FCI/FDI Library Interface Description

Table of Contents

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.

At this time, only one instance of FCI and FDI may be active at any one time; it is *not* permissible to attempt to create two FCI contexts at once, or two FDI contexts at once, for example. However, it *is* permissible to have one FCI instance and one FDI instance active at the same time.

FCI and FDI operate using the technique of function callbacks; many of the FCI and FDI APIs require parameters which are pointers to functions defined in the application. The parameters and purpose of these functions are explained fully in this document. The *fci.h* and *fdi.h* header files provide declaration macros which should be used to declare these callback functions. These macros 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 the *fci.h* and *fdi.h* header files:

```
#define FNALLOC(fn) void HUGE * FAR DIAMONDAPI fn(ULONG cb)
#define FNFREE(fn) void FAR DIAMONDAPI fn(void HUGE *pv)
```

These declarations can be used as follows:

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

FNFREE(mem_free)
{
      return free(pv);
}

some_function()
{
      hfci = FCICreate(
            &erf,
            filedest,
            memalloc,
            memfree,
            gettempfile,
            &ccab
      );
      . . .
}
```

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:

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

FCICreate

Usage

```
HFCI DIAMONDAPI FCICreate(
      PERF              perf,
      PFNFCIFILEPLACED  pfnfcifp,
      PFNALLOC          pfnalloc,
      PFNFREE           pfnfree,
      PFNOPEN           pfnopen,
      PFNREAD           pfnread,
      PFNWRITE          pfnwrite,
      PFNCLOSE          pfnclose,
      PFNSEEK           pfnseek,
      PFNFCIGETTEMPFILE pfnfcigtf,
      PCCAB             pccab
);
```

Parameters

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

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*, 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.

The *pfntemp* parameter should point to a function which returns the name of a suitable temporary file. Two parameters will be passed to this function; pszTempName, an area of memory to store the filename, and cbTempName, the size of the memory area. The filename returned by this function should not occupy more than this number of 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 finalised. 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 a client context value, pv, is also passed as a parameter; this will be the same value that was passed in to the **FCIAddFile** API, and can be used for any purpose. Typically it is used by the application to store internal context information.

The *pccab* parameter should point to an initialised 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 initialised 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\”).

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,
      void                *pv
)
```

Parameters

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

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. The Microsoft EXTRACT.EXE utility will do this, but in any custom extract application it is up to the application to run the file if it detects this flag.

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-specific context data originally passed to **FCIAddFile**.

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, cb1 is the compressed size of the most recently compressed block, and cb2 is the uncompressed size of the most recent block (which is generally 32K, except for the last block which may be smaller). 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; pattribs, a memory location to return FAT-style attributes; and pv, the application-specific context pointer originally passed to **FCIAddFile**. 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.

The *pv* parameter is an application-specific context handle that may be set to anything desired by the application, such as a pointer to an internal state structure. The *pv* parameter is passed as a parameter to the various callback functions described above. The *pv* parameter may be safely set to NULL if it is not required.

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,
      void                   *pv
)
```

Parameters

hfci	FCI Context pointer originally returned by **FCICreate**
fGetNextCab	Name of file to add (should include path information)
GetNextCab	Function called to obtain specifications on the next cabinet to create
pfnProgress	Progress function called to update the user
pv	Application-specific context data

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**.

The *pv* parameter is an application-specific context handle that may be set to anything desired by the application, such as a pointer to an internal state structure. The *pv* parameter will be passed as a parameter to the *pfnProgress* and *GetNextCab* functions. The *pv* parameter may be safely set to NULL if it is not required.

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,
      void                 *pv
)
```

Parameters

hfci	FCI Context pointer originally returned by **FCICreate**
GetNextCab	Function called to obtain specifications on the next cabinet to create
pfnProgress	Progress function called to update the user
pv	Application-specific context data

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**.

The *pv* parameter is an application-specific context handle that may be set to anything desired by the application, such as a pointer to an internal state structure. The *pv* parameter will be passed as a parameter to the *pfnProgress* and *GetNextCab* functions. The *pv* parameter may be safely set to NULL if it is not required.

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:

FDICreate	Create an FDI context
FDIIsCabinet	Determines whether a file is a cabinet, and returns information if so
FDICopy	Extracts files from cabinets
FDIDestroy	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

pfnalloc	Memory allocation function
pfnfree	Memory free function
pfnopen	File open function
pfnread	File read function
pfnwrite	File write function
pfnclose	File close function
pfnseek	File seek function
cpuType	Type of CPU
perf	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.

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 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.

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

hfdi	FDI Context pointer originally returned by **FDICreate**
hf	File handle returned by a call to the application's file open function
pfdici	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,
      PFNFDINOTIFY  pfnfdin,
      PFNFDIDECRYPT pfnfdid,
      void FAR     *pvUser
);
```

Parameters

hfdi	FDI Context pointer originally returned by **FDICreate**
pszCabinet	Name of cabinet file, excluding path information
pszCabPath	File path to cabinet file
flags	Flags to control the extract operation
pfnfdin	Pointer to a notification (status update) function
pfnfdid	Pointer to a decryption function
pvUser	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 pfdin 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 *pfnfdin* 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; 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. It 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.