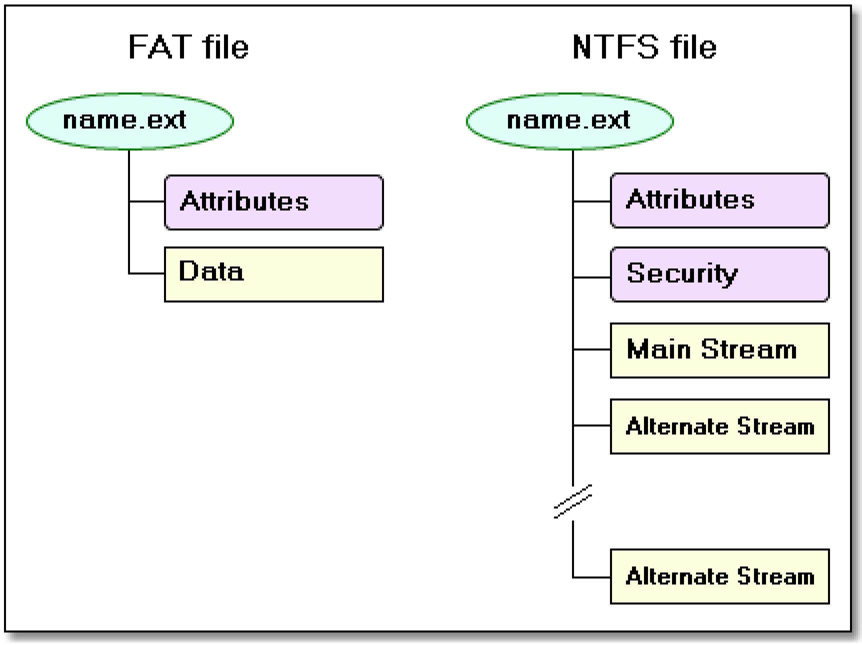
NTFS Alternate Streams: What, When, and How To

flexhex.com/docs/articles/alternate-streams.phtml

What Are Alternate Streams?

NTFS alternate streams, or named streams, or ADS (which stands for Alternate Data Streams) is a little known but very useful NTFS feature. Comparing with earlier file systems like FAT, NTFS significantly expands the customary concept of a file as a named portion of data:



The unnamed stream is a mandatory element and is always present. If you are creating an alternate atream and the file does not exist, the system will automatically create a zero length unnamed stream. If you are deleting the unnamed stream, the system considers it as a request to delete the whole file, and all the alternate streams will also be deleted.

The security descriptor and the file attributes belong to the file as a whole, not to the unnamed stream. For instance, no stream can be opened for writing if the ***read-only*** attribute is set.

Note, however, that not all the attributes are file wide - some are stream based, most notably **encrypted**, **compressed**, and **sparse**.

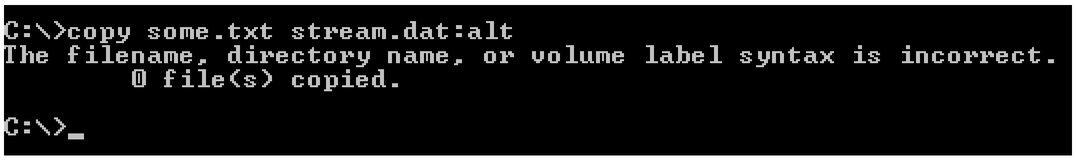
When a program opens an NTFS file, it in fact opens the unnamed stream. In order to specify an alternate stream, append the colon character and the stream name to the file name. That is, **filename.ext** specifies the unnamed stream of the file (or, depending on the context, the whole file), and **filename.ext:strname** specifies the alternate stream **strname**.

A directory also can have alternate streams, which can be accessed exactly the same way as file streams. However a directory can't have an unnamed stream, and any attempt to access it (that is specifying the directory name without a stream name) will result the *Access denied* error.

Because the colon character is used also in drive specification, it may cause an ambiguity. For example, **A:B** may represent either a file **B** in the current directory of the **A:** drive, or a stream **B** of the file **A**. The system always resolves this ambiguity as a drive and a name, so if you want it to be interpreted the other way, specify the current directory - in our example the path should look as **.\A:B**.

System Support for Stream Operations

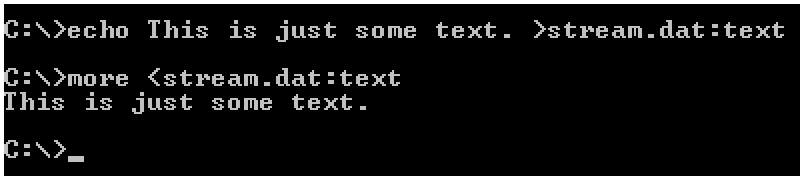
The good news is the Windows Explorer and the command-line **copy** command recognize alternate streams and correctly copy multi-stream files. The bad news is the system support is limited to that. The Windows Explorer does not allow any stream operations, and if you try to specify a stream name in the command line, you will get an error.



```
C:\>copy some.txt stream.dat:alt
The filename, directory name, or volume label syntax is incorrect.
        0 file(s) copied.

C:\>_
```

Other commands, for instance, **echo**, **more**, and **type**, can access alternate streams by using redirectors < and >, which are stream-enabled. MSDN alternate stream example uses those commands with the redirectors for creating an alternate stream and inspecting its contents.



```
C:\>echo This is just some text. >stream.dat:text

C:\>more <stream.dat:text
This is just some text.

C:\>_
```

While these commands undoubtedly work, it is hard to imagine any practical use for this technique. Of course, you can always use FlexHEX to perform any stream operation, however a hex editor is probably not the best tool if all you want is just to copy or rename a stream. So we have developed a complete set of free command line tools for handling alternate streams. Just download and unpack them to your Windows directory.

So When to Use Alternate Streams?

Certainly you should not use alternate streams for storing any critical information. Older file systems are still widely used, and they don't support the advanced NTFS features. If you copy an NTFS file to a USB drive, flash card, CD-R/RW, or any other non-NTFS drive, the system will copy the main stream only and will ignore all the alternate streams. The same is true for FTP/HTTP transfers. No warning is given, and a user, relying on alternate streams, might get a nasty surprise. So the Microsoft reluctance to provide user tools for alternate streams is not all that unfounded.

However alternate streams are still extremely useful. There is a lot of non-critical information that alternate streams is the most natural place to store to. Examples are thumbnails for graphical files, parsing information for program sources, spellcheck and formatting data for documents, or any other data that can easily be rebuilt. This way the file can be stored on any file system, but keeping the file on an NTFS drive will greatly increase processing speed.

Programming Considerations

In order to improve readability, the code examples below don't include any error processing. You should add some error checks if you want to use this code in your program. See the sources in the download section for an example of error handling.

You can use the Win32 API function **GetVolumeInformation** to determine if the drive supports alternate streams.

```
char szVolName[MAX_PATH], szFSName[MAX_PATH];
DWORD dwSN, dwMaxLen, dwVolFlags;
::GetVolumeInformation("C:\\", szVolName, MAX_PATH, &dwSN,
                       &dwMaxLen, &dwVolFlags, szFSName, MAX_PATH);

if (dwVolFlags & FILE_NAMED_STREAMS) {
  // File system supports named streams
}
else {
  // Named streams are not supported
}
```

You might prefer to play safe and check the file system name instead of the flag:

```
if (_stricmp(szFSName, "NTFS") == 0)   // If NTFS
```

Creating/Opening a Stream

You can create or open a named stream exactly the same way you create or open an unnamed stream:

```
HANDLE hFile = ::CreateFile("file.dat:alt", ...
```

Keep in mind that if the file does not exist, creating a named stream will also create a zero-length unnamed stream.

Deleting a Stream

The Win32 API function **DeleteFile** fully supports alternate streams, so deleting a stream is no more complex than deleting a file:

```
::DeleteFile("file.dat:alt");
```

As has been noted before, you can't delete the unnamed stream alone; deleting it also deletes all the alternate streams.

Copying a Stream

You can use the Win32 API functions **CopyFile/CopyFileEx** to copy alternate streams. However these functions are used for copying files as well as streams so you might find the result to be totally unexpected. They perform stream-to-stream copying if the destination is a named stream, but copying to an unnamed stream is treated as a file operation. There are two specific cases you should be aware of:

Unnamed stream to unnamed stream: treated as a file operation, that is all the named streams also get copied. If the target file exists, it is replaced.

Named stream to unnamed stream: also treated as a file operation, although only one stream gets copied. Existing target file gets deleted, so instead of replacing the unnamed stream as you might expect, the function replaces the whole target file with a new single-stream file.

Copying a stream in a simple read/write loop gives a more predictable result and is preferrable in most cases:

```
HANDLE hInFile = ::CreateFile(szFromStream, GENERIC_READ, FILE_SHARE_READ, NULL,
             OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN, NULL);
HANDLE hOutFile = ::CreateFile(szToStream, GENERIC_WRITE, FILE_SHARE_READ, NULL,
             CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL | FILE_FLAG_SEQUENTIAL_SCAN, NULL);

BYTE buf[64*1024];
DWORD dwBytesRead, dwBytesWritten;

do {
  ::ReadFile(hInFile, buf, sizeof(buf), &dwBytesRead, NULL);
  if (dwBytesRead) ::WriteFile(hOutFile, buf, dwBytesRead, &dwBytesWritten, NULL);
} while (dwBytesRead == sizeof(buf));

::CloseHandle(hInFile);
::CloseHandle(hOutFile);
```

The code above is the stream copy loop used in our **CS** command-line tool (the error processing code has been removed to improve readability). The complete sources are available in the download section.

Renaming a Stream

It seems there is no way - documented or undocumented - to rename a stream short of directly modifying the corresponding MFT entry.

Enumerating Streams: Windows Vista and later systems

Windows Vista introduced new functions for stream enumeration: **FindFirstStreamW** / **FindFirstStreamTransactedW** and **FindNextStreamW** (note that only the Unicode versions are available). Their usage is fairly simple and is very similar to the well-known **FindFirstFile** / **FindFirstFile** functions.

```
// Enumerate file's streams and print their sizes and names

WIN32_FIND_STREAM_DATA fsd;
HANDLE hFind = NULL;

try {
  hFind = ::FindFirstStreamW(L"teststreams.dat", FindStreamInfoStandard, &fsd, 0);
  if (hFind == INVALID_HANDLE_VALUE) throw ::GetLastError();

  for (;;) {
    printf("%-12I64u%S\n", fsd.StreamSize, fsd.cStreamName);
    if (!::FindNextStreamW(hFind, &fsd)) {
      DWORD dr = ::GetLastError();
      if (dr != ERROR_HANDLE_EOF) throw dr;
      break;
    }
  }
}
catch (DWORD err) {
  printf("Error! Windows error code: %u\n", err);
}

if (hFind != NULL) ::FindClose(hFind);
```

Enumerating Streams: pre-Vista systems

This is the tricky one. The only Win32 API function that can be used for enumerating streams is **BackupRead** and using it would be a bad idea. The problem with **BackupRead** is that you must actually read all the file streams in order to get their names. Even if the file contains no alternate streams, you will have to read the whole unnamed stream just to establish this fact. As a result any large enough file will bring your application to the screeching halt.

Fortunately there is an undocumented, but quite a reliable way of obtaining stream information using the Native API function **NtQueryInformationFile** (or **ZwQueryInformationFile**).

```
// Open a file and obtain stream information

BYTE InfoBlock[64 * 1024];  // Buffer must be large enough
PFILE_STREAM_INFORMATION pStreamInfo = (PFILE_STREAM_INFORMATION)InfoBlock;
IO_STATUS_BLOCK ioStatus;

HANDLE hFile = ::CreateFile(szPath, 0, FILE_SHARE_READ | FILE_SHARE_WRITE,
                            NULL, OPEN_EXISTING, 0, NULL);
NtQueryInformationFile(hFile, &ioStatus, InfoBlock,
                       sizeof(InfoBlock), FileStreamInformation);
::CloseHandle(hFile);
```

A slightly more complex code is required if you want to open a directory. First, the program must have the **SE_BACKUP_NAME** privilege; second, you must specify the **FILE_FLAG_BACKUP_SEMANTICS** flag when calling **CreateFile**; and third, you must keep in mind the fact, that unlike files, a directory may have no streams at all, and so the program should recognize the situation when no stream info is returned.

```
// Open a directory and obtain stream information

// Obtain backup privilege in case we don't have it
HANDLE hToken;
TOKEN_PRIVILEGES tp;
::OpenProcessToken(::GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES, &hToken);
::LookupPrivilegeValue(NULL, SE_BACKUP_NAME, &tp.Privileges[0].Luid);
tp.PrivilegeCount = 1;
tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
::AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(TOKEN_PRIVILEGES), NULL, NULL);
::CloseHandle(hToken);

HANDLE hFile = ::CreateFile(szPath, 0, FILE_SHARE_READ | FILE_SHARE_WRITE,
                         NULL, OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS, NULL);

BYTE InfoBlock[64 * 1024];  // Buffer must be large enough
PFILE_STREAM_INFORMATION pStreamInfo = (PFILE_STREAM_INFORMATION)InfoBlock;
IO_STATUS_BLOCK ioStatus;

pStreamInfo->StreamNameLength = 0;  // Zero in this field means empty info block
NtQueryInformationFile(hFile, &ioStatus, InfoBlock,
                       sizeof(InfoBlock), FileStreamInformation);
::CloseHandle(hFile);
```

The function **NtQueryInformationFile** places a sequence of **FILE_STREAM_INFORMATION** structures in the **InfoBlock** buffer. **FILE_STREAM_INFORMATION** is a variable-length structure, its size is stored in the

NextEntryOffset field (which can also be interpreted as the offset to the next record). The last structure in the list has zero **NextEntryOffset** field.

The **StreamName** field contains the stream name in UNICODE; the **StreamNameLength** field is the name length in bytes (there is no terminating zero).

Now we have successfully obtained the array of stream information record and can print the stream names:

```
WCHAR wszStreamName[MAX_PATH];

for (;;) {
  // Check if stream info block is empty (directory may have no stream)
  if (pStreamInfo->StreamNameLength == 0) break; // No stream found

  // Get null-terminated stream name
  memcpy(wszStreamName, pStreamInfo->StreamName, pStreamInfo->StreamNameLength);
  wszStreamName[pStreamInfo->StreamNameLength / sizeof(WCHAR)] = L'\0';

  print("%S", wszStreamName);

  if (pStreamInfo->NextEntryOffset == 0) break;   // No more stream records
  pStreamInfo = (PFILE_STREAM_INFORMATION)
     ((LPBYTE)pStreamInfo + pStreamInfo->NextEntryOffset);   // Next stream record
}
```

You can remove the **if (pStreamInfo->StreamNameLength == 0)** check if you don't intend to process directories. A file always has at least one stream so this check is not necessary.

Please note that stream names include the attribute name, that is the unnamed stream name looks as **::$DATA**, a stream named **alt** looks as **:alt:$DATA** and so on.

If you have DDK installed, then you already have all the required headers and import libraries. Otherwise download the sources and include a header file **AltStreams.h** from the **ListStreams** projects that contains all the required definitions. Before calling the **NtQueryInformationFile** function, include the following code for dynamic linking:

```
NTQUERYINFORMATIONFILE NtQueryInformationFile;
(FARPROC&)NtQueryInformationFile = ::GetProcAddress(
                  ::GetModuleHandle("ntdll.dll"), "NtQueryInformationFile");
```

For a real life example please see the source code of our **LS** command line tool. The sources can be downloaded from the download section.

Command Line Tools

All our stream-enabled command line tools are free and can be downloaded from the download section. You cannot distribute these tools separately, however you can distribute the original zip archive freely.

Copy Stream

Usage:

```
cs from_stream to_stream
```

This command copies separate streams, for example

```
cs C:\SomeFile.dat:str stream.dat:alt
```

If the stream is not specified, the command assumes the unnamed stream. For instance, the command

```
cs c:\report.txt reports.txt:r20
```

will copy the file's primary stream. If the file report.txt has any alternate streams, they will be ignored (use the standard **copy** command to copy the file as a whole).

Delete Stream

Usage:

```
ds stream
```

Delete the specified stream, for example

```
ds stream.dat:alt
```

If no stream name is specified, the command deletes the whole file (deleting the unnamed stream causes all the streams to be deleted).

The command don't ask for confirmation, so be careful.

Strip File (Delete All Alternate Streams)

Usage:

```
sf file
```

Deletes all file's named streams, for example

```
sf stream.dat
```

The program leaves the main unnamed stream intact, but all the attached alternate (named) streams will be removed.

The package also includes a batch file SFs.bat, which calls sf.exe for each file in the current directory.

Rename Stream

There is no known method of renaming a stream, so we have to use the copy/delete sequence. While this method will do the trick, renaming a large stream may take considerable time.

Usage:

```
rs file oldname newname
```

Rename the stream **oldname** of the file **file** to **newname**. For example, the command

```
rs stream.dat alt text
```

renames **stream.dat:alt** to **stream.dat:text**.

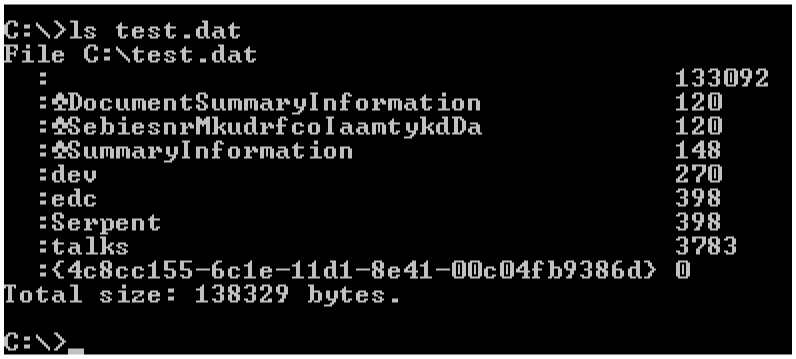
List Streams

This command lists all streams of the specified file and their size.

Usage:

```
ls file
```

Example:



```
C:\>ls test.dat
File C:\test.dat
  :                                        133092
  :♣DocumentSummaryInformation             120
  :♣SebiesnrMkudrfcoIaamtykdDa             120
  :♣SummaryInformation                     148
  :dev                                     270
  :edc                                     398
  :Serpent                                 398
  :talks                                   3783
  :{4c8cc155-6c1e-11d1-8e41-00c04fb9386d}  0
Total size: 138329 bytes.

C:\>_
```

The LS command returns the standard success code 0 only if at least one alternate stream was found. See the topic "Calling From a Batch File" below for a usage example.

Calling From a Batch File

Like most standard command line commands, the stream commands return the standard exit codes that can be analyzed with the **if errorlevel** batch command. There are two possible exit codes: 0 means success, and 1 means error. The technique is illustrated by the following example batch file:

```
@echo off
echo Copying stream...

cs c:\report.txt reports.txt:20
if errorlevel 1 goto cmderr
echo Successfully copied!
goto exitbatch

:cmderr
echo Some error occured.

:exitbatch
rem Exiting the batch file....
```

The LS command returns the standard success code 0 when at least one alternate stream present. The standard error code 1 is returned if the file contains an unnamed stream only or if I/O error occured. The following example shows how to check for presence of alternate streams:

```
@echo off
rem This batch file finds and list all files with ADS in the current directory

echo Files containing alternate streams:

for %%f in (*.*) do call :checkf %%f
goto exitbatch

:checkf
rem We don't want to list streams so throw out the output
ls %1 >nul
if not errorlevel 1 echo %1

:exitbatch
```

This batch file **FS.bat** can be downloaded as a part of the stream tools package.

Please refer to the Windows Help to learn more about Windows batch files and batch commands.