

# Microsoft FCI/FDI Library Description

Copyright © 1996-1997 Microsoft Corporation. All rights reserved.

## *Abstract*

*This document describes use of the functions in the Microsoft File Compression Interface and File Decompression Interface libraries or in CABINET.DLL to create or extract from Microsoft Cabinet Files.*

## **Table of Contents**

<b>INTRODUCTION.....</b>	<b>1</b>
<b>FCI.....</b>	<b>1</b>
FCICREATE.....	1
FCIADDFILE.....	1
FCIFLUSHCABINET.....	1
FCIFLUSHFOLDER.....	1
FCIDESTROY.....	1
<b>FDI.....</b>	<b>1</b>
FDICREATE.....	1
FDIISCABINET.....	1
FDICOPY.....	1
FDIDESTROY.....	1

## Introduction

The FCI (File Compression Interface) and FDI (File Decompression Interface) libraries provide the ability to create and extract files from cabinets (also known as “CAB files”). In addition, the libraries provide compression and decompression capability to reduce the size of file data stored in cabinets.

The FCI and FDI libraries, FCI.LIB and FDI.LIB, are available in both 32-bit and 16-bit forms. However, the 16-bit version will run more slowly than the 32-bit version.

FCI and FDI support multiple simultaneous contexts, so it is possible to create or extract multiple cabinets simultaneously within the same application. If the application is multi-threaded, it is also possible to run a different context in each thread; however, it is not permitted for the application to use the *same* context simultaneously in multiple threads (e.g. one cannot call FCIAAddFile from two different threads, using the same FCI context).

FCI and FDI operate using the technique of function callbacks; some of the parameters of the FCI and FDI APIs are pointers to functions in the client application. The parameters and purpose of these functions are explained fully in this document. The *fc\_i\_int.h* and *fd\_i\_int.h* header files provide macros for declaring the callback functions, and use keywords such HUGE, FAR, and DIAMONDAPI, which ensure that the functions are properly defined for both 32-bit and 16-bit operation. For example, in the case of the memory allocation and memory free functions, the following definitions exist in *fc\_i\_int.h*:

```
#define FNFCIALLOC(fn) void HUGE * FAR DIAMONDAPI fn(ULONG cb)
#define FNFCIFREE(fn) void FAR DIAMONDAPI fn(void HUGE *pv)
```

These declarations can be used as follows:

```
FNFCIALLOC(mem_alloc)
{
    return malloc(cb);
}

FNFCIFREE(mem_free)
{
    return free(memory);
}

some_function()
{
    hfci = FCICreate(
        &erf,
        filedest,
        mem_alloc,
        mem_free,
        etc.
    );
}
```

It should be noted that the FCI callback function names all begin with the string “FCI”. In addition, the FCI and FDI i/o functions (open, close, read, write, seek) take different parameters, and cannot be used interchangeably.

The FDI i/o functions take parameters which are identical to those of the C run-time library routines `_open`, `close`, `read`, `write`, and `lseek`. The FCI i/o functions take similar parameters, with the addition of an error pointer in which to return an i/o error, and the client’s context pointer originally passed in to the FCICreate API.

## Microsoft FCI/FDI Library Description

Two example applications are provided; *testfci* and *testfdi*. These applications demonstrate how all of the FCI and FDI APIs, respectively, may be used.

## FCI

The five FCI (File Compression Interface) APIs are:

<b>FCICreate</b>	Create an FCI context
<b>FCIAddFile</b>	Add a file to the cabinet under construction
<b>FCIFlushCabinet</b>	Complete the current cabinet
<b>FCIFlushFolder</b>	Complete the current folder and start a new folder
<b>FCIDestroy</b>	Destroy an FCI context

## FCICreate

### Usage

```
HFCI DIAMONDAPI FCICreate(
    PERF                perf,
    PFNFCIFILEPLACED   pfnfiledest,
    PFNFCIALLOC        pfnalloc,
    PFNFCIFREE         pfnfree,
    PFNFCIOPEN         pfnopen,
    PFNFCIREAD         pfnread,
    PFNFCIWRITE        pfnwrite,
    PFNFCICLOSE        pfnclose,
    PFNFCISEEK         pfnseek,
    PFNFCIDELETE       pfndelete,
    PFNFCIGETTEMPFILE pfnfcigt,
    PCCAB               pccab,
    void FAR *         pv
);
```

### Parameters

<i>perf</i>	Pointer to an error structure
<i>pfnfiledest</i>	Function to call when a file is placed
<i>pfnalloc</i>	Memory allocation function
<i>pfnfree</i>	Memory free function
<i>pfnopen</i>	Function to open a file
<i>pfnread</i>	Function to read data from a file
<i>pfnwrite</i>	Function to write data to a file
<i>pfnclose</i>	Function to close a file
<i>pfnseek</i>	Function to seek to a new position in a file
<i>pfntemp</i>	Function to obtain a temporary file name
<i>pfndelete</i>	Function to delete a file
<i>pccab</i>	Parameters for creating cabinet
<i>pv</i>	Client context parameter

### Description

The **FCICreate** API creates an FCI context that is passed to other FCI APIs.

The *perf* parameter should point to a global or allocated ERF structure. Any errors returned by **FCICreate** or subsequent FCI APIs using the same context will cause the ERF structure to be filled out.

The *pfnalloc* and *pfnfree* parameters should point to memory allocation and memory free functions which will be called by FCI to allocate and free memory. These two functions take parameters identical to the standard C malloc and free functions.

The *pfnopen*, *pfnread*, *pfnwrite*, *pfnclose*, *pfnseek*, and *pfndelete* parameters should point to functions which perform file open, file read, file write, file close, file seek, and file delete operations respectively. These functions must accept parameters similar to those for the standard *\_open*, *\_read*, *\_write*, *\_close*, *\_lseek*, and *remove* functions, with

the addition of two additional parameters to the list; **err** and **pV**. The **err** parameter is an *int \**, and upon entry into the function, *\*err* will equal zero. However, if the function returns failure, *\*err* should be set to an error code of the application's choosing, which will be returned via *perf* (the error code is not used by FCI, and is not required to conform to C run-time library *errno* conventions). The **pV** parameter will equal the client's context parameter passed in to **FCICreate**.

The *pfntemp* parameter should point to a function which returns the name of a suitable temporary file. Three parameters will be passed to this function; **pszTempName**, an area of memory to store the filename, **cbTempName**, the size of the memory area, and **pV**, the client's context pointer. The filename returned by this function should not occupy more than **cbTempName** bytes. FCI may open several temporary files at once, so it is important to ensure that a different filename is returned each time, and that the file does not already exist. The function should return TRUE for success, or FALSE for failure.

The *pfnfiledest* parameter should point to a function which will be called whenever the location of a file or file segment on a particular cabinet has been finalized. This information is useful only when files are being stored across multiple cabinets. The parameters passed to this function are **pccab**, a pointer to the CCAB structure of the cabinet on which the file has been stored, **pszFile**, the filename of the file which has been placed, **cbFile**, the file size, and **fContinuation**, a boolean which signifies whether the file is a later segment of a file which has been split across cabinets. In addition, the client context value, **pV**, is also passed as a parameter.

The *pccab* parameter should point to an initialized CCAB structure, which will provide FCI with details on how to build the cabinet. The CCAB fields are explained below:

The **cb** field, the media size, specifies the maximum size of a cabinet which will be created by FCI. If necessary, multiple cabinets will be created. To ensure that only one cabinet is created, a sufficiently large number should be used for this parameter.

The **cbFolderThresh** field specifies the maximum number of compressed bytes which may reside in a folder before a new folder is created. A higher folder threshold improves compression performance (since creating a new folder resets the compression history), but increases random access time to the folder.

The **iCab** field is used by FCI to count the number of cabinets that have been created so far. This value can also be read by the application to determine the name of a cabinet. See the *GetNextCab* parameter of the **FCIAddFile** API for details.

The **iDisk** field is used in a similar manner to **iCab**. See the *GetNextCab* parameter of the **FCIAddFile** API for details.

The **setID** field is for the use of the application, and can be initialized with any number. The set ID is stored in the cabinet.

The **szDisk** field should contain a disk-specific string (such as "Disk1", "Disk2", etc.) corresponding to the disk on which the cabinet is placed. Alternatively, if cabinets are not spanning multiple disks, the string can simply be a null string. This field is stored in the cabinet and is used upon extraction to prompt the user to insert the correct disk. See the **FCIAddFile** API for details.

The **szCab** field should contain a string which contains the name of the first cabinet to be created (e.g. "APP1.CAB"). In the event of multiple cabinets being created, the *GetNextCab* function called by the **FCIAddFile** API allows subsequent cabinet names to be specified.

The **szCabPath** field should contain the complete path of where to create the cabinet (e.g. "C:\MYFILES").

## Microsoft FCI/FDI Library Description

The `cbReserveCFHeader`, `cbReserveCFFolder`, and `cbReserveCFData` fields can be set to create per-cabinet, per-folder, and per-datablock reserved sections in the cabinet. For example, setting `cbReserveCFHeader` to 6144 is commonly used to reserve a 6k space in the cabinet file as needed for codesigning. The other reserved sections are not commonly used.

### Returns

If successful, a non-NULL HFCI context pointer is returned. If unsuccessful, NULL is returned, and the error structure pointed to by *perf* is filled out.



## FCIAddFile

### Usage

```

BOOL DIAMONDAPI FCIAddFile(
    HFCI          hfcI,
    char          *pszSourceFile,
    char          *pszFileName,
    BOOL          fExecute,
    PFNFCIGETNEXTCABINET GetNextCab,
    PFNFCISTATUS  pfnProgress,
    PFNFCIGETOPENINFO  pfnOpenInfo,
    TCOMP        typeCompress
)

```

### Parameters

<i>hfcI</i>	FCI Context pointer originally returned by <b>FCICreate</b>
<i>pszSourceFile</i>	Name of file to add (should include path information)
<i>pszFileName</i>	Name under which to store the file in the cabinet
<i>fExecute</i>	Boolean indicating whether the file should be executed when it is extracted
<i>GetNextCab</i>	Function called to obtain specifications on the next cabinet to create
<i>pfnProgress</i>	Progress function called to update the user
<i>pfnOpenInfo</i>	Function called to open a file and return file date, time and attributes
<i>typeCompress</i>	Compression type to use

### Description

The **FCIAddFile** API adds a file to the cabinet under construction.

The *hfcI* parameter must be the context pointer returned by a previous call to **FCICreate**.

The *pszSourceFile* parameter specifies the location of the file to be added to the cabinet, and should therefore include as much path information as possible (e.g. "C:\MYFILES\TEST.EXE").

The *pszFileName* parameter specifies the name of the file inside the cabinet, and should not include any path information (e.g. "TEST.EXE").

The *fExecute* parameter specifies whether the file should be executed automatically when the cabinet is extracted. When set, the `_A_EXEC` attribute will be added to the file entry in the CAB. This mechanism is used in some Microsoft self-extracting executables, and could be used for this purpose in any custom extract application.

The *GetNextCab* parameter should point to a function which is called whenever FCI wishes to create a new cabinet, which will happen whenever the size of the cabinet is about to exceed the media size as specified in the `cb` field of the CCAB structure passed to **FCICreate**. The *GetNextCab* function is called with three parameters which are explained below:

The first parameter, **pccab**, is a pointer to a copy of the CCAB structure of the cabinet which has just been completed. However, the `iCab` field will have been incremented by one. When this function returns, the next cabinet will be created using the fields in this structure, so these fields should be modified as is necessary. In

particular, the *szCab* field (the cabinet name) should be changed. If creating multiple cabinets, typically the *iCab* field is used to create the name; for example, the *GetNextCab* function might include a line which does:

```
sprintf(pccab->szCab, "FOO%d.CAB", pccab->iCab);
```

Similarly, the disk name, media size, folder threshold, etc. parameters may also be modified.

The second parameter, **cbPrevCab**, is an estimate of the size of the cabinet which has just been completed.

The last parameter, **pv**, is the application-defined value originally passed to **FCICreate**.

The *GetNextCab* function should return TRUE for success, or FALSE to abort cabinet creation.

The *pfnProgress* parameter should point to a function which is called periodically by FCI so that the application may send a progress report to the user. The progress function has four parameters; **typeStatus**, which specifies the type of status message, **cb1** and **cb2**, which are numbers, the meaning of which is dependent upon **typeStatus**, and **pv**, the application-specific context pointer.

The **typeStatus** parameter may take on values of **statusFile**, **statusFolder**, or **statusCabinet**. If **typeStatus** equals **statusFile** then it means that FCI is compressing data blocks into a folder. In this case, **cb1** is either zero, or the compressed size of the most recently compressed block, and **cb2** is either zero, or the uncompressed size of the most recently read block (which is usually 32K, except for the last block in a folder, which may be smaller). There is *no* direct relation between **cb1** and **cb2**; FCI may read several blocks of uncompressed data before emitting any compressed data; if this happens, some **statusFile** messages may contain, for example, **cb1** = 0 and **cb2** = 32K, followed later by other messages which contain **cb1** = 20K and **cb2** = 0.

If **typeStatus** equals **statusFolder** then it means that FCI is copying a folder to a cabinet, and **cb1** is the amount copied so far, and **cb2** is the total size of the folder. Finally, if **typeStatus** equals **statusCabinet**, then it means that FCI is writing out a completed cabinet, and **cb1** is the estimated cabinet size that was previously passed to *GetNextCab*, and **cb2** is the actual resulting cabinet size.

The progress function should return 0 for success, or -1 for failure, with an exception in the case of **statusCabinet** messages, where the function should return the desired cabinet size (*cb2*), or possibly a value rounded up to slightly higher than that.

The *pfnOpenInfo* parameter should point to a function which opens a file and returns its datestamp, timestamp, and attributes. The function will receive five parameters; **pszName**, the complete pathname of the file to open; **pdate**, a memory location to return a FAT-style date code; **ptime**, a memory location to return a FAT-style time code; **pattrs**, a memory location to return FAT-style attributes; and **pv**, the application-specific context pointer originally passed to **FCICreate**. The function should open the file using a file open function compatible with those passed in to **FCICreate**, and return the resulting file handle, or -1 if unsuccessful.

The *typeCompress* parameter specifies the type of compression to use, which may be either **tcompTYPE\_NONE** for no compression, or **tcompTYPE\_MSZIP** for Microsoft ZIP compression. Other compression formats may be supported in the future.

## Returns

If successful, TRUE is returned. If unsuccessful, FALSE is returned, and the error structure pointed to by *perf* (from **FCICreate**) is filled out.



## FCIFlushCabinet

### Usage

```

BOOL DIAMONDAPI FCIFlushCabinet(
    HFCI                hfci,
    BOOL                fGetNextCab,
    PFNFCIGETNEXTCABINET GetNextCab,
    PFNFCISTATUS        pfnProgress
)

```

### Parameters

<i>hfci</i>	FCI Context pointer originally returned by <b>FCICreate</b>
<i>fGetNextCab</i>	Name of file to add (should include path information)
<i>GetNextCab</i>	Function called to obtain specifications on the next cabinet to create
<i>pfnProgress</i>	Progress function called to update the user

### Description

The **FCIFlushCabinet** API forces the current cabinet under construction to be completed immediately and written to disk. Further calls to **FCIAddFile** will cause files to be added to another cabinet. It is also possible that there exists pending data in FCI's internal buffers that will may require spillover into another cabinet, if the current cabinet has reached the application-specified media size limit.

The *hfci* parameter must be the context pointer returned by a previous call to **FCICreate**.

The *fGetNextCab* flag determines whether the function pointed to by the supplied *GetNextCab* parameter, will be called. If *fGetNextCab* is TRUE, then *GetNextCab* will be called to obtain continuation information. Otherwise, if *fGetNextCab* is FALSE, then *GetNextCab* will be called only if the cabinet overflows.

The *pfnProgress* parameter should point to a function which is called periodically by FCI so that the application may send a progress report to the user. This function works in an identical manner to the progress function passed to **FCIAddFile**.

### Returns

If successful, TRUE is returned. If unsuccessful, FALSE is returned, and the error structure pointed to by *perf* (from **FCICreate**) is filled out.

## FCIFlushFolder

### Usage

```
BOOL DIAMONDAPI FCIFlushFolder(  
    HFCI                hfci,  
    PFNFCIGETNEXTCABINET GetNextCab,  
    PFNFCISTATUS        pfnProgress  
)
```

### Parameters

<i>hfci</i>	FCI Context pointer originally returned by <b>FCICreate</b>
<i>GetNextCab</i>	Function called to obtain specifications on the next cabinet to create
<i>pfnProgress</i>	Progress function called to update the user

### Description

The **FCIFlushFolder** API forces the current folder under construction to be completed immediately, effectively resetting the compression history at this point (if compression is being used).

The *hfci* parameter must be the context pointer returned by a previous call to **FCICreate**.

The supplied *GetNextCab* function will be called if the cabinet overflows, which is a possibility if the pending data buffered inside FCI causes the application-specified cabinet media size to be exceeded.

The *pfnProgress* parameter should point to a function which is called periodically by FCI so that the application may send a progress report to the user. This function works in an identical manner to the progress function passed to **FCIAddFile**.

## FCIDestroy

### Usage

```
BOOL DIAMONDAPI FCIDestroy(  
    HFCI  hfci  
)
```

### Parameters

*hfci*            FCI context handle returned by **FCICreate**

### Description

The **FCIDestroy** API destroys an *hfci* context, freeing any memory and temporary files associated with the context.

### Returns

If successful, TRUE is returned. If unsuccessful, FALSE is returned. The only reason for failure is that the *hfci* passed in was not a proper context handle.

## FDI

The four FDI (File Decompression Interface) APIs are:

<b>FDICreate</b>	Create an FDI context
<b>FDIsCabinet</b>	Determines whether a file is a cabinet, and returns information if so
<b>FDICopy</b>	Extracts files from cabinets
<b>FDIDestroy</b>	Destroy an FDI context

## FDICreate

### Usage

```
HFCI DIAMONDAPI FDICreate(
    PFNALLOC          pfnalloc,
    PFNFREE           pfnfree,
    PFNOPEN           pfnopen,
    PFNREAD           pfnread,
    PFNWRITE          pfnwrite,
    PFNCLOSE          pfnclose,
    PFNSEEK           pfnseek,
    int               cpuType,
    PERF              perf
)
```

### Parameters

<i>pfnalloc</i>	Memory allocation function
<i>pfnfree</i>	Memory free function
<i>pfnopen</i>	File open function
<i>pfnread</i>	File read function
<i>pfnwrite</i>	File write function
<i>pfnclose</i>	File close function
<i>pfnseek</i>	File seek function
<i>cpuType</i>	Type of CPU
<i>perf</i>	Pointer to an error structure

### Description

The **FDICreate** API creates an FDI context that is passed to other FDI APIs.

The *pfnalloc* and *pfnfree* parameters should point to memory allocation and memory free functions which will be called by FDI to allocate and free memory. These two functions take parameters identical to the standard C malloc and free functions.

The *pfnopen*, *pfnread*, *pfnwrite*, *pfnclose*, and *pfnseek* parameters should point to functions which perform file open, file read, file write, file close, and file seek operations respectively. These functions should accept parameters identical to those for the standard *\_open*, *\_read*, *\_write*, *\_close*, and *\_lseek* functions, and should likewise have identical return codes. Note that the FDI i/o functions do not take the same parameters as the FCI i/o functions.

It is not necessary for these functions to actually call *\_open* etc.; these functions could instead call *fopen*, *fread*, *fwrite*, *fclose*, and *fseek*, or *CreateFile*, *ReadFile*, *WriteFile*, *CloseHandle*, and *SetFilePointer*, etc. However, the parameters and return codes will have to be translated appropriately (e.g. the file open mode passed in to *pfnopen*).

The *cpuType* parameter should equal one of **cpu80386** (indicating that 80386 instructions may be used), **cpu80286** (indicating that only 80286 instructions may be used), or **cpuUNKNOWN** (indicating that FDI should determine the CPU type). The *cpuType* parameter is looked at only by the 16-bit version of FDI; it is ignored by the 32-bit version of FDI.



## Microsoft FCI/FDI Library Description

The *perf* parameter should point to a global or allocated ERF structure. Any errors returned by **FDICreate** or subsequent FDI APIs using the same context will cause the ERF structure to be filled out.

### Returns

If successful, a non-NULL HFDI context pointer is returned. If unsuccessful, NULL is returned, and the error structure pointed to by *perf* is filled out.

## FDIIsCabinet

### Usage

```

BOOL DIAMONDAPI FDIIsCabinet(
    HFDI          hfdi,
    int           hf,
    PFDICABINETINFO pfdici
)

```

### Parameters

<i>hfdi</i>	FDI Context pointer originally returned by <b>FDICreate</b>
<i>hf</i>	File handle returned by a call to the application's file open function
<i>pfdici</i>	Pointer to a cabinet info structure

### Description

The **FDIIsCabinet** API determines whether a given file is a cabinet, and if so, returns information about the cabinet in the provided FDICABINETINFO structure.

The *hfdi* parameter is the context pointer returned by a previous call to **FDICreate**.

The *hf* parameter must be a file handle on the file being examined. The file handle must be of the same type as those used by the file i/o functions passed to **FDICreate**.

The *pfdici* parameter should point to an FDICABINETINFO structure, which will receive the cabinet details if the file is indeed a cabinet. The fields of this structure are as follows:

The **cbCabinet** field contains the length of the cabinet file, in bytes. The **cFolders** field contains the number of folders in the cabinet. The **cFiles** field contains the total number of files in the cabinet. The **setID** field contains the set ID (an application-defined magic number) of the cabinet. The **iCabinet** field contains the number of this cabinet in the set (0 for the first cabinet, 1 for the second, and so forth). The **fReserve** field is a boolean indicating whether there is a reserved area present in the cabinet. The **hasprev** field is a boolean indicating whether this cabinet is chained to the previous cabinet, by way of having a file continued from the previous cabinet into the current one. The **hasnext** field is a boolean indicating whether this cabinet is chained to the next cabinet, by way of having a file continued from this cabinet into the next one.

### Returns

If the file is a cabinet, then TRUE is returned and the FDICABINETINFO structure is filled out. If the file is not a cabinet, or some other error occurred, then FALSE is returned. In either case, it is the responsibility of the application to close the file handle passed to this function.

## FDICopy

### Usage

```

BOOL FAR DIAMONDAPI FDICopy(
    HFDI          hfdi,
    char FAR      *pszCabinet,
    char FAR      *pszCabPath,
    int           flags,
    PFNFIDINOTIFY pfnfdin,
    PFNFIDIDECRYPT pfnfdid,
    void FAR      *pvUser
);

```

### Parameters

<i>hfdi</i>	FDI Context pointer originally returned by <b>FDICreate</b>
<i>pszCabinet</i>	Name of cabinet file, excluding path information
<i>pszCabPath</i>	File path to cabinet file
<i>flags</i>	Flags to control the extract operation
<i>pfnfdin</i>	Pointer to a notification (status update) function
<i>pfnfdid</i>	Pointer to a decryption function
<i>pvUser</i>	Application-specified value to pass to notification function

### Description

The **FDICopy** API extracts one or more files from a cabinet. Information on each file in the cabinet is passed back to the supplied pfnfdin function, at which point the application may decide to extract or not extract the file.

The *hfdi* parameter is the context pointer returned by a previous call to **FDICreate**.

The *pszCabinet* parameter should be the name of the cabinet file, excluding any path information, from which to extract files. If a file is split over multiple cabinets, **FDICopy** does allow subsequent cabinets to be opened.

The *pszCabPath* parameter should be the file path of the cabinet file (e.g. "C:\MYCABS\"). The contents of *pszCabPath* and *pszCabinet* will be strung together to create the full pathname of the cabinet.

The *flags* parameter is used to set flags for the decoder. At this time there are no flags defined, and the *flags* parameter should be set to zero.

The *pfnfdin* parameter should point to a file notification function, which will be called periodically to update the application on the status of the decoder. The *pfnfdin* function takes two parameters; **fdint**, an integral value indicating the type of notification message, and **pfdin**, a pointer to an FDINOTIFICATION structure.

The **fdint** parameter may equal one of the following values; **fdintCABINET\_INFO** (general information about the cabinet), **fdintPARTIAL\_FILE** (the first file in the cabinet is a continuation from a previous cabinet), **fdintCOPY\_FILE** (asks the application if this file should be copied), **fdintCLOSE\_FILE\_INFO** (close the file and set file attributes, date, etc.), or **fdintNEXT\_CABINET** (file continued on next cabinet).

The **pfndin** parameter will point to an **FDINOTIFICATION** structure with some or all of the fields filled out, depending on the value of the **fdint** parameter. Four of the fields are used for general data; **cb** (a long integer), and **psz1**, **psz2**, and **psz3** (pointers to strings), the meaning of which are highly dependent on the **fdint** value. The **pv** field will be the value the application originally passed in as the *pvUser* parameter to **FDICopy**.

The *pfndin* function must return a value to FDI, which tells FDI whether to continue, abort, skip a file, or perform some other operation. The values which can be returned depend on **fdint**, and are explained below.

Note that it is possible that future versions of FDI will have additional notification messages. Therefore, the application should ignore values of **fdint** it does not understand, and return zero to continue (preferably), or -1 (negative one) to abort.

If **fdint** equals **fdintCABINET\_INFO** then the following fields will be filled out; **psz1** will point to the name of the next cabinet (excluding path information); **psz2** will point to the name of the next disk; **psz3** will point to the cabinet path name; **setID** will equal the set ID of the current cabinet; and **iCabinet** will equal the cabinet number within the cabinet set (0 for the first cabinet, 1 for the second cabinet, etc.) The application should return 0 to indicate success, or -1 to indicate failure, which will abort **FDICopy**. An **fdintCABINET\_INFO** notification will be provided exactly once for each cabinet opened by **FDICopy**, including continuation cabinets opened due to files spanning cabinet boundaries.

If **fdint** equals **fdintCOPY\_FILE** then the following fields will be filled out; **psz1** will point to the name of a file in the cabinet; **cb** will equal the uncompressed size of the file; **date** will equal the file's 16-bit FAT date; **time** will equal the file's 16-bit FAT time; and **attribs** will equal the file's 16-bit FAT attributes. The application may return one of three values; 0 (zero) to skip (i.e. not copy) the file; -1 (negative one) to abort **FDICopy**; or a non-zero (and non-negative-one) file handle for the destination to which to write the file. The file handle returned must be compatible with the **PFNCLOSE** function supplied to **FDICreate**. The **fdintCOPY\_FILE** notification is called for each file that *starts* in the current cabinet, providing the opportunity for the application to request that the file be copied or skipped.

If **fdint** equals **fdintCLOSE\_FILE\_INFO** then the following fields will be filled out; **psz1** will point to the name of a file in the cabinet; **hf** will be a file handle (which originated from **fdintCOPY\_FILE**); **date** will equal the file's 16-bit FAT date; **time** will equal the file's 16-bit FAT time; **attributes** will equal the file's 16-bit FAT attributes (minus the **\_A\_EXEC** bit); and **cb** will equal either zero (0) or one (1), indicating whether the file should be executed after extract (one), or not (zero). It is the responsibility of the application to execute the file if **cb** equals one. The **fdintCLOSE\_FILE\_INFO** notification is called after all of the data has been written to a target file. The application must close the file (using the provided **hf** handle), and set the file date, time, and attributes. The application should return **TRUE** for success, or **FALSE** or -1 (negative one) to abort **FDICopy**. FDI assumes that the target file was closed, even if this callback returns failure; FDI will not attempt to use **PFNCLOSE** to close the file.

If **fdint** equals **fdintPARTIAL\_FILE** then the following fields will be filled out; **psz1** will point to the name of the file continued from a previous cabinet; **psz2** will point to the name of the cabinet on which the first segment of the file exists; **psz3** will point to the name of the disk on which the first segment of the file exists. The **fdintPARTIAL\_FILE** notification is called for files at the beginning of a cabinet which are continued from a previous cabinet. This notification will occur only when **FDICopy** is started on the second or subsequent cabinet in a series, which has files continued from a previous cabinet. The application should return zero (0) for success, or -1 (negative one) for failure, which will abort **FDICopy**.

If **fdint** equals **fdintNEXT\_CABINET** then the following fields will be filled out; **psz1** will point to the name of the next cabinet on which the current file is continued; **psz2** will point to the name of the next disk on

which the current file is continued; `psz3` will point to the cabinet path information; and `fdie` will equal a success or error value. The ***fdintNEXT\_CABINET*** notification is called only when ***fdintCOPY\_FILE*** was instructed to copy a file in the current cabinet that is continued in a subsequent cabinet. It is important that the cabinet path name, `psz3`, be validated before returning (`psz3`, which points to a 256 byte array, may be modified by the application; however, it is not permissible to modify `psz1` or `psz2`). The application should ensure that the cabinet exists and is readable before returning; if necessary, the application should issue a disk change prompt and ensure that the cabinet file exists. When this function returns to FDI, FDI will verify that the `setID` and `iCabinet` fields of the supplied cabinet match the expected values for that cabinet. If not, FDI will continue to send ***fdintNEXT\_CABINET*** notification messages with the `fdie` field set to ***FDIERROR\_WRONG\_CABINET***, until the correct cabinet file is specified, or until this function returns -1 (negative one) to abort the ***FDICopy*** call. If after returning from this function, the cabinet file is not present and readable, or has been damaged, then the `fdie` field will equal one of the following values; ***FDIERROR\_CABINET\_NOT\_FOUND***, ***FDIERROR\_NOT\_A\_CABINET***, ***FDIERROR\_UNKNOWN\_CABINET\_VERSION***, ***FDIERROR\_CORRUPT\_CABINET***, ***FDIERROR\_BAD\_COMPR\_TYPE***, ***FDIERROR\_RESERVE\_MISMATCH***, ***FDIERROR\_WRONG\_CABINET***. If there was no error, `fdie` will equal ***FDIERROR\_NONE***. The application should return 0 (zero) to indicate success, or -1 (negative one) to indicate failure, which will abort ***FDICopy***.

The `pfndid` parameter is reserved for encryption, and is currently not used by FDI. This parameter should be set to NULL.

The `pvUser` parameter should contain an application-defined value which will be passed back as a field in the ***FDINOTIFICATION*** structure of the notification function. If not required, this field may be safely set to NULL.

## Returns

If successful, TRUE is returned. If unsuccessful, FALSE is returned, and the error structure pointed to by `perf` (from ***FDICreate***) is filled out.

## FDIDestroy

### Usage

```
BOOL DIAMONDAPI FDIDestroy(  
    HFDI hfdi  
)
```

### Parameters

*hfdi* FDI context handle returned by **FDICreate**

### Description

The **FDIDestroy** API destroys an *hfdi* context, freeing any memory and temporary files associated with the context.

### Returns

If successful, TRUE is returned. If unsuccessful, FALSE is returned. The only reason for failure is that the *hfdi* passed in was not a proper context handle.