVssComponent::SetBackupStamp** after it has called **IVssComponent::GetPreviousBackupStamp**, thus recording in its own format the date of the current incremental or differential backup operation.

Requester Role in VSS Incremental and Differential Backups

3/5/2021 • 8 minutes to read • Edit Online

To support an *incremental* or *differential* backup operation, a requester must do the following:

1. Determine what degree of writer support is available (using **IVssBackupComponents::GetWriterMetadata** to get access to information in Writer Metadata Documents)—in particular, determine which backup schema are supported (**VSS_BACKUP_SCHEMA**).
2. Set an appropriate backup state.
3. Obtain file and file set level specifications for an incremental or differential backup.
4. Perform the backup.

Requester Determination of Incremental and Differential Support and Configuration

A requester needs to obtain information about writer support prior to selecting components for inclusion in an incremental or differential backup or setting its own state.

Determining Writer Support

A requester determines if a given writer supports VSS incremental or differential backups by retrieving the writer's backup schema mask using the **IVssExamineWriterMetadata::GetBackupSchema** method.

The backup schema mask of a writer supporting VSS incremental or differential techniques will contain either **VSS_BS_INCREMENTAL** or **VSS_BS_DIFFERENTIAL**, or both. Writers may also indicate restrictions on their participation with the **VSS_BS_EXCLUSIVE_INCREMENTAL_DIFFERENTIAL** flag. (See **VSS_BACKUP_SCHEMA** for more information on backup schemas).

Setting Requester Backup State

A requester indicates that a backup is an incremental or differential backup by setting a backup type to either **VSS_BT_INCREMENTAL** or **VSS_BT_DIFFERENTIAL** using the **IVssBackupComponents::SetBackupState** method prior to generating a **PrepareForBackup** event.

The **IVssBackupComponents::SetBackupState** method is also used to indicate whether the requester provides *partial file support*, which is frequently used to implement certain incremental backup and restore operations.

Getting Writer Specifications for Incremental and Differential Backups

The file-set-level file backup specification information (**VSS_FILE_SPEC_BACKUP_TYPE**) contained in each writer's Writer Metadata Document is available for examination after the successful return of **IVssBackupComponents::GatherWriterMetadata**.

However, a writer can add *differenced files* or ask for *partial file support* until its successful handling of the *PostSnapshot* event.

Differenced file and partial file support specification can override the file specification backup type, so requesters may want to defer a full analysis of all writer specifications about incremental and differential backups until after the successful return of **IVssBackupComponents::PrepareForBackup**.

Getting File Backup Specification Information

The file-set-level file backup specification information (**VSS_FILE_SPEC_BACKUP_TYPE**) is contained in each writer's Writer Metadata Document, and can be examined immediately after the successful return of **IVssBackupComponents::GatherWriterMetadata**.

Requesters must obtain file backup specification masks (**VSS_FILE_SPEC_BACKUP_TYPE**) for every file set of each of a writer's components to be included in the incremental or differential backup, regardless of whether the component was *explicitly* or *implicitly* included.

A requester can determine which writers' Writer Metadata Document must be queried by using **IVssBackupComponents::GetWriterComponentsCount** and **IVssBackupComponents::GetWriterComponents**. The instance of the **IVssWriterComponentsExt** interface returned by **IVssBackupComponents::GetWriterComponents** provides writer information through the **IVssWriterComponentsExt::GetWriterInfo** method.

The requester obtains component information through instances of the **IVssWMComponent** interface corresponding to an included component managed by a given writer by using **IVssExamineWriterMetadata::GetComponent**.

The information about file sets managed by the component corresponding to the **IVssWMComponent** interface is obtained by calls to **IVssWMComponent::GetFile**, **IVssWMComponent::GetDatabaseFile**, or **IVssWMComponent::GetDatabaseLogFile** (as appropriate).

These calls can return instances of the **IVssWMFiledesc** interface for each of a component's file sets.

A file set's file specification backup type is obtained by calling **IVssWMFiledesc::GetBackupTypeMask**.

Getting Partial File and Differenced File Information

A requester obtains partial file and differenced file information through the **IVssComponent** interface.

A requester can iterate over all writers included in a backup using **IVssBackupComponents::GetWriterComponentsCount** and **IVssBackupComponents::GetWriterComponents**.

The instance of an **IVssWriterComponentsExt** interface returned by **IVssBackupComponents::GetWriterComponents** provides access to all instances of the **IVssComponent** interface corresponding to a given writer's *explicitly included* components through the **IVssWriterComponentsExt::GetComponent** and **IVssWriterComponentsExt::GetComponentCount** methods.

A requester will need to go through all instances of **IVssComponent** for all writers whose schema support the incremental or differential backup—that is, writers whose backup schema mask, as returned by **IVssExamineWriterMetadata::GetBackupSchema**, includes **VSS_BS_INCREMENTAL** when the backup type is **VSS_BT_INCREMENTAL**, or **VSS_BS_DIFFERENTIAL** when the backup type is **VSS_BS_DIFFERENTIAL**.

Partial file information is obtained by calling **IVssComponent::GetPartialFileCount** and **IVssComponent::GetPartialFile** (see Working with Partial Files).

For writers that support backup operations on the basis of a file's last modification data (writers whose backup schema mask, as returned by **IVssExamineWriterMetadata::GetBackupSchema**, includes **VSS_BS_LAST_MODIFY**), differenced file information is obtained by calling **IVssComponent::GetDifferencedFilesCount** and **IVssComponent::GetDifferencedFile**.

Note that differenced files may be new files—that is, files that are not members of any file set currently in a given writer's Writer Metadata Document.

Requesters should not find files included both for partial file operations and as differenced files. If a requester

does encounter such a circumstance, it should return and log a writer error.

A requester may still choose to proceed with backing up the problematic writer's files, but in that case should do so according to the specification found in the differenced file information.

Implementing Incremental or Differential Backups

Prior to implementing a backup, requesters should have information about which writers support an *incremental* or *differential* backup, all requested partial file operations, all differenced files, and the file specification backup type of all other files.

Nonsupporting Writers

Writers whose schema do not support the incremental or differential backup (writers whose backup schema mask, as returned by **IVssExamineWriterMetadata::GetBackupSchema**, includes **VSS_BS_INCREMENTAL** when the backup type is **VSS_BT_INCREMENTAL** or does not include **VSS_BS_DIFFERENTIAL** when the backup type is **VSS_BS_DIFFERENTIAL**) cannot provide any direct support to an incremental or differential backup operation.

This does not necessarily mean that the writers' data will not be involved in an incremental or differential backup operation. However, the choice of what to do is at the discretion of the requester. The requester can do any of the following:

- Back up no files belonging to the nonsupporting writers (clearly indicate this to the user)
- Back up all the files of nonsupporting writers
- Perform an incremental backup using file system data and the requester's own history logs.

The last alternative should be used with great care, and only if the requester understands if the writers involved can support incremental or differential backup and restoration of data independent of the VSS mechanism.

Supporting Writers

A requester needs to process (in order) all of a writer's *differenced files*, then handle any *partial file* requests, and then back up remaining files according to their file specification backup type (**VSS_FILE_SPEC_BACKUP_TYPE**).

1. **Backing up Differenced Files:**

   For writers that support backup operations on the basis of last modification data (writers whose backup schema mask, as returned by **IVssExamineWriterMetadata::GetBackupSchema**, includes **VSS_BS_LAST_MODIFY**), a requester uses the path, file specification, and recursion flag information returned by **IVssComponent::GetDifferencedFile** to generate a list of files as candidates for incremental backup or restore.

   IVssComponent::GetDifferencedFile can also return a time of last modification (expressed as a **FILETIME** structure).

   If the last modification time supplied by the writer is nonzero, then the requester uses it as the basis (rather than file system information or the requester's own stored data) for determining if the file should be included in the *incremental* or *differential* backup.

   For instance, if a file's last modification time as returned by the writer was:

   - After the last full backup, the file should be included in both incremental and differential backups.
   - After the last full backup but prior to the last incremental backup, the file should be included in incremental backup operations, but not differential backups.

   If the last modification time supplied by the writer is zero, then the requester must use file system information and its own stored data to determine the modification time of the differenced file.

2. **Using Partial File Operations:**

   If a writer has requested that a file be backed up using a partial file operation, the requester uses the file offset information to save the indicated file segments to backup media. (See Working with Partial Files for more information on partial file operations).

   As noted above, a writer should not designate a file both as a differenced file and as a participant in a partial file operation. If a requester does encounter such a circumstance, it should return and log a writer error.

   A requester may still choose to proceed with backing up the problematic writer's files, but in that case should do so according to the specification found in the differenced file information.

3. **Working with File Specification Backup Type:**

   Having processed all differenced files and partial file operations, the requester now processes all remaining files in its backup set on the basis of their file specification backup type (**VSS_FILE_SPEC_BACKUP_TYPE**).

   There are three "backup required" values of the **VSS_FILE_SPEC_BACKUP_TYPE** enumeration that affect differential and incremental backups:

   - VSS_FSBT_ALL_BACKUP_REQUIRED
   - VSS_FSBT_INCREMENTAL_BACKUP_REQUIRED
   - VSS_FSBT_DIFFERENTIAL_BACKUP_REQUIRED

   There are three "shadow copy required" values:

   - VSS_FSBT_ALL_SNAPSHOT_REQUIRED
   - VSS_FSBT_INCREMENTAL_SNAPSHOT_REQUIRED
   - VSS_FSBT_DIFFERENTIAL_SNAPSHOT_REQUIRED

   File sets tagged with a file specification backup type of "shadow copy required" indicate that a requester needs to copy data from a shadow copy when performing INCREMENTAL, DIFFERENTIAL, or ALL (which includes both incremental and differential operations) backup operations.

   The "backup required" flag, applied to INCREMENTAL, DIFFERENTIAL, or ALL backup operations, indicates that the writer expects a copy of the current version of the file set to be available following the restore of any backup operation. Typically, this means that if a file set is tagged with "backup required," a requester will copy all its members to backup media during an incremental or differential backup, regardless of when their backup or modification last occurred.

   By default, file sets are added to components with a file specification backup type of VSS_FSBT_ALL_BACKUP_REQUIRED | VSS_FSBT_ALL_SNAPSHOT_REQUIRED. Therefore, unless a writer explicitly sets the file specification backup type otherwise, requesters will need to copy those files not handled by partial file operations or designated differenced files in most file sets will typically be copied in their entirety to backup media.

Backup Stamps

Writers that support backup stamps (VSS_BS_TIMESTAMP) may choose to generate backup stamp information to be used to support future incremental and differential backup and restore operations.

The format and information contained in strings containing backup stamp information are private to the writer that generates them; the requester does not know how to process this information.

Supporting writers store the backup stamp in the Backup Components Document as a string by using the **IVssComponent::SetBackupStamp** method.

The requester's role in handling backup stamp information is (if it exists) to make it available to the writer by

calling **IVssBackupComponents::SetPreviousBackupStamp** in a future backup or restore operation.

Writer Role in Backing Up Complex Stores

6/16/2021 • 11 minutes to read • Edit Online

As with all important operations under VSS, *incremental* and *differential* backups require close cooperation between requesters and writers.

Backup Types

The infrastructure provides special support for five types of backup. They are described as follows:

- Full (**VSS_BT_FULL**). Files will be backed up regardless of their last backup date. The backup history of each file will be updated, and this type of backup can be used as the basis of an incremental or differential backup. If there are log files, they may be truncated as a result of this backup.

  Restoring a full backup requires only a single backup image.

- Differential (**VSS_BT_DIFFERENTIAL**). The VSS API is used to ensure that only files that have been changed or added since the last full backup are to be copied to a storage medium; all intermediate backup information is ignored. This may include entire files, or specific ranges within files. A differential backup is associated with a full backup, and generally cannot be restored until the full backup has been restored. If there are log files, they will usually not be truncated as a result of this backup.

  Restoring a differential backup requires the original backup image and the most recent differential backup image made since the last full backup.

- Incremental (**VSS_BT_INCREMENTAL**). The VSS API is used to ensure that only files that have been changed or added since the last full or incremental backup are to be copied to a storage medium. This may include entire files, or specific ranges within files. Some writers do not allow incremental backups to be mixed with differential backups. If there are log files, they may be truncated as a result of this backup.

  Restoring an incremental backup requires the original backup image and all incremental backup images made since the initial backup.

- Log Backup (**VSS_BT_LOG**). Only a writer's log files (files added to a component with the **IVssCreateWriterMetadata::AddDataBaseLogFiles** method, and retrieved by a call to **IVssWMComponent::GetDatabaseLogFile**) will be backed up. This backup type is specific to VSS. Log backups tend to be taken quite frequently. Typically, the log file will be truncated as a result of this backup.

- Copy Backup (**VSS_BT_COPY**). Like the VSS_BT_FULL backup type, files will be backed up regardless of their last backup date. However, the backup history of each file will not be updated, and this type of backup cannot be used as the basis of an incremental or differential backup. Log files should never be truncated as a result of a copy backup.

Partial File Support

Some writers support file restoration through the overwriting of parts of the files they manage. A requester may be designed to take advantage of this, and if so it indicates this by setting the information in **IVssBackupComponents::SetBackupState**.

The writers indicate what type of backups are supported by calling **IVssCreateWriterMetadata::SetBackupSchema** while processing the *Identify* event. The *dsSchemaMask* parameter to the **IVssCreateWriterMetadata::SetBackupSchema** method is a bit mask indicating what types of backup are supported. All writers must support full backups.

VSS_BS_DIFFERENTIAL

Indicates support for differential backups.

VSS_BS_INCREMENTAL

Indicates support for incremental backups.

VSS_BS_LOG

Indicates support for log backups.

VSS_BS_COPY

Indicates support for copy backups.

VSS_BS_EXCLUSIVE_INCREMENTAL_DIFFERENTIAL

Indicates that a writer does not support mixing incremental backups with differential backups.

The writer can determine what type of backup is being performed by calling **CVssWriter::GetBackupType**. The earliest point when this can be performed is while processing the PrepareForBackup event. **CVssWriter::GetBackupType** will return a member of the **VSS_BACKUP_TYPE** enumeration. If the backup type is not supported by the writer, then the writer should treat the backup as a full backup.

Backup Stamps

Incremental and differential backups are always tied to a previous backup. There are two ways to tie backups. For simple data stores, the requester can keep track of the correlation between backups. However, for more complex data stores, the writer will need to maintain its own timestamp with the backup; this timestamp may keep track of log position, checkpoint information, and so on. A writer indicates that it needs its own timestamps by setting the **VSS_BS_TIMESTAMPED** bit when it calls **IVssCreateWriterMetadata::SetBackupSchema**.

A writer can store a timestamp with each component that is being backed up. The writer stores the timestamp by calling **IVssComponent::SetBackupStamp**, and passing in a string representation of the stamp for the *wszBackupStamp* parameter. Generally, a writer will call this method while processing the *PostSnapshot* event. However, for backups that do not involve a shadow copy, the PostSnapshot event will not be sent. In this case, **IVssComponent::SetBackupStamp** must be called while processing the *PrepareForBackup* event.

When an incremental or differential backup is being performed, the requester will indicate to the writer the backup stamp of the previous backup that is serving as a base for this backup. The writer can access this previous backup stamp while processing either the PrepareForBackup or PostSnapshot event, by calling **IVssComponent::GetPreviousBackupStamp**. The writer can use the returned stamp to determine what needs to be backed up.

Backup Strategies

File Backup Files Strategies

Often, certain files reported in the writer metadata need only be backed up when performing certain types of backup. Some files may only be required when performing a full backup. Other files may only be required when performing an incremental or differential backup. VSS provides a method for the writers to indicate this information to the requester. When adding files to components using **IVssCreateWriterMetadata::AddDatabaseFiles**, **IVssCreateWriterMetadata::AddDatabaseLogFiles**, or **IVssCreateWriterMetadata::AddFilesToFileGroup**, the *dwBackupTypeMask* parameter indicates for which backup types these files must be backed up. The mask can contain one or more of the following values:

VSS_FSBT_FULL_BACKUP_REQUIRED

Required for full backups.

VSS_FSBT_DIFFERENTIAL_BACKUP_REQUIRED

Required for differential backups.

VSS_FSBT_INCREMENTAL_BACKUP_REQUIRED

Required for incremental backups.

VSS_FSBT_LOG_BACKUP_REQUIRED

Required for log backups.

VSS_FSBT_ALL_BACKUP_REQUIRED

Required for all backup types; this is the default.

This specification overrides the component's selectivity specification. For example, consider a component whose files are all marked with **VSS_FSBT_LOG_BACKUP_REQUIRED** but not with **VSS_FSBT_FULL_BACKUP_REQUIRED**. Suppose this component is not selectable for backup (*bSelectable* was false when **IVssCreateWriterMetadata::AddComponent** was called). In the case of a log backup, this means that all the files in this component must always be backed up. However, in the case of a full backup, none of the files need to be backed up, despite the fact that the component's selectivity implies it should be backed up.

Backup By Last Modify Time

One way for a writer to indicate what files have changed is by using the differenced file mechanism. A writer can specify that certain files in a component should only be backed up if they have been modified since a certain time. The writer calls **IVssComponent::AddDifferencedFilesByLastModifyTime** with a file specification and a last modify time. **IVssComponent::AddDifferencedFilesByLastModifyTime** is typically called while processing the PostSnapshot event, although it can be called while processing the PrepareForBackup event. The requester must then back up all files matching the file specification that have changed since the specified time. If the writer is using the backup stamp mechanism, this last modify time will be determined based on the previous backup stamp in the backup document. The writer can also pass in zero for the last modify time, which indicates that the requester is responsible for determining the time of the last backup and the files changed since that time.

Partial File Backup

Another way for a writer to indicate changes to the requester is by using the partial-file mechanism. A writer can specify byte ranges within component files that need to be backed up; the writer may specify these file ranges while processing either the PostSnapshot or PrepareForBackup event. The writer calls **IVssComponent::AddPartialFile** to add partial file specifications to the backup. A partial file specification consists of a path and file name together with information about what ranges in the file need to be backed up.

File Specification Rules

IVssComponent::AddDifferencedFilesByLastModifyTime or **IVssComponent::AddPartialFile** can both be used to modify file specifications given during the Identify event, or to add completely new files to the specification. If the writer is modifying information set during the Identify event using **IVssComponent::AddDifferencedFilesByLastModifyTime**, then the file specification must exactly match one of the file specifications in the current component. The file specification must not partially overlap files in the current component, and it must not match files in any other components. Files specified using **IVssComponent::AddPartialFile** can, however, partially overlap another file specification. Information set by **IVssComponent::AddDifferencedFilesByLastModifyTime** or **IVssComponent::AddPartialFile** overrides the information set earlier using the **IVssCreateWriterMetadata** interface in response to the Identify event.

General file specifications can have an alternate-location value (set by the *wszAlternateLocation* parameter of **IVssCreateWriterMetadata::AddFilesToFileGroup**) that indicates an alternate location to obtain the file from

at backup time. If the file specification set through the differenced-file or partial-file mechanisms matches an existing file specification that has an alternate location, the backup application will obtain the data from this alternate location.

If the file specification set in **IVssComponent::AddDifferencedFilesByLastModifyTime** or in **IVssComponent::AddPartialFile** does not match and files in the component that are being backed up, then all matching files are now added to the backup. Care must be taken that the writer only adds files that exist on a volume that is already being shadow copied while doing this; otherwise, the requester may fail to back up these files. If these functions are called while processing the PostSnapshot event, this can be determined by using the **CVssWriter::IsPathAffected** method. If called while handling the PrepareForBackup event, the writer must make this determination using another method.

Backup Without a Shadow Copy

Certain types of files may not need to be backed up off of a shadow copy volume. For example, this will often be true of database log files. Since log files grow monotonically, and a writer can specify exactly what portions of the file to back up using partial files, it will often be possible to back up the log off the original volume. As an optimization, a writer can mark which files require shadow copies for different backup types using the flags set in the *dwBackupTypeMask* parameter of **IVssCreateWriterMetadata::AddDatabaseFiles**, **IVssCreateWriterMetadata::AddDatabaseLogFiles**, or **IVssCreateWriterMetadata::AddFilesToFileGroup**. Supported flags include the following:

VSS_FSBT_FULL_SNAPSHOT_REQUIRED

Shadow copy required for full backups.

VSS_FSBT_DIFFERENTIAL_SNAPSHOT_REQUIRED

Shadow copy required for differential backups.

VSS_FSBT_INCREMENTAL_SNAPSHOT_REQUIRED

Shadow copy required for incremental backups.

VSS_FSBT_LOG_SNAPSHOT_REQUIRED

Shadow copy required for log backups.

VSS_FSBT_ALL_SNAPSHOT_REQUIRED

Shadow copy required for all backup types; this is the default.

If a specific volume contains only components that do not require a shadow copy for this backup, the requester can skip the step of creating a shadow copy for this volume. All the data on this volume can be copied to the backup media directly from the original volume.

Backup Cleanup

If the writer needs to perform log truncation or other post-backup cleanup, the proper place to do this is while processing the *BackupComplete* event. The *BackupShutdown* event will be sent some time after BackupComplete, so some cleanup may also be done in the BackupShutdown event handler.

The BackupShutdown event is always sent after termination of a backup. If the requester terminates abnormally while performing a backup, BackupShutdown will be sent immediately, without first sending BackupComplete. If the writer needs to clean up any state, that may be done here; however, log truncation should not happen in this event because the backup did not necessarily complete.

Restore Strategies

The basic tasks of writers at restore are to verify that the restore can happen in handling the PreRestore event, and that the restore has happened in handling the PostRestore event. More complex stores will also perform a recovery process in the PostRestore handler. If the restore is part of an incremental or differential restore, the writer will generally want to delay this recovery process until all incremental or differential restores have been completed. **IVssComponent::GetAdditionalRestores** will indicate whether this is the final restore of this component, or whether there are more restores to come. If **IVssComponent::GetAdditionalRestores** returns **true**, the writer should not perform its recovery procedure on that component.

New Targets

If supported by the writer, the requester can restore data files to a location other than the original backup-time location. A writer indicates support for this restore mode by setting the **VSS_BS_WRITER_SUPPORTS_NEW_TARGET** bit in the *dsSchemaMask* parameter when calling **IVssCreateWriterMetadata::SetBackupSchema**. A writer obtains the new locations for component files at restore time by calling **IVssComponent::GetNewTargetCount** and **IVssComponent::GetNewTarget**.

Directed Targets

For complicated restore scenarios, a writer may want to map ranges of a backed-up file onto different ranges of the same or a different file. This can be done by using the directed-target mechanism. To do this, a writer must first indicate this is happening by calling **IVssComponent::SetRestoreTarget**, passing in **VSS_RT_DIRECTED** for the *target* parameter. Then, for each mapping, the writer calls **IVssComponent::AddDirectedTarget**. This method takes a full path to a source file on the backup and a full path to a destination file that will be restored to. It also takes a ranges list for each of these files. The writer calls these functions while handling the PreRestore event, and the requester is then responsible for restoring the specified ranges in the source file to the mapped ranges in the destination file. The format of the ranges string is the same as in **IVssComponent::AddPartialFile**

Private Writer Metadata

It is often useful for a writer to maintain private metadata with a backup to properly perform an incremental or differential restore. A writer may call **IVssComponent::SetBackupMetadata** while handling either PrepareForBackup or PostSnapshot to store metadata. This metadata can be accessed by the writer during either PreRestore or PostRestore by calling **IVssComponent::GetBackupMetadata**. Metadata can also be stored with partial file specification by using the *wszMetadata* parameter of **IVssComponent::AddPartialFile**; this metadata is accessed through the *pbstrMetadata*parameter of **IVssComponent::GetPartialFile**. The writer can also pass metadata to itself between **CVssWriter::OnPreRestore** and **CVssWriter::OnPostRestore**. In **CVssWriter::OnPreRestore**, the metadata is set by calling **IVssComponent::SetRestoreMetadata**. In **CVssWriter::OnPostRestore**, the metadata is retrieved by calling **IVssComponent::GetRestoreMetadata**.

- Writer Role in VSS Incremental and Differential Backups
- Requester Role in VSS Incremental and Differential Backups

Requester Role in Backing Up Complex Stores

6/16/2021 • 10 minutes to read • Edit Online

As with all important operations under VSS, *incremental* and *differential* backups require close cooperation between requesters and writers.

Backup Type

The infrastructure provides special support for five types of backup. They are described as follows:

- Full (**VSS_BT_FULL**). Files will be backed up regardless of their last backup date. The backup history of each file will be updated, and this type of backup can be used as the basis of an incremental or differential backup. If there are log files, they may be truncated as a result of this backup.

  Restoring a full backup requires only a single backup image.

- Differential (**VSS_BT_DIFFERENTIAL**). The VSS API is used to ensure that only files that have been changed or added since the last full backup are to be copied to a storage medium; all intermediate backup information is ignored. This may include entire files, or specific ranges within files. A differential backup is associated with a full backup, and generally cannot be restored until the full backup has been restored. If there are log files, they will usually not be truncated as a result of this backup.

  Restoring a differential backup requires the original backup image and the most recent differential backup image made since the last full backup.

- Incremental (**VSS_BT_INCREMENTAL**). The VSS API is used to ensure that only files that have been changed or added since the last full or incremental backup are to be copied to a storage medium. This may include entire files, or specific ranges within files. Some writers do not allow incremental backups to be mixed with differential backups. If there are log files, they may be truncated as a result of this backup.

  Restoring an incremental backup requires the original backup image and all incremental backup images made since the initial backup.

- Log Backup (**VSS_BT_LOG**). Only a writer's log files (files added to a component with the **IVssCreateWriterMetadata::AddDataBaseLogFiles** method, and retrieved by a call to **IVssWMComponent::GetDatabaseLogFile**) will be backed up. This backup type is specific to VSS. Log backups tend to be taken quite frequently. Typically, the log file will be truncated as a result of this backup.

- Copy Backup (**VSS_BT_COPY**). Like the VSS_BT_FULL backup type, files will be backed up regardless of their last backup date. However, the backup history of each file will not be updated, and this sort of backup cannot be used as the basis of an incremental or differential backup. Log files should never be truncated as a result of a copy backup.

Partial File Support

Some writers support file restoration through the overwriting of parts of the files they manage. A requester may be designed to take advantage of this, and if so it indicates this by setting the information in **IVssBackupComponents::SetBackupState**.

The requester specifies what type of backup is being performed through the *backupType* parameter of **IVssBackupComponents::SetBackupState**. Different writers will support different types of backup. After **IVssBackupComponents::GatherWriterMetadata** has been called, the requester can determine what types of backup a given writer supports by calling **IVssExamineWriterMetadata::GetBackupSchema**. The returned

value is a bit mask indicating support for different backup types. **VSS_BS_DIFFERENTIAL** indicates support for differential backups, **VSS_BS_INCREMENTAL** for incremental backups, **VSS_BS_LOG** for log backups, and **VSS_BS_COPY** for copy backups; all writers must support full backups. If a writer does not support mixing incremental backups with differential backups, the **VSS_BS_EXCLUSIVE_INCREMENTAL_DIFFERENTIAL** flag will be added as well. If the requester is performing an incremental or differential backup and a given writer does not support that backup type, a full backup should be performed on that writer.

Backup Stamps

Incremental and differential backups are always tied to a previous backup. There are two ways to tie backups. For simple data stores, the requester can keep track of the correlation between backups. However, for more complex data stores, the writer will need to maintain its own timestamp with the backup; this timestamp may keep track of log position, checkpoint information, and so on. The requester can determine whether a given writer needs to store its own timestamp by checking for the **VSS_BS_TIMESTAMPED** bit in the value returned by **IVssExamineWriterMetadata::GetBackupSchema**.

Writers that store a timestamp in a backup will add the timestamp either while processing **IVssBackupComponents::PrepareForBackup** or while processing **IVssBackupComponents::DoSnapshotSet**. The requester can obtain this timestamp by calling **IVssComponent::GetBackupStamp**. When performing an incremental or differential backup, the requester needs to tie the current backup to some previous backup. This is done by obtaining the timestamp from the previous backup of a specific component and passing it into the **IVssBackupComponents::SetPreviousBackupStamp** function; this needs to be done for each component that was backed up in the previous backup.

Backing Up Files

Backing Up Files Reported by Writer

Every file specification that a writer adds to its metadata during the GatherWriterMetadata phase contains a backup type mask that specifies when the file should be backed up. The requester determines what this mask is by calling **IVssWMFiledesc::GetBackupTypeMask**. The values in this mask are used to determine for which backup types the file specification needs to be backed up. The mask can contain one or more of the following bit values:

VSS_FSBT_FULL_BACKUP_REQUIRED

Required for full backups.

VSS_FSBT_DIFFERENTIAL_BACKUP_REQUIRED

Required for differential backups.

VSS_FSBT_INCREMENTAL_BACKUP_REQUIRED

Required for incremental backups.

VSS_FSBT_LOG_BACKUP_REQUIRED

Required for log backups.

VSS_FSBT_ALL_BACKUP_REQUIRED

Required for all backup types; this is the default.

This specification overrides the component's selectivity specification. For example, consider a component whose files are all marked with **VSS_FSBT_LOG_BACKUP_REQUIRED** but not with **VSS_FSBT_FULL_BACKUP_REQUIRED**. Suppose this component is not selectable for backup (*bSelectable* was

false when IVssCreateWriterMetadata::AddComponent was called). In the case of a log backup, this means that all the files in this component must always be backed up. However, in the case of a full backup, none of the files need to be backed up, despite the fact that the component's selectivity implies it should be backed up.

Backup by Last Modify Time

The file specification information specified in the IVssBackupComponents::GatherWriterMetadata phase does not give the requester information about what has changed since the last backup. One way for a writer to indicate changes to the requester is by using the differenced file mechanism. A writer can specify that certain files in a component should be backed up only if they have been modified since a certain time; the writer may specify these files either in IVssBackupComponents::PrepareForBackup or in IVssBackupComponents::DoSnapshotSet. A requester can determine these files by calling IVssComponent::GetDifferencedFilesCount and IVssComponent::GetDifferencedFile. If the file specification matches one set in **IVssBackupComponents::GatherWriterMetadata** (that is currently valid based on the backup type mask) the differenced file information overrides the previous information; that is, the files matching that file specification are now backed up only if they have been modified since the specified time. The last-modify time is communicated using a FILETIME structure. If the value of this structure is zero, this implies that the last-modify time should be determined based on the requester's record of the time of last backup.

Partial File Backup

Another way for a writer to indicate changes to the requester is by using the partial file mechanism. A writer can specify byte ranges within component files that need to be backed up; the writer can specify these file ranges either in IVssBackupComponents::PrepareForBackup or in IVssBackupComponents::DoSnapshotSet. A requester can determine these files by calling IVssComponent::GetPartialFileCount and IVssComponent::GetPartialFile. **IVssComponent::GetPartialFile** will return a path and a file name pointing to the file, and a ranges string indicating what needs to be backed up in the file. As with differenced files, if the path and file name matches a file specification set by the writer in IVssBackupComponents::GatherWriterMetadata, the partial-file information overrides the previous setting. The ranges string can have two formats: it can either specify the ranges directly, or it can specify a file containing ranges information. If specifying ranges directly, the syntax is a comma-separated list of the form offset1:length1, offset2:length2, where each offset and length is a 64-bit unsigned integer. If specifying a ranges file, the ranges string should be set to File= *filename*, where *filename* is the full path to the ranges file. The ranges file itself is a binary file that is formatted as a list of 64-bit unsigned integers. The first integer indicates how many ranges are represented in the file. Each subsequent pair of integers represents the offset and length of a range. The requester must take care to back up and restore this ranges file as well.

File Specification Rules

File specifications added through the differenced-file and the partial-file mechanisms will either modify file specifications set in IVssBackupComponents::GatherWriterMetadata or add completely new files. If modifying files specifications set in **IVssBackupComponents::GatherWriterMetadata** using the partial-file mechanism, then a requester can expect the file specification to exactly match one of the file specifications set in the component in **IVssBackupComponents::GatherWriterMetadata**. The file specification will not partially overlap another file specification, and it will not match any file specifications in any other of that writer's components. File specifications added using the partial-file mechanism can, however, partially overlap another file specification. When this is true, the differenced-file or partial-file specification overrides the specification set in **IVssBackupComponents::GatherWriterMetadata**. General file specifications can have an alternate-location value (returned by IVssWMFiledesc::GetAlternateLocation) that indicates an alternate place to obtain the file from at backup time. If the file specifications set through the differenced-file or partial file mechanisms match an existing files specification that has an alternate location, the data should be picked up from this alternate location. If the file specifications set through the differenced-file or partial-file mechanisms do not match any existing specifications for the component, then these file ranges should now be added to the backup. The requester can expect that only files on volumes that have already been included in the shadow copy set will be added using one of these mechanisms.

Backup without a Shadow Copy

Certain types of files may not need to be backed up off of a shadow copy volume. For example, this will often be true of database log files. Since log files grow monotonically, and a writer can specify exactly what portions of the file to back up using partial files, it will often be possible to backup the log off the original volume. As an optimization, a writer can mark which files require shadow copies for different backup types using the backup-type mask. The value returned from **IVssWMFiledesc::GetBackupTypeMask** can contain one or more of the following bit values:

VSS_FSBT_FULL_SNAPSHOT_REQUIRED

Shadow copy required for full backups.

VSS_FSBT_DIFFERENTIAL_SNAPSHOT_REQUIRED

Shadow copy required for differential backups.

VSS_FSBT_INCREMENTAL_SNAPSHOT_REQUIRED

Shadow copy required for incremental backups.

VSS_FSBT_LOG_SNAPSHOT_REQUIRED

Shadow copy required for log backups.

VSS_FSBT_ALL_SNAPSHOT_REQUIRED

Shadow copy required for all backup types; this is the default.

If a specific volume contains only components that do not require a shadow copy for this backup, the requester can skip the step of creating a shadow copy for this volume. All the data on this volume can be copied to the backup media directly from the original volume.

Restoring Files

Sequential Restores

After the requester is done performing a restore operation, it sends the PostRestore event to all writers. Generally, writers handle this event by performing recovery or other post-restore operations. Restoration of incremental backups, however, usually happen as a sequence of restore operations, one per incremental backup. The requester needs to inform the writer that such a restoration is in progress in order to prevent recovery or other undesirable operations from happening until the restore is completely done. This is done by calling **IVssBackupComponents::SetAdditionalRestores**. This method is called per component, and indicates to the writer that more restores are coming for that component. For the final restore in the sequence, the additional-restores flag should be set to false (its default value), which indicates that this is the last restore in the sequence for that component.

New Targets

If supported by the writer, the requester can restore data files to a location other than the original backup-time location. The requester can check for this support by calling **IVssExamineWriterMetadata::GetBackupSchema**. The **VSS_BS_WRITER_SUPPORTS_NEW_TARGET** bit will be set if the writer supports this behavior. The requester informs the writer of the new location by calling **IVssBackupComponents::AddNewTarget** for each relocated file specification. The requester can also decide to restore a specific ranges file to a different location. The requester informs the writer of this change by calling **IVssBackupComponents::SetRangesFilePath**.

Directed Targets

For complicated restore scenarios, a writer may want to map ranges of a backed-up file onto different ranges of the same or a different file. This can be done by using the directed target mechanism. The requester can

determine after the **IVssBackupComponents::PreRestore** phase that this mechanism is being used for a component by calling **IVssComponent::GetRestoreTarget** and checking for a return of **VSS_RT_DIRECTED**. The requester can than obtain all of these redirected restores by calling **IVssComponent::GetDirectedTargetCount** and **IVssComponent::GetDirectedTarget**. **IVssComponent::GetDirectedTarget** will return a full path to a source file on the backup and a full path to a destination file that will be restored to. It also returns a ranges list for each of these files. The requester is then responsible for restoring the specified ranges in the source file to the mapped ranges in the destination file. The format of the ranges string is the same as in **IVssComponent::GetPartialFile**.

- Writer Role in VSS Incremental and Differential Backups
- Requester Role in VSS Incremental and Differential Backups

Backups without Writer Participation

3/5/2021 • 2 minutes to read • Edit Online

When a VSS backup operation is conducted without the involvement of a writer, the shadow copy creation can still occur.

As noted elsewhere (see Default Shadow Copy State), the result of this type of shadow copy is a volume reflecting the state of a disk at the time of the shadow copy: data on the shadow-copied volume may reflect incomplete or partial I/O operations and is described as being in a *crash-consistent state*.

There are several situations that will require a backup application to work with crash consistent shadow copied data:

- **Data is managed by VSS-unaware applications**

  Almost every system will have some applications—text editors, mail readers, word processors, and so forth—that are VSS unaware, so it is always likely that some of the data on a shadow copy will need to be thought of as being in a crash consistent state.

  This sort of data is not typically system- or service-critical, so backing it up should not be problematic, or at least no more problematic than during a conventional backup.

  As with preparations for conventional backups, if possible, backup operators should attempt to suspend or terminate such applications prior to starting a VSS backup job.

- **System with no VSS-compatible writers**

  This situation may be relatively rare because most systems that can be backed up by a VSS backup application will have a VSS-enabled version of Windows, which should contain writers.

  However, there are certain circumstances where this problem could arise—for instance, if you are backing up a system without VSS support but the system uses a VSS-enabled networked appliance for its storage.

  Although backing up a system's state from a crash-consistent image is not optimal, it may be sufficient for a computer to reboot to an operational status. In many cases, the computer will recover from any incomplete I/O operations to the file systems and will operate normally.

- **A requester choosing not to work with the system writers**

  This circumstance would occur if a requester chose to add no writer components, or disabling all writers.

  Performing backups in this way is generally discouraged. Although the shadow-copied volume to back up might be sufficient to restore a system to running, it is not desirable. In fact, given that the writers running on the system are designed with functionality to cooperate with VSS and shadow copies, backing up their data in this way might result in instability.

Working with Mounted Folders and Reparse Points

3/5/2021 • 3 minutes to read • Edit Online

Processing one of a component's file sets may require a requester to traverse a directory tree recursively, which can require the requester to handle mounted folders and reparse points (such as links) that point to data that is not on the current volume.

Requesters are expected to follow mounted folders and reparse points when traversing a directory tree, and VSS has well-defined guidelines for handling them for backup and restore operations.

To illustrate these guidelines, consider the following example:

- The volume \\?\Volume{GUID_1} has the drive letter C:\.
- A file set has a path of C:\WriterData.
- A file specification *.dat is used by the file set.
- The file set's recursion is set to **TRUE**.
- The directory C:\WriterData is located on the volume \\?\Volume{GUID_1}.
- The directory C:\WriterData\Archive is a mounted folder.
- The volume \\?\Volume{GUID_2} is associated with the mounted folder C:\WriterData\Archive.

Handling Mounted Folders and Reparse Points during Backup

The basic rules for handling mounted folders and reparse points under VSS when performing a recursive backup can be summarized as follows:

- Paths are followed across mounted folders and reparse points.
- If a mounted folder or reparse point points to a volume, that volume should be shadow copied.
- If a volume contains mounted folders or reparse points, these will appear in the shadow copy of the volume.
- Data that is under the mounted folder or reparse point is captured in the shadow copy of the volume that the mounted folder or reparse point points to.

To illustrate using the example above, because the recursive flag is set, the requester must examine all data under C:\WriterData\Archive and below.

The requester must add both the volume with drive letter C:\ (\\?\Volume{GUID_1}) and the volume associated with the mounted folder C:\WriterData\Archive (\\?\Volume{GUID_2}) to the shadow copy set using **IVssBackupComponents::AddToSnapshotSet**.

The mounted folder C:\WriterData\Archive appears in the shadow copy of volume \\?\Volume{GUID_1}, which has a device object named deviceObject1.

However, VSS will not copy any data under that mounted folder (data on \\?\Volume{GUID_2}) to the shadow copy referenced by deviceObject1. Instead, that data is captured in the shadow copy of \\?\Volume{GUID_2}, which has a device object named deviceObject2.

Therefore, a requester that is backing up shadow copied files under C:\WriterData will use a path of deviceObject1\WriterData to search for files matching C:\WriterData*.dat.

To back up shadow copied files under C:\WriterData\Archive, the requester will use a path of deviceObject2 (because the root directory of \\?\Volume{GUID_2} was associated with the mounted folder C:\Writer\Archive) to search for files matching C:\WriterData\Archive*.dat.

Note that a reparse point is handled in the same way as a mounted folder. The reparse point appears in the shadow copy of the first volume. The data under the reparse point appears in the shadow copy of the second volume.

During backup, requesters should store information about mounted folders and the volumes associated with them as well as reparse points and their targets to ensure that all data is backed up and restored correctly.

Handling Mount and Reparse Points during Restore

When restoring files, the requester must follow guidelines somewhat different from those that were used during backup (ignoring issues such as *alternate location mapping* and *new target location*):

- As before, if recursion is required, paths are followed across mounted folders and reparse points.
- Mounted folders are to be restored.
- The restoration locations of mounted folders and reparse points are those determined by their original paths.

If the volume names persist between backup and restore—that is, you do not re-create the volumes—restored mounted folders and reparse points should point to the correct volumes.

Therefore, in the example discussed above, if the mounted folder C:\WriterData\Archive was restored to (\\?\Volume{GUID_1}) and the volume previously associated with it was restored to (\\?\Volume{GUID_2}), the restored files and file structure would be correct and consistent.

It may happen that data is restored to a system where volume names changed. This could be due to disk crashes, where you might need to perform a manual system recovery and re-create volumes. In this type of situation, mounted folders and reparse points will no longer be valid after restore. To re-create the files and file structure on the restored volume will require deleting the restored mounted folders and reparse points and re-creating them on disk. It is up to the backup application to decide whether this is appropriate.

Note that it is possible that the restore destination for a mounted folder is already occupied. For example, C:\WriterData\Archive might already contain some files. It is up to the backup application to decide how to handle this situation.

Implementation Details for VSS Restores

3/5/2021 • 2 minutes to read • Edit Online

The following topics describe implementation details for VSS restores:

- Generating a Restore Set
- Restoring Incremental and Differential Backups
- Working with Mounted Folders and Reparse Points
- Working with Alternate Locations during Restore
- Working with New Targets during Restore

Generating A Restore Set

3/5/2021 • 4 minutes to read • Edit Online

A restore set is a list of all files to be restored and the locations to which they will be restored.

As when generating the backup file list (see Generating A Backup Set), an algorithm for determining which files to restore and where to restore them must proceed *writer instance* by writer instance, and on a component-by-component basis for each writer instance.

It is necessary to associate each file on the backup media with the component that managed it. It is also necessary to obtain the managing component's *restore method*, and the file's *restore target* information, and its *alternate location mappings* (if any).

Some files may also require *partial files* operations or *directed targets* for restore.

By examining the components' *selectability for backup* and *logical paths* (see Working with Selectability and Logical Paths), a requester is able to determine the component structure of the backup operation it is going to restore.

With the component structure of the backup established, the requester can get each component's *file set* information (file specification, path, and recursion flag). A requester can then generate a restore set.

Files requiring *partial files*, or *directed targets* provide their own detailed restoration instructions (see Non-Default Backup and Restore Locations), which can then be added to the restore set.

A typical mechanism for generating a restore set for files not involved with partial files operations, or *directed targets* might proceed by doing the following:

1. Obtain a list of files on the backup media, including their original paths.
2. Identify the *writer class* and component for each file on the backup media by doing the following:
   - For each writer, obtain component information (**IVssWMComponent**) by calling **IVssExamineWriterMetadata::GetComponent** on all its components.
   - For each component, obtain file descriptor (**IVssWMFiledesc**) information for every set of files the component contains (depending on the types of data the component contains by calling **IVssWMComponent::GetFile**, **IVssWMComponent::GetDatabaseFile**, and **IVssWMComponent::GetDatabaseLogFile**.
   - Compare the file's name and path information against that returned by the path information contained in the file descriptor for each set of files in a component (returned by **IVssWMFiledesc::GetPath**, **IVssWMFiledesc::GetFilespec**, and **IVssWMFiledesc::GetRecursive**) against stored files path information to determine whether the file is part of the component.

     > **NOTE**
     > You should ignore any alternate location information in the file descriptor retrieved from a component found in a stored Writer Metadata Document (that is, **IVssWMFiledesc::GetAlternateLocation** does not return **NULL**). This alternate location is the *alternate path*, which is used only during backup.

3. Obtain alternate mapping information for each file on the backup media:
    - Alternate file mappings are stored at the writer, not the component level, and are obtained from the object **IVssWMFiledesc** returned by **IVssExamineWriterMetadata::GetAlternateLocationMapping**.
    - You can determine if a particular file has an alternate location mapping by checking the file's path and name against the path and file specification contained in the alternate location mapping returned by **IVssExamineWriterMetadata::GetAlternateLocationMapping**, via **IVssWMFiledesc::GetPath**, **IVssWMFiledesc::GetFilespec**, and **IVssWMFiledesc::GetRecursive**. (If an alternate path was used during backup, that information should be ignored during this check in processing a restore.)
    - If a file and an alternate location mappings file descriptors match, you then use the **IVssWMFiledesc::GetAlternateLocation** method of the **IVssWMFiledesc** object returned by **IVssExamineWriterMetadata::GetAlternateLocationMapping** to find the alternate location to which you can restore the file.
    - The alternate location mapping obtained in this way will not necessarily agree with that returned from the Backup Components Document by **IVssComponent::GetAlternateLocationMapping**. The **IVssWMFiledesc::GetAlternateLocation** value is nonblank only if the alternate location mapping is used for a file.
4. With this file and component information, the Backup Components Document can be queried to obtain information about restore targets, options, and new restore locations for each file. This information can be combined with the list of files, components, and alternate locations.
5. Files not protected by writers can be selected in a manner consistent with traditional restore operations.

At this point, a requester should have a list of all the files it needs to restore, along with instructions about how to restore them, and can begin restoring files on the basis of:

- Whether alternate location mappings, or the original file location is to be used as the target for the restore will depend on the presence or absence of a file at that target location and component settings of **VSS_RESTORE_TARGET** and **VSS_RESTOREMETHOD_ENUM** (see Non-Default Backup and Restore Locations).
- Whether an attempt at restoration succeeds will depend on issues such as the access permissions of the target, if target files are locked, and other conventional issues involved in file restoration.
- The success or failure of restoring a given component for a given writer instance should be preserved in the Backup Components Document by calling **IVssBackupComponents::SetFileRestoreStatus**. This will make the information accessible to writers when they process the PostRestore event.
- If a file is restored to an alternate location mapping, the requester must call **IVssBackupComponents::AddAlternativeLocationMapping**. This will allow writers to determine whether their files have been restored to alternate locations through the **IVssComponent::GetAlternateLocationMapping**.
- Requesters may find it desirable to restore files to completely new locations. This is acceptable, but the requester must indicate this to the writer by using the **IVssBackupComponents::AddNewTarget** method.

Restoring Incremental and Differential Backups

3/5/2021 • 2 minutes to read • Edit Online

Restoring an incremental or differential backup under VSS is not significantly different from any other VSS restore operation.

A writer can modify restore targets or request directed targeting, and a requester must handle alternate location mappings and new targets, just as with any other restore. There are, however, two significant issues to be aware of in handling the restore of an incremental or differential backup: additional restores and backup stamps.

Additional Restores

The first issue is that of additional restores. A backup operator may need to run several restore operations using an initial full and subsequent incremental or differential backup media as a source.

Some writers, typically as part of their handling of a *PostRestore* event using **CVssWriter::OnPostRestore**, use restored files to perform updates of data currently on disk. For some of these writers it is inefficient—or dangerous—to repeatedly update on-disk data from restored files.

Therefore, it is important that backup applications indicate when a component or component set may require subsequent restores by calling **IVssBackupComponents::SetAdditionalRestores**.

A writer would call **IVssComponent::GetAdditionalRestores** to determine whether the backup operator planned more restores of the component or component set.

If the requester had not called **IVssBackupComponents::SetAdditionalRestores**, then **IVssComponent::GetAdditionalRestores** returns false, and the writer can perform any post-restore processing it needs to.

If **IVssBackupComponents::SetAdditionalRestores** had been called, then **IVssComponent::GetAdditionalRestores** returns **true**, and a writer should decide how to handle post-restore operations—for instance, the writer may choose not to update its on-disk data.

Backup Stamps

As part of the previous full backup operation, a writer may have stored a backup stamp in the requester's Backup Components Document.

The backup stamp is stored as a string, and its format and information are not intelligible to the requester. Therefore, the requester cannot make direct use of the backup stamp information.

Instead, its task is to make that information available to the writer, by calling the **IVssBackupComponents::SetPreviousBackupStamp** method prior to the generation of a **PrepareForBackup** event for an incremental backup.

The requester does this on a component-by-component basis. A requester examines stored component or component set backup stamp information using **IVssComponent::GetBackupStamp**.

If backup stamp information is appropriate to the type of restore the requester is undertaking, it makes it available as the time stamp of the last backup of a component with the **IVssBackupComponents::SetPreviousBackupStamp** method.

A writer recovers the backup stamp information using **IVssComponent::GetPreviousBackupStamp**. A writer of this class generated the initial backup stamp, so the writer is able to decode this stamp and use the

information. Based on this, while handling a **PreRestore** event, a writer may choose to take actions such as changing restore targets or requesting directed targeting.

Restores without Writer Participation

3/5/2021 • 2 minutes to read • Edit Online

Writer participation in a VSS backup is designed to allow applications to control what and how their restore data is to be used.

In general, if a writer is available on a system, it is never advisable to restore data to its original location without writer participation. Such a restore would likely encounter locked destination files and runs a significant risk of corrupting data.

However, there are reasons why a backup application might want or need to restore a VSS backup without writer participation:

- Data is managed by VSS-unaware applications. Almost every system will have some applications—text editors, mail readers, word processors, and so forth—that are VSS unaware. This data cannot be restored using writer participation.

  Generally, this type of data is not system- or service-critical, and restoring it should not be problematic, or at least no more problematic than during a conventional restore.

  As with preparations for conventional restores, if possible, restore operators should attempt to suspend or terminate such applications prior to starting a VSS restore.

- Missing VSS writers. This situation may be fairly common when restoring the state of a damaged system. A backup operation must determine whether it is desirable to restore files for missing writers. If restoration is desirable, the files can be restored just as a conventional backup would restore them.

- A private restore of a writer's data. A requester may choose to restore the data of a running writer to some private location without notifying the writer. An example of this might be restoration of the writer's data to support offline comparison. In this sort of situation, a requester would not want to use the *new target location* while doing the restore, because it does not want the writer to access the data.

- A writer does not want to be involved during restore. A writer indicates this by passing in VSS_WRE_NEVER for the *writerRestore* parameter of **IVssCreateWriterMetadata::SetRestoreMethod**.

- A writer requires a custom restore method. A writer indicates that it requires a custom restore by passing in VSS_RME_CUSTOM for the *method* parameter of **IVssCreateWriterMetadata::SetRestoreMethod**. In this case, this writer should not be involved in the restore process unless the custom-restore documentation for that writer indicates otherwise.

A requester involves a writer in the restore process by specifying one of that writer's components in a call to **IVssBackupComponents::SetSelectedForRestore**. A writer's data can be restored without involving the writer by simply not specifying any of that writer's components in a call to **IVssBackupComponents::SetSelectedForRestore**. If a writer does not expect any restore events, involving that writer in the restore process can cause spurious errors to be reported for that writer.

Non-Default Backup and Restore Locations

3/5/2021 • 2 minutes to read • Edit Online

Backup and restore operations typically copy files from a given, default location on disk to backup media and then restore from that media to the same default location on disk.

There are many reasons, particularly when dealing with running processes, not to follow this simple model. VSS supports using nonstandard sources for backup and alternate destinations for restore, and includes mechanisms to work with subsets of files and to remap files.

- Working with Alternate Paths during Backup
- Working with Alternate Locations during Restore
- Working with New Targets during Restore
- Working with Partial Files
- Working with Directed Targets

Working with Alternate Paths during Backup

3/5/2021 • 2 minutes to read • Edit Online

There are certain circumstances where the files to be backed up are not the default location for those files.

For example, some writers cannot guarantee to have flushed of their data within the time window between *Freeze* and *Thaw* events. Such writers may choose to generate duplicate files containing a last known good configuration in a non-default source directory, or *alternate path*.

The term alternate path, as used with VSS, should not be confused with the term *alternate location mapping*. Alternate paths are used only during backup operations, and refer to an alternate source from which to back up. Alternate location mappings are used only during restore operations, and refer to an alternate destination for restore operations.

To use an alternate path during backup

1. During the discovery phase of a backup operation (see Overview of the Backup Discovery Phase) a requester would examine each writer's component data using **IVssExamineWriterMetadata::GetComponent** and get instances of the **IVssWMComponent** interface.
2. A requester then obtains the *file set* managed by each component, represented by instances of the **IVssWMFiledesc** interface, by calling the **IVssWMComponent::GetFile** method.
3. In addition to a path (**IVssWMFiledesc::GetPath**), a file specification (**IVssWMFiledesc::GetFilespec**), and a recursion flag (**IVssWMFiledesc::GetRecursive**), a **IVssWMFiledesc** object may contain an alternate location (used as an alternate path for backup operations and an alternate location mapping for restore operations) by using the **IVssWMFiledesc::GetAlternateLocation** method.
4. If the value returned by **IVssWMFiledesc::GetAlternateLocation** is non-NULL, backup applications use that value instead of the value obtained from **IVssWMFiledesc::GetPath** to select and locate files to back up.
5. Despite using an alternate path, requesters should still respect the file specification and recursive settings returned by **IVssWMFiledesc::GetFilespec** and **IVssWMFiledesc::GetRecursive**.

Note that on restore, any alternate path—that is, an alternate location returned by an instance of **IVssWMFiledesc::GetAlternateLocation** gotten from an instance of **IVssWMComponent**, which in turn was obtained from an instance of **IVssExamineWriterMetadata** gotten by retrieving a stored Writer Metadata Document—is not used during restoration.

Either the default path (returned by the **GetPath** method of the same instance of **IVssWMFiledesc**) is used to define a restore location, or an alternate location mapping—found by using the **IVssWMFiledesc::GetAlternateLocation** method—indicates where a file is to be restored (see Working with Alternate Locations during Restore).

Working with Alternate Locations during Restore

3/5/2021 • 3 minutes to read • Edit Online

There are many reasons why a requester either should not, or may not, be able to restore files from backup media to their original location. For example, a restore method or target may require such a restoration, or the current restoration location may be occupied and unwritable.

To handle such cases, a writer may have defined an *alternate location mapping*, a nonstandard restore destination to be used for special circumstances.

The term alternate location mapping, as used with VSS, should not be confused with the term *alternate path*. Alternate location mappings are used only during restore operations, and refer to an alternate destination for restore operations. Alternate paths are used only during backup operations, and refer to an alternate source from which to back up.

To use alternate location mappings during restore, a requester would do the following (typically following the generation of a **PreRestore** event):

1. Using an instance of the **IVssExamineWriterMetadata** interface obtained by retrieving a stored writer, a requester uses the **IVssExamineWriterMetadata::GetAlternateLocationMapping** method to obtain a writer's alternate location mappings as instances of the **IVssWMFiledesc** interface.

   > **NOTE**
   >
   > The requester uses **IVssExamineWriterMetadata::GetAlternateLocationMapping**, not **IVssComponent::GetAlternateLocationMapping**. The former returns those alternate location mappings available for use by a requester. The latter is used to indicate those alternate location mappings actually used by a requester.

2. The call to the **IVssExamineWriterMetadata::GetAlternateLocationMapping** returns an instance of the **IVssWMFiledesc** interface. This instance contains file set information—a path specified by **IVssWMFiledesc::GetPath**, a file specification returned through **IVssWMFiledesc::GetFilespec**, and a recursion flag obtained from **IVssWMFiledesc::GetRecursive**—matching one of the file sets added (using **IVssCreateWriterMetadata::AddDatabaseFiles**, **IVssCreateWriterMetadata::AddDatabaseLogFiles**, or **IVssCreateWriterMetadata::AddFilesToFileGroup**) to one of the components managed by the writer.

   The value returned by **IVssWMFiledesc::GetAlternateLocation** is the alternate location mapping for this file set.

3. Alternate location mappings do not contain component information, so it will be necessary to compare the file set information (path, file specification, and recursion flag) obtained by calling **IVssExamineWriterMetadata::GetAlternateLocationMapping** to that contained by the writer's components.

   This information can be found by iterating over the writer's components and calling **IVssExamineWriterMetadata::GetComponent** to get an instance of the **IVssWMComponent** interface and use **IVssWMComponent::GetFile** to obtain a **IVssWMFiledesc** instance containing the component file set information.

When the file set information returned by the instance of **IVssWMFiledesc** obtained from **IVssExamineWriterMetadata::GetComponent** and **IVssWMComponent::GetFile** matches that obtained from the **IVssWMFiledesc** instance derived from **IVssWMFiledesc::GetAlternateLocation**, the component managing the files with the specific alternate location mapping has been found.

4. Having located the component, the requester can determine the conditions under which an alternate location mapping should be used by doing the following:

   - Examining the component's restore method, which is obtained by calling **IVssExamineWriterMetadata::GetRestoreMethod**.
   - Checking to see if a restore target overrides the restore method, by calling **IVssComponent::GetRestoreTarget**.

     If the component found in the Writer Metadata Document had been *explicitly included* in the backup, the instance of the **IVssComponent** interface will correspond to that component. If the component had been *implicitly included* in the backup, then the instance of **IVssComponent** will correspond to the component defining the component set of which the component in the Writer Metadata Document is a subcomponent.

5. With this information, the requester can determine on a component-by-component basis if it needs to restore a given file set of a given component to a destination defined by the alternate location mapping.

6. When using an alternate location mapping, the requester respects the file set's file descriptor and recursive flag and uses the path provided by the alternate location mapping.

   The requester indicates that it has used an alternate location mapping during a restore operation by calling the **IVssBackupComponents::AddAlternativeLocationMapping** with the file set's default location information, the alternate restore destination used, and a component name.

   If the file set was managed by a component that was explicitly included in the backup, that component name will be used. If the file set was managed by a component that was implicitly included in the backup, then the name used will be that of the component defining the component set of which the component managing the file set is a subcomponent.

Writers verify whether file sets from one of their components were restored to an alternate location mapping by calling **IVssComponent::GetAlternateLocationMapping**.

Working with New Targets during Restore

3/5/2021 • 2 minutes to read • Edit Online

A requester may need to restore files to a location indicated by something other than a file set's default path or its *alternate location mapping*. There are many reasons why this might happen—for example, neither restore destination was accessible, or a requester user intentionally requests that files be restored to some previously unknown location. In this case, the requester uses the new target mechanism to indicate to writers that it has restored a file to a different area on disk.

Not all writers support a requester changing the restore destination of a file. A requester needs to verify writer support by checking the writer's backup schema mask (returned by **IVssExamineWriterMetadata::GetBackupSchema**) and verifying that it contains the VSS_BS_WRITER_SUPPORTS_NEW_TARGET flag.

The requester indicates such a restoration through the **IVssBackupComponents::AddNewTarget** method. In addition to specifying a file specification and an original and a new restore destination, the requester specifies component information—a logical path and component name.

Which component's information is used depends on whether or not the component managing the file having a new target added was *explicitly included* or *implicitly included* in the backup.

If the managing component was explicitly included, then its information is used. If the managing component was implicitly included, it is a subcomponent in a component set. In this case, the component set's defining component's information is used.

While handling the **PostRestore** event, writers should check to see if any of its files were restored to a new location. This can be done by using the **IVssComponent::GetNewTargetCount** and **IVssComponent::GetNewTarget** methods.

The instance of the **IVssComponent** interface that is used depends on whether the file's managing component was explicitly or implicitly added to the backup.

Working with Partial Files

3/5/2021 • 3 minutes to read • Edit Online

It is sometimes useful to back up and restore only sections of files. VSS provides *partial file* mechanisms, which, if requesters support it, allows writers to specify partial file backups and restores.

Partial file operations are frequently of greatest use to writers that maintain very large files, only a small fraction of which change between backup operations. This being the case, it is frequently useful to copy only that section that changed to backup media. For this reason, partial file operations are typically, but not exclusively, used to support incremental backup and restore operations.

If a writer wants to implement a partial file operation, it uses **CVssWriter::IsPartialFileSupportEnabled** to determine whether the requester it is working with supports the operation.

If the requester supports partial file operations, and if it adds the component that manages the file (or the component that defines the component set that contains the file) to the Backup Components Document, a writer indicates which sections of the file to save (typically while handling a **PrepareForBackup** or *PostSnapshot* event) by calling **IVssComponent::AddPartialFile**.

In addition to a path and file name, the writer supplies the range, optional metadata information to **IVssComponent::AddPartialFile**.

The range information is provided as a string that contains either of the following:

- Pairs of offsets into the file to be backed up (in bytes) and the length of the section to be backed up (in bytes), the offset and length being separated by a colon, and each pair separated by a comma, for example, *Offset1*:**Length1**,* Offset2*:*Length2*.

  Each value is a 64-bit integer (in either hexadecimal or decimal format) specifying a byte offset and length in bytes, respectively.

- The full path, including file name, on the current system of a binary ranges file containing the following:

  - The number (expressed as a 64-bit integer) of distinct file ranges contained in the file
  - Each range expressed as a pair of 64-bit integers: the first member of the pair being the offset into the file being backed up (in bytes), and the second member being the length of data to be backed up (in bytes)

If a writer uses a ranges file to specify a partial file operation, a requester must guarantee that either this file is backed up (even if the file is not necessarily part of the default backup set) or that the ranges information is preserved on the backup media in some other way. If the ranges file's information is not backed up, restoring the partially backed up file will be impossible.

The writer can also add a string containing metadata. This metadata can be in a writer-specific format because it is intended to allow the writer to validate any future restores.

With this information, a supporting requester can perform a partial file backup.

As an example, consider a large file whose header (bytes 64-512) contains a record count and other frequently updated information, and whose most recent data is to be found in the file's last 65536 bytes—bytes 0x1239E8577A to 0x1239E7577A.

A writer could specify a range list as the string "**64:448,0x1239E8577A:65536**."

On restore, and prior to actually performing a restore operation, a requester should check to see if any files

require partial file support.

To do this, the requester first iterates over the writers with stored components in its Backup Components Document using **IVssBackupComponents::GetWriterComponentsCount** and **IVssBackupComponents::GetWriterComponents**.

The **IVssBackupComponents::GetWriterComponents** interface is then used to return instances of the **IVssWriterComponentsExt** interface, which provide **IVssWriterComponentsExt::GetComponent** and **IVssWriterComponentsExt::GetComponentCount**, that allow the requester to obtain **IVssComponent** instances.

This allows a requester to obtain information about the partially backed up files to participate in a restore by using **IVssComponent::GetPartialFileCount** and **IVssComponent::GetPartialFile** for the instance of **IVssComponent** corresponding to the component that manages the file (or the component that defines the component set that contains the file).

If the partial file operation was controlled by a ranges file, that file should be restored prior to copied data back to disk. It may happen that the requester needed to copy the ranges file back to a new location on disk. In this case, it indicates that it has done so through the **IVssBackupComponents::SetRangesFilePath**.

The requester then proceeds to copy data into the appropriate locations in the restore destination already on disk.

A writer (while handling a **PostRestore** event), by examining **IVssComponent::GetFileRestoreStatus** for the files indicated by **IVssComponent::GetPartialFile**, determines if the partial file operation was successful. The writer should always attempt to verify the correctness of this restore using the offset information and any metadata included in the Backup Components Document.

If the requester has had to restore the ranges file to a new location, VSS will update this information so that the path returned by **IVssComponent::GetPartialFile** is correct.

Working with Directed Targets

3/5/2021 • 2 minutes to read • Edit Online

The *directed target* mechanism allows writers to remap files at restore time. This allows writers to do the following:

- Specify new target locations (analogous to a requester's **IVssBackupComponents::AddNewTarget**).
- Reclaim disk space by restoring only needed parts of a file to disk, particularly when a file was backed up using the *partial file* mechanism.
- Change the file format to meet current needs.

Any file to be used with a directed target operation must have a *restore target* of VSS_RT_DIRECTED.

Once it has been established that a requester can support a directed target operation, a writer (while handling the **PreRestore** event) uses the **IVssComponent::AddDirectedTarget** for the instance of **IVssComponent** corresponding to the component that manages the file (or the component that defines the component set that contains the file) to define how the file is to be remapped on restore.

In using **IVssComponent::AddDirectedTarget**, writers specify the file name and path used to back up the file, the file name and path of its restore destination (these values can be the same as the original file name and path), and source and destination file ranges.

As with partial file operations, range lists are pairs of offsets into the file to be backed up (in bytes) and the length of the section to be restored (in bytes), the offset and length being separated by a colon, and each pair separated by a comma: *Offset1*:***Length1***,* *Offset2*:*Length2*. Each value is a 64-bit integer in either hexadecimal or decimal format.

If a writer needs to use the directed target mechanism to have the requester restore a file to a new location, it would have called **IVssComponent::AddDirectedTarget** with the original file name and path and the new file name and path, and specify source destination ranges with a zero offset and a length equal to that of the entire file size.

For instance, if a writer needs to have a 200K file, C:\WriterData\Index.dat, restored as C:\WriterData\OldIndex.dat, the source and destination range string would be "**0:204880**."

To remap a large, partially backed-up file, the requester would use the source range used to back up the file and a destination range that will reduce the file size. The source range information can be obtained by using **IVssComponent::GetPartialFile** for the instance of **IVssComponent** corresponding to the component that manages the file (or the component that defines the component set that contains the file).

If the partially backed-up file was initially a large file whose header, bytes 64-512, contains a records count and other frequently updated information, and whose most recent data is to be found in the file's last 65536 bytes—bytes 0x1239E8577A to 0x1239E7577A, a writer could specify a source range list as the string "**64:448,0x1239E8577A:65536**."

If the writer wanted to remap the restored file to contain only the header and most recent data, the range list could be the string "**0:488,488:65536**."

Prior to actually performing a restore operation, a requester should check to see if any files require directed target support.

To do this, the requester first iterates over the writers with stored components in its Backup Components Document using **IVssBackupComponents::GetWriterComponentsCount** and

IVssBackupComponents::GetWriterComponents.

The IVssBackupComponents::GetWriterComponents interface is then used to return instances of the IVssWriterComponentsExt interface, which provides IVssWriterComponentsExt::GetComponent and IVssWriterComponentsExt::GetComponentCount methods that allow the requester to obtain IVssComponent instances.

This allows a requester to obtain directed target candidates by using IVssComponent::GetDirectedTargetCount and IVssComponent::GetDirectedTarget for the instance of IVssComponent corresponding to the component that manages the file (or the component that defines the component set that contains the file).

Implementation Details for Creating Shadow Copies

3/5/2021 • 2 minutes to read • Edit Online

The following topics provide implementation details for creating shadow copies:

- Simple Shadow Copy Creation for Backup
- Shadow Copy Creation Details
- Selecting Providers

Simple Shadow Copy Creation for Backup

3/5/2021 • 2 minutes to read • Edit Online

There are a number of different types of shadow copy a requester can create. However, for most backup applications, you would do the following:

1. Call **IVssBackupComponents::SetContext** with a context of VSS_CTX_BACKUP.
2. Call **IVssBackupComponents::GatherWriterMetadata** to initialize communication with writers.
3. Call **IVssBackupComponents::AddComponent** to add file and database components to the backup.
4. Call **IVssBackupComponents::StartSnapshotSet** to initialize the shadow copy mechanism.
5. Call **IVssBackupComponents::AddToSnapshotSet** to include volumes in the shadow copy.
6. Call **IVssBackupComponents::PrepareForBackup** to notify writers of backup operations.

Shadow Copy Creation Details

3/5/2021 • 2 minutes to read • Edit Online

In general, how a shadow copy is created depends on the type of shadow copy to be created, its context, and the role accorded to writers in the shadow copy operation. (See Shadow Copy Context Configurations for more information.)

The shadow copy context is set by calling the **IVssBackupComponents::SetContext** method. Before calling the **IVssBackupComponents::DoSnapshotSet** method to create a shadow copy, requesters must call the **IVssBackupComponents** methods in the order specified in the following sections.

Prerequisites for All Shadow Copies

Regardless of the level of writer participation, the creation of any shadow copy will always require the requestor be initialized with calls to **IVssBackupComponents::InitializeForBackup** and **IVssBackupComponents::StartSnapshotSet**.

If this call is not made, the **IVssBackupComponents::DoSnapshotSet** method will return an error.

Shadow Copies with Writer Participation

If the shadow copy context specifies writer participation (that is, **IVssBackupComponents::SetContext** is called with **VSS_CTX_BACKUP**, or **VSS_CTX_APP_ROLLBACK**):

- Requesters must always call **IVssBackupComponents::GatherWriterMetadata** when the shadow copy context supports writer participation. If the shadow copy context supports writer participation and **IVssBackupComponents::GatherWriterMetadata** is not called prior to **IVssBackupComponents::DoSnapshotSet**, an error will be returned.
- If a requester wants to select specific writer components, it must call **IVssBackupComponents::AddComponent** before calling **StartSnapshotSet** to create the shadow copy set.
- **StartSnapshotSet** must be called to create the shadow copy set.
- Requesters may add one or more volumes to the shadow copy set by calling **AddToSnapshotSet**. Some writer components may not specify any affected volumes. In this case, it is acceptable for a snapshot set to be empty (that is, to contain no volumes).
- To guarantee the consistency of writer metadata, a requester should always call **IVssBackupComponents::PrepareForBackup** even if no components are selected. This causes VSS to generate a **PrepareForBackup** event, in which VSS calls the **CVssWriter::OnPrepareBackup** method for each participating writer.
- VSS will generate *PrepareForSnapshot* and *Freeze* events before creating the shadow copy in response to **IVssBackupComponents::DoSnapshotSet**. Writers will handle the events with **CVssWriter::OnPrepareSnapshot** and **CVssWriter::OnFreeze**.
- VSS will generate *Thaw* events and *PostSnapshot* events after creating a shadow copy in response to **IVssBackupComponents::DoSnapshotSet**. Writers will handle the events with **CVssWriter::OnThaw** and **CVssWriter::OnPostSnapshot**.

Shadow Copies without Writer Participation

Creating shadow copies without writer participation is discouraged for standard backup applications (see Backups without Writer Participation).

There are uses, such as fast backups of a disk to provide a safety net against accidental file corruption, which can be conducted without explicit writer participation. Such a shadow copy would have a context of either **VSS_CTX_FILE_SHARE_BACKUP** or **VSS_CTX_NAS_ROLLBACK**.

For this type of shadow copy, note the following:

- Requesters must still call the required methods listed in Prerequisites for All Shadow Copies.
- Requesters may call **IVssBackupComponents::GatherWriterMetadata**, but this is not required.
- If requesters call **IVssBackupComponents::AddComponent**, **IVssBackupComponents::PrepareForBackup**, or **IVssBackupComponents::BackupComplete**, an error will be returned.
- Providers will not generate *PrepareForSnapshot*, *Freeze*, *Thaw*, or *PostSnapshot* events for this type of shadow copy.

Excluding Files from Shadow Copies

3/5/2021 • 3 minutes to read • Edit Online

In Windows Vista and Windows Server 2008 and later, the developer of a VSS writer or application may choose to exclude certain files from shadow copies.

The performance impact and shadow copy storage area (also called "diff area") usage of a file in a shadow copy are directly related to the amount of change in the file's contents after the shadow copy is created. In addition, excluding files from shadow copies may slow down shadow copy creation.

For these reasons, a file should be excluded from shadow copies only if it is large, undergoes significant change between one shadow copy and the next, and does not need to be backed up.

You should only exclude files that belong to your application.

If the VSS_VOLSNAP_ATTR_NO_AUTORECOVERY flag is set in the shadow copy context, this means that auto-recovery is disabled, and no files can be excluded from the shadow copy. For more information, see the **_VSS_VOLUME_SNAPSHOT_ATTRIBUTES** enumeration.

Using the AddExcludeFilesFromSnapshot Method

A VSS writer can exclude files from a shadow copy as follows:

1. Call the **IVssCreateWriterMetadataEx::AddExcludeFilesFromSnapshot** method to report the files to be excluded.
2. In the writer's **CVssWriter::OnPostSnapshot** method, delete the files from the shadow copy.

Using the FilesNotToSnapshot Registry Key

> **NOTE**
>
> The **FilesNotToSnapshot** registry key is intended to be used only by applications. Users who attempt to use it will encounter limitations such as the following:
>
> - It cannot delete files from a shadow copy that was created on a Windows Server by using the Previous Versions feature.
> - It cannot delete files from shadow copies for shared folders.
> - It can delete files from a shadow copy that was created by using the DiskShadow utility, but it cannot delete files from a shadow copy that was created by using the Vssadmin utility.
> - Files are deleted from a shadow copy on a best-effort basis. This means that they are not guaranteed to be deleted.

A VSS application can delete files from a shadow copy during shadow copy creation by using the following registry key:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\BackupRestore\FilesNotToSnapshot

This registry key has REG_MULTI_SZ values for each application whose files can be excluded. The files are specified by fully qualified paths, which can contain the * wildcard.

In all cases, the entry is ignored if there are no files that match the path string.

After a file is added to the appropriate registry key value, it is deleted from the shadow copy during creation by the shadow copy optimization writer on a best-effort basis.

If a fully qualified path cannot be specified, then a path can also be implied by using the $UserProfile$ or $AllVolumes$ variable. For example:

- $UserProfile$\Directory\Subdirectory\FileName.*
- $AllVolumes$\TemporaryFiles*.*

To make the path recursive, append " /s" to the end. For example:

- $UserProfile$\Directory\Subdirectory\FileName.* /s
- $AllVolumes$\TemporaryFiles*.* /s

The $UserProfile$ variable causes the path string to be applied to all user profiles on the computer. The user profiles are enumerated by examining the following registry key:

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\ProfileList

The $AllVolumes$ variable causes the path string to be applied to all shadow copies on the computer. For example, suppose the path is "$AllVolumes$\TemporaryFiles*.* /s", and the computer has three volumes: C:, D:, and E:. If C: and E: contain the path "\TemporaryFiles\", and volume D: contains only the path D:\Data\, the directory tree C:\TemporaryFiles\ is deleted from shadow copies of C:, and the directory tree E:\TemporaryFiles\ is deleted from shadow copies of E:.

Administrators can disable expansion of the $UserProfile$ variable by using the following registry key:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Vss\Settings

Under this registry key, specify DisableUserProfileExpansion for the value name, REG_DWORD for the value type, and a nonzero value for the value data.

About the FilesNotToBackup Registry Key

The **FilesNotToBackup** registry key can be used to specify the names of the files and directories that backup applications should not backup or restore. However, it does not exclude those files from shadow copies. For more information about this registry key, see Registry Keys and Values for Backup and Restore.

Selecting Providers

3/5/2021 • 2 minutes to read • Edit Online

A requester should select a specific provider only if it has some information about the providers available.

Because this will not generally be the case, it is recommended that a requester supply GUID_NULL as a provider ID to **IVssBackupComponents::AddToSnapshotSet**, which allows the system to choose a provider according to the following algorithm:

1. If a hardware provider that supports the given volume is available, it is selected.
2. If no hardware provider is available, then if any software provider specific to the given volume is available, it is selected.
3. If no hardware provider and no software provider specific to the volumes is available, the system provider is selected.

However, a requester can obtain information about available providers by using **IVssBackupComponents::Query**. With this information, and only if the backup application has a good understanding of the various providers, a requester can supply a valid provider ID to **IVssBackupComponents::AddToSnapshotSet**.

Note that all volumes do not need to have the same provider.

Implementation Details for Using Shadow Copies

3/5/2021 • 2 minutes to read • Edit Online

The following topics provide implementation details for using shadow copies:

- Requester Access to Shadow Copied Data
- Importing Transportable Shadow Copied Volumes
- Exposing and Surfacing Shadow Copied Volumes
- Breaking Shadow Copies

Requester Access to Shadow Copied Data

3/5/2021 • 2 minutes to read • Edit Online

Once the shadow copy has completed, the most important mechanism for getting access to the file data it contains is through the use of the shadow copy's *device object*.

The **m_pwszSnapshotDeviceObject** member of a **VSS_SNAPSHOT_PROP** structure is a string containing a shadow-copied volume's device object. A requester can obtain a shadow-copied volume's **VSS_SNAPSHOT_PROP** object if it knows the volume's **VSS_ID** (identifying GUID) and passes it to **IVssBackupComponents::GetSnapshotProperties**.

A requester can also obtain shadow copy property information by using the **Obj.Snap** member of the **VSS_OBJECT_PROP** structure (which is a **VSS_SNAPSHOT_PROP** structure) obtained by using **IVssEnumObject** to iterate over the list of objects returned by a call to **IVssBackupComponents::Query**.

The device object should be interpreted as the root of a shadow-copied volume. For this reason, the device object contains no backslash ("\").

Paths on the shadow copied volume are obtained by replacing the root of the original path with the device object. For example, given a path on the original volume of "C:\DATABASE*.mdb" and a **VSS_SNAPSHOT_PROP** instance of snapProp, you would obtain the path on the shadow copied volume by concatenating snapPropm_pwszShadow copyDeviceObject, "\", and "\DATABASE*.mdb".

The VSS file sets might have wildcard characters in their file descriptors, so obtaining a full list of the files on a shadow copy managed by a component might require the use of methods such as **FindFileFirst**, **FindFileFirstEx**, and **FindNextFile**.

Importing Transportable Shadow Copied Volumes

3/5/2021 • 3 minutes to read • Edit Online

It is sometimes desirable to create a shadow copy on one system, but use the shadow copy on a second system.

Consider the case where data to be backed up is normally managed by a given system (*systemOne*) during normal operations, and that this data is physically stored on a storage array or an appliance.

To minimize any disruption to *systemOne* (because backup operations can be resource intensive), it is desirable to perform the backup using *systemTwo*, a backup server, which has access to the same storage array as *systemOne*.

To ensure a proper shadow copy—cooperating with the writers on *systemOne* and preserving the state appropriately for ongoing tasks—the shadow copy should be performed by *systemOne*.

Therefore, *systemOne* must create a *transportable shadow copy*, which *systemTwo* will then import.

Windows Server 2003, Standard Edition, Windows Server 2003, Web Edition and Windows XP: Transportable shadow copy sets are not supported. All editions of Windows Server 2003 with Service Pack 1 (SP1) support transportable shadow copy sets.

A typical example of importing a transportable shadow copy can proceed in the following way:

1. Initially, the logical unit (LUN) that is provided by the storage array is mounted as a volume on *systemOne* (say, F:).
2. A requester that is running on *systemOne* instantiates an instance of **IVssBackupComponents** and proceeds as if it were preparing for a backup. (See Overview of Backup Initialization, Overview of the Backup Discovery Phase, and Overview of Pre-Backup Tasks for more information.)
3. The requester on *systemOne* modifies the shadow copy context that is typically used for local backup operation (**VSS_CTX_APP_BACKUP**) to indicate that a transportable shadow copy will be created (**VSS_VOLSNAP_ATTR_TRANSPORTABLE**). The transportable attribute can be added to other shadow copy contexts as well.
4. With a shadow copy context of **VSS_CTX_APP_BACKUP | VSS_VOLSNAP_ATTR_TRANSPORTABLE**, the requester that is on *systemOne* creates a shadow copy by calling **IVssBackupComponents::DoSnapshotSet**.
5. *SystemOne* uses **IVssBackupComponents::SaveAsXML** to save the current state of the Backup Components Document and **IVssExamineWriterMetadata::SaveAsXML** to save each writer's Writer Metadata Documents. The XML strings that contain these documents are then made available to a requester that is running on *systemTwo*.

   The requester transfers the Backup Components Document to *systemTwo*.

   Note that the requester on *systemOne* does not release its instance of **IVssBackupComponents** at this point if the purpose of the shadow copy is for backup. The interface should remain open until *systemTwo* successfully finishes its backup operations. Only then should the requester issue a **BackupComplete** event since some writers will truncate logs and do other work after a successful backup. If the goal of the shadow copy is data mining or other purposes, then the interface can be closed at this step.
6. The requester on *systemTwo* then calls **IVssBackupComponents::ImportSnapshots** to get access to the shadow copy that was created by the requester on *systemOne*.

> **NOTE**
>
> The requester is responsible for serializing the import shadow copy operation. Also, if the call to **IVssBackupComponents::ImportSnapshots** fails, the VSS won't clean up LUNs on it's own. The requester has to initiate the cleanup of LUNs.

7. The requester on *systemTwo* proceeds with the backup of the shadow copied material exactly as if it were backing up a shadow copy that it created by itself (see Overview of Actual Backup Of Files).

   The requester on *systemTwo* obtains the shadow copy's *device object* using **IVssBackupComponents::GetSnapshotProperties** on the imported shadow copy and appends that to the beginning of the original file paths that were obtained from the metadata to access files to be backed up.

8. After using the shadow copy, the requester on *systemTwo* must delete the shadow copy. As with non-transportable shadow copies, if the shadow copy context indicates auto-release shadow copies (for example, **VSS_CTX_BACKUP**), then releasing the **IVssBackupComponents** on *systemTwo* will cause the VSS service to delete the shadow copy. Otherwise, if the context indicates a persistent shadow copy (for example, **VSS_CTX_APP_ROLLBACK**), then the requester on *systemTwo* must explicitly delete the shadow copy.

   Then the requester on *systemTwo* signals the requester on *systemOne* that it has finished with the backup of the transportable shadow copy.

9. After the requester on *systemOne* has received notification that the requester on *systemTwo* has finished the backup of the transportable shadow copy, it notifies the writers on its system by generating a **BackupComplete** event with a call to **IVssBackupComponents::BackupComplete**. At this point, the requester on *systemOne* is free to release its instance of **IVssBackupComponents**.

Transportable shadow copies in a cluster: Transportable shadow copies must be imported from outside the cluster as long as the original volume is mounted within the cluster. For information about implementing a fast recovery in a cluster, see Fast Recovery Using Transportable Shadow Copied Volumes.

Fast Recovery Using Transportable Shadow Copied Volumes

3/5/2021 • 2 minutes to read • Edit Online

Transportable shadow copies can be used to facilitate a *fast recovery*.

Fast recovery can be used to quickly revert to an earlier shadow copy. The steps involved are:

1. Create the transportable shadow copy of the appropriate LUNs. The shadow copy can be either *persistent* or nonpersistent.
2. Remove the original LUNs
   a. If the computer is inside a cluster, mark the affected disk resources as offline, or enable the maintenance mode for these disk resources.
   b. Mask the affected LUNs from this computer (and all other nodes if the computer is in a cluster).
3. Import the shadow copy using the **IVssBackupComponents::ImportSnapshots** method.
4. Break the shadow copy using the **IVssBackupComponents::BreakSnapshotSet** method.
5. Mark the new shadow copy volumes as read/write.
6. If the computer is in a cluster, expose the new shadow copy LUNs as the original LUNs.
   a. Unmask the shadow copy LUNs to the other nodes in the cluster.
   b. Mark the resources that were previously marked as offline as online, or disable the maintenance mode for those disk resources.

> **NOTE**
>
> Transportable shadow copies in a cluster are not supported before Windows Server 2003 with Service Pack 1 (SP1). This is only supported with compliant LUNs, which have at least one SCSI Vital Product Data (VPD) page 0x83 STORAGE_IDENTIFIER of type 1,2, or 8, and association 0, and the LUNs must maintain a basic disk with MBR partitioning.

Exposing and Surfacing Shadow Copied Volumes

3/5/2021 • 2 minutes to read • Edit Online

In addition to being accessed through the **IVssBackupComponents** interface by means of its copy's *device object*, a requester can make a shadow copy available to other processes as a mounted read-only device.

This process is known as *exposing a shadow copy*, and is performed using the **IVssBackupComponents::ExposeSnapshot** method.

A shadow copy can be exposed as a local volume—assigned a drive letter or associated with a mounted folder—or as a file share.

To illustrate, consider a shadow copy made of a volume on the system exposedSys mounted at F:\ on whose root are the directories dirOne and dirTwo, and the file FileOne.

Exposing a Shadow Copy Locally

When mounted as a local volume, the shadow copy's root is always visible at the mount point (drive letter or mounted folder) and all the shadow-copied files are visible.

If the shadow copy was locally exposed through the mounted folder C:\ShadowOfF, you would find all files present on disk mounted at F:\ at the time of the shadow copy available under C:\ShadowOfF. Examining C:\ShadowOfF would reveal two directories, dirOne and dirTwo, and one file, fileOne, under C:\ShadowOfF.

A call to locally expose the shadow copy might be:

```
IVssBackupComponents *pReq;
VSS_ID snapID;
PWSTR wszExposed;
//    .
//    .
hr = pReg->ExposeSnapshot(
       snapID,                          // VSS_ID SnapshotId,
       NULL,                            // VSS_PWSZ wszPathFromRoot
       VSS_VOLSNAP_ATTR_EXPOSED_LOCALLY, // LONG lAttributes
       L"C:\ShadowOfF",                 // VSS_PWSZ wszExpose
       LPWSTR &wszExposed,              // VSS_PWSZ* pwszExposed
     );
```

If the shadow copy was successfully exposed locally, *wszExposed* should contain the wide character string "C:\ShadowOfF."

The shadow copy can later be unexposed by calling **IVssBackupComponentsEx2::UnexposeSnapshot**.

Only persistent shadow copies—that is, shadow copies created with either VSS_CTX_NAS_ROLLBACK or VSS_CTX_APP_ROLLBACK—can be exposed locally.

Exposing a Shadow Copy as a Remote Share

Alternatively, you could choose to make the shadow copy of the disk mounted at F:\ available as a remote file share, and expose only data under dirTwo as the file share dirTwoOfF.

In this case, systems could access the shadow copy of files under F:\dirTwo by mapping \\exposedSys\dirTwoOfF as a network drive.

A call to implement remote exposing the shadow copy as a share might be the following:

```
IVssBackupComponents *pReq;
VSS_ID snapID;
LPWSTR wszExposed;
//    .
//    .
hr = pReg->ExposeSnapshot(
             snapID,                          // VSS_ID SnapshotId,
             L"\dirTwo",                      // VSS_PWSZ wszPathFromRoot
             VSS_VOLSNAP_ATTR_EXPOSED_REMOTELY, // LONG lAttributes
             L"dirTwoOfF",                    // VSS_PWSZ wszExpose
             LPWSTR &wszExposed,              // VSS_PWSZ* pwszExposed
     );
```

If the shadow copy was successfully exposed remotely, *wszExposed* should contain the wide character string "dirTwoOfF."

Any system currently mapping the network share of dirTwoOfF could disconnect from it, just as it might disconnect from any ordinary share.

Surfacing a Shadow Copy

A *surfaced shadow copy* is one in which the shadow copy is known to a system's Mount Manager namespace.

This means that you can locate such shadow copies just as you would locate any other available but not-yet-mounted volume—by using **FindFirstVolume** and **FindNextVolume**, for example.

Clearly, then, exposed shadow copies are also surfaced shadow copies. However, the reverse is not necessarily true.

If a locally exposed shadow copy was dismounted, or a system chose to disconnect a remotely exposed shadow copy, that shadow copy would no longer be exposed. However, as long as the shadow copy persisted, the volumes would be surfaced. This means they could be mounted like any other read-only volume.

Breaking Shadow Copies

3/5/2021 • 2 minutes to read • Edit Online

A requester can break a shadow copy set by using the **IVssBackupComponents::BreakSnapshotSet** or **IVssBackupComponentsEx2::BreakSnapshotSetEx** method.

A broken shadow copy is a volume that contains a shadow copy that is no longer managed by VSS. Breaking a shadow copy has meaning only if the shadow copy's *provider* is a *hardware provider*. This is because only hardware providers can implement a shadow copy as an actual volume with a life cycle independent of VSS. If VSS does not manage such a volume, it still is available.

Software providers maintain shadow copies only while involved in VSS operations. For that reason, breaking a shadow copy has no meaning for software providers.

If the shadow copy was managed by a hardware provider, the requester must import the shadow copy before calling **BreakSnapshotSet** or **BreakSnapshotSetEx**. In case of non-transportable (created by a hardware provider) shadow copies, they are imported implicitly as part of the **IVssBackupComponents::DoSnapshotSet** call.

After the requester has called **BreakSnapshotSet** or **BreakSnapshotSetEx**, the shadow copy set is still accessible via its device objects or other access paths, but it is no longer a shadow copy set. It can be managed as one or more LUNs by using the Virtual Disk Service (VDS) COM interfaces. Using VDS, a requester can convert the LUNs to read/write, mount them with drive letters, or mask/unmask them to other computers. See the VDS API documentation for more information.

Troubleshooting VSS Applications

3/5/2021 • 2 minutes to read • Edit Online

The following topics provide information about troubleshooting VSS applications:

- Event and Error Handling Under VSS
- VSS Security Issues
- Special VSS Usage Cases

Event and Error Handling Under VSS

3/5/2021 • 2 minutes to read • Edit Online

VSS provides a well-defined error handling mechanism to allow developers and administrators to manage the system and handle errors:

- Determining Writer Status
- VSS Error Logging
- Writer Errors and Vetoes
- Aborting VSS Operations
- Handling BackupShutdown Events

Determining Writer Status

3/8/2021 • 2 minutes to read • Edit Online

A requester needs to have a well-defined understanding about the status of the writer that participates with it during shadow copy creation, and during backup and restore operations. To do so, it is recommended:

- Requesters use **IVssBackupComponents::GatherWriterStatus**, **IVssBackupComponents::GetWriterStatusCount**, and **IVssBackupComponents::GetWriterStatus**.
- As described in Overview of Processing a Backup Under VSS and Overview of Processing a Restore Under VSS, these methods are most useful when called in a well-defined backup or restore sequence. Typically, this means that the writers should be queried after a requester has completed one of its tasks and generated a VSS event.

  When processing a backup, a requester should query a writer following the completion of the following methods. Requesters must call **GatherWriterStatus** after calling **BackupComplete** to cause the writer session to be set to a completed state.

  > **NOTE**
  > This is only necessary on Windows Server 2008 with Service Pack 2 (SP2) and earlier.

IVssBackupComponents::PrepareForBackup

IVssBackupComponents::DoSnapshotSet

IVssBackupComponents::BackupComplete

During restore operations, a requester should query a writer after completion of these methods:

IVssBackupComponents::PreRestore

IVssBackupComponents::PostRestore

- Calls to **IVssBackupComponents::GatherWriterStatus** that are not part of a well-defined backup or restore sequence do not provide a reliable picture of writer status, because they might reflect conditions that do not indicate failure in the current operation, such as:
  - A failure of a previous shadow copy creation
  - An error in an early backup or restore operation
  - An unresponsive writer currently processing an event

Therefore, developers should not rely on writer status returned by processes other then the requester or attempt to monitor the progress of one instance of the **IVssBackupComponents** interface with another (possibly in a separate thread).

Note that for backup operations, where it is necessary to examine writers' Writer Metadata Documents, there is no need for a requester call to **IVssBackupComponents::GatherWriterStatus** and **IVssBackupComponents::GetWriterStatus** following the generation and handling of the *Identify* event caused by **IVssBackupComponents::GatherWriterMetdata**.

IVssBackupComponents::GetWriterStatus reports only the status of those writers whose metadata was

provided to VSS by writers' *Identify* event handlers, **CVssWriter::OnIdentify** (and returned to the requester by **IVssBackupComponents::GetWriterMetadataCount** and **IVssBackupComponents::GetWriterMetadata**).

If a writer's implementation of **CVssWriter::OnIdentify** fails, that writer's metadata will not be part of the list of Writer Metadata Documents provided to VSS, no status information will be available, and the call would be redundant.

For restore operations, where the requester does not need to examine Writer Metadata Documents of executing writers, calling **IVssBackupComponents::GatherWriterStatus** and **IVssBackupComponents::GetWriterStatus** may be a more efficient way to determine which writers are executing.

VSS Error Logging

3/5/2021 • 2 minutes to read • Edit Online

The following VSS error and state information is written to the Application Event Log:

- Requester errors produced using the **IVssBackupComponents** interface
- Writer errors produced in using the **CVssWriter** class, including overriding methods
- Default-provider-generated errors
- VSS service errors generated in coordinating provider, writer, and requester activity (such as the generation of events)

These errors might have a number of causes, including a programming error in third-party code or VSS-related configuration errors.

VSS drivers and lower-level implementation functionality write errors to the System Log. Third-party software (requester, provider, writer) can choose the Application Log, the System Log, or both, to write error log entries.

It is recommended that high-level applications (such as user-mode code) use the Application Log. Lower-level applications, such as hardware interfaces and drivers, should use the System Log.

Writer Errors and Vetoes

3/5/2021 • 2 minutes to read • Edit Online

A writer can fail for numerous programmatic reasons. When this happens, it should veto the ongoing backup, restore, or shadow copy operation by calling the **CVssWriter::SetWriterFailure** method in one of its handler methods (for example, **CVssWriter::OnFreeze** or **CVssWriter::OnPreRestore**) and returning **TRUE**. It can also optionally set an error message string in response to a failure condition in certain handler methods with the **IVssComponentEx::SetPrepareForBackupFailureMsg**, **IVssComponentEx::SetPostSnapshotFailureMsg**, **IVssComponent::SetPreRestoreFailureMsg**, and **IVssComponent::SetPostRestoreFailureMsg** methods. The requester can accept the veto or continue with the backup, ignoring the veto.

A requester should check the writer status (using **IVssBackupComponents::GatherWriterStatus** and **IVssBackupComponents::GetWriterStatus**) following each event it generates.

In some cases, error messages can be retrieved from these failures (using the **IVssComponentEx::GetPrepareForBackupFailureMsg**, **IVssComponent::GetPreRestoreFailureMsg**, **IVssComponentEx::GetPostSnapshotFailureMsg**, and **IVssComponent::GetPostRestoreFailureMsg** methods), or a writer may choose to set metadata (using **IVssComponent::SetRestoreMetadata** and **IVssComponent::SetBackupMetadata** with error state information). For example code that shows how to view such error messages, see **IVssComponentEx::GetPrepareForBackupFailureMsg**.

Depending on the error state, a requester or its operator could restart the backup and shadow copy with any necessary modification to the state of the backup job or system.

For example, suppose **GetWriterStatus** returned the following:

- **VSS_E_WRITERERROR_INCONSISTENTSNAPSHOT** suggests that a requester might add additional volumes to the shadow copy
- **VSS_E_WRITERERROR_RETRYABLE** indicates that retrying without reconfiguration might work. If the writer continues to return the error after several retries, try restarting the service that hosts the writer. The following writers are hosted in the VSS service: registry writer, COM+ class registration database writer, shadow copy optimization writer, and Automated System Recovery (ASR) writer. If the writer belongs to an application that hosts the writer in its own process, try restarting the application.

  Windows Server 2003 and Windows XP: The following writers are hosted in the VSS service: registry writer, COM+ class registration database writer, application event log writer, and Microsoft SQL Server 2000 Desktop Engine (MSDE) writer.
- VSS_E_WRITER_STATUS_NOT_AVAILABLE indicates that a writer may have reached the maximum number of available backup and restore sessions, and retrying might work when the system is less busy.
- **VSS_E_WRITERERROR_OUTOFRESOURCES** or **VSS_E_WRITERERROR_TIMEOUT** might suggest that system load be reduced prior to retrying
- **VSS_E_WRITERERROR_NONRETRYABLE** or **VSS_E_WRITER_NOT_RESPONDING** would likely indicate a writer error so severe as to preclude trying to back up its data with VSS.

Depending on which writer and which components generate them, it is not always necessary for a backup application to abort following a veto or error.

For example, a requester may decide that the intent of the shadow copy is to back up application A and the veto has been received from the writer for backup application B. In this case, it is perfectly acceptable to continue to

back up application A while ignoring the veto.

The following are examples of a writer veto:

- The writer vetoes the shadow copy creation process when it could not suspend its activities during the time the shadow copy was being created. This indicates that there is a high probability that the shadow copy is not valid because a write operation has occurred during the Freeze state.
- A backup application has requested a shadow copy of only volume C: and a writer determines that a shadow copy of C: and D: is to back up its data. In this case, the writer will veto. The backup application may examine the metadata and determine whether the writer will be ignored or the shadow copy creation process will be aborted and later restarted.

Aborting VSS Operations

3/5/2021 • 2 minutes to read • Edit Online

Abort events can be generated during a backup operation in any of the following cases:

- A requester explicitly generates an *Abort event* by calling **IVssBackupComponents::AbortBackup**.
- A writer's *Freeze* and *Thaw* event handlers (**CVssWriter::OnFreeze** and **CVssWriter::OnThaw**) return **false**, or cannot complete in the time window specified in **CVssWriter::Initialize**.
- There is any failure of the provider or VSS during the creation of a shadow copy following the *PrepareForSnapshot* event.

Aborts are not supported for restore operations.

Requester Handling and Creation of Abort Events

An instance of the **IVSSBackupComponents** interface can be used for only one backup, so if an error occurs in processing a backup it is generally best to release the current instance and start over.

A requester should explicitly signal that it is aborting a backup operation (using **IVssBackupComponents::AbortBackup**) only after the actual preparation for a backup, involving writers, or the creation of a shadow copy has taken place.

Effectively, this means that any time a requester needs to stop a backup operation after generating a *PrepareForBackup* event by calling **IVssBackupComponents::PrepareForBackup**, it should call **IVssBackupComponents::AbortBackup** and await its return prior to releasing the current **IVSSBackupComponents** instance.

For example, if a writer vetoed a backup operation, a requester should use **IVssBackupComponents::AbortBackup** to notify all other writers that the backup operation will not be completed.

Prior to calling **PrepareForBackup**, or if the call to **PrepareForBackup** fails, a requester can release the current instance of the **IVSSBackupComponents** interface without needing to generate an Abort event.

For example, if the current instance of **IVSSBackupComponents** is being used merely to obtain information by calling **IVssBackupComponents::GatherWriterMetadata** and generating an *Identify* event, once information has been returned the instance of **IVSSBackupComponents** can simply be released.

A requester generates a number of events (*PrepareForSnapshot*, *Freeze*, *Thaw*, and *PostSnapshot*) when it calls **IVssBackupComponents::DoSnapshotSet**. While handling the Freeze and Thaw events, a writer may fail and can generate an Abort event on its own. Failure to handle PrepareForSnapshot and PostSnapshot events does not generate an Abort event.

It is not always possible for a requester to know if an Abort event was generated when **IVssBackupComponents::DoSnapshotSet** indicates failure. Therefore, a requester that needs to terminate a backup operation because the status of **IVssBackupComponents::DoSnapshotSet** indicates a problem should still call **IVssBackupComponents::AbortBackup**.

If a requester has called **IVssBackupComponents::AbortBackup**, it is not necessary to call **IVssBackupComponents::BackupComplete** prior to releasing an instance of **IVSSBackupComponents**.

Writer Handling and Creation of Abort Events

As noted previously, a writer can receive an Abort event from a requester, or the provider can trigger one itself. Also, it is possible for a writer to receive multiple Abort events under certain circumstances. Writer developers should code any implementation of **CVssWriter::OnAbort** with this in mind.

In handling an Abort event, a writer should attempt to restore whatever process it managed to its normal running state, as well as perform any error handling and logging.

This may mean that an implementation of **CVssWriter::OnAbort** might have to perform many, if not all, of the same tasks as the Thaw event handler (**CVssWriter::OnThaw**) and the PostSnapshot event handler (**CVssWriter::OnPostSnapshot**), and these handlers can be called from within **CVssWriter::OnAbort**.

Handling BackupShutdown Events

3/5/2021 • 2 minutes to read • Edit Online

It is possible for a backup application (requester) to terminate and not generate a *BackupComplete* event. The backup application could crash, or be terminated (from the Task Manager, for example) and not be able to call **IVssBackupComponents::BackupComplete**.

Therefore, the VSS infrastructure (rather than the requester) generates a *BackupShutdown* event whenever an instance of **IVssBackupComponents** participating in a backup is released, whether it is released by the requester or by the system.

If a backup proceeds properly, a writer will receive a BackupComplete event followed by a BackupShutdown event.

If the operation aborts (the requester generates an *Abort event* by calling **IVssBackupComponents::AbortBackup**) or fails abruptly, a writer may receive only a BackupShutdown event, and it may not receive other events that perform cleanup operations. It is up to a writer to determine whether a BackupShutdown event follows a proper sequence of events, or represents an unexpected failure of the backup operations.

The BackupShutdown event handler, **CVssWriter::OnBackupShutdown**, receives the VSS_ID (GUID) of the shadow copy set of the backup operation being shut down. The writer can use this to determine which backup operation is being shut down, if it has stored the shadow copy set ID during its backup sequence (for example, from within **CVssWriter::OnFreeze**, **CVssWriter::OnThaw**, or **CVssWriter::OnPostSnapshot**) by using **CVssWriter::GetCurrentSnapshotSetId**.

However, a writer should not call **CVssWriter::GetCurrentSnapshotSetId** from within **CVssWriter::OnBackupShutdown**. Also, **CVssWriter::GetCurrentSnapshotSetId** cannot be called after **CVssWriter::OnPostSnapshot** returns.

It is possible for the writer to be involved in multiple backup operations, and if a BackupShutdown event is called because of an abrupt shutdown of a requester, the VSS_ID returned could be that of another backup operation the writer was participating in.

VSS Security Issues

3/5/2021 • 2 minutes to read • Edit Online

The VSS infrastructure is internally based on COM, and therefore allowing writers and requesters to communicate requires COM-based security. COM security provides mechanisms and guidelines that a VSS developer should consider:

- Security Considerations for Writers
- Security Considerations for Requesters

Security Considerations for Writers

3/5/2021 • 5 minutes to read • Edit Online

The VSS infrastructure requires writer processes to be able to function both as COM clients and as servers.

When acting as servers, VSS writers expose COM interfaces (for example, the VSS event handlers such as **CVssWriter::OnIdentify**) and receive incoming COM calls from VSS processes (such as requesters and the VSS service) or RPC calls from processes that are external to VSS, typically when these processes generate VSS events (for example, when a requester calls **IVssBackupComponents::GatherWriterMetadata**). Therefore, a VSS writer needs to securely manage which COM clients are able to make incoming COM calls into its process.

Similarly, VSS writers may also act as COM clients, making outgoing COM calls to callbacks supplied by the VSS infrastructure or RPC calls to processes that are external to VSS. These callbacks that are provided either by a backup application or by the VSS service allow the writer to perform tasks such as updating the Backup Components Document through the **IVssComponent** interface. Therefore, VSS security settings must allow writers to make outgoing COM calls into other VSS processes.

The simplest mechanism for managing writer security issues involves the proper selection of the user account under which it runs. A writer typically needs to run under a user who is a member of either the Administrators group or the Backup Operators group, or it needs to run as the Local System account.

By default, when a writer is acting as a COM client, if it does not run under these accounts, any COM call it makes is automatically rejected with **E_ACCESSDENIED** without even getting to the COM method implementation.

Disabling COM Exception Handling

When developing a writer, set the COM COMGLB_EXCEPTION_DONOT_HANDLE global options flag to disable COM exception handling. It is important to do this because COM exception handling can mask fatal errors in a VSS application. The error that is masked can leave the process in an unstable and unpredictable state, which can lead to corruptions and hangs. For more information about this flag, see IGlobalOptions.

Setting Writer Default COM Access Check Permission

Writers need to be aware that when their processes act as a server (for example, to handle VSS events), they must allow incoming calls from other VSS participants, such as requesters or the VSS service.

However, by default, a process will allow only COM clients that are running under the same logon session the SELF SID) or running under the Local System account. This is a potential problem because these defaults are not sufficient to support the VSS infrastructure. For example, requesters may run as a "Backup Operator" user account that is neither in the same logon session as the writer process nor a Local System account.

To handle this type of problem, every COM server process can exercise further control over whether an RPC or COM client is allowed to perform a COM method the server (a writer in this case) implements by using **CoInitializeSecurity** to set a process-wide default COM access check permission.

Writers can explicitly do the following:

- Allow all processes access to call into the writer process.

  This option may be adequate for many writers, and is used by other COM servers—for example, all SVCHOST-based Windows services are already using this option, as are all COM+ Services by default.

Allowing all processes to perform incoming COM calls is not necessarily a security weakness. A writer acting as a COM server, like all other COM servers, always retains the option of authorizing its clients on every COM method implemented in its process.

To allow all processes COM access to a writer, you can pass a **NULL** security descriptor as the first parameter of **CoInitializeSecurity**. (Note that **CoInitializeSecurity** must be called at most once for the entire process. Please see the COM documentation for more details on **CoInitializeSecurity**.)

The following is a code example that includes a call to **CoInitializeSecurity**:

```
// Initialize COM security.
hr = CoInitializeSecurity(
       NULL,                        // PSECURITY_DESCRIPTOR          pSecDesc,
       -1,                          // LONG                          cAuthSvc,
       NULL,                        // SOLE_AUTHENTICATION_SERVICE  *asAuthSvc,
       NULL,                        // void                         *pReserved1,
       RPC_C_AUTHN_LEVEL_PKT_PRIVACY, // DWORD                       dwAuthnLevel,
       RPC_C_IMP_LEVEL_IDENTIFY,    // DWORD                         dwImpLevel,
       NULL,                        // void                         *pAuthList,
       EOAC_NONE,                   // DWORD                         dwCapabilities,
       NULL                         // void                         *pReserved3
    );
```

When explicitly setting a writer's COM-level security with **CoInitializeSecurity**, you should do the following:

- Set the authentication level to at least **RPC_C_AUTHN_LEVEL_CONNECT**.

  For better security, consider using **RPC_C_AUTHN_LEVEL_PKT_PRIVACY**.

- Set the impersonation level to **RPC_C_IMP_LEVEL_IDENTIFY** unless the writer process needs to allow impersonation for specific RPC or COM calls that are unrelated to VSS.

- Allow only specified processes access to call into the writer process.

  A COM server (such as a writer) that is calling **CoInitializeSecurity** with a non-**NULL** security descriptor can use the descriptor to configure itself to accept incoming calls only from users that belong to a specific set of accounts.

  A writer must ensure that COM clients running under valid users are authorized to call into its process. A writer that specifies a Security Descriptor in the first parameter must allow the following users to perform incoming calls into the requester process:

  - Local System
  - Members of the local Administrators group
  - Members of the local Backup Operators group
  - The account under which the writer is running

Explicitly Controlling User Account Access to a Writer

There are cases where restricting access to a writer to processes running as Local System, or under the local Administrators or local Backup Operators local groups, may be too restrictive.

For example, a writer process (perhaps a third-party non-system writer) might not ordinarily need to be run under an Administrator or Backup Operator account. For security reasons, it would be best not to artificially promote the process's privileges to support VSS.

In these cases, the **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\VSS\VssAccessControl** registry key

must be modified to instruct VSS that a specified user is safe to run a VSS writer.

Under this key, you must create a subkey that has the same name as the account that is to be granted or denied access. This subkey must be set to one of the values in the following table.

VALUE	MEANING
0	Deny the user access to your writer and requester.
1	Grant the user access to your writer.
2	Grant the user access to your requester.
3	Grant the user access to your writer and requester.

The following example grants access to the "MyDomain\MyUser" account:

```
HKEY_LOCAL_MACHINE
   SYSTEM
      CurrentControlSet
         Services
            VSS
               VssAccessControl
                  MyDomain\MyUser = 1<dl>
<dt>

                  Data type
</dt>
<dd>                  REG_DWORD</dd>
</dl>
```

This mechanism can also be used to explicitly restrict an otherwise permitted user from running a VSS writer. The following example will restrict access from the "ThatDomain\Administrator" account:

```
HKEY_LOCAL_MACHINE
   SYSTEM
      CurrentControlSet
         Services
            VSS
               VssAccessControl
                  ThatDomain\Administrator = 0<dl>
<dt>

                  Data type
</dt>
<dd>                  REG_DWORD</dd>
</dl>
```

The user ThatDomain\Administrator would not be able to run a VSS writer.

Security Considerations for Requesters

3/5/2021 • 6 minutes to read • Edit Online

The VSS infrastructure requires VSS requesters, such as backup applications, to be able to function both as COM clients and as a server.

When acting as a server, a requester exposes a set of COM callback interfaces that can be invoked from external processes (such as writers or the VSS service). Therefore, requesters need to securely manage which COM clients are able to make incoming COM calls into its process.

Similarly, requesters can act as a COM client for the COM APIs supplied by a VSS writer or by the VSS service. The requester-specific security settings must allow outgoing COM calls from the requester to these other processes.

The simplest mechanism for managing a requester's security issues involves proper selection of the user account under which it runs.

A requester typically needs to run under a user that is a member of either the Administrators group or the Backup Operators group, or run as the Local System account.

By default, when a requester is acting as a COM client, if it does not run under these accounts, any COM call is automatically rejected with **E_ACCESSDENIED**, without even getting to the COM method implementation.

Disabling COM Exception Handling

When developing a requester, set the COM COMGLB_EXCEPTION_DONOT_HANDLE global options flag to disable COM exception handling. It is important to do this because COM exception handling can mask fatal errors in a VSS application. The error that is masked can leave the process in an unstable and unpredictable state, which can lead to corruptions and hangs. For more information about this flag, see IGlobalOptions.

Setting Requester Default COM Access Check Permission

Requesters need to be aware that when their process acts as a server (for example, allowing writers to modify the Backup Components Document), they must allow incoming calls from other VSS participants, such as writers or the VSS service.

However, by default, a Windows process will allow only COM clients that are running under the same logon session (the SELF SID) or running under the Local System account. This is a potential problem because these defaults are not adequate for the VSS infrastructure. For example, writers might run as a "Backup Operator" user account that is neither in the same logon session as the requester process nor a Local System account.

To handle this type of problem, every COM server process can exercise further control over whether an RPC or COM client is allowed to perform a COM method implemented by the server (a requester in this case) by using **CoInitializeSecurity** to set a process-wide "Default COM Access Check Permission".

Requesters can explicitly do the following:

- Allow all processes access to call into the requester process.

  This option may be adequate for the vast majority of requesters, and is used by other COM servers—for example, all SVCHOST-based Windows services are already using this option, as are all COM+ Services by default.

  Allowing all processes to perform incoming COM calls is not necessarily a security weakness. A requester

acting as a COM server, like all other COM servers, always retains the option to authorize its clients on every COM method implemented in its process.

Note that internal COM callbacks implemented by VSS are secured by default.

To allow all processes COM access to a requester, you can pass a **NULL** security descriptor as the first parameter of **CoInitializeSecurity**. (Note that **CoInitializeSecurity** must be called at most once for the entire process. Please see the COM documentation or MSDN for more information on **CoInitializeSecurity** calls.)

The following code example shows how a requester should call **CoInitializeSecurity** in Windows 8 and Windows Server 2012 and later, in order to be compatible with VSS for remote file shares (RVSS):

```
// Initialize COM security.
   hr = CoInitializeSecurity(
        NULL,                           //  PSECURITY_DESCRIPTOR         pSecDesc,
        -1,                             //  LONG                         cAuthSvc,
        NULL,                           //  SOLE_AUTHENTICATION_SERVICE *asAuthSvc,
        NULL,                           //  void                        *pReserved1,
        RPC_C_AUTHN_LEVEL_PKT_PRIVACY, //  DWORD                        dwAuthnLevel,
        RPC_C_IMP_LEVEL_IMPERSONATE,   //  DWORD                        dwImpLevel,
        NULL,                           //  void                        *pAuthList,
        EOAC_STATIC,                    //  DWORD                        dwCapabilities,
        NULL                            //  void                        *pReserved3
        );
```

When explicitly setting a requester's COM level security with **CoInitializeSecurity**, you should do the following:

- Set the authentication level to at least **RPC_C_AUTHN_LEVEL_PKT_INTEGRITY**. For better security, consider using **RPC_C_AUTHN_LEVEL_PKT_PRIVACY**.
- Set the impersonation level to **RPC_C_IMP_LEVEL_IMPERSONATE**.
- Set the cloaking security capabilities to **EOAC_STATIC**. For more information about cloaking security, see Cloaking.

The following code example shows how a requester should call **CoInitializeSecurity** in Windows 7 and Windows Server 2008 R2 and earlier (or in Windows 8 and Windows Server 2012 and later, if RVSS compatibility is not needed):

```
// Initialize COM security.
   hr = CoInitializeSecurity(
        NULL,                           //  PSECURITY_DESCRIPTOR         pSecDesc,
        -1,                             //  LONG                         cAuthSvc,
        NULL,                           //  SOLE_AUTHENTICATION_SERVICE *asAuthSvc,
        NULL,                           //  void                        *pReserved1,
        RPC_C_AUTHN_LEVEL_PKT_PRIVACY, //  DWORD                        dwAuthnLevel,
        RPC_C_IMP_LEVEL_IDENTIFY,       //  DWORD                        dwImpLevel,
        NULL,                           //  void                        *pAuthList,
        EOAC_NONE,                      //  DWORD                        dwCapabilities,
        NULL                            //  void                        *pReserved3
        );
```

When explicitly setting a requester's COM level security with **CoInitializeSecurity**, you should do the following:

- Set the authentication level to at least **RPC_C_AUTHN_LEVEL_CONNECT**. For better security, consider using **RPC_C_AUTHN_LEVEL_PKT_PRIVACY**.
- Set the impersonation level to **RPC_C_IMP_LEVEL_IDENTIFY** unless the requester process needs to allow impersonation for specific RPC or COM calls that are unrelated to VSS.

- Allow only specified processes access to call into the requester process.

  A COM server (such as a requester) that is calling **CoInitializeSecurity** with a non-**NULL** security descriptor as the first parameter can use the descriptor to configure itself to accept incoming calls only from users that belong to a specific set of accounts.

  A requester must ensure that COM clients running under valid users are authorized to call into its process. A requester that specifies a Security Descriptor in the first parameter must allow the following users to perform incoming calls into the requester process:

  - Local System
  - Local Service

    Windows XP: This value is not supported until Windows Server 2003.
  - Network Service

    Windows XP: This value is not supported until Windows Server 2003.
  - Members of the local Administrators group
  - Members of the local Backup Operators group
  - Special users that are specified in the registry location below, with "1" as their REG_DWORD value

Explicitly Controlling User Account Access to a Requester

There are cases where restricting access to a requester to processes running as Local System, or under the local Administrators or local Backup Operators groups, may be too restrictive.

For example, a specified requester process might not ordinarily need to be run under an Administrator or Backup Operator account. For security reasons, it would be best not to artificially promote the processes privileges to support VSS.

In these cases, the **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\VSS\VssAccessControl** registry key must be modified to instruct VSS that a specified user is safe to run a VSS requester.

Under this key, you must create a subkey that has the same name as the account that is to be granted or denied access. This subkey must be set to one of the values in the following table.

VALUE	MEANING
0	Deny the user access to your writer and requester.
1	Grant the user access to your writer.
2	Grant the user access to your requester.
3	Grant the user access to your writer and requester.

The following example grants access to the "MyDomain\MyUser" account:

```
HKEY_LOCAL_MACHINE
   SYSTEM
      CurrentControlSet
         Services
            VSS
               VssAccessControl
                  MyDomain\MyUser = 2<dl>
<dt>

                  Data type
</dt>
<dd>                  REG_DWORD</dd>
</dl>
```

This mechanism can also be used to explicitly restrict an otherwise permitted user from running a VSS requester. The following example will restrict access from the "ThatDomain\Administrator" account:

```
HKEY_LOCAL_MACHINE
   SYSTEM
      CurrentControlSet
         Services
            VSS
               VssAccessControl
                  ThatDomain\Administrator = 0<dl>
<dt>

                  Data type
</dt>
<dd>                  REG_DWORD</dd>
</dl>
```

The user ThatDomain\Administrator would not be able to run a VSS requester.

Performing a File Backup of the System State

If a requester performs system-state backup by backing up individual files instead of using a volume image for the backup, it must call the **FindFirstFileNameW** and **FindNextFileNameW** functions to enumerate hard links on files that are located in the following directories:

- Windows\system32\WDI\perftrack\
- Windows\WINSXS\

These directories can only be accessed by members of the Administrators group. For this reason, such a requester must run under the system account or a user account that is a member of the Administrators group.

Windows XP and Windows Server 2003: The **FindFirstFileNameW** and **FindNextFileNameW** functions are not supported until Windows Vista and Windows Server 2008.

Special VSS Usage Cases

3/5/2021 • 2 minutes to read • Edit Online

There are cases where VSS applications that implement backup and restore applications should use special care. These include the following:

- Stopping Services for Restore by Requesters
- Backing Up and Restoring System State in Windows Server 2003 R2 and Windows Server 2003 SP1
- Using VSS Automated System Recovery for Disaster Recovery
- Custom Backups and Restores

Stopping Services for Restore by Requesters

3/5/2021 • 2 minutes to read • Edit Online

It may be necessary for a service to be stopped prior to and restarted following a restore operation.

Typically, stopping and starting a service to support a restore would be performed by a writer when handling the *PreRestore* event (with **CVssWriter::OnPreRestore**) and the *PostRestore* event (with **CVssWriter::OnPostRestore**).

However, there may be cases when it is necessary for a requester to explicitly stop a running service. Writers indicate if this is the case by setting the VSS_RME_STOP_RESTORE_START or VSS_RME_RESTORE_STOP_START value of the **VSS_RESTOREMETHOD_ENUM** enumeration as the restore method argument of a call to the **IVssCreateWriterMetadata::SetRestoreMethod** method and specifying the name of the service to be stopped.

A requester obtains information about the restore method and the name of the service to be stopped when working with writer metadata by using the **IVssExamineWriterMetadata::GetRestoreMethod** method.

It is important that the writer, when specifying the name of a service to be stopped, uses the correct publicly known name of that service. An ambiguous or inaccurate name may result in requesters stopping the wrong service or being unable to determine which service to stop.

After the completion of the restore operation, requesters must restart the service.

You must be careful in designing and implementing writers that support services that requesters must stop and restart. Specifically, such writers should not actually be part of the service—that is, the writer itself should not need to be stopped and then restarted in the course of the restore operation.

A writer whose process is stopped will have a different writer instance upon restart. The new instance of the writer will not receive VSS events intended for the original instance of the writer. Specifically, if the process of a writer instance is stopped after handling a PreRestore event, the new instance will not receive the PostRestore event. Further, VSS will generate an error indicating the loss of a participating writer in the restore operation, and **IVssBackupComponents::PostRestore** may return a failure.

Backing Up and Restoring System State in Windows Server 2003 R2 and Windows Server 2003 SP1

3/5/2021 • 4 minutes to read • Edit Online

NOTE

This topic only applies to Windows Server 2003 R2 and Windows Server 2003 with Service Pack 1 (SP1). For information about other operating system versions, see Backing Up and Restoring System State.

NOTE

Microsoft does not provide developer or IT professional technical support for implementing online system state restores on Windows (all releases). For information about using Microsoft-provided APIs and procedures to implement online system state restores, see the community resources available at the MSDN Community Center.

When performing a VSS backup or restore, the Windows system state is defined as being a collection of several key operating system elements and their files. These elements should always be treated by backup and restore operations as a unit.

In Windows Server 2003 R2 and Windows Server 2003 with SP1, there is no Windows API designed to treat these objects as one, so it is recommended that requesters have their own system state object so that they can process these components in a consistent manner.

Because VSS runs on versions of Windows where *System File Protection* (WFP) protects system state files from corruption, special steps are needed to back up and restore system state.

System state consists of the following:

- All files protected by WFP, boot files including ntldr, ntdetect, and performance counter configurations
- The Active Directory (ADSI) (on systems that are domain controllers)
- The System Volume Folder (SYSVOL) that is replicated by the File Replication Service (FRS) (on systems that are domain controllers)
- Certificate server (on systems that provide Certification Authority)
- Cluster database (on systems that are a node of a Windows cluster)
- Registry service
- COM+ class registration database

The system state can be backed up in any order.

However, the restoration of the system state should replace boot files first and commit the system section (hive) of the registry as a final step in the process, and might occur in the following order:

1. Restore the boot files.
2. COM+ class registration database
3. Restore SYSVOL, Certificate Server, and Cluster databases (if needed).

4. Restore Active Directory (if needed).
5. Restore the registry.

Some system services, such as the Certification Authority, have conventional VSS writers and follow the VSS backup and restore algorithms. Others, such as the registry, require custom backup or restore operations; for more information, see Custom Backups and Restores.

VSS Backup and Restores of Boot and System Files

The boot and system files should be backed up and restored only as a single entity.

A requester can safely use shadow-copied versions of these files, and should be sure that the system volume and boot device are *shadow copied*.

The system and boot files include:

- The core boot files:

  NtLdr.exe
  Boot.ini
  NtDetect.com
  NtBootDD.sys (if present)

- The WFP service catalog file must be backed up prior to backing up the WFP files, and it is found under:

  %SystemRoot%\System32\CatRoot\{F750E6C3-38EE-11D1-85E5-00C04FC295EE}

- All files protected by *System File Protection* and enumerated by **SfcGetNextProtectedFile** (see VSS Restore Operations of WFP Protected Files)
- The Performance Counter Configuration files:

  %SystemRoot%\System32\Perf?00?.dat
  %SystemRoot%\System32\Perf?00?.bak

- If present, the Internet Information Server (IIS) metabase file should be included in backup and restore operations because it contains state that is used by Microsoft Exchange and other network applications. This file can be found at:

  %SystemRoot%\System32\InetSrv\Metabase.bin

- If the IIS metabase file is backed up, keys to enable applications to read certain encrypted entries should be restored as part of the system state. The files can be found under:

  %SystemRoot%\System32\Microsoft\Protect*
  %AllUsersProfile%\Microsoft\Crypto\RSA\MachineKeys*

- When backing up boot and system files, it may become necessary to determine the DOS boot device by doing the following:
  1. Find the system partition under **HKEY_LOCAL_MACHINE\System\Setup\SystemPartition**.
  2. Pass the System Root environment variable (%SystemRoot%) to the mount manager to obtain the NT device name.

VSS Restore Operations of WFP Protected Files

The WFP service is designed to prevent accidental or piecemeal replacement of system files. Therefore, special steps need to be undertaken to restore WFP data.

The means the WFP writer should specify the **VSS_RME_RESTORE_AT_REBOOT** restore method when

defining its Writer Metadata Document. If a requester determines that the WFP writer has failed to specify this restore method, it indicates a writer error.

A requester should implement a restore method of **VSS_RME_RESTORE_AT_REBOOT** using the Win32 function **MoveFileEx** with the **MOVEFILE_DELAY_UNTIL_REBOOT** parameter to replace system files. The restored files are not copied into the actual system file directories until after system reboot. The overwriting of protected system files will occur only if the value of the following **REG_WORD** registry entry is set to 1:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\AllowProtectedRenames = 1

This value must be set before any boot where protected files are to be replaced via **MoveFileEx** and is deleted after reboot.

The system dllcache directory should also be backed up or restored, with boot volume backup and restore, and is located by examining the **REG_EXPAND_SZ** registry entry:

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\WinLogon\SfcDllCache

Data type

REG_EXPAND_SZ

The contents of the system dllcache directory are rebuilt by using the System File Checker (SFC) from the command prompt:

- The **/SCANONCE** option scans all protected files at the next system boot.
- The **/SCANNOW** option scans all protected files immediately.

All protected files will be scanned, and any versions that are not valid will be replaced in both the directory and dllcache location.

There are four files that are part of the WFP that cannot be restored with the WFP files:

Ctl3dv2.dll DtcSetup.exe NtDll.dll Smss.exe

These files help in the process of restoring the WFP files and as such there is a circular dependency. Currently, there is no way to restore these files except to reinstall Windows.

Using VSS Automated System Recovery for Disaster Recovery

3/5/2021 • 14 minutes to read • Edit Online

A VSS backup-and-recovery application that performs disaster recovery (also called bare-metal recovery) can use the Automated System Recovery (ASR) writer together with Windows Preinstallation Environment (Windows PE) to back up and restore critical volumes and other components of the bootable system state. The backup application is implemented as a VSS requester.

Note Applications that use ASR must license Windows PE.

Windows Server 2003 and Windows XP: ASR is not implemented as a VSS writer.

For information about tracing tools that you can use with ASR, see Using Tracing Tools with VSS ASR Applications.

In this article:

Overview of Backup Phase Tasks

At backup time, the requester performs the following steps.

> **NOTE**
> All steps are required unless otherwise indicated.

1. Call the **CreateVssBackupComponents** function to create an instance of the **IVssBackupComponents** interface and call the **IVssBackupComponents::InitializeForBackup** method to initialize the instance to manage a backup.
2. Call **IVssBackupComponents::SetContext** to set the context for the shadow copy operation.
3. Call **IVssBackupComponents::SetBackupState** to configure the backup. Set the *bBackupBootableSystemState* parameter to **true** to indicate that the backup will include a bootable system state.
4. Choose which critical components in the ASR writer's Writer Metadata Document to back up and call **IVssBackupComponents::AddComponent** for each of them.
5. Call **IVssBackupComponents::StartSnapshotSet** to create a new, empty shadow copy set.
6. Call **IVssBackupComponents::GatherWriterMetadata** to initiate asynchronous contact with writers.
7. Call **IVssBackupComponents::GetWriterMetadata** to retrieve the ASR writer's Writer Metadata Document. The writer ID for the ASR writer is BE000CBE-11FE-4426-9C58-531AA6355FC4, and the writer name string is "ASR Writer".

8. Call **IVssExamineWriterMetadata::SaveAsXML** to save a copy of the ASR writer's Writer Metadata Document.
9. Call **IVssBackupComponents::AddToSnapshotSet** for each volume that can participate in shadow copies to add the volume to the shadow copy set.
10. Call **IVssBackupComponents::PrepareForBackup** to notify writers to prepare for a backup operation.
11. Call **IVssBackupComponents::GatherWriterStatus** and **IVssBackupComponents::GetWriterStatus** (or **IVssBackupComponentsEx3::GetWriterStatus**) to verify the status of the ASR writer.
12. At this point, you can query for failure messages that were set by the writer in its **CVssWriter::OnPrepareBackup** method. For example code that shows how to view these messages, see **IVssComponentEx::GetPrepareForBackupFailureMsg**.
13. Call **IVssBackupComponents::DoSnapshotSet** to create a volume shadow copy.
14. Call **IVssBackupComponents::GatherWriterStatus** and **IVssBackupComponents::GetWriterStatus** to verify the status of the ASR writer.
15. Back up the data.
16. Indicate whether the backup operation succeeded by calling **IVssBackupComponents::SetBackupSucceeded**.
17. Call **IVssBackupComponents::BackupComplete** to indicate that the backup operation has completed.
18. Call **IVssBackupComponents::GatherWriterStatus** and **IVssBackupComponents::GetWriterStatus**. The writer session state memory is a limited resource, and writers must eventually reuse session states. This step marks the writer's backup session state as completed and notifies VSS that this backup session slot can be reused by a subsequent backup operation.

    > **NOTE**
    > This is only necessary on Windows Server 2008 with Service Pack 2 (SP2) and earlier.

19. Call **IVssBackupComponents::SaveAsXML** to save a copy of the requester's Backup Components Document. The information in the Backup Components Document is used at restore time when the requester calls the **IVssBackupComponents::InitializeForRestore** method.

Choosing Which Critical Components to Back Up

In the backup initialization phase, the ASR writer reports the following types of components in its Writer Metadata Document:

- Critical volumes, such as the boot, system, and Windows Recovery Environment (Windows RE) volumes and the Windows RE partition that is associated with the instance of Windows Vista or Windows Server 2008 that is currently running. A volume is a *critical volume* if it contains system state information. The boot and system volumes are included automatically. The requester must include all volumes that contain system-critical components reported by writers, such as the volumes that contain the Active Directory. System-critical components are marked as "not selectable for backup." In VSS, "not selectable" means "not optional." Thus, the requester is required to back them up as part of system state. For more information, see Backing Up and Restoring System State. Components for which the VSS_CF_NOT_SYSTEM_STATE flag is set are not system-critical.

> **NOTE**
> The ASR component is a system-critical component that is reported by the ASR writer.

- Disks. Every fixed disk on the computer is exposed as a component in ASR. If a disk was not excluded during backup, it will be assigned during restore and can be re-created and reformatted. Note that during restore, the requester can still re-create a disk that was excluded during backup by calling the IVssBackupComponents::SetRestoreOptions method. If one disk in a dynamic disk pack is selected, all other disks in that pack must also be selected. If a volume is selected because it is a critical volume (that is, a volume that contains system state information), every disk that contains an extent for that volume must also be selected. To find the extents for a volume, use the IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS control code.

  > **NOTE**
  > During backup, the requester should include all fixed disks. If the disk that contains the requester's backup set is a local disk, this disk should be included. During restore, the requester must exclude the disk that contains the requester's backup set to prevent it from being overwritten.

  In a clustering environment, ASR does not re-create the layout of the cluster's shared disks. Those disks should be restored online after the operating system is restored in the Windows RE.

- Boot Configuration Data (BCD) store. This component specifies the path of the directory that contains the BCD store. The requester must specify this component and back up all of the files in the BCD store directory. For more information about the BCD store, see About BCD.

  > **NOTE**
  > On computers that use the Extended Firmware Interface (EFI), the EFI System Partition (ESP) is always hidden and cannot be included in a volume shadow copy. The requester must back up the contents of this partition. Because this partition cannot be included in a volume shadow copy, the backup can only be performed from the live volume, not from the shadow copy. For more information about EFI and ESP, see Bring up guide.

The component names use the following formats:

- For disk components, the format is

  where *n* is the disk number. Only the disk number is recorded. To get the disk number, use the IOCTL_STORAGE_GET_DEVICE_NUMBER control code.

- For volume components, the format is

  where *GUID* is the volume GUID.

- For the BCD store component, the format is

  If the system partition has a volume GUID name, this component is selectable. Otherwise, it is not selectable.

NOTE

ASR adds the files to the BCD store component's file group as follows:

- For EFI disks, ASR adds

  SystemPartitionPath\EFI\Microsoft\Boot*.*

  where *SystemPartitionPath* is the path to the system partition.

- For GPT disks, ASR adds

  SystemPartitionPath\Boot*.*

  where *SystemPartitionPath* is the path to the system partition.

- The system partition path can be found under the following registry key:
  HKEY_LOCAL_MACHINE\System\Setup\SystemPartition

On restore, all components that are marked as critical volumes must be restored. If one or more critical volumes cannot be restored, the restore operation fails.

In the **PreRestore** phase of the restore sequence, disks that were not excluded during backup are re-created and reformatted by default. However, they are not re-created or reformatted if they meet the following conditions:

- A basic disk is not re-created if its disk layout is intact or only additive changes have been made to it. The disk layout is intact if the following conditions are true:
  - The disk signature, disk style (GPT or MBR), logical sector size, and volume start offset are not changed.
  - The volume size is not decreased.
  - For GPT disks, the partition identifier is not changed.
- A dynamic disk is not re-created if its disk layout is intact or only additive changes have been made to it. For a dynamic disk to be intact, all of the conditions for a basic disk must be met. In addition, the entire disk pack's volume structure must be intact. The disk pack's volume structure is intact if it meets the following conditions, which apply to both MBR and GPT disks:
  - The number of volumes that are available in the physical pack during restore must be greater than or equal to the number of volumes that were specified in the ASR writer metadata during backup.
  - The number of *plexes* per volume must be unchanged.
  - The number of *members* must be unchanged.
  - The number of physical disk extents must be greater than the number of disk extents specified in the ASR writer metadata.
  - An intact pack remains intact when additional volumes are added, or if a volume in the pack is extended (for example, from a simple volume to a spanned volume).

    NOTE

    If a simple volume is mirrored, the pack is not intact and will be re-created to ensure that the BCD and boot volume state remain consistent after restore. If volumes are deleted, the pack is re-created.

- If the dynamic disk pack's volume structure is intact and only additive changes have been made to it, the disks in the pack are not re-created.

  Windows Vista: Dynamic disks are always re-created. Note that this behavior has changed with Windows Server 2008 and Windows Vista with Service Pack 1 (SP1).

At any time before the beginning of the restore phase, the requester can specify that the disks should be quick-formatted by setting the **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ASR\RestoreSession** registry key. Under this key, there is a value named **QuickFormat** with the data type REG_DWORD. If this value does not exist, you should create it. Set the data of the **QuickFormat** value to 1 for quick formatting or 0 for slow formatting.

If the **QuickFormat** value does not exist, the disks will be slow-formatted.

Quick formatting is significantly faster than slow formatting (also called full formatting). However, quick formatting does not verify each sector on the volume.

Overview of Restore Phase Tasks

At restore time, the requester performs the following steps:

> **NOTE**
> All steps are required unless otherwise indicated.

1. Call the **CreateVssBackupComponents** function to create an instance of the **IVssBackupComponents** interface and call the **IVssBackupComponents::InitializeForRestore** method to initialize the instance for restore by loading the requester's Backup Components Document into the instance.

2. [This step is required only if the requester needs to change whether "IncludeDisk" or "ExcludeDisk" is specified for one or more disks.] Call **IVssBackupComponents::SetRestoreOptions** to set the restore options for the ASR writer components. The ASR writer supports the following options: "IncludeDisk" allows the requester to include a disk on the target system to be considered for restore, even if it was not selected during the backup phase. "ExcludeDisk" allows the requester to prevent a disk on the target system from being re-created. Note that if "ExcludeDisk" is specified for a disk that contains a critical volume, the subsequent call to **IVssBackupComponents::PreRestore** will fail.

   The following example shows how to use **SetRestoreOptions** to prevent disk 0 and disk 1 from being re-created and inject third-party drivers into the restored boot volume.

   Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: Injection of third-party drivers is not supported.

   The example assumes that the **IVssBackupComponents** pointer, m_pBackupComponents, is valid.

```
m_pBackupComponents->SetRestoreOptions(
    AsrWriterId,
    VSS_CT_FILEGROUP,
    NULL,
    TEXT("ASR"),
    TEXT("\"ExcludeDisk\"=\"0\", \"ExcludeDisk\"=\"1\" "),
    TEXT("\"InjectDrivers\"=\"1\" ")
    );
```

To exclude all disks for a specified volume, see the following "Excluding All Disks for a Volume."

3. Call **IVssBackupComponents::PreRestore** to notify the ASR writer to prepare for a restore operation. Call **IVssAsync::QueryStatus** as many times as necessary until the status value returned in the *pHrResult* parameter is not VSS_S_ASYNC_PENDING.

4. Restore the data. In the restore phase, ASR reconfigures the volume GUID path (\\?\Volume{GUID}) for each volume to match the volume GUID path that was used during the backup phase. However, drive letters are not preserved, because this would cause collisions with the drive letters that are automatically assigned in the recovery environment. Thus, when restoring data, the requester must use volume GUID paths, not drive letters, to access the volumes.

5. Set the **HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ASR\RestoreSession** registry key to indicate the set of volumes that have been restored or reformatted.

   Under this key, there is a value named **RestoredVolumes** with the data type REG_MULTI_SZ. If this value does not exist, you should create it. Under this value, your requester should create a volume GUID entry for each volume that has been restored. This entry should be in the following format: \\?\Volume{78618c8f-aefd-11da-a898-806e6f6e6963}. Each time a bare-metal recovery is performed, ASR sets the **RestoredVolumes** value to the set of volumes that ASR restored. If the requester restored additional volumes, it should set this value to the union of the set of volumes that the requester restored and the set of volumes that ASR restored. If the requester did not use ASR, it should replace the list of volumes.

   You should also create a value named **LastInstance** with the data type REG_SZ. This key should contain a random cookie that uniquely identifies the current restore operation. Such a cookie can be created by using the **UuidCreate** and **UuidToString** functions. Each time a bare-metal recovery is performed, ASR resets this registry value to notify requesters and non-VSS backup applications that the recovery has occurred.

6. Call **IVssBackupComponents::PostRestore** to indicate the end of the restore operation. Call **IVssAsync::QueryStatus** as many times as necessary until the status value returned in the *pHrResult* parameter is not VSS_S_ASYNC_PENDING.

In the restore phase, ASR may create or remove partitions to restore the computer to its previous state. Requesters must not attempt to map disk numbers from the backup phase to the restore phase.

On restore, the requester must exclude the disk that contains the requester's backup set. Otherwise, the backup set can be overwritten by the restore operation.

On restore, a disk is excluded if it was not selected as a component during backup, or if it is explicitly excluded by calling **IVssBackupComponents::SetRestoreOptions** with the "ExcludeDisk" option during restore.

It is important to note that during WinPE disaster recovery, ASR writer functionality is present, but no other writers are available, and the VSS service is not running. After WinPE disaster recovery has completed, the computer has restarted, and the Windows operating system is running normally, the VSS service can be started, and the requester can perform any additional restore operations that require participation of writers other than the ASR writer.

If during the restore session the backup application detects that the volume unique IDs are unchanged, and therefore all volumes from the time of the backup are present and intact in WinPE, the backup application can proceed to restore only the contents of the volumes, without involving ASR. In this case, the backup application should indicate that the computer was restored by setting the following registry key in the restored operating system: **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ASR\RestoreSession**

Under this key, specify **LastInstance** for the value name, REG_SZ for the value type, and a random cookie (such

as a GUID created by the **UuidCreate** function) for the value data.

If during the restore session the backup application detects that one or more volumes are changed or missing, the backup application should use ASR to perform the restore. ASR will re-create the volumes exactly the way they were at the time of the backup and set the **RestoreSession** registry key.

Excluding All Disks for a Volume

The following example shows how to exclude all disks for a specified volume.

```
HRESULT BuildRestoreOptionString
(
    const WCHAR             *pwszVolumeNamePath,
    CMyString               *pstrExclusionList
)
{
    HANDLE                  hVolume           = INVALID_HANDLE_VALUE;
    DWORD                   cbSize            = 0;
    VOLUME_DISK_EXTENTS     * pExtents        = NULL;
    DISK_EXTENT             * pExtent         = NULL;
    ULONG                   i                 = 0;
    BOOL                    fIoRet            = FALSE;
    WCHAR                   wszDest[MAX_PATH] = L"";
    CMyString               strVolumeName;
    CMyString               strRestoreOption;

    // Open a handle to the volume device.
    strVolumeName.Set( pwszVolumeNamePath );
    // If the volume name contains a trailing backslash, remove it.
    strVolumeName.UnTrailing( L'\\' );
    hVolume = ::CreateFile(strVolumeName, 0, FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, NULL,
0);
    // Check whether the call to CreateFile succeeded.

    // Get the list of disks used by this volume.
    cbSize = sizeof(VOLUME_DISK_EXTENTS);
    pExtents = (VOLUME_DISK_EXTENTS *)::CoTaskMemAlloc(cbSize);

    ::ZeroMemory(pExtents, cbSize);

    fIoRet = ::DeviceIoControl(hVolume, IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS, NULL, 0, pExtents, cbSize,
&cbSize, 0);
    if ( !fIoRet && GetLastError() == ERROR_MORE_DATA )
    {
        // Allocate more memory.
        cbSize = FIELD_OFFSET(VOLUME_DISK_EXTENTS, Extents) + pExtents->NumberOfDiskExtents *
sizeof(DISK_EXTENT);
        ::CoTaskMemFree(pExtents);
        pExtents = NULL;

        pExtents = (VOLUME_DISK_EXTENTS *) ::CoTaskMemAlloc(cbSize);
        // Check whether CoTaskMemAlloc returned an out-of-memory error.
        ::ZeroMemory(pExtents, cbSize);

        // Now the buffer should be big enough.
        ::DeviceIoControl(hVolume, IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS, NULL, 0, pExtents, cbSize, &cbSize,
0);
        // Check whether the IOCTL succeeded.
    }
    // Check for errors; note that the IOCTL can fail for a reason other than insufficient memory.

    // For each disk, mark it to be excluded in the Restore Option string.
    for (i = 0; i < pExtents->NumberOfDiskExtents; i++)
    {
        pExtent = &pExtents->Extents[i];
```

```
        *wszDest = L'\0';
        StringCchPrintf(wszDest, MAX_PATH, L"\"ExcludeDisk\"=\"%d\", ", pExtent->DiskNumber); // check
errors

        strRestoreOption.Append(wszDest);
        // Check for an out-of-memory error.
    }

    // Remove the trailing comma.
    strRestoreOption.TrimRight();
    strRestoreOption.UnTrailing(',');

    // Set the output parameter.
    strRestoreOption.Transfer( pstrExclusionList );

Exit:
    if( pExtents )
    {
        ::CoTaskMemFree(pExtents);
        pExtents = NULL;
    }

    if( hVolume != INVALID_HANDLE_VALUE )
    {
        ::CloseHandle(hVolume);
        hVolume = INVALID_HANDLE_VALUE;
    }

    return ( hr );
}
```

Using Tracing Tools with ASR Applications

3/5/2021 • 2 minutes to read • Edit Online

You can use the Logman tool to collect tracing information for VSS applications that use Automated System Recovery (ASR). Logman (logman.exe) is a trace controller for trace events and performance counters. It is included in Windows XP and later versions of Windows. Or, if you prefer, you can use the Tracelog tool to collect the same ASR tracing information. Tracelog is included in the Windows Driver Kit (WDK).

To use tracing tools with VSS, see Using Tracing Tools with VSS.

Using Logman

The following procedure describes how to use Logman with your ASR application.

To use Logman with your ASR application

1. Use the following command to start tracing:

   ```
   logman start asr -o *x:\*asr.etl -ets -p {6407345b-94f2-44c8-b3db-4e076be46816} 0xff 0xff
   ```

   > **NOTE**
   > Replace "x:\" with the path to the directory where you would like the trace log file to be stored. For best performance, the trace log file should be located on a volume that is not part of the shadow copy.

2. Use the following command to stop tracing:

   ```
   logman stop asr -ets
   ```

The trace log file is *x:*asr.etl.

For more information about the Logman tool, see Logman.

Using Tracelog

The following procedure describes how to use Tracelog.

To use Tracelog

1. Create a text file named asr.ctl that contains only the following text:

   ```
   6407345b-94f2-44c8-b3db-4e076be46816 asr
   ```

2. Use the following command to start tracing:

   ```
   tracelog -start asr -f *x:\*asr.etl -guid asr.ctl -flag 0xff -level 0xff
   ```

   > **NOTE**
   > Replace "x:\" with the path to the directory where you would like the trace log file to be stored. For best performance, the trace log file should be located on a volume that is not part of the shadow copy.

3. Use the following command to stop tracing:

   ```
   tracelog -stop asr
   ```

The trace log file is *x:*asr.etl.

For more information about the Tracelog tool, see Tracelog.

Custom Backups and Restores

3/5/2021 • 2 minutes to read • Edit Online

The following topics provide information about custom backups and restores:

- VSS Backup and Restore of the Active Directory
- Backing Up and Restoring an FRS-Replicated SYSVOL Folder
- VSS Backups and Restores of the Cluster Database
- Registry Backup and Restore Operations Under VSS
- Backing Up and Restoring the COM+ Class Registration Database Under VSS

VSS Backup and Restore of the Active Directory

3/5/2021 • 2 minutes to read • Edit Online

The Active Directory writer requires no special actions during backup operations. The writer will provide the requester with component and file set information, and the requester uses that information to decide which files to copy to backup media. There is no need to use any special APIs to back up the Active Directory.

How a restore is performed depends on whether the Active Directory is be restored as part of a disaster recovery operation, or if the restore is to a system on which the Active Directory is running. In addition, the age of the backup copy of the Active Directory state may be an issue because of Active Directory tombstones.

Active Directory Restoration following Disaster Recovery

Following a crash requiring disaster recovery, the Active Directory can be restored as part of the restoration of the operating system state.

This restore operation is essentially a writerless restore.

Active Directory Restoration on the System where It Is Running

The system must be rebooted in Directory Services Restore mode if the Active Directory is currently running on the server.

The operating system will then be running without the Active Directory, and all user validation occurs through the Security Accounts Manager (SAM) in the registry. Only the administrator has permission to recover the Active Directory.

Once in Directory Service Restore mode, a VSS restore can proceed normally. There is no reason to use non-VSS Win32 Active Directory APIs to restore the Active Directory state.

Active Directory Restores and Active Directory Tombstones

Any recovery plan should ensure that the age of the backup should not exceed the Active Directory Tombstone Lifetime (default is 60 days).

Restoration of a backup older than the tombstone will cause a domain controller to have objects that are not replicated to the other servers.

Those objects that are not replicated will not be deleted automatically on that (restored) domain controller because the tombstones of those objects on the other replicas have already been deleted.

An administrator will have to manually delete each of the objects on the restored domain controller that are not replicated. Incremental backups of the Active Directory are not supported; a full backup is required.

Backing Up and Restoring an FRS-Replicated SYSVOL Folder

3/5/2021 • 11 minutes to read • Edit Online

The System Volume (SYSVOL) folder provides a standard location to store important elements of Group Policy objects and scripts. A copy of the SYSVOL folder exists on each domain controller in a domain. The SYSVOL folder is replicated by either Distributed File System Replication (DFSR) or the File Replication Service (FRS). This topic explains how to determine whether a SYSVOL folder is replicated by DFSR or FSR and explains how to backup and restore an FRS-replicated SYSVOL folder.

FRS can copy SYSVOL contents to other domain controllers within the domain. FRS monitors the SYSVOL folder and, if a change occurs to any file stored on the SYSVOL folder, then FRS automatically replicates the changed file to the SYSVOL folders on the other domain controllers in the domain.

> **NOTE**
> Only the Group Policy template is replicated by replicating the contents of the SYSVOL folder. The Group Policy container is replicated through Active Directory replication. For Group Policy to operate successfully, both the Group Policy template and the Group Policy container must be available on a domain controller.

This topic covers the following subjects:

- Determining Whether a Domain Controller's SYSVOL Folder is Replicated by DFSR or FRS
- Backing Up a DFSR-Replicated SYSVOL Folder
- Backing Up an FRS-Replicated SYSVOL Folder on a Windows Server 2008 or Windows Server 2003 Domain
- Sample FRS Writer Metadata Document
- Setting Registry Keys for a Restore of an FRS-Replicated SYSVOL Folder
- Performing a Nonauthoritative Restore of an FRS-Replicated SYSVOL Folder
- Performing an Authoritative Restore of an FRS-Replicated SYSVOL Folder

Determining Whether a Domain Controller's SYSVOL Folder is Replicated by DFSR or FRS

The following table summarizes how to determine whether a domain controller's SYSVOL folder is being replicated by DFSR or FRS.

IF THE DOMAIN CONTROLLER IS RUNNING	SYSVOL IS REPLICATED BY
Windows Server 2008 + domain functional level of Windows Server 2008 + SYSVOL migration completed	DFSR
Windows Server 2008 + domain functional level below Windows Server 2008	FRS
Windows Server 2003	FRS

If the domain's functional level is Windows Server 2008 and the domain has undergone SYSVOL migration, DFSR will be used to replicate the SYSVOL folder. If the first domain controller in the domain was promoted directly into the Windows Server 2008 functional level, DFSR is automatically used for SYSVOL replication. In such cases, there is no need for migration of SYSVOL replication from FRS to DFSR. If the domain was upgraded to Windows Server 2008 functional level, FRS is used for SYSVOL replication until the migration process from FRS to DFSR is complete.

To determine whether DFSR or FRS is being used on a domain controller that is running Windows Server 2008, check the value of the **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\DFSR\Parameters\SysVols\Migrating Sysvols\LocalState** registry subkey. If this registry subkey exists and its value is set to 3 (ELIMINATED), DFSR is being used. If the subkey does not exist, or if it has a different value, FRS is being used.

Backing Up a DFSR-Replicated SYSVOL Folder

If the SYSVOL folder is replicated by DFSR, the DFSR VSS writer can be used to back it up. For more information about the DFSR VSS writer, see DFSR Replicated Folders.

Backing Up an FRS-Replicated SYSVOL Folder on a Windows Server 2008 or Windows Server 2003 Domain

On a domain controller that is running Windows Server 2008 or Windows Server 2003, the VSS infrastructure is present, and therefore the FRS VSS writer can be used to back up the SYSVOL folder and FRS components.

The FRS VSS writer's Writer Metadata Document provides information about the location of the SYSVOL folder and the exclusion lists for the writer. Based on this information, a VSS backup application (requester) can back up the SYSVOL folder using the regular VSS-based backup techniques.

The Writer Metadata Document contains information about the writer, the data that the writer owns, and how to restore that data. This is a read-only document that can be retrieved by the backup application before taking a backup. The DiskShadow tool can be used to view the FRS VSS writer's Writer Metadata Document. The DiskShadow list writers command provides information about the writers present on the system. This list contains information about the FRS writer on domain controllers that use FRS for SYSVOL replication or on file servers that use FRS for replication of DFS link targets.

The following Sample FRS Writer Metadata Document section shows a sample FRS Writer Metadata Document for a domain controller that has the SYSVOL folder on D:\Windows\Sysvol. The path shown in the "Excluded files" section will be the same as that obtained when querying the Netlogon service's **SysVol** registry key:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\NetLogon\Parameters\SysVol

The only exception to this rule occurs when the domain controller is in the REDIRECTED state of SYSVOL migration. In this state, the writers corresponding to both FRS and the DFSR service report their respective copies of the SYSVOL folder. However, the DFSR service's copy of the SYSVOL folder (usually a folder called SYSVOL_DFSR) is the one that is shared by the domain controller; this path is the one referenced by the **SysVol** registry key.

The FRS VSS writer requires a custom restore method. This means that certain custom steps must be performed when restoring files that are being replicated by FRS. For more information, see Performing a Nonauthoritative Restore of an FRS-Replicated SYSVOL Folder.

> **NOTE**
>
> System state backups for Windows domain controllers do not include the FRS database that maintains state information for the FRS service pertaining to the files within the SYSVOL folder and other content sets. The FRS database, debug logs, staging area files, and files in the pre-existing data folder are excluded from a system state backup. The following sample FRS writer specification contains the exclusion list in the "Excluded files" section.

Sample FRS Writer Metadata Document

The following is a sample FRS Writer Metadata Document for a domain controller whose SYSVOL folder path is D:\Windows\Sysvol.

```
* WRITER "FRS Writer"
    - Writer ID   = {d76f5a28-3092-4589-ba48-2958fb88ce29}
    - Writer Instance ID = {07ae58e5-6977-4e34-9dfe-399bbd2cbe40}
    - Supports restore events = FALSE
    - Writer restore conditions = VSS_WRE_NEVER
    - Restore method = VSS_RME_CUSTOM
    - Requires reboot after restore = FALSE

    - Excluded files:
       - Exclude: Path = d:\windows\ntfrs\jet, Filespec = *
       - Exclude: Path = d:\Windows\debug\NtFrs, Filespec = NtFrs*
       - Exclude: Path = d:\windows\sysvol\domain\DO_NOT_REMOVE_NtFrs_PreInstall_Directory, Filespec = *
       - Exclude: Path = d:\windows\sysvol\domain\NtFrs_PreExisting___See_EventLog, Filespec = *
       - Exclude: Path = d:\windows\sysvol\staging\domain, Filespec = NTFRS_*

    - Component "FRS Writer:\SYSVOL\da45368c-b2d3-4cf0-bdc338a2cde15a7b"
       - Name: 'da45368c-b2d3-4cf0-bdc338a2cde15a7b'
       - Logical Path: 'SYSVOL'
       - Full Path: '\SYSVOL\da45368c-b2d3-4cf0-bdc338a2cde15a7b'
       - Caption: ''
       - Type: VSS_CT_FILEGROUP [2]
       - Is Selectable: 'TRUE'
       - Is top level: 'TRUE'
       - Notify on backup complete: 'TRUE'
       - Components:
       - File List: Path = d:\windows\sysvol, Filespec = *
       - Affected paths by this component:
         - d:\windows\sysvol
       - Affected volumes by this component:
         - \\?\Volume{da897ba5-5840-11db-93c1-806e6f6e6963}\ [D:\]
       - Component Dependencies:
```

Setting Registry Keys for a Restore of an FRS-Replicated SYSVOL Folder

The **BurFlags** registry key is used to perform authoritative or nonauthoritative restores on FRS members of DFS or SYSVOL replica sets. The global (computer-wide) **BurFlags** registry key contains REG_DWORD values and is located in the following location in the registry:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\NtFrs\Parameters\Backup/Restore\Process at Startup

The most common values for the **BurFlags** registry key are:

- D2, also known as a nonauthoritative mode restore.

- D4, also known as an authoritative mode restore.

There are global and replica set-specific **BurFlags** registry keys. Setting the global **BurFlags** registry key reinitializes all replica sets that the member holds. This global key should be set when the server holds only one replica set, or when the replica sets that it holds are relatively few in number and small in size. For example, if the server is a domain controller that does not host any content sets that are replicated using FRS other than the SYSVOL folder, the global **BurFlags** registry key can be set.

The global **BurFlags** registry key is found in the following location in the registry:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\NtFrs\Parameters\Backup/Restore\Process At Startup

In contrast to the global **BurFlags** key, the replica set-specific **BurFlags** key permits the re-initialization of discrete, individual replica sets, allowing healthy replication sets to be left intact.

Replica set-specific **BurFlags** registry keys can be located by determining the GUID for that particular replica set.

The following procedure describes how to determine which GUID corresponds to a particular replica set and describes how to configure a restore.

To determine which GUID corresponds to a particular replica set and to configure a restore

1. Stop the FRS service.
2. To determine the GUID that represents a particular replica set:
   a. Locate the following key in the registry.

      HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\NtFrs\Parameters\Replica Sets

   b. Below the **Replica Sets** subkey, there are one or more subkeys that are each identified by a GUID.
   c. The **Replica Set Root** value for each GUID is a file system path that indicates the replica set that is represented by this GUID.
   d. Iterate over this list of GUIDs until the desired replica set is located. Note the corresponding GUID.
3. Locate the following subkey in the registry:

   HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\NtFrs\Parameters\Cumulative Replica Sets
4. Below this subkey, locate the same GUID that was noted in step 2. Below the GUID subkey, create an entry for the **BurFlags** key.
5. Restart the FRS service.

Performing a Nonauthoritative Restore of an FRS-Replicated SYSVOL Folder

The nonauthoritative restore is the most common way to reinitialize SYSVOL replication on individual domain controllers. Domain controllers that are nonauthoritatively restored must have inbound connections from other working domain controllers, which are participating in Active Directory and FRS replication. In a large deployment environment consisting of many domain controllers, the remaining domain controllers can be recovered using a nonauthoritative mode restore under the following conditions:

- There must be at least one known good replica member (a domain controller with a healthy SYSVOL folder).

- The other domain controllers must be reinitialized in direct replication partner order.

The following procedure describes how to perform a nonauthoritative restore.

To perform a nonauthoritative restore

1. Stop the FRS service.
2. Restore the backed-up data to the SYSVOL folder.
3. Configure the **BurFlags** registry key by setting the value of the following registry key to the DWORD value D2.

   HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\NtFrs\Parameters\Backup/Restore\Process at Startup\BurFlags
4. Restart the FRS service.

When the FRS service is restarted, the following actions occur:

- The value of the **BurFlags** registry key is reset to zero.
- Files in the reinitialized FRS folders are moved to a pre-existing folder.
- Event 13565 is logged in the FRS event log to signal that a nonauthoritative restore has started.

  > **NOTE**
  > FRS event codes are documented in "FRS event log error codes" in the Help and Support Knowledge Base at https://go.microsoft.com/fwlink/p/?linkid=117779

- The FRS database is rebuilt.
- The member performs an initial join of the replica set from an upstream partner or from the computer that is specified in the **Replica Set Parent** registry key if a parent has been specified for SYSVOL replica sets.
- The reinitialized computer runs a full replication of the affected replica sets when the relevant replication schedule begins.
- When the process is complete, an event 13516 is logged to signal that FRS is operational. If the event is not logged, there is a problem with the FRS configuration.

> **NOTE**
> The placement of files in the pre-existing folder on reinitialized members is a safeguard in FRS that is designed to prevent accidental data loss. Any files destined for the replica that exist only in the local pre-existing folder and were replicated after the initial replication may then be copied to the appropriate folder. When outbound replication has occurred, files in the pre-existing folder can be deleted to free additional drive space.

Performing an Authoritative Restore of an FRS-Replicated SYSVOL Folder

Authoritative restores are used as a last resort in case of critical situations such as divergence of data on the

content set caused by directory collisions. For example, an authoritative restore might be needed to restore an FRS replica set where replication has completely stopped and a rebuild from scratch is required.

If you must perform an authoritative restore of the SYSVOL folder, be aware that it is a very complicated process. Comprehensive guidelines detailing the operations that need to be performed for an authoritative restore of the contents of the SYSVOL folder are documented in "How to rebuild the SYSVOL tree and its content in a domain" in the Help and Support Knowledge Base at https://go.microsoft.com/fwlink/p/?linkid=117780.

The following requirements must be met before an authoritative FRS restore is performed:

1. The FRS service must be disabled on all downstream replication partners (direct and transitive) for the reinitialized SYSVOL folder before the authoritative restore has been configured to occur.
2. Events 13553 and 13516 have been logged in the FRS event log. These events indicate that the membership of the SYSVOL replica set has been established on the domain controller that is configured for the authoritative restore.

   > **NOTE**
   > FRS event codes are documented in "FRS event log error codes" in the Help and Support Knowledge Base at https://go.microsoft.com/fwlink/p/?linkid=117779

3. The domain controller that is configured for the authoritative restore is configured to be authoritative for all the SYSVOL data that is to be replicated to the remaining domain controllers.
4. All other partners in the replica set must be reinitialized with a nonauthoritative restore.

VSS Backups and Restores of the Cluster Database

3/5/2021 • 2 minutes to read • Edit Online

Failover clustering uses the Volume Shadow Copy Service (VSS) for backing up and restoring the failover cluster configuration. For more information, see Backing Up and Restoring the Failover Cluster Configuration Using VSS.

Registry Backup and Restore Operations Under VSS

3/5/2021 • 7 minutes to read • Edit Online

The Windows Registry Service supports a VSS writer, called the registry writer, which allows requesters to back up a system registry using data stored on a shadow copied volume. For more information about the registry writer, see In-Box VSS Writers.

The registry writer performs in-place backups and restores of the registry. In addition, the registry writer reports only system hives; it does not report user hives.

Windows Server 2003: The registry writer uses an intermediate repository file (also known as a spit file) to store registry data. In addition, the registry writer reports system hives and user hives.

The writer ID for the registry writer is AFBAB4A2-367D-4D15-A586-71DBB18F8485.

Windows XP: There is no registry writer. The registry data is reported by the Bootable State writer, whose writer ID is F2436E37-09F5-41AF-9B2A-4CA2435DBFD5.

> **NOTE**
> Microsoft does not provide developer or IT professional technical support for implementing online system state restores on Windows (all releases). For information about using Microsoft-provided APIs and procedures to implement online system state restores, see the community resources available at the MSDN Community Center.

> **NOTE**
> The following information only applies to Windows Server 2003 and Windows XP.

Registry Backup Using VSS

The registry writer will export and save active registry files in the locations defined by the key **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\hivelist**.

The names of the values under this registry entry identify the registry hive to be saved, and the value's data provides the file containing the file (the hive file). The hive files are specified with the following format: \Device*HarddiskVolumeX**path**filename*.

For example, under **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\hivelist**, you might see **\REGISTRY\MACHINE\SOFTWARE** = \Device\HarddiskVolume1\Windows\System32\config\SOFTWARE.

The registry writer ensures that hive files are saved to disk prior to its shadow copy.

When backing up the registry hives, a requester would replace \Device*HarddiskVolumeX* with the *device object* string of the volume's shadow copy.

> **NOTE**
>
> You can convert the \Device*HarddiskVolumeX* path to an equivalent Win32 path by using the **QueryDosDevice** function. For more information, see Obtaining a File Name From a File Handle or Displaying Volume Path Names.

Registry Restore Using Non-VSS Win32 APIs

For an online (safe mode or full operating system) restore, the subkeys in the **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations** registry key must be preserved.

The **MoveFileEx** and **MoveFileTransacted** functions use this registry key to store information about files that were renamed by using the MOVEFILE_DELAY_UNTIL_REBOOT value in the *dwFlags* parameter.

To preserve the contents of the **PendingFileRenameOperations** registry key, your backup application should call the **RegLoadKey** function to connect the registry file to be restored to the active registry. Your backup application can then use the various registry functions to copy the desired keys and values into the loaded hive. After the copy is complete, the **RegFlushKey** and **RegUnloadKey** functions should be called.

For an offline (Windows Recovery Environment or Windows PE) restore, it is not necessary to honor the **PendingFileRenameOperations** registry key.

Registry Restore Using Non-VSS Win32 APIs in Windows Server 2003

> **NOTE**
>
> The following information applies only to restore operations related to disaster recovery (also known as bare-metal recovery) that are performed in Windows Server 2003.

When restoring the registry, a backup application needs to move some of the subkeys from the current registry into the registry that is to be restored.

To do this, a backup application can call **RegLoadKey** to connect the registry file to be restored to the currently active registry. It can then use the various registry functions to copy the desired keys and values into the loaded hive. After the copy is complete, **RegFlushKey** and **RegUnloadKey** are called.

There is a registry key that indicates to a restore application (requester) the registry keys under **HKEY_LOCAL_MACHINE\SYSTEM** that should not be overwritten at restore time:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\BackupRestore\KeysNotToRestore

Part of the system state restore process involves restoring a previously backed-up registry.

Backup applications need to take special care when restoring the **HKEY_LOCAL_MACHINE\SYSTEM** hive because the process of installing a temporary version of the Windows operating system will have established keys in the newly installed system hive whose values must survive the restore operation.

For example, when the replacement system has a network interface card different from the original system, restoration of the original keys for the previous card will lead to unpredictable results. This is because the Plug and Play service has detected and placed proper service and driver registry entries into the registry. These values must be preserved to properly boot after system restore.

This section describes how backup applications can discover which keys and files are to be preserved when executing a restore of the **HKEY_LOCAL_MACHINE\SYSTEM** hive. In some cases, this will involve programmatically copying the keys from the newly installed installation hive into the hive to be restored. In other cases, ensuring that a product's registry keys are not replaced is as simple as specifying such keys in the product's INF configuration file.

Keys (and key data) that are to be preserved are enumerated in the **HKEY_LOCAL_MACHINE\SYSTEM** hive under the

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\BackupRestore\KeysNotToRestore

key as sets of REG_MULTI_SZ strings (called *key strings* in this document).

A backup application (requester) must examine the values of these keys in the active registry and the newly restored registry because any application or service can add values at any time.

How key strings are to be interpreted by backup applications is determined by their terminal character:

1. Key strings terminated with a backslash ('\') are interpreted as subkeys. When such a substring is encountered, the backup application must preserve all data and all subordinate keys.

   For example, the following specifies that all subordinate keys and data are to be preserved across a restore operation:

   HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\dmio\boot info

   To this end, this key and all subordinate keys and data must be copied from the existing registry (that is, the one created by the installation of Windows) into the newly restored registry. This is called a *key replace* operation. This operation replaces the corresponding key in the restored registry.

2. Key strings whose termination character is an asterisk ('*') specifies that all subkeys should be merged. For example, the key string:

   HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services*

   specifies that the services key in the existing registry (for example those created by the installation of Windows) must be merged into the restored registry. This is called a *key merge* operation, and if a subkey exists in both the existing hive and the restored hive, the key in the restored directory is preserved with the following exceptions:

   - If the subkey in the existing hive contains a value named "start", and the subkey in the restored does not.
   - If the subkey in both the existing and restored hive contain a value named "start", and its numeric value in the existing hive is less.

   The "start" value in the registry specifies when a service or driver will start and can have a numeric value from 0-4. The lower the value, the sooner in the boot process the service will start.

   If this key exists in both the existing and the restored directory, you must examine the value of the start key in each hive. If the value in the existing hive is lower than the value in the restored directory, the lower value must be preserved.

   Once again, this key is used to determine whether a service or driver is to be started at boot time, at system time, manually, automatically, or be disabled. The lower value represents an earlier start time. The earlier start time must be applied to the new registry to ensure that the service or drivers are started properly at the next boot.

3. Key strings whose termination character is neither a backslash nor an asterisk are interpreted as registry values to be preserved.

For example, the key string:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations

The mechanism by which keys can be preserved programmatically involves the Win32 registry API. For example, one algorithm is enumerated below:

a. Restore the backed-up system hive file to a file. For this example, let the name be System.reg.

b. Use **RegLoadKey** to load System.reg into **HKEY_LOCAL_MACHINE** under a temporary name. For example, one such name might be

   HKEY_LOCAL_MACHINE\TMP_SYSTEM

c. Enumerate the values in the **KeysNotToRestore** subkey from both of the registry copies and create a superset of the lists. Copy each such key from the existing

   HKEY_LOCAL_MACHINE\SYSTEM

   key into the

   HKEY_LOCAL_MACHINE\TMP_SYSTEM

   key according to the semantics described above.

d. When complete use the **RegFlushKey** & **RegUnloadKey** entry points to save the

   HKEY_LOCAL_MACHINE\TMP_SYSTEM

   key back to System.reg.

e. Finally, use **RegReplaceKey** to specify that System.reg is to replace the

   HKEY_LOCAL_MACHINE\SYSTEM

   hive file, SYSTEM.

Backing Up and Restoring the COM+ Class Registration Database Under VSS

3/5/2021 • 2 minutes to read • Edit Online

The Component Object Model (COM) is a binary standard for writing component software in a distributed computing environment.

The original Win32 implementation stored component identifiers (GUIDs) and other COM state in the registry under **HKEY_CLASSES_ROOT\CLSID**.

The current generation of the Component Object Model, COM+, offers a registry-independent database for storing component registration information: the COM+ Class Registration Database.

The COM+ Class Registration Database supports a VSS writer, which allows requesters to back up the registration database using data stored on a shadow-copied volume.

To restore the COM+ Registration Database, a requester must call the ICOMAdminCatalog::RestoreREGDB method.

For more information about the COM+ Registration Database writer, see In-Box VSS Writers.

VSS Tools and Samples

3/5/2021 • 2 minutes to read • Edit Online

The following topics provide information about VSS tools and sample applications:

- Using VSS Diagnostics
- VssSampleProvider Tool and Sample
- VShadow Tool and Sample
- BETest Tool
- VSS Test Writer Tool
- Using Tracing Tools with VSS

Using Tracing Tools with VSS

3/5/2021 • 6 minutes to read • Edit Online

To collect tracing information for the VSS infrastructure, you can use the VssTrace tool, the Logman tool, or the Tracelog tool. VssTrace is available in the Microsoft Windows Software Development Kit (SDK) and can be used to trace VSS applications on Windows 7 and later versions of the Windows operating system. Logman is a trace controller for trace events and performance counters; it can also be used to trace VSS applications on Windows 7 and later versions of the Windows operating system. Tracelog is included in the Windows Driver Kit (WDK).

To use tracing tools with Automated System Recovery (ASR), see Using Tracing Tools with ASR Applications.

> **NOTE**
> VssTrace, Logman, and Tracelog all require administrator privilege.

For information about each tool, see the following sections:

- Using VssTrace
  - VssTrace Command-Line Options
- Using Logman
- Using Tracelog

Using VssTrace

To run the VssTrace tool from the command line, use the following syntax:

vsstrace *command-line-options*

To display concise command-line help for the VssTrace tool, use the following syntax:

vsstrace -help

To display detailed command-line help for the VssTrace tool, use the following syntax:

vsstrace -help all

VssTrace Command-Line Options

The VssTrace tool uses the following command-line options:

-f *Flags*

Enable the modules whose flags are specified by the *Flags* bitmask. Each flag corresponds to a VSS module. If *Flags* is zero, no modules are enabled. Note that most modules are enabled by default. This option can be combined with the **+**Module option. For example, **vsstrace -f 0 +WRITER +COORD** disables tracing of all of the modules that are enabled by default and enables tracing of VSS writers and the VSS service. Alternatively, **vsstrace +f 0xffff -COORD** enables tracing of all modules except the VSS service.

> **NOTE**
> If you use the **-f** option together with the **+*Module* option, the **-f** must appear before the **+*Module* option.

The following table lists the module name and flag for each available module.

MODULE	FLAG	ENABLED BY DEFAULT	ITEMS TRACED
EXCEPT	0x00000001	Yes	C++ exception handling.
COORD	0x00000002	Yes	The VSS service, which is also called the VSS coordinator.
SWPRV	0x00000004	Yes	The VSS System Shadow Copy Provider service.
BUCOMP	0x00000008	Yes	The VSS requester and backup metadata processing.
WRITER	0x00000010	Yes	VSS writer operations and VSS hosted writer implementations, such as the Windows Registry writer.
VSSAPI	0x00000020	Yes	Miscellaneous functions of the VSS API exported by VSSAPI.DLL.
HWDIAG	0x00000040	Yes	VSS hardware provider infrastructure and operations.
ADMIN	0x00000080	Yes	VSS command line utilities such as VSSADMIN.EXE and DISKSHADOW.EXE.
VSSUI	0x00000100	Yes	The Shadow Copies for Shared Folders configuration user interface (UI). The UI is available only on Windows Server operating systems.
TEST	0x00000200	Yes	Not applicable. (This tracing module is reserved.)
IOCTL	0x00000400	Yes	Details of FSCTL and IOCTL operations that the VSS service has initiated by calling the **DeviceIoControl** function.

MODULE	FLAG	ENABLED BY DEFAULT	ITEMS TRACED
GEN	0x00000800	Yes	General VSS utility functions, such as allocators, string classes, and registry and volume operations.
WRXML	0x00001000	No	XML processing for writer metadata. This module has a very high level of noise.
VSSXML	0x00002000	No	XML processing base classes. This module has a very high level of noise.

+_Module_

Enable the module specified by *Module*. More than one module can be enabled at a time. To list the available modules, type **vsstrace –help modules** at the command-line prompt.

-_Module_

Disable the module specified by *Module*. To list the available modules, type **vsstrace –help modules** at the command-line prompt.

+pid *ProcessId*

Enable the process specified by *ProcessId*. To enable all processes, use "*" for the value of *ProcessId*. More than one **pid** option can be specified at a time. The order of the options determines which processes are enabled or disabled. For example, to enable only the process whose process identifier is 0xe8c, use **vsstrace -pid * +pid 0xe8c**.

-pid *ProcessId*

Disable the process specified by *ProcessId*. To disable all processes, use "*" for the value of *ProcessId*. More than one **pid** option can be specified at a time. The order of the options determines which processes are enabled or disabled. For example, to disable all processes except the process whose process identifier is 0xe8c, use **vsstrace -pid * +pid 0xe8c**.

+tid *ThreadId*

Enable the thread specified by *ThreadId*. To enable all threads, use "*" for the value of *ThreadId*. More than one **tid** option can be specified at a time. The order of the options determines which threads are enabled or disabled. For example, to enable only the thread whose process identifier is 0x31a, use **vsstrace -tid * +tid 0x31a**.

-tid *ThreadId*

Disable the thread specified by *ThreadId*. To disable all threads, use "*" for the value of *ThreadId*. More than one **tid** option can be specified at a time. The order of the options determines which threads are enabled or disabled. For example, to disable all threads except the thread whose process identifier is 0x31a, use **vsstrace -tid * +tid 0x31a**.

-l *Level*

Use the tracing level specified by *Level*. The higher the level, the more verbose the trace output. Each level includes all of the lower levels. The default level is 170. The following levels are available.

LEVEL	INFORMATION INCLUDED IN THE TRACE OUTPUT
000	None
020	Fatal errors
030	Unhandled exceptions
040	Errors
050	Assertions
060	Warnings
080	Exception handling
100	Event Log activity
120	General information
140	Code flow
160	Function enter and exit
170	Function return values
180	Function parameters (terse)
190	Function parameters (verbose)
200	Verbose information level 1
210	Verbose information level 2
220	Verbose information level 3
230	Fast Code Level 1
240	Fast Code Level 2
250	Fast Code Level 3
255	All

+indent

Indent the formatted trace output at each function and subfunction boundary.

-indent

Do not indent the formatted trace output.

-etl *EtlFile*

Convert the Logman output file specified by *EtlFile* into a readable text format.

-o *OutputFile*

Save the tracing information to the output file specified by *OutputFile*. For best performance, the output file should be located on a volume that is not part of the shadow copy.

-help *HelpOption*

Display the command-line help as specified by *HelpOption*. The valid *HelpOption* values are **modules**, **levels**, and **all**. Specifying **modules** causes the modules to be listed. Specifying **levels** causes the available levels to be listed. Specifying **all** causes detailed help to be displayed. If no options are used, concise help is displayed.

Using Logman

The following procedure describes how to use Logman with your VSS application.

To use Logman with your VSS application

1. Use the following command to start tracing:

   logman start vss -o *x:*vss.etl -ets -p {9138500e-3648-4edb-aa4c-859e9f7b7c38} 0xfff 170

   > **NOTE**
   > Replace "x:\" with the path to the directory where you would like the trace log file to be stored.

2. Use the following command to stop tracing:

   logman stop vss -ets

The trace log file is *x:*vss.etl.

For more information about the Logman tool, see Logman.

Using Tracelog

The following procedure describes how to use Tracelog.

To use Tracelog

1. Create a text file named vss.ctl that contains only the following text:

   9138500e-3648-4edb-aa4c-859e9f7b7c38 vss

2. Use the following command to start tracing:

   tracelog -start vss -f *x:*vss.etl -guid vss.ctl -flag 0xff -level 0xaa

   > **NOTE**
   > Replace "x:\" with the path to the directory where you would like the trace log file to be stored.

3. Use the following command to stop tracing:

   tracelog -stop vss

The trace log file is *x:*vss.etl.

For more information about the Tracelog tool, see Tracelog.

Using VSS Diagnostics

3/5/2021 • 2 minutes to read • Edit Online

Vsdiagview and Vssagent are tools that you can use to troubleshoot VSS applications.

> **NOTE**
> These tools are included in the Microsoft Windows Software Development Kit (SDK) for Windows Vista and later. You can download the Windows SDK from https://msdn.microsoft.com/windowsvista.

In the Windows SDK installation, these tools can be found in `%Program Files(x86)%\Windows Kits\8.1\bin\x64` (for 64-bit Windows) and `%Program Files(x86)%\Windows Kits\8.1\bin\x86` (for 32-bit Windows).

Gathering Data

You can gather data by using the Vssagent command.

To gather data using Vssagent

1. To gather data from the command line, use the Vssagent command as follows:

   vssagent -gather *XmlFileName*.xml

2. To view the output file, see the following Viewing Data section.

Viewing Data

The Vsdiagview application can be used to view data that was gathered by using the Vssagent command. Vsdiagview displays the following information about the computer:

- Information about the computer, such as the operating system version, total available memory, and CPU speed
- The names and version numbers of the VSS binaries
- The names and properties of the disk and volume devices
- The names and properties of any COM+ applications
- The names of event logs that can be used for diagnosing VSS problems
- The names of any VSS writers and the events that they handle
- Information about other operating system files

To view data using Vsdiagview

1. In the File menu, choose **Open** to browse for a file. (The default location is C:\vssdiag.)
2. Click **Open** to view the data.

VssSampleProvider Tool and Sample

3/5/2021 • 2 minutes to read • Edit Online

Shows how to use the VSS interfaces to create a VSS hardware provider.

> **NOTE**
> The VssSampleProvider tool and sample are included in the Microsoft Windows Software Development Kit (SDK). You can download the Windows SDK from Windows Software Development Kit (SDK) for Windows 8.

In the Windows SDK installation, the VssSampleProvider tool can be found in `%Program Files(x86)%\Windows Kits\8.1\bin\x64` (for 64-bit Windows) and `%Program Files(x86)%\Windows Kits\8.1\bin\x86` (for 32-bit Windows).

> **NOTE**
> Hardware providers are only supported on Windows Server operating systems. On a Windows Client operating system, you can compile the VssSampleProvider sample, but you can't register it as a hardware provider.

The VssSampleProvider tool consists of the following files:

- Virtualstorage.sys
- Vstorcontrol.exe
- Vssampleprovider.dll
- Vstorinterface.dll

The VssSampleProvider sample includes the following installation and uninstallation scripts:

- Install-sampleprovider.cmd
- Uninstall-sampleprovider.cmd
- Register_app.vbs

To install and use the VssSampleProvider sample

1. Navigate to the `Program Files (x86)\Windows Kits\8.0\bin\` directory. This directory contains virtualstoragevss.sys and vstorcontrol.exe.
2. Open a command prompt window in the specified directory.
3. To install the virtual storage driver, in the command prompt window, type the following command:

```
vstorcontrol.exe install
```

4. To install the VSS sample provider, in the command prompt window, type the following command:

```
install-sampleprovider.cmd
```

5. To create a virtual LUN, do the following.

   a. In the command prompt window, type the following command:

   ```
   vstorcontrol.exe create fixeddisk -
   newimage C:\disk1.image -size 20M -storid "VSS Sample HW Provider"
   ```

   This command creates a virtual LUN whose storage identifier is **VSS Sample HW Provider**. To create additional virtual LUNs, repeat this step.

   The VSS sample provider recognizes a LUN only if **VSS Sample HW Provider** is a part of the storage identifier. For more information about the storage identifier, see the following blog post.

   LUN - Resync with Diskshadow and Virtual Storage

   b. In the command prompt window, use diskpart.exe to format the virtual disk and assign a drive letter to it.

   Here is a sample script to run at the diskpart command prompt.

   ```
   Select disk
   Create partition primary size=<size>
   Format FS=NTFS quick
   Assign Letter=<letter>
   Exit
   ```

6. To run the sample provider, in the command prompt window, type the following command:

   ```
   Run vshadow.exe -p -nw <drive>
   ```

   represents the drive letter of the virtual LUN.

7. To uninstall the VSS sample provider, in the command prompt window, type the following command:

   ```
   uninstall-sampleprovider.cmd
   ```

8. To uninstall the virtual storage driver, in the command prompt window, type the following command:

   ```
   vstorcontrol.exe uninstall
   ```

VShadow Tool and Sample

3/5/2021 • 13 minutes to read • Edit Online

VShadow is a command-line tool that you can use to create and manage volume shadow copies.

> **NOTE**
> VShadow is included in the Microsoft Windows Software Development Kit (SDK) for Windows Vista and later. The VSS 7.2 SDK includes a version of VShadow that runs only on Windows Server 2003. For information about downloading the Windows SDK and the VSS 7.2 SDK, see Volume Shadow Copy Service.

The VShadow tool uses command-line options and optional flags to identify the work to perform. When used without any command-line options, the VShadow command creates a new shadow copy set.

VShadow commands perform the following operations:

- Creating a Shadow Copy Set
- Importing a Shadow Copy Set
- Querying Shadow Copy Properties
- Deleting Shadow Copies
- Breaking a Shadow Copy Set
- Breaking a Shadow Copy Set Using the BreakSnapshotSetEx Method
- Exposing a Shadow Copy Locally
- Exposing a Shadow Copy Remotely
- Listing Writer Status and Metadata
- Performing Restore Operations
- Reverting to a Previous Shadow Copy
- Resynchronizing LUNs

Creating a Shadow Copy Set

vshadow [OptionalFlags] *VolumeList*

This command creates a new shadow copy set.

VolumeList is a list of volume names. VShadow creates one shadow copy for each volume in the list. A volume name can optionally be terminated with a backslash (\). For example, both C: and C:\ are valid volume names. You can also specify mounted folders (for example, C:\DirectoryName) or volume GUID names (for example, \\?\Volume{edbed95e-7e8d-11d8-9d01-505054503030}).

OptionalFlags is a bitmask of optional flag values from the following table.

OPTIONAL FLAG VALUE	DESCRIPTION

OPTIONAL FLAG VALUE	DESCRIPTION
-ad	This optional flag specifies differential hardware shadow copies. This flag and the **-ap** flag are mutually exclusive. [!Note] This flag is supported only on Windows server operating systems.
-ap	This optional flag specifies plex hardware shadow copies. This flag and the **-ad** flag are mutually exclusive. [!Note] This flag is supported only on Windows server operating systems.
-bc= *File*.**xml**	This optional flag specifies non-transportable shadow copies and stores the Backup Components Document into the specified file. This file can be used in a subsequent restore operation. This flag and the **-t** flag are mutually exclusive. [!Note] This flag is supported only on Windows server operating systems.
-exec= *Command*	This optional flag executes a command or script after the shadow copies are created but before the VShadow tool exits. *Command* can specify an executable shell command or a CMD file. If it specifies a shell command, no command parameters can be specified. [!Note] For security reasons and to keep the implementation simple, the **-exec** optional flag will not accept parameters to be passed to the command or script. The entire string passed to the **-exec** optional flag is treated as a single CMD or EXE file. For more information about this limitation, see the VShadow source code.
-forcerevert	This optional flag specifies that the shadow copy operation should be completed only if all disk signatures can be reverted. [!Note] This flag is supported only on Windows server operating systems. **Windows Server 2003:** Not supported.

OPTIONAL FLAG VALUE	DESCRIPTION
-mask	This optional flag specifies that the shadow copy LUNs should be masked from the computer when the shadow copy set is broken. [!Note] This flag is supported only on Windows server operating systems. **Windows Server 2003:** Not supported.
-nar	This optional flag specifies shadow copies without auto-recovery. For more information about this option, see the documentation for the VSS_VOLSNAP_ATTR_NO_AUTORECOVERY flag of the **_VSS_VOLUME_SNAPSHOT_ATTRIBUTES** enumeration.
-norevert	This optional flag specifies that disk signatures should not be reverted. [!Note] This flag is supported only on Windows server operating systems. **Windows Server 2003:** Not supported.
-nw	This optional flag specifies shadow copies without involving writers. For more information about shadow copies without writer participation, see Shadow Copy Creation Details. This flag and the **-wi** and **-wx** flags are mutually exclusive.
-p	This optional flag specifies *persistent shadow copies*. [!Note] This flag is supported only on Windows server operating systems.
-rw	This optional flag specifies that the shadow copy LUNs should be made read/write when the shadow copy set is broken. [!Note] This flag is supported only on Windows server operating systems. **Windows Server 2003:** Not supported.

OPTIONAL FLAG VALUE	DESCRIPTION
-scsf	This optional flag specifies *client-accessible shadow copies*. [!Note] This flag is supported only on Windows server operating systems.
-script= *File*.**cmd**	This optional flag generates a CMD file containing environment variables related to the created shadow copies, such as shadow copy IDs and shadow copy set IDs.
-t= *File*.**xml**	This optional flag specifies transportable shadow copies and stores the Backup Components Document into the file specified by the *File.xml* parameter. This file can be used in a subsequent import and/or restore operation. This flag and the **-bc** flag are mutually exclusive. **Windows Server 2003, Standard Edition and Windows Server 2003, Web Edition:** Transportable shadow copies are not supported. All editions of Windows Server 2003 with Service Pack 1 (SP1) support transportable shadow copies.
-tr	This optional flag specifies that TxF recovery should be enforced during shadow copy creation. [!Note] This flag is supported only on Windows server operating systems.
-tracing	This optional flag generates verbose output that can be used for troubleshooting.
-wait	This optional flag causes the VShadow tool to wait for user confirmation before exiting.
-wi= *Writer*	This optional flag verifies that the specified writer or component is included in the shadow copy. *Writer* can specify a component path, writer name, writer ID, or writer instance ID. This flag and the **-nw** flag are mutually exclusive.
-wx= *Writer*	This optional flag verifies that the specified writer or component is excluded from the shadow copy. *Writer* can specify a component path, writer name, writer ID, or writer instance ID. This flag and the **-nw** flag are mutually exclusive.

Importing a Shadow Copy Set

vshadow [OptionalFlags] **-i=** ***File*.xml**

The **-i** command-line option imports a transportable shadow copy set.

> **NOTE**
> This command-line option is supported only on Windows server operating systems.

The *File.xml* file must be a Backup Components Document file for a transportable shadow copy set that was created with the **-t** or **-bc** optional flag.

A shadow copy set is a *persistent shadow copy* if it was created with the **-p** optional flag. If the transportable shadow copy set is nonpersistent, it appears for a short time while the shadow copy creation command is running, and is automatically deleted when the command returns. You can import nonpersistent shadow copies only during shadow copy creation, by creating the shadow copy set using the **-exec** optional flag to execute a CMD file that calls VShadow **-i**.

The **-i** command-line option cannot be combined with other command-line options.

OptionalFlags is a bitmask of optional flag values from the following table.

OPTIONAL FLAG VALUE	DESCRIPTION
-exec= *Command*	This optional flag executes a command or script after the shadow copies are imported but before the VShadow tool exits. *Command* can specify an executable shell command or a CMD file. If it specifies a shell command, no command parameters can be specified.
-tracing	This optional flag generates verbose output that can be used for troubleshooting.
-wait	This optional flag causes the VShadow tool to wait for user confirmation before exiting.

Querying Shadow Copy Properties

vshadow -q

vshadow **-qx=** *ShadowCopySetId*

vshadow **-s=** *ShadowCopyId*

The **-q** command-line option lists the properties of all shadow copies on the computer.

The **-qx** command-line option lists the properties of all shadow copies in the shadow copy set whose ID is specified by *ShadowCopySetId*.

The **-s** command-line option lists the properties of the shadow copy whose ID is specified by *ShadowCopyId*.

These command-line options use a combination of **IVssBackupComponents::Query** and **IVssBackupComponents::GetSnapshotProperties** to get the properties of the given set of shadow copies.

The **-q**, **-qx**, and **-s** command-line options are mutually exclusive and cannot be combined with other command-line options.

Deleting Shadow Copies

vshadow -da

vshadow -do

vshadow **-dx=** ***ShadowCopySetId***

vshadow **-ds=** ***ShadowCopyId***

The **-da** command deletes all shadow copies on the computer.

> **NOTE**
> The **-da** command-line option requires user confirmation.

The **-do** command-line option deletes the oldest shadow copy on the computer.

The **-dx** command-line option deletes all shadow copies in the shadow copy set whose ID is specified by *ShadowCopySetId*.

The **-ds** command-line option deletes the shadow copy whose ID is specified by *ShadowCopyId*.

These command-line options are for use with *persistent shadow copies* only. Nonpersistent shadow copies do not need to be explicitly deleted, because they are automatically deleted when the VShadow creation command exits.

The **-da**, **-do**, **-dx**, and **-ds** command-line options are mutually exclusive and cannot be combined with other command-line options.

Breaking a Shadow Copy Set

vshadow **-b=** ***ShadowCopySetId***

vshadow **-bw=** ***ShadowCopySetId***

The **-b=** ***ShadowCopySetId*** command-line option converts each shadow copy in the shadow copy set into a stand-alone read-only volume.

The **-bw=** ***ShadowCopySetId*** command-line option converts each shadow copy in the shadow copy set into a stand-alone writable volume.

> **NOTE**
> The **-b** and **-bw** command-line options are supported only on Windows server operating systems.

For information about breaking a shadow copy set, see Breaking Shadow Copies.

After the shadow copy set is broken, the shadow copy set and the individual shadow copies no longer exist and cannot be managed using VSS commands.

A shadow copy set is persistent if it was created with the **-p** optional flag. If the transportable shadow copy set is nonpersistent, it appears for a short time while the shadow copy creation command is running, and is automatically deleted when the command returns. You can break nonpersistent shadow copy sets only during shadow copy creation, by creating the shadow copy set using the **-exec** optional flag to execute a CMD file that calls **vshadow -b** or **vshadow -bw**.

The **-b** and **-bw** command-line options are mutually exclusive and cannot be combined with other command-line options.

Breaking a Shadow Copy Set Using the BreakSnapshotSetEx Method

vshadow -bex

The **-bex** command-line option breaks a shadow copy set according to the options specified by the optional **-mask**, **-rw**, **-forcerevert**, and **-norevert** flags. This command-line option is similar to the **-b** and **-bw** command-line options. The **-bex** command-line option uses the **IVssBackupComponentsEx2::BreakSnapshotSetEx** method, whereas the **-b** and **-bw** command-line options use the **IVssBackupComponents::BreakSnapshotSet** method.

For information about breaking a shadow copy set, see Breaking Shadow Copies.

> **NOTE**
> The **-bex** command-line option is supported only on Windows server operating systems.

vshadow -bex -mask

The **-mask** flag specifies that the shadow copy LUN will be masked from the host. If the **-mask** flag is specified, the **-rw**, **-forcerevert**, and **-norevert** flags cannot be specified.

vshadow -bex -rw

The **-rw** flag specifies that the shadow copy LUN will be exposed to the host as a read/write volume.

vshadow -bex -forcerevert

The **-forcerevert** flag specifies that the disk identifiers of all of the shadow copy LUNs will be reverted to that of the original LUNs. However, if any of the original LUNs are present on the system, the operation will fail and none of the identifiers will be reverted. This flag and the **-norevert** flag are mutually exclusive.

vshadow -bex -norevert

The **-norevert** flag specifies that none of the shadow copy LUN disk identifiers will be reverted. This flag and the **-forcerevert** flag are mutually exclusive.

Exposing a Shadow Copy Locally

vshadow **-el=***ShadowCopyId*,** *LocalEmptyDirectory*

vshadow **-el=***ShadowCopyId*,** *UnusedDriveLetter*

The **-el** command-line option assigns a mounted folder or a drive letter to a persistent shadow copy. Note that the volume contents will remain read-only unless you subsequently call **vshadow -bw** to break the shadow copy set.

> **NOTE**
> Nonpersistent shadow copies and *client-accessible shadow copies* cannot be exposed locally.

For example, if {edbed95e-7e8d-11d8-9d01-505054503030} is the GUID of an existing persistent shadow copy,

and X: is an unused drive letter, the following command exposes the shadow copy under X:

vshadow -el={edbed95e-7e8d-11d8-9d01-505054503030},x:

The following example shows how to expose the same shadow copy under the mounted folder C:\ShadowCopies\June23:

mkdir c:\ShadowCopies\June23

vshadow -el={edbed95e-7e8d-11d8-9d01-505054503030},c:\ShadowCopies\June23

The **-el** command-line option cannot be combined with other command-line options.

Exposing a Shadow Copy Remotely

vshadow **-er=***ShadowCopyId*,** *UnusedShareName*

vshadow **-er=***ShadowCopyId*,*UnusedShareName*,** *PathFromRootOnShadow*

The **-er** command-line option assigns a read-only share name to the root directory (or a subdirectory) from the shadow copy. Note that the share contents will remain read-only unless you subsequently call **vshadow -bw** to break the shadow copy set.

> **NOTE**
> *Client-accessible shadow copies* cannot be exposed remotely.

For example, if {edbed95e-7e8d-11d8-9d01-505054503030} is the GUID of an existing persistent shadow copy, and share_123 is an unused share name, the following command exposes the shadow copy under share_123:

vshadow -er={edbed95e-7e8d-11d8-9d01-505054503030},share_123

The following example shows how to expose only a subtree (named "Folder1\Folder2") of the same shadow copy under the same share:

vshadow -er={edbed95e-7e8d-11d8-9d01-505054503030},share_123,Folder1\Folder2

The **-er** command-line option cannot be combined with other command-line options.

Listing Writer Status and Metadata

vshadow -ws

vshadow -wm

vshadow -wm2

vshadow -wm3

The **-ws** command-line option enumerates the VSS writers that are currently running on the computer and describes their state. This command is the equivalent of the Vssadmin list writers command. There are six possible state values: Stable, Failed, Unknown, Waiting for freeze, Frozen, and Waiting for completion.

The **-wm** command-line option lists a summary of the writer metadata, including the affected volumes.

The **-wm2** command-line option lists all writer metadata.

The **-wm3** command-line option lists all writer metadata in raw XML format.

Windows Vista and Windows Server 2003: The **-wm3** command-line option is not supported.

You can use this information to determine which volumes to include in the volume shadow copy set.

> **NOTE**
> If the writer state is Stable or Waiting for completion, the writer can be considered to be in a stable state, ready for the next backup.

The **-ws**, **-wm**, **-wm2**, and **-wm3** command-line options are mutually exclusive and cannot be combined with other command-line options.

Performing Restore Operations

vshadow [OptionalFlags] **-r=***File***.xml**

vshadow [OptionalFlags] **-rs=***File***.xml**

The **-r** command-line option performs a restore operation.

The **-rs** command-line option performs a simulated restore operation.

The *File.xml* file must be a Backup Components Document file for a shadow copy set that was created with the **-t** or **-bc** optional flag.

The **-r** and **-rs** command-line options are mutually exclusive and cannot be combined with other command-line options.

OptionalFlags is a bitmask of optional flag values from the following table.

OPTIONAL FLAG VALUE	DESCRIPTION
-wi=_Writer_	This optional flag verifies that the specified writer or component is included in the restore. *Writer* can specify a component path, writer name, writer ID, or writer instance ID.
-wx=_Writer_	This optional flag verifies that the specified writer or component is excluded from the restore. *Writer* can specify a component path, writer name, writer ID, or writer instance ID.
-exec=_Command_	This optional flag executes a command or script between the pre-restore and post-restore events that are sent to the writers. *Command* can specify an executable shell command or a CMD file. If it specifies a shell command, no command parameters can be specified.
-tracing	This optional flag generates verbose output that can be used for troubleshooting.

Reverting to a Previous Shadow Copy

vshadow **-revert=**_ShadowCopyId_

> **NOTE**
> This command-line option is supported only on Windows server operating systems.

Windows Server 2008 and Windows Server 2003: Not supported.

The **-revert** command-line option reverts a volume to the previous shadow copy whose ID is specified by *ShadowCopyId*.

The **-revert** command-line option cannot be combined with other command-line options.

Resynchronizing LUNs

vshadow **-addresync=**_ShadowCopyId_ **-resync=**_BCDFileName_ [OptionalResyncFlags]

vshadow -addresync=*ShadowCopyId*, _TargetVolumeDriveLetter_ **-resync=**_BCDFileName_ [OptionalResyncFlags]

The **-addresync** command-line option specifies the volumes to be included in a LUN resynchronization operation. The *ShadowCopyId* parameter specifies the identifier of the shadow copy. The optional *TargetVolumeDriveLetter* parameter specifies the destination volume where the contents of the shadow copy volume are to be copied.

The **-resync** command-line option initiates a LUN resynchronization operation. After the operation is complete, the signature of each target LUN should be identical to that of the target volume LUN. The *BCDFileName* parameter specifies the name of the XML file that contains the Backup Component Document.

> **NOTE**
> The **-addresync** and **-resync** command-line options are supported only on Windows server operating systems.

Windows Server 2008 and Windows Server 2003: Not supported.

OptionalResyncFlags is a bitmask of optional flag values from the following table.

OPTIONAL FLAG VALUE	DESCRIPTION
-revertsig	This optional flag specifies that after the operation is complete, the signature of each target LUN should be identical to that of the original LUN that was used to create the shadow copy, not the target volume LUN. [!Note] The **-revertsig** flag is supported only on Windows server operating systems. **Windows Server 2008 and Windows Server 2003:** Not supported.

OPTIONAL FLAG VALUE	DESCRIPTION
-novolcheck	This optional flag specifies that the VSS service should not check for unselected volumes that would be overwritten by the LUN resynchronization operation. [!Note] The **-novolcheck** flag is supported only on Windows server operating systems. **Windows Server 2008 and Windows Server 2003:** Not supported.

VShadow Tool Examples

3/5/2021 • 7 minutes to read • Edit Online

The following examples show how to use the VShadow tool to perform the most common tasks:

- Accessing Shadow Copy Properties from a CMD Script
- Accessing Nonpersistent Shadow Copies
- Copying a File from a Shadow Copy
- Enumerating All Files on a Shadow Copy Device
- Importing a Transportable Shadow Copy
- Including Writers or Components
- Excluding Writers or Components
- Breaking Shadow Copy Sets Using the BreakSnapshotSetEx Method

For a complete list of command-line options and their use, see VShadow Tool and Sample.

Accessing Shadow Copy Properties from a CMD Script

If you use the **-script** optional flag when you create shadow copies, VShadow creates a CMD script file containing environment variables related to the newly created shadow copies, such as the following:

- The shadow copy set ID, which is stored in the %VSHADOW_SET_ID% environment variable
- The shadow copy IDs, which are stored in variables of the form %VSHADOW_ID_*NNN*%, where *NNN* represents the index of the source volume in the VShadow command line
- The shadow copy device names, which are stored in variables of the form %VSHADOW_DEVICE_*NNN*%, where *NNN* is the index of the source volume in the VShadow command line

You can use the generated CMD file to perform limited management operations on the shadow copies.

> **NOTE**
>
> The **BackupComplete** writer event is sent after the **-exec** script is executed. VSS calls the **IVssBackupComponents::BackupComplete** method to signal to the writers that the backup is completed, and the writers can potentially truncate logs at this point. Therefore, it is important to verify that the shadow/backup actually completed. If the backup failed, you can cancel the creation process by returning a nonzero exit code in the executed script/command. (See the documentation for the exit command in the Windows Command Line Reference.) If the custom command returns with a nonzero exit code, then the shadow copy creation is canceled and **IVssBackupComponents::BackupComplete** will not be called.

The following example shows how to create a persistent shadow copy from the command line and use the CMD script to expose it.

1. **vshadow -p -nw -script=SETVAR1.cmd c:**
2. **call SETVAR1.cmd**
3. **C:\>vshadow -el=%VSHADOW_ID_1%,c:\directory1**

Accessing Nonpersistent Shadow Copies

Nonpersistent shadow copies are automatically deleted when the program that creates them (in this case, VShadow) exits. To access the contents of these shadow copies, VShadow allows you to execute a script after the shadow copies are created but before the VShadow program exits.

The following example shows how to use the -exec command-line option to run a script called "enum.cmd":

vshadow -nw -exec=enum.cmd c:

In the executed script, you can also run the generated SETVAR script in this custom command. This allows the script to access the contents of the shadow copies directly. Remember that the shadow device can be retrieved from the VSHADOW_DEVICE_NNN environment variable.

Here is an example enum.cmd script:

```
call SETVAR1.cmd
for /R %VSHADOW_DEVICE_1%\ %%i in (*.*) do @echo %%i
```

Here is the command line to execute the enum.cmd script:

vshadow -nw -exec=c:\enum.cmd -script=setvar1.cmd c:

Copying a File from a Shadow Copy

The following example shows how to copy a file from a shadow copy.

1. **dir > c:\somefile.txt**
2. **vshadow -p -nw -script=SETVAR1.cmd c:**
3. **call SETVAR1.cmd**
4. **copy %VSHADOW_DEVICE_1%\somefile.txt c:\somefile_bak.txt**

Enumerating All Files on a Shadow Copy Device

The following example shows how to enumerate all files on a shadow copy device from a batch file.

1. **dir > c:\somefile.txt**
2. **vshadow -p -nw -script=SETVAR1.cmd c:**
3. **call SETVAR1.cmd**
4. **for /R %VSHADOW_DEVICE_1%\ %%i in (*) do @echo %%i**

Importing a Transportable Shadow Copy

To create a transportable shadow copy on one machine and import it to another machine, you must have two computers that are connected (through a SAN configuration) to a storage array that supports VSS hardware shadow copies. A VSS hardware provider must be installed on each computer.

The following example shows how to create and import the shadow copy.

1. Create the shadow copy set on computer A (the production server) by typing the following command after the command prompt:

   vshadow -p -t=bc1.xml

   This will not expose any shadow copy devices on computer A.

2. Copy the backup components document (bc1.xml) from computer A to computer B.

3. Import the shadow set on machine B by typing the following command:

```
vshadow -i=bc1.xml
```

Optionally, you could expose these shadow copies as drive letters or mounted folders. Also, you could potentially break the shadow set to make the new shadow copy volumes read-write.

To automate the shadow set management process you might use the **-script** command-line option to generate the CMD script containing the shadow copy IDs. Then you can copy the generated script along with the backup components to the other computer.

The following example shows how to create, expose, and break transportable shadow copies using CMD scripts.

1. Create the shadow copy set on computer A (the production server) from the command line as follows:

   ```
   vshadow -p -t=bc1.xml -script=sc1.cmd
   ```

   Specify the **-script** option to generate the script containing the proper environment variable definitions. Note that this will not expose any shadow copy devices on computer A.

2. Copy the backup components document (bc1.xml) and the generated script (sc1.cmd) from computer A to computer B.

3. Import the shadow set on machine B by typing the following command:

   ```
   vshadow -i=bc1.xml
   ```

   This will surface the created devices on computer B.

4. Run the generated script to set the environment variables for the shadow copy set by typing the file name at the command prompt as in the following example:

   ```
   sc1.cmd
   ```

   This will set the relevant environment variables as described in Access Shadow Copy Properties from a CMD Script.

5. Expose the shadow copies on computer B by typing the following commands:

   a. `rmdir /s c:\mount_point`
   b. `mkdir c:\mount_point`
   c. `vshadow –el=%VSHADOW_ID_1%,c:\mount_point`

   This will surface the created shadow copy devices on computer B.

6. Break the shadow copy set to make the volumes writable by typing the following command:`C:\>vshadow –bw=%VSHADOW_SET_ID%`

Note that nonpersistent transportable shadow copies can also be imported, but they are automatically deleted when the **vshadow -i** process exits. To import the shadow copies before they are deleted, you must use the **-exec** command-line option as described in Access Nonpersistent Shadow Copies.

Including Writers or Components

By default, all writers participate in shadow copy creation. To determine which writers and components will be included in a shadow copy set, use the **-wm** or **-wm2** command-line option as described in Listing Writer Status and Metadata.

To verify that all of a writer's components are included, use the **-wi** optional flag in any of the following formats:

- **vshadow -wi** *WriterName*
- **vshadow -wi "***Writer Name String***"**
- **vshadow -wi {***WriterID***}**

- **vshadow -wi {*InstanceID*}**

To verify that specific components are included, use the **-wi** optional flag in any of the following formats:

- **vshadow -wi** *WriterName***:*LogicalPath****ComponentName*
- **vshadow -wi** ***"Writer Name String*":*LogicalPath****ComponentName*
- **vshadow -wi** ***{*WriterID*}:*LogicalPath****ComponentName*
- **vshadow -wi** ***{*InstanceID*}:*LogicalPath****ComponentName*

The following examples show how to use the **-wi** optional flag.

- Specifying *WriterName*:*LogicalPath**ComponentName*:

  vshadow -wi="Registry Writer:\Registry"

- Specifying {*WriterID*}:

  vshadow -wi={BE000CBE-11FE-4426-9C58-531AA6355FC4}

- Specifying more than one writer or component:

  vshadow -wi="Registry Writer:\Registry" -wi="COM+ REGDB Writer"

Excluding Writers or Components

To exclude all writers, use the **-nw** optional flag when creating the shadow copy.

To exclude all of a writer's components, use the **-wx** optional flag in any of the following formats:

- **vshadow -wx** *WriterName*
- **vshadow -wx "*Writer Name String*"**
- **vshadow -wx {*WriterID*}**
- **vshadow -wx {*InstanceID*}**

To exclude specific components, use the **-wx** optional flag in any of the following formats:

- **vshadow -wx** *WriterName***:*LogicalPath****ComponentName*
- **vshadow -wx** ***"Writer Name String*":*LogicalPath****ComponentName*
- **vshadow -wx** ***{*WriterID*}:*LogicalPath****ComponentName*
- **vshadow -wx** ***{*InstanceID*}:*LogicalPath****ComponentName*

The following examples show how to use the **-wx** optional flag.

- Specifying *WriterName*:*LogicalPath**ComponentName*:

  vshadow -wx="Registry Writer:\Registry"

- Specifying {*WriterID*}:

  vshadow -wx={BE000CBE-11FE-4426-9C58-531AA6355FC4}

- Specifying more than one writer or component:

  vshadow -wx="Registry Writer:\Registry" -wx="COM+ REGDB Writer"

Breaking Shadow Copy Sets Using the BreakSnapshotSetEx Method

The following examples show how to use the **-bex** command-line option.

- Specifying that the shadow copy LUN will be masked from the host:

```
vshadow -bex -mask
```

- Specifying that the shadow copy LUN will be exposed to the host as a read/write volume:

  ```
  vshadow -bex -rw
  ```

- Specifying that the shadow copy LUN will be exposed to the host as a read/write volume, and that none of the disk signatures on the shadow copy LUNs will be reverted to that of the original LUNs:

  ```
  vshadow -bex -norevert
  ```

BETest Tool

3/5/2021 • 6 minutes to read • Edit Online

BETest is a VSS requester that tests advanced backup and restore operations. This tool can be used to test an application's use of complex VSS features such as the following:

- Incremental and differential backup
- Complex restore options, such as authoritative restore
- Rollforward options

> **NOTE**
> BETest is included in the Microsoft Windows Software Development Kit (SDK) for Windows Vista and later. The VSS 7.2 SDK includes a version of BETest that runs only on Windows Server 2003. This topic describes the Windows SDK version of BETest, not the Windows Server 2003 version included in the VSS 7.2 SDK. For information about downloading the Windows SDK and the VSS 7.2 SDK, see Volume Shadow Copy Service.

In the Windows SDK installation, the BETest tool can be found in `%Program Files(x86)%\Windows Kits\8.1\bin\x64` (for 64-bit Windows) and `%Program Files(x86)%\Windows Kits\8.1\bin\x86` (for 32-bit Windows).

Running the BETest Tool

To run the BETest tool from the command line, use the following syntax:

BETest *command-line-options*

The following usage example shows how to use the BETest tool together with the VSS Test Writer tool, which is a VSS writer.

BETest Tool Usage Example

1. Create a test directory named C:\BETest. Copy the following files into this directory:
   - betest.exe
   - vswriter.exe
   - BetestSample.xml
   - VswriterSample.xml
2. Create a directory named C:\TestPath. Put some test data files in this directory.
3. Create a directory named C:\BackupDestination. Leave this directory empty.
4. Open two elevated command windows and set the working directory in each to C:\BETest.
5. In the first command window, start the VSS Test Writer tool as follows:

   vswriter.exe VswriterSample.xml

   The vswriterSample.xml file configures the VSS Test Writer tool (vswriter) to report the contents of the c:\TestPath directory in preparation for a backup operation. Note that the VSS Test Writer tool will not produce output until it detects activity from a requester such as BETest. To stop the VSS Test Writer tool, press CTRL+C.

6. In the second command window, use the BETest tool to perform a backup operation as follows:

   betest.exe /B /S backup.xml /D C:\BackupDestination /X BetestSample.xml

   BETest will back up the files from the C:\TestPath directory to the C:\BackupDestination directory. It will save the backup component document to C:\BETest\backup.xml.

7. If the backup operation is successful, delete the contents of the C:\TestPath directory, and use the BETest tool to perform a restore operation as follows:

   betest.exe /R /S backup.xml /D C:\BackupDestination /X BetestSample.xml

BETest Tool Command-Line Options

The BETest tool uses the following command-line options to identify the work to perform.

/Auth

Performs an authoritative restore operation for Active Directory or Active Directory Application Mode.

Windows Server 2003: This command-line option is not supported.

/B

Performs a backup operation but does not perform a restore.

/BC

Performs only the backup complete operation.

Windows Server 2003: This command-line option is not supported.

/C *Filename*

> **NOTE**
> This command-line option is provided only for backward compatibility. The **/X** command-line option should be used instead.

Selects the components to be backed up or restored based on the contents of the configuration file specified by *Filename*. This file must contain only ANSI characters in the range from 0 through 127, and it must be no larger than 1 MB. Each line in the file must use the following format:

WriterId : *ComponentName*;

Where *WriterId* is the writer ID, and *ComponentName* is the name of one of the writer's components. The writer ID and component names must be in quotation marks, and there must be spaces before and after the colon (:). If two or more components are specified, they must be separated by commas. For example:

"5affb034-969f-4919-8875-88f830d0ef89" : "TestFiles1", "TestFiles2", "TestFiles3";

/D *Path*

Save the backed-up files to or restore them from the backup directory specified by *Path*.

/NBC

Omits the backup complete operation.

Windows Server 2003: This command-line option is not supported.

/O

Specifies that the backup includes a bootable system state.

/P

Creates a persistent shadow copy.

Windows Server 2003: This command-line option is not supported.

/Pre *Filename*

If the backup type specified in the **/T** command-line option is INCREMENTAL or DIFFERENTIAL, set the backup document to the file specified by *Filename* for previous full or incremental backup.

Windows Server 2003 and Windows XP: This command-line option is not supported.

/R

Performs restore but does not perform backup. Must be used together with the **/S** command-line option.

/Rollback

Creates a shadow copy that can be used for application rollback.

Windows Server 2003: This command-line option is not supported.

/S *Filename*

In case of backup, saves the backup document to the file specified by *Filename*. In case of restore only, loads the backup document from this file.

/Snapshot

Creates a volume shadow copy but does not perform backup or restore.

Windows Server 2003: This command-line option is not supported.

/StopError

Stops BETest when the first writer error is encountered.

Windows Server 2003: This command-line option is not supported.

/T *BackupType*

Specifies the backup type. *BackupType* can be FULL, LOG, COPY, INCREMENTAL, or DIFFERENTIAL. For more information about backup types, see VSS_BACKUP_TYPE.

/V

Generates verbose output that can be used for troubleshooting.

Windows Server 2003: This command-line option is not supported.

/X *Filename*

Selects the components to be backed up or restored based on the contents of the XML configuration file specified by *Filename*. This file must contain only ANSI characters in the range from 0 through 127. The format of the XML file is defined by the schema in the BETest.xml file. For a sample configuration file, see BetestSample.xml. Both of these files are in the vsstools directory.

> **NOTE**
> You can view the BETest.xml file in Internet Explorer. Before you open this file, make sure that the xdr-schema.xsl file is in the same directory as BETest.xml. The xdr-schema.xsl file contains rendering instructions that make the BETest.xml file more readable.

Windows Server 2003: This command-line option is not supported.

Sample XML Configuration File: BetestSample.xml

The following sample configuration file, BetestSample.xml, can be found in the Vsstools directory.

```
<BETest>
    <Writer writerid="5affb034-969f-4919-8875-88f830d0ef89">
        <Component componentName="TestFiles">
        </Component>
    </Writer>
</BETest>
```

This example of a simple configuration file selects one component to be backed up or restored.

Sample XML Configuration File: VswriterSample.xml

The following sample configuration file, VswriterSample.xml, can be found in the Vsstools directory.

```
<TestWriter   usage="USER_DATA"
                    deleteFiles="no">

    <RestoreMethod method="RESTORE_IF_CAN_BE_REPLACED"
                   writerRestore="always"
                   rebootRequired="no" />

    <Component componentType="filegroup"
               componentName="TestFiles">
               <ComponentFile path="c:\TestPath" filespec="*" recursive="no" />
    </Component>

</TestWriter>
```

The root element in this configuration file is named TestWriter. All other elements are arranged under the TestWriter element.

The first attribute associated with TestWriter is the usage attribute. This attribute specifies the usage type reported through the **IVssExamineWriterMetadata::GetIdentity** method. One of the possible values for this attribute is USER_DATA.

The second attribute is the deleteFiles attribute. This attribute is described in Configuring Writer Attributes.

The first child element of the root element is a RestoreMethod element. This element specifies the following:

- The restore method (in this case, RESTORE_IF_CAN_BE_REPLACED)
- Whether the writer requires restore events (in this case, always)
- Whether a reboot is required after the writer is restored (in this case, no)

This element can optionally specify an alternate-location mapping. (In this case, no alternate location is

specified.) For more information, see Specifying Alternate Location Mappings.

The second child element is a Component element. This element causes the writer to add a component to its metadata. A Component element contains attributes to describe the component and child elements to describe the content of the component, such as the following:

- componentType to indicate whether this is a filegroup or a database (in this case, a filegroup)
- logicalPath for the component logical path (in this case, none is specified)
- componentName for the name of the component (in this case, "TestFiles")
- selectable to indicate selectable-for-backup status

The Component element also has a child element named ComponentFile to add a file specification to this component. (A Component element can have an arbitrary number of ComponentFile elements that can be specified for each component.) This ComponentFile element has the following attributes:

- path="c:\TestPath"
- filespec="*"
- recursive="no"

VSS Test Writer Tool

3/5/2021 • 13 minutes to read • Edit Online

The Test Writer is a utility that you can use to test VSS requester applications. This writer can be configured to perform almost all of the actions that a VSS writer can perform. In addition, the Test Writer performs extensive checks to ensure that the requester has dealt with these writer actions correctly.

Each instance of the writer is initialized with an XML configuration file that describes exactly what components the writer will report on, and how the writer behaves. The writer can be configured to report on various types of scenarios, including more complicated scenarios using the incremental and differential interfaces. The writer will perform checks at various points during the process to ensure that the requester is behaving in a proper manner. After the restore is completed, the writer will check that all necessary files have been restored without corruption. The writer can also be configured to perform other operations such as failing specific events.

> **NOTE**
> This tool is included in the Microsoft Windows Software Development Kit (SDK) for Windows Vista and later. You can download the Windows SDK from https://msdn.microsoft.com/windowsvista.

In the Windows SDK installation, the VssSampleProvider tool can be found in `%Program Files(x86)%\Windows Kits\8.1\bin\x64` (for 64-bit Windows) and `%Program Files(x86)%\Windows Kits\8.1\bin\x86`.

Running the Test Writer Tool

To start the Test Writer, type the following at the command prompt:

vswriter.exe *config-file*

where *config-file* is the path to a configuration file that specifies the behavior of this writer.

To stop the Test Writer, press CTRL+C.

Multiple instances of the Test Writer can be run at the same time. However, each instance of the writer will report the same writer class ID (though a different writer instance ID), so logical paths and component names must be unique across all simultaneously running instances of the writer.

To ensure that the writer can properly check that the requester has honored the exclude file specifications, all files that were backed up should be deleted from the original volume before attempting to restore them. It is recommended that a template of each writer scenario be stored, so the scenario can be easily recreated.

Using a Configuration File

The following sample configuration file, vswriter_config.xml, can be found in `%ProgramFiles%\Microsoft SDKs\Windows\v7.0\bin\vsstools` on x86 platforms and `%ProgramFiles%\Microsoft SDKs\Windows\v7.0\bin\x64\vsstools` on x64 platforms.

```
<TestWriter usage="USER_DATA">

    <RestoreMethod method="RESTORE_IF_CAN_BE_REPLACED"
                   writerRestore="always"
                   rebootRequired="no" />

    <ExcludeFile path="c:\writerData\notTheseFiles"
                 filespec="excludeThisFile.txt"
                 recursive="no"/>

    <Component componentType="filegroup"
               componentName="TestFiles">
        <ComponentFile path="c:\writerData\myFilesHere"
                       filespec="*"
                       recursive="no" />
    </Component>

</TestWriter>
```

The root element in this configuration file is named TestWriter. All other elements are arranged under the TestWriter element.

One of the attributes associated with TestWriter is the usage attribute. This attribute specifies the usage type reported through **IVssExamineWriterMetadata::GetIdentity**. One of the possible values for this attribute is USER_DATA.

The first child element of the root element must always be a RestoreMethod element. This element specifies the following:

- The restore method (in this case, RESTORE_IF_CAN_BE_REPLACED)
- Whether the writer requires restore events (in this case, always)
- Whether a reboot is required after the writer is restored (in this case, no)

This element can optionally specify an alternate-location mapping. (In this case, no alternate location is specified.)

The second child element is an ExcludeFile element. This element causes the writer to exclude a file or a set of files from backup.

The third child element is a Component element. This element causes the writer to add a component to its metadata. A Component element contains attributes to describe the component and child elements to describe the content of the component, such as the following:

- componentType to indicate whether this is a filegroup or a database
- logicalPath for the component logical path
- componentName for the name of the component
- selectable to indicate selectable-for-backup status

The Component element also has a child element named ComponentFile to add a file specification to this component. (A Component element can have an arbitrary number of ComponentFile elements that can be specified for each component.) This ComponentFile element has the following attributes:

- path="c:\writerData\myFilesHere"
- filespec="*"
- recursive="no"

Although the Test Writer verifies requester behavior, it does not verify that the configuration file always maintains the documented semantics for a writer. There are many operations that a well-behaved writer should

not do (such as report the same file in multiple components), and it is up to the author of the XML configuration file to maintain these semantics.

Configuring Writer Attributes

The TestWriter root element contains attributes that determine various behaviors of the writer. Some of the behaviors that can be altered are the following:

ATTRIBUTE	DESCRIPTION
verbosity	The writer prints the status to the console as it receives events and processes them. The level of verbosity displayed is specified by the verbosity attribute. There are three verbosity levels to choose from: low Only failures in the writer or incorrect behavior from the requester will be printed. medium Everything at the low verbosity level is printed in addition to extra status information such as when events are received. This is the default level. high Detailed status information about the operation of the writer is reported.
deleteFiles	To perform extra verification, set this attribute to "yes" to cause the writer to delete all of its files immediately after the volume shadow copy is created. The requester must then copy the files from the shadow copy, because they no longer exist on the original volume. [!Note] In the case of spit writers, the files are deleted from the original location after the spit but before the shadow copy is created. After the shadow copy is created, the files are deleted from the spit directory.
deletePartialFiles	To delete partial files, set this attribute to "yes".
deleteDifferencedFiles	To delete differenced files, set this attribute to "yes".
checkIncludes	Set this attribute to "yes" to cause the writer to check that every file that has been backed up has been restored to an appropriate location, and that the file has not been corrupted. Partial files and differenced files are also correctly handled.
checkExcludes	Set this attribute to "yes" to cause the writer to check that files matching a file specification in the exclude list are not restored. For this to function correctly, the restore directories must be emptied prior to restore.

Specifying Alternate Location Mappings

An alternate location mapping specifies a location to restore to if the restore method is VSS_WRE_RESTORE_TO_ALTERNATE_LOCATION or if normal restore of a component fails. A writer can report its alternate location mappings to the requester by using the IVssExamineWriterMetadata::GetAlternateLocationMapping method. You can add alternate location mappings to the Test Writer configuration file by adding AlternateLocationMapping subelements to the RestoreMethod element.

The following RestoreMethod element contains a AlternateLocationMapping subelement.

```
<RestoreMethod method="RESTORE_IF_CAN_REPLACE"
               writerRestore="always"
               rebootRequired="no">
    <AlternateLocationMapping path="c:\files"
                              filespec="*.txt"
                              recursive="no"
                              alternatePath="c:\altfiles"/>
</RestoreMethod>
```

This example specifies that the requester should first attempt to restore all of the files matching c:\files*.txt to the c:\files directory. If one of these files cannot be replaced, the requester should restore all the files to the c:\altfiles directory instead. The requester should save this alternate location mapping by using the IVssBackupComponents::AddAlternativeLocationMapping method. If the Test Writer is configured to check whether files have been restored, it will also check whether the requester has called **AddAlternativeLocationMapping**.

Specifying Files to Be Excluded

Every writer can specify a list of file specifications that specify files for the requester to exclude from backup. You can add these file specifications to the Test Writer configuration file by adding ExcludeFile subelements to the TestWriter root element.

Here is an example of an ExcludeFile subelement that excludes all files in the C:\files directory that match C:\temp*.*.

```
    <ExcludeFile path="c:\files"
                 filespec="temp*.*"
                 recursive="no"/>
```

Backing Up Spit Writers

Many writers act as "spit writers." A spit writer creates intermediate files, or "spit files," based on an original set of files, and puts the spit files in an alternate location at backup time. The writer uses the IVssWMFiledesc::GetAlternateLocation method to notify the requester that it should back these files up from the alternate location. However, these files should still be restored to the active location of the original files. The Test Writer can be configured to act as a spit writer for a particular file specification.

The following ComponentFile element contains an alternatePath attribute:

```
    <ComponentFile path="c:\files"
                   filespec="*.txt"
                   recursive="no"
                   alternatePath="c:\files\spit" />
```

This example configures the Test Writer to copy all files matching c:\files*.txt to the c:\files\spit directory immediately before the volume shadow copy is created. The requester must back up the files from the

c:\files\spit directory. If the Test Writer is configured to delete files, it deletes the original files before the shadow copy is created, so they do not appear in the c:\files directory on the shadow copy volume. In this case, the files in c:\files\spit are deleted after the shadow copy is created, so they must be backed up from the c:\files\spit directory on the shadow copy volume.

Reporting Component Dependencies

Writers can specify a dependency between a local component and a component that exists in another writer. These dependencies are reported to the requester using **IVssWMComponent::GetDependency**. The Test Writer can be configured to report these dependencies by adding one or more Dependency subelements to the Component element.

The following Component element contains a Dependency subelement:

```
<Component componentType="filegroup"
           logicalPath=""
           componentName="WriterComponent"
           selectable="yes">
     <Dependency writerId="<GUID of target writer>"
                 logicalPath="ComponentPath"
                 componentName="Writer2Component" />
</Component>
```

The dependency is specified as attributes of the Dependency element. writerId is the class id of the writer containing the target of the dependency, logicalPath is the logical path to the component in that writer, and componentName is the name of the component. In this case, if the requester is backing up "WriterComponent" in the current writer, it should also back up the component "ComponentPath\Writer2Component" in the target writer.

> **NOTE**
> The Test Writer performs no checking to ensure dependencies are honored.

Failing Events

The Test Writer can be configured to fail any of the normal events that a writer receives. These events are as follows:

EVENT	DESCRIPTION
Identify	Received as a response to a **IVssBackupComponents::GatherWriterMetadata** call. Failure here will cause the writer to not be reported.
PrepareForBackup	Received as a response to the requester calling **IVssBackupComponents::PrepareForBackup**.
PrepareForSnapsot	Received when the requester calls **IVssBackupComponents::DoSnapshotSet**, but before the shadow copy is created.
Freeze	Received immediately after *PrepareForSnapshot*, but still before the shadow copy is created.

EVENT	DESCRIPTION
Thaw	Received after the shadow copy creation is finished.
PostSnapshot	Received after Thaw completes, but before **IVssBackupComponents::DoSnapshotSet** completes.
Abort	Received if too much time elapses between *Freeze* and *Thaw* or if the requester calls **IVssBackupComponents::AbortBackup**.
BackupComplete	Received when the requester exits. Failures here will never be reported.
PreRestore	Received when the requester calls **IVssBackupComponents::PreRestore**.
PostRestore	Received when the requester calls **IVssBackupComponents::PostRestore**.

These failures are configured by adding one or more FailEvent subelements to the TestWriter root element. There are two classes of failures that can be set: retryable and non-retryable. Retryable errors are errors that may go away if the entire backup process is retried, while non-retryable errors are unlikely to disappear. The failure type for the event is chosen based on the retryable attribute.

Here is an example of a basic non-retryable failure:

```
    <FailEvent writerEvent="Freeze"
               retryable="no" />
```

This example will cause the writer to fail during the *Freeze* event. **IVssBackupComponents::GatherWriterStatus** will report the writer failure to be VSS_E_WRITERERROR_NONRETRYABLE.

Here is an example of a basic retryable error:

```
    <FailEvent writerEvent="Freeze"
               retryable="yes"
               numFailures="2"/>
```

This example will cause the writer to fail the first two times it receives a *Freeze* event. After the first two times, the writer will always succeed. The writer failure reported through **GatherWriterStatus** will be a random retryable error code.

Declaring Supported Backup Types

Writers communicate what backup types are supported through the requester's calling **IVssExamineWriterMetadata::GetBackupSchema**. The TestWriter root element contains attributes for each backup type to indicate support. These attributes are supportsCopy, supportsDifferential, supportsIncremental, and supportsLog. To indicate support for a particular backup type, set the corresponding attribute to "yes".

If the requester sets the backup type to a backup type that is not supported by the writer, the writer will note this fact during **PrepareForBackup**, but otherwise work correctly.

Indicating the File Backup Type

The **IVssWMFiledesc::GetBackupTypeMask** method returns a bitmask to the requester indicating how a file should be backed up. This will indicate whether a file is required or not required during specific types of backup, and also whether a shadow copy is required during specific types of backup. The backup types in this bitmask are explained at greater length in the document Requester Role in Backing Up Complex Stores.

In the Test Writer, the elements of this bitmask are specified by setting attributes in each ComponentFile element. The fullBackupRequired, diffBackupRequired, incBackupRequired, and logBackupRequired attributes specify when a file needs to be backed up. The fullSnapshotRequired, diffSnapshotRequired, incSnapshotRequired, and logSnapshotRequired attributes specify when a file must be backed up from a shadow copy of a volume (and never from the original volume). The default for all of these values is "yes", indicating that a file must always be backed up and must be backed up from a shadow copy of a volume.

Adding Partial Files

During the processing of **DoSnapshotSet**, a writer has a chance to add partial files to each component. These partial files are reported using **IVssComponent::GetPartialFile**. The Test Writer can be configured to add partial files by specifying PartialFile subelements in a Component element.

Here is an example of a Component element that has two PartialFile subelements:

```
<Component componentType="filegroup"
           logicalPath=""
           componentName="WriterComponent"
           selectable="yes">
    <ComponentFile path="c:\files"
                   filespec="*.txt"
                   recursive="no"/>
    <PartialFile path="c:\files"
                 filespec="partial.txt"
                 ranges="0:5,100:20" />
    <PartialFile path="c:\files2"
                 filespec="partial.txt"
                 ranges="0:5,100:20" />
</Component>
```

Only partial files that partially match an existing ComponentFile (as in the first partial file in the example) or new partial files that are on the same volume as an existing ComponentFile (as in the second partial file) should be specified this way. For this component, the requester must fully back up all files matching c:\files*.txt except for partial.txt. The requester must then back up the specified ranges (where a range is an offset followed by a length) for the files c:\files\partial.txt and c:\files2\partial.txt.

If the writer is configured to check file restores, only the backed-up ranges of the partial file will be checked at restore time. Modifications to other portions of the file will go unnoticed. If the deletePartialFiles attribute of the TestWriter root element is set, the partial files are deleted from the original volume immediately after the shadow copy is created.

Adding Differenced Files

During the processing of **DoSnapshotSet**, a writer can add differenced files to each component. These differenced files are reported using **IVssComponent::GetDifferencedFile**. The Test Writer can be configured to add differenced files by specifying DifferencedFile subelements in a Component element.

Here is an example of a Component element that has two DifferencedFile subelements:

```
<Component componentType="filegroup"
           logicalPath=""
           componentName="WriterComponent"
           selectable="yes">
    <ComponentFile path="c:\files"
                   filespec="*.txt"
                   recursive="no"/>
    <DifferencedFile path="c:\files"
                     filespec="*.txt"
                     year="2007"
                     month="1"
                     day="22"
                     hour="12"
                     minute="44"
                     second="17" />
    <DifferencedFile path="c:\files2"
                     filespec="*.txt"
                     year="2007"
                     month="1"
                     day="22"
                     hour="12"
                     minute="44"
                     second="17" />
</Component>
```

Unlike partial files, differenced files should never partially match a ComponentFile specification. The file specification in a DifferencedFile element should either exactly match a ComponentFile (as in the first differenced file in the example) or it should not match it at all, but be on a volume that is referenced in a ComponentFile (as in the second differenced file). The date and time values should be relative to the local time zone, but they will be converted to GMT before being reported to the requester. In the example, only files matching c:\files*.txt or c:\files2*.txt that have been modified since 1/22/2003:12:44:17 will be backed up.

If the Test Writer is configured to check file restores, only the modified files will be checked for restore. If the deleteDifferencedFiles attribute of the TestWriter root element is set, the differenced files are deleted from the original volume immediately after the shadow copy is created.

New Targets

Certain writers allow the requester to inform them that a new location has been chosen to restore certain files to. The writer indicates that it supports this mode as part of the bitmask returned by **IVssExamineWriterMetadata::GetBackupSchema**. By default, the Test Writer always supports new targets. This support can be disabled by setting the supportsNewTarget attribute in the TestWriter root element to "no".

If a writer supports new targets, the requester can inform the writer of new targets by calling **IVssBackupComponents::AddNewTarget**. The writer will then check the new target location to verify restore rather than the original location.

More Information

The Test Writer supports more configuration options that are not described here. The full schema for all of the configuration capabilities of the Test Writer is specified in swriter.xml in `%ProgramFiles%\Microsoft SDKs\Windows\v7.0\bin\x64\vsstools` (for 64-bit Windows) and `%ProgramFiles%\Microsoft SDKs\Windows\v7.0\bin\vsstools` (for 32-bit Windows). This file contains an XML schema that completely describes all of the elements and attributes that make up a configuration file. Each element and each attribute in this file is commented with a description that documents that element's or attribute's use.

Volume Shadow Copy API Reference

3/5/2021 • 2 minutes to read • Edit Online

The Volume Shadow Copy Service (VSS) is a set of COM and C++ APIs that enable volume backups to be performed while applications on the system (called writers) continue to write to the volumes.

VSS supports these backups through the creation of shadow copies, a duplicate of the original volume at a given point in time.

Third-party developers can use the VSS API to create requesters (such as a backup application) to create and manage shadow copies.

The actual work of creating and representing shadow copies is conducted by system-level applications known as providers.

Developers can also use the API to construct writers: VSS-aware applications that are able to coordinate I/O operations with the creation and manipulation of shadow copies.

- Volume Shadow Copy API Classes
- Volume Shadow Copy API Data Types
- Volume Shadow Copy API Enumerations
- Volume Shadow Copy API Functions
- Volume Shadow Copy API Interfaces
- Volume Shadow Copy API Structures
- Volume Shadow Copy Glossary

For more information on writing requesters and writers, see Using the Volume Shadow Copy Service.

Volume Shadow Copy API Classes

3/5/2021 • 2 minutes to read • Edit Online

The following are the Volume Shadow Copy API public classes:

- CVssWriter
- CVssWriterEx
- CVssWriterEx2

Volume Shadow Copy API Data Types

3/5/2021 • 2 minutes to read • Edit Online

The following data types are defined in the Volume Shadow Copy API:

VSS_ID

```
typedef GUID VSS_ID;
```

The **VSS_ID** definition specifies the general VSS identifier format. The **VSS_ID** is a standard GUID data type.

VSS_IDs are used to identify the following: shadow copies, shadow copy sets, providers, provider versions, writers (writer's class identifier), and instances of writers.

VSS_PWSZ

```
typedef /* [string][unique] */  __RPC_unique_pointer  __RPC_string WCHAR *VSS_PWSZ;
```

The **VSS_PWSZ** definition specifies a null-terminated wide character string (wchar).

VSS_TIMESTAMP

```
typedef LONGLONG VSS_TIMESTAMP;
```

The **VSS_TIMESTAMP** definition holds time-stamp information as a 64-bit integer value containing the number of 100-nanosecond intervals since January 1, 1601 (UTC). This definition is interchangeable with the **FILETIME** structure.

Volume Shadow Copy API Enumerations

3/5/2021 • 2 minutes to read • Edit Online

The following enumerations are defined in the Volume Shadow Copy API.

NAME	DESCRIPTION
VSS_ALTERNATE_WRITER_STATE	Defines the set of valid states used to indicate whether a writer has an associated alternate writer.
VSS_APPLICATION_LEVEL	Defines the set of valid application levels.
VSS_BACKUP_SCHEMA	Defines the set of backup schemas—operations a writer can participate in.
VSS_BACKUP_TYPE	Defines the set of backup types.
VSS_COMPONENT_FLAGS	Indicates support for auto-recovery.
VSS_COMPONENT_TYPE	Defines the set of component types.
VSS_FILE_RESTORE_STATUS	Defines the set of statuses of a file restore operation.
VSS_FILE_SPEC_BACKUP_TYPE	Defines the set of backup operations supported by writers.
_VSS_HARDWARE_OPTIONS	Defines shadow copy LUN flags.
VSS_MGMT_OBJECT_TYPE	Discriminant for the VSS_MGMT_OBJECT_UNION union within the VSS_MGMT_OBJECT_PROP structure.
VSS_OBJECT_TYPE	Used to identify an object as a shadow copy set, shadow copy, or provider.
VSS_PROTECTION_FAULT	Defines the set of shadow copy protection faults.
VSS_PROTECTION_LEVEL	Defines the set of volume shadow copy protection levels.
VSS_PROVIDER_CAPABILITIES	Reserved for future use.
VSS_PROVIDER_TYPE	Defines the set of provider types.
VSS_RECOVERY_OPTIONS	Used by a requester to specify how a resynchronization operation is to be performed.
VSS_RESTORE_TARGET	Defines the set of restore targets.
VSS_RESTORE_TYPE	Defines the set of restore operation to be performed.
VSS_RESTOREMETHOD_ENUM	Defines the set of default file restore methods for a writer.

NAME	DESCRIPTION
VSS_ROLLFORWARD_TYPE	Used by a requester to indicate the type of roll-forward operation it is about to perform.
VSS_SNAPSHOT_COMPATIBILITY	Defines the set of volume control or file I/O operations that can be disabled for a volume that has been shadow copied.
_VSS_SNAPSHOT_CONTEXT	Specifies how a shadow copy is to be created, queried, or deleted and the degree of writer involvement.
VSS_SNAPSHOT_STATE	Defines the set of states of a shadow copy operation.
VSS_SOURCE_TYPE	Defines the set of types of data that a writer can manage.
VSS_SUBSCRIBE_MASK	Defines the set of events that a writer can receive.
VSS_USAGE_TYPE	Specifies how the host system uses the data managed by a writer.
_VSS_VOLUME_SNAPSHOT_ATTRIBUTES	Defines the set of attributes of a shadow copy.
VSS_WRITER_STATE	Defines the set of states of the writer.
VSS_WRITERRESTORE_ENUM	Defines the set of conditions under which a writer will handle events generated during a restore operation.

Volume Shadow Copy API Functions

3/5/2021 • 2 minutes to read • Edit Online

The following are the VSS functions defined in Vssapi.dll and declared in VsBackup.h.

- CreateVssBackupComponents
- CreateVssExamineWriterMetadata
- CreateVssExpressWriter
- CreateWriter
- CreateWriterEx
- IsVolumeSnapshotted
- ShouldBlockRevert
- VssFreeSnapshotProperties

Volume Shadow Copy API Interfaces

3/5/2021 • 2 minutes to read • Edit Online

The Volume Shadow Copy Service (VSS) API provides both COM and C++ interfaces to support the creation of requesters and writers.

VSS Requester-Specific Interfaces

- IVssBackupComponents
- IVssBackupComponentsEx
- IVssBackupComponentsEx2
- IVssBackupComponentsEx3
- IVssExamineWriterMetadata
- IVssExamineWriterMetadataEx
- IVssExamineWriterMetadataEx2
- IVssWMComponent
- IVssWriterComponentsExt

VSS Writer-Specific Interfaces

- IVssCreateExpressWriterMetadata
- IVssCreateWriterMetadata
- IVssCreateWriterMetadataEx
- IVssExpressWriter
- IVssWriterComponents

VSS Hardware Provider-Specific Interfaces

- IVssAdmin
- IVssHardwareSnapshotProvider
- IVssHardwareSnapshotProviderEx
- IVssProviderCreateSnapshotSet
- IVssProviderNotifications

VSS Software Provider-Specific Interfaces

- IVssSoftwareSnapshotProvider

VSS Shared Interfaces

- IVssAsync
- IVssComponent
- IVssComponentEx
- IVssComponentEx2
- IVssEnumObject
- IVssWMDependency
- IVssWMFiledesc

VSS Management Interfaces

- IVssDifferentialSoftwareSnapshotMgmt
- IVssDifferentialSoftwareSnapshotMgmt2
- IVssDifferentialSoftwareSnapshotMgmt3
- IVssEnumMgmtObject
- IVssSnapshotMgmt
- IVssSnapshotMgmt2

Volume Shadow Copy API Structures

3/5/2021 • 2 minutes to read • Edit Online

The following structures are defined in the Volume Shadow Copy API.

STRUCTURE	DESCRIPTION
VSS_COMPONENTINFO	Contains information about a given component.
VSS_DIFF_AREA_PROP	Describes associations between source volumes containing the original file data and volumes containing the shadow copy storage area.
VSS_DIFF_VOLUME_PROP	Describes a shadow copy storage area volume.
VSS_MGMT_OBJECT_PROP	Describes the properties of a volume, shadow copy storage volume, or a shadow copy storage area.
VSS_MGMT_OBJECT_UNION	A union of VSS_VOLUME_PROP, VSS_DIFF_VOLUME_PROP, and VSS_DIFF_AREA_PROP structures used by the VSS_MGMT_OBJECT_PROP structure.
VSS_OBJECT_PROP	Describes the properties of a provider, volume, shadow copy, or shadow copy set.
VSS_OBJECT_UNION	A union of VSS_SNAPSHOT_PROP and VSS_PROVIDER_PROP structures, used by the VSS_OBJECT_PROP structure.
VSS_PROVIDER_PROP	Specifies shadow copy provider properties.
VSS_SNAPSHOT_PROP	Contains the properties of a shadow copy or shadow copy set.
VSS_VOLUME_PROP	Describes the properties of a shadow copy source volume.
VSS_VOLUME_PROTECTION_INFO	Contains information about a volume's shadow copy protection level.

Volume Shadow Copy Glossary

3/5/2021 • 2 minutes to read • Edit Online

The Volume Shadow Copy Service (VSS) API uses its own collection of terms. Many of these terms are familiar to developers, but have new or altered definitions in the VSS API environment. Review the following definitions to understand the VSS API implementation of these terms.

A B C D E F G H I J K *L* M *N O P* Q *R S T* U *V W* X Y Z

A (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

Abort event

A VSS event issued by the Volume Shadow Copy Service indicating that a compliant backup or restore operation has been aborted. The event handler is **CVssWriter::OnAbort**.

Active Directory

Active Directory Service (ADSI) is a COM-based service providing a mechanism for locating, identifying, and accessing the users and resources available in a distributed computing environment system. In particular, the Active Directory service is used to manage enterprise directory information for distribution by a Windows domain server.

alternate location mapping

A path, other than a file's original path, to which the file may be restored under certain conditions. Typically, alternate location mappings are not defined for a single well-defined file, but for a path/file specification pair, and may be recursive.

An alternate location mapping is used only during a restore operation and should not be confused with an alternate path, which is used only during a backup operation. See also *alternate path*.

alternate path

During backup operations, a path/file specification pair (handled by an instance of the **IVssWMFiledesc** interface) is said to have an alternate path if its file descriptor (as returned by **IVssWMFiledesc::GetAlternateLocation**) is non-NULL. It is from this path, rather than the default specified path, that files are to be copied when a volume is backed up.

An alternate path is used only during a backup operation and should not be confused with an alternate location mapping. An alternate location mapping is used only during restore operations. See also *alternate location mapping*.

application level

Used by VSS to indicate the point in the course of the creation of a shadow copy that a writer is notified of a freeze. See also *back-end level applications*, *freeze*, *front-end level applications*, *shadow copy*, *system-level applications*, *writer*.

auto-recovered shadow copy

A shadow copy that has had additional processing by a writer to be in a completely consistent state for backup or data mining actions (for example, rolling back all transactions that did not yet complete at the point that the shadow copy was created.) This can be initiated by a writer by specifying an appropriate flag from the **VSS_COMPONENT_FLAGS** enumeration in the **dwComponentFlags** member of the **VSS_COMPONENTINFO** structure or by a requester by adding the **VSS_VOLSNAP_ATTR_ROLLBACK_RECOVERY** flag to the context for the shadow copy. Auto-recovery allows the shadow copied data to be used on a read-only volume, for data mining, partial restores (for example selected items in a database), or other purposes.

See also *transportable shadow copy*.

auto-release shadow copy

A shadow copy that will be deleted upon the termination of a backup operation. Programmatically, this means the shadow copy will be deleted when the **IVssBackupComponents** interface is released. An auto-release shadow copy cannot be persistent.

By default, all shadow copies are auto-release. See also *persistent shadow copy*, *transportable shadow copy*.

B (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

back-end level applications

Indicates the point at which a writer is notified of a freeze by VSS. Writers that are initialized as back-end level applications are notified after writers initialized as front-end level applications and prior to those initialized as system-level applications. See also *application level*, *front-end level applications*, *system-level applications*, *writer*.

BackupComplete event

A VSS event indicating that a VSS backup has completed. This event should be followed by a BackupShutdown event under normal operation. See also *BackupShutdown event*.

Backup Components Document

An XML document created by a requester (using the **IVssBackupComponents** interface) in the course of setting up a restore or backup operation. The Backup Components Document contains a list of those explicitly included components, from one or more writers, participating in a backup or restore operation. It does not contain implicitly included component information. In contrast, a Writer Metadata Document contains only writer components that may participate in a backup.

A Backup Components Document can be saved to disk. See also *explicit component inclusion*, *implicit component inclusion*.

backup set

Those files to be backed up. It includes files selected by inclusion in a component, those indicated by writer-level include statements, less those files that have been explicitly included.

BackupShutdown event

A VSS event indicating that a compliant backup application has shut down. Normally, this is preceded by a BackupComplete event. However, if a backup terminates unexpectedly, this event can be generated without a preceding BackupComplete event. See also *BackupComplete event*.

backup stamp

A string containing information as to when a backup took place. VSS places no restriction on the format of this string, except that it be intelligible to all the writer instances of a given class. It may contain time and date information, logical sequence numbers, or any other information that will allow a writer of the same class to determine when the last backup has take place.

C (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

certification authority

An entity entrusted to issue certificates asserting that the recipient individual, machine, or organization requesting the certificate fulfills the conditions of an established policy.

client-accessible shadow copy

A shadow copy created by the system provider to support Shadow Copies for Shared Folders and other rollback mechanisms, which allow clients to see old versions of files and undo mistakes without requiring a full restore. A client-accessible shadow copy is created using the **VSS_CTX_CLIENT_ACCESSIBLE** value of the **_VSS_SNAPSHOT_CONTEXT** enumeration. In addition, the VSS_VOLSNAP_ATTR_CLIENT_ACCESSIBLE value of the **_VSS_VOLUME_SNAPSHOT_ATTRIBUTES** enumeration is set automatically for client-accessible shadow copies. See also *Shadow Copies for Shared Folders*.

component

A group of files, defined by a writer, that must be handled as a unit during backup and restore operations. See also *database component*, *file group component*.

component dependency

A situation in which a component (and the component set it defines) managed by one writer cannot be backed up or restore independently of components managed by others writers. A dependency does not indicate an order of preference between the component with the documented dependencies and the components it depends on: a dependency merely indicates that the component and the components it depends on must always be backed up or restored together.

component mode

A mode in which a backup or restore operation makes use of a writer's component information. See also *selectable component*.

component set

A group of components with at least one selectable (for either backup or restore) component and a number of nonselectable components organized in a hierarchy by their logical paths. The implicit participation in a backup or restore operation depends on the explicit inclusion of the top-level selectable component. Only the component information of this selectable component is included in the Backup Components Document. A component set may include selectable and nonselectable subcomponents. See also *logical path*, *selectable component*.

copy-on-write shadow copy

A shadow copy created by saving only the differences from the original volume.

crash consistent state

The state of disks equivalent to what would be found following a catastrophic failure that abruptly shuts down the system. A restore from such a shadow copy set would be equivalent to a reboot following an abrupt shutdown. This is the default state of data that has been shadow copied without the support of writers.

D (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

database component

A group of files used by a database and defined by a writer as needing to be handled as a unit during backup and restore operations. See also *component*, *file group component*.

default provider

See *system provider*.

deleted shadow copy

Deleted shadow copies are not accessible at all and must not be made accessible at any time in the future.

dependency

See *component dependency*.

device object

A string indicating the "root" of a shadow copied volume. Files and directories on a shadow copied volume can be referenced by prepending the device object string to a corresponding path on the original volume. As obtained as the **m_pwszSnapshotDeviceObject** member of the VSS_SNAPSHOT_PROP structure, the device object has no backslash ("\").

diff area

The location on the volume where the *differential data* is stored. This is also known as shadow copy storage space.

differenced files

Files involved in an incremental or differential backup operation implemented by updating entire files (as opposed to modifying sections of those files using partial file support). For instance, if to implement a differential backup an entire file (instead of just the modified section) is copied to a backup media, that file is a differenced file. See also *incremental backup operations*, *differential backup operations*, *partial file support*.

differential backup operations

A backup or restore operation performed only on files created or changed since the last full backup was saved. See also *incremental backup operations*, *partial file support*, *differenced files*.

differential data

Data that can be applied to an original volume to generate a shadow copy volume. This is also commonly known as copy-on-write data, but this documentation uses the term differential data.

directed targeting

A restoration mode by which a writer indicates that when a file is to be restored, it (the source file) should be remapped. The file may be restored to a new restore location and/or ranges of its data restored to different offset with the restore location.

E (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

explicit component inclusion

Adding of a component's information to a requester's Backup Components Document when it participates in a backup or restore operation. Requesters use **IVssBackupComponents::AddComponent** to explicitly include a component in a backup operation, and **IVssBackupComponents::SetSelectedForRestore** or **IVssBackupComponents::AddRestoreSubcomponent** to explicitly include a component in a restore operation.

If any component of a writer is backed up or restored, all nonselectable components without any selectable ancestors must be explicitly included in a backup or restore operation.

Selectable components without selectable ancestors chosen by a requester for participation in a backup or restore must be explicitly included in the operation.

Nonselectable components with a selectable ancestor are never explicitly included.

Selectable components with a selectable ancestor may be included explicitly, or may be included implicitly if the ancestor is included explicitly.

exposed shadow copy

A volume shadow copy that is mounted on a system and available to processes other than the one that manages it. A shadow copy volume mounted under a drive letter or a directory location is referred to as "locally exposed." A shadow copy volume accessible through a share (except for client-accessible shadow copies) is referred to as "remotely exposed." All exposed shadow copies are also surfaced shadow copies. See also *client accessible shadow copy*, *shadow copy*.

F (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

file group component

A group of files other than those used as a database and defined by a writer as needing to be handled as a unit during backup and restore operations. See also *component*, *database component*.

file set

The combination of a path, file specification, and recursive flag describing one of a file or group of files. For example, files are added to components in file sets.

Unless otherwise indicated, a file set's paths are standard Windows paths and can contain environment variables (for example, %SystemRoot%) but cannot contain wildcard characters. There is no requirement that the path end with a backslash ("\"). It is up to applications that retrieve this information to check it.

The file specification contained in a file set indicates the name of the file or files it includes. This file specification cannot contain directory specifications (for example, no backslashes) but can contain the ? and * wildcard characters.

The recursive tag is a Boolean specifying whether the file set's path should be treated as a only a single directory or if it indicates a hierarchy of directories to be traversed recursively. The Boolean is **true** if the path is treated as a hierarchy of directories to be traversed recursively, or **false** if not.

Information about a file set is returned through instances of the **IVssWMFiledesc** interface, and can contain additional information includes alternate paths, alternate location mappings, and file-level schema support settings.

See also *alternate path*, *alternate location mapping*, *component*, *file specification*.

file specification

Used to match files within a directory and to define a file set. It cannot contain directory specifications (for example, no backslashes) but it can contain the ? and * wildcard characters.

File Replication Service

Used to replicate system files in a redundant system volume (SysVol) directory to support file system survivability, particularly for distributed file systems.

freeze

A period of time during the shadow copy creation process when all writers have flushed their writes to the volumes and are not initiating additional writes.

Freeze event

A VSS event indicating that a shadow copy freeze is in progress. See also *freeze*, *shadow copy*.

front-end level applications

Indicates the point at which a writer is notified of a freeze by VSS. Writers that were initialized as front-end level applications are notified prior to writers initialized either as back-end level applications or as system-level applications. See also *application level*, *back-end level applications*, *system-level applications*, *writer*.

G (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

GUID

Globally unique identifier. A 16-byte data structure you can use to identify objects in a globally unique way.

H (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

hardware provider

A provider that manages shadow copies at the hardware level by working in conjunction with a hardware storage adapter or controller. See also *provider*, *software provider*.

hidden shadow copy

A hidden volume, which is a volume that is not mounted and not exposed in the Windows name space as a drive letter or mounted folder.

I (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

Identify event

A VSS event indicating that a writer or a requester needs to query the current writer. It is used to construct the current writer's metadata information. See also *requester*, *writer*.

implicit component inclusion

Not adding a component's information to a requester's Backup Components Document when it participates in a backup or restore operation.

Nonselectable components with a selectable ancestor are only implicitly included if their ancestor is included.

Selectable components with a selectable ancestor may be included implicitly, if the ancestor is included, or may be included explicitly.

Nonselectable components without any selectable ancestors are never implicitly included in a backup or restore operation—all such components must be explicitly added to the Backup Components Document if any of the writer's components participate.

Selectable components without selectable ancestors chosen by a requester for participation in a backup or restore cannot be implicitly included in the operation, but must be explicitly added to the Backup Components Document.

incremental backup operations

A backup or restore operation is performed only on files created or changed since the last full, incremental, or differential backup was saved. See also *differential backup operations*.

L (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

logical path

A mechanism for describing a writer's components. It is used to express relationships between components, in particularly their hierarchy. For instance, if a writer managed several electronic books, it might assign logical paths of "\ebook\book1" and "\ebook\book2" to file group components containing each book. Logical paths are required when specifying subcomponents.

By convention the backslash "\" is used to separate elements of a logical path. Beyond that, there are no restrictions on the characters that can appear in a non-**NULL** logical path.

A logical path can be **NULL**. By convention, components with **NULL** logical paths are said to be at the top level of a writer's logical path hierarchy.

See also *component*, *subcomponent*.

N (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

new target location

A path that only a requester can specify if it requires a file to be restored to a new location. This should not be confused with alternate location mapping. See also *alternate location mapping*.

O (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

original volume

The volume from which a shadow copy is derived.

P (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

partial file support

The capacity to implement backup operations by modifying subsections of the files involved, rather than working with the entire files. See also *directed targeting*.

persistent shadow copy

A shadow copy that is not deleted automatically when the requester releases an **IVssBackupComponents** object or when the computer is restarted. See also *auto-release shadow copy*, *shadow copy*.

Plex

A special type of shadow copy volume that fully and completely represents a shadow copy volume without the need for either original volume. This is also commonly known as a split-mirror, but this documentation uses the term Plex.

PostRestore event

A VSS event indicating that a VSS restore has completed.

PostSnapshot event

A VSS event indicating that a shadow copy has been completed. See also *shadow copy*.

PrepareForBackup event

A VSS event indicating that a backup operation is about to take place.

PrepareForSnapshot event

A VSS event indicating that writers should take steps to prepare for an upcoming shadow copy operation. See also *shadow copy*.

PreRestore event

A VSS event indicating that a restore operation is about to take place.

provider

An operating system object that intercepts and manages I/O requests to create and represent volume shadow copies to the file system. See also *hardware provider*, *software provider*.

R (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

requester

Minimally, any process that interacts with VSS to create and manage shadow copies and shadow copy sets of one or more volumes.

Typically, a requester manages shadow copies to support some other functionality, such as backup and restore operations and disk mirroring. See also *shadow copy*, *shadow copy set*.

restore method

A specification of the restore process method. It controls such issues as whether files are overwritten on restore. If a restore method setting is in conflict with those of the restore target, then the restore target takes precedence. See also *restore target*.

restore target

Under VSS, a restore target is a specification of how a requester should restore a file, for example, if it should overwrite existing files.

S (Volume Shadow Copy Service)

3/5/2021 • 3 minutes to read • Edit Online

selectability for backup

A component is said to be selectable for backup if its explicit inclusion in a backup operation is optional. Selectability for backup is enabled if the value of the **bSelectable** member of the **VSS_COMPONENTINFO** structure is **true**. There is no default value for a component's selectability for backup; a writer must always set this value explicitly.

Writers also use selectability for restore to organize their component's participation in restore operations.

See also *selectable component*, *selectability for restore*, *explicit component inclusion*, *implicit component inclusion*.

selectability for restore

A component is said to be selectable for restore if it can be implicitly backed up as part of a component set, and then later individually restored without the rest of the component set. Selectability for restore is enabled if the value of the **bSelectableForRestore** member of the **VSS_COMPONENTINFO** structure is **true**. By default, a component's selectability for restore is **false**. Selectability for restore has meaning only when a component has not been added to the backup document.

See also *selectable component*, *selectability for backup*, *explicit component inclusion*, *implicit component inclusion*.

selectable component

A component is said to be selectable if its explicit inclusion in a backup or restore operation is optional.

Selectability for backup and selectability for restore are independent of each other—a component may be selectable for both operations, for restore and not backup, or for backup and not restore.

Writers also use selectability to organize their components into groups:

- Nonselectable components with no selectable ancestors in their logical paths must always be explicitly included if a writer is to be backed up or restored.
- Nonselectable components with selectable ancestors will only be included in a backup or restore if at least one selectable ancestor is included.
- Selectable components without selectable ancestors can only be included in a backup or restore operation explicitly.
- Selectable components with selectable ancestors can be explicitly included in a backup or restore operation, or can be implicitly included if one of their selectable ancestors is included.

See also *components*, *selectability for backup*, *selectability for restore*, *explicit component inclusion*, *implicit component inclusion*.

shadow copy

A read-only point-in-time replica of an original volume's contents. Each shadow copy is keyed by a persistent GUID. Also called a volume shadow copy. See also *shadow copy set*.

Shadow Copies for Shared Folders

A feature of Windows that creates lightweight, online backup copies of volumes using VSS. Clients can access these backups through the Windows shell to see old versions of files and undo mistakes without requiring a full restore. See also *client accessible shadow copy*.

shadow copy set

A collection of volume shadow copies created at the same time and identified by a common shadow copy set ID (a persistent GUID) value. This is the mechanism used to allow a shadow copy operation to be managed across volumes. See also *shadow copy*.

software provider

A provider that intercepts I/O requests at the software level between the file system and the volume manager. See also *hardware provider*, *provider*.

subcomponent

Any component that has a logical path containing a selectable (for either backup or restore) parent component. Subcomponents must have a logical path of the form {component_name}\{subcomponent_name}. If a subcomponent's selectable (for either backup or restore) ancestor is explicitly included in a backup or restore, then the subcomponent is implicitly included in the operation. Implicitly included subcomponents do not have their data included in the Backup Components Document—regardless of whether they are selectable (for either restore or backup) or not. See also *component*, *logical path*.

surfaced shadow copy

A shadow copied volume visible to a system's Mount Manager namespace—meaning **FindFirstVolume** and **FindNextVolume** can find it and that volume arrival and departure notifications are generated. All exposed shadow copies are also surfaced shadow copies. However, a surfaced shadow copy need not be exposed. If a shadow copy is transportable, it cannot be surfaced. See also *exposed shadow copy*, *transportable shadow copy*.

System File Protection

See *Windows File Protection*.

system-level applications

Indicates the point at which VSS notifies a writer of a freeze. Writers that are initialized as system-level applications are notified after writers initialized either as front-end level applications or as back-end level applications. See also *application level*, *back-end level applications*, *front-end level applications*.

system provider

The default preinstalled provider provided as part of Windows.

System Volume Folder

A shared directory that stores the shared copies of a domain's public files—that is, those files that are replicated among all domain controllers in the domain.

T (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

Thaw event

A VSS event issued by VSS indicating that a shadow copy freeze has completed. It is used to remove a writer's preparations for a freeze. See also *freeze*, *shadow copy*.

transportable shadow copy

A shadow copy that can be moved from one system to another. Typically, a transportable shadow copy is not surfaced locally. See also *surfaced shadow copy*.

V (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

volume

A logical storage unit. A volume is the target of shadow copy creation operations.

volume GUID name

A unique name for a volume. A volume GUID name is a string of the following form:

\\?\Volume{GUID}\

(note the trailing "\") where GUID is a globally unique identifier for the volume. The operating system assigns a volume name when it first encounters a volume, for example, during formatting or installation.

Note that a volume can have more than one volume GUID name.

W (Volume Shadow Copy Service)

3/5/2021 • 2 minutes to read • Edit Online

Windows File Protection

A system service that protects special operating system files. If one of these files is deleted or overwritten, Windows File Protection replaces the file with the original from its cache.

writer

An application that coordinates its I/O operations with VSS shadow copy and shadow copy related operations (such as backups and restores) so that their data contained on the shadow copied volume is in a consistent state.

To support this coordination, a writer must implement an instance of a class derived from the abstract base class **CVssWriter** to communicate with the VSS infrastructure. See also *shadow copy*.

writer class

A globally unique identifier (GUID) supplied by a writer during initialization to indicate that it belongs to a given type of writers. For instance, multiple implementations of a writer could share the same writer class ID.

Writers of the same class may be distinguished by their writer instances. It is up to an application developer to specify writer classes. See also *writer instance*.

writer instance

A globally unique identifier (GUID) supplied by VSS for each writer process running on a system. A unique value for the writer instance is generated each time the writer starts.

Writer Metadata Document

An XML document created by a writer (using the **IVssCreateWriterMetadata** interface) containing information about the writer's state and components. A requester can query Writer Metadata Documents (using the **IVssExamineWriterMetadata** interface) when performing a restore or backup operation.

A Writer Metadata Document contains the list of all of a writer's components, any one of which might participate in a backup. This differs from the requester's Backup Components Document, which contains only those components explicitly included for a backup or restore operation.

Once constructed, the Writer Metadata Document should be viewed as a read-only object. It can be saved to disk. See also *Backup Components Document*, *explicit component inclusion*, *implicit component inclusion*.