The Windows File Management

What do we have for this session?

Brief Intro
File Attributes
Files and Clusters
Creating, Deleting, and Maintaining Files
File Names, Paths, and Namespaces
Basic Naming Conventions
Path Names and Namespaces
Maximum Path Length
Relative Paths
Short and Long File Names and Paths
Creating and Opening Files
CreateFile() Scenarios
File Attributes and Directories
Moving and Replacing Files
Closing and Deleting Files
Defragmenting Files
Minimizing interactions between defragmentation and shadow copies
Limitations under Windows 2000
Retrieving File Type Information
Determining the Size of a File
Searching for One or More Files
Setting and Getting the Timestamp of a File
File Type, Size and Timestamp Program Example
Determining the Current Character Set Code Page
AreFileApisANSI() Program Example
Reading From and Writing to Files
Positioning a File Pointer
Reading From or Writing To Files Using a Scatter-Gather Scheme
Flushing System-Buffered I/O Data to Disk
Truncating or Extending Files
File and Directory Linking
File Compression and Decompression
The NTFS File System File Compression
Compression Attribute
Compression State
Obtaining the Size of a Compressed File
Compressed File (Attributes) Program Example
File Compression and Decompression Libraries
Decompressing a Single File
Decompressing Multiple Files
Reading from Compressed Files

Using Windows Compressed Functions Program Example
Cabinets
File Encryption
Handling Encrypted Files and Directories
Encrypted Files and User Keys
Backup and Restore of Encrypted Files
Adding Users to an Encrypted File Program Example
Input and Output (I/O)
I/O Concepts
File Buffering
Alignment and File Access Requirements
File Caching
Synchronous and Asynchronous I/O
Synchronous and Asynchronous I/O Considerations
Canceling Pending I/O Operations
Cancellation Considerations
Operations That Cannot Be Canceled
Canceling Asynchronous I/O
Canceling Synchronous I/O
Alertable I/O
I/O Completion Ports
How I/O Completion Ports Work
Threads and Concurrency
Supported I/O Functions
Network I/O Concepts
Description of a Network I/O Operation
Microsoft SMB Protocol and CIFS Protocol Overview
Opportunistic Locks
Sparse Files
Sparse File Operations
Obtaining the Size of a Sparse File
Sparse Files and Disk Quotas
Symbolic Links
Symbolic Link Effects on File Systems Functions
Programming Considerations
Creating Symbolic Links
Example of an Absolute Symbolic Link
Example of a Relative Symbolic Links
File Management Program Examples
- **Appending One File to Another File Program Example**
- **Creating and Using a Temporary File Program Example**
- **Locking and Unlocking Byte Ranges in Files Program Example**
- **Opening a File for Reading or Writing**
- **Open a File for Writing Program Example**
- **Open a File for Reading Program Example**
- **Retrieving and Changing File Attributes**
- **Testing for the End of a File**
- **Using Streams**

Alternate File Stream
File Management Reference
File Management Control Codes
File Management Enumerations
File Management Functions
File Management Structures

Brief Intro

A file object provides **a representation of a resource (either a physical device or a resource located on a physical device) that can be managed by the I/O system**. Like other objects, they enable **sharing of the resource**, they **have names**, they **are protected by object-based security**, and they **support synchronization**. The I/O system also enables reading from or writing to the resource.

File Attributes

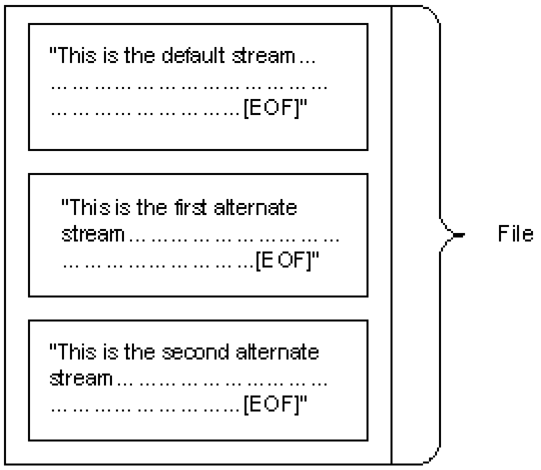
File attributes are **metadata** values stored by the file system on disk and are used by the system and are available to developers via various file I/O APIs. The following table lists file attribute constant names and values with descriptions.

Name	Value (Dec)	Value (Hex)	Description
FILE_ATTRIBUTE_ARCHIVE	32	0x0020	A file or directory that is an archive file or directory. Applications typically use this attribute to mark files for backup or removal.
FILE_ATTRIBUTE_COMPRESSED	2048	0x0800	A file or directory that is compressed. For a file, all of the data in the file is compressed. For a directory, compression is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_DEVICE	64	0x0040	This value is reserved for system use.
FILE_ATTRIBUTE_DIRECTORY	16	0x0010	The handle that identifies a directory.
FILE_ATTRIBUTE_ENCRYPTED	16384	0x4000	A file or directory that is encrypted. For a file, all data streams in the file are encrypted. For a directory, encryption is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_HIDDEN	2	0x0002	The file or directory is hidden. It is not included in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL	128	0x0080	A file that does not have other attributes set. This attribute is valid only when used alone.
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED	8192	0x2000	The file or directory is not to be indexed by the content indexing service.
FILE_ATTRIBUTE_OFFLINE	4096	0x1000	The data of a file is not available immediately. This attribute indicates that the file data is physically moved to offline storage. This

			attribute is used by Remote Storage, which is the hierarchical storage management software. Applications should not arbitrarily change this attribute.
FILE_ATTRIBUTE_READONLY	1	0x0001	A file that is read-only. Applications can read the file, but cannot write to it or delete it. This attribute is not honored on directories.
FILE_ATTRIBUTE_REPARSE_POINT	1024	0x0400	A file or directory that has an associated reparse point, or a file that is a symbolic link.
FILE_ATTRIBUTE_SPARSE_FILE	512	0x0200	A file that is a sparse file.
FILE_ATTRIBUTE_SYSTEM	4	0x0004	A file or directory that the operating system uses a part of, or uses exclusively.
FILE_ATTRIBUTE_TEMPORARY	256	0x0100	A file that is being used for temporary storage. File systems avoid writing data back to mass storage if sufficient cache memory is available, because typically, an application deletes a temporary file after the handle is closed. In that scenario, the system can entirely avoid writing the data. Otherwise, the data is written after the handle is closed.
FILE_ATTRIBUTE_VIRTUAL	65536	0x10000	This value is reserved for system use.

Files and Clusters

A file is **a unit of data in the file system** that a user can access and manage. A file must have a unique name in its directory. It consists of **one or more streams of bytes that hold a set of related data**, plus **a set of attributes** (also called properties) that describe the file or the data within the file. The creation time of a file is an example of a file attribute.
When a file is created, one **unnamed default stream is created to store all data written to the file while it is open**. You **can also create additional streams within the file**. These additional streams are referred to as **alternate streams**. The following figure depicts a file with the default stream and two alternate streams.


"This is the default stream... ... ...[EOF]"
"This is the first alternate stream... ... ...[EOF]"
"This is the second alternate stream... ... ...[EOF]"
File

File attributes are not stored in the data streams with the file data, but are stored elsewhere and managed by the operating system.
All file system data, including the system bootstrap code and directories, are stored by the NTFS file system in files. Other file systems store this information in disk regions external to the file system. An advantage of storing this information in files is that Windows can locate, access, and maintain the information easily. Other advantages are that each of these files may be protected by a security descriptor and, in the case of partial disk corruption, they may be quickly relocated to a safer part of the disk.
The fundamental storage unit of all supported file systems is a **cluster**, **which is a group of sectors**. This allows the file system to optimize the administration of disk data independently of the disk sector size set by the hardware disk controller. If the disk to be administered is large and large amounts of data are moved and organized in a single operation, the administrator can adjust the cluster size to accommodate this. Windows manages files through file objects, file handles, and file pointers.

Creating, Deleting, and Maintaining Files

File Names, Paths, and Namespaces

All file systems follow the same general naming conventions for an individual file: **a base file name and an optional extension**, **separated by a period**. However, each file system, such as NTFS and FAT, can have specific and differing rules about the formation of the individual components in a directory or file name. Character count limitations can also be different and can vary depending on the path name prefix format used. This is further complicated by support for backward compatibility mechanisms. Any Windows (Win32) application developer should be aware of these limitations and differences and know which file and path names are valid.
For example, the older MS-DOS FAT file system supports a maximum of 8 characters for the base file name and 3 characters for the extension, for a total of 12 characters including the dot separator. This is commonly known as an 8.3 file name. The Windows FAT and NTFS file systems are not limited to 8.3 file names, because they have long file name support, but they still support the 8.3 version of long file names.
Be aware that the term directory simply refers to a special type of file as far as the file system is concerned; therefore in certain contexts some reference material will use the general term file to encompass both concepts of directories and data files as such. Because of the higher level nature of this topic, it will use the term file to refer to actual data files only.
Some file systems, such as NTFS, support linked files and directories, which also follow file naming conventions and rules just as a regular file or directory would.

Basic Naming Conventions

The following fundamental rules enable applications to create and process valid names for files and directories, regardless of the file system:

1. Use a period to separate the base file name from the extension in the name of a directory or file.

2. Use a backslash (\) to separate the components of a path. The backslash divides the file name from the path to it, and one directory name from another directory name in a path. For additional details about what a path is, see the Path Names and Namespaces section below.
3. Use a backslash as required as part of volume names, for example, the "C:\" in "C:\path\file" or the "\\server\share" in "\\server\share\path\file" for Universal Naming Convention (UNC) names. You cannot use a backslash in the actual file or directory name components because it separates the names into components.
4. Use almost any character in the current code page for a name, including Unicode characters and characters in the extended character set (128 - 255), except for the following:

   - The following reserved characters are not allowed: < > : " / \ | ? *
   - Characters whose integer representations are in the range from zero through 31 are not allowed.
   - Any other character that the target file system does not allow.

5. Use a period as a directory component in a path to represent the current directory, for example ".\tmp.txt".
6. Use two consecutive periods (..) as a directory component in a path to represent the parent of the current directory, for example "..\tmp.txt".
7. Do not use the following reserved device names for the name of a file:

   CON, PRN, AUX, NUL, COM1, COM2, COM3, COM4, COM5, COM6, COM7, COM8, COM9, LPT1, LPT2, LPT3, LPT4, LPT5, LPT6, LPT7, LPT8, and LPT9
   Also avoid these names followed immediately by an extension; for example, NUL.txt is not recommended.

8. Do not assume case sensitivity. For example, consider the names OSCAR, Oscar, and oscar to be the same, even though some file systems (such as a POSIX-compliant file system) may consider them as different. Note that NTFS supports POSIX semantics for case sensitivity but this is not the default behavior.
9. Do not end a file or directory name with a trailing space or a period. Although the underlying file system may support such names, the operating system does not. However, it is acceptable to start a name with a period.

Path Names and Namespaces

The path to a specified file consists of one or more components, separated by special characters, with each component usually being a directory name or file name, with some notable exceptions discussed below. It is often critical to the system's interpretation of a path what the beginning of the path (the prefix) looks like and what special characters are used in which position within the path, including the last character. If a component of a path is a file name, it must be the last component. Each component of a path will also be constrained by the maximum length specified for a particular file system. In general, these rules fall into two categories: short and long. Note that directory names are stored by the file system as a special type of file, but naming rules for files also apply to directory names. A path is simply the string representation of the hierarchy between all of the directories that exist for a particular file or directory name.
Any discussion of path names needs to include the concept of a namespace in Windows. There are two main categories of namespace conventions used in the Win32 APIs, commonly referred to as

the NT namespace and the Win32 namespace. The NT namespace was designed to be the lowest level namespace on which other subsystems and namespaces could exist, including the Win32 subsystem and, by extension, the Win32 file and device namespaces. POSIX is another example of a subsystem in Windows that is built on top of the NT namespace. Early versions of Windows also defined several predefined, or reserved, names for certain special devices such as communications (serial and parallel) ports and the default display console as part of what is now called the NT device namespace, and are still supported in current versions of Windows for backward compatibility.
To sort out some of this, the following items are different examples of Win32 namespace prefixing and conventions, and summarize how they are used. Note that these examples are intended for use with the Win32 API functions and do not all necessarily work with Windows shell applications such as Windows Explorer.
The "\\**?**\" prefix tells the Win32 APIs to **disable all string parsing and to send this string straight to the file system**. For example, if the file system supports large paths and file names, you can exceed the MAX_PATH limits that are otherwise enforced by the Win32 APIs. This also allows you to turn off automatic expansion of "**..**" and "**.**" in the path names. Many but not all file APIs support "\\?\"; you should look at the reference topic for each API to be sure.
The "\\.\" prefix will access the **device namespace** instead of the **file namespace**. This is how you access physical disks and volumes directly, without going through the file system, if the API supports this type of access. You can access many other devices this way (using the CreateFile() and DefineDosDevice() functions, for example). Most APIs won't support "\\**.**\", only those that are designed to work with the device namespace.
For example, if you want to open the system's serial communications port 1, you can use either "\\.\COM1" or "COM1" in the call to the CreateFile() function. This works because COM1-COM9 are part of the reserved names in the NT file namespace as previously mentioned. But if you have a 100 port serial expansion board and want to open COM56, you need to open it using "\\.\COM56". This works because "\\.\" goes to the device namespace, and there is no predefined NT namespace for COM56. Another example of this is using the CreateFile() function on "\\.\PhysicalDiskX" or "\\.\CdRom1" allow you to access those devices, bypassing the file system. It just happens that the device driver that implements the name "C:\" has its own namespace that is the file system. APIs that go through the CreateFile() function should work because CreateFile() is the same API to open files and devices. If you're working with Win32 functions, you should use only "\\.\" to access devices and not files.
There are also APIs that allow the use of the NT namespace convention, but the Windows Object Manager makes that unnecessary in most cases. To illustrate, it is useful to browse the Windows namespaces in the system object browser using the Windows Sysinternals WinObj tool. When you run this tool, what you see is the NT namespace rooted at "\". The subdirectory "**Global??**" is where the Win32 namespace resides. Named device objects reside in the NT namespace within the "Device" subdirectory. Here you may also find **Serial0** and **Serial1**, the device objects representing the two COM ports if present on your system. The device object representing a volume would be something like "**HarddiskVolume1**", although the numeric suffix may vary. The name "**DR0**" under subdirectory "**Harddisk0**" would be the device object representing a disk, and so on. To make these device objects accessible by Win32 applications, the device drivers create a symbolic link (symlink) in the Win32 namespace to their respective device objects. For example, COM0 and COM1 under the "Global??" subdirectory are simply symlinks to **Serial0** and **Serial1**, "**C:**" is a symlink to **HarddiskVolume1**, "**Physicaldrive0**" is a symlink to **DR0**, and so on. Without a symlink, a specified device "Xxx" will not be available to any Win32 application using Win32 namespace conventions as described previously. However, a handle could be opened to that device using any APIs that support the NT namespace absolute path of the format "**\Device\Xxx**".

With the addition of multi-user support via Terminal Services and virtual machines, it has further become necessary to virtualize the system-wide root device within the Win32 namespace. This was accomplished by adding the symlink named "**GLOBALROOT**" to the Win32 namespace, which you can see in the "Global??" subdirectory of the WinObj browser tool previously discussed, and can access via the path "\\**?\GLOBALROOT**". This prefix ensures that the path following it looks in the true root path of the system object manager and not a session-dependent path.

Maximum Path Length

In the Windows API (with some exceptions discussed in the following paragraphs), the maximum length for a path is MAX_PATH, which is defined as 260 characters. A local path is structured in the following order: drive letter, colon, backslash, components separated by backslashes, and a terminating null character. For example, the maximum path on drive D is:

```
"D:\<some 256 character path string><NUL>"
```

Where "<NUL>" represents the invisible terminating null character for the current system codepage. (The characters < > are used here for visual clarity and cannot be part of a valid path string.)
File I/O functions in the Windows API convert "/" to "\" as part of converting the name to an NT-style name, except when using the "\\?\" prefix as detailed in the following sections.
The Windows API has many functions that also have Unicode versions to permit an extended-length path for a maximum total path length of 32,767 characters. This type of path is composed of components separated by backslashes, each up to the value returned in the lpMaximumComponentLength parameter of the GetVolumeInformation() function (this value is commonly 255 characters). To specify an extended-length path, use the "\\?\" prefix. For example, "\\**?\D:\<very long path>**". (The characters < > are used here for visual clarity and cannot be part of a valid path string.)
The maximum path of 32,767 characters is approximate, because the "\\?\" prefix may be expanded to a longer string by the system at run time, and this expansion applies to the total length.
The "\\?\" prefix can also be used with paths constructed according to the universal naming convention (UNC). To specify such a path using UNC, use the "\\**?\UNC**\" prefix. For example, "\\**?\UNC\server\share**", where "server" is the name of the machine and "share" is the name of the shared folder. These prefixes are not used as part of the path itself. They indicate that the path should be passed to the system with minimal modification, which means that you cannot use forward slashes to represent path separators, or a period to represent the current directory. Also, you cannot use the "\\?\" prefix with a relative path, therefore relative paths are limited to MAX_PATH characters as previously stated for paths not using the "\\?\" prefix.
There is no need to perform any UNICODE normalization on path and file name strings for use by the Win32 file I/O API functions because the filesystem treats path and file names as an opaque sequence of WCHARs. Any normalization your application requires should be performed with this in mind, external to any calls to related Win32 file I/O API functions.
When using an API to create a directory, the specified path cannot be so long that you cannot append an 8.3 file name (that is, the directory name cannot exceed MAX_PATH minus 12).
The shell and the file system have different requirements. It is possible to create a path with the Windows API that the shell user interface might not be able to handle.

Relative Paths

For functions that manipulate files, the file names can be relative to the current directory. A file name is relative to the current directory if it does not begin with one of the following:

1. A UNC name of any format.
2. A disk designator with a backslash, for example "C:\".
3. A backslash, for example, "\directory").

If the file name begins with a disk designator with a backslash, it is a fully qualified path (for example, "C:\tmp"). If a file name begins with only a disk designator but not the backslash after the colon, it is interpreted as a relative path to the current directory on the drive with the specified letter. Examples of this format are as follows:

1. "C:tmp.txt" refers to a file in the current directory on drive C.
2. "C:tempdir\tmp.txt" refers to a file in a subdirectory to the current directory on drive C.

A path is also said to be relative if it contains "double-dots"; that is, two periods together in one component of the path. This special specifier is used to denote the directory above the current directory, otherwise known as the "parent directory". Examples of this format are as follows:

1. "..\tmp.txt" specifies a file named tmp.txt located in the parent of the current directory.
2. "..\..\tmp.txt" specifies a file that is two directories above the current directory.
3. "..\tempdir\tmp.txt" specifies a file named tmp.txt located in a directory named tempdir that is a peer directory to the current directory.

Relative paths can combine both example types, for example "C:..\tmp.txt". This is useful because, although the system keeps track of the current drive along with the current directory of that drive, it also keeps track of the current directories of all of the different drive letters if your system has more than one. As stated previously, you cannot use the "\\?\" prefix with a relative path because it is considered a form of UNC naming.

Short and Long File Names and Paths

Typically, Windows stores the long file names on disk as special directory entries, which can be disabled systemwide for performance reasons depending on the particular file system. When you create a long file name, Windows may also create a short MS-DOS (8.3) form of the name, called the **8.3 alias**, and store it on disk. Starting with **Windows 7 and Windows Server 2008 R**2, this can also be disabled for a specified volume. On many file systems, a short file name contains a tilde (~) character within each component when it is too long to comply with 8.3 naming rules, as previously discussed.
Take note that not all file systems follow this convention, and systems can be configured to disable 8.3 alias generation even if they normally support it. Therefore, do not make the assumption that the

8.3 alias already exists. To request 8.3 file names, long file names, or the full path of a file from the system, consider the following options:

1. To get an 8.3 file name that has a long file name, use GetShortPathName().
2. To get the long file name that has a short name, use GetLongPathName().
3. To get the full path of a file, use GetFullPathName().

On newer file systems, such as NTFS, ex-FAT, UDFS, and FAT32, Windows stores the long file names on disk in Unicode, which means that the original long file name is always preserved. This is true even if a long file name contains extended characters and regardless of the code page that is active during a disk read or write operation. The case of the file name is preserved, even when the file system is not case-sensitive.
Files using long file names can be copied between NTFS file system partitions and Windows FAT file system partitions without losing any file name information. This may not be true for MS-DOS FAT and some types of **CDFS** (CD-ROM) file systems, depending on the actual file name. In this case, the short file name is substituted if possible.

Creating and Opening Files

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.
The operating system assigns a unique identifier, called a handle, to each file that is opened or created using CreateFile(). An application can use this handle with functions that read from, write to, and describe the file. It is valid until all references to that handle are closed. When an application starts, it inherits all open handles from the process that started it if the handles were created as inheritable.
An application should check the value of the handle returned by CreateFile() before attempting to use the handle to access the file. If an error occurs, the handle value will be INVALID_HANDLE_VALUE and the application can use the GetLastError() function for extended error information.
When an application uses CreateFile(), it must use the dwDesiredAccess parameter to specify whether it intends to read from the file, write to the file, both read and write, or neither. This is known as requesting an access mode. The application must also use the dwCreationDisposition parameter to specify what action to take if the file already exists, known as the creation disposition. For example, an application can call CreateFile() with dwCreationDisposition set to CREATE_ALWAYS to always create a new file, even if a file of the same name already exists (thus overwriting the existing file). Whether this succeeds or not depends on factors such as the previous file's attributes and security settings (see the following sections for more information).
An application also uses CreateFile() to specify whether it wants to share the file for reading, writing, both, or neither. This is known as the sharing mode. An open file that is not shared (dwShareMode set to zero) cannot be opened again, either by the application that opened it or by another application, until its handle has been closed. This is also referred to as exclusive access.
When a process uses CreateFile() to attempt to open a file that has already been opened in a sharing mode (dwShareMode set to a valid non-zero value), the system compares the requested access and sharing modes to those specified when the file was opened. If you specify an access or sharing mode that conflicts with the modes specified in the previous call, CreateFile() fails.

The following table illustrates the valid combinations of two calls to CreateFile using various access modes and sharing modes (dwDesiredAccess, dwShareMode respectively). It does not matter in which order the CreateFile() calls are made. However, any subsequent file I/O operations on each file handle will still be constrained by the current access and sharing modes associated with that particular file handle.

First call to CreateFile()	**Valid second calls to CreateFile()**
GENERIC_READ, FILE_SHARE_READ	GENERIC_READ, FILE_SHARE_READ GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE
GENERIC_READ, FILE_SHARE_WRITE	GENERIC_WRITE, FILE_SHARE_READ GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE
GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE	GENERIC_READ, FILE_SHARE_READ GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE GENERIC_WRITE, FILE_SHARE_READ GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE
GENERIC_WRITE, FILE_SHARE_READ	GENERIC_READ, FILE_SHARE_WRITE GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE
GENERIC_WRITE, FILE_SHARE_WRITE	GENERIC_WRITE, FILE_SHARE_WRITE GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE
GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE	GENERIC_READ, FILE_SHARE_WRITE GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE GENERIC_WRITE, FILE_SHARE_WRITE GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE GENERIC_READ | GENERIC_WRITE, FILE_SHARE_WRITE GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE
GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ	GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE
GENERIC_READ | GENERIC_WRITE, FILE_SHARE_WRITE	GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE
GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE	GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE GENERIC_READ | GENERIC_WRITE,

	FILE_SHARE_READ | FILE_SHARE_WRITE

In addition to the standard file attributes, you can also specify security attributes by including a pointer to a SECURITY_ATTRIBUTES structure as the fourth parameter of CreateFile(). However, the underlying file system must support security for this to have any effect (for example, the NTFS file system supports it but the various FAT file systems do not).
An application creating a new file can supply an optional handle to a template file, from which CreateFile() takes file attributes and extended attributes for creation of the new file.

CreateFile() Scenarios

There are several fundamental scenarios for initiating access to a file using the CreateFile() function. These are summarized as:

1. Creating a new file when a file with that name does not already exist.
2. Creating a new file even if a file of the same name already exists, clearing its data and starting empty.
3. Opening an existing file only if it exists, and only intact.
4. Opening an existing file only if it exists, truncating it to be empty.
5. Opening a file always: as-is if it exists, creating a new one if it doesn't exist.

These scenarios are controlled by the proper use of the dwCreationDisposition parameter. Below is a breakdown of how these scenarios map to values for this parameter and what happens when they are used.
When creating or opening a new file when a file with that name does not already exist (dwCreationDisposition set to either CREATE_NEW, CREATE_ALWAYS, or OPEN_ALWAYS), the CreateFile() function performs the following actions:

1. Combines the file attributes and flags specified by dwFlagsAndAttributes with FILE_ATTRIBUTE_ARCHIVE.
2. Sets the file length to zero.
3. Copies the extended attributes supplied by the template file to the new file if the hTemplateFile parameter is specified (this overrides all FILE_ATTRIBUTE_* flags specified earlier).
4. Sets the inherit flag specified by the bInheritHandle member and the security descriptor specified by the lpSecurityDescriptor member of the lpSecurityAttributes parameter (SECURITY_ATTRIBUTES structure), if supplied.

When creating a new file even if a file of the same name already exists (dwCreationDisposition set to CREATE_ALWAYS), the CreateFile() function performs the following actions:

1. Checks current file attributes and security settings for write access, failing if denied.
2. Combines the file attributes and flags specified by dwFlagsAndAttributes with FILE_ATTRIBUTE_ARCHIVE and the existing file attributes.
3. Sets the file length to zero (that is, any data that was in the file is no longer available and the file is empty).
4. Copies the extended attributes supplied by the template file to the new file if the hTemplateFile parameter is specified (this overrides all FILE_ATTRIBUTE_* flags specified earlier).

5. Sets the inherit flag specified by the bInheritHandle member of the lpSecurityAttributes parameter (SECURITY_ATTRIBUTES structure) if supplied, but ignores the lpSecurityDescriptor member of the SECURITY_ATTRIBUTES structure.
6. If otherwise successful (that is, CreateFile() returns a valid handle), calling GetLastError() will yield the code ERROR_ALREADY_EXISTS, even though for this particular use-case it is not actually an error as such (if you intended to create a "new" (empty) file in place of the existing one).

When opening an existing file (dwCreationDisposition set to either OPEN_EXISTING, OPEN_ALWAYS, or TRUNCATE_EXISTING), the CreateFile() function performs the following actions:

1. Checks current file attributes and security settings for requested access, failing if denied.
2. Combines the file flags (FILE_FLAG_*) specified by dwFlagsAndAttributes with existing file attributes, and ignores any file attributes (FILE_ATTRIBUTE_*) specified by dwFlagsAndAttributes.
3. Sets the file length to zero only if dwCreationDisposition is set to TRUNCATE_EXISTING, otherwise the current file length is maintained and the file is opened as-is.
4. Ignores the hTemplateFile parameter.
5. Sets the inherit flag specified by the bInheritHandle member of the lpSecurityAttributes parameter (SECURITY_ATTRIBUTES structure) if supplied, but ignores the lpSecurityDescriptor member of the SECURITY_ATTRIBUTES structure.

File Attributes and Directories

File attributes are part of the metadata associated with a file or directory, each with its own purpose and rules on how it can be set and changed. Some of these attributes apply only to files, and some only to directories. For example, the FILE_ATTRIBUTE_DIRECTORY attribute applies only to directories: It is used by the file system to determine whether an object on disk is a directory, but it cannot be changed for an existing file system object.
Some file attributes can be set for a directory but have meaning only for files created in that directory, acting as default attributes. For example, FILE_ATTRIBUTE_COMPRESSED can be set on a directory object, but because the directory object itself contains no actual data, it is not truly compressed; however, directories marked with this attribute tell the file system to compress any new files added to that directory. Any file attribute that can be set on a directory and will also be set for new files added to that directory is referred to as an inherited attribute.
The CreateFile() function provides a parameter for setting certain file attributes when a file is created. In general, these attributes are the most common for an application to use at file creation time, but not all possible file attributes are available to CreateFile(). Some file attributes require the use of other functions, such as SetFileAttributes(), DeviceIoControl(), or DecryptFile() after the file already exists. In the case of FILE_ATTRIBUTE_DIRECTORY, the CreateDirectory() function is required at creation time because CreateFile() cannot create directories. The other file attributes that require special handling are FILE_ATTRIBUTE_REPARSE_POINT and FILE_ATTRIBUTE_SPARSE_FILE, which require DeviceIoControl().
As stated previously, file attribute inheritance occurs when a file is created with file attributes read from the directory attributes where the file will be located. The following table summarizes these inherited attributes and how they relate to CreateFile() capabilities.

Directory attribute state	CreateFile inheritance override capability for new files
FILE_ATTRIBUTE_COMPRESSED set.	No control. Use DeviceIoControl() to unset.
FILE_ATTRIBUTE_COMPRESSED not set.	No control. Use DeviceIoControl() to set.
FILE_ATTRIBUTE_ENCRYPTED set.	No control. Use DecryptFile().
FILE_ATTRIBUTE_ENCRYPTED not set.	Can be set using CreateFile().
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED set.	No control. Use SetFileAttributes() to unset.
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED not set.	No control. Use SetFileAttributes() to set.

Moving and Replacing Files

Before a file can be copied, it must be closed or opened only for reading. No thread can have the file opened for writing. To copy an existing file to a new one, use the CopyFile() or CopyFileEx() function. Applications can specify whether CopyFile() and CopyFileEx() fail if the destination file already exists. If the file does exist and is open, it must have been opened with applicable sharing permissions.
The CopyFileEx() function also allows an application to specify the address of a callback function that is called each time another portion of the file has been copied. The application can use this information to display an indicator that shows the total number of bytes copied as a percent of the total file size.
The ReplaceFile() function replaces one file with another file, with the option of creating a backup copy of the original file. The function preserves attributes of the original file, such as its creation time, ACLs, and encryption attribute.
A file must also be closed before an application can move it. The MoveFile() and MoveFileEx() functions copy an existing file to a new location and deletes the original.
The MoveFileEx() function also allows an application to specify how to move the file. The function can replace an existing file, move a file across volumes, and delay moving the file until the operating system is restarted.

Closing and Deleting Files

To use operating system resources efficiently, an application should close files when they are no longer needed by using the CloseHandle() function. If a file is open when an application terminates, the system closes it automatically.
The DeleteFile() function can be used to delete a file on close. A file cannot be deleted until all handles to it are closed. If a file cannot be deleted, its name cannot be reused. To reuse a file name immediately, rename the existing file.
If you are deleting an open file or directory on a remote machine and it has already been opened on the remote machine without the read share permission set, do not call CreateFile() or OpenFile() to open the file or directory for deletion first. Doing so will result in a sharing violation.

Defragmenting Files

When a file is written to a disk, the file cannot be written in contiguous clusters. Noncontiguous clusters slow down the process of reading and writing a file. The further apart on a disk the

noncontiguous clusters are, the worse the issue, because of the time it takes to move the read/write head of a hard drive. A file with noncontiguous clusters is fragmented. To optimize files for fast access, a volume can be defragmented.
Defragmentation is the process of moving portions of files around on a disk to defragment files, that is, the process of moving file clusters on a disk to make them contiguous.
In a simple single-tasking operating system, the defragmentation software is the only task, and there are no other processes to read from or write to the disk. However, in a multitasking operating system, some processes can be reading from and writing to a hard drive while another process is defragmenting that hard drive. The trick is to avoid writes to a file that is being defragmented without stopping the writing process for very long. Solving this problem is not trivial, but it is possible.
To allow defragmentation without requiring detailed knowledge of a file system disk structure, a set of three control codes is provided. The control codes provide the following functionality.

1. Enable applications to locate empty clusters.
2. Determine the disk location of file clusters.
3. Move clusters on a disk.

The control codes also transparently handle the problem of inhibiting and allowing other processes to read from and write to files during moves.
These operations can be performed without inhibiting other processes from running. However, the other processes have slower response times while a disk drive is being defragmented. To defragment a file:

1. Use the FSCTL_GET_VOLUME_BITMAP control code to find a place on the volume that is large enough to accept an entire file.
   If necessary, move other files to make a place that is large enough. Ideally, there is enough unallocated clusters after the first extent of the file that you can move subsequent extents into the space after the first extent.
2. Use the FSCTL_GET_RETRIEVAL_POINTERS control code to get a map of the current layout of the file on the disk.
3. Walk the RETRIEVAL_POINTERS_BUFFER structure returned by FSCTL_GET_RETRIEVAL_POINTERS.
4. Use the FSCTL_MOVE_FILE control code to move each cluster as you walk the structure. You may need to renew either the bitmap or the retrieval structure, or both at various times as other processes write to the disk.

Two of the operations that are used in the defragmentation process require a handle to a volume. Only administrators can obtain a handle to a volume, so only administrators can defragment a volume. An application should check the rights of a user who attempts to run defragmentation software, and it should not allow a user to defragment a volume if the user does not have the appropriate rights.
When using CreateFile() to open a directory during defragmentation of a FAT or FAT32 file system volume, specify the GENERIC_READ access mask value. Do not specify the MAXIMUM_ALLOWED access mask value. Access to the directory is denied if that is done.
Do not attempt to move allocated clusters in an NTFS file system that extend beyond the cluster rounded file size, because the result is an error.

Re-parse points, bitmaps, and attribute lists in NTFS file system volumes can be defragmented, opened for reading and synchronization, and named using the file:name:type syntax; for example, $i30:$INDEX_ALLOCATION, mrp::$DATA, mrp::$REPARSE_POINT, and mrp::$ATTRIBUTE_LIST.
When defragmenting NTFS file system volumes, defragmenting a virtual cluster beyond the allocation size of a file is allowed.

Minimizing interactions between defragmentation and shadow copies

When possible, move data in blocks aligned relative to each other in 16-kilobyte (KB) increments. This reduces **copy-on-write overhead when shadow copies are enabled**, because shadow copy space is increased and performance is reduced when the following conditions occur:

1. The move request block size is less than or equal to 16 KB.
2. The move delta is not in increments of 16 KB.

The move delta is the number of bytes between the start of the source block and the start of the target block. In other words, a block starting at offset X (on-disk) can be moved to a starting offset Y if the absolute value of X minus Y is an even multiple of 16 KB. So, assuming 4-KB clusters, a move from cluster 3 to cluster 27 will be optimized, but a move from cluster 18 to cluster 24 will not. Note that mod(3,4) = 3 = mod(27,4). Mod 4 is chosen because four clusters at 4 KB each is equivalent to 16 KB. Therefore, a volume formatted to a 16-KB cluster size will result in all move files being optimized.

Limitations under Windows 2000

The following list identifies the limitations of the file defragmentation API under Windows 2000, and they do not apply to later versions of Windows.

1. The NTFS file system defragments data by using the system cache, therefore, encrypted files must be opened with read access.
2. The NTFS file system defragments uncompressed files at the page boundary.
3. The NTFS file system cannot defragment the master file table (MFT), re-parse points, bitmaps, and NTFS file system attribute lists.
4. The NTFS file system cannot defragment the space between the valid data length and the end of the file.

Retrieving File Type Information

The GetFileType() function retrieves the type of a file: disk, character (such as a console), pipe, or unknown. The GetBinaryType() function determines whether a file is executable, and if so, the type of executable file it is. See GetBinaryType() section for a list of the supported executable types.

Determining the Size of a File

The GetFileSize() function retrieves the size of a file. Because the NTFS file system implementation of files allows for multiple streams within a file, any application you write that accesses files must account for the possibility that the creator of the file can include multiple streams in the file. If multiple streams are not checked for in a file, the application can underestimate the total size of the file, among other errors.

Searching for One or More Files

An application can search the current directory for all file names that match a given pattern by using the FindFirstFile(), FindFirstFileEx(), FindNextFile(), and FindClose() functions. The pattern must be a valid file name and can include wildcard characters.
The FindFirstFile() and FindFirstFileEx() functions create handles that FindFirstFileEx() uses to search for other files with the same pattern. All functions return information about the file that was found. This information includes the file name, size, attributes, and time.
The FindFirstFileEx() function also allows an application to search for files based on additional search criteria. The function can limit searches to device names or directory names.
The FindClose() function destroys handles created by FindFirstFile() and FindFirstFileEx().
An application can search for a single file on a specific path by using the SearchPath() function.

Setting and Getting the Timestamp of a File

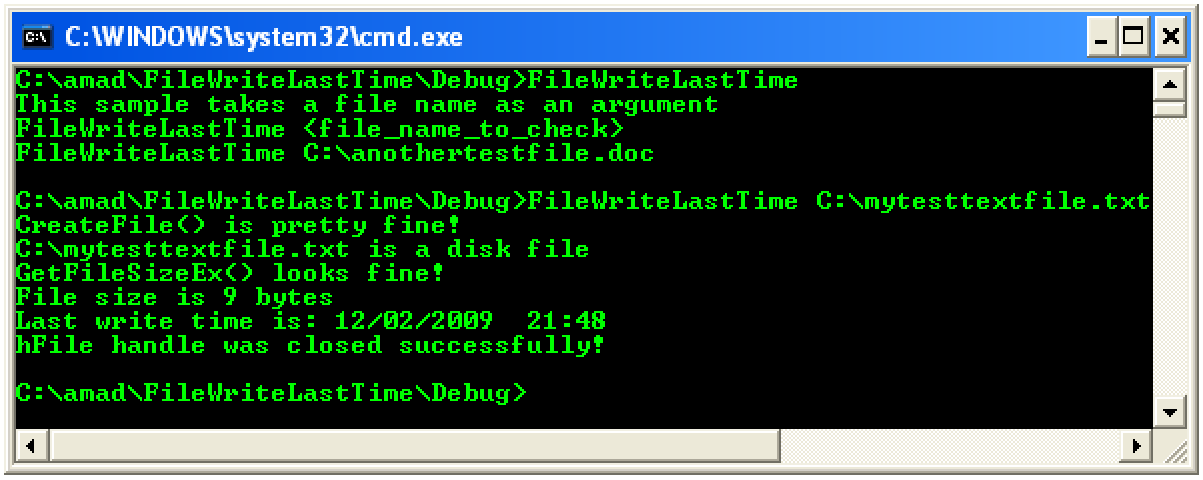
Applications can retrieve and set the date and time a file is created, last modified, or last accessed by using the GetFileTime() and SetFileTime() functions.
Note that not all file systems can record creation and last access times, and not all file systems record them in the same manner. For example, on FAT file system, create time has a resolution of 10 milliseconds, write time has a resolution of 2 seconds, and access time has a resolution of 1 day (really, the access date). The NTFS file system delays update to the last access time for a file by up to one hour after the last access.

File Type, Size and Timestamp Program Example

The following program example demonstrates the use of file type, file size and timestamp functions.
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.

```
// GetLastWriteTime - Retrieves the last-write time and converts
//                    the time to a string
//
// Return value - TRUE if successful, FALSE otherwise
// hFile      - Valid file handle
// lpszString - Pointer to buffer to receive string
#include <windows.h>
#include <strsafe.h>

// A prototype that receives a function name, displaying
// system error code and its respective message
void DisplayErrorBox(LPTSTR lpszFunction);

BOOL GetLastWriteTime(HANDLE hFile, LPTSTR lpszString, DWORD dwSize)
{
    FILETIME ftCreate, ftAccess, ftWrite;
    SYSTEMTIME stUTC, stLocal;
    DWORD dwRet;

    // Retrieve the file times for the file.
    if (!GetFileTime(hFile, &ftCreate, &ftAccess, &ftWrite))
        return FALSE;

    // Convert the last-write time to local time.
    FileTimeToSystemTime(&ftWrite, &stUTC);
    SystemTimeToTzSpecificLocalTime(NULL, &stUTC, &stLocal);

    // Build a string showing the date and time.
    dwRet = StringCchPrintf(lpszString, dwSize,
        L"%02d/%02d/%d  %02d:%02d",
        stLocal.wMonth, stLocal.wDay, stLocal.wYear,
        stLocal.wHour, stLocal.wMinute);

    if( S_OK == dwRet )
        return TRUE;
    else return FALSE;
}

int wmain(int argc, WCHAR *argv[])
{
    HANDLE hFile;
    WCHAR szBuf[MAX_PATH];
      DWORD dwFileType = 0;
      LARGE_INTEGER lpFileSize;
      BOOL bRetVal;

    if(argc != 2)
    {
        wprintf(L"This sample takes a file name as an argument\n");
            wprintf(L"%s <file_name_to_check>\n", argv[0]);
            wprintf(L"%s C:\\anothertestfile.doc\n", argv[0]);
            // Non-zero means error
        return 1;
    }

    hFile = CreateFile(argv[1], GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_EXISTING, 0, NULL);

    if(hFile == INVALID_HANDLE_VALUE)
    {
        DisplayErrorBox(L"CreateFile()");
        return 1;
    }
      else
            wprintf(L"CreateFile() is pretty fine!\n");

      dwFileType = GetFileType(hFile);

      switch(dwFileType)
      {
      case FILE_TYPE_CHAR: wprintf(L"%s is a character file, typically an LPT
device or a console.\n", argv[1]);
            break;
      case FILE_TYPE_DISK: wprintf(L"%s is a disk file\n", argv[1]);
            break;
      case FILE_TYPE_PIPE: wprintf(L"%s is a socket, a named pipe, or an
anonymous pipe.\n", argv[1]);
            break;
      case FILE_TYPE_UNKNOWN: wprintf(L"%s is unknown, or the function
failed!\n", argv[1]);
            break;
      default:
            wprintf(L"%s is unknown type!\n", argv[1]);
      }

      bRetVal = GetFileSizeEx(hFile, &lpFileSize);

      // TRUE
      if(bRetVal != 0)
      {
            wprintf(L"GetFileSizeEx() looks fine!\n");
            wprintf(L"File size is %u bytes\n", lpFileSize.QuadPart);
      }
      else
            DisplayErrorBox(L"GetFileSizeEx()");


    if(GetLastWriteTime(hFile, szBuf, MAX_PATH))
        wprintf(L"Last write time is: %s\n", szBuf);

    if(CloseHandle(hFile) != 0)
            wprintf(L"hFile handle was closed successfully!\n");
      else
            DisplayErrorBox(L"CloseHandle()");

      return 0;
}

void DisplayErrorBox(LPTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL );

    // Display the error message and clean up
    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,

(lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(WCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(WCHAR),  L"%s failed with error %d:
%s",
        lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, L"Error", MB_OK);
    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}
```

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



```
C:\amad\FileWriteLastTime\Debug>FileWriteLastTime
This sample takes a file name as an argument
FileWriteLastTime <file_name_to_check>
FileWriteLastTime C:\anothertestfile.doc

C:\amad\FileWriteLastTime\Debug>FileWriteLastTime C:\mytesttextfile.txt
CreateFile() is pretty fine!
C:\mytesttextfile.txt is a disk file
GetFileSizeEx() looks fine!
File size is 9 bytes
Last write time is: 12/02/2009  21:48
hFile handle was closed successfully!

C:\amad\FileWriteLastTime\Debug>
```

Determining the Current Character Set Code Page

The AreFileApisANSI() function determines whether the file I/O functions are using the ANSI or OEM character set code page. The SetFileApisToANSI() function causes the functions to use the ANSI code page. The SetFileApisToOEM() function causes the functions to use the OEM code page. By default, file I/O functions use ANSI file names. Functions exported by Kernel32.dll that accept or return file names are affected by the file code page setting.
Both SetFileApisToANSI() and SetFileApisToOEM() set the code page per process, rather than per thread or per computer.

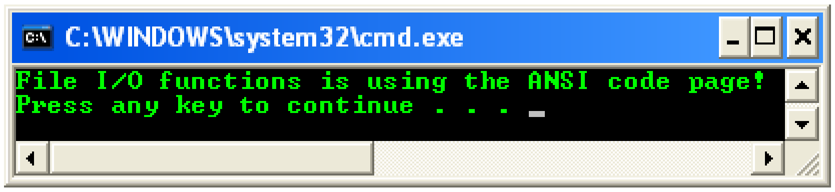
AreFileApisANSI() Program Example

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 wmain(int argc, WCHAR *argv[])
{
      BOOL bRetVal;

      bRetVal = AreFileApisANSI();

      if(bRetVal !=  0)
            wprintf(L"File I/O functions is using the ANSI code page!\n");
      else
```

```
            wprintf(L"File I/O functions is using the OEM code page!\n");

    return 0;
}
```

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



```
File I/O functions is using the ANSI code page!
Press any key to continue . . .
```

Reading From and Writing to Files

An application reads from and writes to a file by using the ReadFile(), ReadFileEx(), WriteFile(), and WriteFileEx() functions. These functions require a handle to a file to be opened for reading and writing, respectively. They read and write a specified number of bytes at the location indicated by the file pointer. The data is read and written exactly as specified; the functions do not format the data.
When the file pointer reaches the end of a file and the application attempts to read from the file, no error occurs, but no bytes are read. Therefore, reading zero bytes without an error means the application has reached the end of the file. Writing zero bytes does nothing.

Positioning a File Pointer

When an application calls CreateFile() to open a file for the first time, Windows places the file pointer at the beginning of the file. As bytes are read from or written to the file, Windows advances the file pointer the number of bytes read or written.
An application can position the file pointer to a specified offset by calling SetFilePointer().
The SetFilePointer() function can also be used to query the current file pointer position by specifying a move method of FILE_CURRENT and a distance of zero.

Reading From or Writing To Files Using a Scatter-Gather Scheme

A scatter-gather scheme uses the operating system to deliver in one operation multiple discrete chunks of data (such as database records) from a file to separate, noncontiguous buffers in memory. A scatter-gather scheme also writes the data from noncontiguous buffers in one operation.
An application can implement a scatter-gather scheme with ReadFileScatter() and WriteFileGather().

Flushing System-Buffered I/O Data to Disk

Windows stores the data in file read and write operations in system-maintained data buffers to optimize disk performance. When an application writes to a file, the system usually buffers the data and writes the data to the disk on a regular basis. An application can force the operating system to write the contents of these data buffers to the disk by using the FlushFileBuffers() function.
Alternatively, an application can specify that write operations are to bypass the data buffer and write directly to the disk by setting the FILE_FLAG_NO_BUFFERING flag when the file is created or opened using the CreateFile() function.

If there is data in the internal buffer when the file is closed, the operating system does not automatically write the contents of the buffer to the disk before closing the file. If the application does not force the operating system to write the buffer to disk before closing the file, the caching algorithm determines when the buffer is written.
Accessing a data buffer while a read or write operation is using it may corrupt the buffer. Applications must not read from, write to, reallocate, or free the data buffer that a read or write operation is using until the operation completes.

Truncating or Extending Files

An application can truncate or extend a file by calling SetEndOfFile(). This function sets the end-of-file marker to the current position of the file pointer. Note that when a file is extended, the contents between the old and new end-of-file locations are not defined.

File and Directory Linking

The NTFS file system provides the ability to create a system representation of a file or directory in a location in the directory structure that is different from the file or directory object that is being linked to. This process is called linking. There are two types of links supported in the NTFS file system: hard links and junctions. The NTFS file system also provides the distributed link tracking service, which automatically tracks links as they are moved.

File Compression and Decompression

The NTFS file system volumes support file compression on an individual file basis. The file compression algorithm used by the NTFS file system is **Lempel-Ziv** compression. This is a lossless compression algorithm, which means that no data is lost when compressing and decompressing the file, as opposed to lossy compression algorithms such as JPEG, where some data is lost each time data compression and decompression occur.
Data compression reduces the size of a file by minimizing redundant data. In a text file, redundant data can be frequently occurring characters, such as the space character, or common vowels, such as the letters e and a; it can also be frequently occurring character strings. Data compression creates a compressed version of a file by minimizing this redundant data.
Each type of data-compression algorithm minimizes redundant data in a unique manner. For example, the Huffman encoding algorithm assigns a code to characters in a file based on how frequently those characters occur. Another compression algorithm, called run-length encoding, generates a two-part value for repeated characters: the first part specifies the number of times the character is repeated, and the second part identifies the character. Another compression algorithm, known as the **Lempel-Ziv** algorithm, converts variable-length strings into fixed-length codes that consume less space than the original strings.

The NTFS File System File Compression

On the NTFS file system, compression is performed transparently. This means it can be used without requiring changes to existing applications. The compressed bytes of the file are not accessible to applications; they see only the uncompressed data. Therefore, applications that open a compressed file can operate on it as if it were not compressed. However, these files cannot be copied to another file system. If you compress a file that is larger than 30 gigabytes, the compression may not succeed. The following topics identify the NTFS file system file compression:

Compression Attribute

On an NTFS file system volume, each file and directory has a compression attribute. Other file systems may also implement a compression attribute for individual files and directories.
You can determine whether a file system supports a compression attribute for files and directories by calling the GetVolumeInformation() function and examining the FS_FILE_COMPRESSION bit flag.
Use the GetFileAttributes() or GetFileAttributesEx() function to determine the compression attribute of a file or directory.
If a file's compression attribute is set (FILE_ATTRIBUTE_COMPRESSED), all data in the file is compressed. If the attribute is clear, none of the data in the file is compressed. There is no partially compressed state from a user-mode programming perspective; the compression attribute is a simple Boolean indicator of compression state.
A directory's compression attribute provides a default compression attribute for newly created files and subdirectories. When you call CreateFile() or CreateDirectory() to create a new file or directory, the new file or directory inherits the compression attribute of its parent directory.
To modify the FILE_ATTRIBUTE_COMPRESSED attribute for a file or directory, you must use the DeviceIoControl() function with the FSCTL_SET_COMPRESSION control code.

Compression State

Each file and directory on a volume that supports compression for individual files and directories has a compression state. Whereas the compression attribute of a file or directory indicates simply whether the file or directory is compressed or not compressed, the compression state also specifies the format of any compressed data. Use the FSCTL_GET_COMPRESSION control code to determine the compression state of a file or directory. Compression state is encoded as a 16-bit value.
A compression state value of COMPRESSION_FORMAT_NONE indicates that a file is not compressed. A value of COMPRESSION_FORMAT_DEFAULT indicates that a file is compressed, using the default compression format. Any other value indicates that a file is compressed, using the compression format specified by the compression state value.
Use the FSCTL_SET_COMPRESSION control code to set the compression state of a file or directory. This operation also sets the compression attribute of the file or directory.
Setting the compression state of a file to a nonzero value compresses the file, using the compression format encoded by the compression state value. Setting a file's compression state to zero decompresses the file. These are synchronous operations. The file is compressed or decompressed immediately when you set its compression state.
Setting a directory's compression state does not cause any immediate compression or decompression. Instead, setting a directory's compression state sets a default compression state that will be given to all newly created files and subdirectories.

Obtaining the Size of a Compressed File

Use the GetCompressedFileSize() function to obtain the compressed size of a file. If the file is compressed, its compressed size will be less than its uncompressed size. Use the GetFileSize() function to determine the uncompressed size of a file.

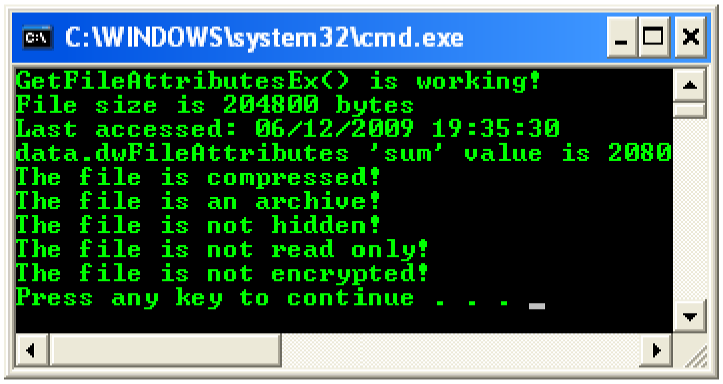
Compressed File (Attributes) Program Example

The following program example demonstrates some of the compressed file functions.
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 wmain(int argc, WCHAR *argv[])
{
	BOOL bRetVal;
	// Make sure the file is exist
	LPCTSTR lpFileName = L"\\\\?\\C:\\WINDOWS\\ie7\\iexplore.chm";
	// LPCTSTR lpFileName =
L"\\\\?\\C:\\amad\\AddUserEncryptFile\\Debug\\testfilencrypt.doc";
	WIN32_FILE_ATTRIBUTE_DATA data;
	GET_FILEEX_INFO_LEVELS fInfoLevelId = GetFileExInfoStandard;
	SYSTEMTIME times, stLocal;
	DWORD dwFileSize = 0;
	// File  less than 4GB does not need this
	LPDWORD lpFileSizeHigh = NULL;

	bRetVal = GetFileAttributesEx(lpFileName, fInfoLevelId, &data);

	if(bRetVal == 0)
		wprintf(L"GetFileAttributesEx() pretty damn failed, error %u\n",
GetLastError());
	else
	{
		wprintf(L"GetFileAttributesEx() is working!\n");

		// dwFileSize is the low-order DWORD of the actual number
		// of bytes of disk storage used to store the specified file
		dwFileSize = GetCompressedFileSize(lpFileName, lpFileSizeHigh);

		wprintf(L"File size is %u bytes\n", dwFileSize);

		FileTimeToSystemTime(&data.ftLastAccessTime, &times);
		SystemTimeToTzSpecificLocalTime(NULL, &times, &stLocal);
		wprintf(L"Last accessed: %02d/%02d/%04d %02d:%02d:%02d\n",
stLocal.wDay , stLocal.wMonth,
			stLocal.wYear, stLocal.wHour, stLocal.wMinute,
stLocal.wSecond);

		// The value is bitwise AND lol
		wprintf(L"data.dwFileAttributes \'sum\' value is %u\n",
data.dwFileAttributes);

		// Let play around with some of the attributes
		if(data.dwFileAttributes & FILE_ATTRIBUTE_COMPRESSED)
			wprintf(L"The file is compressed!\n");
		else
			wprintf(L"The file is not compressed!\n");
```

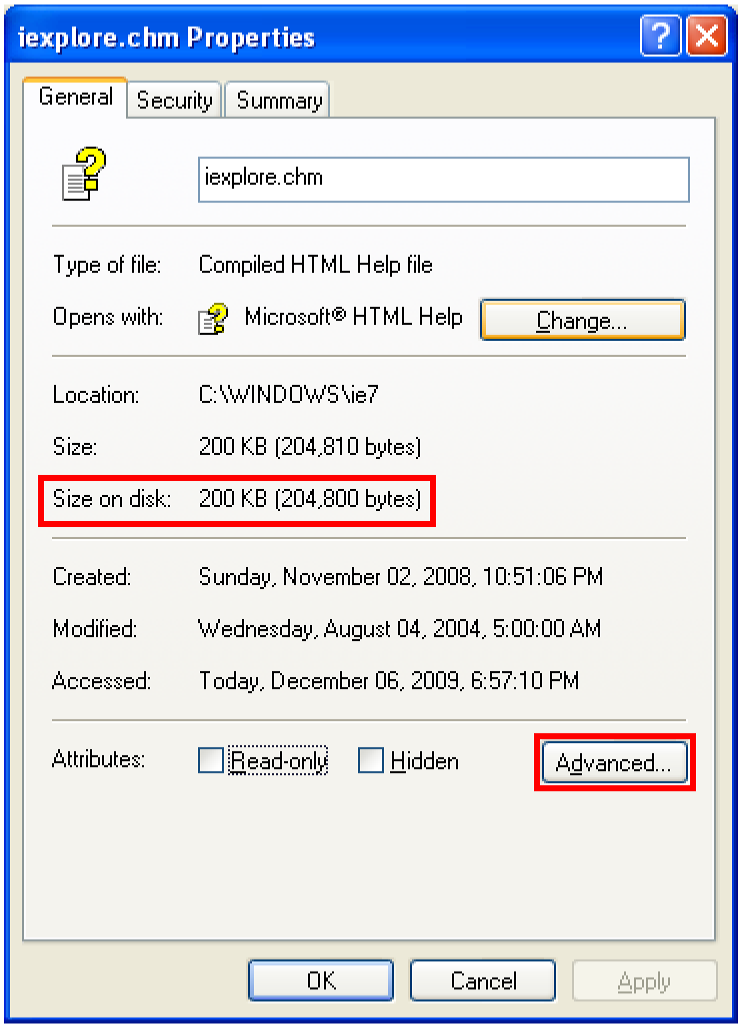
```
            if(data.dwFileAttributes & FILE_ATTRIBUTE_ARCHIVE)
                  wprintf(L"The file is an archive!\n");
            else
                  wprintf(L"The file is not an archive!\n");

            if(data.dwFileAttributes & FILE_ATTRIBUTE_HIDDEN)
                  wprintf(L"The file is hidden!\n");
            else
                  wprintf(L"The file is not hidden!\n");

            if(data.dwFileAttributes & FILE_ATTRIBUTE_READONLY)
                  wprintf(L"The file is read only!\n");
            else
                  wprintf(L"The file is not read only!\n");

            if(data.dwFileAttributes & FILE_ATTRIBUTE_ENCRYPTED)
                  wprintf(L"The file is an encrypted!\n");
            else
                  wprintf(L"The file is not encrypted!\n");
      }


      // You may want to try other attributes:
      // data.ftCreationTime;
      // data.ftLastWriteTime;
      // data.nFileSizeHigh;
      // data.nFileSizeLow;

            // Close handle


      return 0;
}
```

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



```
GetFileAttributesEx() is working!
File size is 204800 bytes
Last accessed: 06/12/2009 19:35:30
data.dwFileAttributes 'sum' value is 2080
The file is compressed!
The file is an archive!
The file is not hidden!
The file is not read only!
The file is not encrypted!
Press any key to continue . . .
```

From the file property's page, we can see that the size on disk. Click the Advanced button for other information.



iexplore.chm

Type of file: Compiled HTML Help file

Opens with: Microsoft® HTML Help

Location: C:\WINDOWS\ie7

Size: 200 KB (204,810 bytes)

Size on disk: 200 KB (204,800 bytes)

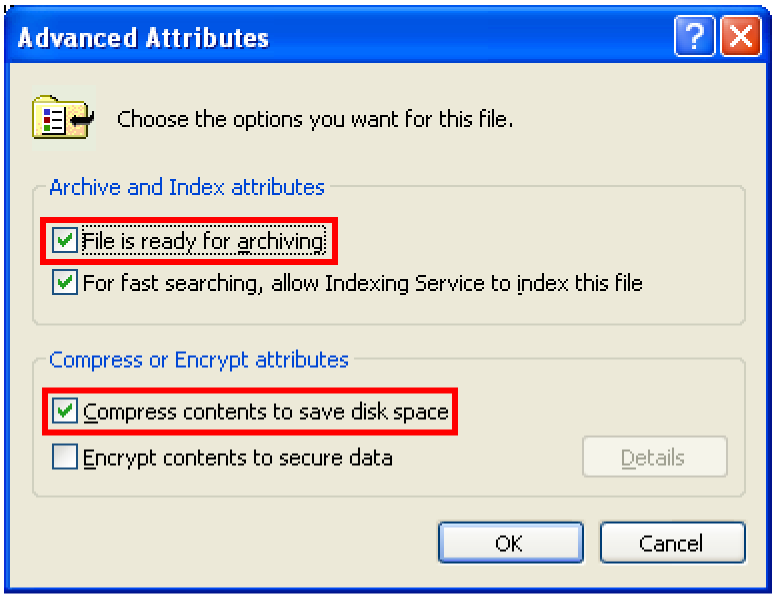
Created: Sunday, November 02, 2008, 10:51:06 PM

Modified: Wednesday, August 04, 2004, 5:00:00 AM

Accessed: Today, December 06, 2009, 6:57:10 PM

Attributes: ☐ Read-only ☐ Hidden

Well, the file is a compressed file (blue in colour). Both file size and compressed attribute should confirm our program output.



Choose the options you want for this file.

Archive and Index attributes

☑ File is ready for archiving

☑ For fast searching, allow Indexing Service to index this file

Compress or Encrypt attributes

☑ Compress contents to save disk space

☐ Encrypt contents to secure data

File Compression and Decompression Libraries

The file compression and decompression libraries take an existing file or files and produce a file or files that are compressed versions of the originals. The compression is also lossless, but the compression is not transparent to applications. An application can only operate on such files with the assistance of a file compression library. In addition, the only operations you can perform on such files are creating a compressed file from an original and recovering the original data from the decompressed version. Editing is typically not supported, and seeking is limited if supported at all. Typically, an application calls functions in **LzExpand.dll** to decompress data that was compressed using **Compress.exe**. The functions can also process files without attempting to decompress them. You can use the functions in LzExpand.dll to decompress single or multiple files. You can also use them to decompress compressed files a portion at a time.

Decompressing a Single File

An application can decompress a single compressed file by performing the following tasks:

1. Open the source file by calling the LZOpenFile() function.
2. Open the destination file by calling LZOpenFile().
3. Copy the source file to the destination file by calling the LZCopy() function and passing the handles returned by LZOpenFile().
4. Close the files by calling the LZClose() function.

Decompressing Multiple Files

An application can decompress multiple files by performing the following tasks:

1. Open the source files by calling the LZOpenFile() function.
2. Open the destination files by calling LZOpenFile().
3. Copy the source files to the destination files by calling the LZCopy() function.
4. Close the files by calling the LZClose() function.

Reading from Compressed Files

In addition to decompressing a complete file in a single operation, an application can decompress a compressed file a portion at a time by using the LZSeek() and LZRead() functions. These functions are particularly useful when it is necessary to extract parts of large files. For example, a font manufacturer may have compressed files containing font metrics in addition to character data. To use the information in these files, an application would need to decompress the file; however, most applications would use only part of the file at any particular time. To get information about font metrics, the application would extract data from the header. To get information from the text, the application would reposition the file pointer by calling LZSeek() and extract character data by calling LZRead().

Using Windows Compressed Functions Program Example

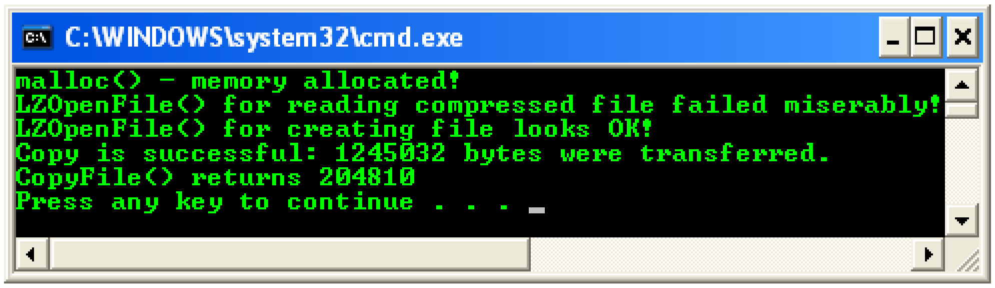
The following program example demonstrates some of Windows compressed functions.

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.

```
/*
How to use the LZOpenFile(), LZCopy(), and LZClose() functions from
the LZ Expand/Compress library (LZ32.dll). You can use these
functions to make a copy of an existing file.
The LZCopy function creates a decompressed destination file
if the source file is compressed with the
Microsoft File Compression Utility (Compress.exe).
*/
#include <windows.h>
#include <stdio.h>

// #pragma comment(lib, "LzExpand")
#pragma comment(lib, "LZ32")

int CopyFile(LPWSTR Source, LPWSTR Destination)
{
    INT hsource;
    INT hdest;
    DWORD iret;
      LPOFSTRUCT OpenStruct = {0};

      // Allocate buffer
      OpenStruct = (LPOFSTRUCT)malloc(sizeof(LPOFSTRUCT));

      if(OpenStruct != NULL)
            wprintf(L"malloc() - memory allocated!\n");
      else
            wprintf(L"malloc() - memory allocation failed! error %u\n",
GetLastError());

    // Open the source and destination files.
      // Opens the file for reading only.
    hsource = LZOpenFile(Source, OpenStruct, OF_READ);
      if(hsource)
            wprintf(L"LZOpenFile() for reading compressed file looks OK!\n");
      else
            // Can't use GetLastError() lol
            wprintf(L"LZOpenFile() for reading compressed file failed
miserably!\n");

      // Directs LZOpenFile to create a new file. If the file already exists,
      // it is truncated to zero length.
    hdest = LZOpenFile(Destination, OpenStruct, OF_CREATE);

      if(hdest)
            wprintf(L"LZOpenFile() for creating file looks OK!\n");
      else
            // Can't use GetLastError() lol
            wprintf(L"LZOpenFile() for creating file failed miserabily!\n");

    // Copy the source file to the destination location, and
    // decompress the Source file if it was compressed.
      // Fail - returns less than 1
```

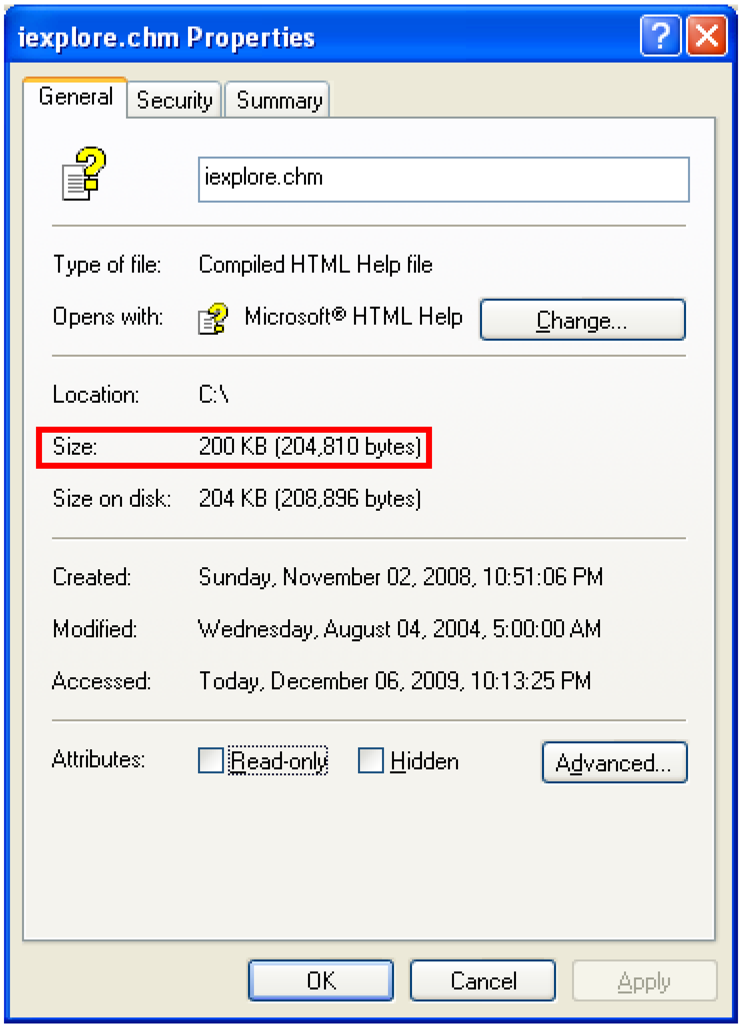
```
          // Success - returns the size, in bytes, of the destination file
        iret = LZCopy(hsource, hdest);
        // Close those files. This function does not return a value.
        LZClose(hdest);
        LZClose(hsource);

        if(iret == -1)
                wprintf(L"File transfer failed\n");
        else
            wprintf(L"Copy is successful: %u bytes were transferred.\n");

          // Failed to free the allocated buffer lol!
          // free(OpenStruct);

          // Let see what is returned
          return iret;
}

int wmain(int argc, WCHAR *argv[])
{
          LPWSTR pSourceFileName = NULL;
          LPWSTR pDestFileName = NULL;
          DWORD dwRetVal = 0;

          // Just a test. iexplore.chm is a compressed file
          pSourceFileName = L"\\\\?\\C:\\WINDOWS\\ie7\\iexplore.chm";
          // We uncompress and copy to C:
          pDestFileName = L"\\\\?\\C:\\iexplore.chm";

          // Call CopyFile()
          dwRetVal = CopyFile(pSourceFileName,pDestFileName);
          wprintf(L"CopyFile() returns %u\n", dwRetVal);

          return 0;
}
```

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



```
malloc() - memory allocated!
LZOpenFile() for reading compressed file failed miserably!
LZOpenFile() for creating file looks OK!
Copy is successful: 1245032 bytes were transferred.
CopyFile() returns 204810
Press any key to continue . . .
```

Find the uncompressed file and open its property page. The size is matched with the program output.


iexplore.chm Properties

General | Security | Summary

iexplore.chm

Type of file: Compiled HTML Help file

Opens with: Microsoft® HTML Help

Location: C:\

Size: 200 KB (204,810 bytes)

Size on disk: 204 KB (208,896 bytes)

Created: Sunday, November 02, 2008, 10:51:06 PM

Modified: Wednesday, August 04, 2004, 5:00:00 AM

Accessed: Today, December 06, 2009, 10:13:25 PM

Attributes: ☐ Read-only ☐ Hidden

Cabinets

Cabinets are created by a compression library that supports features such as disk spanning and multi-file compression. For additional information, see the Cabinet Software Development Kit.

File Encryption

The Encrypted File System, or EFS, was introduced in NTFS 5.0 to provide an additional level of security for files and directories. It provides cryptographic protection of individual files on NTFS file system volumes using a public-key system.
Typically, the access control to file and directory objects provided by the Windows security model is sufficient to protect unauthorized access to sensitive information. However, if a laptop that contains sensitive data is lost or stolen, the security protection of that data may be compromised. Encrypting the files increases security.
To determine whether a file system supports file encryption for files and directories, call the GetVolumeInformation() function and examine the FS_FILE_ENCRYPTION bit flag. Note that the following items cannot be encrypted:

1. Compressed files
2. System files

3. System directories
4. Root directories
5. Transactions

Sparse files can be encrypted. Transactional NTFS (TxF) does not support most operations on Encrypted File System (EFS) files. The only operations TxF supports are read operations, such as ReadEncryptedFileRaw().

Handling Encrypted Files and Directories

A programmer or user may mark a directory or file as encrypted. A file marked encrypted is encrypted by the NTFS file system using the current encryption driver. If at a later date the file is marked as not encrypted, it is decrypted and left in a plain text (unsecured) state.
Directories are not themselves encrypted. Rather, by default, in an "encrypted" directory all new files in the directory are encrypted at creation. A user must specifically change the status of a new file to decrypted if the user does not want the file to be encrypted. An encrypted directory is visible. To make a directory inaccessible to other users, use the standard methods of access control.
The encryption functions cannot be used with the Backup API.
To encrypt a new file, use the CreateFile() function with the FILE_ATTRIBUTE_ENCRYPTED flag. To encrypt an existing file, use the EncryptFile() function. All data streams in the file are encrypted. If the file is already encrypted, EncryptFile() does nothing but returns a nonzero value, which indicates success. If the file is compressed, EncryptFile() decompresses the file before encrypting it.
To decrypt an encrypted file, use the DecryptFile() function. If the file is not encrypted, DecryptFile() does nothing but returns a nonzero value indicating success.
The EncryptionDisable() function disables or enables the encryption of the indicated directory and the files in it. It does not affect the encryption of subdirectories below the indicated directory.
To retrieve the encryption status of a file, use the FileEncryptionStatus() function. Alternatively, call the GetFileAttributes() function and examine the FILE_ATTRIBUTE_ENCRYPTED flag in the return value.
When encrypted files are copied using CopyFile() and CopyFileEx() under Windows 2000, the functions attempt to encrypt the destination file. No attempt is made to encrypt the destination file with the keys used in the encryption of the source file. If it cannot be encrypted, CopyFile() and CopyFileEx() complete the copy operation without encrypting the destination file.
In Windows XP, CopyFile() and CopyFileEx() attempt to encrypt the destination file with the keys used in the encryption of the source file. If this cannot be done, both functions attempt to encrypt the destination file with default keys, as in Windows 2000. If both of these methods cannot be done, CopyFile() and CopyFileEx() fail with an ERROR_ENCRYPTION_FAILED error. If you want CopyFileEx() to complete the copy operation even when the destination file cannot be encrypted, include the COPY_FILE_ALLOW_DECRYPTED_DESTINATION flag in the value of the dwCopyFlags parameter in your call to CopyFileEx().

Encrypted Files and User Keys

To create a new key for a user, use the SetUserFileEncryptionKey() function. To add user keys to an encrypted file, use the AddUsersToEncryptedFile() function. To query the user keys for an encrypted file, use the QueryUsersOnEncryptedFile() function. To remove user keys from an encrypted file, use the RemoveUsersFromEncryptedFile() function.

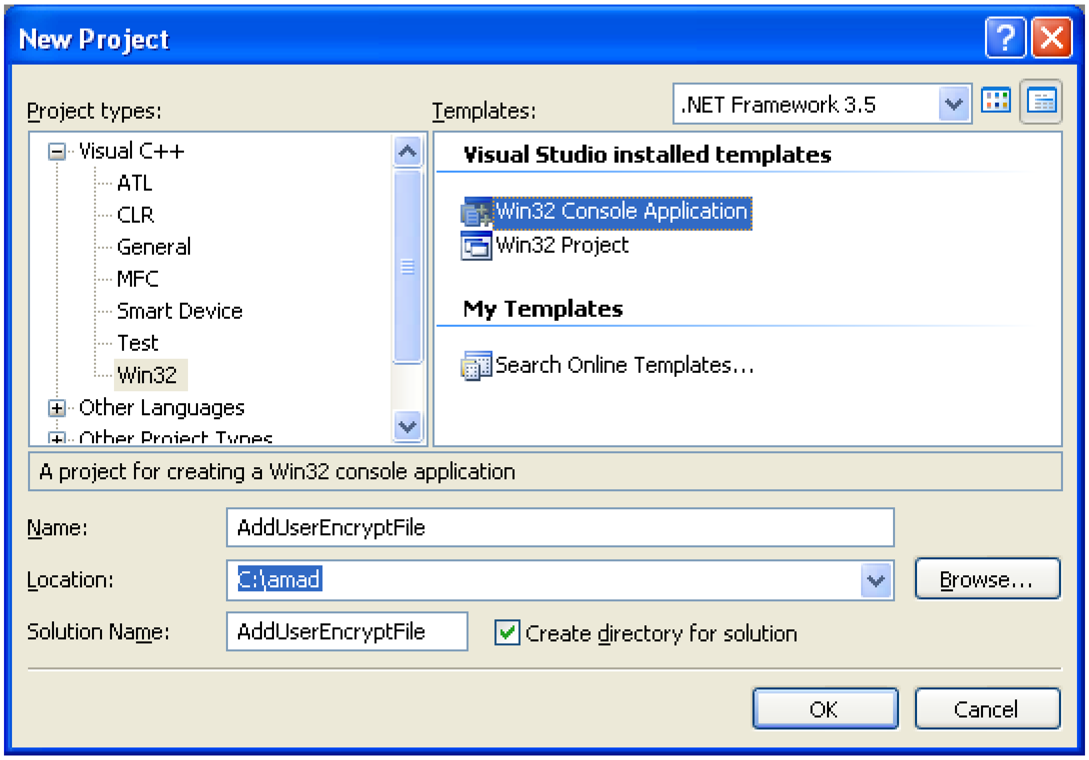
Backup and Restore of Encrypted Files

The Encrypting File System (EFS) filters the opening of an encrypted file in such a way that the application that opened the file gets access to the unencrypted information, provided of course it has the proper credentials to access the file and get the key necessary to decrypt the file. Subsequent read operations on this file will yield unencrypted text. This is very desirable for typical access to encrypted files, and keeps the encryption and decryption of the files transparent. However, it hinders backup of encrypted files, because if backup is attempted with the standard file I/O calls like CreateFile(), ReadFile(), and WriteFile(), the files backed up will be the plain text version.
The raw encryption functions are provided to solve this problem. Backup applications are a primary intended user for these functions. Raw encryption functions differ from other file system functions in that open, read, and write functions allow access to the raw encrypted data streams and also allow reading/writing of the $EFS stream. Therefore, the caller of the raw encryption functions does not need access to the cryptographic keys that decrypt the file. The following raw encryption APIs are available for use with backup and restore applications:

Raw Encryption API	Description
OpenEncryptedFileRaw()	Open an encrypted file with access to data in encrypted format.
CloseEncryptedFileRaw()	Close an encrypted file opened with OpenEncryptedFileRaw()
ReadEncryptedFileRaw()	Read an encrypted file leaving its data in encrypted format
WriteEncryptedFileRaw()	Write an encrypted file leaving its data in encrypted format
ImportCallback()	Application-defined callback for use with WriteEncryptedFileRaw()
ExportCallback()	Application-defined callback for use with ReadEncryptedFileRaw()

Adding Users to an Encrypted File Program Example

The following code sample adds a new user to an existing encrypted file by using the AddUsersToEncryptedFile() function. It requires the user's EFS certificate (from the Active Directory) to exist in the Trusted People user certificate store.
This sample adds a new Data Recovery Field to the encrypted file. As a result, the newly added user can decrypt the encrypted file. The caller must already have access to the encrypted file, either as the original owner, the data recovery agent, or as a user who was previously added to the encrypted file. This example is not supported in Windows 2000 because you cannot add multiple users to an encrypted file.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.


New Project

Project types:
- Visual C++
  - ATL
  - CLR
  - General
  - MFC
  - Smart Device
  - Test
  - Win32
- Other Languages
- Other Project Types

Templates: .NET Framework 3.5

Visual Studio installed templates
- Win32 Console Application
- Win32 Project

My Templates
- Search Online Templates...

A project for creating a Win32 console application

Name: AddUserEncryptFile

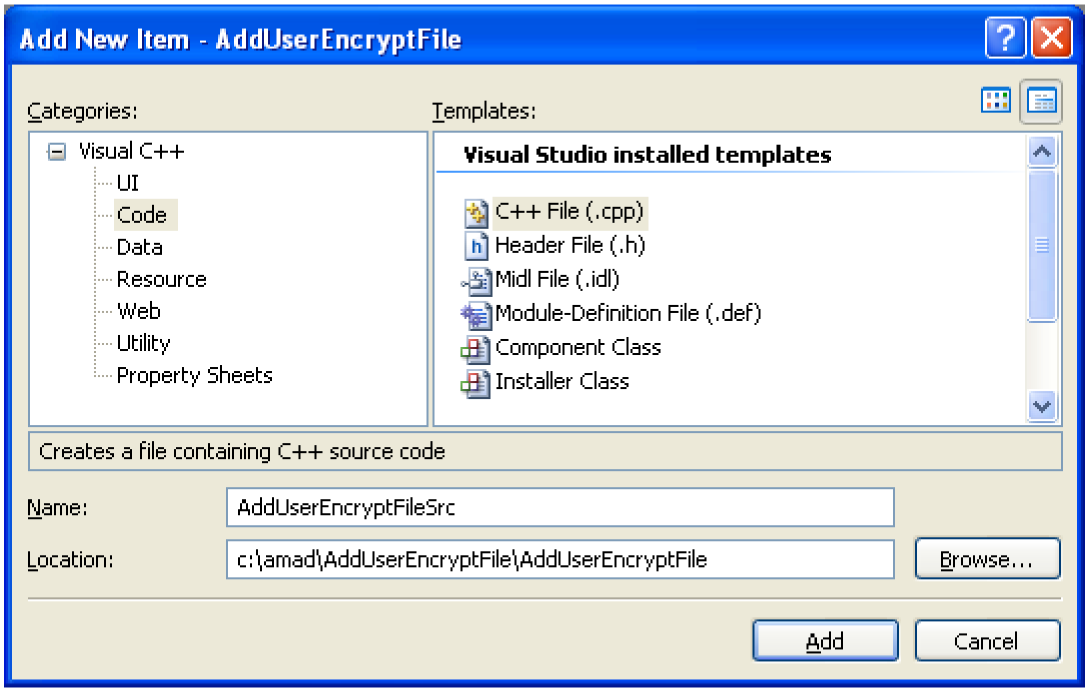
Location: C:\amad

Solution Name: AddUserEncryptFile

[✓] Create directory for solution

OK Cancel

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


Add New Item - AddUserEncryptFile

Categories:
- Visual C++
  - UI
  - Code
  - Data
  - Resource
  - Web
  - Utility
  - Property Sheets

Templates:

Visual Studio installed templates
- C++ File (.cpp)
- Header File (.h)
- Midl File (.idl)
- Module-Definition File (.def)
- Component Class
- Installer Class

Creates a file containing C++ source code

Name: AddUserEncryptFileSrc

Location: c:\amad\AddUserEncryptFile\AddUserEncryptFile

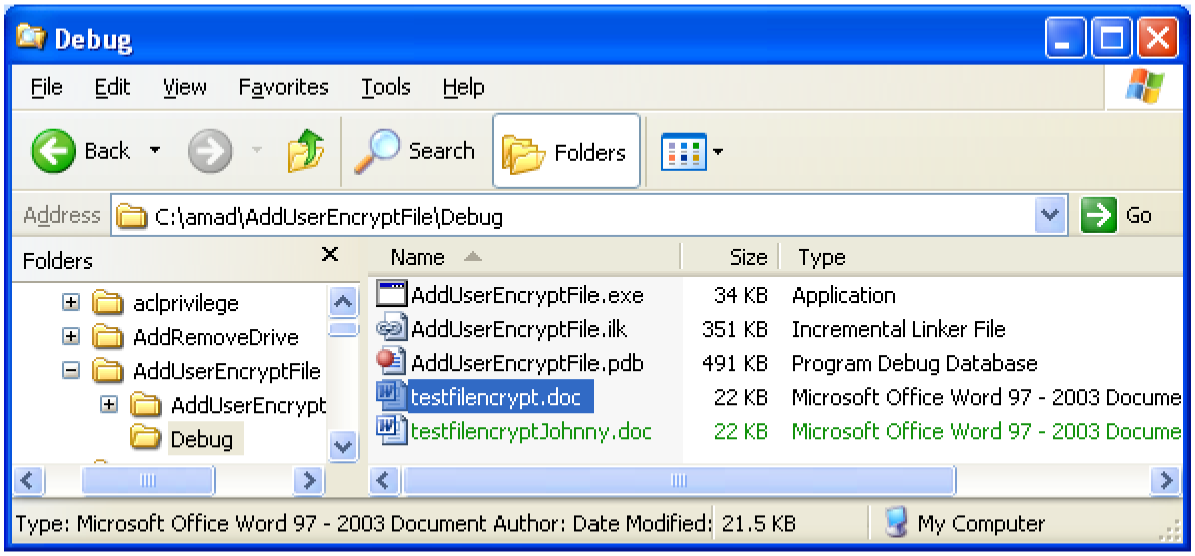
Add Cancel

Next, add the following source code.

```
// Adds a user to an encrypted file example
#include <Windows.h>
#include <malloc.h>
#include <stdio.h>
```

```
#include <wincrypt.h>
#include <winefs.h>
// A safer version for string manipulation
#include <strsafe.h>

// Link with the following libraries
#pragma comment(lib, "Advapi32.lib")
#pragma comment(lib, "Crypt32.lib")

// A prototype that receives a function name, displaying
// system error code and its respective message
void DisplayErrorBox(LPTSTR lpszFunction);

// Utility function that outputs this application's usage instructions.
void Usage(LPWSTR wszAppName)
{
    wprintf(L"\n%s: Adds users to encrypted files.\n", wszAppName);
    wprintf(L"Usage:\t%s <file> <username> <subjectname>\n\n", wszAppName);
    wprintf(L"\t<file> is the name of the file\n");
    wprintf(L"\t<username> is the name of the user's account\n");
    wprintf(L"\t\tExample: for name@example.com, use \"name\"\n");
    wprintf(L"\t<subjectname> is the \"IssuedTo\" name on the ");
    wprintf(L"certificate\n\t\tfrom the TrustedPeople store.\n");
    exit(1);
}

//  void ErrorExit(LPWSTR wszErrorMessage, DWORD dwErrorCode);

void wmain(int argc, WCHAR *argv[])
{
    LPWSTR wszFile    = NULL;
    LPWSTR wszAccount = NULL;
    LPWSTR wszSubject = NULL;
    PSID   pSid       = NULL;
    DWORD  cbSid      = 0;
    LPWSTR wszDomain  = NULL;
    DWORD  cchDomain  = 0;
    SID_NAME_USE SidType = SidTypeUser;
    HCERTSTORE hStore = NULL;
    PCCERT_CONTEXT pCertContext = NULL;
    PENCRYPTION_CERTIFICATE      pEfsEncryptionCert     = NULL;
    PENCRYPTION_CERTIFICATE_LIST pEfsEncryptionCertList = NULL;
    DWORD dwResult = ERROR_SUCCESS;

    // Simple check whether to explain usage to the user.
    if(argc !=4)
    {
        Usage(argv[0]);
    }

    // TODO: Check the parameters for correctness.
    wszFile = argv[1];
    wszAccount = argv[2];
    wszSubject = argv[3];

    // First, look up the user's SID using the specified account name.
    // Call LookupAccountName twice; first to find the size of the
    // SID, and a second time to retrieve the SID.
    LookupAccountName(NULL,wszAccount,pSid,&cbSid, wszDomain,
&cchDomain,&SidType);
    if(cbSid == 0)
    {
            DisplayErrorBox(L"LookupAccountName()");
            exit(1);
    }
      else
            wprintf(L"LookupAccountName() looks fine!\n");

    pSid = (PSID)malloc(cbSid);

    if(!pSid)
    {
            DisplayErrorBox(L"malloc()");
            exit(1);
    }
      else
            wprintf(L"SID allocated successfully!\n");

    wszDomain = (LPWSTR)malloc(cchDomain * sizeof(WCHAR));

    if(!wszDomain)
    {
            DisplayErrorBox(L"malloc()");
            exit(1);
    }
      else
            wprintf(L"malloc() for domain name looks fine!\n");

    if(!LookupAccountName(NULL,wszAccount,pSid,&cbSid,
wszDomain,&cchDomain,&SidType))
    {
            DisplayErrorBox(L"malloc()");
            exit(1);
    }
      else
            wprintf(L"LookupAccountName() looks fine!\n");

    // Obtain the user's certificate.
    // Search the TrustedPeople store for the specified subject name.
    // Anyone who has encrypted a file on the computer has an
    // encryption certificate placed the TrustedPeople store by the
    // system. It is likely that the user has a matching private key.
    hStore = CertOpenSystemStore( (HCRYPTPROV)NULL,L"TrustedPeople");
    if (!hStore)
    {
            DisplayErrorBox(L"CertOpenSystemStore()");
    }
      else
            wprintf(L"CertOpenSystemStore() looks fine!\n");

    pCertContext = CertFindCertificateInStore(hStore,
                           X509_ASN_ENCODING | PKCS_7_ASN_ENCODING,
                           0,
                           CERT_FIND_SUBJECT_STR,
                           (VOID*)wszSubject,
                           NULL);
    if(!pCertContext)
    {
            DisplayErrorBox(L"FindCertificateInStore()");
            exit(1);
    }
      else
            wprintf(L"FindCertificateInStore() looks fine!\n");

    // Create the ENCRYPTION_CERTIFICATE using the cert context and
    // the user's SID.
    pEfsEncryptionCert =
(PENCRYPTION_CERTIFICATE)malloc(sizeof(ENCRYPTION_CERTIFICATE));

    if(!pEfsEncryptionCert)
    {
            DisplayErrorBox(L"malloc()");
            exit(1);
    }
      else
            wprintf(L"malloc() for structure looks fine!\n");

    pEfsEncryptionCert->cbTotalLength = sizeof(ENCRYPTION_CERTIFICATE);
    pEfsEncryptionCert->pUserSid = (SID *)pSid;
    pEfsEncryptionCert->pCertBlob =
(PEFS_CERTIFICATE_BLOB)malloc(sizeof(EFS_CERTIFICATE_BLOB));

    if(!pEfsEncryptionCert->pCertBlob)
    {
            DisplayErrorBox(L"malloc()");
            exit(1);
    }
      else
            wprintf(L"malloc() cert blob allocated!\n");

    pEfsEncryptionCert->pCertBlob->dwCertEncodingType = pCertContext-
>dwCertEncodingType;
    pEfsEncryptionCert->pCertBlob->cbData = pCertContext->cbCertEncoded;
    pEfsEncryptionCert->pCertBlob->pbData = pCertContext->pbCertEncoded;

    // AddUsersToEncryptedFile takes an ENCRYPTION_CERTIFICATE_LIST;
    // create one with only one ENCRYPTION_CERTIFICATE in it.
    pEfsEncryptionCertList =
(PENCRYPTION_CERTIFICATE_LIST)malloc(sizeof(ENCRYPTION_CERTIFICATE_LIST));

    if(!pEfsEncryptionCertList)
    {
            DisplayErrorBox(L"malloc()");
            exit(1);
    }
      else
            wprintf(L"malloc() - structure allocation looks fine!\n");

    pEfsEncryptionCertList->nUsers = 1;
    pEfsEncryptionCertList->pUsers = &pEfsEncryptionCert;

    // Call the API to add the user.
      // Adds user keys to the specified encrypted file.
    dwResult = AddUsersToEncryptedFile(wszFile,pEfsEncryptionCertList);
    if(dwResult == ERROR_SUCCESS)
    {
        wprintf(L"The user was successfully added to the file.\n");
    }
    else
    {
            DisplayErrorBox(L"AddUsersToEncryptedFile()");
    }

    // Clean up all allocated resources
      if(pSid)
      {
            wprintf(L"Freeing up the SID...\n");
            free(pSid);
      }

    if(wszDomain)
      {
            wprintf(L"Freeing up the domain name...\n");
            free(wszDomain);
      }

    if(pCertContext)
      {
            wprintf(L"Freeing up the CertFreeCertificateContext...\n");
            CertFreeCertificateContext(pCertContext);
      }

    if(hStore)
      {
            wprintf(L"Closing the cert store...\n");
            CertCloseStore(hStore,CERT_CLOSE_STORE_FORCE_FLAG);
      }

      wprintf(L"Freeing up other allocated resources...\n");

    if(pEfsEncryptionCertList)
    {
            // Just free up the allocated storage pUsers[0]
        //if (pEfsEncryptionCertList->pUsers)
            //    {
            if(pEfsEncryptionCertList->pUsers[0])
            {
                  if((pEfsEncryptionCertList->pUsers[0])->pCertBlob)
                        free((pEfsEncryptionCertList->pUsers[0])->pCertBlob);
                  free(pEfsEncryptionCertList->pUsers[0]);
            }
            //      free(pEfsEncryptionCertList->pUsers);
            //   }
        free(pEfsEncryptionCertList);
    }

    wprintf(L"The program ran to completion without error.\n");
    exit(0);
}

void DisplayErrorBox(LPTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL );

    // Display the error message and clean up
    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,

(lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(WCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(WCHAR),  L"%s failed with error %d:
%s",
        lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, L"Error", MB_OK);

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}
```

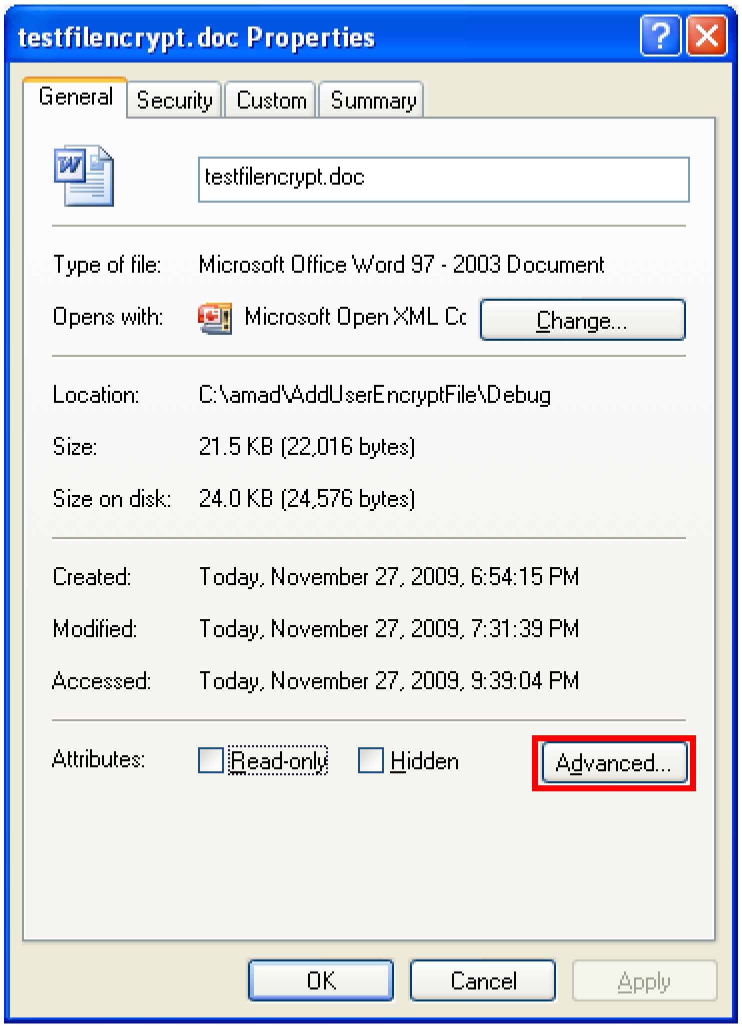
To test this program, it is better to run it in a domain based, active directory environment. In the following test, we just run it on standalone Windows XP Pro.
Firstly, we create a word file, testfilencrypt.doc.



Name	Size	Type
AddUserEncryptFile.exe	34 KB	Application
AddUserEncryptFile.ilk	351 KB	Incremental Linker File
AddUserEncryptFile.pdb	491 KB	Program Debug Database
testfilencrypt.doc	22 KB	Microsoft Office Word 97 - 2003 Docume
testfilencryptJohnny.doc	22 KB	Microsoft Office Word 97 - 2003 Docume

Type: Microsoft Office Word 97 - 2003 Document Author: Date Modified: 21.5 KB

Next, we invoke the file's property page. Click the Advanced button.



testfilencrypt.doc Properties

General | Security | Custom | Summary

testfilencrypt.doc

Type of file: Microsoft Office Word 97 - 2003 Document

Opens with: Microsoft Open XML Cc

Location: C:\amad\AddUserEncryptFile\Debug

Size: 21.5 KB (22,016 bytes)

Size on disk: 24.0 KB (24,576 bytes)

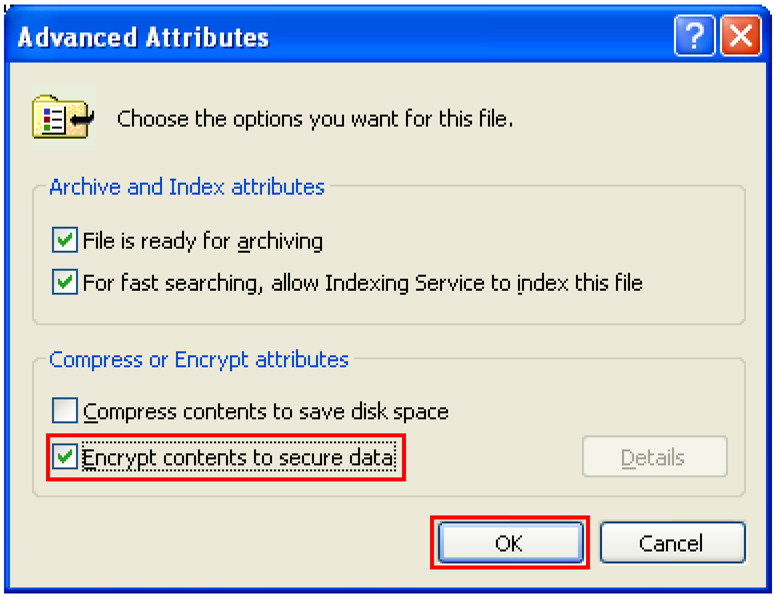
Created: Today, November 27, 2009, 6:54:15 PM

Modified: Today, November 27, 2009, 7:31:39 PM

Accessed: Today, November 27, 2009, 9:39:04 PM

Attributes: ☐ Read-only ☐ Hidden

Tick the Encrypt contents to secure data tick box and click OK.



Advanced Attributes

Choose the options you want for this file.

Archive and Index attributes

☑ File is ready for archiving

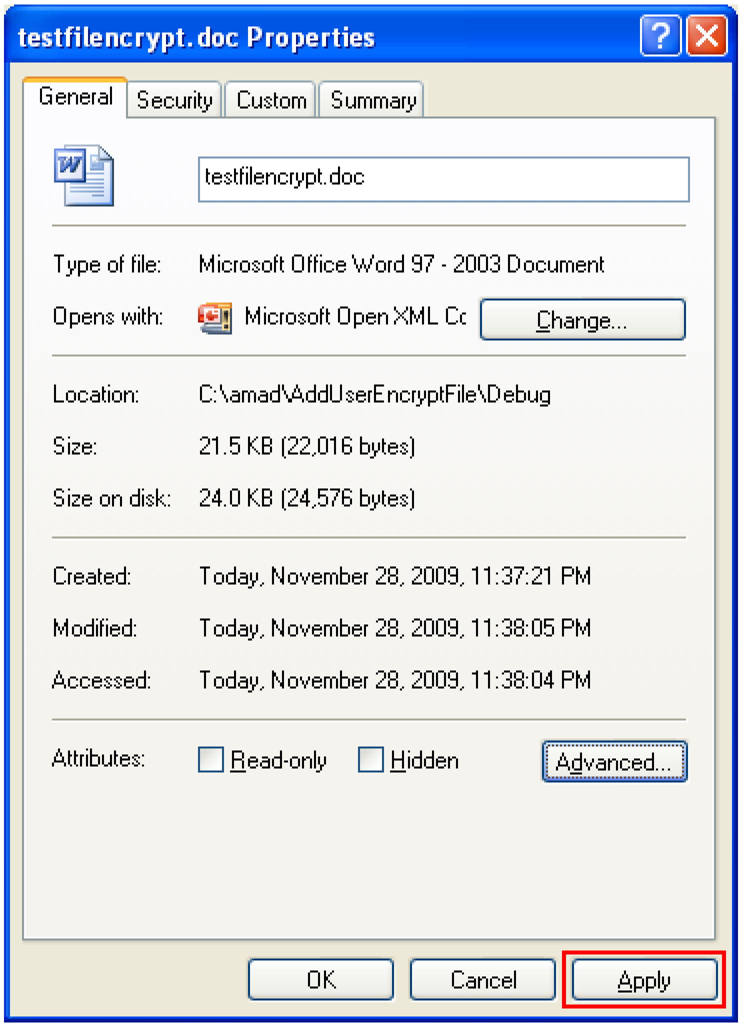
☑ For fast searching, allow Indexing Service to index this file

Compress or Encrypt attributes

☐ Compress contents to save disk space

☑ Encrypt contents to secure data

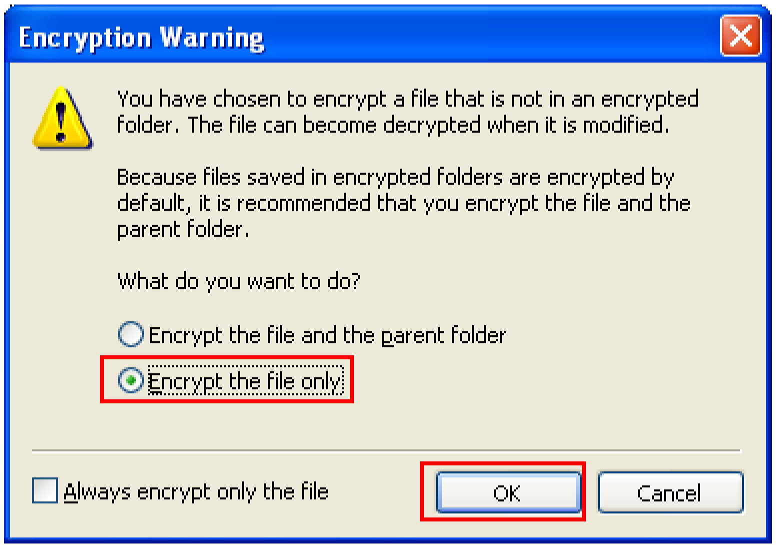
Then click the OK button.


testfilencrypt.doc Properties

testfilencrypt.doc

Type of file:	Microsoft Office Word 97 - 2003 Document
Opens with:	Microsoft Open XML Cc
Location:	C:\amad\AddUserEncryptFile\Debug
Size:	21.5 KB (22,016 bytes)
Size on disk:	24.0 KB (24,576 bytes)
Created:	Today, November 28, 2009, 11:37:21 PM
Modified:	Today, November 28, 2009, 11:38:05 PM
Accessed:	Today, November 28, 2009, 11:38:04 PM
Attributes:	☐ Read-only ☐ Hidden

Select Encrypt the file only and click OK.


Encryption Warning

You have chosen to encrypt a file that is not in an encrypted folder. The file can become decrypted when it is modified.

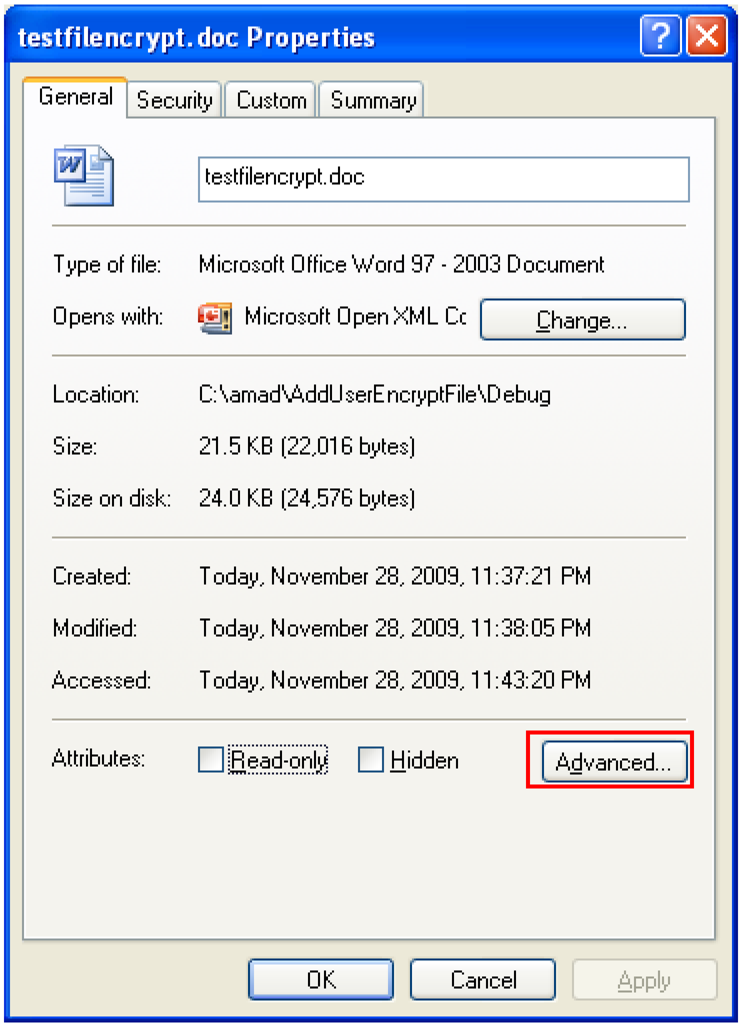
Because files saved in encrypted folders are encrypted by default, it is recommended that you encrypt the file and the parent folder.

What do you want to do?

○ Encrypt the file and the parent folder
● Encrypt the file only

☐ Always encrypt only the file

Then click the Advanced button again.



testfilencrypt.doc

Type of file: Microsoft Office Word 97 - 2003 Document

Opens with: Microsoft Open XML Cc

Location: C:\amad\AddUserEncryptFile\Debug

Size: 21.5 KB (22,016 bytes)

Size on disk: 24.0 KB (24,576 bytes)

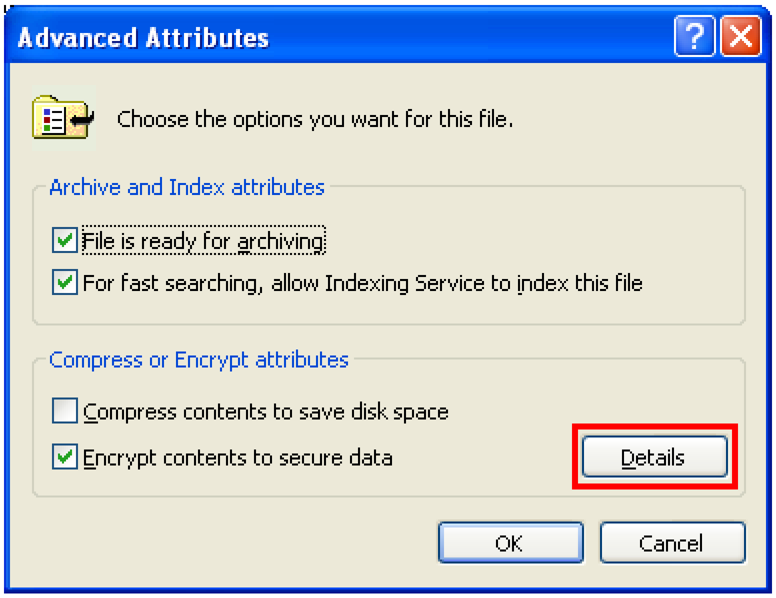
Created: Today, November 28, 2009, 11:37:21 PM

Modified: Today, November 28, 2009, 11:38:05 PM

Accessed: Today, November 28, 2009, 11:43:20 PM

Attributes: ☐ Read-only ☐ Hidden

Click the Details button.



Choose the options you want for this file.

Archive and Index attributes

☑ File is ready for archiving

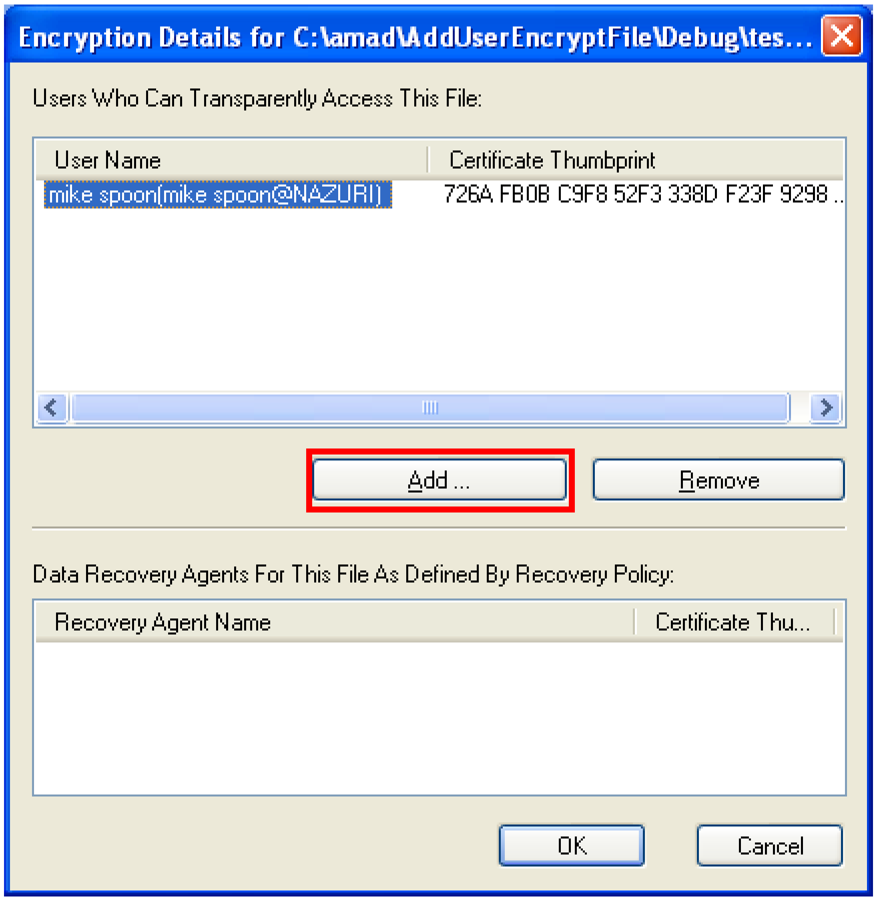
☑ For fast searching, allow Indexing Service to index this file

Compress or Encrypt attributes

☐ Compress contents to save disk space

☑ Encrypt contents to secure data

The certificate for user mike spoon (in this case, the logged user) will be created using local certificate store. Manually we can add other user, click the Add button.


Encryption Details for C:\amad\AddUserEncryptFile\Debug\tes...

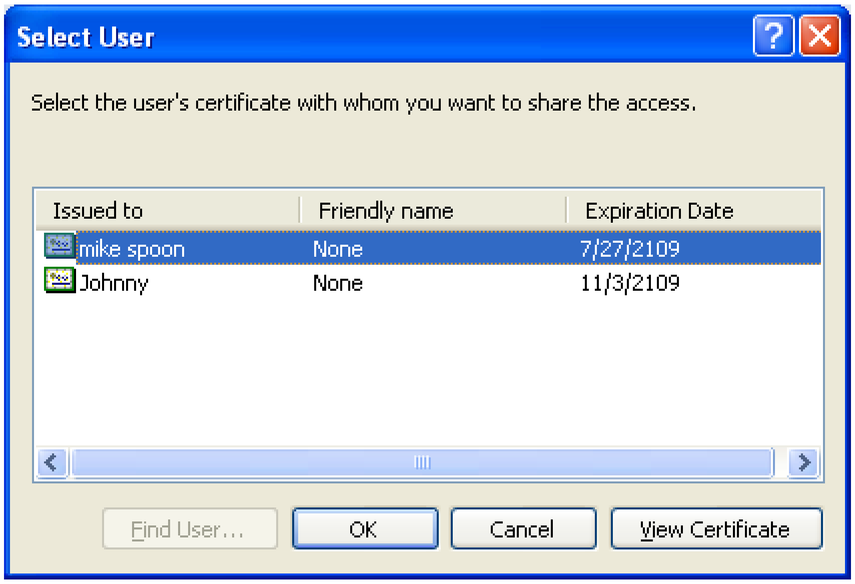
Users Who Can Transparently Access This File:

User Name	Certificate Thumbprint
mike spoon(mike spoon@NAZURI)	726A FB0B C9F8 52F3 338D F23F 9298 ..

Data Recovery Agents For This File As Defined By Recovery Policy:

Recovery Agent Name	Certificate Thu...

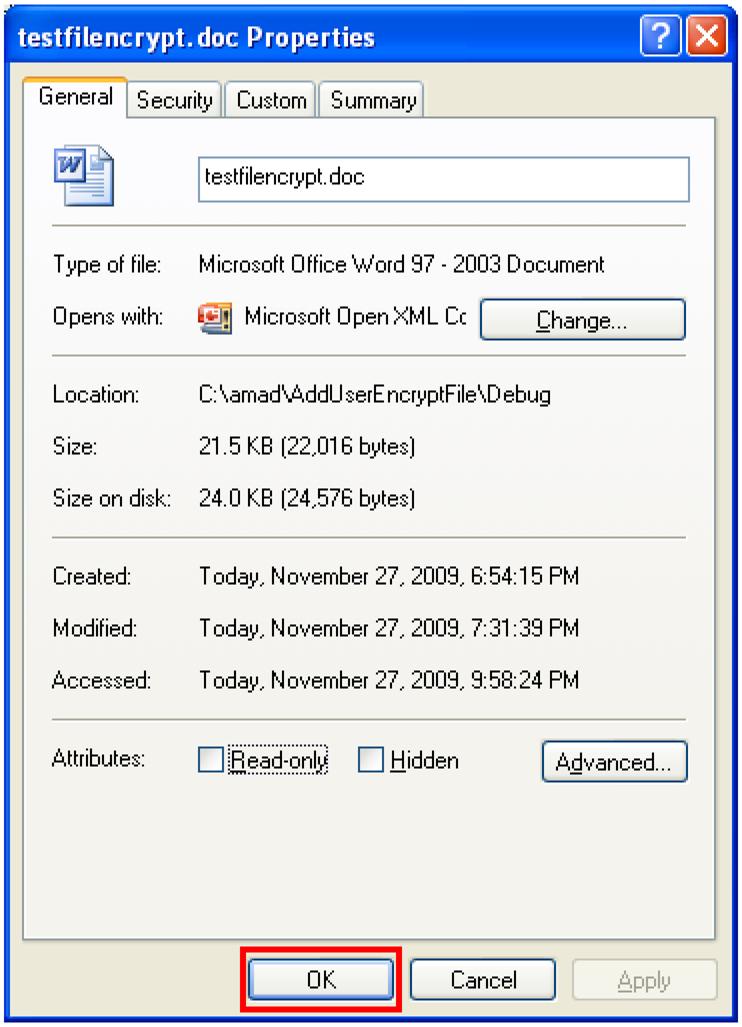
In this case we have another user which having a certificate.


Select User

Select the user's certificate with whom you want to share the access.

Issued to	Friendly name	Expiration Date
mike spoon	None	7/27/2109
Johnny	None	11/3/2109

Just click the Cancel button but for the following image, click OK button.



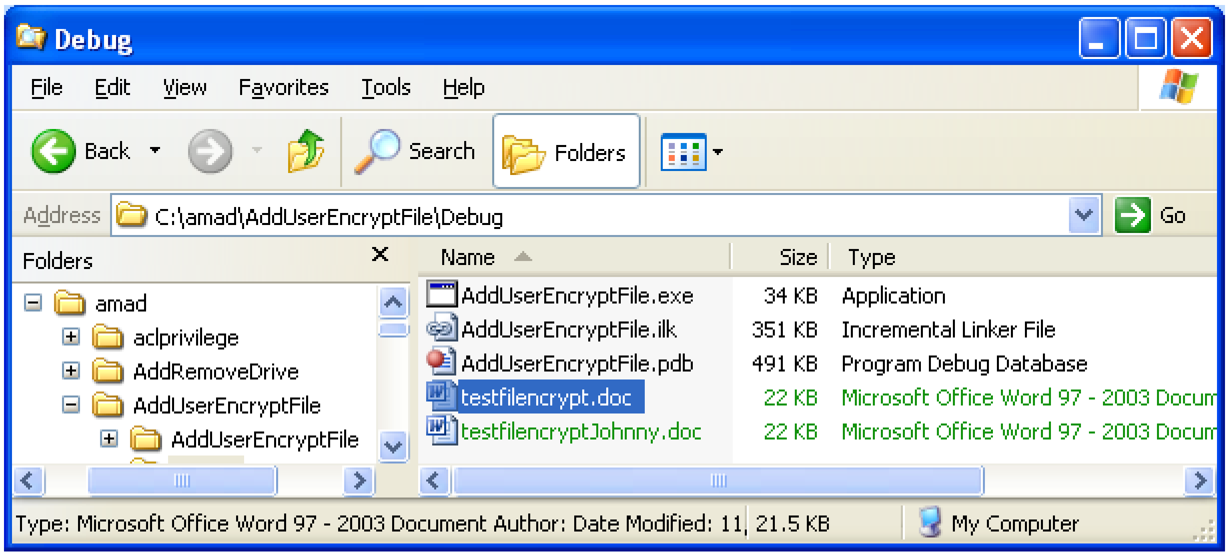
testfilencrypt.doc Properties

General | Security | Custom | Summary

testfilencrypt.doc

Type of file: Microsoft Office Word 97 - 2003 Document
Opens with: Microsoft Open XML Cc
Location: C:\amad\AddUserEncryptFile\Debug
Size: 21.5 KB (22,016 bytes)
Size on disk: 24.0 KB (24,576 bytes)
Created: Today, November 27, 2009, 6:54:15 PM
Modified: Today, November 27, 2009, 7:31:39 PM
Accessed: Today, November 27, 2009, 9:58:24 PM
Attributes: ☐ Read-only ☐ Hidden

In Windows, the encrypted file is shown in green colour.



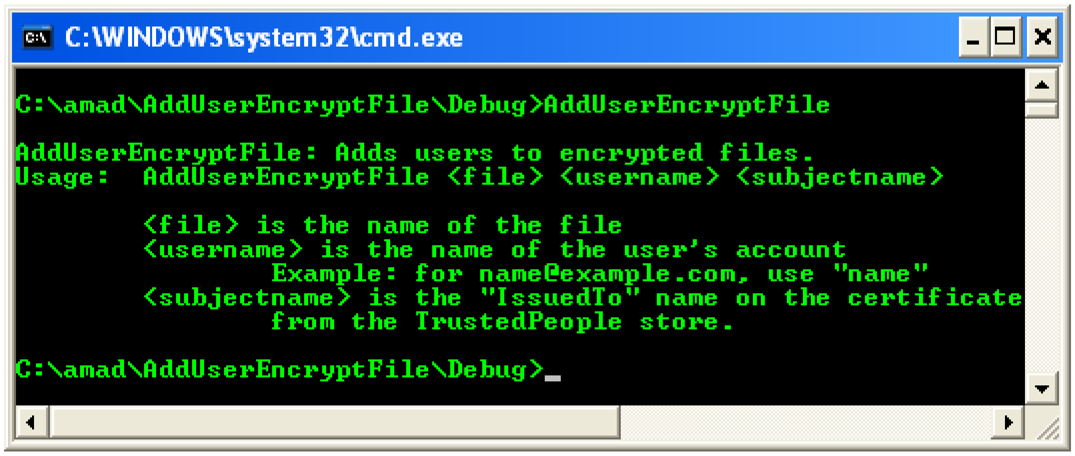
Debug

Address: C:\amad\AddUserEncryptFile\Debug

Name	Size	Type
AddUserEncryptFile.exe	34 KB	Application
AddUserEncryptFile.ilk	351 KB	Incremental Linker File
AddUserEncryptFile.pdb	491 KB	Program Debug Database
testfilencrypt.doc	22 KB	Microsoft Office Word 97 - 2003 Docum
testfilencryptJohnny.doc	22 KB	Microsoft Office Word 97 - 2003 Docum

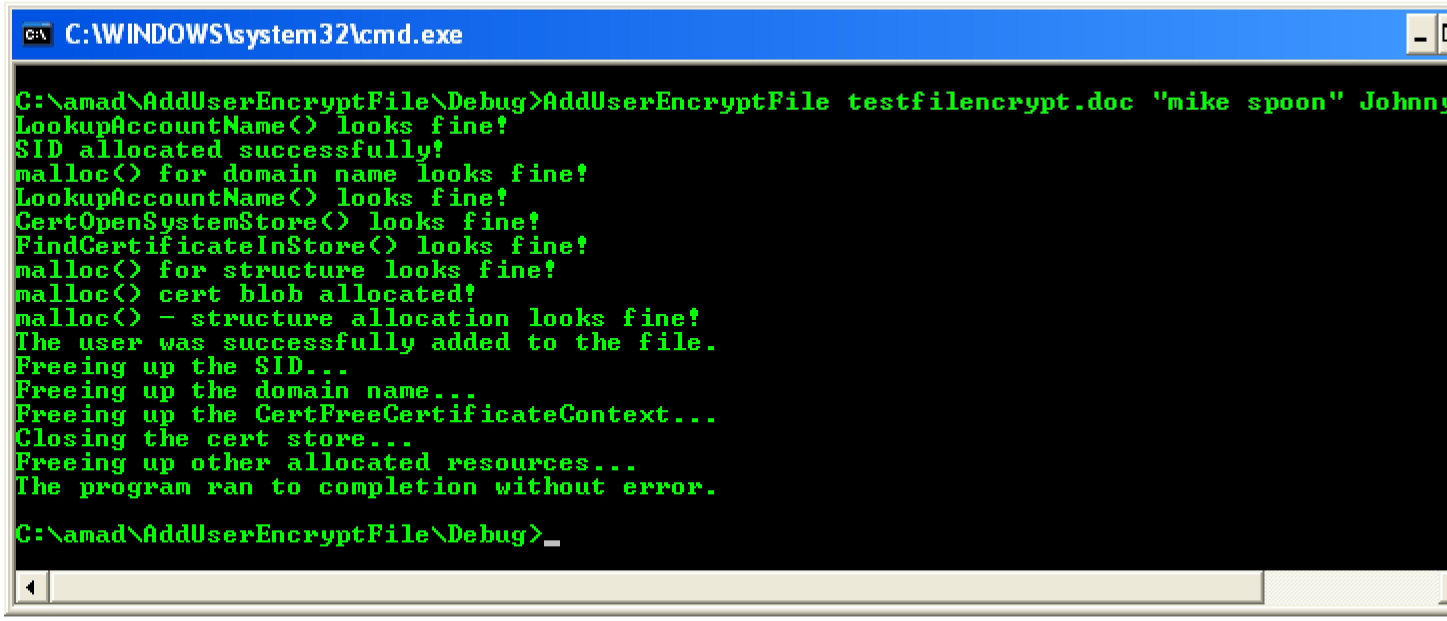
Type: Microsoft Office Word 97 - 2003 Document Author: Date Modified: 11, 21.5 KB

Next, run the program.



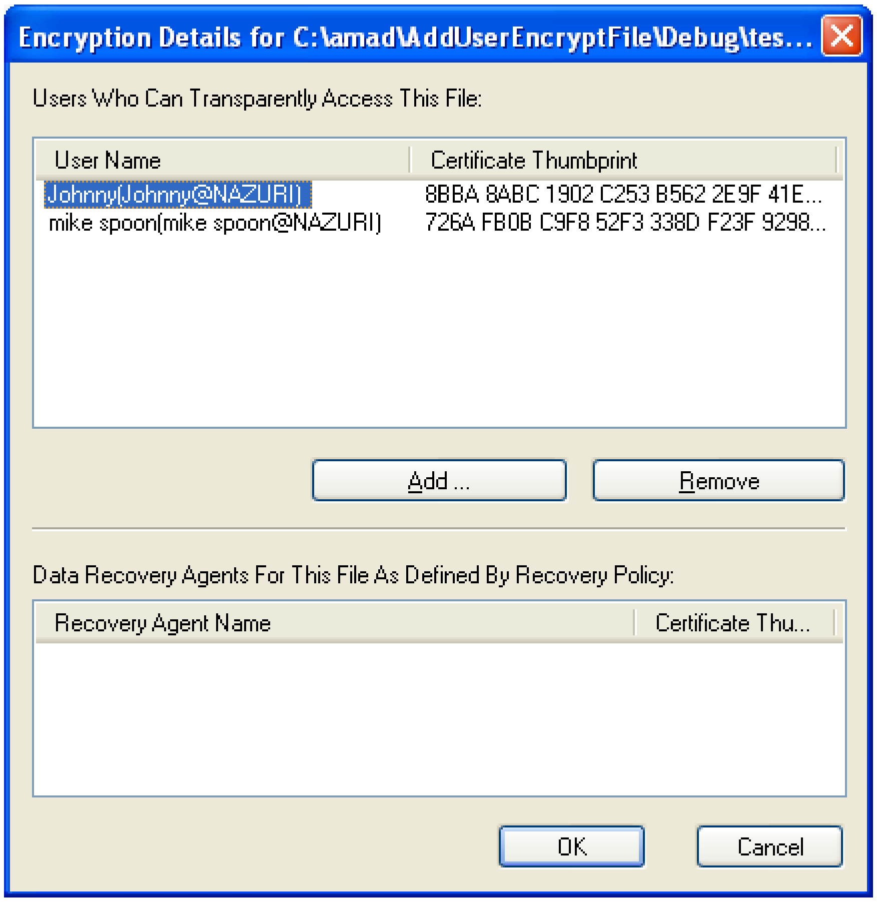
```
C:\amad\AddUserEncryptFile\Debug>AddUserEncryptFile

AddUserEncryptFile: Adds users to encrypted files.
Usage:  AddUserEncryptFile <file> <username> <subjectname>

        <file> is the name of the file
        <username> is the name of the user's account
                Example: for name@example.com, use "name"
        <subjectname> is the "IssuedTo" name on the certificate
                from the TrustedPeople store.

C:\amad\AddUserEncryptFile\Debug>_
```

We are adding a user Johnny (also has a local certificate) to testfilencrypt.doc which owned by mike spoon.



```
C:\amad\AddUserEncryptFile\Debug>AddUserEncryptFile testfilencrypt.doc "mike spoon" Johnny
LookupAccountName() looks fine!
SID allocated successfully!
malloc() for domain name looks fine!
LookupAccountName() looks fine!
CertOpenSystemStore() looks fine!
FindCertificateInStore() looks fine!
malloc() for structure looks fine!
malloc() cert blob allocated!
malloc() - structure allocation looks fine!
The user was successfully added to the file.
Freeing up the SID...
Freeing up the domain name...
Freeing up the CertFreeCertificateContext...
Closing the cert store...
Freeing up other allocated resources...
The program ran to completion without error.

C:\amad\AddUserEncryptFile\Debug>_
```

Finally, let verify the task. The user seems was added successfully and it is bad if the purposes are misused!


Encryption Details for C:\amad\AddUserEncryptFile\Debug\tes...

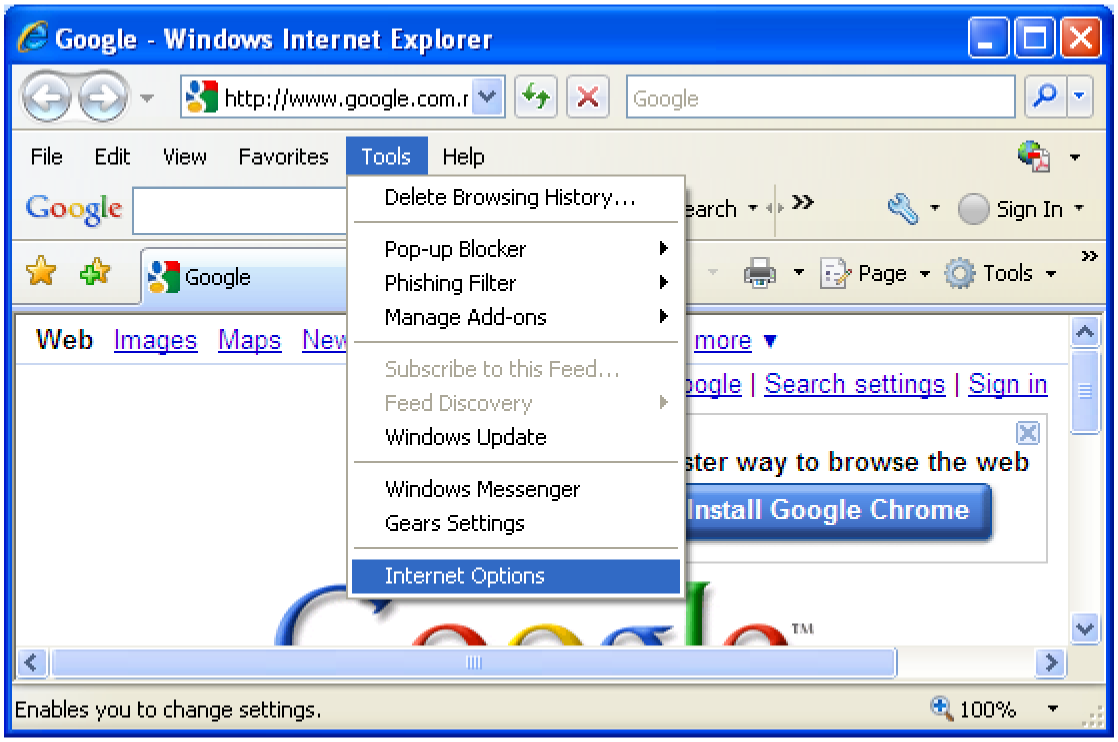
Users Who Can Transparently Access This File:

User Name	Certificate Thumbprint
Johnny(Johnny@NAZURI)	8BBA 8ABC 1902 C253 B562 2E9F 41E...
mike spoon(mike spoon@NAZURI)	726A FB0B C9F8 52F3 338D F23F 9298...

Data Recovery Agents For This File As Defined By Recovery Policy:

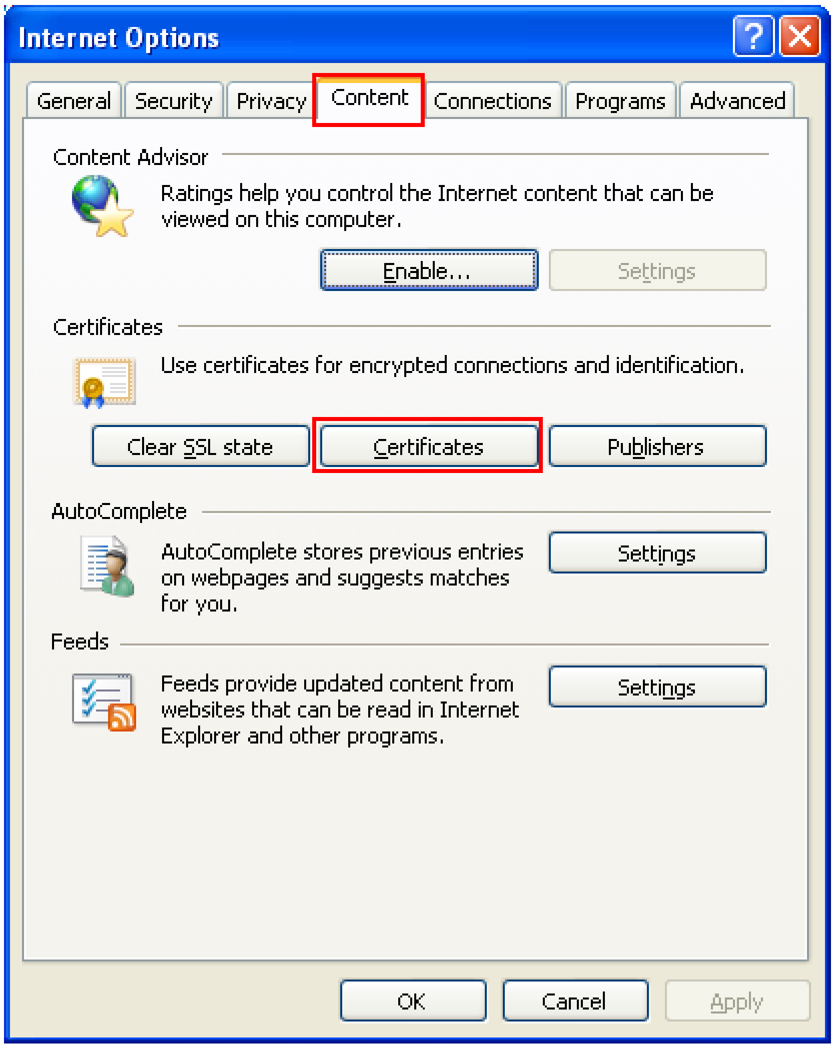
Recovery Agent Name	Certificate Thu...

The self-signed certificate can be seen in Internet Explorer. Select Tools > Internet Options menu.



Enables you to change settings.

Click the Content tab and then Certificate button.



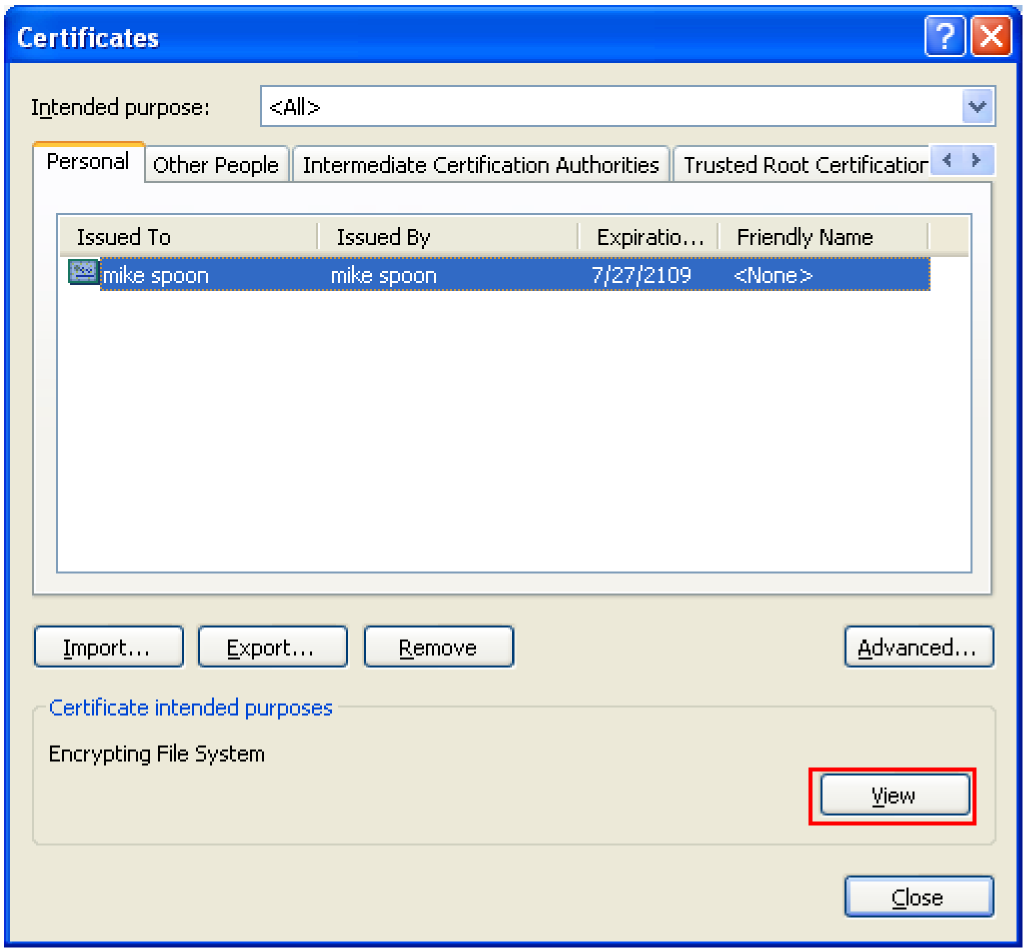
Ratings help you control the Internet content that can be viewed on this computer.

Use certificates for encrypted connections and identification.

AutoComplete stores previous entries on webpages and suggests matches for you.

Feeds provide updated content from websites that can be read in Internet Explorer and other programs.

Select the certificate for the current user. In this case, mike spoon and click the View button.

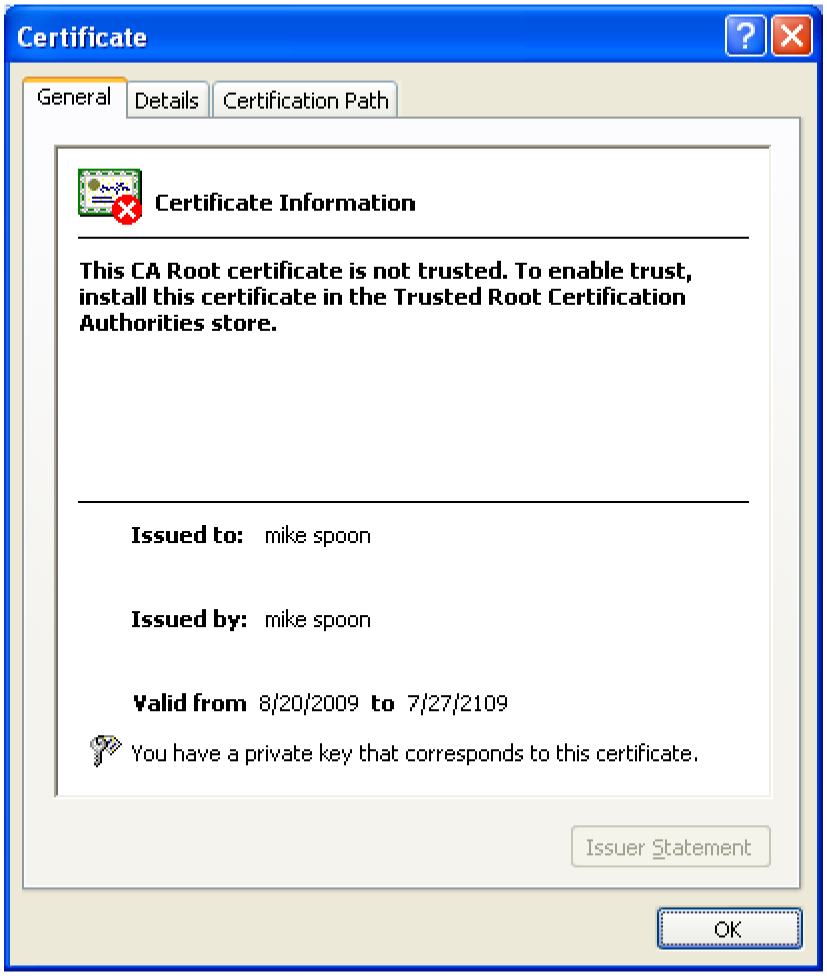


Issued To	Issued By	Expiratio...	Friendly Name
mike spoon	mike spoon	7/27/2109	<None>

Certificate intended purposes

Encrypting File System

Well, the CA Root certificate is not trusted. Let make it a trusted CA Root certificate by putting the certificate in the Trusted Root Certification Authorities store. Click the OK button to close the page.



This CA Root certificate is not trusted. To enable trust, install this certificate in the Trusted Root Certification Authorities store.

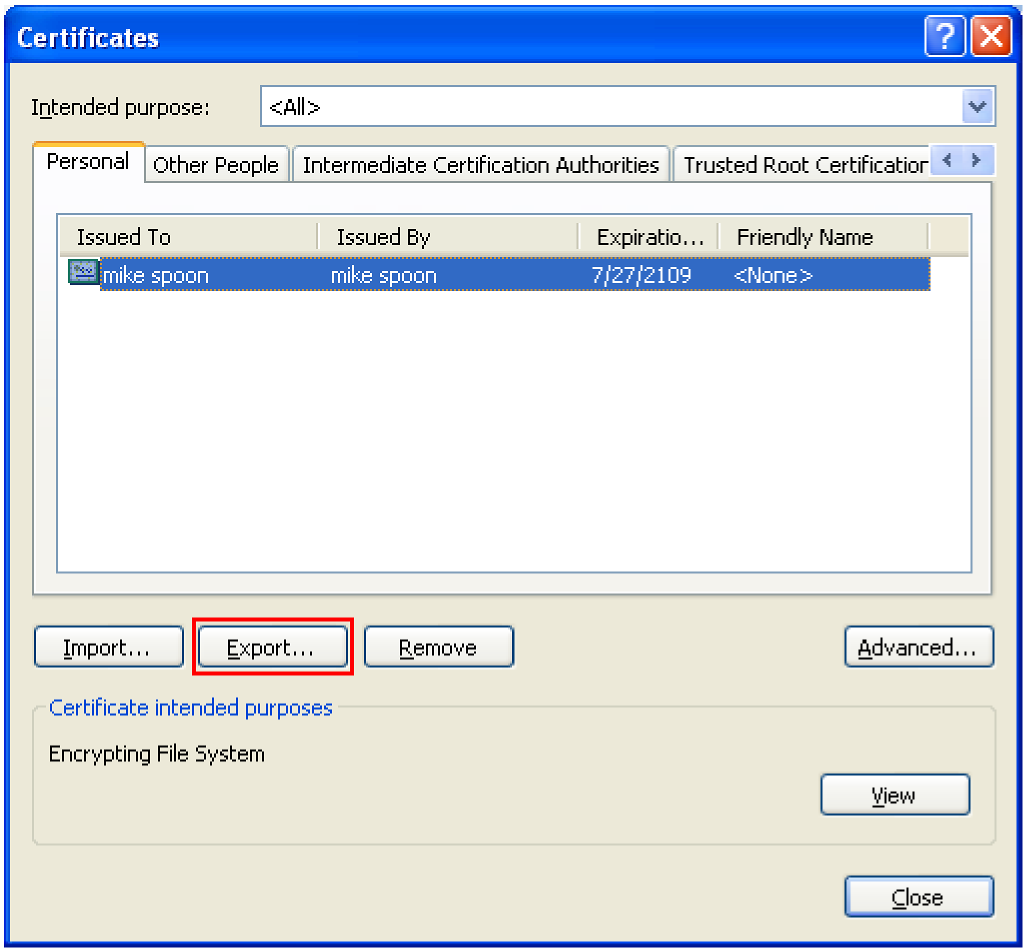
Issued to: mike spoon

Issued by: mike spoon

Valid from 8/20/2009 to 7/27/2109

You have a private key that corresponds to this certificate.

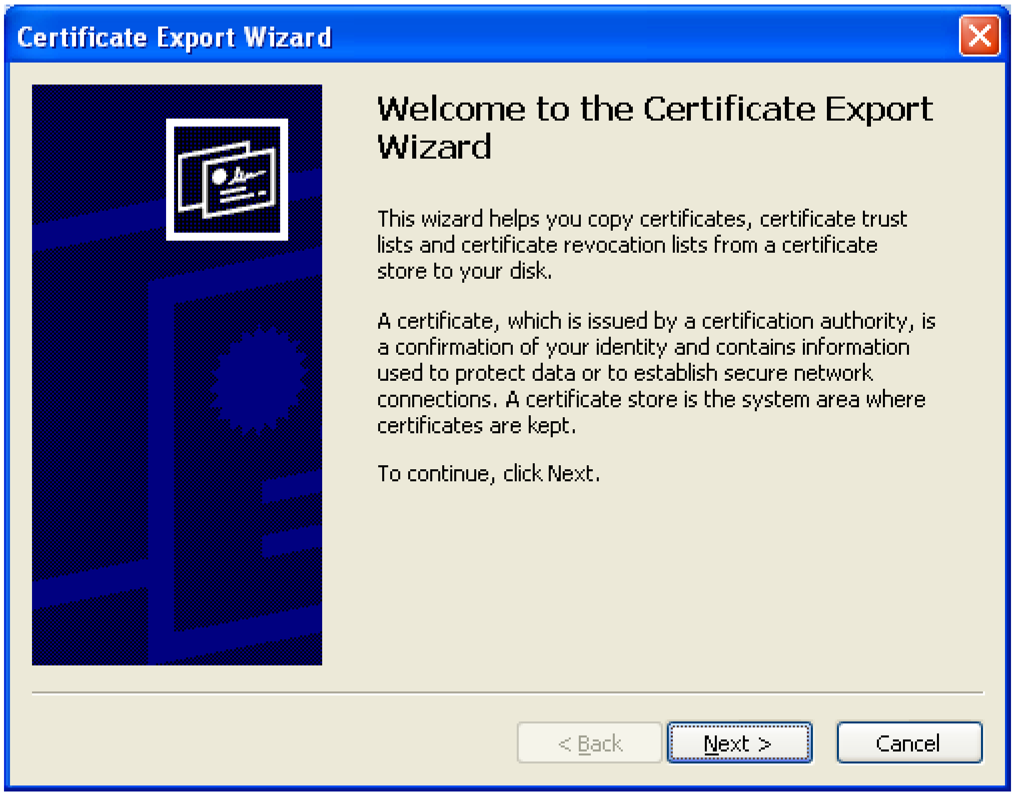
Firstly, we need to export the certificate details, and then we can import it into the Trusted Root Certification Authorities store. Select the current user and click the export button.



Issued To	Issued By	Expiratio...	Friendly Name
mike spoon	mike spoon	7/27/2109	<None>

Encrypting File System

The certificate export wizard launched. Click the Next button.


Certificate Export Wizard

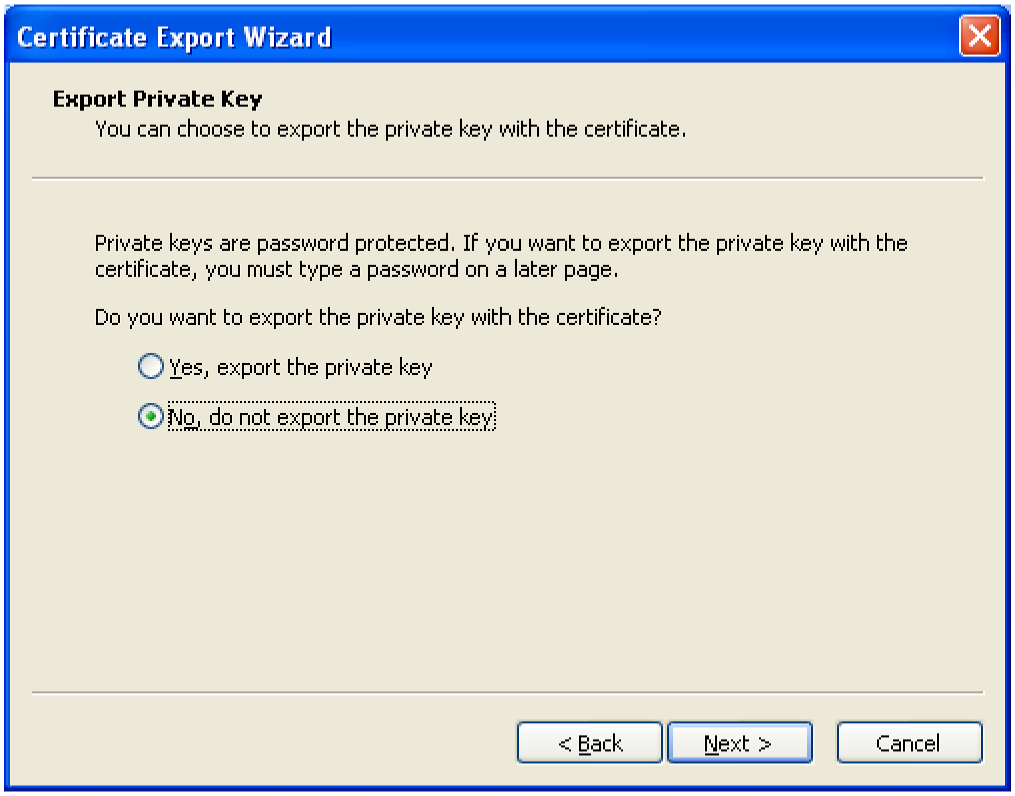
Welcome to the Certificate Export Wizard

This wizard helps you copy certificates, certificate trust lists and certificate revocation lists from a certificate store to your disk.

A certificate, which is issued by a certification authority, is a confirmation of your identity and contains information used to protect data or to establish secure network connections. A certificate store is the system area where certificates are kept.

To continue, click Next.

Select the second radio button and click Next. If you want to export the private key as well, please select the first radio button.


Certificate Export Wizard

Export Private Key

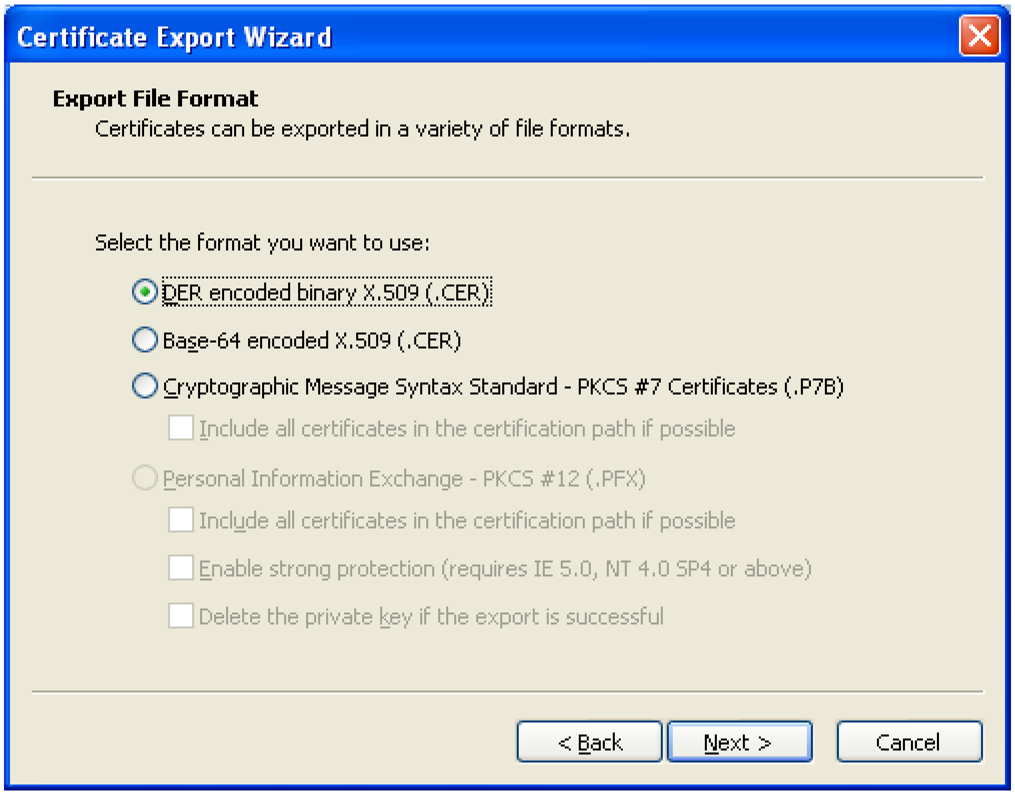
You can choose to export the private key with the certificate.

Private keys are password protected. If you want to export the private key with the certificate, you must type a password on a later page.

Do you want to export the private key with the certificate?

Yes, export the private key [✓ No, do not export the private key]

Select the first radio button and click the Next button. Again, depend on your choice and needs. Take note the file extension.

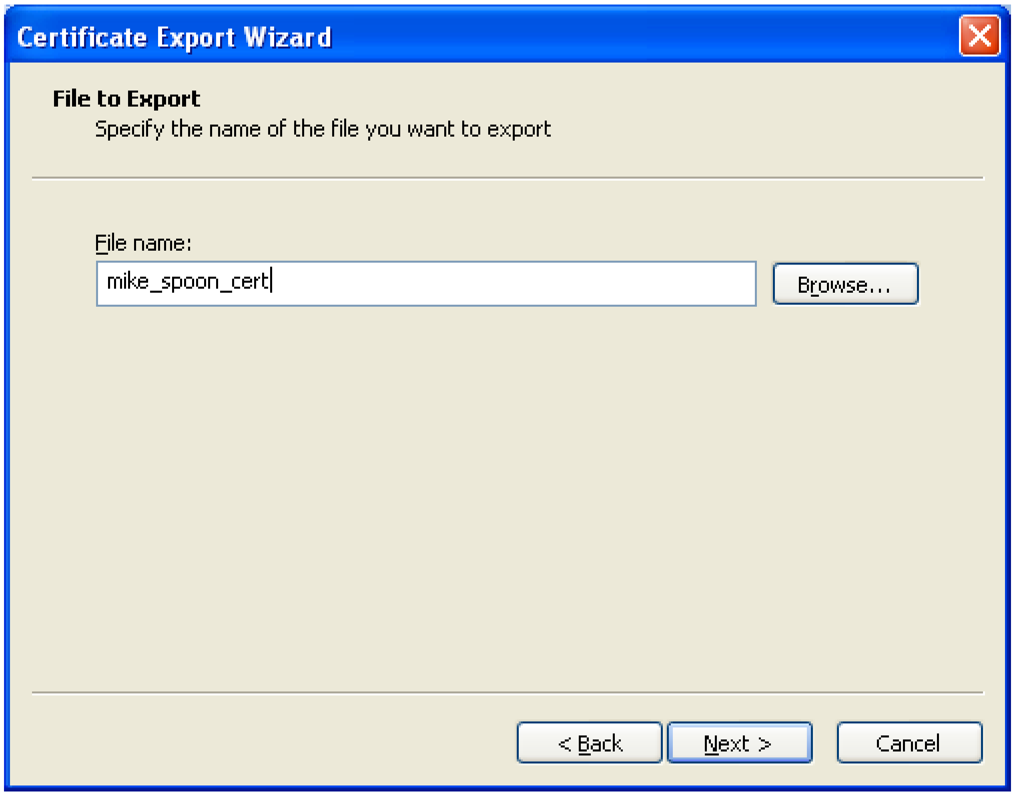


Export File Format
Certificates can be exported in a variety of file formats.

Select the format you want to use:

- [✓ DER encoded binary X.509 (.CER)]
- Base-64 encoded X.509 (.CER)
- Cryptographic Message Syntax Standard - PKCS #7 Certificates (.P7B)
  - Include all certificates in the certification path if possible
- Personal Information Exchange - PKCS #12 (.PFX)
  - Include all certificates in the certification path if possible
  - Enable strong protection (requires IE 5.0, NT 4.0 SP4 or above)
  - Delete the private key if the export is successful

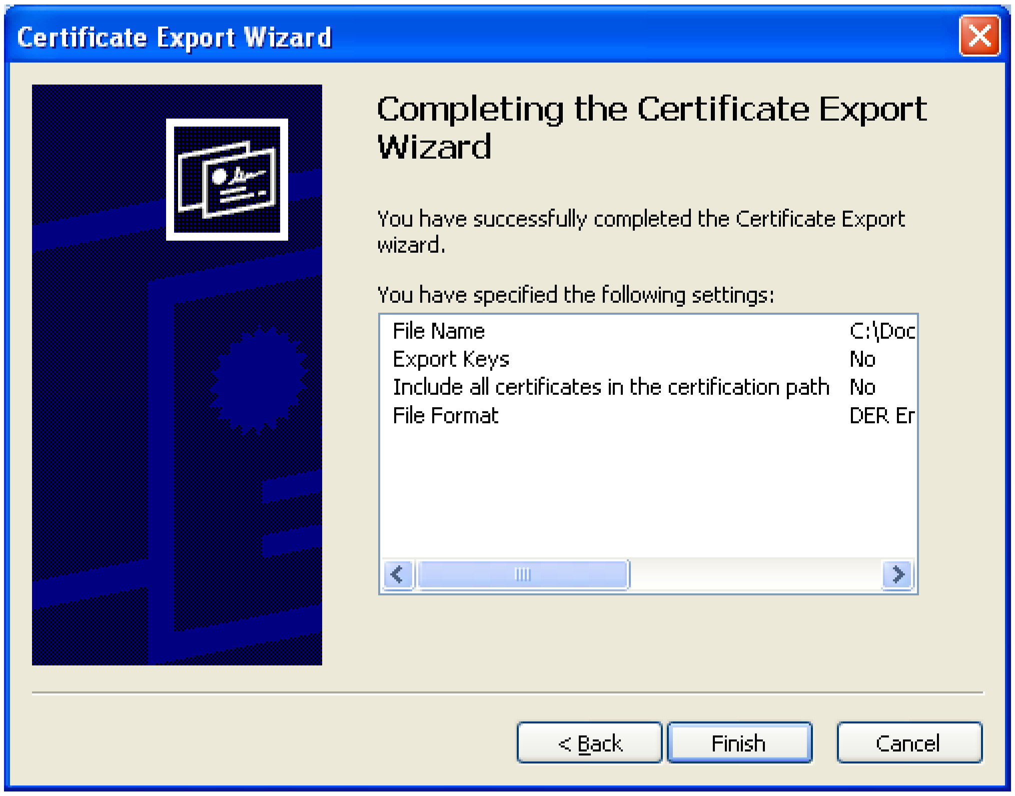
Type the certificate file name. The extension will be automatically attached as selected in the previous step. Select the file location to be saved using the Browse if needed. Click the Next button.



File to Export
Specify the name of the file you want to export

File name:
mike_spoon_cert

For the certificate export wizard summary page, review the settings. Use the Back button if you want to change any settings otherwise clicks the Finish button.



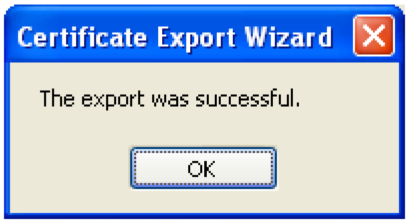
Completing the Certificate Export Wizard

You have successfully completed the Certificate Export wizard.

You have specified the following settings:

File Name	C:\Doc
Export Keys	No
Include all certificates in the certification path	No
File Format	DER Er

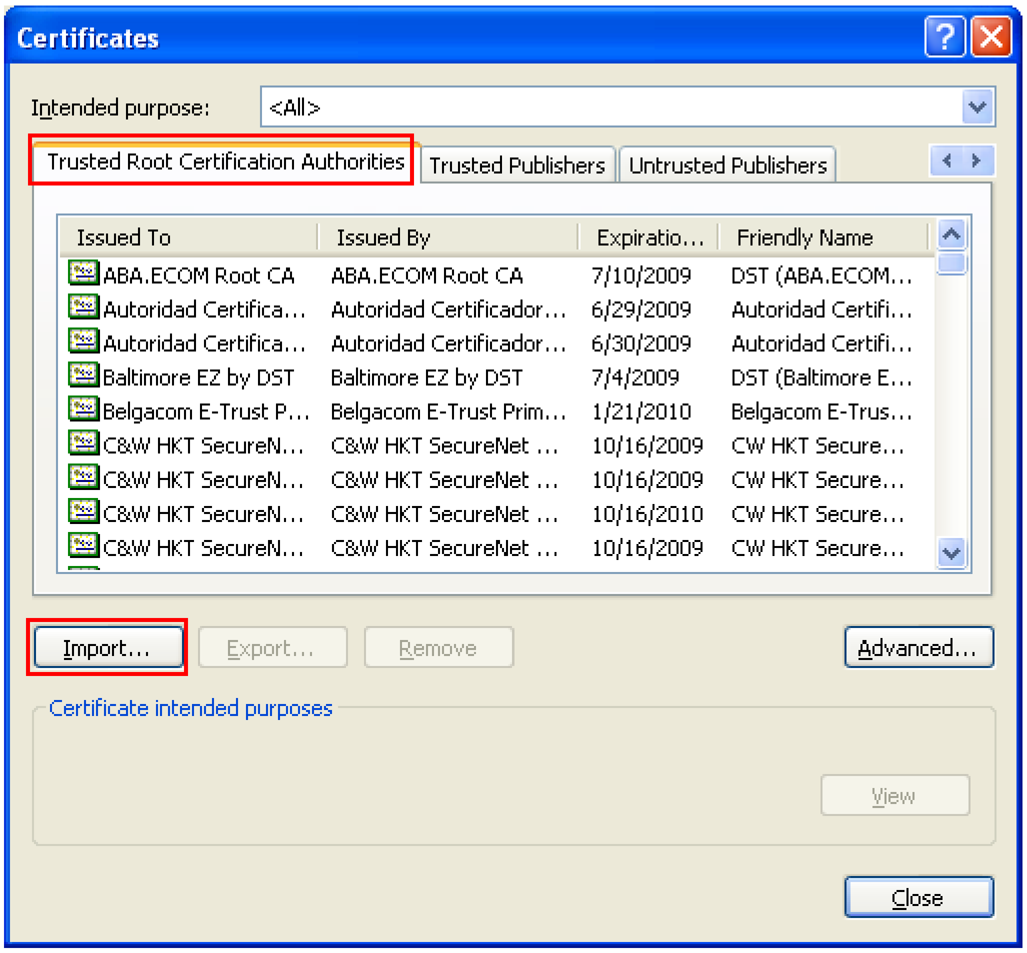
The following message box should be displayed if there is no error. Click OK.



Certificate Export Wizard

The export was successful.

Next, we are ready to import the certificate into the Trusted Root Certification Authorities store. Select the Trusted Root Certification Authorities tab and click the Import button.

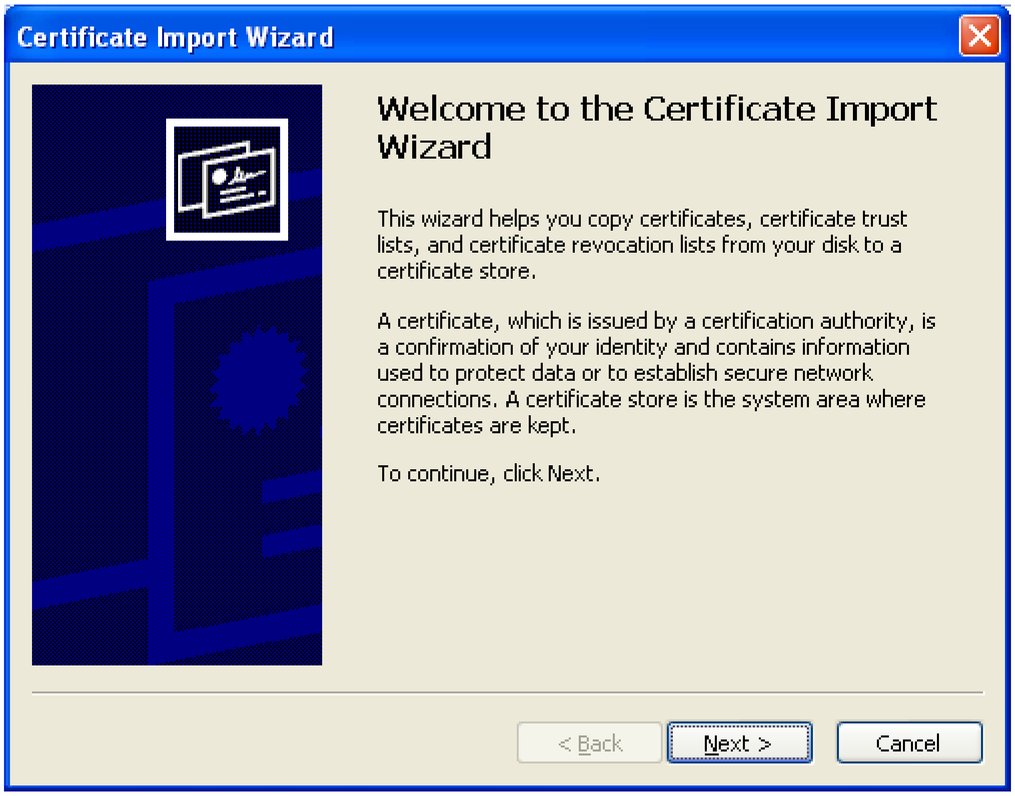


Certificates

Intended purpose: <All>

Issued To	Issued By	Expiratio...	Friendly Name
ABA.ECOM Root CA	ABA.ECOM Root CA	7/10/2009	DST (ABA.ECOM...
Autoridad Certifica...	Autoridad Certificador...	6/29/2009	Autoridad Certifi...
Autoridad Certifica...	Autoridad Certificador...	6/30/2009	Autoridad Certifi...
Baltimore EZ by DST	Baltimore EZ by DST	7/4/2009	DST (Baltimore E...
Belgacom E-Trust P...	Belgacom E-Trust Prim...	1/21/2010	Belgacom E-Trus...
C&W HKT SecureN...	C&W HKT SecureNet ...	10/16/2009	CW HKT Secure...
C&W HKT SecureN...	C&W HKT SecureNet ...	10/16/2009	CW HKT Secure...
C&W HKT SecureN...	C&W HKT SecureNet ...	10/16/2010	CW HKT Secure...
C&W HKT SecureN...	C&W HKT SecureNet ...	10/16/2009	CW HKT Secure...

When the certificate import wizard page launched, click the Next button.


Certificate Import Wizard

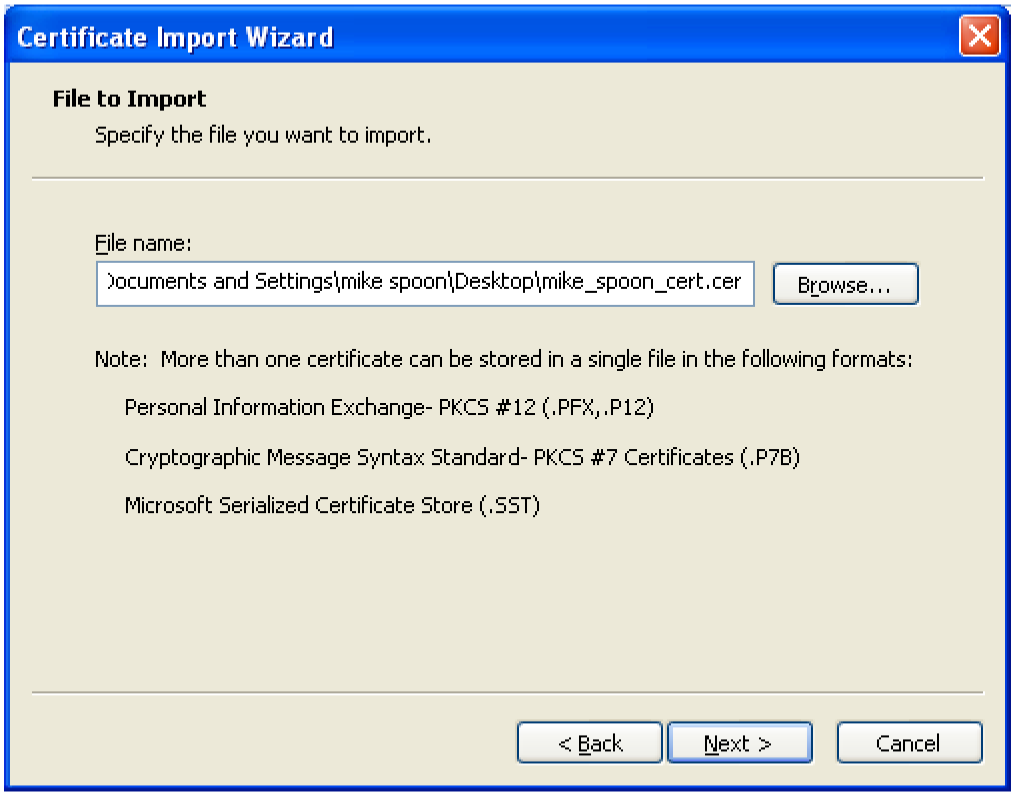
Welcome to the Certificate Import Wizard

This wizard helps you copy certificates, certificate trust lists, and certificate revocation lists from your disk to a certificate store.

A certificate, which is issued by a certification authority, is a confirmation of your identity and contains information used to protect data or to establish secure network connections. A certificate store is the system area where certificates are kept.

To continue, click Next.

By using the Browse button, find the previously saved certificate file and click the Next button.


Certificate Import Wizard

File to Import
Specify the file you want to import.

File name:
)ocuments and Settings\mike spoon\Desktop\mike_spoon_cert.cer

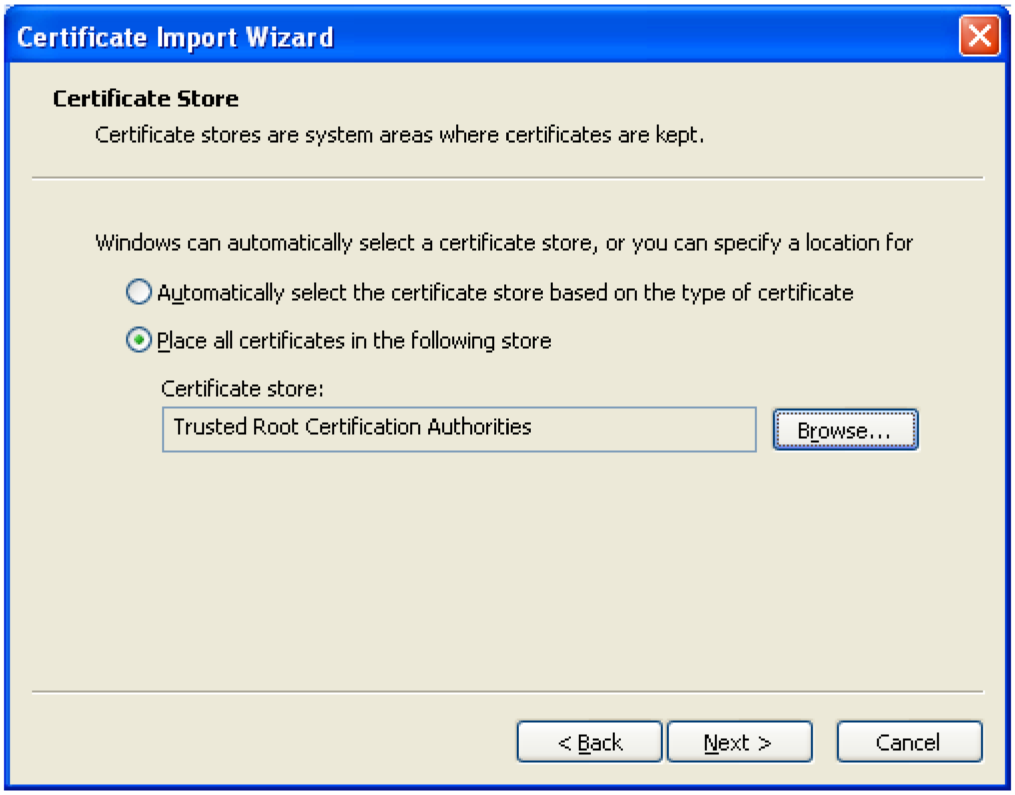
Note: More than one certificate can be stored in a single file in the following formats:

Personal Information Exchange- PKCS #12 (.PFX,.P12)

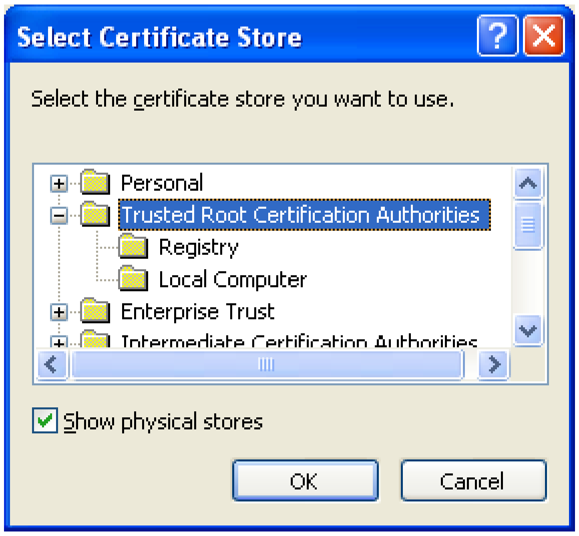
Cryptographic Message Syntax Standard- PKCS #7 Certificates (.P7B)

Microsoft Serialized Certificate Store (.SST)

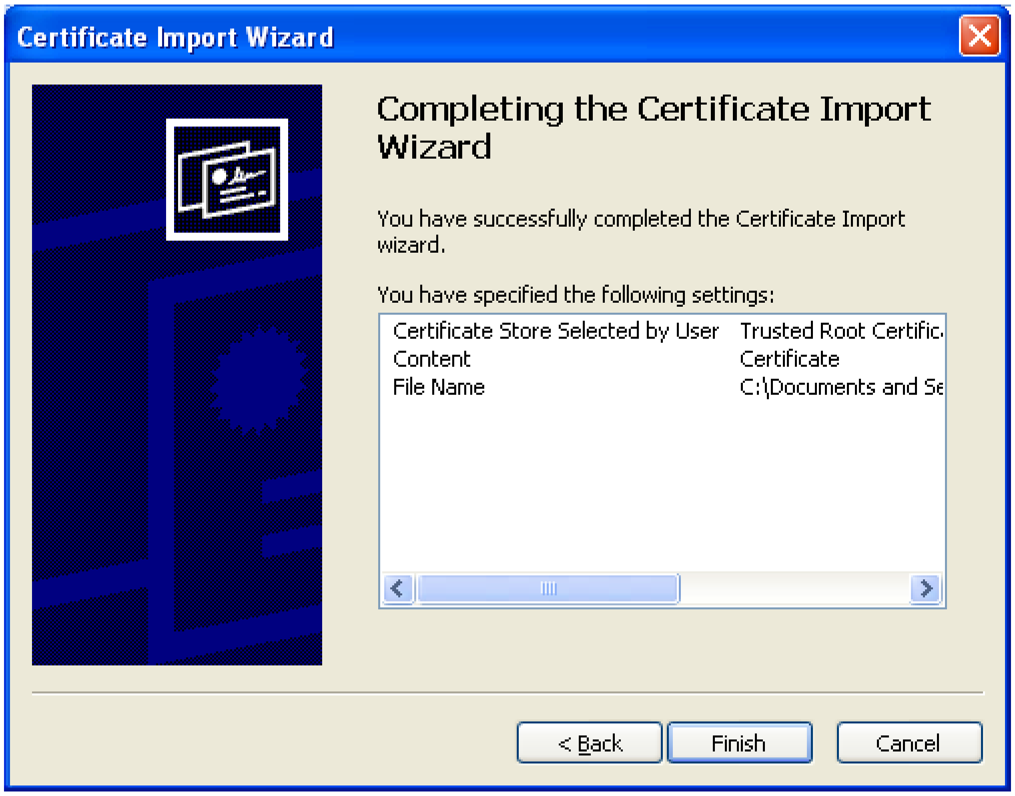
Select the appropriate certificate store. In this case it is Trusted Root Certification Authorities (default). Click the Next button.


Certificate Import Wizard
Certificate Store
Certificate stores are system areas where certificates are kept.
Windows can automatically select a certificate store, or you can specify a location for
Automatically select the certificate store based on the type of certificate
Place all certificates in the following store
Certificate store:
Trusted Root Certification Authorities
Browse...
< Back
Next >
Cancel

If you want to store it in other certificate store, click the Browse button.


Select Certificate Store
Select the certificate store you want to use.
Personal
Trusted Root Certification Authorities
Registry
Local Computer
Enterprise Trust
Intermediate Certification Authorities
Show physical stores
OK
Cancel

Clicks Finish for the certificate import wizard summary page. If you want to change any settings, use the Back button.



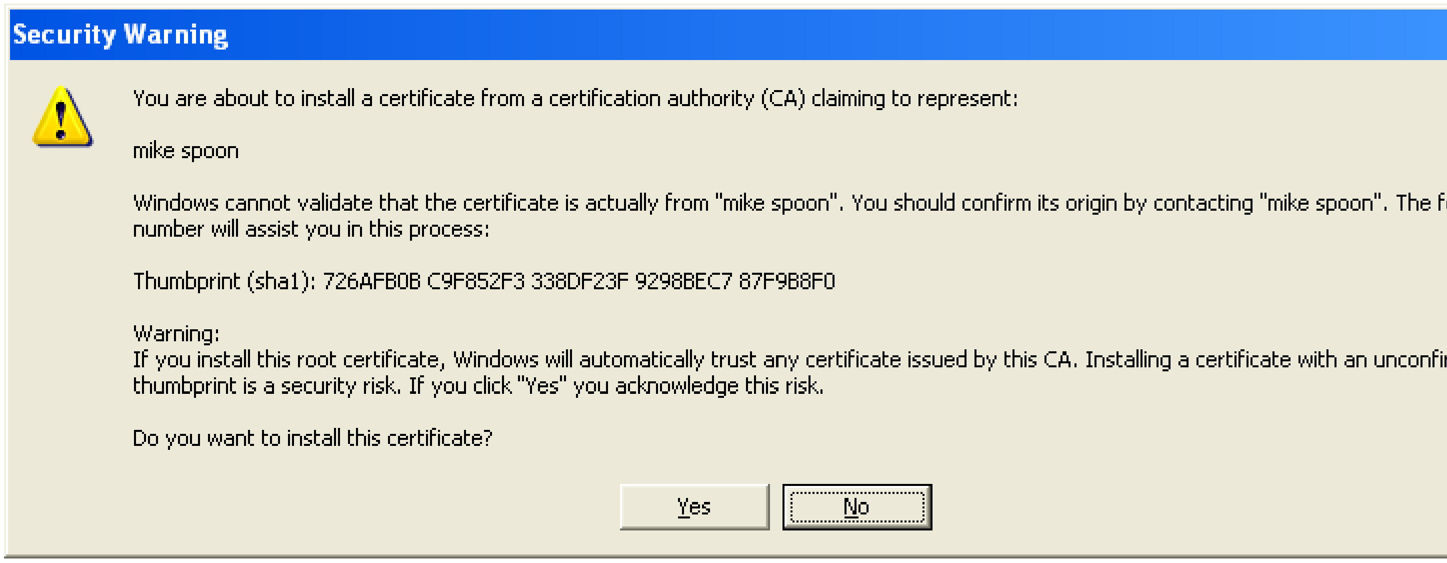
Completing the Certificate Import Wizard

You have successfully completed the Certificate Import wizard.

You have specified the following settings:

Certificate Store Selected by User	Trusted Root Certific
Content	Certificate
File Name	C:\Documents and Se

For the following message, click the Yes button.



You are about to install a certificate from a certification authority (CA) claiming to represent:

mike spoon

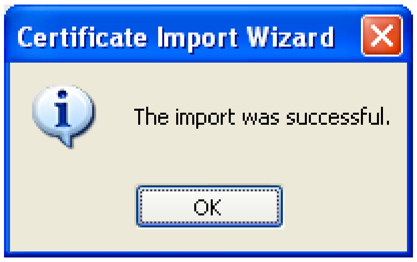
Windows cannot validate that the certificate is actually from "mike spoon". You should confirm its origin by contacting "mike spoon". The f
number will assist you in this process:

Thumbprint (sha1): 726AFB0B C9F852F3 338DF23F 9298BEC7 87F9B8F0

Warning:
If you install this root certificate, Windows will automatically trust any certificate issued by this CA. Installing a certificate with an unconfi
thumbprint is a security risk. If you click "Yes" you acknowledge this risk.

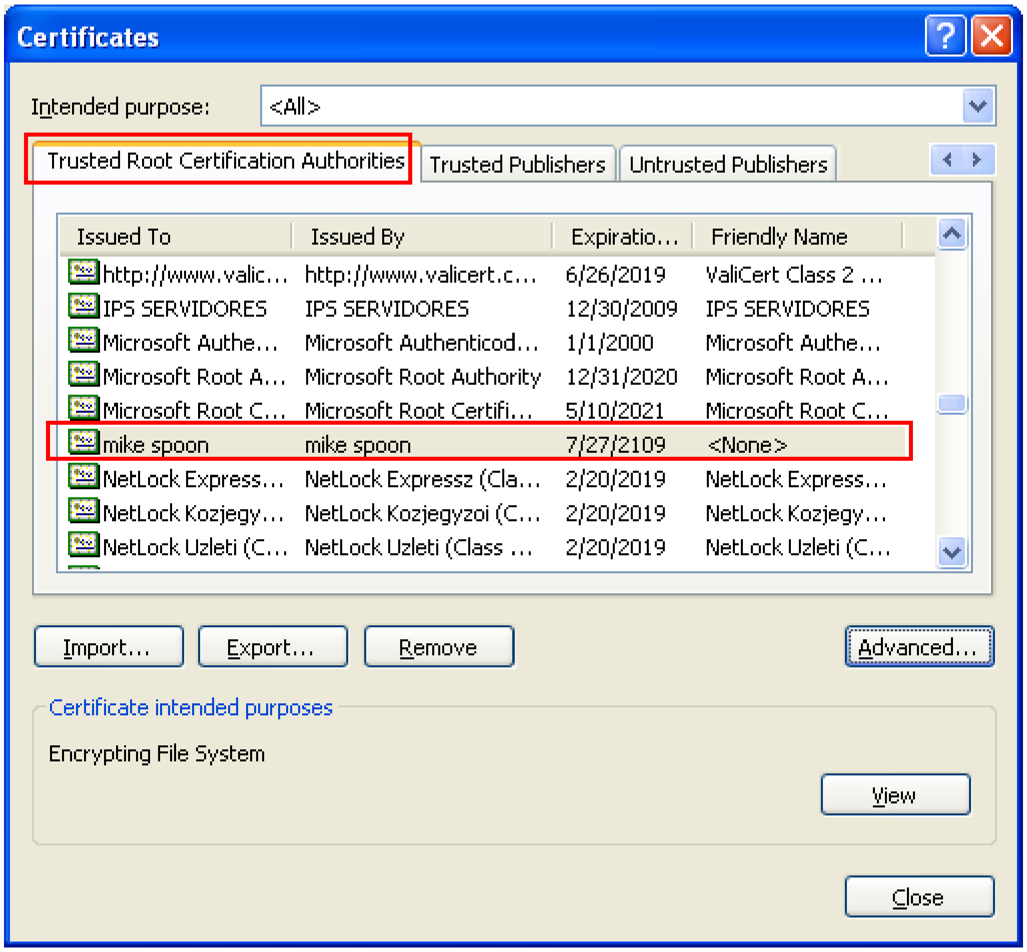
Do you want to install this certificate?

The following message box indicates the task completed successfully. Dismiss it by clicking the OK


Certificate Import Wizard

The import was successful.

Then, we can see the certificate in the Trusted Root Certification Authorities store.


Certificates

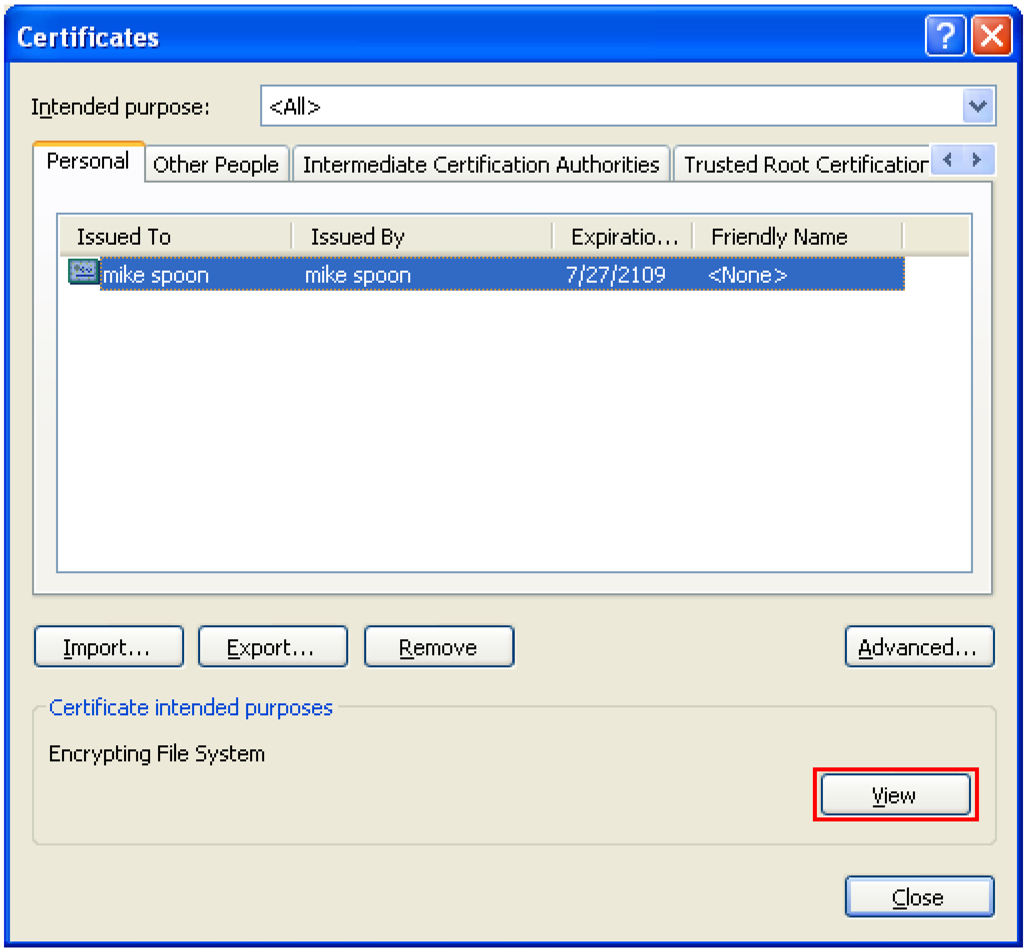
Intended purpose: <All>

Issued To	Issued By	Expiratio...	Friendly Name
http://www.valic...	http://www.valicert.c...	6/26/2019	ValiCert Class 2 ...
IPS SERVIDORES	IPS SERVIDORES	12/30/2009	IPS SERVIDORES
Microsoft Authe...	Microsoft Authenticod...	1/1/2000	Microsoft Authe...
Microsoft Root A...	Microsoft Root Authority	12/31/2020	Microsoft Root A...
Microsoft Root C...	Microsoft Root Certifi...	5/10/2021	Microsoft Root C...
mike spoon	mike spoon	7/27/2109	<None>
NetLock Express...	NetLock Expressz (Cla...	2/20/2019	NetLock Express...
NetLock Kozjegy...	NetLock Kozjegyzoi (C...	2/20/2019	NetLock Kozjegy...
NetLock Uzleti (C...	NetLock Uzleti (Class ...	2/20/2019	NetLock Uzleti (C...

Certificate intended purposes

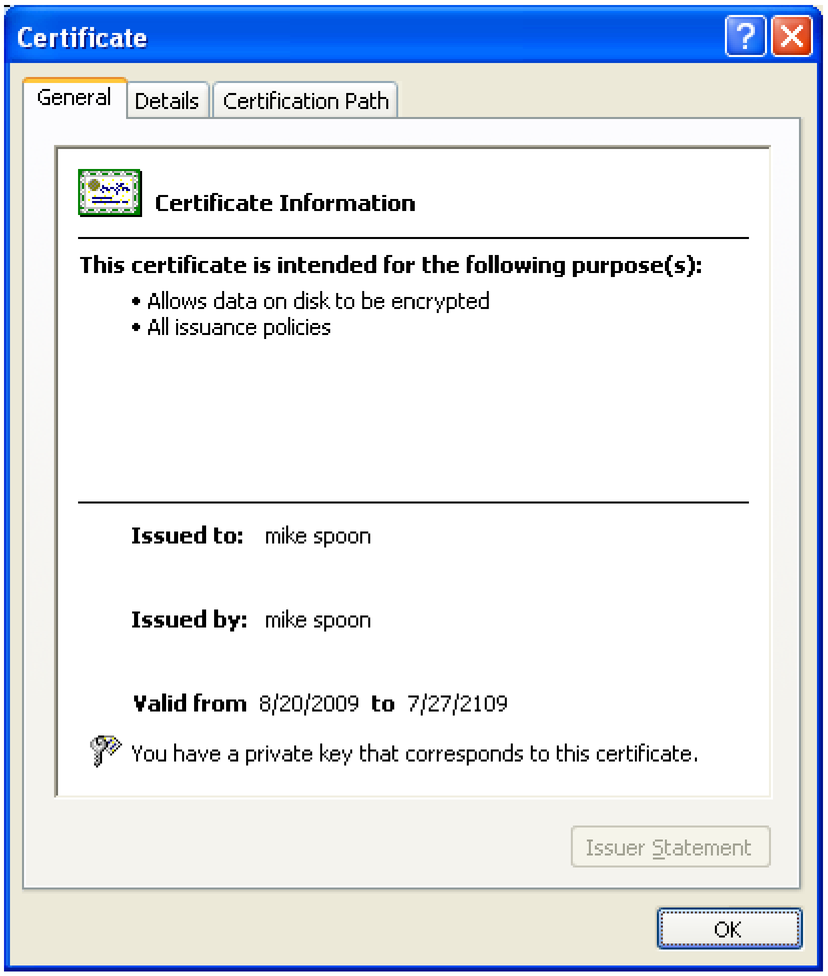
Encrypting File System

And the previous error should be disappeared.


Certificates
Intended purpose: <All>
Personal
Other People
Intermediate Certification Authorities
Trusted Root Certification

Issued To	Issued By	Expiratio...	Friendly Name
mike spoon	mike spoon	7/27/2109	<None>

Encrypting File System



Certificate Information

This certificate is intended for the following purpose(s):

- Allows data on disk to be encrypted
- All issuance policies

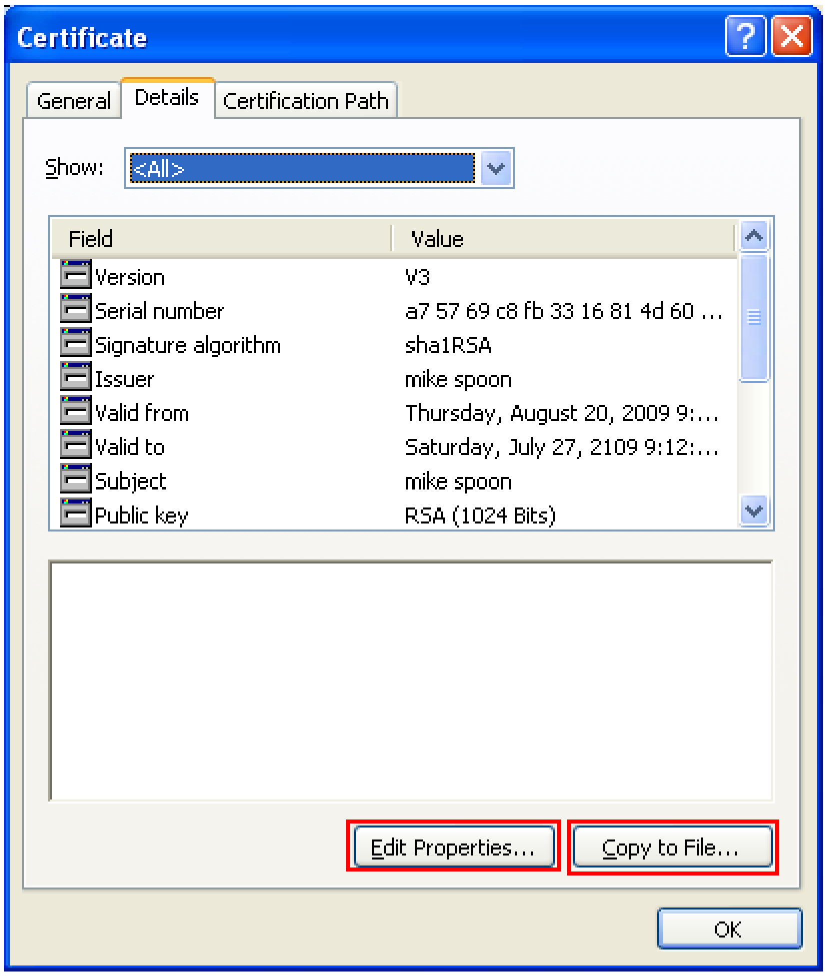
Issued to: mike spoon

Issued by: mike spoon

Valid from 8/20/2009 to 7/27/2109

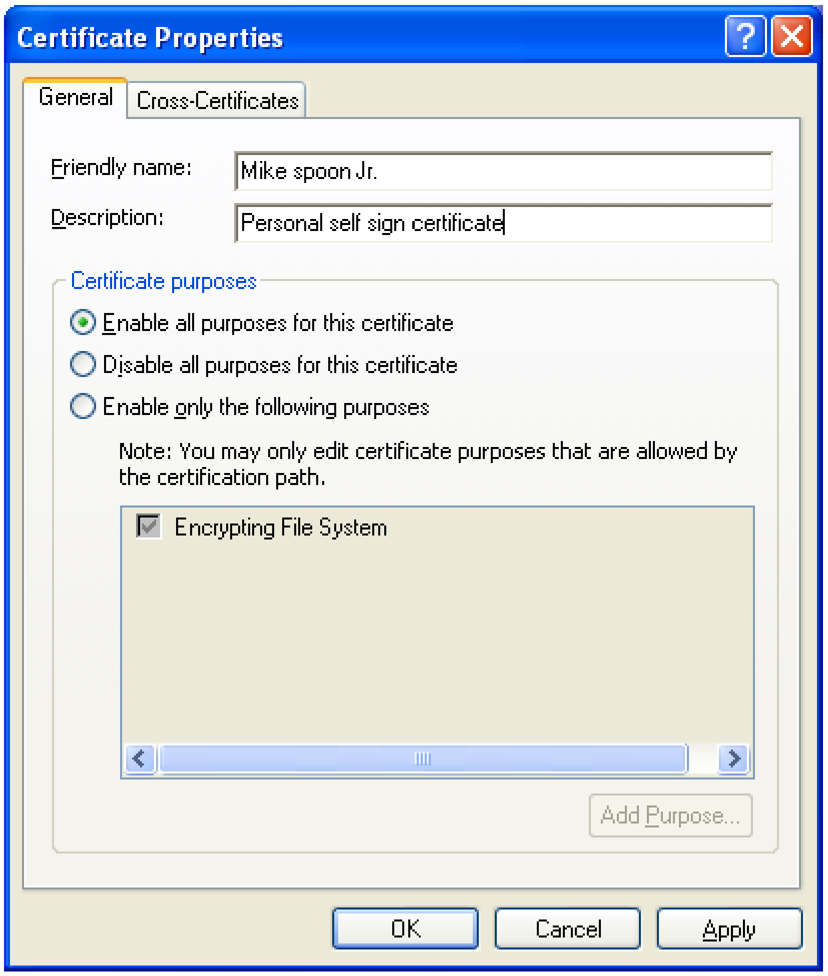
You have a private key that corresponds to this certificate.

In the meantime, we can edit some of the certificate properties. Click the Details tab.
Click the Edit Properties button. The Copy To File button is another short-cut which can be used to export the certificate.



Field	Value
Version	V3
Serial number	a7 57 69 c8 fb 33 16 81 4d 60 ...
Signature algorithm	sha1RSA
Issuer	mike spoon
Valid from	Thursday, August 20, 2009 9:...
Valid to	Saturday, July 27, 2109 9:12:...
Subject	mike spoon
Public key	RSA (1024 Bits)

For the Certificate Properties page, edit the Friendly name and Description. Click the OK button.



Friendly name: Mike spoon Jr.

Description: Personal self sign certificate

Certificate purposes

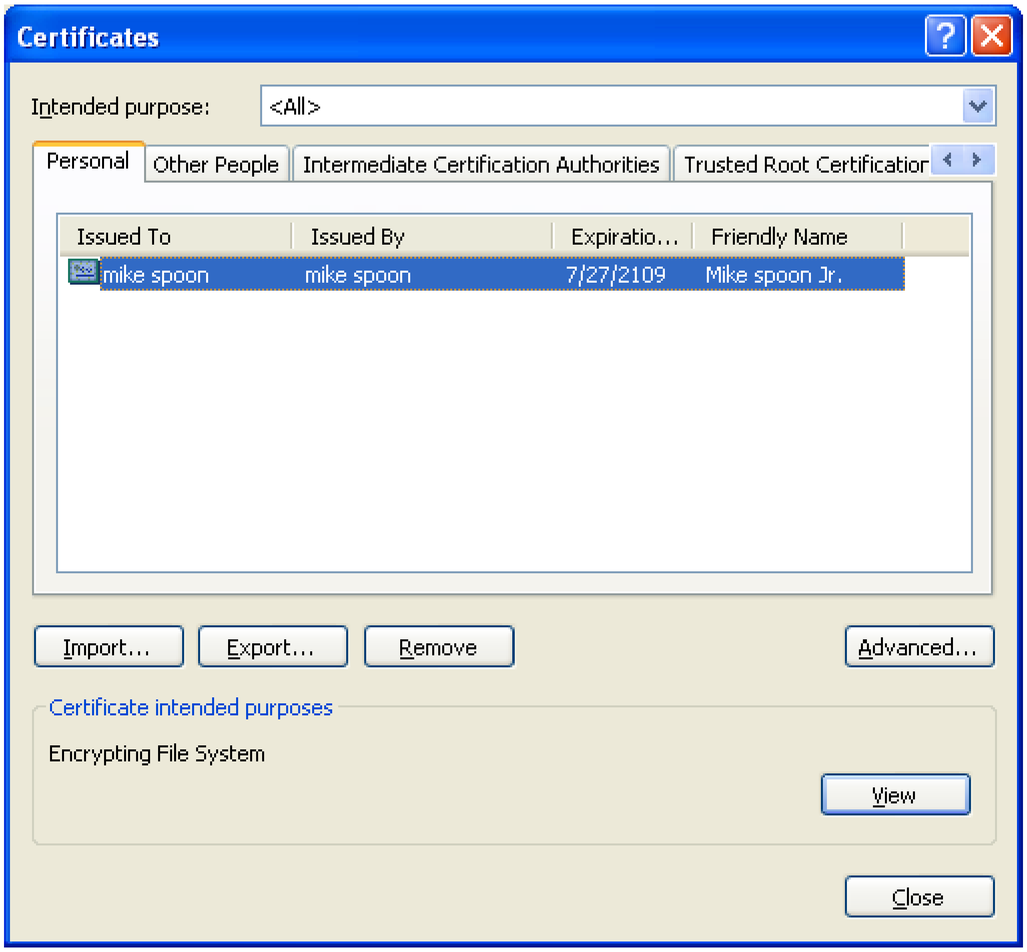
(●) Enable all purposes for this certificate

( ) Disable all purposes for this certificate

( ) Enable only the following purposes

Note: You may only edit certificate purposes that are allowed by the certification path.

[✓] Encrypting File System



Issued To	Issued By	Expiratio...	Friendly Name
mike spoon	mike spoon	7/27/2109	Mike spoon Jr.

Encrypting File System

File Security and Access Rights

Because files are securable objects, access to them is regulated by the access-control model that governs access to all other securable objects in Windows.
You can specify a security descriptor for a file or directory when you call the CreateFile(), CreateDirectory(), or CreateDirectoryEx() function. If you specify NULL for the lpSecurityAttributes parameter, the file or directory gets a default security descriptor. The access control lists (ACL) in the default security descriptor for a file or directory are inherited from its parent directory. Note that a default security descriptor is assigned only when a file or directory is newly created, and not when it is renamed or moved. To retrieve the security descriptor of a file or directory object, call the GetNamedSecurityInfo() or GetSecurityInfo() function. To change the security descriptor of a file or directory object, call the SetNamedSecurityInfo() or SetSecurityInfo() function.
The valid access rights for files and directories include the DELETE, READ_CONTROL, WRITE_DAC, and WRITE_OWNER standard access rights. The following table lists the access rights that are specific to files and directories.

Access right	Value	Description
FILE_ADD_FILE	0x0002	For a directory, the right to create a file in the directory.
FILE_ADD_SUBDIRECTORY	0x0004	For a directory, the right to create a subdirectory.

FILE_ALL_ACCESS		All possible access rights for a file.
FILE_APPEND_DATA	0x0004	For a file object, the right to append data to the file. (For local files, write operations will not overwrite existing data if this flag is specified without FILE_WRITE_DATA.) For a directory object, the right to create a subdirectory (FILE_ADD_SUBDIRECTORY).
FILE_CREATE_PIPE_INSTANCE	0x0004	For a named pipe, the right to create a pipe.
FILE_DELETE_CHILD	0x0040	For a directory, the right to delete a directory and all the files it contains, including read-only files.
FILE_EXECUTE	0x0020	For a native code file, the right to execute the file. This access right given to scripts may cause the script to be executable, depending on the script interpreter.
FILE_LIST_DIRECTORY	0x0001	For a directory, the right to list the contents of the directory.
FILE_READ_ATTRIBUTES	0x0080	The right to read file attributes.
FILE_READ_DATA	0x0001	For a file object, the right to read the corresponding file data. For a directory object, the right to read the corresponding directory data.
FILE_READ_EA	0x0008	The right to read extended file attributes.
FILE_TRAVERSE	0x0020	For a directory, the right to traverse the directory. By default, users are assigned the BYPASS_TRAVERSE_CHECKING privilege, which ignores the FILE_TRAVERSE access right.
FILE_WRITE_ATTRIBUTES	0x0100	The right to write file attributes.
FILE_WRITE_DATA	0x0002	For a file object, the right to write data to the file. For a directory object, the right to create a file in the directory (FILE_ADD_FILE).
FILE_WRITE_EA	0x0010	The right to write extended file attributes.
STANDARD_RIGHTS_READ		Includes READ_CONTROL, which is the right to read the information in the file or directory object's security descriptor. This does not include the information in the SACL.
STANDARD_RIGHTS_WRITE		Same as STANDARD_RIGHTS_READ.
SYNCHRONIZE		The right to specify a file handle in one of the wait functions. However, for asynchronous file I/O operations, you should wait on the event handle in an OVERLAPPED structure rather than using the file handle for synchronization.

The following are the generic access rights for files and directories.

Access right	Description
GENERIC_EXECUTE	FILE_READ_ATTRIBUTES STANDARD_RIGHTS_EXECUTE SYNCHRONIZE

GENERIC_READ	FILE_READ_ATTRIBUTES FILE_READ_DATA FILE_READ_EA STANDARD_RIGHTS_READ SYNCHRONIZE
GENERIC_WRITE	FILE_APPEND_DATA FILE_WRITE_ATTRIBUTES FILE_WRITE_DATA FILE_WRITE_EA STANDARD_RIGHTS_WRITE SYNCHRONIZE

Windows compares the requested access rights and the information in the thread's access token with the information in the file or directory object's security descriptor. If the comparison does not prohibit all of the requested access rights from being granted, a handle to the object is returned to the thread and the access rights are granted.

By default, authorization for access to a file or directory is controlled strictly by the ACLs in the security descriptor associated with that file or directory. In particular, the security descriptor of a parent directory is not used to control access to any child file or directory. The FILE_TRAVERSE access right can be enforced by removing the BYPASS_TRAVERSE_CHECKING privilege from users. This is not recommended in the general case, as many programs do not correctly handle directory traversal errors. The primary use for the FILE_TRAVERSE access right on directories is to enable conformance to certain IEEE and ISO POSIX standards when interoperability with Unix systems is a requirement.

The Windows security model provides a way for a child directory to inherit, or to be prevented from inheriting, one or more of the ACEs in the parent directory's security descriptor. Each ACE contains information that determines how it can be inherited, and whether it will have an effect on the inheriting directory object. For example, some inherited ACEs control access to the inherited directory object, and these are called effective ACEs. All other ACEs are called inherit-only ACEs.

The Windows security model also enforces the automatic inheritance of ACEs to child objects according to the ACE inheritance rules. This automatic inheritance, along with the inheritance information in each ACE, determines how security restrictions are passed down the directory hierarchy.

Note that you cannot use an access-denied ACE to deny only GENERIC_READ or only GENERIC_WRITE access to a file. This is because for file objects, the generic mappings for both GENERIC_READ or GENERIC_WRITE include the SYNCHRONIZE access right. If an ACE denies GENERIC_WRITE access to a trustee, and the trustee requests GENERIC_READ access, the request will fail because the request implicitly includes SYNCHRONIZE access which is implicitly denied by the ACE, and vice versa. Instead of using access-denied ACEs, use access-allowed ACEs to explicitly allow the permitted access rights.

Another means of managing access to storage objects is encryption. The implementation of file system encryption in Windows is the Encrypted File System, or EFS. EFS encrypts only files and not directories. The advantage of encryption is that it provides additional protection to files that is applied on the media and not through the file system and the standard Windows access control architecture.

In most cases, the ability to read and write the security settings of a file or directory object is restricted to kernel-mode processes. Clearly, you would not want any user process to be able to change the ownership or access restriction on your private file or directory. However, a backup

application would not be able to complete its job of backing up your file if the access restrictions you have placed on your file or directory does not allow the application's user-mode process to read it. Backup applications must be able to override the security settings of file and directory objects to ensure a complete backup. Similarly, if a backup application attempts to write a backup copy of your file over the disk-resident copy, and you explicitly deny write privileges to the backup application process, the restore operation cannot complete. In this case also, the backup application must be able to override the access control settings of your file.
The SE_BACKUP_NAME and SE_RESTORE_NAME access privileges were specifically created to provide this ability to backup applications. If these privileges have been granted and enabled in the access token of the backup application process, it can then call CreateFile() to open your file or directory for backup, specifying the standard READ_CONTROL access right as the value of the dwDesiredAccess parameter. However, to identify the calling process as a backup process, the call to CreateFile() must include the FILE_FLAG_BACKUP_SEMANTICS flag in the dwFlagsAndAttributes parameter. The full syntax of the function call is the following:

```
HANDLE hFile = CreateFile(
                 fileName,                   // lpFileName
                 READ_CONTROL,               // dwDesiredAccess
                 0,                          // dwShareMode
                 NULL,                       // lpSecurityAttributes
                 OPEN_EXISTING,              // dwCreationDisposition
                 FILE_FLAG_BACKUP_SEMANTICS, // dwFlagsAndAttributes
                 NULL                        // hTemplateFile
               );
```

This will allow the backup application process to open your file and override the standard security checking. To restore your file, the backup application would use the following CreateFile() call syntax when opening your file to be written.

```
HANDLE hFile = CreateFile( fileName,         // lpFileName
                 WRITE_OWNER|WRITE_DAC,      // dwDesiredAccess
                 0,                          // dwShareMode
                 NULL,                       // lpSecurityAttributes
                 CREATE_ALWAYS,              // dwCreationDisposition
                 FILE_FLAG_BACKUP_SEMANTICS, // dwFlagsAndAttributes
                 NULL                        // hTemplateFile
              );
```

There are situations when a backup application must be able to change the access control settings of a file or directory. An example is when the access control settings of the disk-resident copy of a file or directory is different from the backup copy. This would happen if these settings were changed after the file or directory was backed up, or if it was corrupted.
The FILE_FLAG_BACKUP_SEMANTICS flag specified in the call to CreateFile() gives the backup application process permission to read the access-control settings of the file or directory. With this permission, the backup application process can then call GetKernelObjectSecurity() and SetKernelObjectSecurity() to read and than reset the access-control settings.
If a backup application must have access to the system-level access control settings, the ACCESS_SYSTEM_SECURITY flag must be specified in the dwDesiredAccess parameter value passed to CreateFile(). Backup applications call BackupRead() to read the files and directories specified for the restore operation, and BackupWrite() to write them.

Input and Output (I/O)

Windows provides the ability to perform input and output (I/O) operations on storage components located on local and remote computers.

I/O Concepts

File Buffering

This topic covers the various considerations for application control of file buffering, also known as unbuffered file input/output (I/O). File buffering is usually handled by the system behind the scenes and is considered part of file caching within the Windows operating system unless otherwise specified. Although the terms caching and buffering are sometimes used interchangeably, this topic uses the term buffering specifically in the context of explaining how to interact with data that is not being cached (buffered) by the system, where it is otherwise largely out of the direct control of user-mode applications.

When opening or creating a file with the CreateFile() function, the FILE_FLAG_NO_BUFFERING flag can be specified to disable system caching of data being read from or written to the file. Although this gives complete and direct control over data I/O buffering, in the case of files and similar devices there are data alignment requirements that must be considered.

Note that, this alignment information applies to I/O on devices such as files that support seeking and the concept of file position pointers (or offsets). For nonseeking devices, such as named pipes or communications devices, turning off buffering may not require any particular alignment. Any limitations or efficiencies that may be gained by alignment in that case are dependent on the underlying technology.

In a simple example, the application would open a file for write access with the FILE_FLAG_NO_BUFFERING flag and then perform a call to the WriteFile() function using a data buffer defined within the application. This local buffer is, in these circumstances, effectively the only file buffer that exists for this operation. Because of physical disk layout, file system storage layout, and system-level file pointer position tracking, this write operation will fail unless the locally-defined data buffers meet certain alignment criteria, discussed in the following section.

Discussion of caching does not consider any hardware caching on the physical disk itself, which is not guaranteed to be within the direct control of the system in any case. This has no effect on the requirements specified in this topic.

Alignment and File Access Requirements

As previously discussed, an application must meet certain requirements when working with files opened with FILE_FLAG_NO_BUFFERING. The following specifics apply:

1. File access sizes, including the optional file offset in the OVERLAPPED structure, if specified, must be for a number of bytes that is an integer multiple of the volume sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1,024, 1,536, or 2,048 bytes, but not of 335, 981, or 7,171 bytes.
2. File access buffer addresses for read and write operations should be sector-aligned, which means aligned on addresses in memory that are integer multiples of the volume sector size. Depending on the disk, this requirement may not be enforced.

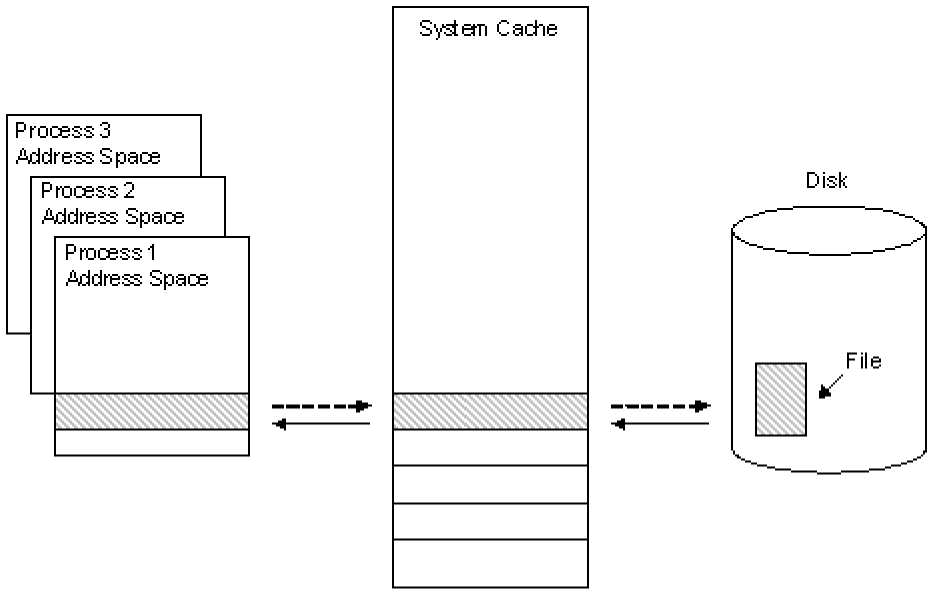
An application can determine a volume sector size by calling the GetDiskFreeSpace() function. Because buffer addresses for read and write operations must be sector-aligned, the application must have direct control of how these buffers are allocated. One way to sector-align buffers is to use the VirtualAlloc() function to allocate the buffers. Consider the following:

1. VirtualAlloc allocates memory that is aligned on addresses that are integer multiples of the system's page size. Page size is 4,096 bytes on x64 and x86 or 8,192 bytes for the Intel Itanium processor family.
2. Sector size is typically 512 to 4,096 bytes for direct-access storage devices (hard drives) and 2,048 bytes for CD-ROMs.
3. Both page and sector sizes are powers of 2.

Therefore, in most situations, page-aligned memory will also be sector-aligned, because the case where the sector size is larger than the page size is rare. Another way to obtain manually-aligned memory buffers is to use the _aligned_malloc() function from the C Run-Time library.

File Caching

By default, Windows caches file data that is **read from disks and written to disks**. This implies that read operations read file data from an area in system memory known as the **system file cache**, rather than from the physical disk. Correspondingly, write operations write file data to the system file cache rather than to the disk, and this type of cache is referred to as a **write-back cache**. Caching is managed per file object.
Caching occurs under the direction of the cache manager, which operates continuously while Windows is running. File data in the system file cache is written to the disk at intervals determined by the operating system, and the memory previously used by that file data is freed, this is referred to as flushing the cache. The policy of delaying the writing of the data to the file and holding it in the cache until the cache is flushed is called lazy writing, and it is triggered by the cache manager at a determinate time interval. The time at which a block of file data is flushed is partially based on the amount of time it has been stored in the cache and the amount of time since the data was last accessed in a read operation. This ensures that file data that is frequently read will stay accessible in the system file cache for the maximum amount of time. This file data caching process is illustrated in the following figure.



As depicted by the solid arrows in the previous figure, a 256KB region of data is read into a 256KB cache "slot" in system address space when it is first requested by the cache manager during a file read operation. A user-mode process then copies the data in this slot to its own address space. When the process has completed its data access, it writes the altered data back to the same slot in the system cache, as shown by the dotted arrow between the process address space and the system cache. When the cache manager has determined that the data will no longer be needed for a certain amount of time, it writes the altered data back to the file on the disk, as shown by the dotted arrow between the system cache and the disk.

The amount of I/O performance improvement that file data caching offers depends on the size of the file data block being read or written. When large blocks of file data are read and written, it is more likely that disk reads and writes will be necessary to finish the I/O operation. I/O performance will be increasingly impaired as more of this kind of I/O operation occurs.

In these situations, caching can be turned off. This is done at the time the file is opened by passing FILE_FLAG_NO_BUFFERING as a value for the dwFlagsAndAttributes parameter of CreateFile(). When caching is disabled, all read and write operations directly access the physical disk. However, the file metadata may still be cached. To flush the metadata to disk, use the FlushFileBuffers() function.

The frequency at which flushing occurs is an important consideration that balances system performance with system reliability. If the system flushes the cache too often, the number of large write operations flushing incurs will degrade system performance significantly. If the system is not flushed often enough, then the likelihood is greater that either system memory will be depleted by the cache, or a sudden system failure (such as a loss of power to the computer) will happen before the flush. In the latter instance, the cached data will be lost.

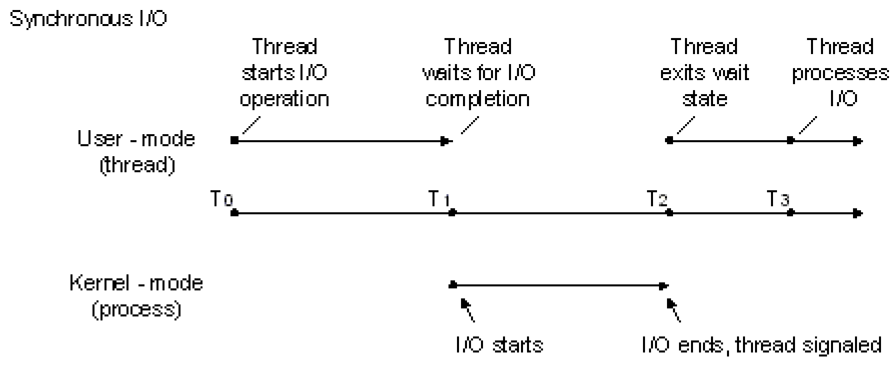
To ensure that the right amount of flushing occurs, the cache manager spawns a process every second called a lazy writer. The lazy writer process queues one-eighth of the pages that have not been flushed recently to be written to disk. It constantly reevaluates the amount of data being flushed for optimal system performance, and if more data needs to be written it queues more data. **Lazy writers** do not flush temporary files, because the assumption is that they will be deleted by the application or system.

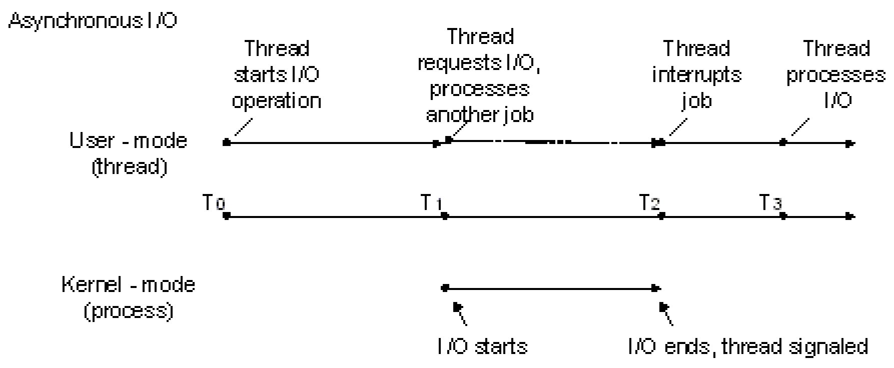
Some applications, such as virus-checking software, require that their write operations be flushed to disk immediately; Windows provides this ability through write-through caching. A process enables

write-through caching for a specific I/O operation by passing the FILE_FLAG_WRITE_THROUGH flag into its call to CreateFile(). With write-through caching enabled, data is still written into the cache, but the cache manager writes the data immediately to disk rather than incurring a delay by using the lazy writer. A process can also force a flush of a file it has opened by calling the FlushFileBuffers() function.
File system metadata is always cached. Therefore, to store any metadata changes to disk, the file must either be flushed or be opened with FILE_FLAG_WRITE_THROUGH.

Synchronous and Asynchronous I/O

There are two types of input/output (I/O) synchronization: synchronous I/O and asynchronous I/O. Asynchronous I/O is also referred to as overlapped I/O. In synchronous file I/O, a thread starts an I/O operation and immediately enters a wait state until the I/O request has completed. A thread performing asynchronous file I/O sends an I/O request to the kernel by calling an appropriate function. If the request is accepted by the kernel, the calling thread continues processing another job until the kernel signals to the thread that the I/O operation is complete. It then interrupts its current job and processes the data from the I/O operation as necessary. The two synchronization types are illustrated in the following figure.





In situations where an I/O request is expected to take a large amount of time, such as a refresh or backup of a large database or a slow communications link, asynchronous I/O is generally a good way to optimize processing efficiency. However, for relatively fast I/O operations, the overhead of processing kernel I/O requests and kernel signals may make asynchronous I/O less beneficial, particularly if many fast I/O operations need to be made. In this case, synchronous I/O would be better. The mechanisms and implementation details of how to accomplish these tasks vary depending on the type of device handle that is used and the particular needs of the application. In other words, there are usually multiple ways to solve the problem.

Synchronous and Asynchronous I/O Considerations

If a file or device is opened for synchronous I/O (that is, FILE_FLAG_OVERLAPPED is not specified), subsequent calls to functions such as WriteFile() can block execution of the calling thread until one of the following events occurs:

1. The I/O operation completes (in this example, a data write).
2. An I/O error occurs. (For example, the pipe is closed from the other end.)
3. An error was made in the call itself (for example, one or more parameters are not valid).
4. Another thread in the process calls the CancelSynchronousIo() function using the blocked thread's thread handle, which terminates I/O for that thread, failing the I/O operation.
5. The blocked thread is terminated by the system; for example, the process itself is terminated, or another thread calls the TerminateThread() function using the blocked thread's handle. (This is generally considered a last resort and not good application design.)

In some cases, this delay may be unacceptable to the application's design and purpose, so application designers should consider using asynchronous I/O with appropriate thread synchronization objects such as I/O completion ports.
A process opens a file for asynchronous I/O in its call to CreateFile() by specifying the FILE_FLAG_OVERLAPPED flag in the dwFlagsAndAttributes parameter. If FILE_FLAG_OVERLAPPED is not specified, the file is opened for synchronous I/O. When the file has been opened for asynchronous I/O, a pointer to an OVERLAPPED structure is passed into the call to ReadFile() and WriteFile(). When performing synchronous I/O, this structure is not required in calls to ReadFile() and WriteFile().
If a file or device is opened for asynchronous I/O, subsequent calls to functions such as WriteFile() using that handle generally return immediately but can also behave synchronously with respect to blocked execution. For more information, see http://support.microsoft.com/kb/156932.
Although CreateFile() is the most common function to use for opening files, disk volumes, anonymous pipes, and other similar devices, I/O operations can also be performed using a handle typecast from other system objects such as a socket created by the socket() or accept() functions.
Handles to directory objects are obtained by calling the CreateFile() function with the FILE_FLAG_BACKUP_SEMANTICS attribute. Directory handles are almost never used, backup applications are one of the few applications that will typically use them.
After opening the file object for asynchronous I/O, an OVERLAPPED structure must be properly created, initialized, and passed into each call to functions such as ReadFile() and WriteFile(). Keep the following in mind when using the OVERLAPPED structure in asynchronous read and write operations:

1. Do not deallocate or modify the OVERLAPPED structure or the data buffer until all asynchronous I/O operations to the file object have been completed.
2. If you declare your pointer to the OVERLAPPED structure as a local variable, do not exit the local function until all asynchronous I/O operations to the file object have been completed. If the local function is exited prematurely, the OVERLAPPED structure will go out of scope and it

will be inaccessible to any ReadFile() or WriteFile() functions it encounters outside of that function.

You can also create an event and put the handle in the OVERLAPPED structure; the wait functions can then be used to wait for the I/O operation to complete by waiting on the event handle.
As previously stated, when working with an asynchronous handle, applications should use care when making determinations about when to free resources associated with a specified I/O operation on that handle. If the handle is deallocated prematurely, ReadFile() or WriteFile() may incorrectly report that the I/O operation is complete. Further, the WriteFile() function will sometimes return TRUE with a GetLastError() value of ERROR_SUCCESS, even though it is using an asynchronous handle (which can also return FALSE with ERROR_IO_PENDING). Programmers accustomed to synchronous I/O design will usually release data buffer resources at this point because TRUE and ERROR_SUCCESS signify the operation is complete. However, if I/O completion ports are being used with this asynchronous handle, a completion packet will also be sent even though the I/O operation completed immediately. In other words, if the application frees resources after WriteFile() returns TRUE with ERROR_SUCCESS in addition to in the I/O completion port routine, it will have a double-free error condition. In this example, the recommendation would be to allow the completion port routine to be solely responsible for all freeing operations for such resources.
The system does not maintain the file pointer on asynchronous handles to files and devices that support file pointers (that is, seeking devices), therefore the file position must be passed to the read and write functions in the related offset data members of the OVERLAPPED structure.
File pointer position for a synchronous handle is maintained by the system as data is read or written and can also be updated using the SetFilePointer() or SetFilePointerEx() function.
An application can also wait on the file handle to synchronize the completion of an I/O operation, but doing so requires extreme caution. Each time an I/O operation is started, the operating system sets the file handle to the nonsignaled state. Each time an I/O operation is completed, the operating system sets the file handle to the signaled state. Therefore, if an application starts two I/O operations and waits on the file handle, there is no way to determine which operation is finished when the handle is set to the signaled state. If an application must perform multiple asynchronous I/O operations on a single file, it should wait on the event handle in the specific OVERLAPPED structure for each I/O operation, rather than on the common file handle.
To cancel all pending asynchronous I/O operations, use either:

1. CancelIo() - this function only cancels operations issued by the calling thread for the specified file handle.
2. CancelIoEx() - this function cancels all operations issued by the threads for the specified file handle.

Use CancelSynchronousIo() to cancel pending synchronous I/O operations. The ReadFileEx() and WriteFileEx() functions enable an application to specify a routine to execute when the asynchronous I/O request is completed.

Canceling Pending I/O Operations

Allowing users to cancel I/O requests that are slow or blocked can enhance the usability and robustness of your application. For example, if a call to the OpenFile() function is blocked because the call is to a very slow device, canceling it enables the call to be made again, with new parameters, without terminating the application.

Windows Vista extends the cancellation capabilities and includes support for canceling synchronous operations. Take note that by calling the CancelIoEx() function does not guarantee that an I/O operation will be canceled; the driver which is handling the operation must support cancellation and the operation must be in a state that can be canceled.

Cancellation Considerations

When programming cancellation calls, keep in mind the following considerations:

1. There is no guarantee that underlying drivers correctly support cancellation.
2. When canceling asynchronous I/O, when no overlapped structure is supplied to the CancelIoEx() function, the function attempts to cancel all outstanding I/O on the file on all threads in the process. Each thread is processed individually, so after a thread has been processed it may start another I/O on the file before all the other threads have had their I/O for the file canceled, causing synchronization issues.
3. When canceling asynchronous I/O, do not reuse overlapped structures with targeted cancellation. Once the I/O operation completes (either successfully or with a canceled status) then the overlapped structure is no longer in use by the system and can be reused.
4. When canceling synchronous I/O, calling the CancelSynchronousIo() function attempts to cancel any current synchronous call on the thread. You must take care to ensure the synchronization of the calls is correct; the wrong call in a series of calls could get canceled. For example, if the CancelSynchronousIo() function is called for a synchronous operation, X, operation Y only starts after that operation X is completed (normally or with an error). If the thread that called operation X then starts another synchronous call to X, the cancel call could interrupt this new I/O request.
5. When canceling synchronous I/O, be aware that a race condition can exist whenever a thread is shared between different parts of an application, for example, with a thread-pool thread.

Operations That Cannot Be Canceled

Some functions cannot be canceled using the CancelIo(), CancelIoEx(), or CancelSynchronousIo() function. Some of these functions have been extended to allow cancellation (for example, the CopyFileEx() function) and you should use these instead. In addition to supporting cancellation, these functions also have built-in callbacks to support you when tracking the progress of the operation. The following functions do not support cancellation:

1. CopyFile() - use CopyFileEx()
2. MoveFile() - use MoveFileWithProgress()
3. MoveFileEx() - use MoveFileWithProgress()
4. ReplaceFile()

Canceling Asynchronous I/O

You can cancel asynchronous I/O from any thread in the process that issued the I/O operation. You must specify the handle which the I/O was performed on and, optionally, the overlapped structure that was used to perform the I/O. You can determine if the cancel occurred by examining the status returned in the overlapped structure or in the completion callback. A status of ERROR_OPERATION_ABORTED indicates that the operation was canceled.

The following code snippet example shows a routine that takes a timeout and attempts a read operation, canceling it with the CancelIoEx() function if the timeout expires.

```
#include <windows.h>

BOOL DoCancelableRead(HANDLE hFile,
                 LPVOID lpBuffer,
                 DWORD nNumberOfBytesToRead,
                 LPDWORD lpNumberOfBytesRead,
                 LPOVERLAPPED lpOverlapped,
                 DWORD dwTimeout,
                 LPBOOL pbCancelCalled)
//
// Parameters:
//      hFile - An open handle to a readable file or device.
//      lpBuffer - A pointer to the buffer to store the data being read.
//      nNumberOfBytesToRead - The number of bytes to read from the file or
//          device. Must be less than or equal to the actual size of
//          the buffer referenced by lpBuffer.
//      lpNumberOfBytesRead - A pointer to a DWORD to receive the number
//          of bytes read after all I/O is complete or cancelled.
//      lpOverlapped - A pointer to a preconfigured OVERLAPPED structure that
//          has a valid hEvent.
//          If the caller does not properly initialize this structure, this
//          routine will fail.
//
//      dwTimeout - The desired time-out, in milliseconds, for the I/O read.
//          After this time expires, the I/O is cancelled.
//      pbCancelCalled - A pointer to a Boolean to notify the caller if this
//          routine attempted to perform an I/O cancel.
// Return Value:
//      TRUE on success, FALSE on failure.
{
    BOOL result;
    DWORD waitResult;
    DWORD waitTimer;
    BOOL bIoComplete = FALSE;
    const DWORD PollInterval = 100; // milliseconds

    // Initialize "out" parameters
    *pbCancelCalled = FALSE;
    *lpNumberOfBytesRead = 0;

    // Perform the I/O, in this case a read operation.
    result = ReadFile(hFile,
                  lpBuffer,
                  nNumberOfBytesToRead,
                  lpNumberOfBytesRead,
                  lpOverlapped );

    if (result == FALSE)
    {
        if (GetLastError() != ERROR_IO_PENDING)
        {
            // The function call failed. ToDo: Error logging and recovery.
            return FALSE;
        }
    }
    else
    {
        // The I/O completed, done.
        return TRUE;
    }

    // The I/O is pending, so wait and see if the call times out.
    // If so, cancel the I/O using the CancelIoEx function.
    for (waitTimer = 0; waitTimer < dwTimeout; waitTimer += PollInterval)
    {
        result = GetOverlappedResult( hFile, lpOverlapped, lpNumberOfBytesRead,
FALSE );
        if (result == FALSE)
        {
            if (GetLastError() != ERROR_IO_PENDING)
            {
                // The function call failed. ToDo: Error logging and recovery.
                return FALSE;
            }
            Sleep(PollInterval);
        }
        else
        {
            bIoComplete = TRUE;
            break;
        }
    }

    if (bIoComplete == FALSE)
    {
        result = CancelIoEx( hFile, lpOverlapped );

        *pbCancelCalled = TRUE;

        if (result == TRUE || GetLastError() != ERROR_NOT_FOUND)
        {
            // Wait for the I/O subsystem to acknowledge our cancellation.
            // Depending on the timing of the calls, the I/O might complete with
a
            // cancellation status, or it might complete normally (if the
ReadFile was
            // in the process of completing at the time CancelIoEx was called,
or if
            // the device does not support cancellation).
            // This call specifies TRUE for the bWait parameter, which will
block
            // until the I/O either completes or is cancelled, thus resuming
execution,
            // provided the underlying device driver and associated hardware are
functioning
            // properly. If there is a problem with the driver it is better to
stop
            // responding, or "hang," here than to try to continue while masking
the problem.

            result = GetOverlappedResult( hFile, lpOverlapped,
lpNumberOfBytesRead, TRUE );

            // ToDo: check result and log errors.
        }
    }
    return result;
}
```

Canceling Synchronous I/O

You can cancel synchronous I/O from any thread in the process that issued the I/O operation. You must specify the handle to the thread which is currently performing the I/O operation. The following example shows two routines:

1. The SynchronousIoWorker routine is a worker thread that implements some synchronous file I/O, starting with a call to the CreateFile() function. If the routine is successful, the routine can be followed by additional operations, which are not included here. The global variable gCompletionStatus can be used to determine whether all operations succeeded or whether an operation failed or was canceled. The global variable dwOperationInProgress indicates whether file I/O is still in progress.
   Take note that in this example, the UI thread could also check for the existence of the worker thread. Additional manual checks, which are not included here, are required in the SynchronousIoWorker() routine to ensure that if the cancellation was requested during the brief periods between file I/O calls, the rest of the operations will be canceled.

2. The MainUIThreadMessageHandler() routine simulates the message handler within a UI thread's window procedure. The user requests a set of synchronous file operations by clicking a control, which generates a user-defined window message, (in the section marked by WM_MYSYNCOPS). This creates a new thread using the CreateFileThread() routine, which then starts the SynchronousIoWorker() routine. The UI thread continues to process messages while the worker thread performs the requested I/O. If the user decides to cancel the unfinished operations (typically by clicking a cancel button), the routine (in the section marked by WM_MYCANCEL) calls the CancelSynchronousIo() function using the thread handle returned by the CreateFileThread() routine. The CancelSynchronousIo() function returns immediately after the cancellation attempt. Finally, the user or application may later request some other operation that depends on whether the file operations have completed. In this case, the routine (in the section marked by WM_PROCESSDATA) first verifies that the operations have completed and then executes the clean-up operations. Note that in the following code snippet example, since the cancellation could have occurred anywhere in the sequence of operations, it may be necessary for the caller to ensure that the state is consistent, or at least understood, before proceeding.

```
// User-defined codes for the message-pump, which is outside the scope
// of this sample. Windows messaging and message pumps are well-documented
// elsewhere.
#define WM_MYSYNCOPS    1
#define WM_MYCANCEL     2
#define WM_PROCESSDATA  3

VOID SynchronousIoWorker( VOID *pv )
{
      LPCSTR lpFileName = (LPCSTR)pv;
      HANDLE hFile;
      g_dwOperationInProgress = TRUE;
      g_CompletionStatus = ERROR_SUCCESS;

      hFile = CreateFileA(lpFileName,
                              GENERIC_READ,
                              0,
                              NULL,
                              OPEN_EXISTING,
                              0,
                              NULL);


      if (hFile != INVALID_HANDLE_VALUE)
    {
        BOOL result = TRUE;
        // TODO: CreateFile succeeded.
        // Insert your code to make more synchronous calls with hFile.
        // The result variable is assumed to act as the error flag here,
        // but can be whatever your code needs.

        if (result == FALSE)
        {
                  // An operation failed or was canceled. If it was canceled,
                  // GetLastError() returns ERROR_OPERATION_ABORTED.
                  g_CompletionStatus = GetLastError();
        }
       CloseHandle(hFile);
    }
      else
    {
        // CreateFile failed or was canceled. If it was canceled,
        // GetLastError() returns ERROR_OPERATION_ABORTED.
        g_CompletionStatus = GetLastError();
    }

      g_dwOperationInProgress = FALSE;
}

//---------------------------------------------------------
LRESULT
CALLBACK
MainUIThreadMessageHandler(HWND hwnd,
                           UINT uMsg,
                           WPARAM wParam,
                           LPARAM lParam)
{
      HANDLE syncThread = INVALID_HANDLE_VALUE;

      //  Insert your initializations here.
      switch (uMsg)
    {
            // User requested an operation on a file.  Insert your code to
            // retrieve filename from parameters.
            case WM_MYSYNCOPS:
                  syncThread = CreateThread(
                          NULL,
                          0,
                          (LPTHREAD_START_ROUTINE)SynchronousIoWorker,
                           &g_lpFileName,
                           0,
                           NULL);

                    if (syncThread == INVALID_HANDLE_VALUE)
                {
                            // Insert your code to handle the failure.
                }
            break;

                // User clicked a cancel button.
                case WM_MYCANCEL:
                    if (syncThread != INVALID_HANDLE_VALUE)
                    {
                            CancelSynchronousIo(syncThread);
                    }
                break;

                // User requested other operations.
                case WM_PROCESSDATA:
                    if (!g_dwOperationInProgress)
                {
                            if (g_CompletionStatus == ERROR_OPERATION_ABORTED)
                    {
                                // Insert your cleanup code here.
                    }
                            else
                    {
                        // Insert code to handle other cases.
                    }
                }
            break;
        }
          return TRUE;
}
```

Alertable I/O

Alertable I/O is **the method by which application threads process asynchronous I/O requests only when they are in an alertable state**. To understand when a thread is in an alertable state, consider the following scenario:

1. A thread initiates an asynchronous read request by calling ReadFileEx() with a pointer to a callback function.
2. The thread initiates an asynchronous write request by calling WriteFileEx() with a pointer to a callback function.
3. The thread calls a function that fetches a row of data from a remote database server.

In this scenario, the calls to ReadFileEx() and WriteFileEx() will most likely return before the function call in step 3. When they do, the kernel places the pointers to the callback functions on the thread's Asynchronous Procedure Call (APC) queue. The kernel maintains this queue specifically to hold returned I/O request data until it can be processed by the corresponding thread.
When the row fetch is complete and the thread returns from the function, its highest priority is to process the returned I/O requests on the queue by calling the callback functions. To do this, it must

enter an alertable state. A thread can only do this by calling one of the following functions with the appropriate flags:

1. SleepEx()
2. WaitForSingleObjectEx()
3. WaitForMultipleObjectsEx()
4. SignalObjectAndWait()
5. MsgWaitForMultipleObjectsEx()

When the thread enters an alertable state, the following events occur:

1. The kernel checks the thread's APC queue. If the queue contains callback function pointers, the kernel removes the pointer from the queue and sends it to the thread.
2. The thread executes the callback function.
3. Steps 1 and 2 are repeated for each pointer remaining in the queue.
4. When the queue is empty, the thread returns from the function that placed it in an alertable state.

In this scenario, once the thread enters an alertable state it will call the callback functions sent to ReadFileEx() and WriteFileEx(), then return from the function that placed it in an alertable state. If a thread enters an alertable state while its APC queue is empty, the thread's execution will be suspended by the kernel until one of the following occurs:

1. The kernel object that is being waited on becomes signaled.
2. A callback function pointer is placed in the APC queue.

A thread that uses alertable I/O processes asynchronous I/O requests more efficiently than when they simply wait on the event flag in the OVERLAPPED structure to be set, and the alertable I/O mechanism is less complicated than I/O completion ports to use. However, alertable I/O returns the result of the I/O request only to the thread that initiated it. I/O completion ports do not have this limitation.

I/O Completion Ports

I/O completion ports provide an efficient threading model for processing multiple asynchronous I/O requests on a multiprocessor system. When a process creates an I/O completion port, the system creates an associated queue object for requests whose sole purpose is to service these requests. Processes that handle many concurrent asynchronous I/O requests can do so more quickly and efficiently by using I/O completion ports in conjunction with a pre-allocated thread pool than by creating threads at the time they receive an I/O request.

How I/O Completion Ports Work

The CreateIoCompletionPort() function creates an I/O completion port and associates one or more file handles with that port. When an asynchronous I/O operation on one of these file handles completes, an I/O completion packet is queued in first-in-first-out (FIFO) order to the associated I/O completion port. One powerful use for this mechanism is to combine the synchronization point for multiple file handles into a single object, although there are also other useful applications.

The term file handle as used here refers to a system abstraction representing an overlapped I/O endpoint, not only a file on disk. For example, it can be a network endpoint, TCP socket, named pipe, or mail slot. Any system object that supports overlapped I/O can be used.
When a file handle is associated with a completion port, the status block passed in will not be updated until the packet is removed from the completion port. The only exception is if the original operation returns synchronously with an error. A thread (either one created by the main thread or the main thread itself) uses the GetQueuedCompletionStatus() function to wait for a completion packet to be queued to the I/O completion port, rather than waiting directly for the asynchronous I/O to complete. Threads that block their execution on an I/O completion port are released in last-in-first-out (LIFO) order, and the next completion packet is pulled from the I/O completion port's FIFO queue for that thread. This means that, when a completion packet is released to a thread, the system releases the last (most recent) thread associated with that port, passing it the completion information for the oldest I/O completion.
Although any number of threads can call GetQueuedCompletionStatus() for a specified I/O completion port, when a specified thread calls GetQueuedCompletionStatus() the first time, it becomes associated with the specified I/O completion port until one of three things occurs: The thread exits, specifies a different I/O completion port, or closes the I/O completion port. In other words, a single thread can be associated with, at most, one I/O completion port.
When a completion packet is queued to an I/O completion port, the system first checks how many threads associated with that port are running. If the number of threads running is less than the concurrency value (discussed in the next section), one of the waiting threads (the most recent one) is allowed to process the completion packet. When a running thread completes its processing, it typically calls GetQueuedCompletionStatus() again, at which point it either returns with the next completion packet or waits if the queue is empty.
Threads can use the PostQueuedCompletionStatus() function to place completion packets in an I/O completion port's queue. By doing so, the completion port can be used to receive communications from other threads of the process, in addition to receiving I/O completion packets from the I/O system. The PostQueuedCompletionStatus() function allows an application to queue its own special-purpose completion packets to the I/O completion port without starting an asynchronous I/O operation. This is useful for notifying worker threads of external events, for example.
The I/O completion port handle and every file handle associated with that particular I/O completion port are known as references to the I/O completion port. The I/O completion port is released when there are no more references to it. Therefore, all of these handles must be properly closed to release the I/O completion port and its associated system resources. After these conditions are satisfied, an application should close the I/O completion port handle by calling the CloseHandle() function.
An I/O completion port is associated with the process that created it and is not shareable between processes. However, a single handle is shareable between threads in the same process.

Threads and Concurrency

The most important property of an I/O completion port to consider carefully is the concurrency value. The concurrency value of a completion port is specified when it is created with CreateIoCompletionPort() via the NumberOfConcurrentThreads parameter. This value limits the number of runnable threads associated with the completion port. When the total number of runnable threads associated with the completion port reaches the concurrency value, the system blocks the execution of any subsequent threads associated with that completion port until the number of runnable threads drops below the concurrency value.

The most efficient scenario occurs when there are completion packets waiting in the queue, but no waits can be satisfied because the port has reached its concurrency limit. Consider what happens with a concurrency value of one and multiple threads waiting in the GetQueuedCompletionStatus() function call. In this case, if the queue always has completion packets waiting, when the running thread calls GetQueuedCompletionStatus(), it will not block execution because, as mentioned earlier, the thread queue is LIFO. Instead, this thread will immediately pick up the next queued completion packet. No thread context switches will occur, because the running thread is continually picking up completion packets and the other threads are unable to run.

In the previous example, the extra threads appear to be useless and never run, but that assumes that the running thread never gets put in a wait state by some other mechanism, terminates, or otherwise closes its associated I/O completion port. Consider all such thread execution ramifications when designing the application.

The best overall maximum value to pick for the concurrency value is the number of CPUs on the computer. If your transaction required a lengthy computation, a larger concurrency value will allow more threads to run. Each completion packet may take longer to finish, but more completion packets will be processed at the same time. You can experiment with the concurrency value in conjunction with profiling tools to achieve the best effect for your application.

The system also allows a thread waiting in GetQueuedCompletionStatus() to process a completion packet if another running thread associated with the same I/O completion port enters a wait state for other reasons, for example the SuspendThread() function. When the thread in the wait state begins running again, there may be a brief period when the number of active threads exceeds the concurrency value. However, the system quickly reduces this number by not allowing any new active threads until the number of active threads falls below the concurrency value. This is one reason to have your application create more threads in its thread pool than the concurrency value. Thread pool management is beyond the scope of this topic, but a good rule of thumb is to have a minimum of twice as many threads in the thread pool as there are processors on the system.

Supported I/O Functions

The following functions can be used to start I/O operations that complete by using I/O completion ports. You must pass the function an instance of the OVERLAPPED structure and a file handle previously associated with an I/O completion port (by a call to CreateIoCompletionPort()) to enable the I/O completion port mechanism:

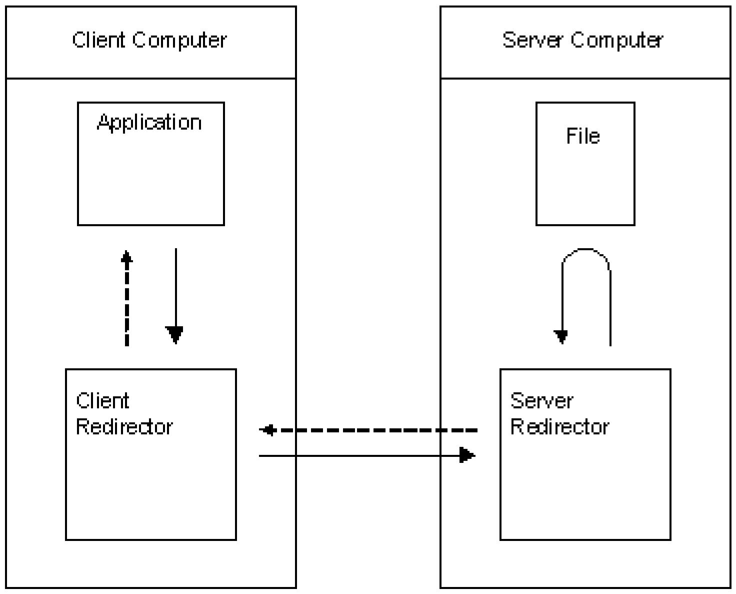
1. ConnectNamedPipe()
2. DeviceIoControl()
3. LockFileEx()
4. ReadDirectoryChangesW()
5. ReadFile()
6. TransactNamedPipe()
7. WaitCommEvent()
8. WriteFile()

Network I/O Concepts

When the underlying network protocol and redirector support I/O operations, you can use the file API to perform network I/O. The following topics provide information related to network I/O:

Description of a Network I/O Operation

The following figure illustrates the process of a network I/O operation under Windows.



When an application calls a file I/O function to access a file on a remote computer, the following events occur:

1. The I/O request is intercepted by a network redirector, also referred to simply as a redirector, on the local computer. This is depicted in the preceding figure by the solid arrow between the application and the client redirector.
2. The redirector constructs a data packet containing all of the information about the request, and sends it to the server where the file is located. This is depicted in the preceding figure by the solid arrow between the client redirector and the server redirector.
3. The redirector on the server receives the packet from the client, authenticates the access to the file required by the I/O request, and, if authenticated, executes the request on behalf of the client. If not, it returns an error code to the redirector on the client. This is depicted in the preceding figure by the curved solid arrow between the server redirector and the file.
4. When the request has been executed, the redirector on the server sends any data resulting from the I/O request to the redirector on the client along with a success notification. This is depicted in the preceding figure by the dotted arrow between the server and the client redirector.
5. The redirector on the client receives the packet from the server and passes the data in the packet to the application along with a success notification. This is depicted in the preceding figure by the dotted arrow between the client redirector and the application.

Windows can use a variety of networking protocols to perform a network I/O operation, including Microsoft SMB Protocol and CIFS Protocol Overview and NFS.

Microsoft SMB Protocol and CIFS Protocol Overview

The Server Message Block (SMB) Protocol is a network file sharing protocol, and as implemented in Microsoft Windows is known as Microsoft SMB Protocol. The set of message packets that defines a particular version of the protocol is called a dialect. The Common Internet File System (CIFS) Protocol is a dialect of SMB. Both SMB and CIFS are also available on VMS, several versions of Unix, and other operating systems.
The technical reference to CIFS is available from Microsoft Corporation at here. Information on licensing CIFS can be found at here. Information on licensing Microsoft SMB Protocol can be found at here.
Although its main purpose is file sharing, additional Microsoft SMB Protocol functionality includes the following:

1. Dialect negotiation
2. Determining other Microsoft SMB Protocol servers on the network, or network browsing
3. Printing over a network
4. File, directory, and share access authentication
5. File and record locking
6. File and directory change notification
7. Extended file attribute handling
8. Unicode support
9. Opportunistic locks

In the OSI networking model, Microsoft SMB Protocol is most often used as an Application layer or a Presentation layer protocol, and it relies on lower-level protocols for transport. The transport layer protocol that Microsoft SMB Protocol is most often used with is NetBIOS over TCP/IP (NBT). However, Microsoft SMB Protocol can also be used without a separate transport protocol, the Microsoft SMB Protocol/NBT combination is generally used for backward compatibility.
The Microsoft SMB Protocol is a client-server implementation and consists of a set of data packets, each containing a request sent by the client or a response sent by the server. These packets can be broadly classified as follows:

1. Session control packets - Establishes and discontinues a connection to shared server resources.
2. File access packets - Accesses and manipulates files and directories on the remote server.
3. General message packets - Sends data to print queues, mailslots, and named pipes, and provides data about the status of print queues.

Some message packets may be grouped and sent in one transmission to reduce response latency and increase network bandwidth. This is called "batching.".

Opportunistic Locks

An opportunistic lock (also called an oplock) is a lock placed by a client on a file residing on a server. In most cases, a client requests an opportunistic lock so it can cache data locally, thus reducing network traffic and improving apparent response time. Opportunistic locks are used by network redirectors on clients with remote servers, as well as by client applications on local servers. Opportunistic locks coordinate data caching and coherency between clients and servers and among multiple clients. Data that is coherent is data that is the same across the network. In other words, if data is coherent, data on the server and all the clients is synchronized.

Opportunistic locks are not commands by the client to the server. They are requests from the client to the server. From the point of view of the client, they are opportunistic. In other words, the server grants such locks whenever other factors make the locks possible.
When a local application requests access to a remote file, the implementation of opportunistic locks is transparent to the application. The network redirector and the server involved open and close the opportunistic locks automatically. However, opportunistic locks can also be used when a local application requests access to a local file and access by other applications and processes must be delegated to prevent corruption of the file. In this case, the local application directly requests an opportunistic lock from the local file system and caches the file locally. When used in this way, the opportunistic lock is effectively a semaphore managed by the local server, and is mainly used for the purposes of data coherency in the file and file access notification.
Before using opportunistic locks in your application, you should be familiar with the file access and sharing modes.
The maximum number of concurrent opportunistic locks that you can create is limited only by the amount of available memory. For example, the maximum number created in test conditions with Windows 2000 and the NTFS file system is 359,000 locks.
Local applications should not attempt to request opportunistic locks from remote servers. An error will be returned by DeviceIoControl() if an attempt is made to do this.
Opportunistic locks are of very limited use for applications. The only practical use is to test a network redirector or a server opportunistic lock handler. Typically, file systems implement support for opportunistic locks. Applications generally leave opportunistic lock management to the file system drivers. Anyone implementing a file system should use the Installable File System (IFS) Kit. Anyone developing a device driver other than an installable file system should use the Windows Driver Kit (WDK).
Opportunistic locks and the associated operations are a superset of the opportunistic lock portion of the Common Internet File System (CIFS) protocol, an Internet Draft. The CIFS protocol is an enhanced version of the Server Message Block (SMB) protocol. The CIFS Internet Draft explicitly identifies that a CIFS implementation may implement opportunistic locks by refusing to grant them. For additional information about opportunistic locks, please refer to the CIFS Internet Draft document. Any discrepancies between this topic and the current CIFS Internet Draft should be resolved in favor of the CIFS Internet Draft.

Sparse Files

A file in which much of the data is zeros is said to contain a sparse data set. Files like these are typically very large, for example, a file containing image data to be processed or a matrix within a high-speed database. The problem with files containing sparse data sets is that the majority of the file does not contain useful data and, because of this, they are an inefficient use of disk space.
The file compression in the NTFS file system is a partial solution to the problem. All data in the file that is not explicitly written is explicitly set to zero. File compression compacts these ranges of zeros. However, a drawback of file compression is that access time may increase due to data compression and decompression.
Support for sparse files is introduced in the NTFS file system as another way to make disk space usage more efficient. When sparse file functionality is enabled, the system does not allocate hard drive space to a file except in regions where it contains nonzero data. When a write operation is attempted where a large amount of the data in the buffer is zeros, the zeros are not written to the file. Instead, the file system creates an internal list containing the locations of the zeros in the file, and this list is consulted during all read operations. When a read operation is performed in areas of the

file where zeros were located, the file system returns the appropriate number of zeros in the buffer allocated for the read operation. In this way, maintenance of the sparse file is transparent to all processes that access it, and is more efficient than compression for this particular scenario.
The default data value of a sparse file is zero; however, it can be set to other values.

Sparse File Operations

To determine whether a file system supports sparse files, call the GetVolumeInformation() function and examine the FILE_SUPPORTS_SPARSE_FILES bit flag.
Most applications are not aware of sparse files and will not create sparse files. The fact that an application is reading a sparse file is transparent to the application. An application that is aware of sparse-files should determine whether its data set is suitable to be kept in a sparse file. After that determination is made, the application must explicitly declare a file as sparse, using the FSCTL_SET_SPARSE control code.
After an application has set a file to be sparse, the application can use the FSCTL_SET_ZERO_DATA control code to set a region of the file to zero. In addition, the application can use the FSCTL_QUERY_ALLOCATED_RANGES control code to speed searches for nonzero data in the sparse file.
When you perform a write operation (with a function or operation other than FSCTL_SET_ZERO_DATA) whose data consists of nothing but zeros, zeros will be written to the disk for the entire length of the write. To zero out a range of the file and maintain sparseness, use FSCTL_SET_ZERO_DATA.
A sparseness-aware application may also set an existing file to be sparse. If an application sets an existing file to be sparse, it should then scan the file for regions which contain zeros, and use FSCTL_SET_ZERO_DATA to reset those regions, thereby possibly deallocating some physical disk storage. An application upgraded to sparse file awareness should perform this conversion.
When you perform a read operation from a zeroed-out portion of a sparse file, the operating system may not read from the hard drive. Instead, the system recognizes that the portion of the file to be read contains zeros, and it returns a buffer full of zeros without actually reading from the disk.
As with any other file, the system can write data to or read data from any position in a sparse file. Nonzero data being written to a previously zeroed portion of the file may result in allocation of disk space. Zeros being written over nonzero data (only with FSCTL_SET_ZERO_DATA) may result in a deallocation of disk space. Note that it is up to the application to maintain sparseness by writing zeros with FSCTL_SET_ZERO_DATA.
Defragmentation tools that handle compressed files on NTFS file systems will correctly handle sparse files on NTFS file system volumes. Large and highly fragmented sparse files can exceed the NTFS limitation on disk extents before available space is used.

Obtaining the Size of a Sparse File

Use the GetCompressedFileSize() function to obtain the actual size allocated on disk for a sparse file. This total does not include the size of the regions which were deallocated because they were filled with zeroes. Use the GetFileSize() function to determine the total size of a file, including the size of the sparse regions that were deallocated.

Sparse Files and Disk Quotas

A sparse file affects user quotas by the nominal size of the file, not the actual allocated amount of disk space. That is, creating a 50-MB file with all zero bytes consumes 50 MB of that user's quota. This means that as the user writes data to the sparse file, he need not worry about exceeding his hard quota limit, because he has already been charged for the space. System administrators should not count on the size of a sparse file remaining small. Therefore they should not grant their users hard quota limits that exceed the physical space available, despite the use of sparse files.

Symbolic Links

A symbolic link is a file-system object that points to another file system object. The object being pointed to is called the target. Symbolic links are transparent to users; the links appear as normal files or directories, and can be acted upon by the user or application in exactly the same manner. Symbolic links are designed to aid in migration and application compatibility with UNIX operating systems. Microsoft has implemented its symbolic links to function just like UNIX links. Symbolic links are available in NTFS starting with Windows Vista.

Symbolic Link Effects on File Systems Functions

Several standard file functions are affected by the use of symbolic links. The following Table lists those functions and describes the changes in behavior. In the descriptions column, the following terms are used:

1. Source file - The original file that is to be copied.
2. Destination file - The newly created copy of the file.
3. Target - The entity that a symbolic link points to.

Functions	Description
CopyFile()	If the source file is a symbolic link, the actual file copied is the target of the symbolic link. If the destination file already exists and is a symbolic link, the symbolic link is overwritten by the source file.
CopyFileEx()	The flag COPY_FILE_COPY_SYMLINK is now available. If COPY_FILE_COPY_SYMLINK is specified and: 1. If the source file is a symbolic link, the symbolic link is copied, not the target file. 2. If the source file is not a symbolic link, there is no change in behavior. 3. If the destination file is an existing symbolic link, the symbolic link is overwritten, not the target file. 4. If COPY_FILE_FAIL_IF_EXISTS is also specified, and the destination file is an existing symbolic link, the operation fails in all cases. If COPY_FILE_COPY_SYMLINK is not specified and: 1. If COPY_FILE_FAIL_IF_EXISTS is also specified, and the

	destination file is an existing symbolic link, the operation fails only if the target of the symbolic link exists. 2. If COPY_FILE_FAIL_IF_EXISTS is not specified, there is no change in behavior.
CreateFile()	If the call to this function creates a new file, there is no change in behavior. If FILE_FLAG_OPEN_REPARSE_POINT is specified and: 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 CREATE_ALWAYS, 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 and: 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.
CreateHardLink()	If the path points to a symbolic link, the function creates a hard link to the target.
DeleteFile()	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.
FindFirstChangeNotification()	If the path points to a symbolic link, the notification handle is created for the target. If an application has registered to receive change notifications for a directory that contains symbolic links, the application is only notified when the symbolic links have been changed, not the target files.
FindFirstFile()	If the path points to a symbolic link, the WIN32_FIND_DATA buffer contains information about the symbolic link, not the target.
FindFirstFileEx()	If the path points to a symbolic link, the WIN32_FIND_DATA buffer contains information about the symbolic link, not the target.
FindNextFile()	If the path points to a symbolic link, the WIN32_FIND_DATA buffer contains information about the symbolic link, not the target.
GetBinaryType()	If the path points to a symbolic link, the target file is used.
GetCompressedFileSize()	If the path points to a symbolic link, the function returns the file size of the target.
GetDiskFreeSpace()	If the path points to a symbolic link, the operation is performed on the target.
GetDiskFreeSpaceEx()	If the path points to a symbolic link, the operation is performed on the target.
GetFileAttributes()	If the path points to a symbolic link, the function returns attributes for the symbolic link.
GetFileAttributesEx()	If the path points to a symbolic link, the function returns attributes for the symbolic link.

GetFileSecurity()	If the path points to a symbolic link, the function returns attributes for the symbolic link.
GetTempPath()	If the path points to a symbolic link, the temp path name maintains any symbolic links.
GetVolumeInformation()	If the path points to a symbolic link, the function returns volume information for the target.
SetFileAttributes()	If the path points to a symbolic link, the function returns attributes for the symbolic link.
SetFileSecurity()	If the path points to a symbolic link, the function returns attributes for the symbolic link.
SetVolumeName()	If the path points to a symbolic link, the function acts on the target.

Programming Considerations

Keep the following programming considerations in mind when working with symbolic links:

1. Symbolic links can point to a non-existent target.
2. When creating a symbolic link, the operating system does not check to see if the target exists.
3. If an application tries to open a non-existent target, ERROR_FILE_NOT_FOUND is returned.
4. Symbolic links are reparse points.
5. There is a maximum of 31 reparse points (and therefore symbolic links) allowed in a particular path.

Creating Symbolic Links

The function CreateSymbolicLink() allows you to create symbolic links using either an absolute or relative path. Symbolic links can either be absolute or relative links. Absolute links are links that specify each portion of the path name; relative links are determined relative to where relative–link specifiers are in a specified path. Relative links are specified using the following conventions:

1. Dot (. and ..) conventions - for example, "..\" resolves the path relative to the parent directory.
2. Names with no slashes (\) - for example, "tmp" resolves the path relative to the current directory.
3. Root relative, for example, "\windows\system32" resolves to the <current drive>:\windows\system32. directory
4. Current working directory-relative - for example, if the current working directory is c:\windows\system32, "c:file.txt" resolves to c:\windows\system32\file.txt.
   Note: If you specify a current working directory–relative link, it is created as an absolute link, due to the way the current working directory is processed based on the user and the thread.

A symbolic link can also contain both junction points and mounted folders as a part of the path name. Symbolic links can point directly to a remote file or directory using the UNC path. Relative symbolic links are restricted to a single volume.

Example of an Absolute Symbolic Link

In this example, the original path contains a component, 'x', which is an absolute symbolic link. When 'x' is encountered, the fragment of the original path up to and including 'x' is completely replaced by the path that is pointed to by 'x'. The remainder of the path after 'x' is appended to this new path. This now becomes the modified path.

X: C:\alpha\beta\absLink\gamma\file
Link: absLink maps to \\machineB\share
Modified Path: \\machineB\share\gamma\file

Example of a Relative Symbolic Links

In this example, the original path contains a component 'x', which is a relative symbolic link. When 'x' is encountered, 'x' is completely replaced by the new fragment pointed to by 'x'. The remainder of the path after 'x', is appended to the new path. Any dots (..) in this new path replace components that appear before the dots (..). Each set of dots replace the component preceding. If the number of dots (..) exceed the number of components, an error is returned. Otherwise, when all component replacement has finished, the final, modified path remains.

X: C:\alpha\beta\link\gamma\file
Link: link maps to ..\..\theta
Modified Path: C:\alpha\beta\..\..\theta\gamma\file
Final Path: C:\theta\gamma\file

File Management Program Examples

The following samples use the file management functions:

1. Appending One File to Another File
2. Creating and Using a Temporary File
3. Locking and Unlocking Byte Ranges in Files
4. Opening a File for Reading or Writing
5. Retrieving and Changing File Attributes
6. Testing for the End of a File
7. Using Streams

Appending One File to Another File Program Example

The following code example shows you how to open and close files, read and write to files, and lock and unlock files.
In the example, the application appends one file to the end of another file. First, the application opens the file being appended with permissions that allow only the application to write to it. However, during the append process other processes can open the file with read-only permission, which provides a snapshot view of the file being appended. Then, the file is locked during the actual append process to ensure the integrity of the data being written to the file.
This example does not use transactions. If you were using transacted operations, you would only be able have read-only access. In this case, you would only see the appended data after the transaction commit operation completed. The example also shows that the application opens two files by using CreateFile():

1. testcreatefile1.txt is opened for reading.

2. testcreatefile2.txt is opened for writing and shared reading.

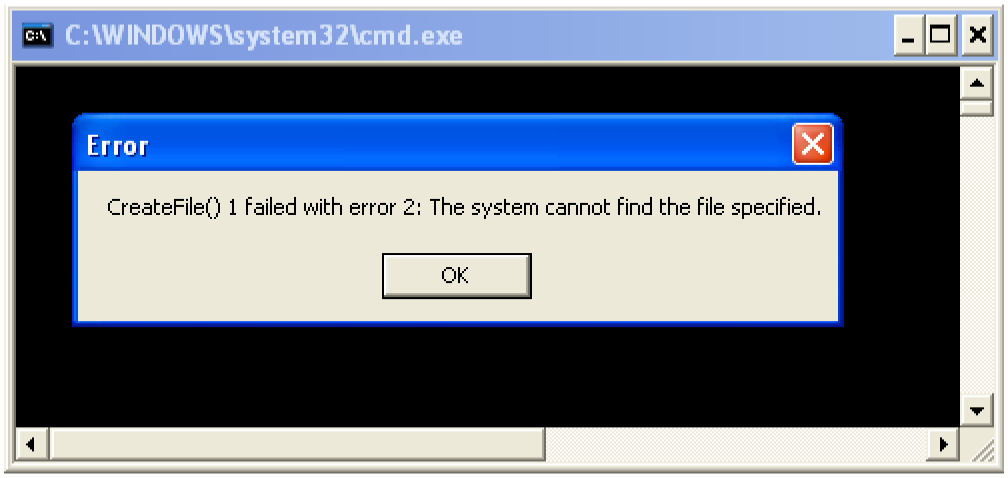
Then the application uses ReadFile() and WriteFile() to append the contents of testcreatefile1.txt to the end of testcreatefile2.txt by reading and writing the 4K blocks. However, before writing to the second file, the application uses SetFilePointer() to set the pointer of the second file to the end of that file, and uses LockFile() to lock the area to be written. This prevents another thread or process with a duplicate handle from accessing the area while the write operation is in progress. When each write operation is complete, UnlockFile() is used to unlock the locked area.

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>
// A safer version for string manipulation
#include <strsafe.h>

// A prototype that receives a function name, displaying
// system error code and its respective message
void DisplayErrorBox(LPTSTR lpszFunction);

int main(int argc, WCHAR *argv[])
{
      HANDLE hFile;
      HANDLE hAppend;
      LPWSTR pFileName1 = L"C:\\testcreatefile1.txt";
      LPWSTR pFileName2 = L"testcreatefile2.txt";
      DWORD  dwBytesRead, dwBytesWritten, dwPos;
      BYTE   buff[4096];

      // Open the existing file
      hFile = CreateFile(pFileName1, // open testcreatefile1.txt
                    GENERIC_READ,             // open for reading
                    0,                        // do not share
                    NULL,                     // no security
                    OPEN_EXISTING,            // existing file only
                    FILE_ATTRIBUTE_NORMAL,    // normal file
                    NULL);                    // no attr. template

      if (hFile == INVALID_HANDLE_VALUE)
      {
         DisplayErrorBox(L"CreateFile() 1");
         return 1;
      }
      else
            wprintf(L"CreateFile() 1 is OK. %s was opened successfully!\n",
pFileName1);

      // Open the existing file, or if the file does not exist, create a new
file
      hAppend = CreateFile(pFileName2, // open testcreatefile2.txt
                        FILE_APPEND_DATA,         // open for writing
                        FILE_SHARE_READ,          // allow multiple readers
                        NULL,                     // no security
```

```
                         OPEN_ALWAYS,                // open or create
                         FILE_ATTRIBUTE_NORMAL,      // normal file
                         NULL);                      // no attr. template

      if (hAppend == INVALID_HANDLE_VALUE)
      {
         DisplayErrorBox(L"CreateFile() 2");
         return 1;
      }
      else
            wprintf(L"CreateFile() 2 is OK. %s was created  &  opened
successfully!\n", pFileName2);

      // Append the first file to the end of the second file.
      // Lock the second file to prevent another process from
      // accessing it while writing to it. Unlock the file when writing is
complete
      wprintf(L"Reading %s file content...\n", pFileName1);
      while (ReadFile(hFile, buff, sizeof(buff), &dwBytesRead, NULL)&&
dwBytesRead > 0)
        {
            dwPos = SetFilePointer(hAppend, 0, NULL, FILE_END);
            wprintf(L"SetFilePointer() returns %u\n", dwPos);

            if(LockFile(hAppend, dwPos, 0, dwBytesRead, 0) != 0)
                  wprintf(L"LockFile() is pretty fine!\n");
            else
                  DisplayErrorBox(L"LockFile()");

            if(WriteFile(hAppend, buff, dwBytesRead, &dwBytesWritten, NULL) !=
0)
                  wprintf(L"WriteFile() is pretty fine!\n");
            else
                  DisplayErrorBox(L"WriteFile()");

            if(UnlockFile(hAppend, dwPos, 0, dwBytesRead, 0) != 0)
                  wprintf(L"UnlockFile() is pretty fine!\n");
            else
                  DisplayErrorBox(L"UnlockFile()");
        }

      // Close both files.
      if(CloseHandle(hFile) != 0)
            wprintf(L"Closing hFile handle!\n");
      else
            DisplayErrorBox(L"CloseHandle(hFile)");

      if(CloseHandle(hAppend) != 0)
            wprintf(L"Closing hAppend handle!\n");
      else
            DisplayErrorBox(L"CloseHandle(hAppend)");
      return 0;
}

void DisplayErrorBox(LPTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL );

    // Display the error message and clean up
    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,

(lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(WCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(WCHAR),  L"%s failed with error %d:
%s",
        lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, L"Error", MB_OK);

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}
```

Build and run the project. The following screenshot is a sample output when the first file cannot be found.



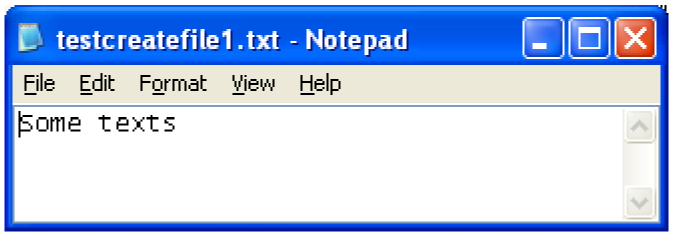
C:\WINDOWS\system32\cmd.exe

Error

CreateFile() 1 failed with error 2: The system cannot find the file specified.

OK

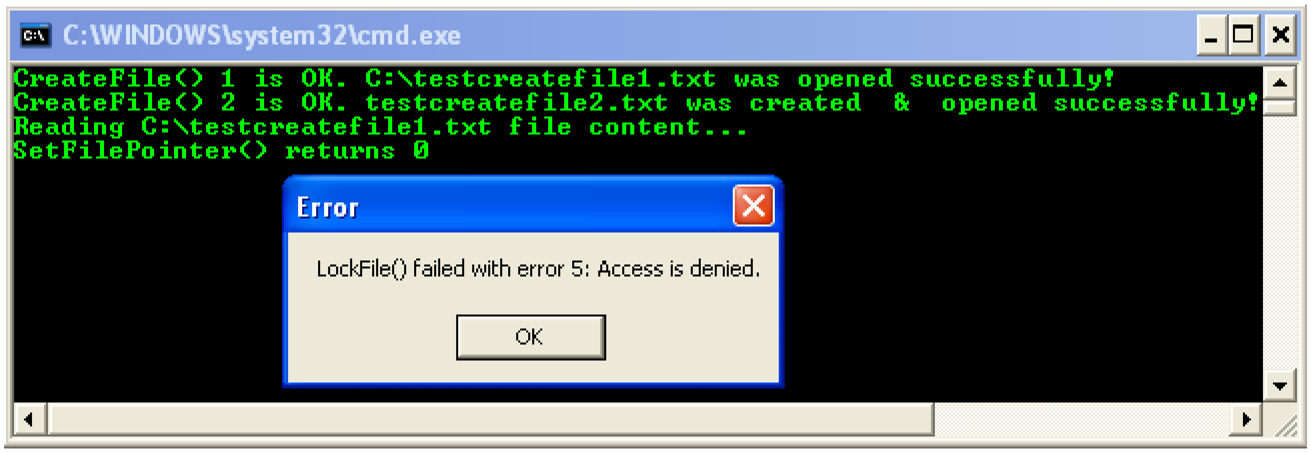
Create a text file named testcreatefile1.txt, write some text and save it in the root C: drive.



testcreatefile1.txt - Notepad

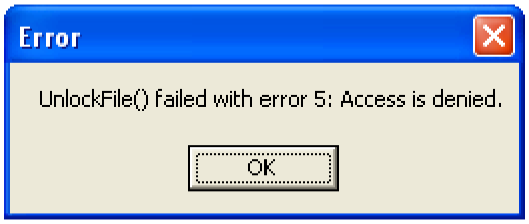
Some texts

Next, we re-run the project.



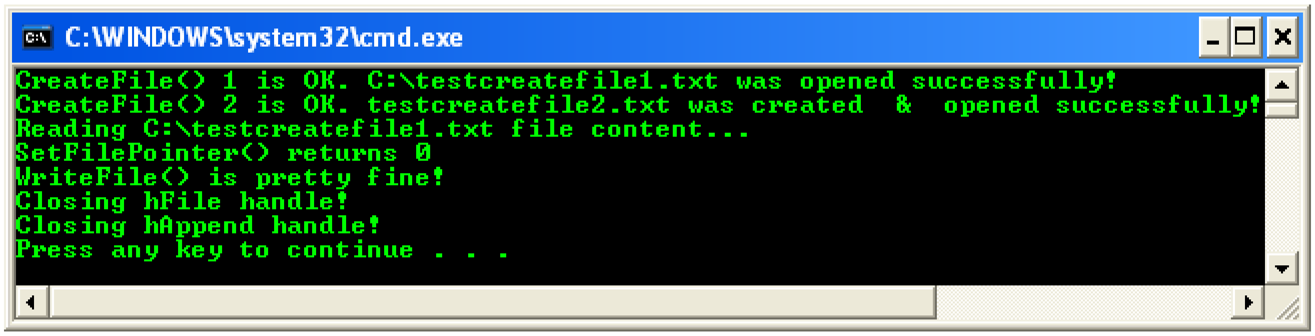
```
CreateFile() 1 is OK. C:\testcreatefile1.txt was opened successfully!
CreateFile() 2 is OK. testcreatefile2.txt was created  &  opened successfully!
Reading C:\testcreatefile1.txt file content...
SetFilePointer() returns 0
```

LockFile() failed with error 5: Access is denied.



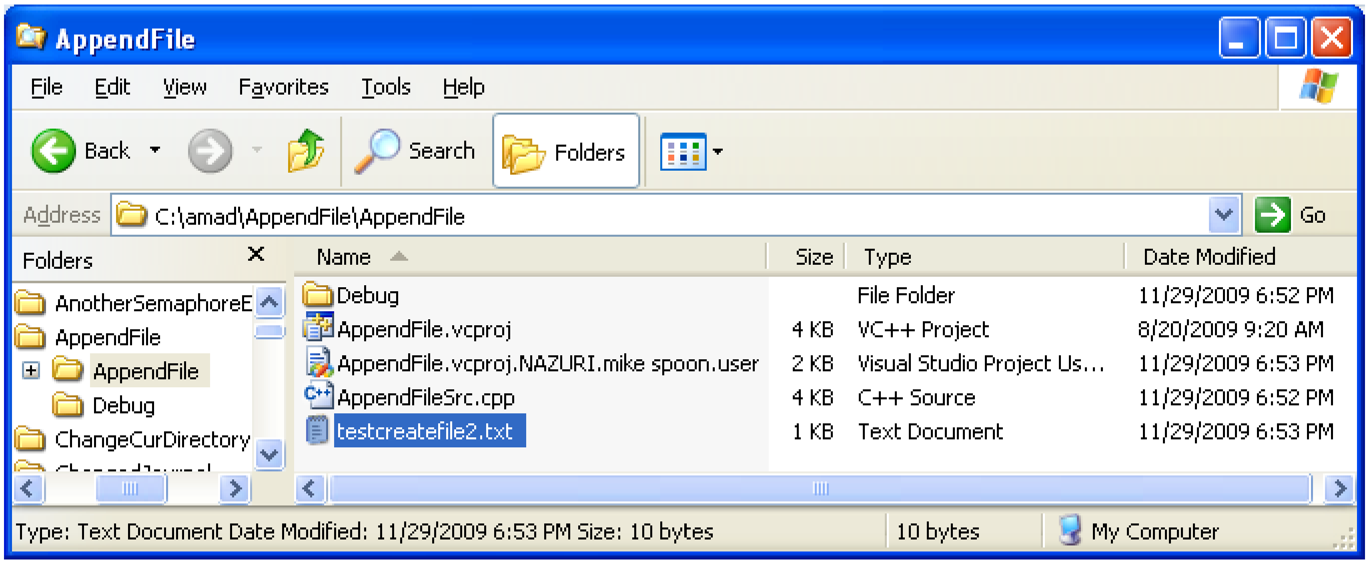
UnlockFile() failed with error 5: Access is denied.

Well, there are errors. However, when we re-run the program second times, those errors disappeared.



```
CreateFile() 1 is OK. C:\testcreatefile1.txt was opened successfully!
CreateFile() 2 is OK. testcreatefile2.txt was created  &  opened successfully!
Reading C:\testcreatefile1.txt file content...
SetFilePointer() returns 0
WriteFile() is pretty fine!
Closing hFile handle!
Closing hAppend handle!
Press any key to continue . . .
```

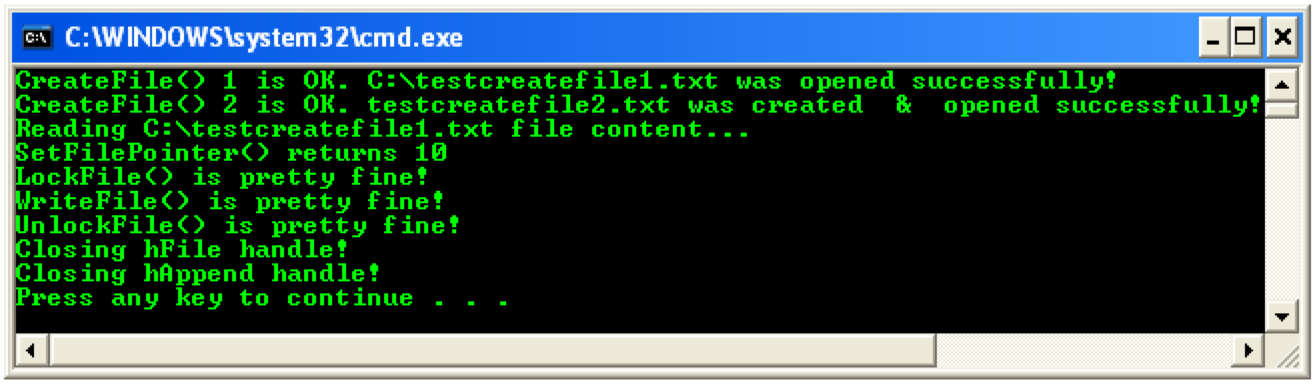
The second file was created successfully under the project's folder as shown below.



Name	Size	Type	Date Modified
Debug		File Folder	11/29/2009 6:52 PM
AppendFile.vcproj	4 KB	VC++ Project	8/20/2009 9:20 AM
AppendFile.vcproj.NAZURI.mike spoon.user	2 KB	Visual Studio Project Us...	11/29/2009 6:53 PM
AppendFileSrc.cpp	4 KB	C++ Source	11/29/2009 6:52 PM
testcreatefile2.txt	1 KB	Text Document	11/29/2009 6:53 PM

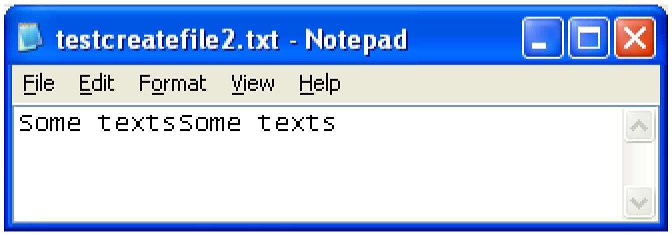
Type: Text Document Date Modified: 11/29/2009 6:53 PM Size: 10 bytes 10 bytes

Then, re-run the program again so that the second run will append the first file's content to the second file.



```
CreateFile() 1 is OK. C:\testcreatefile1.txt was opened successfully!
CreateFile() 2 is OK. testcreatefile2.txt was created  &  opened successfully!
Reading C:\testcreatefile1.txt file content...
SetFilePointer() returns 10
LockFile() is pretty fine!
WriteFile() is pretty fine!
UnlockFile() is pretty fine!
Closing hFile handle!
Closing hAppend handle!
Press any key to continue . . .
```

Then, open the testcreatefile2.txt. We can see in testcreatefile2.txt that the content of testcreatefile1.txt was appended.



```
Some textsSome texts
```

Creating and Using a Temporary File Program Example

Applications can obtain unique file names for temporary files by using the GetTempFileName() function. The GetTempPath() function retrieves the path to the directory where temporary files should be created. The following procedure shows you how an application copies one file to another.

To copy one file to another:

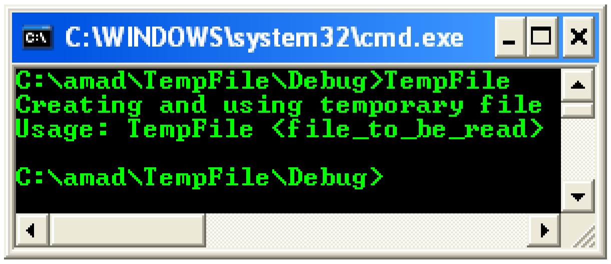
1. The application opens the Original.txt file by using CreateFile().
2. The application retrieves a temporary file path and file name by using the GetTempPath() and GetTempFileName() functions, and then uses CreateFile() to create the temporary file.
3. The application reads 64K blocks into a buffer, converts the buffer contents to uppercase, and writes the converted buffer to the temporary file.
4. When all of Original.txt is written to the temporary file, the application closes both files, and renames the temporary file to Allcaps.txt by using the MoveFileEx() function.

The following program example shows you the code how to copy one file to another, and notes that the target file is an uppercase version of the first file.

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.

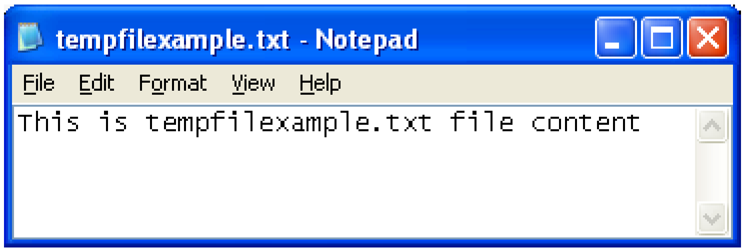
```
#include <windows.h>
#include <stdio.h>
// A safer version for string manipulation
#include <strsafe.h>

// A prototype that receives a function name, displaying
// system error code and its respective message
void DisplayErrorBox(LPTSTR lpszFunction);

#define BUFSIZE 512

int wmain(int argc, WCHAR *argv[])
{
    HANDLE hFile;
    HANDLE hTempFile;
    DWORD dwRetVal, dwRetVal2;
    DWORD dwBytesRead;
    DWORD dwBytesWritten;
    DWORD dwBufSize=BUFSIZE;
    UINT uRetVal;
    WCHAR szTempName[BUFSIZE];
    char buffer[BUFSIZE];
    WCHAR lpPathBuffer[BUFSIZE];
    BOOL fSuccess;
      LPWSTR pFileName = L"Allcapfile.txt";

    if(argc != 2)
    {
            wprintf(L"Creating and using temporary file\n");
        wprintf(L"Usage: %s <file_to_be_read>\n", argv[0]);
            // Non-zero means error
        return -1;
    }

    // Open the existing file.
    hFile = CreateFile(argv[1],                // file name
                       GENERIC_READ,           // open for reading
                       0,                      // do not share
                       NULL,                   // default security
                       OPEN_EXISTING,         // existing file only
                       FILE_ATTRIBUTE_NORMAL, // normal file
                       NULL);                 // no template
    if (hFile == INVALID_HANDLE_VALUE)
    {
        DisplayErrorBox(L"First CreateFile()");
        return (1);
    }
      else
            wprintf(L"%s was successfully opened for reading!\n", argv[1]);

     // Get the temp path.
    dwRetVal = GetTempPath(dwBufSize,     // length of the buffer
                           lpPathBuffer); // buffer for path
    if (dwRetVal > dwBufSize || (dwRetVal == 0))
    {
        DisplayErrorBox(L"GetTempPath()");
        return (2);
    }
      else
            wprintf(L"GetTempPath() - Windows temp path is:\n %s\n",
lpPathBuffer);

    // Create a temporary file.
    uRetVal = GetTempFileName(lpPathBuffer, // directory for tmp files
                              L"NEW",   // temp file name prefix
                              0,            // create unique name
                              szTempName);  // buffer for name
    if (uRetVal == 0)
    {
        DisplayErrorBox(L"GetTempFileName()");
        return (3);
    }
      else
            wprintf(L"GetTempFileName() - temp file name is:\n %s\n",
szTempName);

    // Create the new file to write the upper-case version to.
    hTempFile = CreateFile((LPTSTR) szTempName, // file name
                           GENERIC_READ | GENERIC_WRITE, // open r-w
                           0,                     // do not share
                           NULL,                  // default security
                           CREATE_ALWAYS,         // overwrite existing
                           FILE_ATTRIBUTE_NORMAL,// normal file
                           NULL);                 // no template
    if (hTempFile == INVALID_HANDLE_VALUE)
    {
        DisplayErrorBox(L"Second CreateFile()");
        return (4);
    }
      else
            wprintf(L"%s was successfully created/overwritten\n   for read &
write!\n", szTempName);

    // Read BUFSIZE blocks to the buffer. Change all characters in
    // the buffer to upper case. Write the buffer to the temporary file.
    do
    {
        if (ReadFile(hFile,buffer,BUFSIZE,&dwBytesRead,NULL))
        {
                  wprintf(L"Reading from %s...\n", argv[1]);

                  // The following Unicode version fail, though the program runs
                  //  without error
                  // dwRetVal2 = CharUpperBuff((LPWSTR)buffer, dwBytesRead);
                  // dwRetVal2 = CharUpperBuffW((LPWSTR)buffer, dwBytesRead);
            dwRetVal2 = CharUpperBuffA(buffer, dwBytesRead);

                  if(dwRetVal2 = dwBytesRead)
                        wprintf(L"CharUpperBuffA() looks fine! Return %d\n",
dwRetVal2);
                  else
                        DisplayErrorBox(L"CharUpperBuffA()");

            fSuccess =
WriteFile(hTempFile,buffer,dwBytesRead,&dwBytesWritten,NULL);

            if (!fSuccess)
            {
                DisplayErrorBox(L"WriteFile()");
                return (5);
            }
                  else
                        wprintf(L"Writing to %s...\n", szTempName);
        }
        else
        {
            DisplayErrorBox(L"ReadFile()");
            return (6);
        }
    } while (dwBytesRead == BUFSIZE);

    // Close the handles to the files.
    fSuccess = CloseHandle (hFile);
    if (!fSuccess)
    {
       DisplayErrorBox(L"CloseHandle(hFile)");
       return (7);
    }
      else
            wprintf(L"Closing hFile handle...\n");

    fSuccess = CloseHandle (hTempFile);
    if (!fSuccess)
    {
       DisplayErrorBox(L"CloseHandle(hTempFile)");
       return (8);
    }
      else
            wprintf(L"Closing hTempFile handle...\n");

    // Move the temporary file to the new text file.
    fSuccess = MoveFileEx(szTempName, pFileName, MOVEFILE_REPLACE_EXISTING);
    if (!fSuccess)
    {
        DisplayErrorBox(L"MoveFileEx()");
        return (9);
    }
    else
      {
            wprintf(L"Moving (renaming) the %s to %s\n", szTempName,pFileName);
        wprintf(L"All caps version of %s written to %s\n", argv[1], pFileName);
      }

    return (0);
}

void DisplayErrorBox(LPTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL );

    // Display the error message and clean up
    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,

(lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(WCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(WCHAR),  L"%s failed with error %d:
%s",
        lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, L"Error", MB_OK);

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}
```

Build and run the project. The following screenshot is a sample output without any argument.



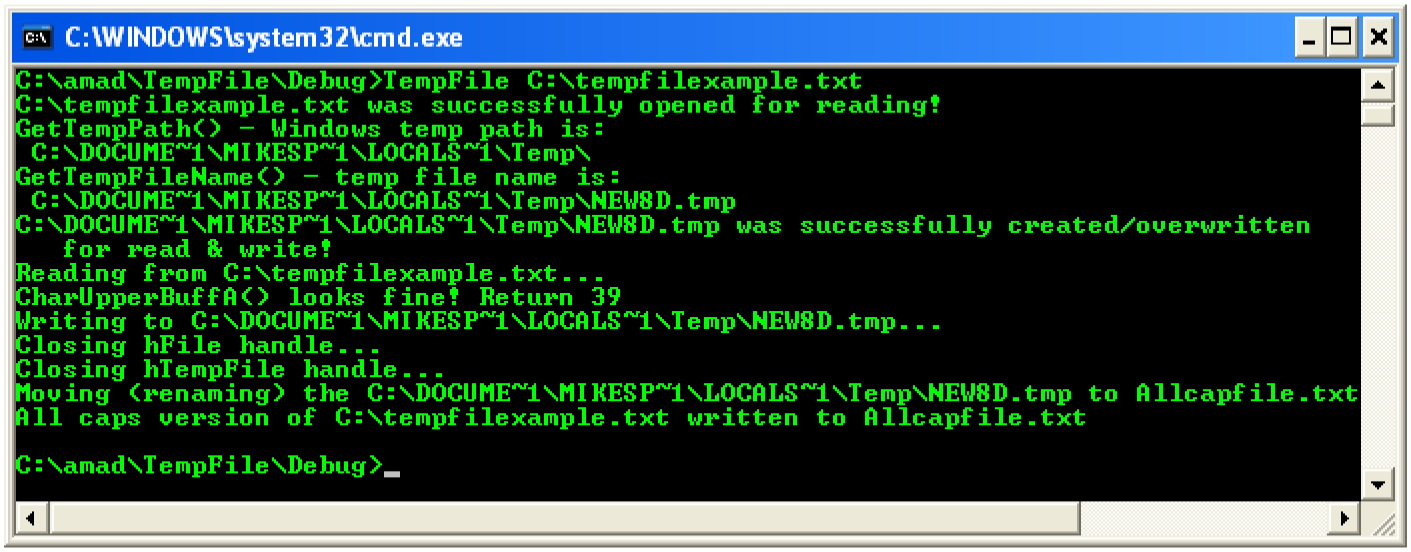
```
C:\amad\TempFile\Debug>TempFile
Creating and using temporary file
Usage: TempFile <file_to_be_read>

C:\amad\TempFile\Debug>
```

Create a text file named tempfilexample.txt on C drive. Put some text and save it.



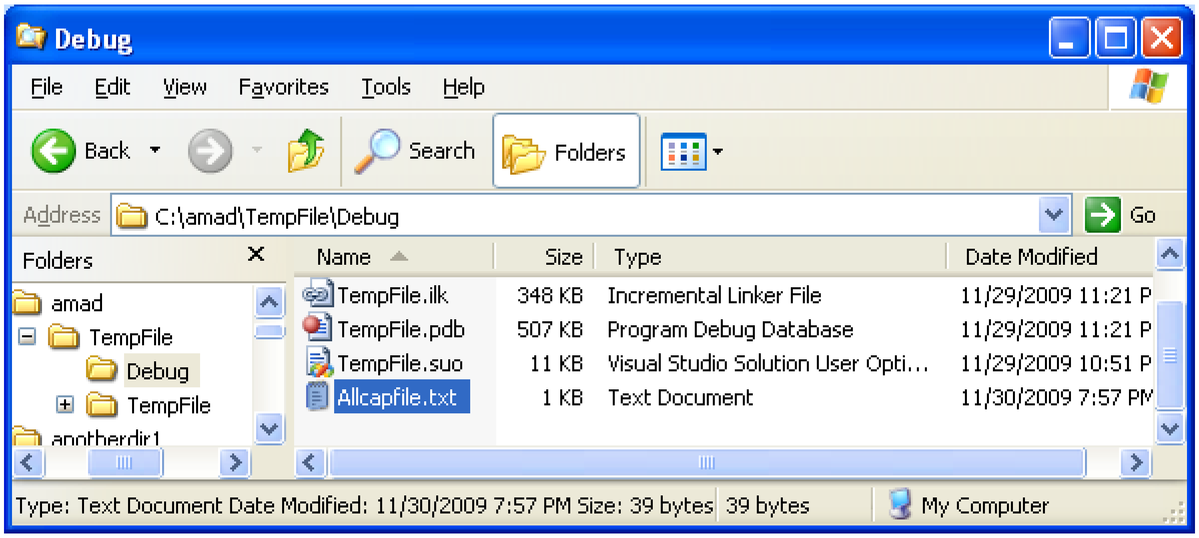
```
This is tempfilexample.txt file content
```

Re-run the project using the previously text file name as the argument.



```
C:\amad\TempFile\Debug>TempFile C:\tempfilexample.txt
C:\tempfilexample.txt was successfully opened for reading!
GetTempPath() - Windows temp path is:
 C:\DOCUME~1\MIKESP~1\LOCALS~1\Temp\
GetTempFileName() - temp file name is:
 C:\DOCUME~1\MIKESP~1\LOCALS~1\Temp\NEW8D.tmp
C:\DOCUME~1\MIKESP~1\LOCALS~1\Temp\NEW8D.tmp was successfully created/overwritten
   for read & write!
Reading from C:\tempfilexample.txt...
CharUpperBuffA() looks fine! Return 39
Writing to C:\DOCUME~1\MIKESP~1\LOCALS~1\Temp\NEW8D.tmp...
Closing hFile handle...
Closing hTempFile handle...
Moving (renaming) the C:\DOCUME~1\MIKESP~1\LOCALS~1\Temp\NEW8D.tmp to Allcapfile.txt
All caps version of C:\tempfilexample.txt written to Allcapfile.txt

C:\amad\TempFile\Debug>_
```

If there is no error, the content of the first file will be written to the second file with all the text converted to the capital letters.

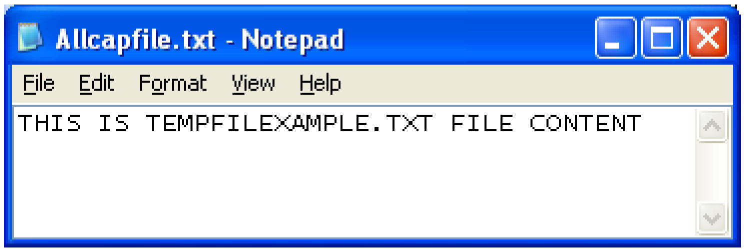


Address: C:\amad\TempFile\Debug

Name	Size	Type	Date Modified
TempFile.ilk	348 KB	Incremental Linker File	11/29/2009 11:21 P
TempFile.pdb	507 KB	Program Debug Database	11/29/2009 11:21 P
TempFile.suo	11 KB	Visual Studio Solution User Opti...	11/29/2009 10:51 P
Allcapfile.txt	1 KB	Text Document	11/30/2009 7:57 PM

Type: Text Document Date Modified: 11/30/2009 7:57 PM Size: 39 bytes 39 bytes

The following is the Allcapfile.txt file's content.



THIS IS TEMPFILEXAMPLE.TXT FILE CONTENT

However in this program example, we fail for the Unicode version although the program runs without any error. It is found that the CharUpperBuffW() failed to do its job.

Locking and Unlocking Byte Ranges in Files Program Example

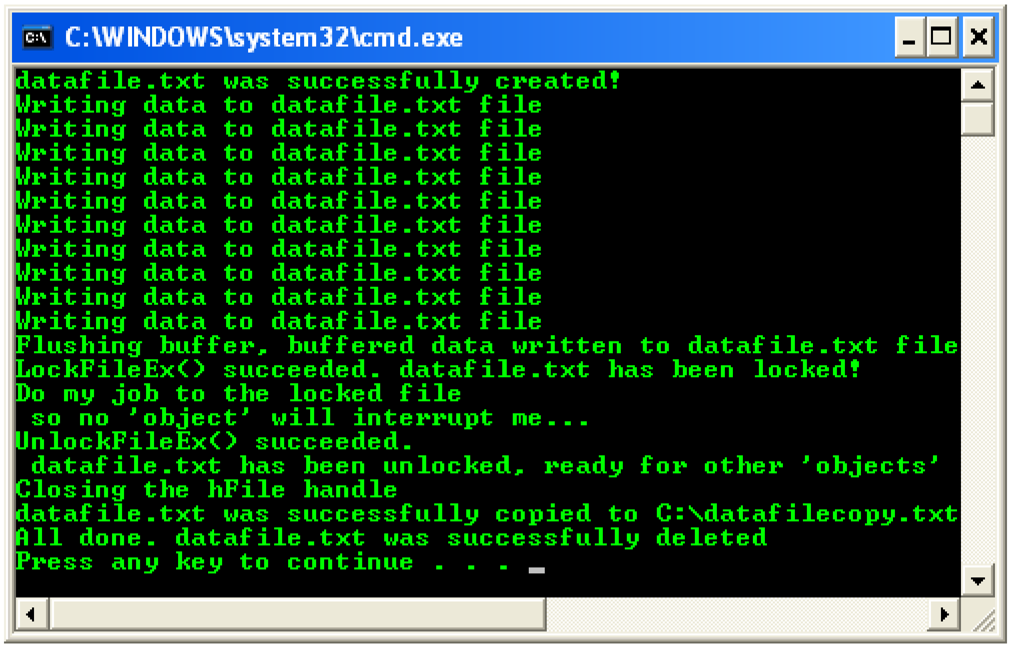
Although the system allows more than one application to open a file and write to it, applications must not write over each other's work. An application can prevent this problem by temporarily locking a byte range in a file.
The LockFile() and LockFileEx() functions lock a specified range of bytes in a file. The range may extend beyond the current end of the file. Locking part of a file gives the threads of the locking processes exclusive access to the specified byte range by using the specified file handle. Attempts to access a byte range that is locked by another process always fail. If the locking process attempts to access a locked byte range through a second file handle, the attempt fails.
The LockFileEx() function allows an application to specify one of two types of locks. An exclusive lock denies all other processes both read and write access to the specified byte range of a file. A shared lock denies all processes write access to the specified byte range of a file, including the process that first locks the byte range. This can be used to create a read-only byte range in a file.
The application unlocks the byte range by using the UnlockFile() or UnlockFileEx() function. An application should unlock all locked areas before closing a file.
The following program example shows you how to use LockFileEx(). It creates a file, writes some data to it, and then locks a section in the middle. This example does not change the data once the file is locked.

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>
// A safer version for string manipulation
#include <strsafe.h>

#define NUMWRITES 10
#define TESTSTRLEN 12

// A prototype that receives a function name, displaying
// system error code and its respective message
void DisplayErrorBox(LPTSTR lpszFunction);

// 2D  arrays
const WCHAR TestData[NUMWRITES][TESTSTRLEN] =
{
    L"TestData0 \n",
    L"TestData1 \n",
    L"TestData2 \n",
    L"TestData3 \n",
    L"TestData4 \n",
    L"TestData5 \n",
    L"TestData6 \n",
    L"TestData7 \n",
    L"TestData8 \n",
    L"TestData9 \n"
};

int wmain(int argc, WCHAR *argv[])
{
    BOOL fSuccess = FALSE;
      HANDLE hFile;
      LPWSTR pFileName = L"datafile.txt";
      LPWSTR pFileNameCpy = L"C:\\datafilecopy.txt";
      DWORD dwNumBytesWritten;

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

    if (hFile == INVALID_HANDLE_VALUE)
    {
        // Handle the error.
        DisplayErrorBox(L"CreateFile()");
            // Non-zero is error
        return (1);
    }
      else
            wprintf(L"%s was successfully created!\n", pFileName);

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

    for (int i=0; i<NUMWRITES; i++)
    {
        fSuccess = WriteFile(hFile,
                             TestData[i],
                             TESTSTRLEN,
                             &dwNumBytesWritten,
                             NULL);  // sync operation.
        if (!fSuccess)
        {
           // Handle the error.
           DisplayErrorBox(L"WriteFile()");
           return (2);
        }
            else
                  wprintf(L"Writing data to %s file\n", pFileName);
    }
```

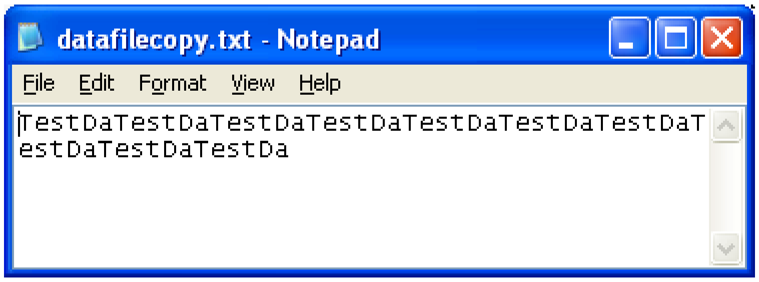
```
      // Flushes the buffers of a specified file and causes
      // all buffered data to be written to a file.
    if(FlushFileBuffers(hFile) == 0)
            DisplayErrorBox(L"FlushFileBuffers()");
      else
            wprintf(L"Flushing buffer, buffered data written to %s file\n",
pFileName);

    // 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.
       DisplayErrorBox(L"LockFileEx()");
       return (3);
    }
    else
            wprintf(L"LockFileEx() succeeded. %s has been locked!\n",
pFileName);

      wprintf(L"Do my job to the locked file\n so no \'object\' will interrupt
me...\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.
       DisplayErrorBox(L"UnlockFileEx()");
       return (4);
    }
    else
            wprintf(L"UnlockFileEx() succeeded.\n %s has been unlocked, ready
for other \'objects\'\n", pFileName);

    // Clean up handles, memory, and the created file.
    fSuccess = CloseHandle(hFile);
    if (!fSuccess)
    {
       // Handle the error.
       DisplayErrorBox(L"CloseHandle()");
       return (5);
    }
      else
            wprintf(L"Closing the hFile handle\n");

      // Just in case if you want to see the file
      if(CopyFile(pFileName, pFileNameCpy, FALSE) == 0)
            DisplayErrorBox(L"CopyFile()");
      else
            wprintf(L"%s was successfully copied to %s\n", pFileName,
pFileNameCpy);

      // Finally delete the original file
    fSuccess = DeleteFile(pFileName);

    if (!fSuccess)
    {
        // Handle the error.
       DisplayErrorBox(L"DeleteFile()");
       return (6);
    }
      else
            wprintf(L"All done. %s was successfully deleted\n", pFileName);
    return (0);
}

void DisplayErrorBox(LPTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL );

    // Display the error message and clean up
    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,

(lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(WCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(WCHAR),  L"%s failed with error %d:
%s",
        lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, L"Error", MB_OK);

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}
```

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



```
datafile.txt was successfully created!
Writing data to datafile.txt file
Writing data to datafile.txt file
Writing data to datafile.txt file
Writing data to datafile.txt file
Writing data to datafile.txt file
Writing data to datafile.txt file
Writing data to datafile.txt file
Writing data to datafile.txt file
Writing data to datafile.txt file
Writing data to datafile.txt file
Flushing buffer, buffered data written to datafile.txt file
LockFileEx() succeeded. datafile.txt has been locked!
Do my job to the locked file
 so no 'object' will interrupt me...
UnlockFileEx() succeeded.
 datafile.txt has been unlocked, ready for other 'objects'
Closing the hFile handle
datafile.txt was successfully copied to C:\datafilecopy.txt
All done. datafile.txt was successfully deleted
Press any key to continue . . . _
```

The following is a copy of the datafilecopy.txt text file's content.



```
TestDaTestDaTestDaTestDaTestDaTestDaTestDaT
estDaTestDaTestDa
```

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.

Open a File for Writing Program Example

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.

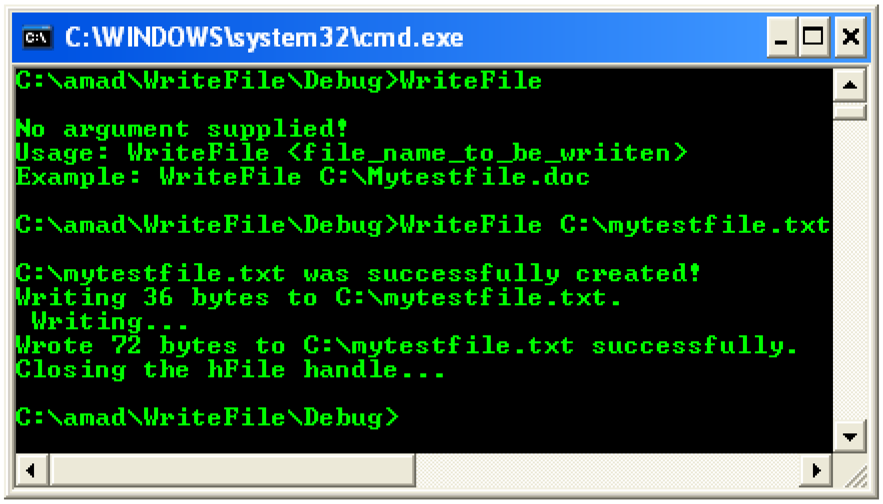
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.
Then, add the source file and give it a suitable name.

Next, add the following source code.

```
#include <windows.h>
#include <stdio.h>
// A safer version for string manipulation
#include <strsafe.h>

// A prototype that receives a function name, displaying
// system error code and its respective message
void DisplayErrorBox(LPTSTR lpszFunction);

int wmain(int argc, WCHAR *argv[])
{
    HANDLE hFile;
    WCHAR DataBuffer[] = L"This is a test string to be written.";
    DWORD dwBytesToWrite = (DWORD)wcslen(DataBuffer);
    DWORD dwBytesWritten = 0;

    printf("\n");
      // Verify the argument
    if(argc != 2)
    {
        wprintf(L"No argument supplied!\n");
            wprintf(L"Usage: %s <file_name_to_be_wriiten>\n", argv[0]);
            wprintf(L"Example: %s C:\\Mytestfile.doc\n", argv[0]);
        return 1;
    }

    hFile = CreateFile((LPCWSTR)argv[1],                  // name of the 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)
    {
        DisplayErrorBox(L"CreateFile()");
        return 1;
    }
      else
            wprintf(L"%s was successfully created!\n", argv[1]);

    wprintf(L"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
                                            // Note the Unicode/multibyte size!
                               2*(dwBytesToWrite - dwBytesWritten), // number of
bytes to write
```

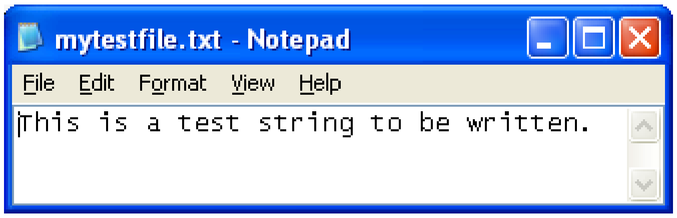
```
                                    &dwBytesWritten, // number of bytes that were
written
                                    NULL)            // no overlapped structure
            )
          {
              DisplayErrorBox(L"WriteFile()");
              if(CloseHandle(hFile) == 0)
                          DisplayErrorBox(L"CloseHandle()");
                    else
                          wprintf(L"Closing the hFile handle...\n");

              return 1;
          }
              else
                    wprintf(L" Writing...\n");
      }

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

      if(CloseHandle(hFile) == 0)
              DisplayErrorBox(L"CloseHandle()");
        else
              wprintf(L"Closing the hFile handle...\n");

        return 0;
}

void DisplayErrorBox(LPTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL );

    // Display the error message and clean up
    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,

(lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(WCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(WCHAR),  L"%s failed with error %d:
%s",
        lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, L"Error", MB_OK);

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}
```

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



```
C:\amad\WriteFile\Debug>WriteFile

No argument supplied!
Usage: WriteFile <file_name_to_be_wriiten>
Example: WriteFile C:\Mytestfile.doc

C:\amad\WriteFile\Debug>WriteFile C:\mytestfile.txt

C:\mytestfile.txt was successfully created!
Writing 36 bytes to C:\mytestfile.txt.
 Writing...
Wrote 72 bytes to C:\mytestfile.txt successfully.
Closing the hFile handle...

C:\amad\WriteFile\Debug>
```

The following is the mytestfile.txt file's content as expected.



```
This is a test string to be written.
```

Open a File for Reading Program Example

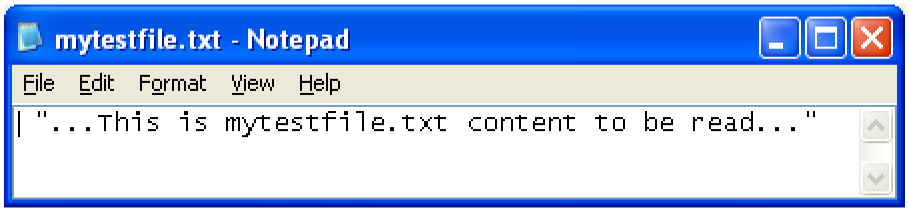
The following example uses CreateFile() to open an existing file for reading and ReadFile() to characters synchronously from the file.
In this case, CreateFile() succeeds only if the specified file already exists in the current directory. A subsequent call to open this file with CreateFile() will succeed 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.

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 <windows.h>
#include <stdio.h>
// A safer version for string manipulation
#include <strsafe.h>

// A prototype that receives a function name, displaying
// system error code and its respective message
void DisplayErrorBox(LPTSTR lpszFunction);
```

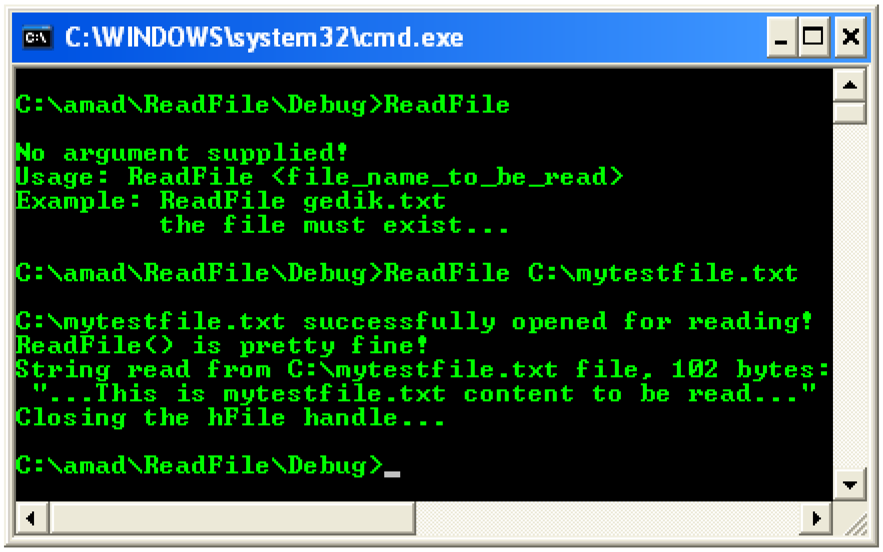
```
// Provide enough room for Unicode/multibyte lol
#define BUFFER_SIZE 200

int wmain(int argc, WCHAR *argv[])
{
    HANDLE hFile;
    DWORD dwBytesRead = 0;
    WCHAR ReadBuffer[BUFFER_SIZE] = {0};

    printf("\n");
      // Verify the argument number
    if(argc != 2)
    {
            // The file must be available
        wprintf(L"No argument supplied!\n");
            wprintf(L"Usage: %s <file_name_to_be_read>\n", argv[0]);
            wprintf(L"Example: %s gedik.txt\n", argv[0]);
            wprintf(L"         the file must exist...\n");
        return 1;
    }

    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)
    {
        DisplayErrorBox(L"CreateFile()");
        return 1;
    }
      else
            wprintf(L"%s successfully opened for reading!\n", argv[1]);

    // Read one character less than the buffer size to save room for
    // the terminating NULL character.
      // Take note also for Unicode/multibyte size
    if(ReadFile(hFile, ReadBuffer, (BUFFER_SIZE-1), &dwBytesRead, NULL)== FALSE)
    {
        DisplayErrorBox(L"ReadFile()");
        CloseHandle(hFile);
        return 1;
    }
      else
            wprintf(L"ReadFile() is pretty fine!\n");

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

        wprintf(L"String read from %s file, %d bytes: \n", argv[1],
dwBytesRead);
            // Why we need to minus 1???
        wprintf(L"%s\n", ReadBuffer+1);
    }
    else
    {
        wprintf(L"No data read from file %s\n", argv[1]);
    }

    if(CloseHandle(hFile) != 0)
            wprintf(L"Closing the hFile handle...\n");
      else
            DisplayErrorBox(L"CloseHandle()");

      return 0;
}

void DisplayErrorBox(LPTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
            FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL);

    // Display the error message and clean up
    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,

(lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(WCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(WCHAR),  L"%s failed with error %d:
%s",
        lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, L"Error", MB_OK);

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}
```

To test this program, firstly we create a simple text file, mytestfile.txt with some content on C drive.



```
"...This is mytestfile.txt content to be read..."
```

Then we run the program using the previously created text file name as the argument.



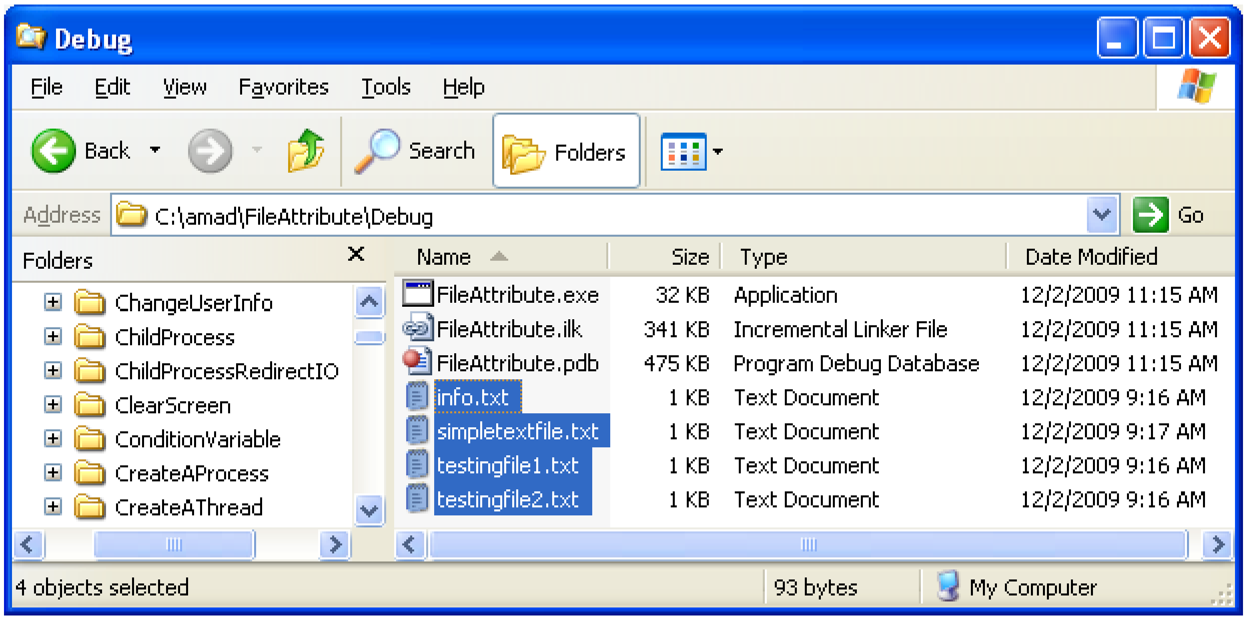
```
C:\amad\ReadFile\Debug>ReadFile

No argument supplied!
Usage: ReadFile <file_name_to_be_read>
Example: ReadFile gedik.txt
         the file must exist...

C:\amad\ReadFile\Debug>ReadFile C:\mytestfile.txt

C:\mytestfile.txt successfully opened for reading!
ReadFile() is pretty fine!
String read from C:\mytestfile.txt file, 102 bytes:
 "...This is mytestfile.txt content to be read..."
Closing the hFile handle...

C:\amad\ReadFile\Debug>_
```

Retrieving and Changing File Attributes

An application can retrieve the file attributes by using the GetFileAttributes() or GetFileAttributesEx() function. The CreateFile() and SetFileAttributes() functions can set many of the attributes. However, applications cannot set all attributes.
The following program example uses the CopyFile() function to copy all text files (.txt) in the current directory to a new directory of read-only files. Files in the new directory are changed to read only, if necessary. The application creates the directory specified as a parameter by using the CreateDirectory() function. The directory must not exist already.
The application searches the current directory for all text files by using the FindFirstFile() and FindNextFile() functions. Each text file is copied to the \TextRO directory. After a file is copied, the GetFileAttributes() function determines whether or not a file is read only. If the file is not read only, the application changes directories to \TextRO and converts the copied file to read only by using the SetFileAttributes() function. After all text files in the current directory are copied, the application closes the search handle by using the FindClose() function.
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 <strsafe.h>

// A prototype that receives a function name, displaying
// system error code and its respective message
void DisplayErrorBox(LPTSTR lpszFunction);

int wmain(int argc, WCHAR* argv[])
{
   WIN32_FIND_DATA FileData;
   HANDLE hSearch;
   DWORD dwAttrs;
   TCHAR szNewPath[MAX_PATH];
   BOOL fFinished = FALSE;

   if(argc != 2)
   {
      wprintf(L"Usage: %s <new_directory_to_be_created>\n", argv[0]);
        wprintf(L"Example: %s C:\\NewDirectory\n", argv[0]);
      return 1;
   }

   // Create a new directory
   if (!CreateDirectory(argv[1], NULL))
   {
      DisplayErrorBox(L"CreateDirectory()");
      return 1;
   }
   else
         wprintf(L"%s directory was successfully created!\n", argv[1]);

   // Start searching for text files in the current directory
   hSearch = FindFirstFile(L"*.txt", &FileData);
   if (hSearch == INVALID_HANDLE_VALUE)
   {
      DisplayErrorBox(L"FindFirstFile()");
      return 1;
   }
   else
         wprintf(L"FindFirstFile() - text file found!\n");

   // Copy each .TXT file to the new directory and change it to read only, if
not already
   wprintf(L"Copying all the text files to %s & change them to Read Only &
Hidden\n", argv[1]);
   while (!fFinished)
   {
        if(StringCchPrintf(szNewPath, MAX_PATH, L"%s\\%s", argv[1],
FileData.cFileName) == S_OK)
              wprintf(L"StringCchPrintf() is OK!\n");
        else
              DisplayErrorBox(L"StringCchPrintf()");

      if (CopyFile(FileData.cFileName, szNewPath, FALSE))
      {
         dwAttrs = GetFileAttributes(FileData.cFileName);
         if (dwAttrs==INVALID_FILE_ATTRIBUTES)
                   return 1;
             else
                   wprintf(L"GetFileAttributes() looks OK!\n");

         if (!(dwAttrs & FILE_ATTRIBUTE_READONLY))
         {
            if(SetFileAttributes(szNewPath, dwAttrs | FILE_ATTRIBUTE_READONLY |
FILE_ATTRIBUTE_HIDDEN) != 0)
                        wprintf(L"SetFileAttributes() to read only & hidden is
OK!\n");
                  else
                        DisplayErrorBox(L"SetFileAttributes()");
         }
      }
      else
      {
         DisplayErrorBox(L"CopyFile()");
         return 1;
      }

      if (!FindNextFile(hSearch, &FileData))
      {
         if (GetLastError() == ERROR_NO_MORE_FILES)
         {
            wprintf(L"Copied *.txt to %s\n", argv[1]);
            fFinished = TRUE;
         }
         else
         {
            DisplayErrorBox(L"FindNextFile()");
            return 1;
         }
      }
   }

   // Close the search handle
   if(FindClose(hSearch) != 0)
         wprintf(L"FindClose() is OK...\n");
   else
         DisplayErrorBox(L"FindClose()");

   return 0;
}

void DisplayErrorBox(LPTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
            FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL);

    // Display the error message and clean up
    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,

(lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(WCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(WCHAR), L"%s failed with error %d: %s",
        lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, L"Error", MB_OK);

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}
```

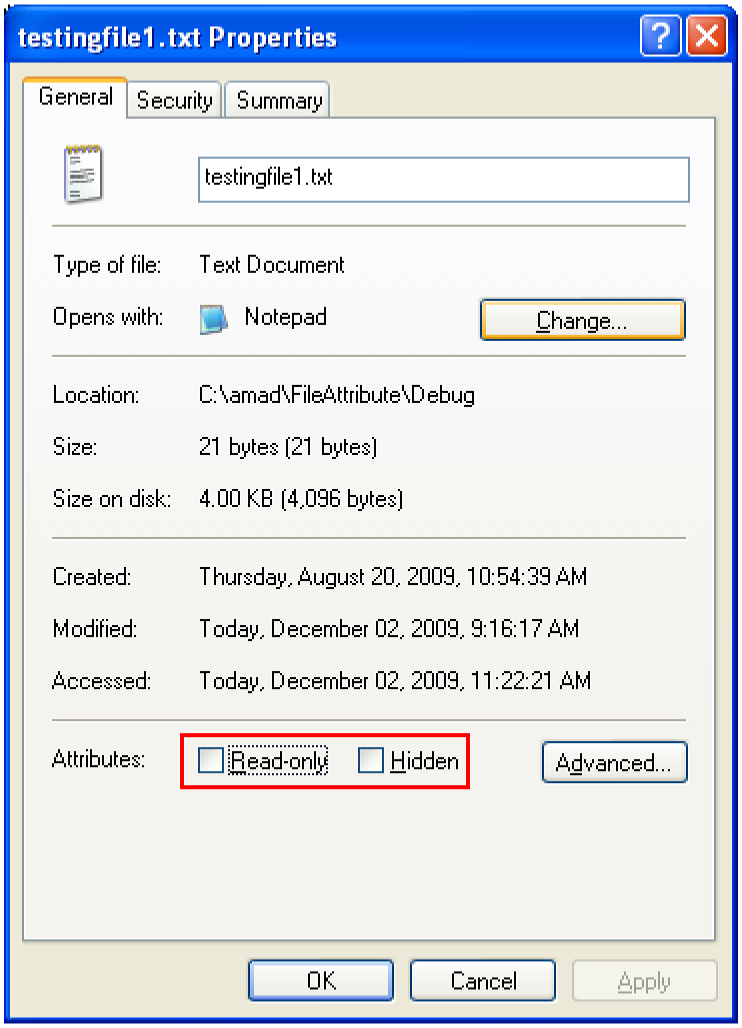
Firstly, let create a few text files under the project's Debug folder.



Name	Size	Type	Date Modified
FileAttribute.exe	32 KB	Application	12/2/2009 11:15 AM
FileAttribute.ilk	341 KB	Incremental Linker File	12/2/2009 11:15 AM
FileAttribute.pdb	475 KB	Program Debug Database	12/2/2009 11:15 AM
info.txt	1 KB	Text Document	12/2/2009 9:16 AM
simpletextfile.txt	1 KB	Text Document	12/2/2009 9:17 AM
testingfile1.txt	1 KB	Text Document	12/2/2009 9:16 AM
testingfile2.txt	1 KB	Text Document	12/2/2009 9:16 AM

4 objects selected | 93 bytes | My Computer

The default files' properties are shown below, at the beginning.



testingfile1.txt

Type of file: Text Document

Opens with: Notepad

Location: C:\amad\FileAttribute\Debug

Size: 21 bytes (21 bytes)

Size on disk: 4.00 KB (4,096 bytes)

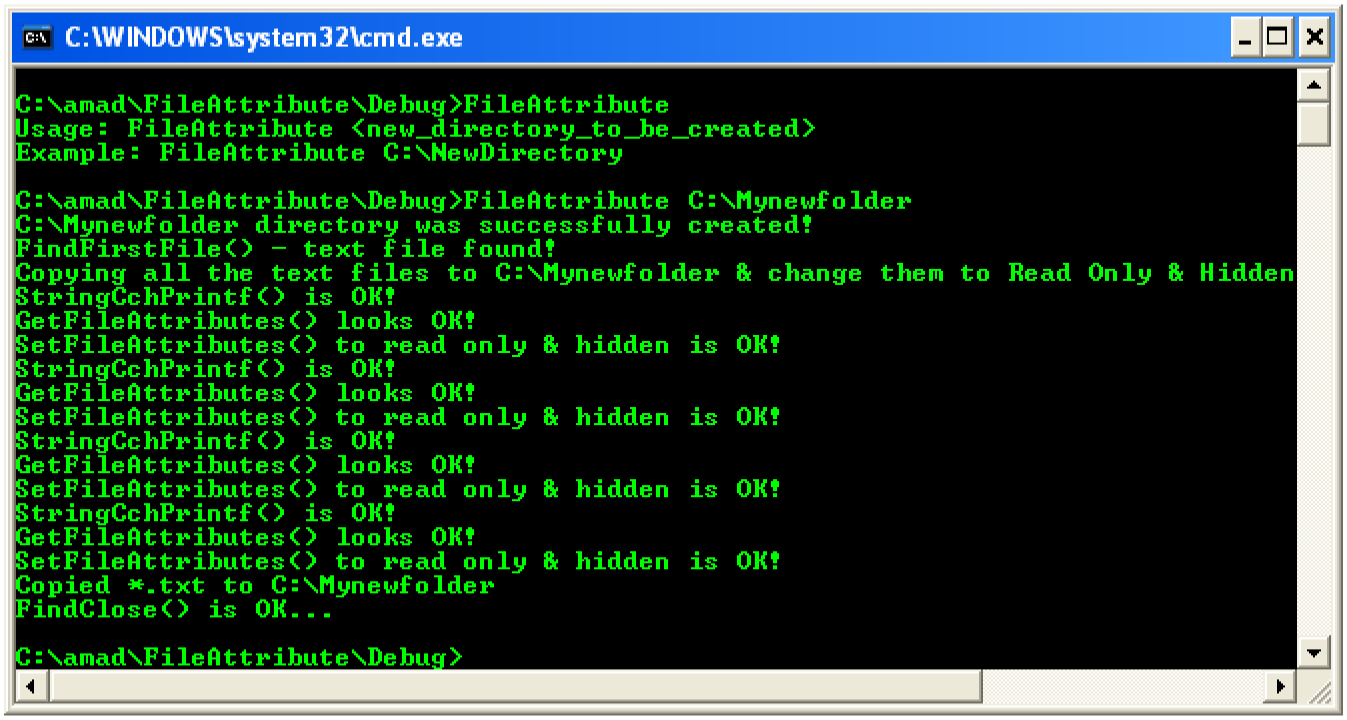
Created: Thursday, August 20, 2009, 10:54:39 AM

Modified: Today, December 02, 2009, 9:16:17 AM

Accessed: Today, December 02, 2009, 11:22:21 AM

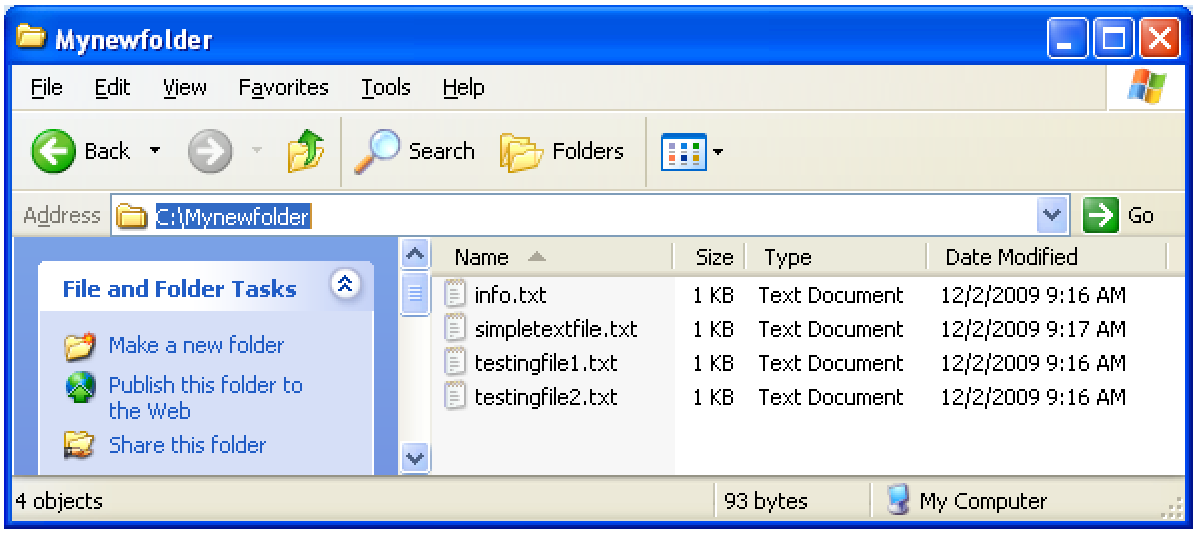
Attributes: ☐ Read-only ☐ Hidden

Next, run the program with a folder name as the argument.



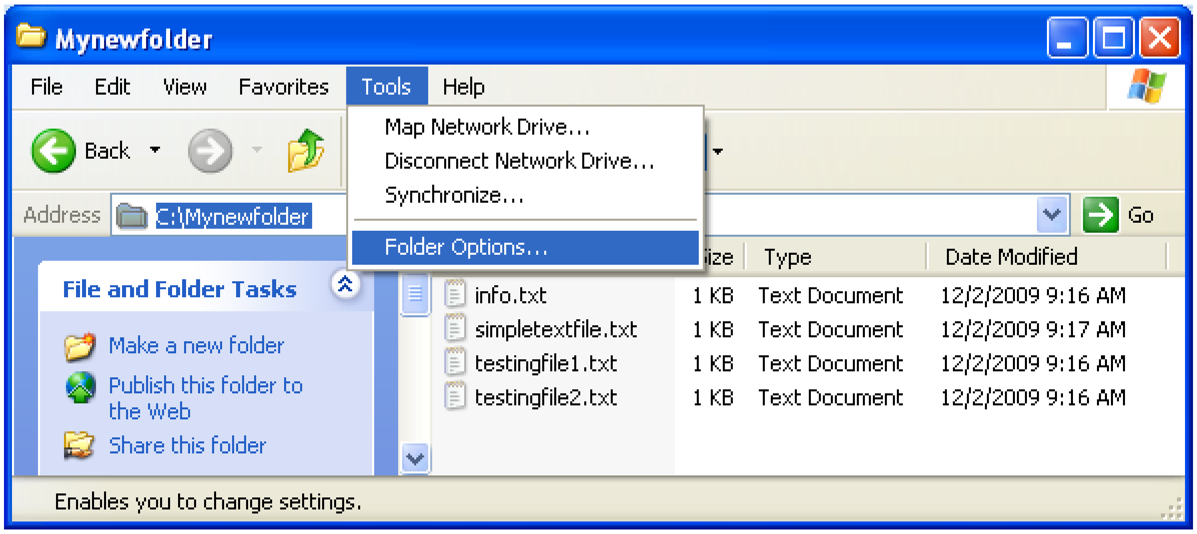
```
C:\amad\FileAttribute\Debug>FileAttribute
Usage: FileAttribute <new_directory_to_be_created>
Example: FileAttribute C:\NewDirectory

C:\amad\FileAttribute\Debug>FileAttribute C:\Mynewfolder
C:\Mynewfolder directory was successfully created!
FindFirstFile() - text file found!
Copying all the text files to C:\Mynewfolder & change them to Read Only & Hidden
StringCchPrintf() is OK!
GetFileAttributes() looks OK!
SetFileAttributes() to read only & hidden is OK!
StringCchPrintf() is OK!
GetFileAttributes() looks OK!
SetFileAttributes() to read only & hidden is OK!
StringCchPrintf() is OK!
GetFileAttributes() looks OK!
SetFileAttributes() to read only & hidden is OK!
StringCchPrintf() is OK!
GetFileAttributes() looks OK!
SetFileAttributes() to read only & hidden is OK!
Copied *.txt to C:\Mynewfolder
FindClose() is OK...

C:\amad\FileAttribute\Debug>
```

Then, verify that the new folder has been created, with all the text files have been moved to the new folder.



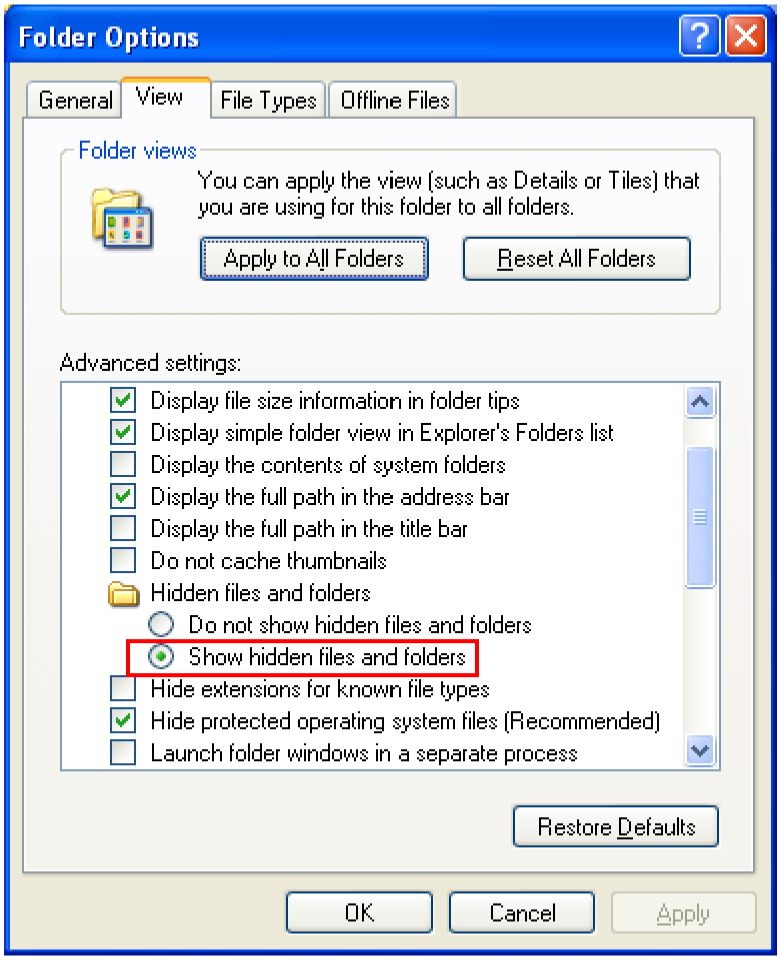
Name	Size	Type	Date Modified
info.txt	1 KB	Text Document	12/2/2009 9:16 AM
simpletextfile.txt	1 KB	Text Document	12/2/2009 9:17 AM
testingfile1.txt	1 KB	Text Document	12/2/2009 9:16 AM
testingfile2.txt	1 KB	Text Document	12/2/2009 9:16 AM

Those files' properties are hidden and read only so you may need to enable the Windows explorer to show the hidden files and folders. The steps are shown in the following Figures.



	Size	Type	Date Modified
info.txt	1 KB	Text Document	12/2/2009 9:16 AM
simpletextfile.txt	1 KB	Text Document	12/2/2009 9:17 AM
testingfile1.txt	1 KB	Text Document	12/2/2009 9:16 AM
testingfile2.txt	1 KB	Text Document	12/2/2009 9:16 AM

Enables you to change settings.

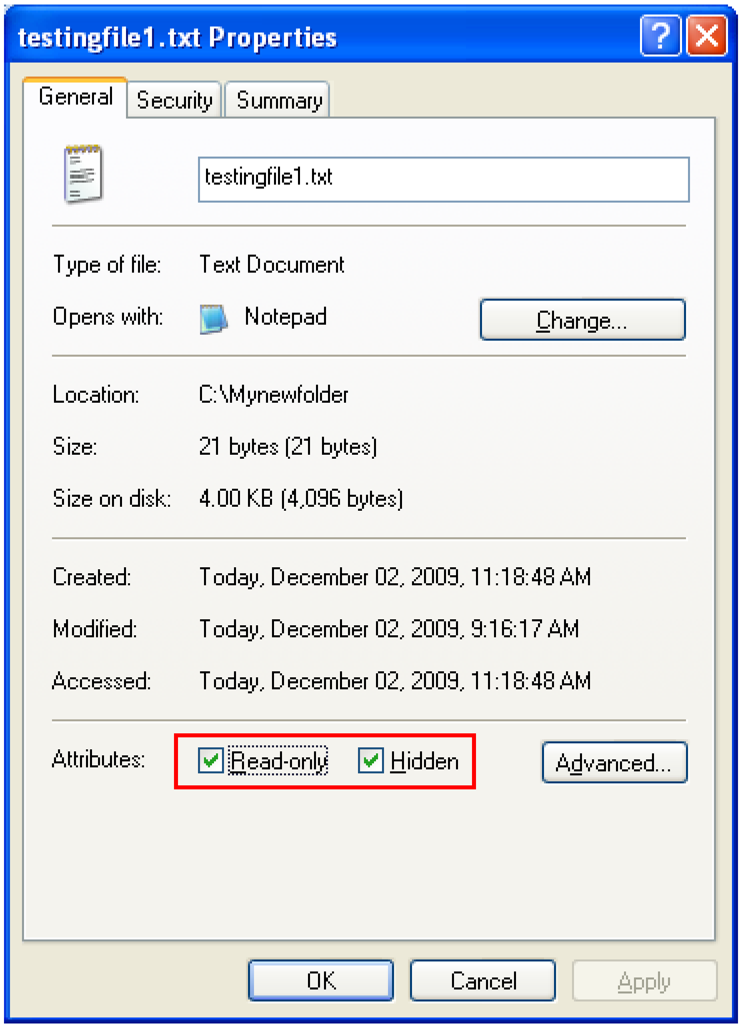


You can apply the view (such as Details or Tiles) that you are using for this folder to all folders.

Advanced settings:

- [✓] Display file size information in folder tips
- [✓] Display simple folder view in Explorer's Folders list
- [ ] Display the contents of system folders
- [✓] Display the full path in the address bar
- [ ] Display the full path in the title bar
- [ ] Do not cache thumbnails
- Hidden files and folders
  - Do not show hidden files and folders [✓ Show hidden files and folders]
- [ ] Hide extensions for known file types
- [✓] Hide protected operating system files (Recommended)
- [ ] Launch folder windows in a separate process

Then, verify the properties one of those files.



testingfile1.txt

Type of file: Text Document

Opens with: Notepad

Location: C:\Mynewfolder

Size: 21 bytes (21 bytes)

Size on disk: 4.00 KB (4,096 bytes)

Created: Today, December 02, 2009, 11:18:48 AM

Modified: Today, December 02, 2009, 9:16:17 AM

Accessed: Today, December 02, 2009, 11:18:48 AM

Attributes: [✓ Read-only] [✓ Hidden]

Testing for the End of a File Program Example

The ReadFile() function checks for the end-of-file condition (eof) differently for synchronous and asynchronous read operations. When a synchronous read operation gets to the end of a file, ReadFile() returns TRUE and sets the variable pointed to by the lpNumberOfBytesRead parameter to zero. An asynchronous read operation can encounter the end of a file during the initiating call to ReadFile() or during subsequent asynchronous operations if the file pointer is programmatically advanced beyond the end of the file. The following C++ code snippet shows how to test for the end of a file during a synchronous read operation.

```
// Attempt a synchronous read operation.
bResult = ReadFile(hFile, &inBuffer, nBytesToRead, &nBytesRead, NULL);

// Check for eof.
if (bResult &&  nBytesRead == 0)
{
    // at the end of the file
}
```

The test for end-of-file during an asynchronous read operation is slightly more involved than for a similar synchronous read operation. The end-of-file indicator for asynchronous read operations is when GetOverlappedResult() returns FALSE and GetLastError() returns ERROR_HANDLE_EOF. The following C++ example shows how to test for the end of file during an asynchronous read operation.

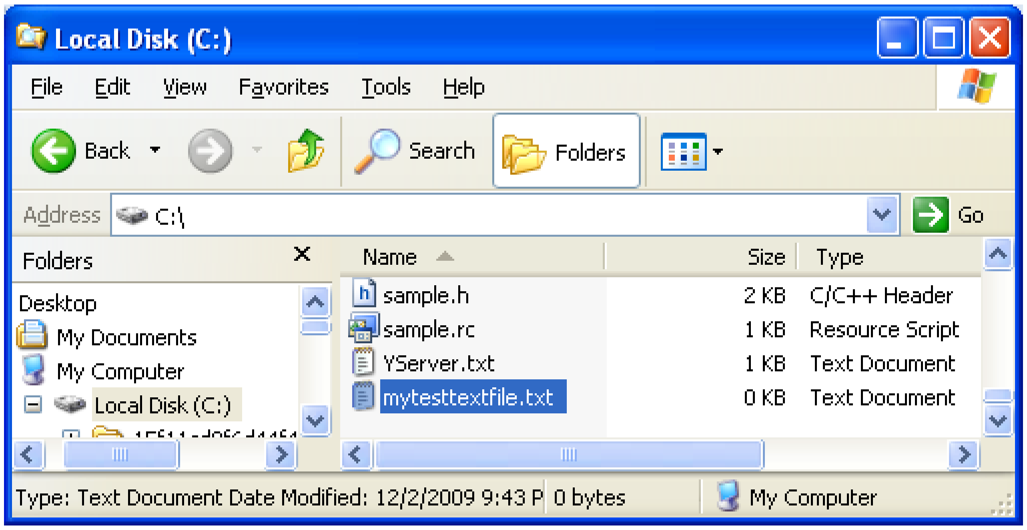
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.
Then, add the source file and give it a suitable name.
Next, add the following source code.

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

#define BUF_SIZE 100

// Routine Description: Retrieve the system error message for the last-error
code
LPCTSTR ErrorMessage(DWORD error)
{
      LPVOID lpMsgBuf;

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        error,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL );

    return((LPCTSTR)lpMsgBuf);
}

// Routine Description: Placeholder to demo when async I/O
// might want to do other processing.
void GoDoSomethingElse(void)
{
    wprintf(L"Inside GoDoSomethingElse()...\n");
}

// Routine Description:
//      Demonstrate async ReadFile() operations that can catch
//      End-of-file conditions. Unless the operation completes
//      synchronously or the file size happens to be an exact
//      multiple of BUF_SIZE, this routine will eventually force
//      an EOF condition on any file.

// Parameters:
//      hEvent - pre-made manual-reset event.
//      hFile - pre-opened file handle, overlapped.
//      inBuffer - the buffer to read in the data to.
//      nBytesToRead - how much to read (usually the buffer size).
// Return Value: Number of bytes read.
DWORD AsyncTestForEnd(HANDLE hEvent, HANDLE hFile)
```

```
{
    WCHAR inBuffer[BUF_SIZE];
    DWORD nBytesToRead      = BUF_SIZE;
    DWORD dwBytesRead       = 0;
    DWORD dwFileSize        = GetFileSize(hFile, NULL);
    OVERLAPPED stOverlapped = {0};
    DWORD dwError  = 0;
    LPCTSTR errMsg = NULL;
    BOOL bResult   = FALSE;
    BOOL bContinue = TRUE;

    // Set up overlapped structure event. Other members are already
    // initialized to zero.
    stOverlapped.hEvent = hEvent;

    // This is an intentionally brute-force loop to force the EOF trigger.
    // A properly designed loop for this simple file read would use the
    // GetFileSize() API to regulate execution. However, the purpose here
    // is to demonstrate how to trigger the EOF error and handle it.
    while(bContinue)
    {
        // Default to ending the loop.
        bContinue = FALSE;

        // Attempt an asynchronous read operation.
            wprintf(L"Asynchronous read attempt...\n");
        bResult =
ReadFile(hFile,inBuffer,nBytesToRead,&dwBytesRead,&stOverlapped);

        dwError = GetLastError();

        // Check for a problem or pending operation.
        if (!bResult)
        {
                  // If the function fails, or is completing asynchronously
                  // the return value is zero (FALSE)
                  wprintf(L"ReadFile() returns %u. Failed or asynchronous
completion...\n", bResult);
            switch (dwError)
            {
                  case ERROR_HANDLE_EOF:
                        {
                              // Another check
                    wprintf(L"\nReadFile() returned FALSE and EOF condition,
async EOF not triggered.\n");
                    break;
                }
                case ERROR_IO_PENDING:
                {
                    BOOL bPending=TRUE;

                    // Loop until the I/O is complete, that is: the overlapped
                    // event is signaled.
                    while(bPending)
                    {
                        bPending = FALSE;

                        // Pending asynchronous I/O, do something else
                        // and re-check overlapped structure.
                        wprintf(L"\nReadFile() operation is pending...\n");

                        // Do something else then come back to check.
                        GoDoSomethingElse();

                        // Check the result of the asynchronous read
                        // without waiting (forth parameter FALSE).
                        bResult =
GetOverlappedResult(hFile,&stOverlapped,&dwBytesRead,FALSE) ;

                        if (!bResult)
                        {
                            switch (dwError = GetLastError())
                            {
                                case ERROR_HANDLE_EOF:
                                {
                                    // Handle an end of file
                                    wprintf(L"GetOverlappedResult() found
EOF\n");

                                    break;
                                }

                                case ERROR_IO_INCOMPLETE:
                                {
                                    // Operation is still pending, allow while
loop

                                    // to loop again after printing a little
progress.

                                    wprintf(L"GetOverlappedResult() I/O
Incomplete\n");

                                    bPending = TRUE;
                                    bContinue = TRUE;
                                    break;
                                }

                                default:
                                {
                                    // Decode any other errors codes.
                                    errMsg = ErrorMessage(dwError);
                                    wprintf(L"GetOverlappedResult() failed,
error %d: %s\n", dwError, errMsg);
                                    LocalFree((LPVOID)errMsg);
                                }
                            }
                        }
                        else
                        {
                                        wprintf(L"GetOverlappedResult() is
pretty fine!\n");
                            wprintf(L"ReadFile() operation completed\n");

                            // Manual-reset event should be reset since it is
now signaled.
                            if(ResetEvent(stOverlapped.hEvent) != 0)
                                            wprintf(L"ResetEvent() is
OK!\n");
                                        else
                                            wprintf(L"ResetEvent() failed!
Error %u\n", GetLastError());
```

```
                        }
                    }
                    break;
                }

                default:
                {
                    // Decode any other errors codes.
                    errMsg = ErrorMessage(dwError);
                    wprintf(L"ReadFile() GetLastError() unhandled, error %d: 
%s\n", dwError, errMsg);
                    LocalFree((LPVOID)errMsg);
                    break;
                }
            }
        }
        else
        {
            // EOF demo did not trigger for the given file.
            // Note that system caching may cause this condition on most files
            // after the first read. CreateFile() can be called using the
            // FILE_FLAG_NOBUFFERING parameter but it would require reads are
            // always aligned to the volume's sector boundary. This is beyond
            // the scope of this example. See comments in the main() function.
                  wprintf(L"ReadFile() returns %u, completed synchronously\n", 
bResult);
        }

        // The following operation assumes the file is not extremely large, 
otherwise
        // logic would need to be included to adequately account for very large
        // files and manipulate the OffsetHigh member of the OVERLAPPED 
structure.
        stOverlapped.Offset += dwBytesRead;
        if (stOverlapped.Offset < dwFileSize)
            bContinue = TRUE;
    }

    return stOverlapped.Offset;
}

// To force an EOF condition, execute this application specifying a
// zero-length file. This is because the offset (file pointer) must be
// at or beyond the end-of-file marker when ReadFile() is called.
int wmain(int argc, WCHAR *argv[])
{
    HANDLE hEvent;
    HANDLE hFile;
    DWORD dwReturnValue;
      DWORD dwError;
      LPCTSTR errMsg;

    if( argc != 2 )
    {
        wprintf(L"No arguments supplied!\n");
            wprintf(L"Usage: %s <file_name_to_be_read>\n", argv[0]);
            wprintf(L"Example: %s C:\\mytestfile.txt\n", argv[0]);
            wprintf(L"         The file must exist\n");
        return 1;
    }

    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_FLAG_OVERLAPPED,     // overlapped operation
                       NULL);                    // no attr. template

    if (hFile == INVALID_HANDLE_VALUE)
    {
        dwError = GetLastError();
        errMsg = ErrorMessage(dwError);
        wprintf(L"Could not open file, error %d: %s\n", dwError, errMsg);
        LocalFree((LPVOID)errMsg);
        return 1;
    }
      else
            wprintf(L"CreateFile() is pretty fine!\n");

    hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

    if (hEvent == NULL)
    {
        dwError = GetLastError();
        errMsg = ErrorMessage(dwError);
        wprintf(L"Could not create event: %d %s\n", dwError, errMsg);
        LocalFree((LPVOID)errMsg);
        return 1;
    }
      else
            wprintf(L"CreateEvent() is  working!\n");

    dwReturnValue = AsyncTestForEnd(hEvent, hFile);
    wprintf(L"Read complete. Bytes read: %d\n", dwReturnValue);

    if(CloseHandle(hFile) != 0)
            wprintf(L"hFile handle was closed!\n");
      else
            wprintf(L"Failed to close hFile handle!\n");

    if(CloseHandle(hEvent) != 0)
            wprintf(L"hEvent handle was closed!\n");
      else
            wprintf(L"Failed to close hEvent handle!\n");

      return 0;
}
```

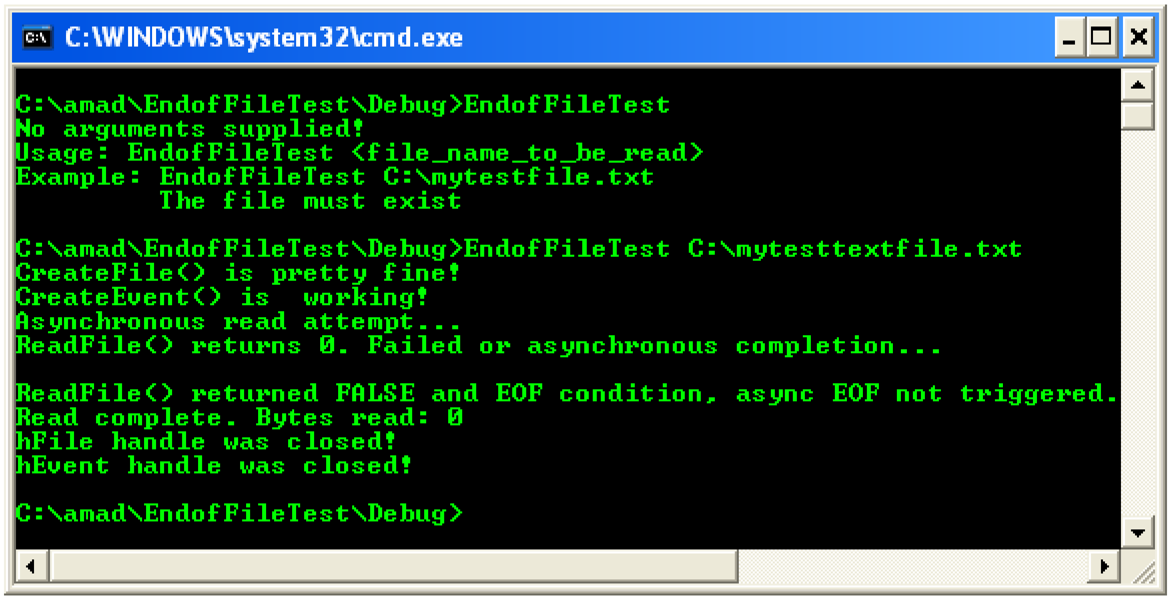
To test this program, firstly, create an empty text file on C drive. In this case, mytesttextfile.txt.



Name	Size	Type
sample.h	2 KB	C/C++ Header
sample.rc	1 KB	Resource Script
YServer.txt	1 KB	Text Document
mytesttextfile.txt	0 KB	Text Document

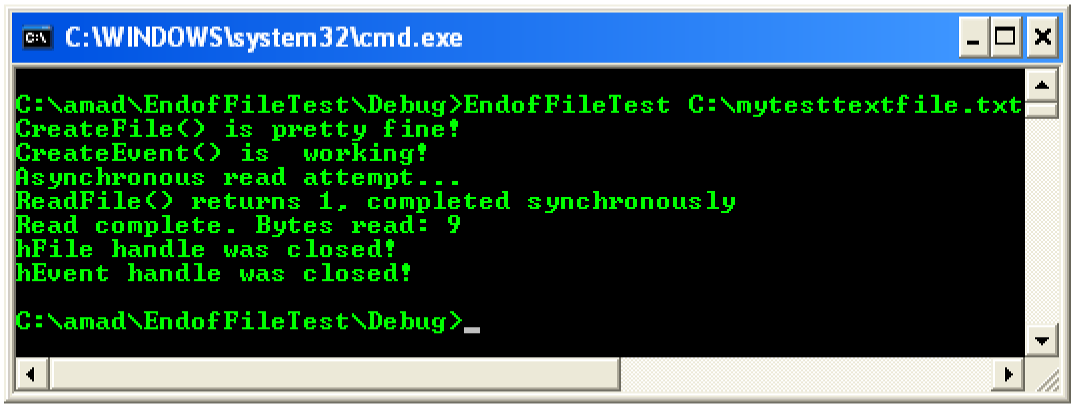
Type: Text Document Date Modified: 12/2/2009 9:43 P 0 bytes

Build and run the project with the empty file name as the argument. The following Figure shows the sample output.



```
C:\amad\EndofFileTest\Debug>EndofFileTest
No arguments supplied!
Usage: EndofFileTest <file_name_to_be_read>
Example: EndofFileTest C:\mytestfile.txt
         The file must exist

C:\amad\EndofFileTest\Debug>EndofFileTest C:\mytesttextfile.txt
CreateFile() is pretty fine!
CreateEvent() is  working!
Asynchronous read attempt...
ReadFile() returns 0. Failed or asynchronous completion...

ReadFile() returned FALSE and EOF condition, async EOF not triggered.
Read complete. Bytes read: 0
hFile handle was closed!
hEvent handle was closed!

C:\amad\EndofFileTest\Debug>
```

Then, put some text in the text file and save it. Re-run the project. The following screenshot shows the sample output.



```
C:\amad\EndofFileTest\Debug>EndofFileTest C:\mytesttextfile.txt
CreateFile() is pretty fine!
CreateEvent() is  working!
Asynchronous read attempt...
ReadFile() returns 1, completed synchronously
Read complete. Bytes read: 9
hFile handle was closed!
hEvent handle was closed!

C:\amad\EndofFileTest\Debug>_
```

Using Streams Example

The example in this topic demonstrates how to use basic NTFS file system streams. This example creates a file, called "anothertestfile," with a size of 46 bytes. However, the file also has a file stream, called "stream," which adds an additional 24 bytes that is not reported by the operating system. Therefore, when you view the file-size property for the file, you see only the size of the file testfile.

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 <strsafe.h>

// A prototype that receives a function name, displaying
// system error code and its respective message
void DisplayErrorBox(LPTSTR lpszFunction);

int wmain(int argc, WCHAR *argv[])
{
    HANDLE hFile, hStream;
    DWORD dwRet;

    hFile = CreateFile(L"anothertestfile",
                        GENERIC_WRITE,
                        FILE_SHARE_WRITE,
                        NULL,
                        OPEN_ALWAYS,
                        0,
                        NULL );

      if(hFile == INVALID_HANDLE_VALUE )
    {
       DisplayErrorBox(L"CreateFile()");
       return 1;
    }
    else
      {
          wprintf(L"CreateFile() should be fine!\n");
          wprintf(L"Writing some bytes to anothertestfile...\n");
```

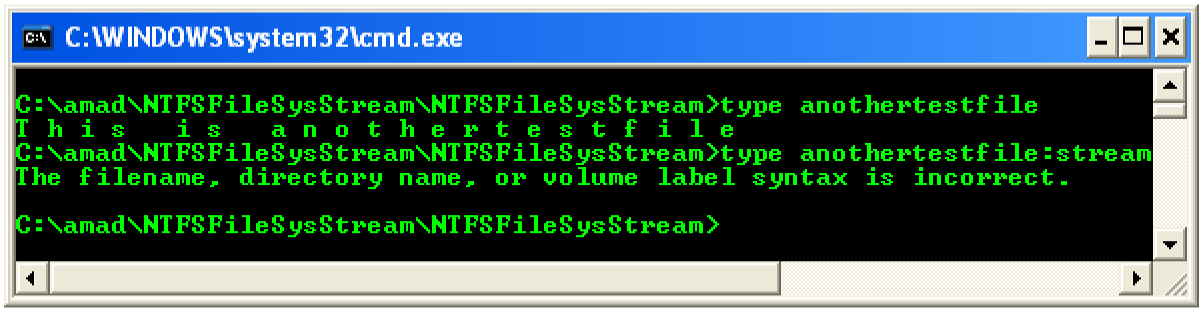
```
            if(WriteFile(hFile, L"This is anothertestfile", 46, &dwRet, NULL )
!= 0)
                  wprintf(L"WriteFile() is fine!\n");
            else
                  DisplayErrorBox(L"WriteFile()");
      }

      hStream = CreateFile(L"anothertestfile:stream",
                              GENERIC_WRITE,
                              FILE_SHARE_WRITE,
                              NULL,
                              OPEN_ALWAYS,
                              0,
                              NULL );

      if(hStream == INVALID_HANDLE_VALUE)
        DisplayErrorBox(L"CreateFile()");
    else
      {
            printf("CreateFile() should be fine!\n");
            printf("Writing some stream to the file...\n");

            // Take note the 3rd parameter which must be ample for
Unicode/multibyte
        if(WriteFile(hStream, L"This is anothertestfile:stream", 60, &dwRet,
NULL) != 0)
                  wprintf(L"WriteFile() is fine!\n");
            else
                  DisplayErrorBox(L"WriteFile()");
      }
      return 0;
}

void DisplayErrorBox(LPTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
            FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL);

    // Display the error message and clean up
    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,

(lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(WCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(WCHAR), L"%s failed with error %d: %s",
        lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, L"Error", MB_OK);
    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}
```

If you type `type anothertestfile` at a command prompt, it displays the following output:

```
T h i s  i s  a n o t h e r t e s t f i l e
```

However, if you type the words `type testfile:stream`, it generates the following error:

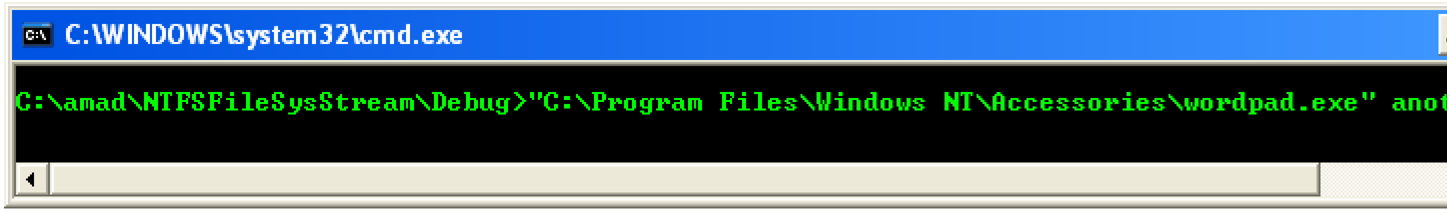
"The filename, directory name, or volume label syntax is incorrect."



```
C:\amad\NTFSFileSysStream\NTFSFileSysStream>type anothertestfile
T h i s   i s   a n o t h e r t e s t f i l e
C:\amad\NTFSFileSysStream\NTFSFileSysStream>type anothertestfile:stream
The filename, directory name, or volume label syntax is incorrect.

C:\amad\NTFSFileSysStream\NTFSFileSysStream>
```

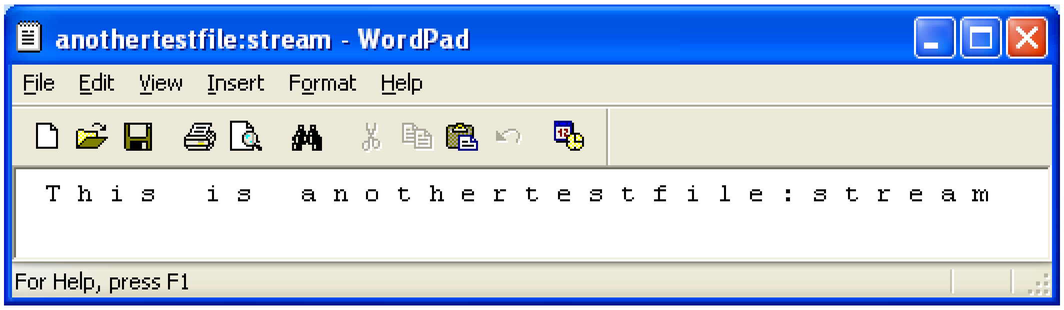
To view what is in `testfile:stream`, use Wordpad by entering the following command:

```
"C:\Program Files\Windows NT\Accessories\wordpad.exe"
anothertestfile:stream
```



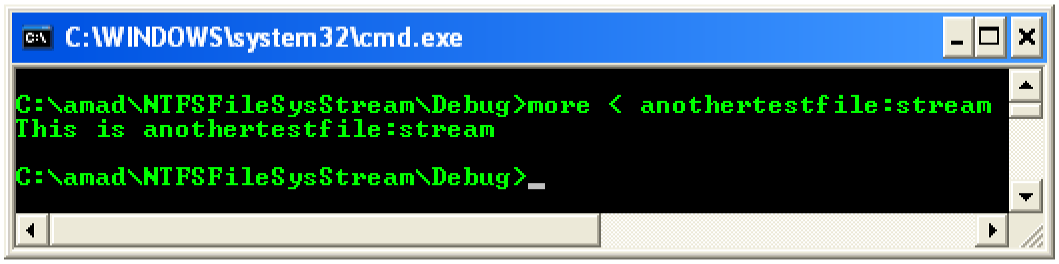
```
C:\amad\NTFSFileSysStream\Debug>"C:\Program Files\Windows NT\Accessories\wordpad.exe" anot
```

The text displayed in Wordpad is as follows.


anothertestfile:stream - WordPad
File Edit View Insert Format Help
For Help, press F1

```
T h i s   i s   a n o t h e r t e s t f i l e : s t r e a m
```

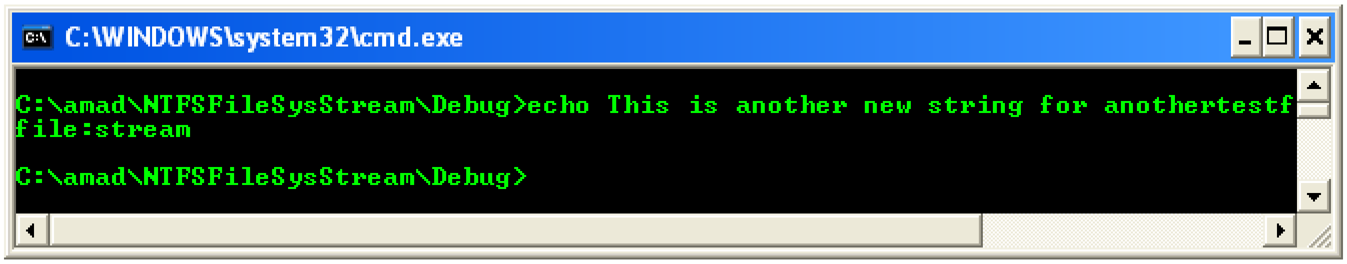
Or you can use the `more` command.



```
C:\amad\NTFSFileSysStream\Debug>more < anothertestfile:stream
This is anothertestfile:stream

C:\amad\NTFSFileSysStream\Debug>_
```

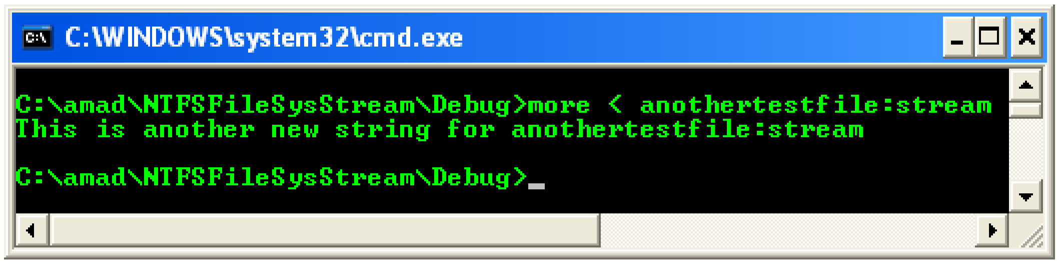
To edit the file content, we can use the `echo` command. For example:

```
echo This is another new string for anothertestfile:stream >
anothertestfile:stream
```



```
C:\amad\NTFSFileSysStream\Debug>echo This is another new string for anothertestf
file:stream

C:\amad\NTFSFileSysStream\Debug>
```

The following is the sample output for the edited file stream when verified using `more` command.



```
C:\amad\NTFSFileSysStream\Debug>more < anothertestfile:stream
This is another new string for anothertestfile:stream

C:\amad\NTFSFileSysStream\Debug>_
```

Alternate File Stream

From user point of view, tools and story for NTFS Alternate Stream please refer to the following link: NTFS Alternate File Stream – What, When and How-to.

File Management Reference

File Management Control Codes

The following control codes are used in file management.

Control code	Operation
FSCTL_ALLOW_EXTENDED_DASD_IO	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.
FSCTL_FILESYSTEM_GET_STATISTICS	Retrieves statistics from the file system.

FSCTL_FIND_FILES_BY_SID	Searches a directory for a file whose creator owner matches the specified SID.
FSCTL_GET_NTFS_FILE_RECORD	Retrieves the complete NTFS file system file record identified by the specified file identifier.
FSCTL_RECALL_FILE	Recalls a file from storage media managed by Remote Storage, the hierarchical storage management software.
FSCTL_SET_ZERO_ON_DEALLOCATION	Indicates an NTFS file system file handle should have its clusters filled with zeros when it is deallocated.

The following control codes are used with file compression and decompression.

Control code	Operation
FSCTL_GET_COMPRESSION	Obtains the compression state of a file or directory.
FSCTL_SET_COMPRESSION	Sets the compression state of a file or directory.

The following control codes are used with object identifiers.

Control code	Operation
FSCTL_CREATE_OR_GET_OBJECT_ID	Creates or retrieves the object identifier for the specified file or directory.
FSCTL_DELETE_OBJECT_ID	Removes the object identifier from the specified file or directory.
FSCTL_GET_OBJECT_ID	Retrieves the object identifier for the specified file or directory.
FSCTL_SET_OBJECT_ID	Sets the object identifier for the specified file or directory.
FSCTL_SET_OBJECT_ID_EXTENDED	Modifies user data associated with the object identifier of the specified file or directory.

The following control codes are used with opportunistic locks.

Control code	Operation
FSCTL_OPBATCH_ACK_CLOSE_PENDING	Notifies a server that a client application is about to close a file. An application uses this operation following notification that an opportunistic lock on the file is about to be broken.
FSCTL_OPLOCK_BREAK_ACK_NO_2	Responds to notification that an opportunistic lock on a file is about to be broken. An application uses this operation to unlock all opportunistic locks on the file but keep the file open.
FSCTL_OPLOCK_BREAK_ACKNOWLEDGE	Responds to notification that an exclusive opportunistic lock on a file is about to be broken. An application uses this operation to indicate that the file should receive a level 2 opportunistic

	lock.
FSCTL_OPLOCK_BREAK_NOTIFY	Allows the calling application to wait for completion of an opportunistic lock break.
FSCTL_REQUEST_BATCH_OPLOCK	Requests a batch opportunistic lock on a file.
FSCTL_REQUEST_FILTER_OPLOCK	Requests a filter opportunistic lock on a file.
FSCTL_REQUEST_OPLOCK_LEVEL_1	Requests a level 1 opportunistic lock on a file.
FSCTL_REQUEST_OPLOCK_LEVEL_2	Requests a level 2 opportunistic lock on a file.

The following control codes are used with sparse files.

Control code	Operation
FSCTL_QUERY_ALLOCATED_RANGES	Scans a file for ranges of the file for which disk space is allocated.
FSCTL_SET_SPARSE	Marks a file as a sparse file.
FSCTL_SET_ZERO_DATA	Sets a range of a files bytes to zeroes.

The following control codes are used with the NTFS self-healing mechanism.

Control code	Operation
FSCTL_GET_REPAIR	Retrieves information about the NTFS file system's self-healing mechanism.
FSCTL_INITIATE_REPAIR	Triggers the NTFS file system to start a self-healing cycle on a single file.
FSCTL_SET_REPAIR	Sets the mode of an NTFS file system's self-healing capability.
FSCTL_WAIT_FOR_REPAIR	Returns when the specified repairs are completed.

The following control codes are used with UDF.

Control code	Operation
FSCTL_MAKE_MEDIA_COMPATIBLE	Closes an open UDF session on write-once media to make the media ROM compatible. Used in conjunction with FILE_SEQUENTIAL_WRITE_ONCE volume flag. This call must be issued on the volume handle.
FSCTL_QUERY_ON_DISK_VOLUME_INFO	Requests UDF-specific volume information.
FSCTL_QUERY_SPARING_INFO	Retrieves the defect management properties of the volume. Used for UDF file systems.
FSCTL_SET_DEFECT_MANAGEMENT	Sets the software defect management state for the specified file. Used for UDF file systems.

File Management Enumerations

The following enumerations are used in file management:

1. FILE_ID_TYPE - Discriminator for the union in the FILE_ID_DESCRIPTOR structure.

2. FILE_INFO_BY_HANDLE_CLASS - Identifies the type of file information that GetFileInformationByHandleEx() should retrieve or SetFileInformationByHandle() should set.
3. FINDEX_INFO_LEVELS - Defines values that are used with the FindFirstFileEx() function to specify the information level of the returned data.
4. FINDEX_SEARCH_OPS - Defines values that are used with the FindFirstFileEx() function to specify the type of filtering to perform.
5. GET_FILEEX_INFO_LEVELS - Defines values that are used with the GetFileAttributesTransacted() function to specify the information level of the returned data.
6. PRIORITY_HINT - Defines values that are used with the FILE_IO_PRIORITY_HINT_INFO structure to specify the priority hint for a file I/O operation.
7. STREAM_INFO_LEVELS - Defines values that are used with the FindFirstStreamW() function to specify the information level of the returned data.

File Management Functions

The following functions are used to manage files.

Function	Description
AreFileApisANSI()	Determines whether the file I/O functions are using the ANSI or OEM character set code page.
CheckNameLegalDOS8Dot3()	Determines whether a specified name can be used to create a file on a FAT file system.
CloseHandle()	Closes an open object handle.
CopyFile()	Copies an existing file to a new file.
CopyFileEx()	Copies an existing file to a new file, and notifies an application of the progress through a callback function.
CopyFileTransacted()	Copies an existing file to a new file as a transacted operation, notifying the application of its progress through a callback function.
CreateFile()	Creates or opens a file, directory, physical disk, volume, console buffer, tape drive, communications resource, mailslot, or named pipe.
CreateFileTransacted()	Creates or opens a file, file stream, or directory as a transacted operation.
CreateHardLink()	Establishes a hard link between an existing file and a new file.
CreateHardLinkTransacted()	Establishes a hard link between an existing file and a new file as a transacted operation.
CreateSymbolicLink()	Creates a symbolic link.
CreateSymbolicLinkTransacted()	Creates a symbolic link as a transacted operation.
DeleteFile()	Deletes an existing file.
DeleteFileTransacted()	Deletes an existing file as a transacted operation.
FindClose()	Closes a file search handle that the FindFirstFile(), FindFirstFileEx(), or FindFirstStreamW() function opens.
FindFirstFile()	Searches a directory for a file or subdirectory name that matches a specified name.
FindFirstFileEx()	Searches a directory for a file or subdirectory name and

	attributes that match those that are specified.
FindFirstFileNameTransactedW()	Creates an enumeration of all the hard links to the specified file as a transacted operation. The function returns a handle to the enumeration that can be used on subsequent calls to the FindNextFileNameW() function.
FindFirstFileNameW()	Creates an enumeration of all the hard links to the specified file. The FindFirstFileNameW() function returns a handle to the enumeration that can be used on subsequent calls to the FindNextFileNameW() function.
FindFirstFileTransacted()	Searches a directory for a file or subdirectory with a name that matches a specific name as a transacted operation.
FindFirstStreamTransactedW()	Enumerates the first stream in the specified file or directory as a transacted operation.
FindFirstStreamW()	Enumerates the first stream in a specified file or directory.
FindNextFile()	Continues a file search.
FindNextFileNameW()	Continues enumerating the hard links to a file using the handle returned by a successful call to the FindFirstFileNameW() function.
FindNextStreamW()	Continues a stream search.
GetBinaryType()	Determines whether a file is an executable (.exe) file, and if so, which subsystem runs the executable file.
GetCompressedFileSize()	Retrieves the actual number of disk storage bytes that are used to store a specified file.
GetCompressedFileSizeTransacted()	Retrieves the actual number of bytes of disk storage used to store a specified file as a transacted operation.
GetFileAttributes()	Retrieves file system attributes for a specified file or directory.
GetFileAttributesEx()	Retrieves attributes for a specified file or directory.
GetFileAttributesTransacted()	Retrieves attributes for a specified file or directory as a transacted operation.
GetFileBandwidthReservation()	Retrieves the bandwidth reservation properties of the volume on which the specified file resides.
GetFileInformationByHandle()	Retrieves file information for a specified file.
GetFileInformationByHandleEx()	Retrieves file information for the specified file.
GetFileSize()	Retrieves the size of a specified file, in bytes. The file size that can be reported by this function is limited to a DWORD value.
GetFileSizeEx()	Retrieves the size of a specified file.
GetFileType()	Retrieves the file type of a specified file.
GetFinalPathNameByHandle()	Retrieves the final path for the specified file.
GetFullPathName()	Retrieves the full path and file name of a specified file.
GetFullPathNameTransacted()	Retrieves the full path and file name of a specified file as a transacted operation.
GetLongPathName()	Converts a specified path to its long form.
GetLongPathNameTransacted()	Converts the specified path to its long form as a transacted operation.
GetShortPathName()	Retrieves the short path form of a specified path.

GetTempFileName()	Creates a name for a temporary file.
GetTempPath()	Retrieves the path of the directory that is designated for temporary files.
MoveFile()	Moves an existing file or directory and its children.
MoveFileEx()	Moves an existing file or directory.
MoveFileTransacted()	Moves an existing file or a directory, including its children, as a transacted operation.
MoveFileWithProgress()	Moves a file or directory. You can provide a callback function that receives progress notifications.
OpenFile()	Creates, opens, reopens, or deletes a file.
OpenFileById()	Opens the file that matches the specified identifier.
ReOpenFile()	Reopens a specified file system object with different access rights, a different sharing mode, and different flags than it was previously opened with.
ReplaceFile()	Replaces one file with a different file, and optionally creates a backup copy of the original file.
RtlIsNameLegalDOS8Dot3()	Determines whether or not a specified name can be used to create a file on the FAT file system.
SearchPath()	Searches for a specified file in a specified path.
SetFileApisToANSI()	Indicates that the file I/O functions must use the ANSI character set code page.
SetFileApisToOEM()	Causes the file I/O functions to use the OEM character set code page.
SetFileAttributes()	Sets the attributes of a file.
SetFileAttributesTransacted()	Sets the attributes for a file or directory as a transacted operation.
SetFileBandwidthReservation()	Requests that bandwidth for the specified file stream be reserved.
SetFileInformationByHandle()	Sets the information for the specified file.
SetFileShortName()	Sets the short name for a specified file.
SetFileValidData()	Sets the valid data length of a specified file.
SetSearchPathMode()	Sets the search mode used by the SearchPath function for the process.

The following functions are used with file I/O.

Function	Description
CancelIo()	Cancels all pending I/O operations that are issued by the calling thread for a specified file handle.
CancelIoEx()	Marks all pending I/O operations for the specified file handle in the current process as canceled, regardless of which thread created the I/O operation.
CancelSynchronousIo()	Marks pending synchronous I/O operations that are issued by the specified thread as canceled.
CreateIoCompletionPort()	Associates an I/O completion port with one or more file handles, or creates an I/O completion port that is not associated with a file handle.

FlushFileBuffers()	Flushes the buffers for a specified file, and causes all buffered data to be written to the file.
GetQueuedCompletionStatus()	Attempts to dequeue an I/O completion packet from a specified I/O completion port.
GetQueuedCompletionStatusEx()	Retrieves multiple completion port entries simultaneously.
LockFile()	Locks a specified file for exclusive access by the calling process.
LockFileEx()	Locks a specified file for exclusive access by the calling process. This function can operate synchronously or asynchronously.
PostQueuedCompletionStatus()	Posts an I/O completion packet to an I/O completion port.
ReadFile()	Reads data from a file, starting at the position that is indicated by a file pointer. This function can operate synchronously and asynchronously.
ReadFileEx()	Reads data from a file asynchronously.
ReadFileScatter()	Reads data from a file and stores it in an array of buffers.
SetEndOfFile()	Moves the end-of-file position for a specified file to the current position of a file pointer.
SetFileCompletionNotificationModes()	Sets the notification modes for a file handle.
SetFileIoOverlappedRange()	Associates a virtual address range with a file handle.
SetFilePointer()	Moves the file pointer of an open file.
SetFilePointerEx()	Moves the file pointer of a specified file.
UnlockFile()	Unlocks a region in an open file.
UnlockFileEx()	Unlocks a region in an open file. This function can operate synchronously or asynchronously.
WriteFile()	Writes data to a file at a position that a file pointer specifies. This function can operate synchronously and asynchronously.
WriteFileEx()	Writes data to a file. This function reports the completion status asynchronously by calling a specified completion routine when writing is completed or canceled and when the calling thread is in an alertable wait state.
WriteFileGather()	Retrieves data from an array of buffers, and then writes the data to a file.

The following functions are used with the encrypted file system.

Function	Description
AddUsersToEncryptedFile()	Adds user keys to a specified encrypted file.
CloseEncryptedFileRaw()	Closes an encrypted file after a backup or restore operation, and frees the associated system resources.
DecryptFile()	Decrypts an encrypted file or directory.
DuplicateEncryptionInfoFile()	Copies the EFS metadata from one file or directory to another.
EncryptFile()	Encrypts a file or directory.
EncryptionDisable()	Disables or enables encryption of a specified directory and the files in the directory.

FileEncryptionStatus()	Retrieves the encryption status of a specified file.
FreeEncryptionCertificateHashList()	Frees a certificate hash list.
OpenEncryptedFileRaw()	Opens an encrypted file to backup (export) or restore (import) the file.
QueryRecoveryAgentsOnEncryptedFile()	Retrieves a list of recovery agents for a specified file.
QueryUsersOnEncryptedFile()	Retrieves a list of users for a specified file.
ReadEncryptedFileRaw()	Backs up (exports) encrypted files.
RemoveUsersFromEncryptedFile()	Removes specified certificate hashes from a specified file.
SetUserFileEncryptionKey()	Sets a current user key to a specified certificate.
WriteEncryptedFileRaw()	Restores (imports) encrypted files.

The following functions are used with the file system redirector.

Function	Description
Wow64DisableWow64FsRedirection()	Disables file system redirection for the calling thread.
Wow64EnableWow64FsRedirection()	Enables or disables file system redirection for the calling thread.
Wow64RevertWow64FsRedirection()	Restores file system redirection for the calling thread.

The following functions are used to decompress files that are compressed by the Lempel-Ziv algorithm.

Function	Description
GetExpandedName()	Retrieves the original name of a compressed file, only if the file is compressed by the Lempel-Ziv algorithm.
LZClose()	Closes a file that was opened by using the LZOpenFile() function.
LZCopy()	Copies a source file to a destination file.
LZInit()	Allocates memory for the internal data structures that are required to decompress files, and then creates and initializes the files.
LZOpenFile()	Creates, opens, reopens, or deletes a specified file.
LZRead()	Reads a specified number of bytes from a file and copies them into a buffer.
LZSeek()	Moves a file pointer a specified number of bytes from a starting position.

The following callback functions are used in file I/O.

Function	Description
CopyProgressRoutine()	Callback function used with the CopyFileEx() and MoveFileWithProgress() functions, called when a portion of a copy or move operation is completed.
ExportCallback()	Callback function used with ReadEncryptedFileRaw(), called one or more times, each time with a block of the encrypted file's data, until it has received all of the file data.
FileIOCompletionRoutine()	Callback function used with the ReadFileEx() and WriteFileEx() functions, called when the asynchronous input and output (I/O) operation is completed or canceled.
ImportCallback()	Callback function used with WriteEncryptedFileRaw(), called one or

	more times, each time to retrieve a portion of a backup file's data.

File Management Structures

The following structures are used in file management:

Structure	Description
BY_HANDLE_FILE_INFORMATION	Contains information that the GetFileInformationByHandle() function retrieves.
EFS_CERTIFICATE_BLOB	Contains a certificate.
EFS_HASH_BLOB	Contains a certificate hash.
ENCRYPTION_CERTIFICATE	Contains a certificate and the SID of its owner.
ENCRYPTION_CERTIFICATE_HASH	Contains a certificate hash and display information for the certificate.
ENCRYPTION_CERTIFICATE_HASH_LIST	Contains a list of certificate hashes.
ENCRYPTION_CERTIFICATE_LIST	Contains a list of certificates.
EXFAT_STATISTICS	Contains statistical information from the exFAT file system.
FAT_STATISTICS	Contains statistical information from the FAT file system.
FILE_ALLOCATED_RANGE_BUFFER	Indicates a range of bytes in a file. This structure is used with the FSCTL_QUERY_ALLOCATED_RANGES control code. On input, the structure indicates the range of the file to search. On output, the operation retrieves an array of FILE_ALLOCATED_RANGE_BUFFER structures to indicate the allocated ranges within the search range.
FILE ALLOCATION INFO	Contains the total number of bytes that should be allocated for a file. This structure is used when calling the SetFileInformationByHandle() function.
FILE_ATTRIBUTE_TAG_INFO	Receives the requested file attribute information. Used for any handles. Use only when calling GetFileInformationByHandleEx().
FILE_BASIC_INFO	Contains the basic information for a file. Used for file handles.
FILE_COMPRESSION_INFO	Receives file compression information. Used for any handles. Use only when calling GetFileInformationByHandleEx().
FILE_DISPOSITION_INFO	Indicates whether a file should be deleted. Used for any handles. Use only when calling SetFileInformationByHandle().
FILE_END_OF_FILE_INFO	Contains the specified value to which the end of the file should be set. Used for file handles. Use

	only when calling SetFileInformationByHandle().
FILE_ID_BOTH_DIR_INFO	Contains information about files in the specified directory. Used for directory handles. Use only when calling GetFileInformationByHandleEx(). The number of files that are returned for each call to GetFileInformationByHandleEx() depends on the size of the buffer that is passed to the function. Any subsequent calls to GetFileInformationByHandleEx() on the same handle will resume the enumeration operation after the last file is returned.
FILE_ID_DESCRIPTOR	Specifies the type of ID that is being used.
FILE_IO_PRIORITY_HINT_INFO	Specifies the priority hint for a file I/O operation.
FILE_MAKE_COMPATIBLE_BUFFER	Specifies the disc to close the current session for. This control code is used for UDF file systems. This structure is used for input when calling FSCTL_MAKE_MEDIA_COMPATIBLE.
FILE_NAME_INFO	Receives the file name. Used for any handles. Use only when calling GetFileInformationByHandleEx().
FILE_OBJECTID_BUFFER	Contains an object identifier and user-defined metadata associated with the object identifier.
FILE_QUERY_ON_DISK_VOL_INFO_BUFFER	Receives the volume information from a call to FSCTL_QUERY_ON_DISK_VOLUME_INFO. FILE_QUERY_SPARING_BUFFER - Contains defect management properties.
FILE_REMOTE_PROTOCOL_INFORMATION	Contains file remote protocol information. This structure is not declared in a public header.
FILE_RENAME_INFO	Contains the name to which the file should be renamed. Use only when calling SetFileInformationByHandle().
FILE_SET_DEFECT_MGMT_BUFFER	Specifies the defect management state to be set.
FILE_SET_SPARSE_BUFFER	Specifies the sparse state to be set. Windows Server 2003 and Windows XP/2000: This structure is optional.
FILE_STANDARD_INFO	Receives extended information for the file. Used for file handles. Use only when calling GetFileInformationByHandleEx().
FILE_STREAM_INFO	Receives file stream information for the specified file. Used for any handles. Use only when calling GetFileInformationByHandleEx().
FILE_ZERO_DATA_INFORMATION	Contains a range of a file to set to zeroes. This structure is used by the FSCTL_SET_ZERO_DATA control code

FILESYSTEM_STATISTICS	Contains statistical information from the file system.
FIND_BY_SID_DATA	Contains data for the FSCTL_FIND_FILES_BY_SID control code.
FIND_BY_SID_OUTPUT	Represents a file name. This structure is not yet declared in this manner in WinIoCtl.h. To use this structure, you must define it yourself using a different name, such as FIND_BY_SID_OUTPUT_TEMP. When the structure is declared correctly in the header, you will be able to download the header file and modify your code to use the structure declared in the header file.
NTFS_FILE_RECORD_INPUT_BUFFER	Contains data for the FSCTL_GET_NTFS_FILE_RECORD control code.
NTFS_FILE_RECORD_OUTPUT_BUFFER	Receives output data from the FSCTL_GET_NTFS_FILE_RECORD control code.
NTFS_STATISTICS	Contains statistical information from the NTFS file system.
OFSTRUCT	Contains information about a file that the OpenFile() function opened or attempted to open.
WIN32_FILE_ATTRIBUTE_DATA	Contains attribute information for a file or directory. The GetFileAttributesEx() function uses this structure.
WIN32_FIND_DATA	Contains information about the file that is found by the FindFirstFile(), FindFirstFileEx(), or FindNextFile() function.
WIN32_FIND_STREAM_DATA	Contains information about the stream found by the FindFirstStreamW() or FindNextStreamW() function.