

The Disk Management

What do we have in this session?

Introduction

Disk Devices and Partitions

Basic and Dynamic Disks

Basic Disks

Dynamic Disks

Partition Styles

Master Boot Record

GUID Partition Table

Detecting the Type of Disk

Defining an MS-DOS Device Name

Managing Disk Quotas

User-level Administration of Disk Quotas

System-level Administration of Disk Quotas

Disk Quota Limits

Disk Quota Interfaces

Disk Management Control Codes

Disk Management Enumeration Types

MEDIA_TYPE Enumeration Definition

Constants

PARTITION_STYLE Enumeration Definition

Constants

Disk Management Functions

CreateFile() Function

Parameters

Return Value

Symbolic Link Behavior

Caching Behavior

Files

Synchronous and Asynchronous I/O Handles

File Streams

Directories

Physical Disks and Volumes

Changer Device

Tape Drives

Communications Resources

Consoles

Mailslots

Pipes

Opening a File for Reading or Writing

Example: Open a File for Writing

Example: Open a File for Reading

How Do I Get the Security tab in Folder Properties?

Calling DeviceIoControl() Program Example

DeleteFile() Function

Parameters

Return Value

Remarks

Symbolic link behavior

Deleting a File Program Example

Another Deleting File Program Example

GetDiskFreeSpace() Function

Parameters

Return Value

Remarks

GetDiskFreeSpaceEx() Function Example

Parameters

Return Value

Remarks

Notes on 64-bit Integer Math

Disk Management Interfaces

Disk Management Structures

Disk Partition Types

Introduction

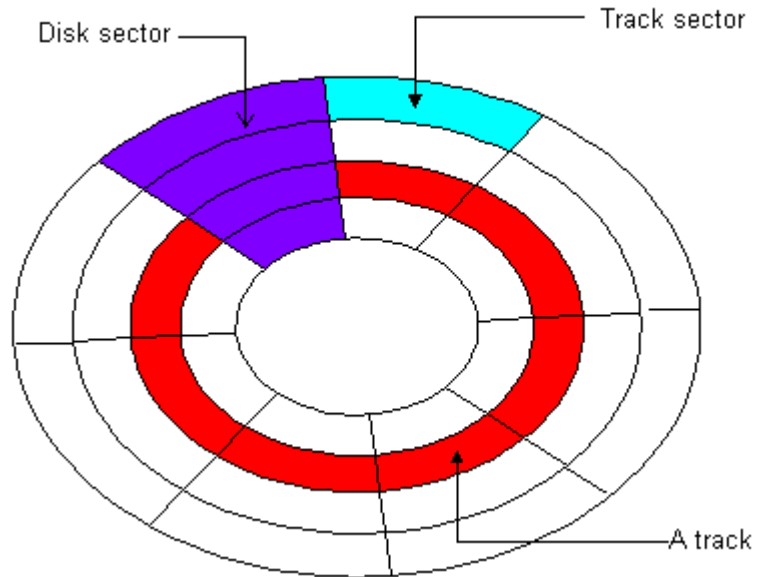
A hard disk is a rigid disk inside a computer that stores and provides relatively quick access to large amounts of data. It is the type of storage most often used with Windows. The system also supports removable media.

The file system provides an abstraction of the physical characteristics of storage devices so that applications can simply write to and read from files. However, storage in an enterprise relies heavily on the concept of disks.

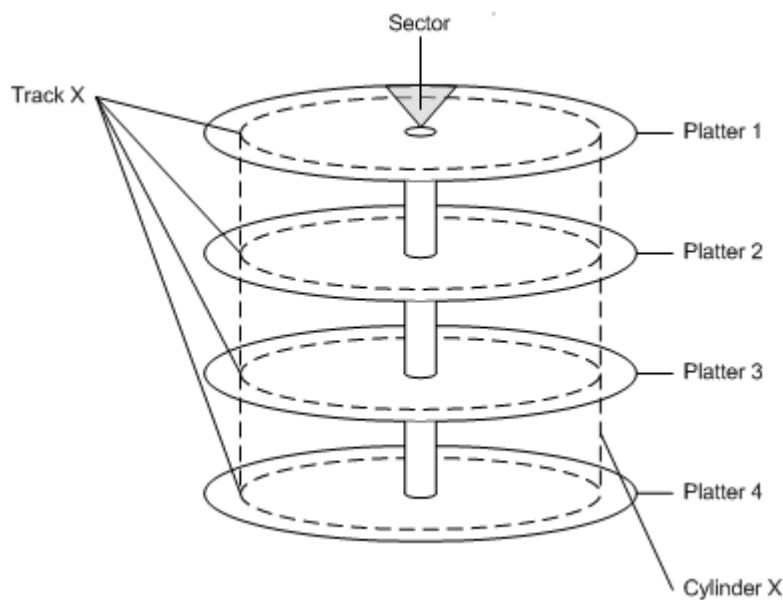
Disk Devices and Partitions

Basically, a hard disk consists of a set of **stacked platters**, each of which has data stored electromagnetically in **concentric circles**, or **tracks**. Each platter has **two heads**, one on each side of the platter that reads or writes data as the disk spins. A hard disk drive controls the positioning, reading, and writing of the hard disk. Note that the heads of all platters are positioned as a unit.

The smallest addressable unit of a track is a sector. A disk sector is a wedge-shaped piece of the disk. Each sector is numbered and a track sector is the area of intersection of a track and a sector. A cluster is a set of track sectors, ranging from 2 to 32 or more, depending on the formatting scheme in use.



A cylinder is defined as the set of tracks that appear in the same location on each platter or a set of matched tracks. For example, the following diagram shows a hard disk with four platters. Cylinder X consists of eight tracks (track X from each side of each platter).



A hard disk can contain one or more **logical regions** called **partitions**. Partitions are created when the user formats a hard disk as a basic disk. Windows also supports **dynamic disks**, which are not discussed in this topic.

The creation of multiple partitions on a drive allows the appearance of having separate hard drives. For example, a system with one hard disk that has one partition contains a single volume, designated by the system as drive C. A system with a hard disk with two partitions typically contains drives C and D. Having multiple partitions on a hard disk can make it easier to manage the system, for example to organize files or to support multiple users. The **first physical sector on a basic disk contains a data structure known as the Master Boot Record (MBR)**. The MBR contains the following:

1. A boot program (up to 442 bytes in size)
2. A disk signature (a unique 4-byte number)
3. A partition table (up to four entries)
4. An end-of-MBR marker (always **0x55AA**)

In the latest development, [Solid State Drive \(SSD\)](#) seems promising to replace the 'mechanical' type HDD in near future.

Basic and Dynamic Disks

Before partitioning a drive or getting information about the partition layout of a drive, you must first understand the features and limitations of basic and dynamic disk storage types.

For the purposes of this topic, the term volume is used to refer to the concept of a disk partition formatted with a valid file system, most commonly NTFS, which is used by the Windows operating system to store files. A volume has a Win32 path name, can be enumerated by the FindFirstVolume() and FindNextVolume() functions, and usually has a drive letter assigned to it, such as C:.

There are two types of disks when referring to storage types in this context: basic disks and dynamic disks. Note that the storage types discussed here are not the same as physical disks or partition styles, which are related but separate concepts. For example, referring to a basic disk does not imply a particular partition style, the partition style used for the disk under discussion would also need to be specified.

Basic Disks

Basic disks are the storage types most often used with Windows. The term basic disk refers to a disk that contains partitions, such as primary partitions and logical drives, and these in turn are usually formatted with a file system to become a volume for file storage. Basic disks provide a simple storage solution that can accommodate a useful array of changing storage requirement scenarios. Basic disks also support clustered disks, Institute of Electrical and Electronics Engineers (IEEE) 1394 disks, and universal serial bus (USB) removable drives. For backward compatibility, basic disks usually use the same Master Boot Record (MBR) partition style as the

disks used by the Microsoft MS-DOS operating system and all versions of Windows but can also support GUID Partition Table (GPT) partitions on systems that support it.

You can add more space to existing primary partitions and logical drives by extending them into adjacent, contiguous unallocated space on the same disk. To extend a basic volume, it must be formatted with the NTFS file system. You can extend a logical drive within contiguous free space in the extended partition that contains it. If you extend a logical drive beyond the free space available in the extended partition, the extended partition grows to contain the logical drive as long as the extended partition is followed by contiguous unallocated space. The following operations can be performed only on basic disks:

1. Create and delete primary and extended partitions.
2. Create and delete logical drives within an extended partition.
3. Format a partition and mark it as active.

Dynamic Disks

Dynamic disks were first introduced with Windows 2000 and provide features that basic disks do not, such as the ability to create volumes that span multiple disks (spanned and striped volumes) and the ability to create fault-tolerant volumes (mirrored and RAID-5 volumes). Like basic disks, dynamic disks can use the MBR or GPT partition styles on systems that support both. All volumes on dynamic disks are known as dynamic volumes. Dynamic disks offer greater flexibility for volume management because they use a database to track information about dynamic volumes on the disk and about other dynamic disks in the computer. Because each dynamic disk in a computer stores a replica of the dynamic disk database, for example, a corrupted dynamic disk database can repair one dynamic disk by using the database on another dynamic disk. The location of the database is determined by the partition style of the disk. On MBR partitions, the database is contained in the last 1 megabyte (MB) of the disk. On GPT partitions, the database is contained in a 1-MB reserved (hidden) partition.

Dynamic disks are a separate form of volume management that allows volumes to have noncontiguous extents on one or more physical disks. Dynamic disks and volumes rely on the Logical Disk Manager (LDM) and Virtual Disk Service (VDS) and their associated components. These components enable you to perform tasks such as converting basic disks into dynamic disks, and creating fault-tolerant volumes. To encourage the use of dynamic disks, multi-partition volume support was removed from basic disks, and is now exclusively supported on dynamic disks. The following operations can be performed only on dynamic disks:

1. Create and delete simple, spanned, striped, mirrored, and RAID-5 volumes.
2. Extend a simple or spanned volume.
3. Remove a mirror from a mirrored volume or break the mirrored volume into two volumes.
4. Repair mirrored or RAID-5 volumes.
5. Reactivate a missing or offline disk.

Another difference between basic and dynamic disks is that dynamic disk volumes can be composed of a set of noncontiguous extents on one or multiple physical disks. By contrast, a volume on a basic disk consists of one set of contiguous extents on a single disk. Because of the

location and size of the disk space needed by the LDM database, Windows cannot convert a basic disk to a dynamic disk unless there is at least 1MB of unused space on the disk. Regardless of whether the dynamic disks on a system use the MBR or GPT partition style, you can create up to 2,000 dynamic volumes on a system, although the recommended number of dynamic volumes is 32 or less. The operations common to basic and dynamic disks are the following:

1. Support both MBR and GPT partition styles.
2. Check disk properties, such as capacity, available free space, and current status.
3. View partition properties, such as offset, length, type, and if the partition can be used as the system volume at boot.
4. View volume properties, such as size, drive-letter assignment, label, type, Win32 path name, partition type, and file system.
5. Establish drive-letter assignments for disk volumes or partitions, and for CD-ROM devices.
6. Convert a basic disk to a dynamic disk, or a dynamic disk to a basic disk.

Unless specified otherwise, Windows initially partitions a drive as a basic disk by default. You must explicitly convert a basic disk to a dynamic disk. However, there are disk space considerations that must be accounted for before you attempt to do this.

Partition Styles

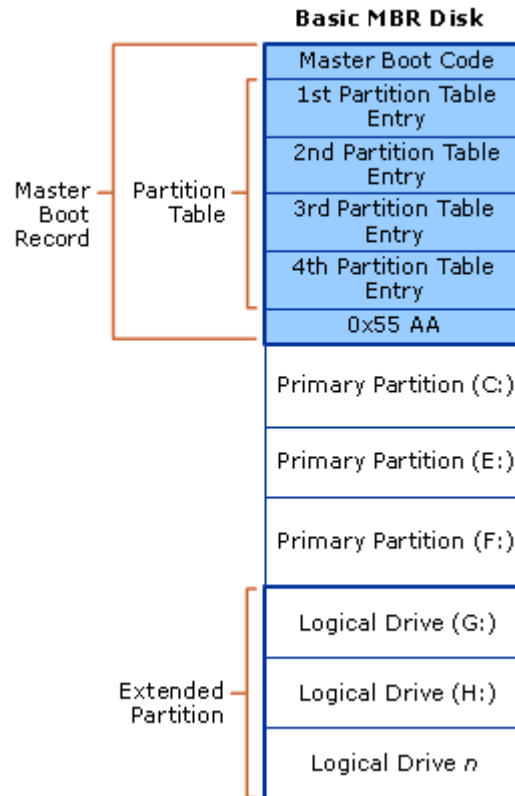
Partition styles, also sometimes called partition schemes, is a term that refers to the particular underlying structure of the disk layout and how the partitioning is actually arranged, what the capabilities are, and also what the limitations are. To boot Windows, the BIOS implementations in x86-based and x64-based computers require a basic disk that must contain at **least one master boot record (MBR) partition marked as active where information about the Windows operating system** (but not necessarily the entire operating system installation) and where information about the partitions on the disk are stored. This information is placed in separate places, and these **two places may be located in separate partitions or in a single partition**. All other physical disk storage can be set up as various combinations of the two available partition styles, described in the following sections.

Dynamic disks follow slightly different usage scenarios, as previously outlined, and the way they utilize the two partition styles is affected by that usage. Because dynamic disks are not generally used to contain system boot volumes, this discussion is simplified to exclude special-case scenarios.

Master Boot Record

All x86-based and x64-based computers running Windows can use the partition style known as **master boot record (MBR)**. The MBR partition style contains a **partition table** that describes where the partitions are located on the disk. Because MBR is the only partition style available on x86-based computers prior to Windows Server 2003 with Service Pack 1 (SP1), you do not need to choose this style. It is used automatically.

You can create up to **four partitions** on a basic disk using the MBR partition scheme: either **four primary partitions**, or **three primary and one extended**. The extended partition can contain one or more **logical drives**. The following figure illustrates an example layout of three primary partitions and one extended partition on a basic disk using MBR. The extended partition contains four extended logical drives within it. The extended partition may or may not be located at the end of the disk, but it is always a single contiguous space for logical drives 1-n.



Each partition, whether primary or extended, can be formatted to be a **Windows volume**, with a one-to-one correlation of volume-to-partition. In other words, a single partition cannot contain more than a single volume. In this example, there would be a **total of seven volumes** available to Windows for file storage. An unformatted partition is not available for file storage in Windows. The dynamic disk MBR layout looks very similar to the basic disk MBR layout, except **that only one primary partition is allowed** (referred to as the LDM partition), **no extended partitioning is allowed**, and there is a **hidden partition at the end of the disk for the LDM database**.

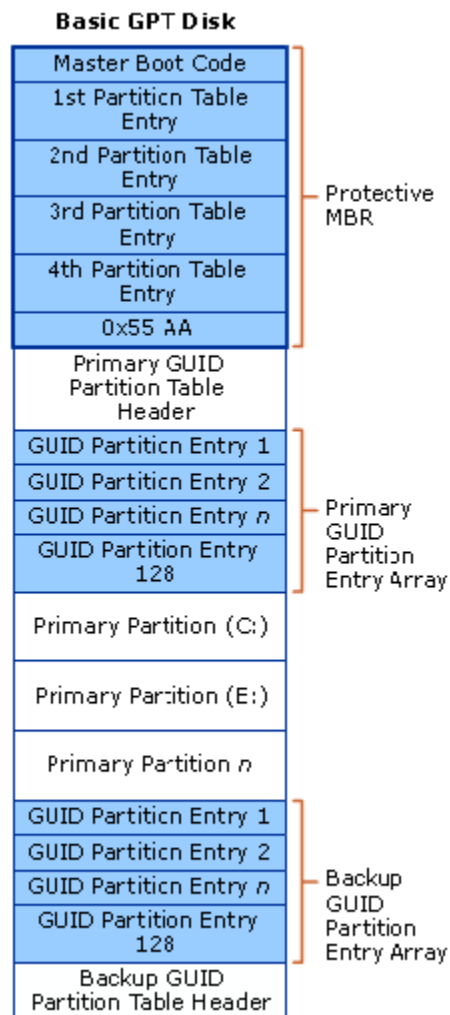
GUID Partition Table

Systems running Windows Server 2003 with SP1 and later can use a partition style known as the **globally unique identifier (GUID) partition table (GPT)** in addition to the MBR partition style. A basic disk using the GPT partition style can have up to **128 primary partitions**, while dynamic disks will have a single LDM partition as with MBR partitioning. Because basic disks

using GPT partitioning do not limit you to **four partitions**, you do not need to create extended partitions or logical drives. The GPT partition style **also** has the following properties:

1. Allows partitions larger than **2 terabytes**.
2. Added reliability from replication and cyclic redundancy check (CRC) protection of the partition table.
3. Support for additional partition type GUIDs defined by original equipment manufacturers (OEMs), independent software vendors (ISVs), and other operating systems.

The GPT partitioning layout for a basic disk is illustrated in the following figure.



The protective MBR area exists on a GPT partition layout for backward compatibility with disk management utilities that operate on MBR. The GPT header defines the range of logical block addresses that are usable by partition entries. The GPT header also defines its location on the disk, its GUID, and a 32-bit cyclic redundancy check (CRC32) checksum that is used to verify the integrity of the GPT header. Each GUID partition entry begins with a partition type GUID.

The 16-byte partition type GUID, which is similar to a System ID in the partition table of an MBR disk, identifies the type of data that the partition contains and identifies how the partition is used, for example if it is a basic disk or a dynamic disk. Note that each GUID partition entry has a backup copy.

Dynamic disk GPT partition layouts looks similar to this basic disk example, but as stated previously have only one LDM partition entry rather than 1-n primary partitions as allowed on basic disks. There is also a hidden LDM database partition with a corresponding GUID partition entry for it.

Detecting the Type of Disk

There is no specific function to programmatically detect the type of disk a particular file or directory is located on. There is an indirect method.

First, call `GetVolumePathName()`. Then, call `CreateFile()` to open the volume using the path. Next, use `IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS` with the volume handle to obtain the disk number and use the disk number to construct the disk path, such as `"\\?\PhysicalDriveX"`. Finally, use `IOCTL_DISK_GET_DRIVE_LAYOUT_EX` to obtain the partition list, and check the **PartitionType** for each entry in the partition list.

Defining an MS-DOS Device Name

An MS-DOS device name is a junction that points to the path of an MS-DOS device. These junctions comprise the MS-DOS device namespace. Call the `DefineDosDevice()` and `SetVolumeMountPoint()` functions to create and modify these junctions.

`DeleteVolumeMountPoint()` deletes a junction created by `SetVolumeMountPoint()`, and `DefineDosDevice()` deletes junctions it creates.

After an MS-DOS device name is defined, it remains visible to all processes. Before Windows 2000 Professional with Service Pack 2 (SP2), a device name remained globally visible until either explicitly removed or the system restarted. Security issues made changes to this policy necessary in later versions of Windows.

Starting with Windows 2000 Professional with SP2, when a user that is logged in through an interactive console session (that is, by running a console application on a local machine), any drive letters defined by running a program that calls `DefineDosDevice()` are deleted when the interactive console user logs out. Also, a security policy is implemented regulating the circumstances under which drive letters can be deleted. This deletion policy is defined as follows:

1. If the user attempting the deletion is logged in as an Administrator, or another account that belongs to the Administrator group, the user can delete any drive letter.
2. If the user attempting the deletion is not logged in as an Administrator, and is logged in through the interactive console session, the user can delete any drive letter except those created by other users logged in as Administrators through logon sessions that are not the interactive console session, such as background scheduled tasks or during system startup.
3. If the user attempting the deletion is not logged in as an Administrator, and is logged in through a logon session that is not the interactive console session, the user can delete only the drive letters that he or she has created during the session.

This security fix does not affect Terminal Services sessions, because each Terminal Services session defines its own MS-DOS Device namespace. In Windows 2000 Professional with Service Pack 2 (SP2) and earlier, calls to QueryDosDevice() return all MS-DOS devices that have been defined on the local machine. In Windows XP, the policy changes defined in Windows 2000 Professional with SP2 were removed and replaced with an architecture based on the following:

1. All MS-DOS devices are identified by Windows through an **authentication ID**. An authentication ID is the LUID (**locally unique identifier**) associated with each logon session when created.
2. The visibility of an MS-DOS device name is categorized as either **global** or **local**, and is defined as such by its inclusion in the **Global MS-DOS Device** and **Local MS-DOS Device** namespaces. The contents of MS-DOS devices in the Global namespace can be accessed by all users, and the contents of MS-DOS devices in the Local namespace can be accessed only by the user whose **access token** contains the AuthenticationID associated with that Local MS-DOS device namespace.

Multiple Local MS-DOS Device namespaces and only one Global MS-DOS Device namespace may exist at one time and on one machine.

Note that only processes running in the LocalSystem context can call DefineDosDevice() to create an MS-DOS device in the Global MS-DOS device namespace. Also, the Local MS-DOS device namespace corresponding to a specific AuthenticationID is deleted when the last reference to that AuthenticationID is removed.

When your code queries an existing MS-DOS device name by calling QueryDosDevice(), it first searches the Local MS-DOS Device namespace. If it is not found there, the function will then search the Global MS-DOS Device namespace. When your code queries all existing MS-DOS device names through this function, the list of names that are returned is dependent on whether it is running in the LocalSystem context. If so, only the MS-DOS device names included in the Global MS-DOS Device namespace will be returned. If not, a concatenation of the device names in the Global and Local MS-DOS Device namespaces will be returned. If a device name exists in both namespaces, QueryDosDevice() will return the entry in the Local MS-DOS Device namespace. This also applies to the list of all MS-DOS device names returned by GetLogicalDrives() and GetLogicalDriveStrings(). Note that the following scenario may occur:

1. User A, who is not running within the LocalSystem context, creates a device name in the corresponding Local MS-DOS Device namespace, and that device name does not exist in the Global MS-DOS Device namespace.
2. User B, who is running within the LocalSystem context, creates the same device name in the Global MS-DOS Device namespace.

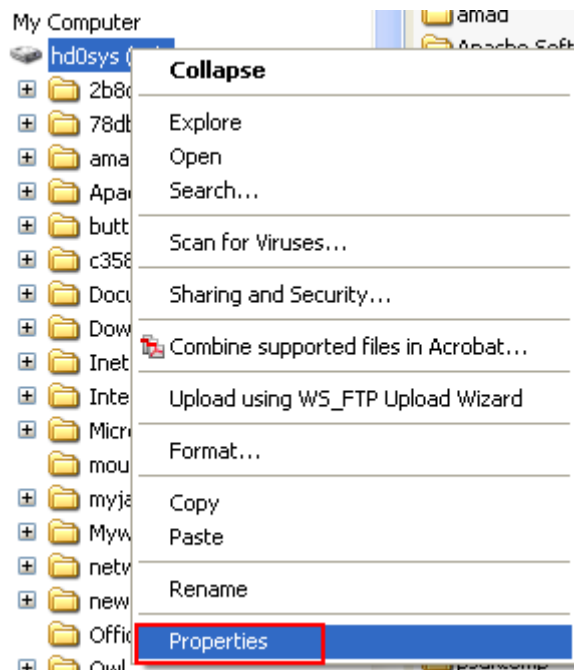
In this scenario, User A will not have access to the device name in the Global MS-DOS Device namespace until he or she removes or renames the device name in his or her Local MS-DOS Device namespace. To reduce the likelihood of this scenario occurring, MS-DOS drive letters should be allocated in the Global MS-DOS Device namespace starting with C: and ending with

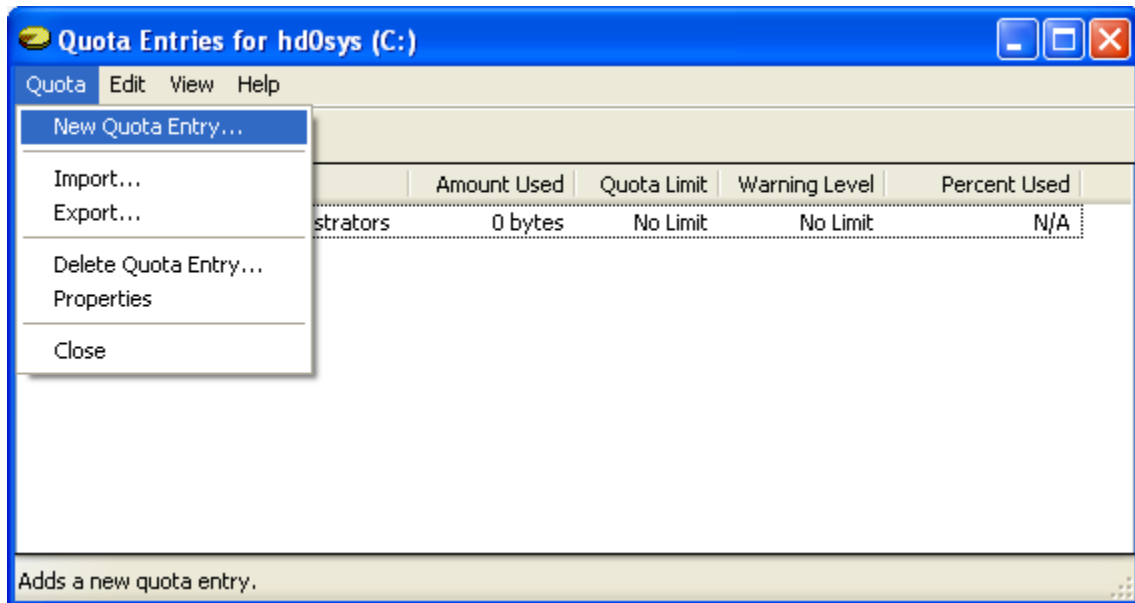
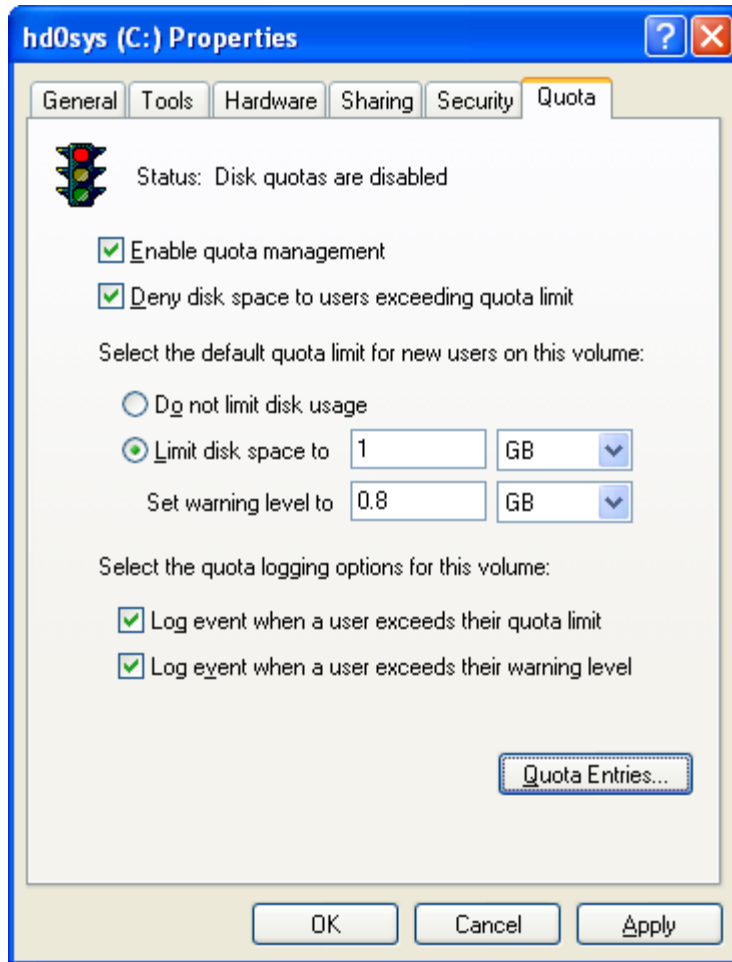
Z:. This sequence should be reversed for the allocation of MS-DOS drive letters in the Local MS-DOS Device namespace.

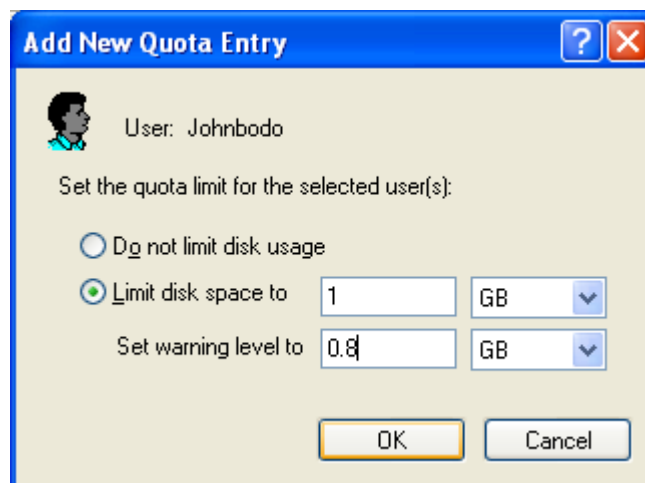
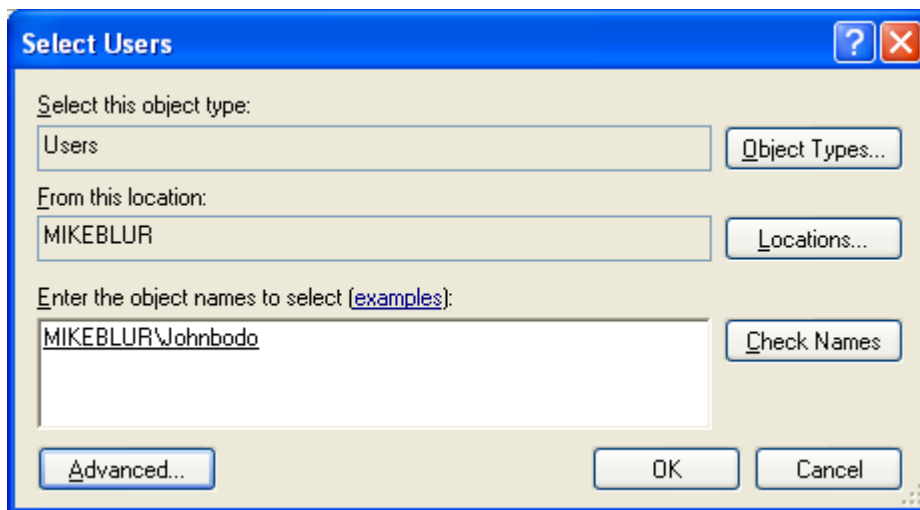
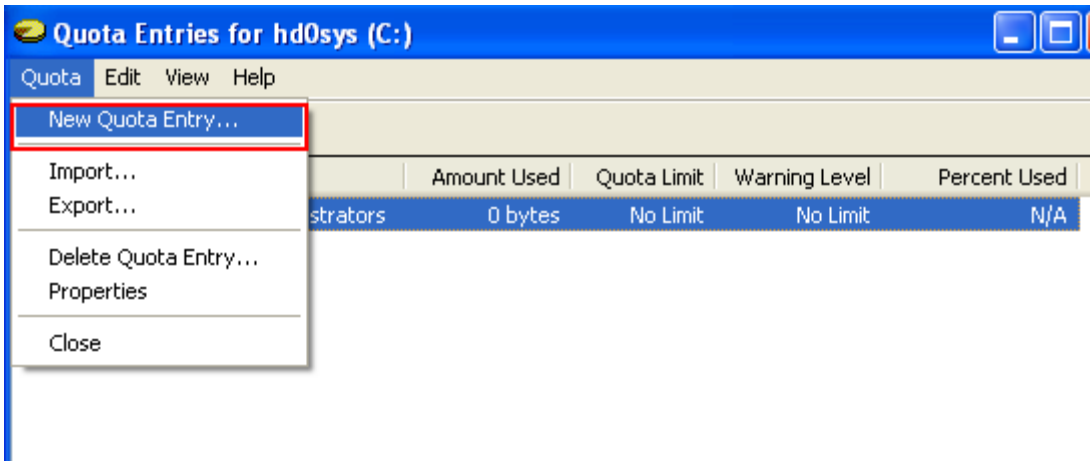
If you are not running within the LocalSystem context, DefineDosDevice() will not allow you to define a device name in the Local MS-DOS Device namespace if that device name already exists in your Local or Global MS-DOS Device namespaces. Call QueryDosDevice() before calling DefineDosDevice() to determine whether the device name you intend to define exists in your MS-DOS Device namespaces.

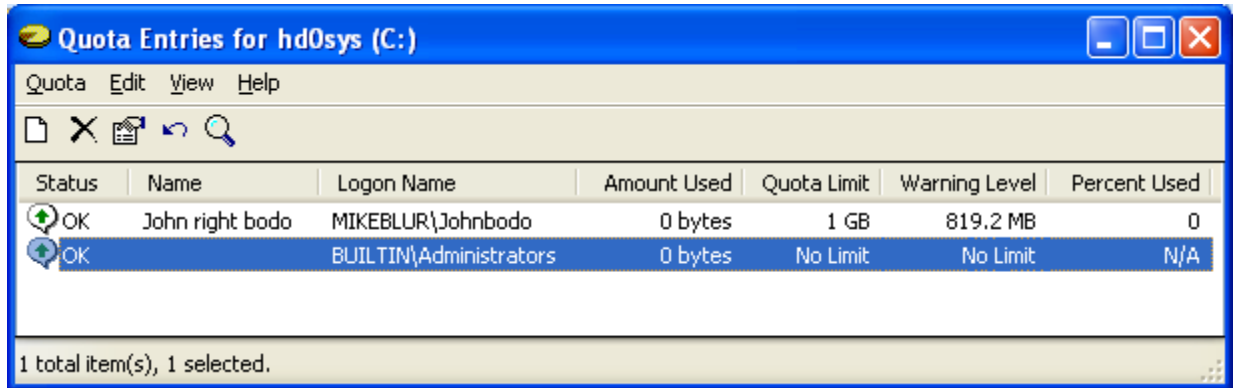
Managing Disk Quotas

The NTFS file system supports **disk quotas**, which allow administrators to control the amount of data that each user can store on an NTFS file system volume. Administrators can optionally configure the system to log an event when users are near their quota, and to deny further disk space to users who exceed their quota. Administrators can also generate reports, and use the event monitor to track quota issues. You can determine whether a file system supports disk quotas by calling the GetVolumeInformation() function and examining the FILE_VOLUME_QUOTAS bit flag.









User-level Administration of Disk Quotas

Disk quotas are transparent to the user. When a user asks how much space is free on a disk, the system reports only the available quota allowance the user has available. If the user exceeds this allowance, the system returns the `ERROR_DISK_FULL` error, just as it would to indicate that the disk was full. To obtain more free disk space after exceeding the quota allowance, the user must do one of the following:

1. Delete some files.
2. Have another user claim ownership of some files.
3. Have the administrator increase the quota allowance.

Programs that need to retrieve the actual amount of free disk space can call the `GetDiskFreeSpaceEx()` function and look at the `TotalNumberOfFreeBytes` parameter.

System-level Administration of Disk Quotas

The system administrator can set quotas for specific users on a volume. The administrator can also set default quotas for the volume. A new user on the volume receives the default quota unless the administrator established a quota specifically for that user.

The administrator can query the level of quota tracking and enforcement (or quota states), the default quota limits, and the per-user quota information. The per-user quota information contains the user's hard quota limit, warning threshold, and the quota usage. The administrator can also enable or disable quota enforcement. There are three quota states, as shown in the following table.

State	Description
Quota disabled	Quota usage changes are not tracked, but the quota limits are not removed. In this state, performance is not affected by disk quotas. This is the default state.
Quota tracked	Quota usage changes are tracked, but quota limits are not enforced. In this state, no quota violation events are generated and no file operations fail because of disk quota violations.
Quota enforced	Quota usage changes are tracked and quota limits are enforced.

Disk Quota Limits

The disk space that each file uses is charged directly to the user who owns the file. The owner of a file is identified by the security identifier (SID) in the security information for the file. The total disk space charged to a user is the sum of the length of all data streams. In other words, property set streams and resident user data streams affect the user's quota.

Quota is not charged for re-parse points, security descriptors, or other metadata that is associated with the files. Compressing or decompressing files does not affect the disk space reported for the files. Therefore, quota settings on one volume can be compared to settings on another volume.

The following list identifies the types of disk quota limits:

1. **Warning threshold.** You can configure the system to generate a system logfile entry when the disk space charged to the user exceeds this value.
2. **Hard quota.** You can configure the system to generate a system logfile entry when the disk space charged to the user exceeds this value. You can also configure the system to deny additional disk space to the user when the disk space charged to the user exceeds this value.

The NTFS file system automatically creates a user quota entry when a user first writes to the volume. Entries that are created automatically are assigned the default warning threshold and hard quota limit values for the volume.

Disk Quota Interfaces

The following interfaces are used with disk quotas:

Interface	Description
IDiskQuotaControl()	Controls the disk quota facilities of a single NTFS file system volume. The client can query and set volume-specific quota attributes through IDiskQuotaControl. The client can also enumerate all per-user quota entries on the volume. A client instantiates this interface by calling the CoCreateInstance() function using the class identifier CLSID_DiskQuotaControl.
IDiskQuotaEvents()	A client must implement the IDiskQuotaEvents interface as an event sink that receives the quota-related event notifications. Its methods are called by the system whenever significant quota events have occurred. Currently, the only event supported is the asynchronous resolution of user account name information.
IDiskQuotaUser()	Represents a single user quota entry in the volume quota information file. Through this interface, you can query and modify user-specific quota information on an NTFS file system volume. This interface is instantiated by using IEnumDiskQuotaUsers(), IDiskQuotaControl::FindUserSid(),

	IDiskQuotaControl::FindUserName(), IDiskQuotaControl::AddUserSid(), or IDiskQuotaControl::AddUserName().
IDiskQuotaUserBatch()	Adds multiple quota user objects to a container that is then submitted for update in a single call. This reduces the number of calls to the underlying file system, improving update efficiency when a large number of user objects must be updated. This interface is instantiated by using the IDiskQuotaControl::CreateUserBatch() method.
IEnumDiskQuotaUsers()	Enumerates user quota entries on the volume. This interface is instantiated by using the IDiskQuotaControl::CreateEnumUsers() method.

Disk Management Control Codes

The file system provides an abstraction of the physical characteristics of storage devices so that applications can simply write to and read from files. However, storage in an enterprise relies heavily on the concept of disks. The following table identifies the control codes that are used in disk management.

Control code	Operation
IOCTL_DISK_CREATE_DISK	Initializes the specified disk and disk partition table by using the specified information.
IOCTL_DISK_DELETE_DRIVE_LAYOUT	Removes the boot signature from the master boot record.
IOCTL_DISK_FORMAT_TRACKS	Formats a contiguous set of floppy disk tracks.
IOCTL_DISK_FORMAT_TRACKS_EX	Formats a contiguous set of floppy disk tracks with an extended set of track specification parameters.
IOCTL_DISK_GET_CACHE_INFORMATION	Retrieves the disk cache configuration data.
IOCTL_DISK_GET_DRIVE_GEOMETRY_EX	Retrieves information about the physical disk's geometry.
IOCTL_DISK_GET_DRIVE_LAYOUT_EX	Retrieves information about the number of partitions on a disk and the features of each partition.
IOCTL_DISK_GET_LENGTH_INFO	Retrieves the length of the specified disk, volume, or partition.
IOCTL_DISK_GET_PARTITION_INFO_EX	Retrieves partition information for AT and EFI (Extensible Firmware Interface) partitions.
IOCTL_DISK_GROW_PARTITION	Enlarges the specified partition.
IOCTL_DISK_IS_WRITABLE	Determines whether the specified disk is

	writable.
IOCTL_DISK_PERFORMANCE	Provides disk performance information.
IOCTL_DISK_PERFORMANCE_OFF	Disables disk performance information.
IOCTL_DISK_REASSIGN_BLOCKS	Maps disk blocks to spare-block pool.
IOCTL_DISK_SET_CACHE_INFORMATION	Sets the disk cache configuration data.
IOCTL_DISK_SET_DRIVE_LAYOUT_EX	Partitions a disk.
IOCTL_DISK_SET_PARTITION_INFO_EX	Sets the disk partition type.
IOCTL_DISK_UPDATE_PROPERTIES	Invalidates the cached partition table of the specified disk and re-enumerates the disk.
IOCTL_DISK_VERIFY	Performs logical format of a disk extent.

The following list identifies the obsolete control codes:

1. IOCTL_DISK_CONTROLLER_NUMBER
2. IOCTL_DISK_GET_DRIVE_GEOMETRY
3. IOCTL_DISK_GET_DRIVE_LAYOUT
4. IOCTL_DISK_GET_PARTITION_INFO
5. IOCTL_DISK_HISTOGRAM_DATA
6. IOCTL_DISK_HISTOGRAM_RESET
7. IOCTL_DISK_HISTOGRAM_STRUCTURE
8. IOCTL_DISK_LOGGING
9. IOCTL_DISK_REQUEST_DATA
10. IOCTL_DISK_REQUEST_STRUCTURE
11. IOCTL_DISK_SET_DRIVE_LAYOUT
12. IOCTL_DISK_SET_PARTITION_INFO

Disk Management Enumeration Types

The following enumeration types are used with disk management:

1. MEDIA_TYPE
2. PARTITION_STYLE

MEDIA_TYPE Enumeration Definition

Represents the various forms of device media. The Syntax:

```
typedef enum _MEDIA_TYPE {  
    Unknown,  
    F5_1Pt2_512,  
    F3_1Pt44_512,  
    F3_2Pt88_512,  
    F3_20Pt8_512,
```

```

F3_720_512,
F5_360_512,
F5_320_512,
F5_320_1024,
F5_180_512,
F5_160_512,
RemovableMedia,
FixedMedia,
F3_120M_512,
F3_640_512,
F5_640_512,
F5_720_512,
F3_1Pt2_512,
F3_1Pt23_1024,
F5_1Pt23_1024,
F3_128Mb_512,
F3_230Mb_512,
F8_256_128,
F3_200Mb_512,
F3_240M_512,
F3_32M_512
} MEDIA_TYPE;

```

Constants

Constant	Meaning
Unknown	Format is unknown
F5_1Pt2_512	A 5.25" floppy, with 1.2MB and 512 bytes/sector.
F3_1Pt44_512	A 3.5" floppy, with 1.44MB and 512 bytes/sector.
F3_2Pt88_512	A 3.5" floppy, with 2.88MB and 512 bytes/sector.
F3_20Pt8_512	A 3.5" floppy, with 20.8MB and 512 bytes/sector.
F3_720_512	A 3.5" floppy, with 720KB and 512 bytes/sector.
F5_360_512	A 5.25" floppy, with 360KB and 512 bytes/sector.
F5_320_512	A 5.25" floppy, with 320KB and 512 bytes/sector.
F5_320_1024	A 5.25" floppy, with 320KB and 1024 bytes/sector.
F5_180_512	A 5.25" floppy, with 180KB and 512 bytes/sector.
F5_160_512	A 5.25" floppy, with 160KB and 512 bytes/sector.
RemovableMedia	Removable media other than floppy.
FixedMedia	Fixed hard disk media.
F3_120M_512	A 3.5" floppy, with 120MB and 512 bytes/sector.
F3_640_512	A 3.5" floppy, with 640KB and 512 bytes/sector.
F5_640_512	A 5.25" floppy, with 640KB and 512 bytes/sector.
F5_720_512	A 5.25" floppy, with 720KB and 512 bytes/sector.
F3_1Pt2_512	A 3.5" floppy, with 1.2MB and 512 bytes/sector.
F3_1Pt23_1024	A 3.5" floppy, with 1.23MB and 1024 bytes/sector.

F5_1Pt23_1024	A 5.25" floppy, with 1.23MB and 1024 bytes/sector.
F3_128Mb_512	A 3.5" floppy, with 128MB and 512 bytes/sector.
F3_230Mb_512	A 3.5" floppy, with 230MB and 512 bytes/sector.
F8_256_128	An 8" floppy, with 256KB and 128 bytes/sector.
F3_200Mb_512	A 3.5" floppy, with 200MB and 512 bytes/sector. (HiFD).
F3_240M_512	A 3.5" floppy, with 240MB and 512 bytes/sector. (HiFD).
F3_32M_512	A 3.5" floppy, with 32MB and 512 bytes/sector.
...	...

The MediaType member of the DISK_GEOMETRY data structure is of type MEDIA_TYPE. The DeviceIoControl() function receives a DISK_GEOMETRY structure in response to an IOCTL_DISK_GET_DRIVE_GEOMETRY control code. The DeviceIoControl() function receives an array of DISK_GEOMETRY structures in response to an IOCTL_STORAGE_GET_MEDIA_TYPES control code. The STORAGE_MEDIA_TYPE enumeration type extends this enumeration type.

PARTITION_STYLE Enumeration Definition

Represents the format of a partition. The syntax is:

```
typedef enum _PARTITION_STYLE {
    PARTITION_STYLE_MBR    = 0,
    PARTITION_STYLE_GPT    = 1,
    PARTITION_STYLE_RAW    = 2
} PARTITION_STYLE;
```

Constants

Constant	Meaning
PARTITION_STYLE_MBR	Master boot record (MBR) format. This corresponds to standard AT-style MBR partitions.
PARTITION_STYLE_GPT	GUID Partition Table (GPT) format.
PARTITION_STYLE_RAW	Partition not formatted in either of the recognized formats, MBR or GPT.

Disk Management Functions

The following functions are used in disk management.

Function	Description
CreateFile()	Creates or opens a file object.
DeleteFile()	Deletes an existing file.
GetDiskFreeSpace()	Retrieves information about the specified disk, including the amount of

	free space on the disk.
GetDiskFreeSpaceEx()	Retrieves information about the specified disk, including the amount of free space on the disk.

CreateFile() Function

Creates or opens a file or I/O device. The most commonly used I/O devices are as follows: file, file stream, directory, physical disk, volume, console buffer, tape drive, communications resource, mailslot, and pipe. The function returns a handle that can be used to access the file or device for various types of I/O depending on the file or device and the flags and attributes specified. To perform this operation as a transacted operation, which results in a handle that can be used for transacted I/O, use the CreateFileTransacted() function. The syntax is:

```
HANDLE WINAPI CreateFile(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);
```

Parameters

lpFileName [in] - The name of the file or device to be created or opened. In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. To create a file stream, specify the name of the file, a colon, and then the name of the stream.

dwDesiredAccess [in] - The requested access to the file or device, which can be summarized as read, write, both or neither (zero). The most commonly used values are GENERIC_READ, GENERIC_WRITE, or both (GENERIC_READ | GENERIC_WRITE).

If this parameter is zero, the application can query certain metadata such as file, directory, or device attributes without accessing that file or device, even if GENERIC_READ access would have been denied. You cannot request an access mode that conflicts with the sharing mode that is specified by the *dwShareMode* parameter in an open request that already has an open handle.

dwShareMode [in] - The requested sharing mode of the file or device, which can be read, write, both, delete, all of these, or none (refer to the following table). Access requests to attributes or extended attributes are not affected by this flag.

If this parameter is zero and CreateFile() succeeds, the file or device cannot be shared and cannot be opened again until the handle to the file or device is closed.

You cannot request a sharing mode that conflicts with the access mode that is specified in an existing request that has an open handle. CreateFile() would fail and the GetLastError() function

would return ERROR_SHARING_VIOLATION. To enable a process to share a file or device while another process has the file or device open, use a compatible combination of one or more of the following values. The sharing options for each open handle remain in effect until that handle is closed, regardless of process context.

Value	Meaning
0 (0x00000000)	Prevents other processes from opening a file or device if they request delete, read, or write access.
FILE_SHARE_DELETE (0x00000004)	Enables subsequent open operations on a file or device to request delete access. Otherwise, other processes cannot open the file or device if they request delete access. If this flag is not specified, but the file or device has been opened for delete access, the function fails. The delete access allows both delete and rename operations.
FILE_SHARE_READ (0x00000001)	Enables subsequent open operations on a file or device to request read access. Otherwise, other processes cannot open the file or device if they request read access. If this flag is not specified, but the file or device has been opened for read access, the function fails.
FILE_SHARE_WRITE (0x00000002)	Enables subsequent open operations on a file or device to request write access. Otherwise, other processes cannot open the file or device if they request write access. If this flag is not specified, but the file or device has been opened for write access or has a file mapping with write access, the function fails.

lpSecurityAttributes [in, optional] - A pointer to a SECURITY_ATTRIBUTES structure that contains two separate but related data members: an optional security descriptor, and a Boolean value that determines whether the returned handle can be inherited by child processes. This parameter can be NULL. If this parameter is NULL, the handle returned by CreateFile() cannot be inherited by any child processes the application may create and the file or device associated with the returned handle gets a default security descriptor.

The lpSecurityDescriptor member of the structure specifies a SECURITY_DESCRIPTOR for a file or device. If this member is NULL, the file or device associated with the returned handle is assigned a default security descriptor.

CreateFile() ignores the lpSecurityDescriptor member when opening an existing file or device, but continues to use the **bInheritHandle** member. The bInheritHandle member of the structure specifies whether the returned handle can be inherited.

dwCreationDisposition [in] - An action to take on a file or device that exists or does not exist. For devices other than files, this parameter is usually set to OPEN_EXISTING. This parameter must be one of the following values, which cannot be combined:

Value	Meaning
CREATE_ALWAYS (2)	Creates a new file, always. If the specified file exists and is writable, the function overwrites

	the file, the function succeeds, and last-error code is set to ERROR_ALREADY_EXISTS (183). If the specified file does not exist and is a valid path, a new file is created, the function succeeds, and the last-error code is set to zero.
CREATE_NEW (1)	Creates a new file, only if it does not already exist. If the specified file exists, the function fails and the last-error code is set to ERROR_FILE_EXISTS (80). If the specified file does not exist and is a valid path to a writable location, a new file is created.
OPEN_ALWAYS (4)	Opens a file, always. If the specified file exists, the function succeeds and the last-error code is set to ERROR_ALREADY_EXISTS (183). If the specified file does not exist and is a valid path to a writable location, the function creates a file and the last-error code is set to zero.
OPEN_EXISTING (3)	Opens a file or device, only if it exists. If the specified file or device does not exist, the function fails and the last-error code is set to ERROR_FILE_NOT_FOUND (2).
TRUNCATE_EXISTING (5)	Opens a file and truncates it so that its size is zero bytes, only if it exists. If the specified file does not exist, the function fails and the last-error code is set to ERROR_FILE_NOT_FOUND (2). The calling process must open the file with the GENERIC_WRITE bit set as part of the dwDesiredAccess parameter.

dwFlagsAndAttributes [in] - The file or device attributes and flags, FILE_ATTRIBUTE_NORMAL being the most common default value for files. This parameter can include any combination of the available file attributes (FILE_ATTRIBUTE_*). All other file attributes override FILE_ATTRIBUTE_NORMAL. This parameter can also contain combinations of flags (FILE_FLAG_*) for control of file or device caching behavior, access modes, and other special-purpose flags. These combine with any FILE_ATTRIBUTE_* values.

This parameter can also contain Security Quality of Service information by specifying the SECURITY_SQOS_PRESENT flag. Additional SQOS-related flags information is presented in the table following the attributes and flags tables.

When CreateFile() opens an existing file, it generally combines the file flags with the file attributes of the existing file, and ignores any file attributes supplied as part of *dwFlagsAndAttributes*. Some of the following file attributes and flags may only apply to files and not necessarily all other types of devices that CreateFile() can open.

Attribute	Meaning
FILE_ATTRIBUTE_ARCHIVE (32 (0x20))	The file should be archived. Applications use this attribute to mark files for backup or removal.
FILE_ATTRIBUTE_ENCRYPTED	The file or directory is encrypted. For a file, this means

(16384 (0x4000))	that all data in the file is encrypted. For a directory, this means that encryption is the default for newly created files and subdirectories. This flag has no effect if FILE_ATTRIBUTE_SYSTEM is also specified.
FILE_ATTRIBUTE_HIDDEN (2 (0x2))	The file is hidden. Do not include it in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL (128 (0x80))	The file does not have other attributes set. This attribute is valid only if used alone.
FILE_ATTRIBUTE_OFFLINE (4096 (0x1000))	The data of a file is not immediately available. This attribute indicates that file data is physically moved to offline storage. This attribute is used by Remote Storage, the hierarchical storage management software. Applications should not arbitrarily change this attribute.
FILE_ATTRIBUTE_READONLY (1 (0x1))	The file is read only. Applications can read the file, but cannot write to or delete it.
FILE_ATTRIBUTE_SYSTEM (4 (0x4))	The file is part of or used exclusively by an operating system.
FILE_ATTRIBUTE_TEMPORARY (256 (0x100))	The file is being used for temporary storage.

Flag	Meaning
FILE_FLAG_BACKUP_SEMANTICS (0x02000000)	The file is being opened or created for a backup or restore operation. The system ensures that the calling process overrides file security checks when the process has SE_BACKUP_NAME and SE_RESTORE_NAME privileges. You must set this flag to obtain a handle to a directory. A directory handle can be passed to some functions instead of a file handle.
FILE_FLAG_DELETE_ON_CLOSE (0x04000000)	The file is to be deleted immediately after all of its handles are closed, which includes the specified handle and any other open or duplicated handles. If there are existing open handles to a file, the call fails unless they were all opened with the FILE_SHARE_DELETE share mode. Subsequent open requests for the file fail, unless the FILE_SHARE_DELETE share mode is specified.

<p>FILE_FLAG_NO_BUFFERING (0x20000000)</p>	<p>The file or device is being opened with no system caching for data reads and writes. This flag does not affect hard disk caching or memory mapped files. There are strict requirements for successfully working with files opened with CreateFile() using the FILE_FLAG_NO_BUFFERING flag.</p>
<p>FILE_FLAG_OPEN_NO_RECALL (0x00100000)</p>	<p>The file data is requested, but it should continue to be located in remote storage. It should not be transported back to local storage. This flag is for use by remote storage systems.</p>
<p>FILE_FLAG_OPEN_REPARSE_POINT (0x00200000)</p>	<p>Normal reparse point processing will not occur; CreateFile() will attempt to open the reparse point. When a file is opened, a file handle is returned, whether or not the filter that controls the reparse point is operational. This flag cannot be used with the CREATE_ALWAYS flag. If the file is not a reparse point, then this flag is ignored.</p>
<p>FILE_FLAG_OVERLAPPED (0x40000000)</p>	<p>The file or device is being opened or created for asynchronous I/O. When subsequent I/O operations are completed on this handle, the event specified in the OVERLAPPED structure will be set to the signaled state. If this flag is specified, the file can be used for simultaneous read and write operations. If this flag is not specified, then I/O operations are serialized, even if the calls to the read and write functions specify an OVERLAPPED structure.</p>
<p>FILE_FLAG_POSIX_SEMANTICS (0x01000000)</p>	<p>Access will occur according to POSIX rules. This includes allowing multiple files with names, differing only in case, for file systems that support that naming. Use care when using this option, because files created with this flag may not be accessible by applications that are written for MS-DOS or 16-bit Windows.</p>
<p>FILE_FLAG_RANDOM_ACCESS (0x10000000)</p>	<p>Access is intended to be random. The system can use this as a hint to optimize file caching. This flag has no effect if the file system does not support cached I/O and FILE_FLAG_NO_BUFFERING.</p>

FILE_FLAG_SEQUENTIAL_SCAN (0x08000000)	Access is intended to be sequential from beginning to end. The system can use this as a hint to optimize file caching. This flag should not be used if read-behind (that is, backwards scans) will be used. This flag has no effect if the file system does not support cached I/O and FILE_FLAG_NO_BUFFERING.
FILE_FLAG_WRITE_THROUGH (0x80000000)	Write operations will not go through any intermediate cache, they will go directly to disk.

The *dwFlagsAndAttributes* parameter can also specify Security Quality of Service information. When the calling application specifies the SECURITY_SQOS_PRESENT flag as part of *dwFlagsAndAttributes*, it can also contain one or more of the following values.

Security flag	Meaning
SECURITY_ANONYMOUS	Impersonates a client at the Anonymous impersonation level.
SECURITY_CONTEXT_TRACKING	The security tracking mode is dynamic. If this flag is not specified, the security tracking mode is static.
SECURITY_DELEGATION	Impersonates a client at the Delegation impersonation level.
SECURITY_EFFECTIVE_ONLY	Only the enabled aspects of the client's security context are available to the server. If you do not specify this flag, all aspects of the client's security context are available. This allows the client to limit the groups and privileges that a server can use while impersonating the client.
SECURITY_IDENTIFICATION	Impersonates a client at the Identification impersonation level.
SECURITY_IMPERSONATION	Impersonate a client at the impersonation level. This is the default behavior if no other flags are specified along with the SECURITY_SQOS_PRESENT flag.

hTemplateFile [in, optional] - A valid handle to a template file with the GENERIC_READ access right. The template file supplies file attributes and extended attributes for the file that is being created. This parameter can be NULL. When opening an existing file, CreateFile() ignores this parameter. When opening a new encrypted file, the file inherits the discretionary access control list from its parent directory.

Return Value

If the function succeeds, the return value is an open handle to the specified file, device, named pipe, or mail slot. If the function fails, the return value is `INVALID_HANDLE_VALUE`. To get extended error information, call `GetLastError()`.

`CreateFile()` was originally developed specifically for file interaction but has since been expanded and enhanced to include most other types of I/O devices and mechanisms available to Windows developers. This section attempts to cover the varied issues developers may experience when using `CreateFile()` in different contexts and with different I/O types. The text attempts to use the word *file* only when referring specifically to data stored in an actual file on a file system. However, some uses of *file* may be referring more generally to an I/O object that supports file-like mechanisms. This liberal use of the term *file* is particularly prevalent in constant names and parameter names because of the previously mentioned historical reasons.

When an application is finished using the object handle returned by `CreateFile()`, use the `CloseHandle()` function to close the handle. This not only frees up system resources, but can have wider influence on things like sharing the file or device and committing data to disk. Specifics are noted within this topic as appropriate.

For Windows Server 2003 and Windows XP/2000: A sharing violation occurs if an attempt is made to open a file or directory for deletion on a remote computer when the value of the `dwDesiredAccess` parameter is the `DELETE` access flag OR'ed with any other access flag, and the remote file or directory has not been opened with `FILE_SHARE_DELETE`. To avoid the sharing violation in this scenario, open the remote file or directory with the `DELETE` access right only, or call `DeleteFile()` without first opening the file or directory for deletion.

Some file systems, such as the NTFS file system, support compression or encryption for individual files and directories. On volumes that have a mounted file system with this support, a new file inherits the compression and encryption attributes of its directory. You cannot use `CreateFile()` to control compression, decompression, or decryption on a file or directory.

For Windows Server 2003 and Windows XP/2000: For backward compatibility purposes, `CreateFile()` does not apply inheritance rules when you specify a security descriptor in `lpSecurityAttributes`. To support inheritance, functions that later query the security descriptor of this file may heuristically determine and report that inheritance is in effect. As stated previously, if the `lpSecurityAttributes` parameter is `NULL`, the handle returned by `CreateFile()` cannot be inherited by any child processes your application may create. The following information regarding this parameter also applies:

1. If the `bInheritHandle` member variable is not `FALSE`, which is any non-zero value, then the handle can be inherited. Therefore it is critical this structure member be properly initialized to `FALSE` if you do not intend the handle to be inheritable.
2. The access control lists (ACL) in the default security descriptor for a file or directory are inherited from its parent directory.
3. The target file system must support security on files and directories for the `lpSecurityDescriptor` member to have an effect on them, which can be determined by using `GetVolumeInformation()`.

Symbolic Link Behavior

If the call to this function creates a file, there is no change in behavior. Also, consider the following information regarding `FILE_FLAG_OPEN_REPARSE_POINT`:

If `FILE_FLAG_OPEN_REPARSE_POINT` is specified:

1. If an existing file is opened and it is a symbolic link, the handle returned is a handle to the symbolic link.
2. If `TRUNCATE_EXISTING` or `FILE_FLAG_DELETE_ON_CLOSE` are specified, the file affected is a symbolic link.

If `FILE_FLAG_OPEN_REPARSE_POINT` is not specified:

1. If an existing file is opened and it is a symbolic link, the handle returned is a handle to the target.
2. If `CREATE_ALWAYS`, `TRUNCATE_EXISTING`, or `FILE_FLAG_DELETE_ON_CLOSE` are specified, the file affected is the target.

Caching Behavior

Several of the possible values for the *dwFlagsAndAttributes* parameter are used by `CreateFile()` to control or affect how the data associated with the handle is cached by the system. They are:

1. `FILE_FLAG_NO_BUFFERING`
2. `FILE_FLAG_RANDOM_ACCESS`
3. `FILE_FLAG_SEQUENTIAL_SCAN`
4. `FILE_FLAG_WRITE_THROUGH`
5. `FILE_ATTRIBUTE_TEMPORARY`

If none of these flags is specified, the system uses a default general-purpose caching scheme.

Otherwise, the system caching behaves as specified for each flag.

Some of these flags should not be combined. For instance, combining

`FILE_FLAG_RANDOM_ACCESS` with `FILE_FLAG_SEQUENTIAL_SCAN` is self-defeating.

Specifying the `FILE_FLAG_SEQUENTIAL_SCAN` flag can increase performance for applications that read large files using sequential access. Performance gains can be even more noticeable for applications that read large files mostly sequentially, but occasionally skip forward over small ranges of bytes. If an application moves the file pointer for random access, optimum caching performance most likely will not occur. However, correct operation is still guaranteed. The flags `FILE_FLAG_WRITE_THROUGH` and `FILE_FLAG_NO_BUFFERING` are independent and may be combined.

If `FILE_FLAG_WRITE_THROUGH` is used but `FILE_FLAG_NO_BUFFERING` is not also specified, so that system caching is in effect, then the data is written to the system cache but is flushed to disk without delay.

If `FILE_FLAG_WRITE_THROUGH` and `FILE_FLAG_NO_BUFFERING` are both specified, so that system caching is not in effect, then the data is immediately flushed to disk without going

through the Windows system cache. The operating system also requests a write-through of the hard disk's local hardware cache to persistent media.

Not all hard disk hardware supports this write-through capability.

Proper use of the `FILE_FLAG_NO_BUFFERING` flag requires special application considerations.

A write-through request via `FILE_FLAG_WRITE_THROUGH` also causes NTFS to flush any metadata changes, such as a time stamp update or a rename operation, that result from processing the request. For this reason, the `FILE_FLAG_WRITE_THROUGH` flag is often used with the `FILE_FLAG_NO_BUFFERING` flag as a replacement for calling the `FlushFileBuffers()` function after each write, which can cause unnecessary performance penalties. Using these flags together avoids those penalties.

When `FILE_FLAG_NO_BUFFERING` is combined with `FILE_FLAG_OVERLAPPED`, the flags give maximum asynchronous performance, because the I/O does not rely on the synchronous operations of the memory manager. However, some I/O operations take more time, because data is not being held in the cache. Also, the file metadata may still be cached (for example, when creating an empty file). To ensure that the metadata is flushed to disk, use the `FlushFileBuffers()` function.

Specifying the `FILE_ATTRIBUTE_TEMPORARY` attribute causes file systems to avoid writing data back to mass storage if sufficient cache memory is available, because an application deletes a temporary file after a handle is closed. In that case, the system can entirely avoid writing the data. Although it doesn't directly control data caching in the same way as the previously mentioned flags, the `FILE_ATTRIBUTE_TEMPORARY` attribute does tell the system to hold as much as possible in the system cache without writing and therefore may be of concern for certain applications.

Files

If you rename or delete a file and then restore it shortly afterward, the system searches the cache for file information to restore. Cached information includes its short/long name pair and creation time.

If you call `CreateFile()` on a file that is pending deletion as a result of a previous call to `DeleteFile()`, the function fails. The operating system delays file deletion until all handles to the file are closed. `GetLastError()` returns `ERROR_ACCESS_DENIED`.

The `dwDesiredAccess` parameter can be zero, allowing the application to query file attributes without accessing the file if the application is running with adequate security settings. This is useful to test for the existence of a file without opening it for read and/or write access, or to obtain other statistics about the file or directory.

For Windows Server 2003 and Windows XP/2000: If `CREATE_ALWAYS` and `FILE_ATTRIBUTE_NORMAL` are specified, `CreateFile()` fails and sets the last error to `ERROR_ACCESS_DENIED` if the file exists and has the `FILE_ATTRIBUTE_HIDDEN` or `FILE_ATTRIBUTE_SYSTEM` attribute. To avoid the error, specify the same attributes as the existing file.

When an application creates a file across a network, it is better to use `GENERIC_READ | GENERIC_WRITE` for `dwDesiredAccess` than to use `GENERIC_WRITE` alone. The resulting code is faster, because the redirector can use the cache manager and send fewer SMBs with more

data. This combination also avoids an issue where writing to a file across a network can occasionally return `ERROR_ACCESS_DENIED`.

Synchronous and Asynchronous I/O Handles

`CreateFile()` provides for creating a file or device handle that is either synchronous or asynchronous. A synchronous handle behaves such that I/O function calls using that handle are blocked until they complete, while an asynchronous file handle makes it possible for the system to return immediately from I/O function calls, whether they completed the I/O operation or not. As stated previously, this synchronous versus asynchronous behavior is determined by specifying `FILE_FLAG_OVERLAPPED` within the `dwFlagsAndAttributes` parameter. There are several complexities and potential pitfalls when using asynchronous I/O;

File Streams

On NTFS file systems, you can use `CreateFile()` to create separate streams within a file.

Directories

An application cannot create a directory by using `CreateFile()`, therefore only the `OPEN_EXISTING` value is valid for `dwCreationDisposition` for this use case. To create a directory, the application must call `CreateDirectory()` or `CreateDirectoryEx()`.

To open a directory using `CreateFile()`, specify the `FILE_FLAG_BACKUP_SEMANTICS` flag as part of `dwFlagsAndAttributes`. Appropriate security checks still apply when this flag is used without `SE_BACKUP_NAME` and `SE_RESTORE_NAME` privileges.

When using `CreateFile()` to open a directory during defragmentation of a FAT or FAT32 file system volume, do not specify the `MAXIMUM_ALLOWED` access right. Access to the directory is denied if this is done. Specify the `GENERIC_READ` access right instead.

Physical Disks and Volumes

Direct access to the disk or to a volume is restricted. For more information, see "Changes to the file system and to the storage stack to restrict direct disk access and direct volume access in Windows Vista and in Windows Server 2008" in the Help and Support Knowledge Base at [Changes to the file system and to the storage stack to restrict direct disk access and direct volume access in Windows Vista and in Windows Server 2008](#).

For Windows Server 2003 and Windows XP/2000 the direct access to the disk or to a volume is not restricted in this manner.

You can use the `CreateFile()` function to open a physical disk drive or a volume, which returns a **direct access storage device (DASD)** handle that can be used with the `DeviceIoControl()` function. This enables you to access the disk or volume directly, for example such disk metadata as the partition table. However, this type of access also exposes the disk drive or volume to potential data loss, because an incorrect write to a disk using this mechanism could make its contents inaccessible to the operating system. To ensure data integrity, be sure to become familiar with `DeviceIoControl()` and how other APIs behave differently with a direct access

handle as opposed to a file system handle. The following requirements must be met for such a call to succeed:

1. The caller must have administrative privileges.
2. The `dwCreationDisposition` parameter must have the `OPEN_EXISTING` flag.
3. When opening a volume or floppy disk, the `dwShareMode` parameter must have the `FILE_SHARE_WRITE` flag.

Take note that the `dwDesiredAccess` parameter can be zero, allowing the application to query device attributes without accessing a device. This is useful for an application to determine the size of a floppy disk drive and the formats it supports without requiring a floppy disk in a drive, for instance. It can also be used for reading statistics without requiring higher-level data read/write permission. When opening a physical drive *x*, the `lpFileName` string should be the following form: `\\.\PhysicalDriveX`. Hard disk numbers start at zero. The following table shows some examples of physical drive strings.

String	Meaning
\\.\PhysicalDrive0	Opens the first physical drive.
\\.\PhysicalDrive2	Opens the third physical drive.

To obtain the physical drive identifier for a volume, open a handle to the volume and call the `DeviceIoControl()` function with `IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS`. This control code returns the disk number and offset for each of the volume's one or more extents; a volume can span multiple physical disks.

When opening a volume or removable media drive (for example, a floppy disk drive or flash memory thumb drive), the `lpFileName` string should be the following form: `\\.\X:`. Do not use a trailing backslash (`\`), which indicates the root directory of a drive. The following table shows some examples of drive strings.

String	Meaning
\\.\A:	Opens floppy disk drive A.
\\.\C:	Opens the C: volume.
\\.\C:\	Opens the file system of the C: volume.

You can also open a volume by referring to its volume name. A volume contains one or more mounted file systems. Volume handles can be opened as noncached at the discretion of the particular file system, even when the noncached option is not specified in `CreateFile()`. You should assume that all Microsoft file systems open volume handles as noncached. The restrictions on noncached I/O for files also apply to volumes.

A file system may or may not require buffer alignment even though the data is noncached. However, if the noncached option is specified when opening a volume, buffer alignment is enforced regardless of the file system on the volume. It is recommended on all file systems that you open volume handles as noncached, and follow the noncached I/O restrictions.

To read or write to the last few sectors of the volume, you must call DeviceIoControl() and specify FSCTL_ALLOW_EXTENDED_DASD_IO. This signals the file system driver not to perform any I/O boundary checks on partition read or write calls. Instead, boundary checks are performed by the device driver.

Changer Device

The IOCTL_CHANGER_* control codes for DeviceIoControl() accept a handle to a changer device. To open a changer device, use a file name of the following form: \\.\Changerx where x is a number that indicates which device to open, starting with zero. To open changer device zero in an application that is written in C or C++, use the following file name: "\\\\.\\Changer0".

Tape Drives

You can open tape drives by using a file name of the following form: \\.\TAPEx where x is a number that indicates which drive to open, starting with tape drive zero. To open tape drive zero in an application that is written in C or C++, use the following file name: "\\\\.\\TAPE0".

Communications Resources

The CreateFile() function can create a handle to a communications resource, such as the serial port COM1. For communications resources, the dwCreationDisposition parameter must be OPEN_EXISTING, the dwShareMode parameter must be zero (exclusive access), and the hTemplateFile parameter must be NULL. Read, write, or read/write access can be specified, and the handle can be opened for overlapped I/O.

To specify a COM port number greater than 9, use the following syntax: "\\.\COM10". This syntax works for all port numbers and hardware that allows COM port numbers to be specified.

Consoles

The CreateFile() function can create a handle to console input (CONIN\$). If the process has an open handle to it as a result of inheritance or duplication, it can also create a handle to the active screen buffer (CONOUT\$). The calling process must be attached to an inherited console or one allocated by the AllocConsole() function. For console handles, set the CreateFile() parameters as follows.

Parameters	Value
lpFileName	Use the CONIN\$ value to specify console input. Use the CONOUT\$ value to specify console output. CONIN\$ gets a handle to the console input buffer, even if the SetStdHandle() function redirects the standard input handle. To get the standard input handle, use the GetStdHandle() function. CONOUT\$ gets a handle to the active screen buffer, even if SetStdHandle() redirects the standard output handle. To get the

	standard output handle, use GetStdHandle().
dwDesiredAccess	GENERIC_READ GENERIC_WRITE is preferred, but either one can limit access.
dwShareMode	When opening CONIN\$, specify FILE_SHARE_READ. When opening CONOUT\$, specify FILE_SHARE_WRITE. If the calling process inherits the console, or if a child process should be able to access the console, this parameter must be FILE_SHARE_READ FILE_SHARE_WRITE.
lpSecurityAttributes	If you want the console to be inherited, the bInheritHandle member of the SECURITY_ATTRIBUTES structure must be TRUE.
dwCreationDisposition	You should specify OPEN_EXISTING when using CreateFile() to open the console.
dwFlagsAndAttributes	Ignored.
hTemplateFile	Ignored.

The following table shows various settings of dwDesiredAccess and lpFileName.

lpFileName	dwDesiredAccess	Result
CON	GENERIC_READ	Opens console for input.
CON	GENERIC_WRITE	Opens console for output.
CON	GENERIC_READ GENERIC_WRITE	Causes CreateFile() to fail; GetLastError() returns ERROR_FILE_NOT_FOUND.

[Mailslots](#)

If CreateFile() opens the client end of a mailslot, the function returns INVALID_HANDLE_VALUE if the mailslot client attempts to open a local mailslot before the mailslot server has created it with the CreateMailSlot() function.

[Pipes](#)

If CreateFile() opens the client end of a named pipe, the function uses any instance of the named pipe that is in the listening state. The opening process can duplicate the handle as many times as required, but after it is opened, the named pipe instance cannot be opened by another client. The access that is specified when a pipe is opened must be compatible with the access that is specified in the dwOpenMode parameter of the CreateNamedPipe() function.

If the CreateNamedPipe() function was not successfully called on the server prior to this operation, a pipe will not exist and CreateFile() will fail with ERROR_FILE_NOT_FOUND. If there is at least one active pipe instance but there are no available listener pipes on the server, which means all pipe instances are currently connected, CreateFile() fails with ERROR_PIPE_BUSY.

Opening a File for Reading or Writing

The `CreateFile()` function can create a new file or open an existing file. You must specify the file name, creation instructions, and other attributes. When an application creates a new file, the operating system adds it to the specified directory.

Example: Open a File for Writing

The following example uses `CreateFile()` to create a new file and open it for writing and `WriteFile()` to write a simple string synchronously to the file. A subsequent call to open this file with `CreateFile()` will fail until the handle is closed.

```
#include <windows.h>
#include <stdio.h>

void main(int argc, CHAR *argv[])
{
    HANDLE hFile;
    char DataBuffer[] = "This is a test string to be written.";
    DWORD dwBytesToWrite = (DWORD)strlen(DataBuffer);
    DWORD dwBytesWritten = 0;

    printf("\n");
    // Verify the argument
    if(argc != 2)
    {
        printf("ERROR:\tIncorrect number of arguments!\n\n");
        printf("%s <file_name>\n", argv[0]);
        return;
    }

    hFile = CreateFile(argv[1],                // file name to write
                      GENERIC_WRITE,         // open for writing
                      0,                      // do not share
                      NULL,                  // default security
                      CREATE_ALWAYS,        // overwrite existing
                      FILE_ATTRIBUTE_NORMAL, // normal file
                      NULL);                 // no attr. template

    if (hFile == INVALID_HANDLE_VALUE)
    {
        printf("Could not open %s file, error %d\n", argv[1],
              GetLastError());
        return;
    }

    printf("Writing %d bytes to %s.\n", dwBytesToWrite, argv[1]);

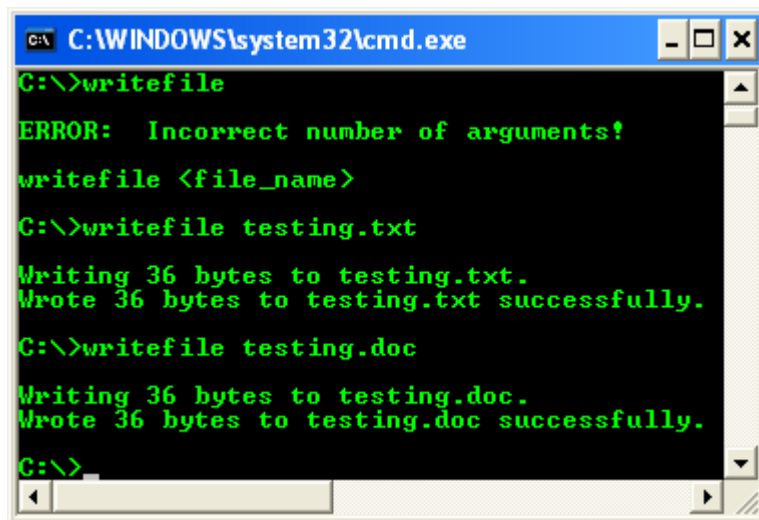
    // This loop would most likely never repeat for this synchronous example.
    // However, during asynchronous writes the system buffer may become full,
    // requiring additional writes until the entire buffer is written
    while (dwBytesWritten < dwBytesToWrite)
```

```
{
    if( FALSE == WriteFile(hFile,          // open file handle
                           DataBuffer + dwBytesWritten, // start of
data to write
                           dwBytesToWrite - dwBytesWritten, // number of
bytes to write
                           &dwBytesWritten, // number of bytes that were
written
                           NULL)          // no overlapped structure
        )
    {
        printf("Could not write to %s file, error %d\n", argv[1],
GetLastError());
        CloseHandle(hFile);
        return;
    }
}

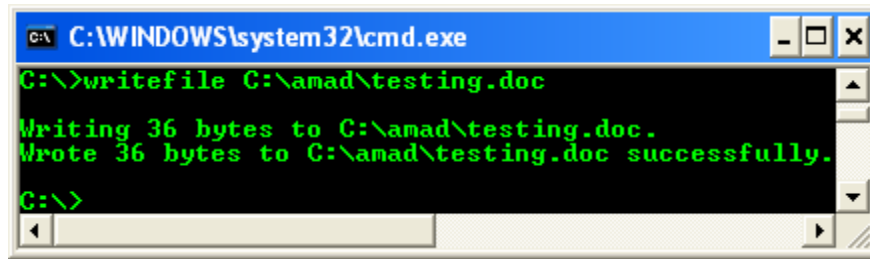
printf("Wrote %d bytes to %s successfully.\n", dwBytesWritten, argv[1]);

CloseHandle(hFile);
}
```

To test this program, we copy the executable to C: and create two files named testing.txt and testing.doc. We put both files at C:

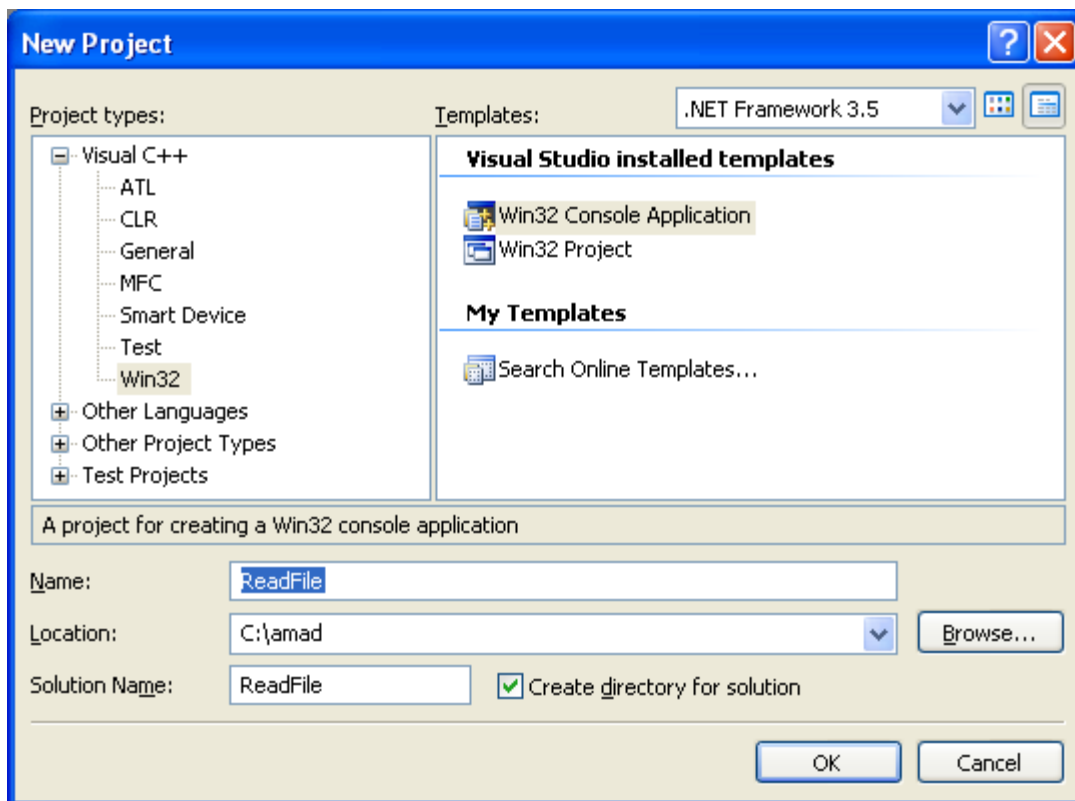


The file to be opened and written can have a relative path as shown in the following screenshot.



Example: Open a File for Reading

The following example uses `CreateFile()` to open an existing file for reading and `ReadFile()` to read up to 80 characters synchronously from the file. In this case, `CreateFile()` succeeds only if the specified file already exists in the current directory. A subsequent call to open this file with `CreateFile()` will succeed if the call uses the same access and sharing modes. You can use the file you created with the previous `WriteFile()` example to test this example.



```
#include <windows.h>
#include <stdio.h>

#define BUFFER_SIZE 82

void main(int argc, WCHAR *argv[])
{
    HANDLE hFile;
```

```
DWORD dwBytesRead = 0;
char ReadBuffer[BUFFER_SIZE] = {0};

printf("\n");
// Verify the argument number
if(argc != 2)
{
    // The file must be available
    printf("ERROR:\tIncorrect number of arguments!\n\n");
    printf("%s <text_file_name>\n", argv[0]);
    return;
}

hFile = CreateFile(argv[1], // file to open
                  GENERIC_READ, // open for reading
                  FILE_SHARE_READ, // share for reading
                  NULL, // default security
                  OPEN_EXISTING, // existing file only
                  FILE_ATTRIBUTE_NORMAL, // normal file
                  NULL); // no attr. template

if (hFile == INVALID_HANDLE_VALUE)
{
    printf("Could not open %s file, error %d\n", argv[1],
GetLastError());
    return;
}

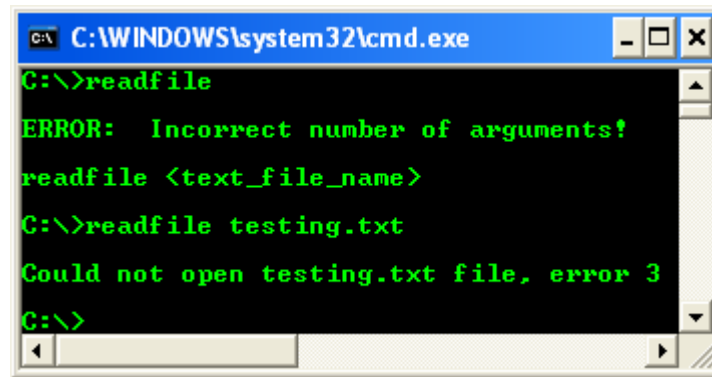
// Read one character less than the buffer size to save room for
// the terminating NULL character
if( FALSE == ReadFile(hFile, ReadBuffer, BUFFER_SIZE-2, &dwBytesRead,
NULL) )
{
    printf("Could not read from %s, error %d\n", argv[1],
GetLastError());
    CloseHandle(hFile);
    return;
}

if (dwBytesRead > 0)
{
    ReadBuffer[dwBytesRead+1]='\0'; // NULL character

    printf("Text read from %s file, %d bytes: \n", argv[1], dwBytesRead);
    printf("%s\n", ReadBuffer);
}
else
{
    printf("No data read from file %s\n", argv[1]);
}

CloseHandle(hFile);
}
```

The output is expected having an error. Please rectify the error.

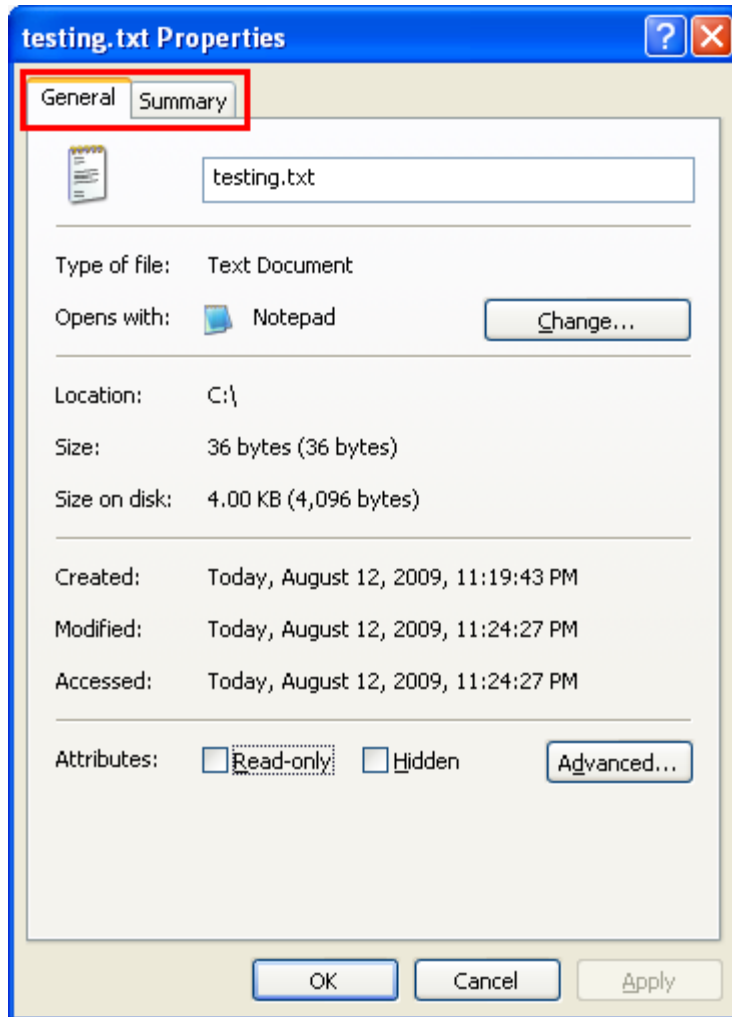


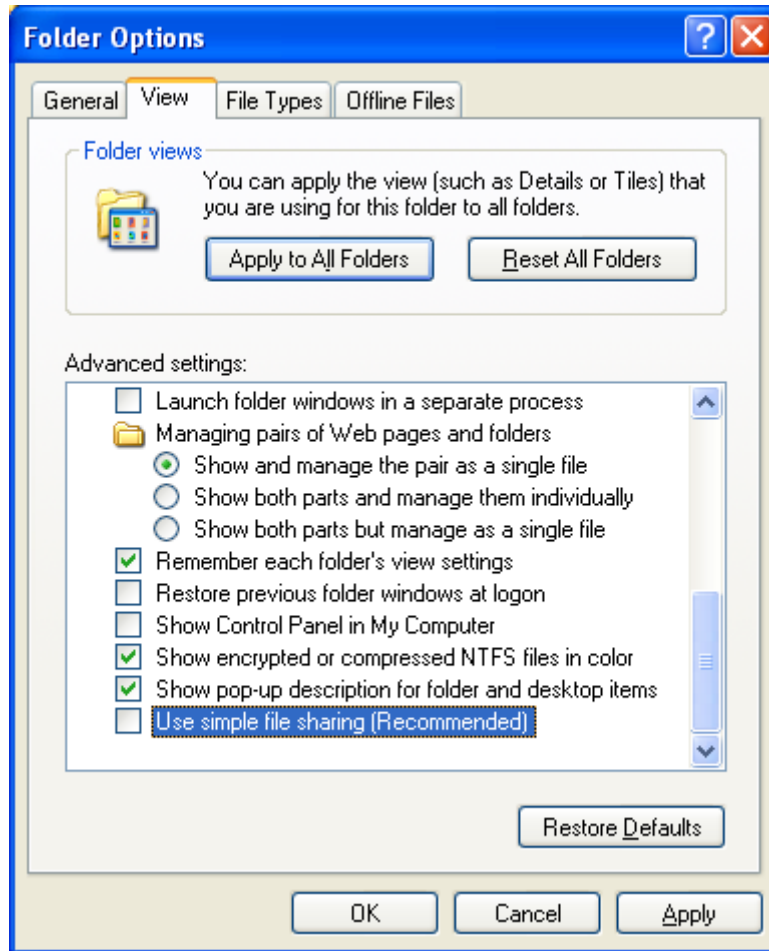
```
C:\WINDOWS\system32\cmd.exe
C:\>readfile
ERROR: Incorrect number of arguments!
readfile <text_file_name>
C:\>readfile testing.txt
Could not open testing.txt file, error 3
C:\>
```

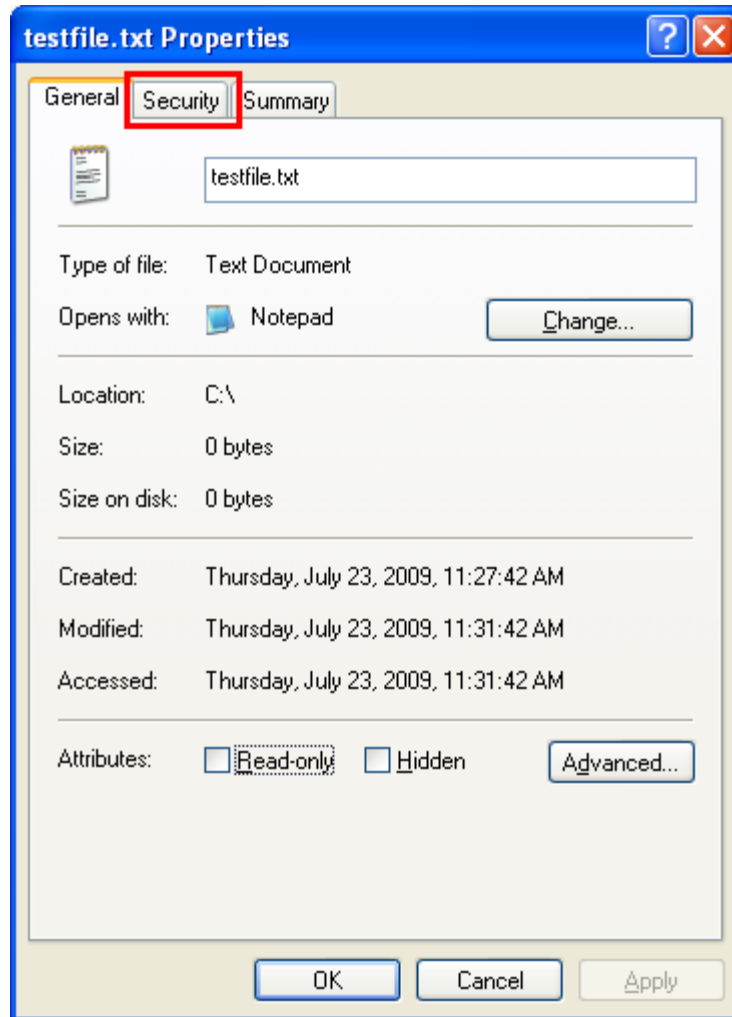
How Do I Get the Security tab in Folder Properties?

First, your hard drive must be formatted NTFS for this tab to show up.

Second, if you're running XP Pro, you must open Windows Explorer > go to Tools > Folder Options > View and uncheck Use Simple File Sharing.







Third, if you're running XP Home, Simple File Sharing is enforced by default and cannot be disabled. You must boot the computer into Safe Mode and log in with the Administrator account, in order to see the Security tab or go to: [Windows XP Home, Simple File Sharing steps](#). Download the x86 (Intel) version of the Security Configuration Manager and save it to your hard disk. Double click the SCESP4I.EXE file you downloaded and extract the contents to a temporary location on your hard disk. Then open the folder you extracted the files to and locate the Setup.inf (Setup Information) file. Right click Setup.inf and select Install. After the installation is finished, reboot your computer. If the download link on the page, listed above, does not work, try this one: [The Security Configuration Manager download](#).

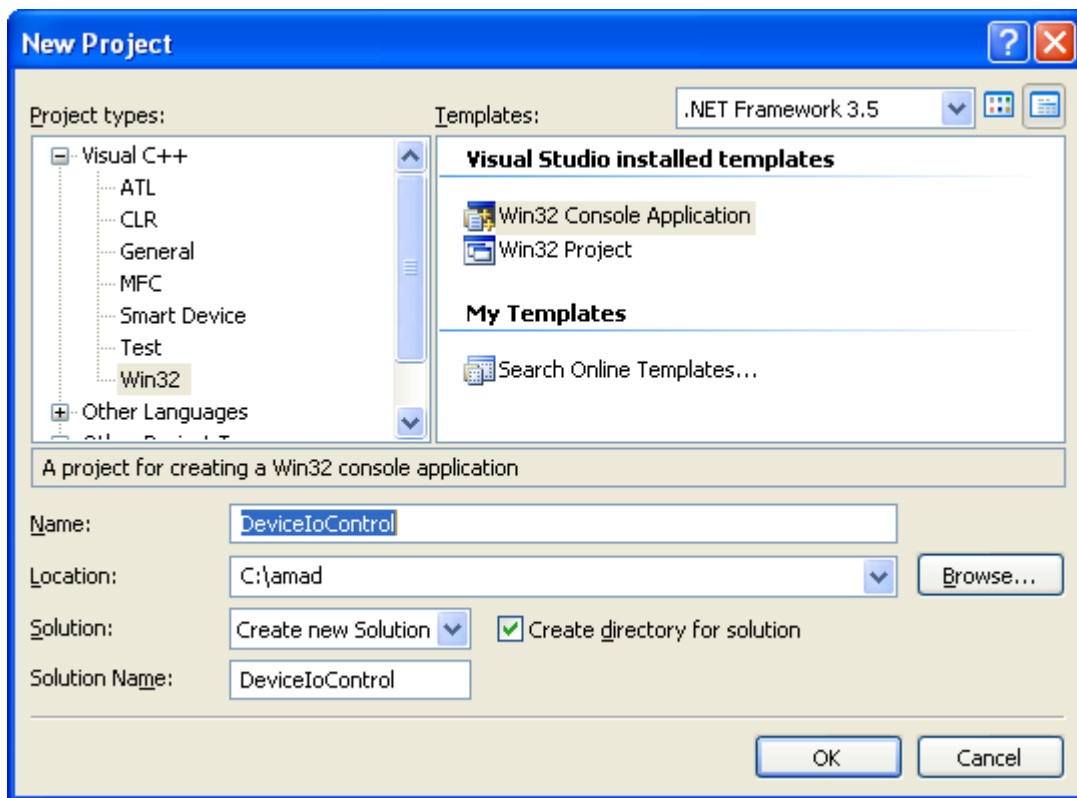
WARNING: Adjusting the permissions on a drive, file or folder can lock even the Administrator account out of that drive/file/folder. Deny Permissions take precedence over Allow Permissions, regardless of your group membership. Administrators are members of the User's group, by default. Uncheck Allow, rather than using Deny.

Calling DeviceIoControl() Program Example

An application can use the DeviceIoControl() function to perform direct input and output operations on, or retrieve information about, a floppy disk drive, hard disk drive, tape drive, or CD-ROM drive.

The following example demonstrates how to retrieve information about the first physical drive in the system. It uses the CreateFile() function to retrieve the device handle to the first physical drive, and then uses DeviceIoControl() with the IOCTL_DISK_GET_DRIVE_GEOMETRY control code to fill a DISK_GEOMETRY structure with information about the drive.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



Then, add the source file and give it a suitable name.

Next, add the following source code.

```
/* The code of interest is in the subroutine GetDriveGeometry(). The
   code in main shows how to interpret the results of the call. */
```

```
#include <windows.h>
#include <winioctl.h>
#include <stdio.h>
```

```
BOOL GetDriveGeometry(DISK_GEOMETRY *pdg)
```

```
{
HANDLE hDevice;           // handle to the drive to be examined
BOOL bResult;            // results flag
DWORD junk;              // discard results

hDevice = CreateFile(TEXT("\\\\.\\PhysicalDrive0"), // drive to open
                    0, // no access to the drive
                    FILE_SHARE_READ | // share mode
                    FILE_SHARE_WRITE,
                    NULL, // default security attributes
                    OPEN_EXISTING, // disposition
                    0, // file attributes
                    NULL); // do not copy file attributes

if (hDevice == INVALID_HANDLE_VALUE) // cannot open the drive
{
    printf("CreateFile() failed!\n");
    return (FALSE);
}

bResult = DeviceIoControl(hDevice, // device to be queried
                          IOCTL_DISK_GET_DRIVE_GEOMETRY, // operation to perform
                          NULL, 0, // no input buffer
                          pdg, sizeof(*pdg), // output buffer
                          &junk, // # bytes returned
                          (LPOVERLAPPED) NULL); // synchronous I/O

CloseHandle(hDevice);

return (bResult);
}

int main(int argc, char *argv[])
{
    DISK_GEOMETRY pdg; // disk drive geometry structure
    BOOL bResult; // generic results flag
    ULONGLONG DiskSize; // size of the drive, in bytes

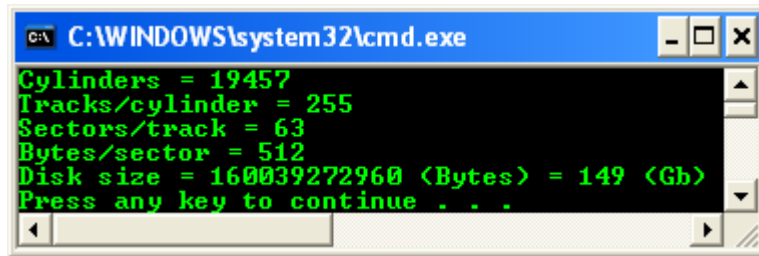
    bResult = GetDriveGeometry (&pdg);

    if (bResult)
    {
        printf("Cylinders = %I64d\n", pdg.Cylinders);
        printf("Tracks/cylinder = %ld\n", (ULONG) pdg.TracksPerCylinder);
        printf("Sectors/track = %ld\n", (ULONG) pdg.SectorsPerTrack);
        printf("Bytes/sector = %ld\n", (ULONG) pdg.BytesPerSector);

        DiskSize = pdg.Cylinders.QuadPart * (ULONG)pdg.TracksPerCylinder *
            (ULONG)pdg.SectorsPerTrack * (ULONG)pdg.BytesPerSector;
        printf("Disk size = %I64d (Bytes) = %I64d (Gb)\n", DiskSize,
            DiskSize / (1024 * 1024 * 1024));
    }
    else
    {
        printf ("GetDriveGeometry failed. Error %ld.\n", GetLastError ());
    }
}
```

```
    return ((int)bResult);  
}
```

Build and run the project. The following screenshot is a sample output.



DeleteFile() Function

Deletes an existing file. To perform this operation as a transacted operation, use the DeleteFileTransacted() function. The syntax is:

```
BOOL WINAPI DeleteFile(LPCTSTR lpFileName);
```

Parameters

lpFileName [in] - The name of the file to be deleted. In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, call the Unicode version of the function and prepend "\\?" to the path.

Return Value

If the function succeeds, the return value is nonzero and if the function fails, the return value is zero (0). To get extended error information, call **GetLastError()**.

Remarks

If an application attempts to delete a file that does not exist, the **DeleteFile()** function fails with ERROR_FILE_NOT_FOUND. If the file is a read-only file, the function fails with ERROR_ACCESS_DENIED. The following list identifies some tips for deleting, removing, or closing files:

1. To delete a read-only file, first you must remove the read-only attribute.
2. To delete or rename a file, you must have either delete permission on the file, or delete child permission in the parent directory.
3. To recursively delete the files in a directory, use the **SHFileOperation()** function.
4. To remove an empty directory, use the **RemoveDirectory()** function.

5. To close an open file, use the **CloseHandle()** function.

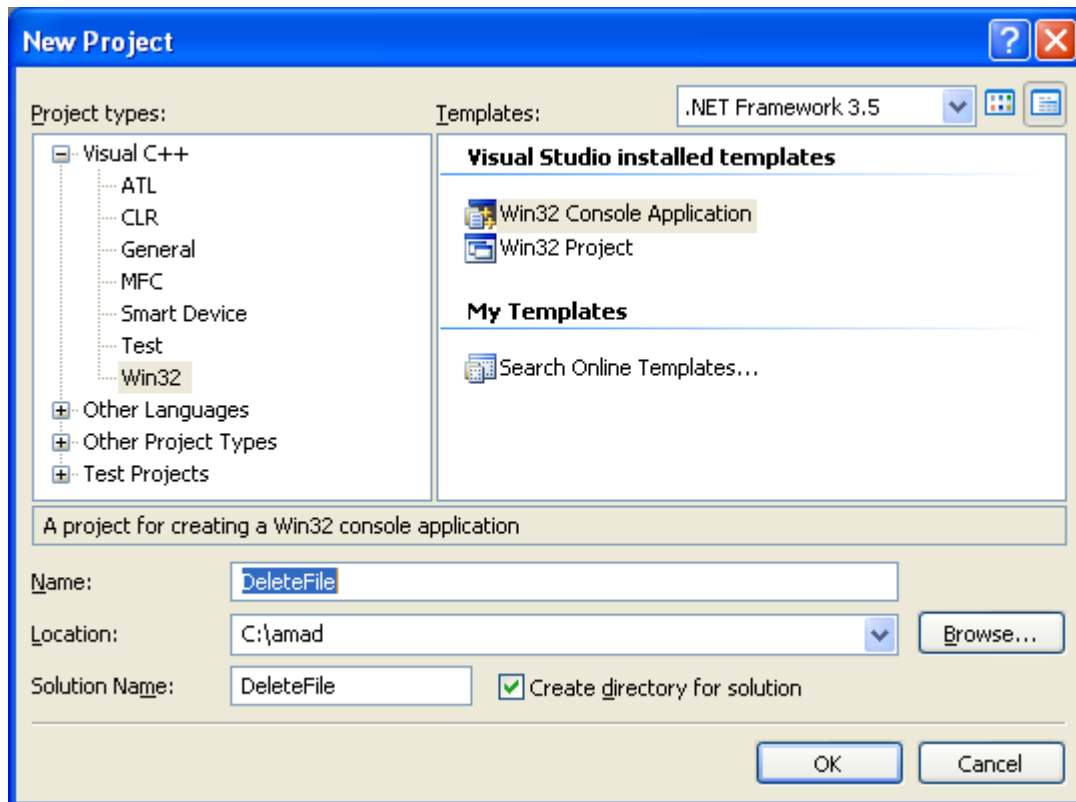
If you set up a directory with all access except delete and delete child, and the access control lists (ACL) of new files are inherited, then you can create a file without being able to delete it. However, then you can create a file, and then get all the access you request on the handle that is returned to you at the time you create the file. If you request delete permission at the time you create a file, you can delete or rename the file with that handle, but not with any other handle. The **DeleteFile()** function fails if an application attempts to delete a file that is open for normal I/O or as a memory-mapped file. The **DeleteFile()** function marks a file for deletion on close. Therefore, the file deletion does not occur until the last handle to the file is closed. Subsequent calls to **CreateFile()** to open the file fail with **ERROR_ACCESS_DENIED**.

Symbolic link behavior

If the path points to a symbolic link, the symbolic link is deleted, not the target. To delete a target, you must call **CreateFile()** and specify **FILE_FLAG_DELETE_ON_CLOSE**.

Deleting a File Program Example

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



Then, add the source file and give it a suitable name.

Next, add the following source code.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMWRITES 10
#define TESTSTRLEN 11

const char TestData[NUMWRITES][TESTSTRLEN] =
{
    "TestData0\n",
    "TestData1\n",
    "TestData2\n",
    "TestData3\n",
    "TestData4\n",
    "TestData5\n",
    "TestData6\n",
    "TestData7\n",
    "TestData8\n",
    "TestData9\n"
};

int main(int argc, char *argv[])
{
    BOOL fSuccess = FALSE;

    // Create the file, open for both read and write.
    HANDLE hFile = CreateFile(TEXT("datafile.txt"),
        GENERIC_READ | GENERIC_WRITE,
        0, // open with exclusive access
        NULL, // no security attributes
        CREATE_NEW, // creating a new temp file
        0, // not overlapped index/0
        NULL);

    if (hFile == INVALID_HANDLE_VALUE)
    {
        // Handle the error.
        printf("CreateFile() failed, error %d\n", GetLastError());
        return (1);
    }

    printf("CreateFile() is OK!\n");

    // Write some data to the file.
    DWORD dwNumBytesWritten = 0;

    for (int i=0; i<NUMWRITES; i++)
    {
        fSuccess = WriteFile(hFile,
            TestData[i],
            TESTSTRLEN,
            &dwNumBytesWritten,
            NULL); // sync operation.
    }
}
```

```
if (!fSuccess)
{
    // Handle the error.
    printf("WriteFile() failed, error %d\n", GetLastError());
    return (2);
}
else
    printf("WriteFile() is OK!\n");
}

FlushFileBuffers(hFile);

// Lock the 4th write-section.
// First, set up the Overlapped structure with the file offset
// required by LockFileEx, three lines in to the file.
OVERLAPPED sOverlapped;
sOverlapped.Offset = TESTSTRLEN * 3;
sOverlapped.OffsetHigh = 0;

// Actually lock the file. Specify exclusive access, and fail
// immediately if the lock cannot be obtained.
fSuccess = LockFileEx(hFile,          // exclusive access,
                    LOCKFILE_EXCLUSIVE_LOCK |
                    LOCKFILE_FAIL_IMMEDIATELY,
                    0,                // reserved, must be zero
                    TESTSTRLEN,      // number of bytes to lock
                    0,
                    &sOverlapped); // contains the file offset

if (!fSuccess)
{
    // Handle the error.
    printf ("LockFileEx() failed (%d)\n", GetLastError());
    return (3);
}
else printf("LockFileEx() succeeded\n");

////////////////////////////////////
// Add code that does something interesting to locked section, /
// which should be line 4                                     /
////////////////////////////////////

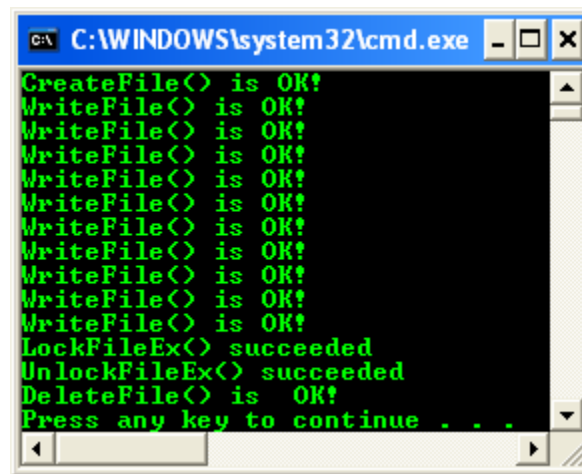
// Unlock the file.
fSuccess = UnlockFileEx(hFile,
                      0,                // reserved, must be zero
                      TESTSTRLEN,      // num. of bytes to unlock
                      0,
                      &sOverlapped); // contains the file offset

if (!fSuccess)
{
    // Handle the error.
    printf ("UnlockFileEx() failed (%d)\n", GetLastError());
    return (4);
}
else printf("UnlockFileEx() succeeded\n");

// Clean up handles, memory, and the created file.
```

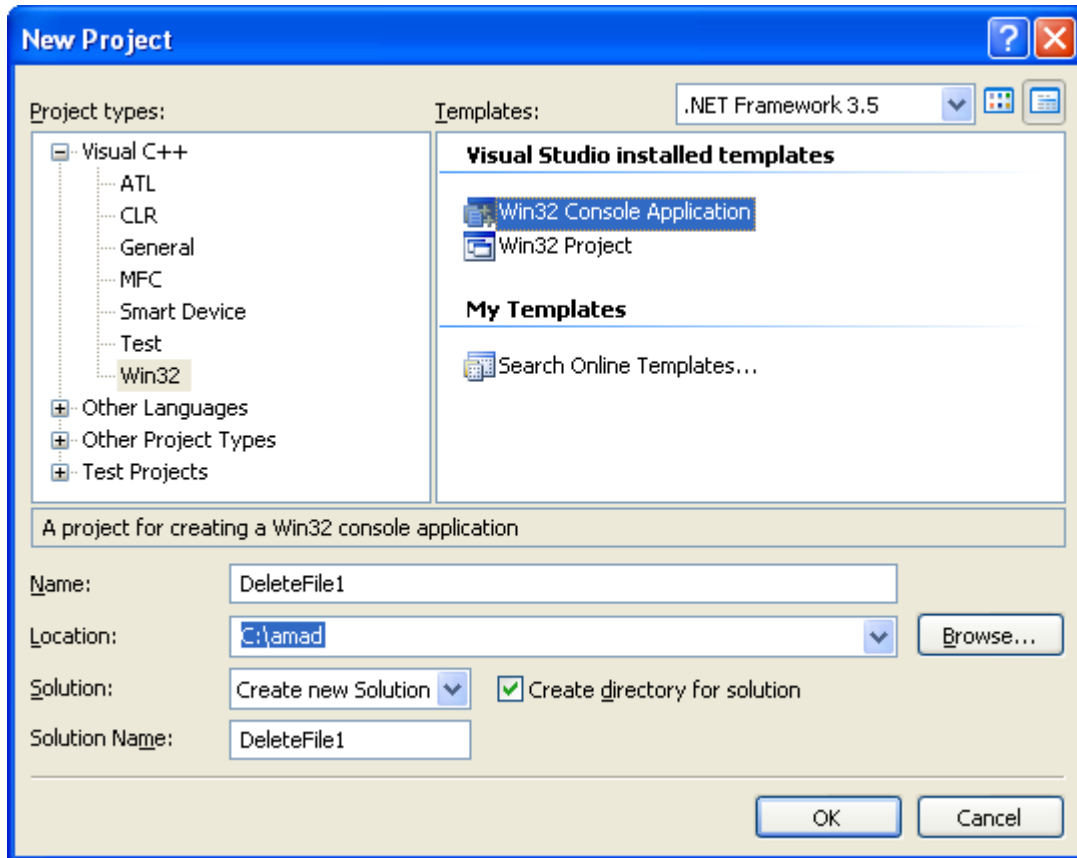
```
fSuccess = CloseHandle(hFile);  
if (!fSuccess)  
{  
    // Handle the error.  
    printf ("CloseHandle failed (%d)\n", GetLastError());  
    return (5);  
}  
  
fSuccess = DeleteFile(TEXT("datafile.txt"));  
if (!fSuccess)  
{  
    // Handle the error.  
    printf ("DeleteFile() failed, error %d\n", GetLastError());  
    return (6);  
}  
else  
    printf ("DeleteFile() is OK!\n");  
return (0);  
}
```

Build and run the project. The following screenshot is a sample output.



Another Deleting File Program Example

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



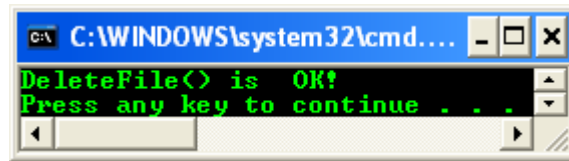
Then, add the source file and give it a suitable name.
Next, add the following source code.

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    BOOL fSuccess = FALSE;

    fSuccess = DeleteFile(TEXT("C:\\testing.txt"));
    if (!fSuccess)
    {
        // Handle the error.
        printf ("DeleteFile() failed, error %d\n", GetLastError());
        return (6);
    }
    else
        printf ("DeleteFile() is OK!\n");
    return (0);
}
```

Build and run the project. The following screenshot is a sample output.



GetDiskFreeSpace() Function

Retrieves information about the specified disk, including the amount of free space on the disk. The GetDiskFreeSpace() function cannot report volume sizes that are greater than 2 gigabytes (GB). To ensure that your application works with large capacity hard drives, use the GetDiskFreeSpaceEx() function. The syntax is:

```
BOOL WINAPI GetDiskFreeSpace (  
    LPCTSTR lpRootPathName,  
    LPDWORD lpSectorsPerCluster,  
    LPDWORD lpBytesPerSector,  
    LPDWORD lpNumberOfFreeClusters,  
    LPDWORD lpTotalNumberOfClusters);
```

Parameters

lpRootPathName [in] - The root directory of the disk for which information is to be returned. If this parameter is NULL, the function uses the root of the current disk. If this parameter is a UNC name, it must include a trailing backslash (for example, \\MyServer\MyShare\). Furthermore, a drive specification must have a trailing backslash (for example, C:\). The calling application must have FILE_LIST_DIRECTORY access rights for this directory.

lpSectorsPerCluster [out] - A pointer to a variable that receives the number of sectors per cluster.

lpBytesPerSector [out] - A pointer to a variable that receives the number of bytes per sector.

lpNumberOfFreeClusters [out] - A pointer to a variable that receives the total number of free clusters on the disk that are available to the user who is associated with the calling thread.

If per-user disk quotas are in use, this value may be less than the total number of free clusters on the disk.

lpTotalNumberOfClusters [out] - A pointer to a variable that receives the total number of clusters on the disk that are available to the user who is associated with the calling thread.

If per-user disk quotas are in use, this value may be less than the total number of clusters on the disk.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To get extended error information, call GetLastError().

Remarks

The `GetDiskFreeSpaceEx()` function lets you avoid some of the arithmetic that is required by the `GetDiskFreeSpace()` function. For symbolic link behavior - If the path points to a symbolic link, the operation is performed on the target.

GetDiskFreeSpaceEx() Function Example

Retrieves information about the amount of space that is available on a disk volume, which is the total amount of space, the total amount of free space, and the total amount of free space available to the user that is associated with the calling thread. The syntax is:

```
BOOL WINAPI GetDiskFreeSpaceEx(  
    LPCTSTR lpDirectoryName,  
    PULARGE_INTEGER lpFreeBytesAvailable,  
    PULARGE_INTEGER lpTotalNumberOfBytes,  
    PULARGE_INTEGER lpTotalNumberOfFreeBytes);
```

Parameters

lpDirectoryName [in, optional] - A directory on the disk. If this parameter is NULL, the function uses the root of the current disk. If this parameter is a UNC name, it must include a trailing backslash, for example, "\\MyServer\MyShare\". This parameter does not have to specify the root directory on a disk. The function accepts any directory on a disk. The calling application must have `FILE_LIST_DIRECTORY` access rights for this directory.

lpFreeBytesAvailable [out, optional] - A pointer to a variable that receives the total number of free bytes on a disk that are available to the user who is associated with the calling thread. This parameter can be NULL. If per-user quotas are being used, this value may be less than the total number of free bytes on a disk.

lpTotalNumberOfBytes [out, optional] - A pointer to a variable that receives the total number of bytes on a disk that are available to the user who is associated with the calling thread. This parameter can be NULL. If per-user quotas are being used, this value may be less than the total number of bytes on a disk. To determine the total number of bytes on a disk or volume, use `IOCTL_DISK_GET_LENGTH_INFO`.

lpTotalNumberOfFreeBytes [out, optional] - A pointer to a variable that receives the total number of free bytes on a disk. This parameter can be NULL.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero (0). To get extended error information, call `GetLastError()`.

Remarks

The values obtained by this function are of the type `ULARGE_INTEGER`. Do not truncate these values to 32 bits. The `GetDiskFreeSpaceEx()` function returns zero (0) for `lpTotalNumberOfFreeBytes` and `lpFreeBytesAvailable` for all CD requests unless the disk is an unwritten CD in a CD-RW drive. Symbolic link behavior, if the path points to a symbolic link, the operation is performed on the target.

The following sample code demonstrates how to use `GetDiskFreeSpaceEx()` and `GetDiskFreeSpace()` on all Windows platforms. Important elements of the code include: How to determine at run time whether `GetDiskFreeSpaceEx()` is present and if not, how to revert to `GetDiskFreeSpace()`. How to use 64-bit math to report the returned sizes for all volumes, even if they are larger than 2 GB.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.

Then, add the source file and give it a suitable name.

Next, add the following source code.

```
/*    Determines the amount of free space available for the caller.
      Runs on Windows 95 retail and later, and on Windows 4.0 and later.
      Uses GetDiskFreeSpaceEx() if available, otherwise reverts to
      GetDiskFreeSpace.

      To determine the amount of available space correctly:

      Use 64-bit math with the return values of both GetDiskFreeSpace()
      and GetDiskFreeSpaceEx() so that you can determine the sizes of
      volumes that are larger than 2GB.

      Programs that need to determine how much free space the current user
      can have (such as whether
      there is enough space to complete an installation)
      have an additional requirement:

      Use the lpFreeBytesAvailableToCaller value from
      GetDiskFreeSpaceEx() rather than lpTotalNumberOfFreeBytes.
      This is because Windows 2000 has disk quota management that
      administrators may use to limit the amount of disk space that users
      may use.
      */

#include <windows.h>
#include <stdio.h>

// typedef
typedef BOOL (WINAPI *P_GDFSE) (LPCTSTR, PULARGE_INTEGER, PULARGE_INTEGER,
PULARGE_INTEGER);

int wmain(int argc, WCHAR **argv)
{
    BOOL fResult;
    WCHAR *pszDrive = NULL, szDrive[4];
    DWORD dwSectPerClust, dwBytesPerSect, dwFreeClusters, dwTotalClusters;
    P_GDFSE pGetDiskFreeSpaceEx = NULL;
```

```
unsigned __int64 i64FreeBytesToCaller, i64TotalBytes, i64FreeBytes;

/* Command line parsing.
If the drive is a drive letter and not a UNC path, append a trailing
backslash to the drive letter and colon. This is required on Windows
95 and 98. */

if (argc != 2)
{
    wprintf(L"usage:  %s <drive|UNC path>\n", argv[0]);
    wprintf(L"\texample:  %s C:\\\n", argv[0]);
    return 1;
}

pszDrive = argv[1];
// Parse the drive
if (pszDrive[1] == ':')
{
    szDrive[0] = pszDrive[0];
    szDrive[1] = ':';
    szDrive[2] = '\\';
    szDrive[3] = '\\0';

    pszDrive = szDrive;
}

/* Use GetDiskFreeSpaceEx() if available; otherwise, use
GetDiskFreeSpace().
Note: Since GetDiskFreeSpaceEx() is not in Windows 95 Retail, we
dynamically link to it and only call it if it is present. We
don't need to call LoadLibrary() on KERNEL32.DLL because it is
already loaded into every Win32 process's address space. */
pGetDiskFreeSpaceEx = (P_GDFSE)GetProcAddress(GetModuleHandle
(L"kernel32.dll"), "GetDiskFreeSpaceExW");
if (pGetDiskFreeSpaceEx)
{
    fResult = pGetDiskFreeSpaceEx((LPCTSTR)pszDrive,
(PULARGE_INTEGER)&i64FreeBytesToCaller,
(PULARGE_INTEGER)&i64TotalBytes,
(PULARGE_INTEGER)&i64FreeBytes);

    if (fResult)
    {
        wprintf(L"\nGetDiskFreeSpaceExW reports:\n\n");
        wprintf(L"Available space to caller = %I64u MB\n",
i64FreeBytesToCaller / (1024*1024));
        wprintf(L"Total space                = %I64u MB\n",
i64TotalBytes / (1024*1024));
        wprintf(L"Free space on drive         = %I64u MB\n",
i64FreeBytes / (1024*1024));
    }
}
else
{
    fResult = GetDiskFreeSpace((LPCWSTR)pszDrive, &dwSectPerClust,
&dwBytesPerSect, &dwFreeClusters, &dwTotalClusters);
```

```

        if (fResult)
        {
            /* force 64-bit math */
            i64TotalBytes = (__int64)dwTotalClusters * dwSectPerClust *
dwBytesPerSect;
            i64FreeBytes = (__int64)dwFreeClusters * dwSectPerClust *
dwBytesPerSect;

            wprintf(L"GetDiskFreeSpace reports\n");
            wprintf(L"Free space = %I64u MB\n", i64FreeBytes /
(1024*1024));
            wprintf(L"Total space = %I64u MB\n", i64TotalBytes /
(1024*1024));
        }

        if (!fResult)
            wprintf(L"error: %lu: could not get free space for \"%s\"\n",
GetLastError(), argv[1]);

        return 0;
    }

```

Build and run the project. The following screenshot is a sample output.

```

C:\WINDOWS\system32\cmd.exe
C:\amad\GetDiskFreeSpaceEx\Debug>GetDiskFreeSpaceEx
usage:  GetDiskFreeSpaceEx <drive !UNC path>
example: GetDiskFreeSpaceEx C:\

C:\amad\GetDiskFreeSpaceEx\Debug>GetDiskFreeSpaceEx C:\

GetDiskFreeSpaceExW reports:
Available space to caller = 65395 MB
Total space                = 149981 MB
Free space on drive       = 65395 MB

C:\amad\GetDiskFreeSpaceEx\Debug>_

```

64-bit Integer Math

Microsoft Visual C++ versions 4.0 and later support a 64-bit integer type called `__int64`. The compiler generates code to do the 64-bit math because the Intel x86 family of microprocessors supports 8-bit, 16-bit, and 32-bit integer math, but not 64-bit integer math.

To perform a 64-bit integer multiply, one of the arguments must be 64-bit; the other can be either 32-bit or 64-bit. When functions such as `GetDiskFreeSpace()` return only 32-bit integer values that will be multiplied together but you need to have a 64-bit integer to contain the product, cast one of the values to an `__int64` as follows:

```
i64TotalBytes = (__int64)dwTotalClusters * dwSectPerClust *
dwBytesPerSect;
```

The first multiply is of a 64-bit integer with a 32-bit integer; the result is a 64-bit integer, which is then multiplied by another 32-bit integer, resulting in a 64-bit product. Many Win32 API functions that take a 64-bit quantity do so as two separate 32-bit quantities. Others, such as QueryPerformanceCounter(), take a single 64-bit quantity. The LARGE_INTEGER union type defined in the Platform SDK WINNT.H header file manages these differing ways to handle 64-bit integers. There's a corresponding ULARGE_INTEGER for unsigned large integers. The LARGE_INTEGER union consists of a 64-bit __int64 member (QuadPart) and two 32-bit values (HighPart and LowPart). Each of the two 32-bit values is one-half of the 64-bit integer. The HighPart member is a signed long integer, while the LowPart is an unsigned long integer. Since the LARGE_INTEGER.QuadPart member is an __int64, you can easily intermix LARGE_INTEGER variables with __int64 variables. To perform integer math with LARGE_INTEGER variables, always use the QuadPart member to treat the LARGE_INTEGER as the single 64-bit value it represents. Use the 32-bit HighPart and LowPart members when you must pass a LARGE_INTEGER to a function in two 32-bit parts. An equivalent to the above example using LARGE_INTEGERs instead of __int64 variables is:

```
liTotalBytes.QuadPart = (__int64)dwTotalClusters *
dwSectPerClust * dwBytesPerSect;
```

Disk Management Interfaces

Component Object Model (COM) programming provides a rich set of standards for implementing and using objects and for inter-object communication. Interfaces are used in COM programming and for every interface there are many methods that can be used. The following interfaces are used in disk management:

Interface	Description
IDiskQuotaControl()	Controls the disk quota facilities of a single NTFS file system volume. The client can query and set volume-specific quota attributes through IDiskQuotaControl() . The client can also enumerate all per-user quota entries on the volume. A client instantiates this interface by calling the CoCreateInstance() function using the class identifier CLSID_DiskQuotaControl.
IDiskQuotaEvents()	A client must implement the IDiskQuotaEvents() interface as an event sink that receives the quota-related event notifications. Its methods are called by the system whenever significant quota events have occurred. Currently, the only event supported is the asynchronous resolution of user account name information.
IDiskQuotaUser()	Represents a single user quota entry in the volume quota

	information file. Through this interface, you can query and modify user-specific quota information on an NTFS file system volume. This interface is instantiated by using IEnumDiskQuotaUsers() , IDiskQuotaControl::FindUserSid() , IDiskQuotaControl::FindUserName() , IDiskQuotaControl::AddUserSid() , or IDiskQuotaControl::AddUserName() .
IDiskQuotaUserBatch()	Adds multiple quota user objects to a container that is then submitted for update in a single call. This reduces the number of calls to the underlying file system, improving update efficiency when a large number of user objects must be updated. This interface is instantiated by using the IDiskQuotaControl::CreateUserBatch() method
IEnumDiskQuotaUsers()	Enumerates user quota entries on the volume. This interface is instantiated by using the IDiskQuotaControl::CreateEnumUsers() method.

Disk Management Structures

The following list identifies the structures that are used in disk management:

Structure	Description
CREATE_DISK	Contains information that the IOCTL_DISK_CREATE_DISK control code uses to initialize GUID partition table (GPT), master boot record (MBR), or raw disks.
CREATE_DISK_GPT	Contains information used by the IOCTL_DISK_CREATE_DISK control code to initialize GUID partition table (GPT) disks.
CREATE_DISK_MBR	Contains information that the IOCTL_DISK_CREATE_DISK control code uses to initialize master boot record (MBR) disks.
DISK_CACHE_INFORMATION	Provides information about the disk cache. This structure is used by the IOCTL_DISK_GET_CACHE_INFORMATION and IOCTL_DISK_SET_CACHE_INFORMATION control codes.
DISK_DETECTION_INFO	Contains detected drive parameters.
DISK_EX_INT13_INFO	Contains extended Int13 drive parameters.

DISK_EXTENT	Represents a disk extent.
DISK_GEOMETRY	Describes the geometry of disk devices and media.
DISK_GEOMETRY_EX	Describes the extended geometry of disk devices and media.
DISK_GROW_PARTITION	Contains information used to increase the size of a partition. This structure is used by the IOCTL_DISK_GROW_PARTITION control code.
DISK_INT13_INFO	Contains standard Int13 drive geometry parameters.
DISK_PARTITION_INFO	Contains the disk partition information.
DISK_PERFORMANCE	Provides disk performance information. It is used by the IOCTL_DISK_PERFORMANCE control code.
DISKQUOTA_USER_INFORMATION	Represents the per-user quota information.
DRIVE_LAYOUT_INFORMATION_EX	Contains extended information about a drive's partitions.
DRIVE_LAYOUT_INFORMATION_GPT	Contains information about a drive's GUID partition table (GPT) partitions.
DRIVE_LAYOUT_INFORMATION_MBR	Provides information about a drive's master boot record (MBR) partitions.
FORMAT_PARAMETERS	- Contains information used in formatting a contiguous set of disk tracks. It is used by the IOCTL_DISK_FORMAT_TRACKS control code.
FORMAT_EX_PARAMETERS	- Contains information used in formatting a contiguous set of disk tracks. It is used by the IOCTL_DISK_FORMAT_TRACKS_EX control code.
GET_LENGTH_INFORMATION	- Contains disk, volume, or partition length information used by the IOCTL_DISK_GET_LENGTH_INFO control code.
PARTITION_INFORMATION_EX	- Contains partition information for standard AT-style master boot record (MBR) and Extensible Firmware Interface (EFI) disks.
PARTITION_INFORMATION_GPT	- Contains GUID partition table (GPT) partition information.
PARTITION_INFORMATION_MBR	- Contains partition information specific to master boot record (MBR) disks.
REASSIGN_BLOCKS	- Contains disk block reassignment data. This is a variable length structure where the last member is an array of block numbers to be reassigned. It is used by the

	IOCTL_DISK_REASSIGN_BLOCKS control code.
VERIFY_INFORMATION	- Contains information used to verify a disk extent. It is the output buffer for the IOCTL_DISK_VERIFY control code.

The following list identifies the device input and output structures that are obsolete:

1. DRIVE_LAYOUT_INFORMATION
2. PARTITION_INFORMATION
3. SET_PARTITION_INFORMATION

Disk Partition Types

The following table identifies the valid partition types that are used by disk drivers.

Constant/value	Description
PARTITION_ENTRY_UNUSED (0x00)	An unused entry partition.
PARTITION_EXTENDED (0x05)	An extended partition.
PARTITION_FAT_12 (0x01)	A FAT12 file system partition.
PARTITION_FAT_16 (0x04)	A FAT16 file system partition.
PARTITION_FAT32 (0x0B)	A FAT32 file system partition.
PARTITION_IFS (0x07)	An IFS partition.
PARTITION_LDM (0x42)	A logical disk manager (LDM) partition.
PARTITION_NTFT (0x80)	An NTFT partition.
VALID_NTFT (0xC0)	A valid NTFT partition. The high bit of a partition type code indicates that a partition is part of an NTFT mirror or striped array.

There are several macros that can help you detect the partition type which are IsContainerPartition(), IsFTPPartition(), and IsRecognizedPartition().