Peregrine

Desktop Inventory Plug-in Interface Guide

For use with Desktop Inventory 7.2.0

PART NO: PDI-7.2.0-ENG-01006-00287

Peregrine SYSTEMS®

This edition applies to version 7.2.0 of the licensed program.

Peregrine Systems, Inc.
Worldwide Corporate Headquarters
3611 Valley Centre Drive San Diego, CA 92130
Tel 800.638.5231 or 858.481.5000
Fax 858.481.1751
www.peregrine.com

About this Guide

Structure of the guide

This guide consists of two chapters:

- The Overview chapter is intended for anyone with a desire to know about the capabilities of the Desktop Inventory Scanner Plug-in Interface.
- The Technical chapter is intended for IT staff that require intimate knowledge of the interface in order to implement customized plug-ins.

Contacting Peregrine Systems

For technical support on this or any other product from Peregrine Systems, Inc., refer to the Customer Support Web site at:

http://support.peregrine.com

Plug-in Interface Guide

Table of Contents

CHAPTER 1 Overview

The Desktop Inventory Scanners provide an interface for “plug-in” modules. Organizations that require additional information to be collected during the scan, or wish to develop plug-ins for reselling, can use the information in this document to assist in writing a plug-in module.

The current plug-in interface allows two different kinds of data to be collected:

- Using the *Data File Recognition* method, information can be collected from each file the scanner accesses.
- Using the *Archive Processing* method, a plug-in can implement scanning of a new archive type, recognize a file as an archive, and send data (filename, size, CRC, etc.) on each file in the archive back to the scanner.

The possibilities for collecting data for individual files on a machine are limited only by imagination – with the appropriate plug-in, it would be possible to extract keywords from documents, highlight Year 2000 problems in spreadsheets or databases, scan for possible virus infections, etc.

The archive processing can be used to implement support for one or more of the less common archive types not natively supported by the Scanners, such as SQZ. In addition, it would be possible to implement support for disk image files, which also can be considered archive files as they are files that themselves contain directories and files. With the right code available, it is also feasible to use this feature to scan message databases or other file stores using this feature.

Implementation and distribution

A plug-in consists of a number of dynamic link libraries, along with a configuration file that identifies the plug-in and the files that it comprises. Due to the fact that DOS has no support for DLLs, the DOS scanners do not support plug-ins.

A Scanner plug-in is made up of at least one but optionally several executable components:

- Scanner Generator DLL:

  This DLL is required if the plug-in needs advanced configuration not covered using the standard plug-in options of the Scanner Generator. If present, the DLL should display a dialog box displaying relevant options, initialize its controls from data obtained from the Scanner Generator, and send the modified configuration information back to the Scanner Generator.

- Scanner DLLs:

  This is the core of the plug-in, which implements the information gathering functionality of the plug-in. The DLL is initialized at scan-time with the configuration obtained from the Scanner Generator, and is then passed a reference to each file that the scanner examines. The DLL is free to do its own examination of the file, is allowed full read access to the file through scanner interface functions, and can store any information (or none) that it gathers into the generated fingerprint.
  For full platform-support, separate scanner DLLs for Win32, Win16 and OS/2 should be produced.

- Analysis DLL:

  Provides an interface to the plug-in data for those Desktop Inventory tools parsing the scan result. The interface allows the plug-in to structure the data efficiently, while allowing the data to be displayed and queried like any other data collected by the Scanners. An analysis DLL is not required for plug-ins using the *Archive Recognition* method.

All of the DLLs in question, except for the Win16 and OS/2 Scanner DLLs, must be Win32 PE DLLs.

Each of the files must be given a file name of the format "pg*NNNTTT.EXT*", where

- *NNN* is a unique plug-in ID, which is a number in the range [100 <= ID <= 999]. The plug-in ID must be unique across all installed plug-ins; to obtain a unique ID for a new plug-in, please contact Peregrine Systems, Inc. Technical Support. All files that make up the plug-in must have the same ID in the filename.

Note: Compiled DLL files have an internally hard-coded library name, which is used by the operating system to determine the module handle of the DLL. The actual name of the DLL must be the same as its internal name.

Plug-in IDs below 100 are reserved for the use of Peregrine Systems, Inc.

- *TTT* is the type of the file; any names can be used. The convention used for Desktop Inventory plug-ins is:
  - cfg: Indicates that this is a Scanner Generator DLL.
  - w16, w32 or os2: Identifies that this is a Scanner DLL, and further identifies the platform that this DLL is intended for. The platforms are Win32, Win16 and OS/2 respectively.
  - anl: Indicates that this is an Analysis DLL.
  - ini: Indicates that this is a plug-in configuration file.
- *EXT* is the filename extension.

To package a plug-in such that the Scanner Generator will be able to install it using the Install button of the plug-in configuration page, compress all of the files comprising the plug-in into a ZIP format archive with an extension of ".ZIP".

Sample distribution

A plug-in with a unique plug-in ID of 218, implementing a plug-in for collecting extra file information when using the Win32 and OS/2 scanners, and requires special setup in the Scanner Generator would consist of the following files:

- pg218cfg.dll Scanner Generator configuration DLL
- pg218w32.dll Scanner DLL for the Win32 scanner
- pg218os2.dll Scanner DLL for the OS/2 scanner

- pg218anl.dll Analysis DLL
- pg218ini.ini Plug-in configuration file used by the Scanner Generator

To distribute this plug-in, all of these files should be compressed using PKZip. The file name of the archive is irrelevant, as long as the extension used is .zip.

Using the plug in

When a plug-in is configured in the Scanner Generator, the Configuration DLL is invoked (using the interface discussed in the Technical Guide section of this document) when the user presses the Advanced button for the plug-in. If the configuration file indicates that the plug-in does not contain a Configuration DLL, the Advanced button will be grayed out.

When the Scanner Generator generates the scanner executables, all relevant Scanner DLLs are packaged inside the scanner executable itself and are extracted on demand. Even when using one or more plug-ins, the scanner consists of a single self-contained executable only.

When launching the Desktop Inventory Viewer, all Analysis DLLs available are enumerated and initialized. Any scan files (FSFs or xml.gz) containing data collected using plug-ins are processed normally, using the relevant Analysis DLL to parse the plug-in data stored. Plug-in data for which an Analysis DLL cannot be found is ignored.

In the Desktop Inventory Analysis Workbench, all Analysis DLLs are also enumerated, and the Options dialog allows the user to choose which plug-in generated data should be loaded, if any. As for the Viewer, all scans containing any of the plug-in data selected for loading is parsed using the Analysis DLLs and is available for detailed analysis. Plug-in data for which an Analysis DLL cannot be found is ignored.

Files included with this SDK

This Software Development Kit includes several files demonstrating how to write a plug-in using either MS Visual Studio (C source code) or Borland Delphi (Pascal source code).

The source code included was developed and tested using Microsoft Visual Studio v5.0 and Borland Delphi v3.02.

The plug-in is given an ID of 13, and contains a Configuration DLL, a Scanner DLL for Win32 only, and an Analysis DLL. The plug-in, which uses the Data File Recognition method, extracts and stores the first 4 bytes of every file scanned.

Directory	File Name	Purpose
sdk\plugin\1.0	Plugin Interface.pdf	This document
sdk\plugin\1.0\c	fp_def.h	Header file defining Desktop Inventory data type aliases
	fpplgint.h	Header file defining the Desktop Inventory Plug-in Interface
	pg13cnst.h	Header file defining constants specific to this plug-in
sdk\plugin\1.0\c\pg013cfg	pg013cfg.h	Header file defining the exported functions of the Configuration DLL
	pg013cfg.cpp	C source code for the Configuration DLL
	pg013cfg.def	Linker definition file for the Configuration DLL
	pg013cfg.res	Resource file containing dialog design for the Configuration DLL
sdk\plugin\1.0\c\pg013w32	pg013w32.h	Header file defining the exported functions and Instance Data structure of the Scanner DLL
	pg013w32.cpp	C source code for the Scanner DLL
	pg013w32.def	Linker definition file for the Scanner DLL
sdk\plugin\1.0\c\pg013anl	pg013anl.h	Header file defining the exported functions and Instance Data structure of the Analysis DLL
	pg013anl.cpp	C source code for the Analysis DLL

	pg013anl.def	Linker definition file for the Analysis DLL
sdk\plugin\1.0\pascal	use32.pas	Unit redefining various Integer types based on target platform and compiler
	fpplgint.pas	Unit defining the Desktop Inventory Plug-in Interface
sdk\plugin\1.0\pascal\pg013	pg13cnst.pas	Unit with constant definitions specific to this plug-in
	pg013cfg.dpr	Delphi source file for the Configuration DLL
	pg013cfg.r32	Resource file containing dialog design for the Configuration DLL
	pg013w32.dpr	Delphi source file for the Scanner DLL
	pg013anl.dpr	Delphi source file for the Analysis DLL
sdk\plugin\1.0\dist	pg013c.zip	Sample plug-in distribution archive, based on output from the C source code
	pg013pas.zip	Sample plug-in distribution archive, based on output from the Pascal source code

Configuration file format

The plug-in configuration file describes the version information, components and default configuration of the plug-in. The file, which is used by the Scanner Generator, is a Windows INI file, with the following sections, of which only the Options section is mandatory:

- Options: Describes version information and specifies the default configuration of the plug-in.
- <*platform*>ScannerFiles: Indicates the name of the Scanner DLL and any data files required for the Scanner plug-in for the platform <*platform*>.
- CfgFiles: Indicates the name of the scanner generator DLL.
- AnalysisFiles: Indicates the name of the analysis DLL.

Options

The options section of the configuration file can contain the following key values:

- Name: The name of the plug-in.
- Description: A brief description of the plug-in.
- ID: The ID of the plug-in.
- LimitByName, LimitBySize: Default configuration information. These are boolean values. A ‘0’ indicates ‘false’, a ‘1’ indicates ‘true’.
- MinFileSize, MaxFileSize: Default configuration information. These values are only pertinent if ‘LimitBySize’ is set to true. These are numerical integer values, in bytes.
- IncludeFileMask, ExcludeFileMask: Default configuration information. These keys are only pertinent if ‘LimitByName’ is set to true. These values are unquoted, text values containing comma-delimited lists of file-masks.

<platform>ScannerFiles

These sections must contain at least one key:

```
<filenameN>= Description of file
```

Valid values for <*platform*> are Win16, Win32 and OS2. The filenames listed must start with the name of the Scanner DLL for the platform. Following this, the names and descriptions of any data files used by the Scanner DLL for the platform can be specified, if any are required.

CfgFiles

This section contains must contain at least one key:

```
<CfgFileName>= Description of Configuration DLL.
```

If the Configuration DLL requires additional files, these should be listed in this section as well.

AnalysisFiles

This section must contain at least one key:

```
<AnalysisFileName> = Description of Analysis plug-in DLL.
```

If the Analysis DLL requires additional files, these should be listed in this section as well.

Example

The content of the configuration file for the hypothetical plug-in mentioned above, pg218ini.ini, could be:

```
[Options]
Name=Sample
Description=Hypothetical Plug-in for Scanners
ID=218
LimitByName=1
IncludeFileMask=*.EXE;*.COM
ExcludeFileMask=WIN*.*
LimitBySize=1
MinFileSize=1000
MaxFileSize=100000

[Win32ScannerFiles]
pg218w32.dll=The hypothetical Win32 Scanner DLL

[Os2ScannerFiles]
pg218os2.dll=The hypothetical OS/2 Scanner DLL

[CfgFiles]
pg218cfg.dll=The hypothetical Configuration DLL

[AnalysisFiles]
pg218anl.dll=The hypothetical Analysis DLL
```

API Overview

The table below lists the names of all entry points relevant to plug-ins, where they are or should be defined, and how they can be used. Please refer to the Technical Guide chapter for detailed information on each API, or refer to the source code files defining the interface (fp_def.h, fpPlgInt.h for C programmers and fpPlgInt.pas for Pascal programmers):

API Name	Defined in	Usage
PlgGetAPIVersion	Configuration DLL Scanner DLLs Analysis DLL	Called by Desktop Inventory components to verify that the plug-in is compatible with the Desktop Inventory software used.
PlgConfigure	Configuration DLL	Allows the DLL to call and interface to the Scanner Generator functions.
Int->OptSetValue	Scanner Generator	Internal Scanner Generator function, the address of which is passed to the Configuration DLL. Used to set the value of a plug-in specific option.
Int->OptGetValue	Scanner Generator Scanners Analysis DLL	Internal Scanner Generator, Scanner and Analysis function. Used to retrieve the value of a plug-in specific option.
PlgInit	Scanner DLLs	Called by the Scanner at start-up. During this call, the DLL is expected to initialize any required internal data structures, as well as return information about itself to the scanner.
PlgRecogniseDataFile	Scanner DLLs	Called by the Scanner (for Data File Recognition plug-ins) for each file the plug-in has been set up to process.
PlgIsArchive	Scanner DLLs	Called by the Scanner (for Archive Processing) for each file the plug-in has been set up to process.

PlgFindFileInArchive	Scanner DLLs	Called by the Scanner (for Archive Processing) several times for each file identified as a supported archive.
PlgStoreData	Scanner DLLs	Called by the Scanner when scanning is complete. During this call, the plug-in can call the `MemStore` Scanner function to store any data collected by the plug-in.
PlgDone	Scanner DLLs	Called by the Scanner prior to exiting. The plug-in should free all memory previously allocated.
ScanInt->FileSeek	Scanners	Seek to a given offset of a file.
ScanInt->FileRead	Scanners	Read a block of data from a file.
Int->MemGet	Scanners Analysis DLL	Can be used by the plug-in to dynamically allocate memory.
Int->MemFree	Scanners Analysis DLL	Frees memory allocated using `MemGet`
ScanInt->MemStore	Scanners	Store a block of data in the scan file.
PlgGetColumns	Analysis DLL	Called to retrieve a list of columns the plug-in defines.
PlgNewData	Analysis DLL	Called when a new scan file is about to be loaded, allowing the plug-in data block to be retrieved from the scan file.
PlgGetColumnData	Analysis DLL	Called to retrieve plug-in data for a specific column for a given file.
PlgFreeData	Analysis DLL	Called when all data from the current scan file has been retrieved; memory allocated can be freed.
AnalysisInt->MemLoad	Desktop Inventory tool	Load a block of data previously saved by the Scanner DLL from the scan file.

2 Technical Reference

CHAPTER

Returning API version information

Every DLL component of a plug-in must export a function similar to the following:

```
unsigned short FPCALLCONV PlgGetAPIVersion()
{
return PlgAPIMinorVer + (PlgAPIMajorVer << 8);
}
```

The various Desktop Inventory applications will query the component in order to find out what version of the plug-in API it supports. In the current implementation, this function must return a major version of 1 and a minor version of 0.

Scanner Generator Interface – Configuration DLL

The Scanner Generator is able to store advanced options for a plug-in. The format of the advanced options data is completely arbitrary, and at the sole discretion of the author of the plug-in. The data is communicated as pairs of NULL terminated strings, one for the name of the option, and one for its value.

The Scanner Generator will only store an option if it actually contains data. The following statement has no effect:

```
_SGInt->OptSetValue("MyOption", "");
```

Similarly, if a non-existing option is queried, it will return NULL (no data).

The interface between the Configuration DLL and the Scanner Generator works as follows:

- The Scanner Generator invokes the Configuration DLL by calling its single exported function, declared as:

  ```
  void FPCALLCONV PlgConfigure(const TPlgSGInterface* _SGInt);
  ```

  The `_SGInt` parameter is a data structure containing pointers to the SG Interface functions. Refer to the file "`fpPlgInt.h`" for the full structure of this item.

- `PlgConfigure` should then display a dialogue box, initializing its controls with data obtained from the Scanner Generator via the function

  ```
  _SGInt->OptGetValue
  ```

  which is declared as:

  ```
  void (FPCALLCONV *OptGetValue)(const char* Name, char* Value);
  ```

- The user then configures the plug-in via this dialogue box. When the configuration is completed, the DLL can send the new configuration back to the Scanner Generator via the function:

  ```
  _SGInt->OptSetValue
  ```

  which is declared as:

  ```
  void (FPCALLCONV *OptSetValue)(const char* Name, char* Value);
  ```

  For plug-ins that feature a Configuration DLL, the user may not have used the Advanced button when configuring the plug-in in the Scanner Generator. In this case, no advanced configuration data is stored and the Scanner DLLs must be able to assume reasonable defaults in this case. Only settings differing from the default should be saved using the `_SGInt->OptSetValue` function.

Collecting information on a file by file basis (Data File Recognition)

Scanner Interface – Data File Recognition

The Scanner DLL will be given an opportunity to examine every file that the scanner examines. The DLL can extract data from these files, and store it in data structures internal to the DLL. For each file, the DLL passes a unique index back to the scanner. The scanner stores this index, along with the plug-in ID, with the file's data. At the end of the scanning process, the DLL will be given an opportunity to write the data that it has collected into the fingerprint file. The Scanner Generator treats this information as a blob, writing it 'as-is' into the scan file.

At analysis time, the information is read from the fingerprint and passed to the Analysis DLL as a blob, exactly as it was written to the scan file. When the plug-in data for a specific file is needed, the Analysis DLL is passed the indices that were generated by the Scanner DLL and is expected to extract the pertinent data from the blob and pass it back to the caller.

The interface between the scanner and the Scanner DLL works as follows:

- The scanner calls a function of the DLL, where any initialization required by the DLL can be performed:

  ```
  void FPCALLCONV PlgInit(const TPlgScannerInterface *ScanInt, r
  TPlgScanInfo Info)
  ```

  When this function is called, the DLL is also required to send return information to the scanner. This information is returned via the `Info` structure, defined as follows:

  ```
  typedef struct _TplgScanInfo
  {
  long  Id;          /* Unique ID of the plug-in        */
  long  Flags;        /* Bitmapped field                 */
  char* Description;  /* Description of the plug-in      */
  void* InstanceData; /* Per-instance data of the plug-in */
  long  Reserved[4];
  } TPlgScanInfo;
  ```

  The plug-in ID is sent back via the `Id` member.

  The plug-in ID is sent back via the `Id` member.

The `Flags` member is a bitmapped value used to indicate the type of plug-in, that is, a Data File Recognition plug-in, or an Archive plug-in.

A brief description of the plug-in is sent back via the `Description` member. This description will be displayed in the 'Messages' text box on the software page of the scanner.

The `InstanceData` member is used to store a reference to any internal data that the DLL allocates. This `InstanceData` pointer is passed to every subsequent function call, so that the DLL always has access to its allocated data. Allocating data and storing the references to it in variables that are global to the DLL should be avoided, as this could lead to conflicts in a multi-threading environment, or when using a DLL instancing the Data segment only once.

Along with the references to its own internal data, the Scanner DLL must ensure that `InstanceData` points to a structure that also stores a reference to `ScanInt`, as this pointer is not passed to any further functions. `ScanInt` is a structure that contains pointers to all of the internal Scanner functions necessary to seek and read in files, extract advanced configuration options and write information to the scan file. For the exact structure of this item, see the header file "`fpPlgInt.h`".

- For each file read by the scanner, matching the criteria specified in the Scanner Generator plug-in configuration (File size and name restrictions), it invokes the DLL's exported function

```
FP_BYTE FPCALLCONV PlgRecogniseDataFile(void* InstData,
                        const TPlgFileInfo *DataFile,
                        TPlgFileData *Data,
                        const void *_Buffer);
```

The scanner DLL can now read the file using the functions provided by `ScanInt`. The file handle required by `ScanInt->FileSeek` and `ScanInt->FileRead` can be obtained from `DataFile->Handle`. Any information collected can be stored in the DLL's internal data structures, and an identifying index passed back via `Data->DataID`. Other useful information about the file can be found in the `DataFile` structure.

If the plug-in has data to store for the file, the function should return `1`. If not, it should return a value of `0`, in which case the value of `Data->DataID` is ignored.

The first 8192 bytes of the file have already been read by the scanner, and are cached in a buffer pointed to by the _Buffer parameter. When at all possible, this buffer should be used instead of the file reading functions, for example, for signature scanning. This could potentially save one read per scanned file, which amounts to thousands of reads over the course of the scan, a significant time saving.

- When the scan is complete, the Scanner will invoke the DLL's exported function

  ```
  void FPCALLCONV PlgStoreData(void *InstData);
  ```

  Using the function `ScanInt->MemStore`, the DLL can now write its data to the scan file. `MemStore` may be called more than once if necessary, for example, to store blocks larger than 64kb in a Win16 environment. If `MemStore` is called multiple times, the individual data blocks will be stored sequentially, right behind each other, so that the contiguity of the blob is preserved. It is advised that the data collected by the DLL be stored in as tight a format as possible, to minimize the size of the scan file.

- Finally, the scanner will invoke the function

  ```
  void FPCALLCONV PlgDone(void *InstData);
  ```

  During this function, the DLL should de-allocate any memory that it has allocated.

Analysis Interface – Data File Recognition

When Desktop Inventory tools read scan files containing plug-in data, the Analysis DLLs are used to parse the data. On startup, the Analysis DLL is called to get information about the number and type of 'columns' required to display its data. This function is called only once for each Analysis DLL.

Each time a new scan file with plug-in data is read, the Analysis DLL is notified in order for it to initialize its data based on the data blob stored in the fingerprint by the plug-ins Scanner DLL. To do this, the Analysis DLL is supplied with the addresses of a set of entry points that allow the data block to be read from the scan file, plug-in options to be queried, etc.

As software data is read from the scan file, the Analysis DLL is called for each file of interest, passing the file index generated by the Scanner DLL along with a plug-in column identifier. When this occurs, the DLL should extract the pertinent data from the its data structures and pass it back in the format specified. The only format currently supported by Desktop Inventory is null-terminated string. Note that not all files or columns may be queried, and that the order in which the data is retrieved is indeterminate.

When all software data has been read, the Analysis DLL may be asked to free memory used by internal data structures. Whether this function is called or not, the Analysis DLL should be ready to for the next scan file and be able to handle the situation where plug-in data from several scan file is requested in any order.

- The columns defined by the plug-in are queried via the function

  ```
  void FPCALLCONV PlgGetColumns(FPULONG ID, FP_ULONG* Count,
                          TPlgColumnStruc** Columns);
  ```

  The DLL passes back a pointer to the first element in an array of `TplgColumnStruc` structures. For the format of this structure, see the file "`fpPlgInt.h`".

- The initialization call of the DLL takes the form

  ```
  void FPCALLCONV PlgNewData(FP_ULONG ID,
                      const TPlgAnalysisInterface* AnalysisInt,
                      TPlgAnalysisInfo* Info);
  ```

  The `AnalysisInt` member contains pointers to the functions used to read the blob from the scan file, as well as functions that permit the analysis DLL to obtain the plug-in configuration data from the scan file. The DLL should read the blob back from the scan file, and construct data structures that permit it to extract the correct information depending on the file index it is passed. A reference to internal data structures should again be stored in the `Info->InstanceData` pointer.

  This function is called for every scan file to be processed, so the Analysis DLL should allocate separate Instance data for each scan file.

- The Analysis tool requests data from the plug-in on a per file/per column basis. The function

  ```
  void FPCALLCONV PlgGetColumnData(void *InstData,
                          FP_ULONG DataID,
                          FP_ULONG Column,
                          TPlgColumnData* Data);
  ```

requests data from the DLL for the file identified by `DataID`, and the column identified by `Column`. The pertinent data should be passed back via one of the members of the TplgColumnData union.

- After all of the files have been read from an scan file, the DLL may be asked to free all of its internal data structures for that scan file via the function

  ```
  void FPCALLCONV PlgFreeData(void *InstData);
  ```

Collecting information from the new Archive formats

A plug-in that provides support for new archive formats requires no Analysis DLL. The files that it reports the existence of are included with the rest of the files in the generated fingerprint. This section deals only with the scanning portion of this type of plug-in, as the configuration section is identical to that of a Data File Recognition plug-in.

The general logic of the interface is as follows:

- The scanner invokes the DLL with a call to

  ```
  void FPCALLCONV PlgInit(const TPlgScannerInterface *ScanInt,
              TPlgScanInfo *Info);
  ```

  The DLL is initialized in the same way that a Data File Recognition Scanner DLL is initialized. However, the DLL sends a flag back to the scanner to indicate that it is an archive scanning DLL (`spfArchive`, instead of `spfDataFileRecognition`).

- The Scanner DLL is given a reference to each file as the scanner examines it via a call to

  ```
  FP_BYTE FPCALLCONV PlgIsArchive(void *InstData,
                const TPlgFileInfo* ArcFile,
                const void *Buffer);
  ```

  During this call, the Scanner DLL can now examine the file to determine if it is an archive of the format(s) that the DLL is meant to recognize. In order for the check to be fast, the archive signature should be checked as this eliminates the need for reading files that are obviously not of the desired archive types. For performance reasons, the 8kb buffer pointed to by the `Buffer` parameter should be used in preference to reading the file when possible.

  The DLL must return a value of `1` if this is a readable archive, and `0` otherwise.

- If the Scanner DLL indicated that the file was indeed a readable archive, the DLL is repeatedly queried about the files in the archive until it indicates that there are no more files. The method is conceptually identical to a 'Find First…Find Next' iteration. The function that is repeatedly called is

  ```
  FP_BYTE FPCALLCONV PlgFindFileInArchive(void *InstData,
                                  const TPlgFileInfo *Archive,
                      TPlgArcFileInfo* FileInArc,
                                  FP_BYTE First);
  ```

  The DLL reports information on the files by filling in the members of the `FileInArc` parameter, and returning a value of `1`. The DLL indicates that there are no more files to report in this archive by returning a value of `0`. When this function is called for the first time, the `First` parameter is set to `1` and `0` otherwise.

- The DLL is finalized by the scanner after the scan is complete via a call to

  ```
  void FPCALLCONV PlgDone(void *InstData);
  ```

  The DLL is required to free any memory that it has allocated for its own uses.

Peregrine
SYSTEMS®

April 10, 2003