IBM Lotus Notes / Domino

Parsing and extracting content from

Embedded OLE objects

Technical Guide

Version 1.00

agecom

Author: Alex Elliott

https://www.agecom.com.au

CONTENTS

INTRODUCTION

This article provides a detailed and easily understandable description of Lotus Notes Object Linking & Embedded (OLE) objects / files. Content covered in this article includes the structure of OLE objects, how to parse them, and how to extract the usable / editable content from them.

The information and steps covered in this guide were performed using Lotus Notes version 9.0.1 running on Microsoft Windows 10. The information may or may not be applicable to other versions of Lotus Notes and platforms however should at the very least provide some guidance.

A LotusScript class has been developed for the processing of OLE objects and extracting the content from them. The class (called *CompoundFileParser*) is available from the AGECOM website along with other sample code. This class handles all the steps of extracting an OLE object from a Lotus Notes document, parsing it, and extracting the embedded content to a file that can be used. The LotusScript library containing this class may be used in your own applications. More information regarding the class and sample application can be found in this guide.

ABOUT THE AUTHOR

This article and the CompoundFileParser class was written by Alex Elliott of AGECOM.

Alex has been working with Lotus Notes since version 3 and is one of the world's experts in migration and transformation of data in and out of Lotus Notes. He also has an extensive background in programming languages such as C, Java, JavaScript, LotusScript and more, as well as extensive experience in application and network security. He has developed commercial Lotus Notes / Domino applications which are used by companies all around the world.

LOTUS NOTES OLE OBJECTS

OLE objects stored in Lotus Notes documents are a special type of attachment containing embedded data from other applications.

The content for OLE objects may either be embedded (contained within the OLE object itself) or linked (where the content is contained in an externally stored file). Linked objects all point to the same source data so any updates performed on a linked object update the centrally linked file – this also requires end users to be able to access the linked file. Embedded objects have the full content stored within the object itself (and the object is stored in the Notes document) and can therefore be updated by anyone with edit access rights to the document.

The application used to create or update the content is stored within the OLE object by way of a Class id. The class id points to the application via the Windows registry. Therefore when you edit an OLE object from within Lotus Notes the associated application launches and the content is then loaded into the application.

Lotus Notes supports the embedding of files as OLE/1 or OLE/2 objects. For platforms that support OLE the files are embedded as OLE/2 objects and for all other platforms as OLE/1 objects. If calling the EmbedObject method from Java files are stored as OLE/1 objects. OLE objects created in Lotus Notes 3.x or earlier are also stored in OLE/1 format.

OLE/1 objects are typically stored in Lotus Notes documents under a single $File item with a file attachment with a .ole extension (example: ole12345.ole).

OLE/2 objects are typically stored in Lotus Notes documents under multiple $File items – one $File item containing a file with a prefix of ‘EXT’ (eg. EXT82641) and one or more other $File items containing files with a prefix of ‘STG’ (eg. STG45298). There is also an additional $OLEOBJINFO item containing information about the OLE object.

This document is primarily concerned with OLE/2 objects. The CompoundFileParser class detects if the object being processed is an OLE/1 or OLE/2 file and will process it accordingly.

The structure of Lotus Notes OLE objects is based upon the Microsoft Compound Binary File Format (also known as the Microsoft Compound Document File Format). This format clearly defines how the data in an OLE file should be organised. The format is very similar to the Windows filesystem whereby you have a top-level Header sector, file allocation tables (FAT), directory entries, files, and finally the content within those files.

There are two versions of the Compound Document File structure – version 3 and version 4. Version 3 files have a maximum file size of 2 gigabytes and version 4 can grow to as large as 16 terabytes.

HOW OLE OBJECTS ARE STORED IN LOTUS NOTES

OLE objects are stored in Lotus Notes documents as outlined below

OLE/1 OBJECTS

When an OLE/1 object is created in a Lotus Notes document a filename is assigned to the object in the format of 'OLE12345.OLE'. The numeric part of the filename is randomly generated.

For each OLE/1 object stored in a Notes document the following actions are performed:

- The rich text field contains an icon for the object and a pointer to the single $FILE item which contains the attachment content.
- An item named '*$FILE*' is created for the OLE object (pointed to by the rich text item).

OLE/2 OBJECTS

When an OLE/2 object is created in a Lotus Notes document a filename is assigned to the object in the format of 'EXT12345'. The filename always begins with 'EXT' with the numeric part of the name being randomly generated.

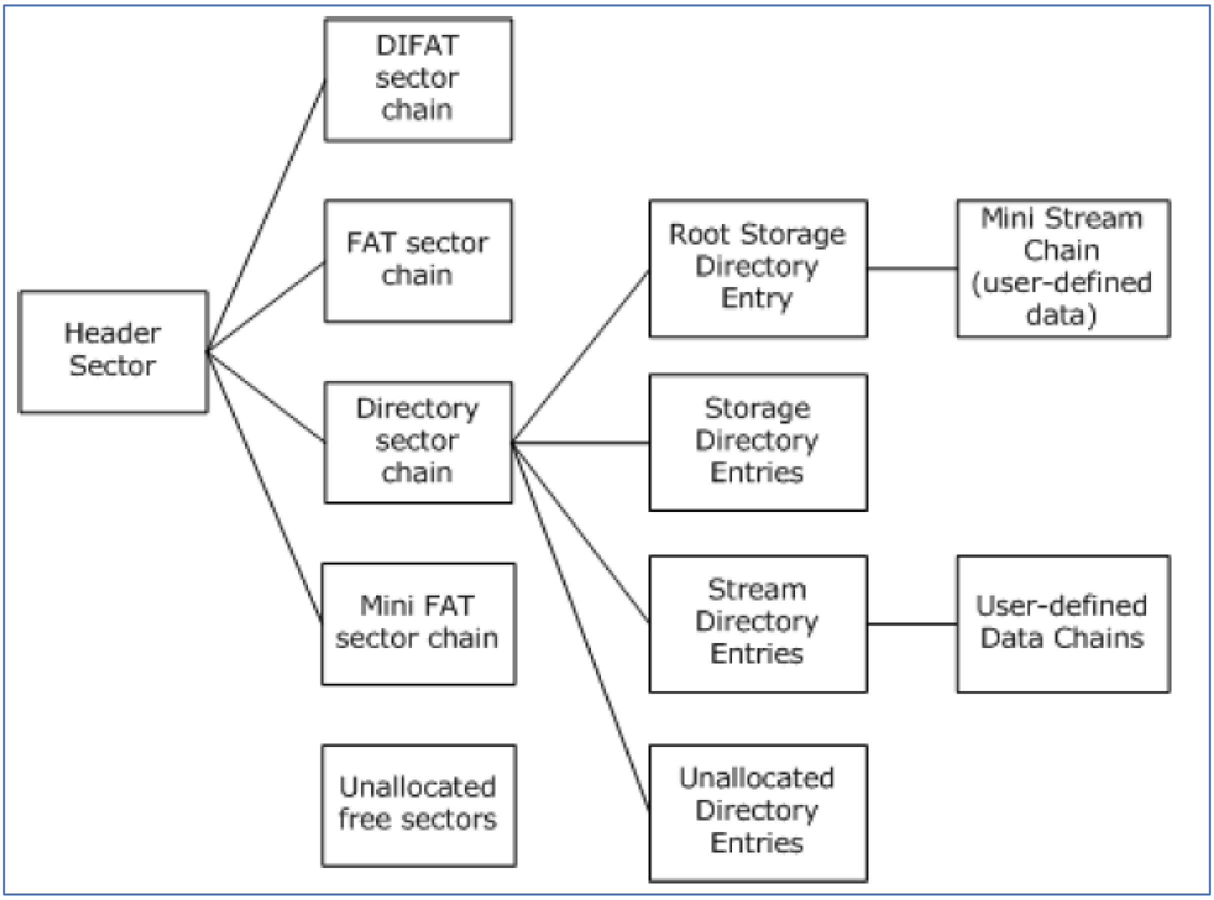
For each OLE/2 object stored in a Notes document the following actions are performed:

- The rich text field contains an icon for the object and a pointer to the master OLE file object ($FILE item which has an attachment name in the format of EXT12345).
- An item named '*$OLEOBJINFO*' is created. One of these items is created for each OLE object embedded in the document. This item enables quicker access to OLE objects without having to parse each rich text item looking for OLE objects.
- An item named '*$FILE*' is created for the master file pointer for the OLE object (pointed to by the rich text item). This item in turn contains the names of other $FILE items which contain the full content of the OLE object (see following point)
- One or more additional $FILE items are created which contain the complete data for the OLE object. The name of the attachment in each item is in the format of STG12345 (where STG is an abbreviation for Storage Format). Each $FILE item is about 65Kb in size.

THE STRUCTURE OF AN OLE/2 OBJECT

As mentioned previously Lotus Notes OLE/2 objects are based on the Microsoft Compound File structure. All data within the file is stored in Sectors. The size of each sector varies according to the version of the compound file. For version 3 the sector size is 512 bytes and for version 4 it is 4,096 bytes.

The basic structure of compound file is shown in the below diagram.



HEADER SECTOR

The header sector begins at the first byte of the file and the content of the header occupies 512 bytes. If the compound file is version 4 the header sector is padded out to 4,096 bytes with zeros. The settings found in the header sector are:

- **Signature** – A specific sequence of bytes that identifies the file as being a compound structure file. (8 bytes)
- **CLSID** – Not used and must be set to all zeros. (16 bytes)
- **Minor Version** – The minor (point) version of the file. Should be set to 0x003E if the major version of the file is either 3 or 4. (2 bytes)
- **Major Version** – The major version of the file. Must be either 3 or 4. (2 bytes)
- **Byte Order** – Indicates the byte order for all integer fields. Must be set to 0xFFFE which indicates Little-Endian order. (2 bytes)
- **Sector Shift** – Specifies the size of sectors in the file as a power of 2. If the major version is 3 then this should be 9 (0x0009) and if the major version is 4 this should be 12 (0x000C). For version 3, 2 to the power of 9 is 512 (indicating each sector is 512 bytes) and for version 4, 2 to the power of 12 is 4,096 (indicating each sector is 4,096 bytes). (2 bytes)
- **Mini Sector Shift** – Specifies the size of sectors in the mini stream. This should be 6 (0x006). 2 to the power of 6 is 64 indicating each sector is 64 bytes. (2 bytes)
- **Reserved** – not used. (6 bytes)
- **Number of Directory Sectors** – Indicates the number of directory sectors in the compound file. This field is not applicable to version 3 files in which case it should be set to all zeros. (4 bytes)
- **Number of FAT Sectors** – The number of FAT Sectors in the file. (4 bytes)
- **First Directory Sector Location** – The starting sector number for the directory stream. (4 bytes)
- **Transaction Signature Number** - This integer field MAY contain a sequence number that is incremented every time the compound file is saved by an implementation that supports file transactions. Will be set to all zeroes if file transactions are not implemented. (4 bytes)
- **Mini Stream Cutoff Size** - Specifies the maximum size of a user-defined data stream that is allocated from the mini FAT and mini stream, and that cutoff is 4,096 bytes (0x00001000). Any user-defined data stream that is greater than or equal to this cutoff size must be allocated as normal sectors from the FAT. (4 bytes)
- **First Mini FAT Sector Location** – The starting sector number for the mini FAT. (4 bytes)
- **Number of Mini FAT Sectors** – The number of mini FAT sectors in the file. (4 bytes)
- **First DIFAT Sector Location** – The starting sector number for the double-indirect file allocation table (DIFAT) – which is basically a directory for navigating FAT sectors. This is only applicable if the file is 6.875Mb or larger. (4 bytes)
- **Number of DIFAT Sectors** – The count of the number of DIFAT sectors in the file. (4 bytes)
- **DIFAT** - An array of 32-bit integer fields contains the first 109 FAT sector locations of the compound file. (436 bytes)

DIFAT SECTOR CHAIN

The DIFAT (double-indirect file allocation table) is a sector that contains an array of integers (up to 109 integers) which point to sectors that have been allocated for the FAT (file allocation table).

For example the first 4 numbers retrieved from this sector might be: 3, 5, 6, 10. When parsing the file we would first read the data from sector 3, then sector 5, then 6, and then 10. For each FAT sector all fields are traversed to build up a map of location (sector) sequences for various content within the file.

For files that contain more than 109 sectors (files which are 6.875Mb or larger) the DIFAT is also stored in DIFAT sectors. The starting DIFAT sector is obtained from the header. Each field in the DIFAT sector represents a FAT sector to navigate to for the file content. If there are no further FAT sectors to navigate to the field will be set to '*Free sector*'.

The last field in the DIFAT sector points to the next DIFAT sector to navigate to or will be set to '*End Of Chain*' if there are no further DIFAT sectors.

FAT SECTOR CHAIN

For each FAT sector that is obtained from the DIFAT Sector Chain the following operations are performed:

- **Iterate over all fields within the FAT sector**

  Each field occupies 4 bytes therefore the number of fields in each sector is determined by dividing the sector size by 4. For a version 3 file this is 512 divided by 4 which equals 128. For a version 4 file this is 4,096 divided by 4 which equals 1,024.

- **For each field record the value**

  This value will be one of the following values:

  - Free Sector
  - FAT Sector
  - Next Sector Number
  - End of Chain.

  The main value we are interested in is 'Next Sector Number'. Since content within the file can be spread across multiple sectors by building up the chain of 'Next Sectors' it gives us a complete map of the sectors we need to iterate and combine to assimilate the full content.

  The 'End of Chain' value indicates where the data for the specific content in the file ends.

For a version 3 compound file each sector occupies 512 bytes. For a version 4 compound file each sector occupies 4.096 bytes.

DIRECTORY SECTOR CHAIN

This sector contains an array of Directory Entry structures. Directories contain information about the stream and storage objects in a compound file. The stream is the set of sectors containing the content we are wanting to extract. For an OLE object that contains a Microsoft Word Document the stream is the actual contents of the Word Document.

Each directory entry is identified by a non-negative number that is called the stream ID with the starting directory number being zero. The first sector of the directory sector chain MUST contain the root storage directory entry as the first directory entry.

Directory entries for Lotus Notes OLE files appear in the following order:

0. Root Entry - the root parent for child directory entries and also contains the size and starting sector for the mini stream (applicable when the content to be extracted is less than the mini-stream size of 4.096 bytes)
1. The name of this directory is usually related to the application class of the content contained within the file, for example: Word.Document.12. This directory entry contains a pointer to the 'Stream' directory entry.
2. Stream directory – this directory entry contains the starting sector number for the content that we want to extract (example Word document) and the size of the content to be extracted in bytes (applicable when the content to be extracted is equal to or greater than the mini-stream size of 4,096 bytes). The name of this directory might be 'Package' or 'Ole10Native'.
3. A directory entry called 'CompObj' which contains additional information about the content in the file. It contains the starting sector location and size in byte of this information.

There may be other directory entries but in my experience these directory entries appear in most OLE files that have been examined.

Note: if the stream directory entry is called 'Ole10Native' it indicates that the content of the stream is itself another structure that we need to parse once obtained. This is covered though later in this document.

Information contained in directory entries that are of importance are:

- **Directory Entry Name** – name of the directory
- **Directory Type** – can be any of the following: Unknown, Storage Object, Stream Object, Root Storage.
- **Child ID** – the id of the directory to navigate to from this directory.
- **CLSID** – the class identifier of the application that is used to open and edit the content of the OLE object.
- **Creation Time** – the date / time the content was created
- **Modified Time** – the date / time the content was last updated
- **Starting Sector Location** – the sector number to begin from for extracting the object content.
- **Stream Size** – the size in bytes of the content.

MINI FAT SECTOR CHAIN

The Mini FAT Sector Chain is similar to a standard FAT Sector Chain except each Mini FAT Sector occupies only 64 bytes. This makes the storage more efficient instead of using the larger FAT Sectors and wasting space.

Mini FAT Sectors are used when the content stored in the compound file is less than 4,096 bytes.

The starting sector number that contains the Mini FAT sectors is obtained from the header of the file.

The starting Mini FAT sector (within the total Mini FAT sector chain) is obtained from the Root Storage directory entry.

USER DEFINED DATA CHAINS

User defined data chains are additional areas within the compound file mainly used by the vendor (ie. IBM / Lotus Notes Client) to store vendor-specific data. We do not need to worry about these for extracting the content.

SUMMARY

Now that the structure of a compound file has been explained we can summarise the steps required to parse the file and extract the content:

- Open the file in binary format
- Read the header (first 512 bytes of the file) and get the settings for the structure of the file
- Read all sectors
- Parse all FAT sectors as specified in the double-indirect file allocation table (DIFAT)
- Parse the directory entries
- Parse any Mini FAT sectors
- Build the FAT sector and Mini FAT sector chains
- Locate the directory entry for the stream data and get the starting sector, and stream size
- Retrieve the stream data
- Process the retrieved stream data as required (eg. write it out to a file).

The above steps are described in more detail in this guide.

EXTRACTING AN OLE/1 OBJECT

The easiest way that has been found to extract the content from OLE/1 objects is to use the NotesDXLExporter class.

The steps are as follows:

- Use the NotesDXLExporter class to get the DXL code for the document containing the OLE object and stream the output to the NotesDOMParser class.
- Get the NotesDOMDocumentNode property from the NotesDOMParser class
- Iterate over all items in the NotesDOMDocumentNode object looking for $FILE nodes.
- For each $FILE node check if the 'name' attribute matches the ole object name.
- When the matching $FILE node has been found get the child 'filedata' node and get all the data under the node. This is the binary data for the ole object encoded in base64 format.
- Create 2 NotesStream objects – one for output (opening a new file object as binary – this will be a temporary file so should be given a temporary filename) and one for input.
- Write the retrieved data to the input stream (using the WriteText method)
- Create a new NotesDocument object and create a NotesMIMEEntity item in the document
- Call the SetContentFromBytes method in the NotesMIMEEntity item and set the input stream object as the input.
- Call the GetContentFromBytes method in the NotesMIMEEntity item and set the output stream object as the output.
- Call the DecodeContent method in the NotesMIMEEntity item.
- Close the input stream object.
- Create a new NotesStream object and open a new file object for binary output. This will be the target file with the extracted content so should be given the preferred final filename.
- Read through the original output stream byte by byte looking for the byte which has value 'D0' (208 in decimal). This byte indicates where the start of the ole content begins. Write this byte to the target output stream.
- Write the rest of the data in the original output stream to the target output stream.
- Close the original output stream and close the target output stream.
- Delete the temporary file created from the original output stream.

That's it! The file that has been created contains the extracted content and can be used as a normal file.

The above processing is handled for you by the *CompoundFileParser* class available on the AGECOM website.

EXTRACTING AN OLE/2 OBJECT

The first step in extracting the content from the OLE/2 object is to extract it from the Lotus Notes document and save it to the file system. We can easily do this by making a call to a Lotus C API function called '*NSFNoteExtractOLE2Object*' from LotusScript

The API function call requires the following arguments to be passed to it:

- A handle to the Lotus Notes document containing the OLE object
- The internal name of the OLE object
- The target folder / filename to save the OLE object to
- The encryption key for the document
- A Boolean value to indicate if any existing target file should be overwritten
- Additional flags (currently not used)

The function can be declared in LotusScript as:

```
Const LIB_W32 = "nnotes.dll"
```

```
Declare Function W32_NSFNoteExtractOLE2Object Lib LIB_W32 Alias "NSFNoteExtractOLE2Object"
(ByVal hNote As Long, ByVal pszObjectName As LMBCS String, ByVal pszFileName As LMBCS String,
pEncryptionKey As ENCRYPTION_KEY, ByVal fOverwrite As Long, ByVal dwFlags As Long) As Integer
```

We also need to declare the ENCRYPTION_KEY object as such:

```
Type ENCRYPTION_KEY
	Byte1 As Byte
	Word1 As Integer
	Text (16-1) As Byte
End Type
```

The following sample code demonstrates calling the C-API function to extract an OLE object with a name of ‘EXT12345’ from the selected Lotus Notes document and saving it to a file called ‘EXT12345.ole’ in the ‘C:\Temp’ folder on the filesystem.

```
Option Public
Option Declare

' Global Declarations
Const LIB_W32 = "nnotes.dll"

Type ENCRYPTION_KEY
        Byte1 As Byte
        Word1 As Integer
        Text (16-1) As Byte
End Type

Declare Function W32_NSFNoteExtractOLE2Object Lib LIB_W32 Alias "NSFNoteExtractOLE2Object" (_
  ByVal hNote As Long, _
  ByVal pszObjectName As LMBCS String, _
  ByVal pszFileName As LMBCS String, _
  pEncryptionKey As ENCRYPTION_KEY, _
  ByVal fOverwrite As Long, _
  ByVal dwFlags As Long) As Integer

Sub Initialize
        Dim Session As New NotesSession
        Dim Doc As NotesDocument
        Dim Status As Integer
        Dim Key As ENCRYPTION_KEY

        ' Error Handler
        On Error GoTo Error_Handler

        Set Doc = Session.DocumentContext

        Status = W32_NSFNoteExtractOLE2Object(Doc.Handle, "EXT12345", "c:\temp\EXT12345.ole", Key, 1, 0)

        Exit Sub

Error_Handler:
        Print "Error on line " & Trim(CStr(Erl)) & ": " & Error$ & " (" & Trim(CStr(Err)) & ")"
        Exit Sub
End Sub
```

The value returned to ‘Status’ from the API function call indicates the success or failure of the call. A value of zero indicates success. Any other value indicates the error code relating to the error that occurred in the call – you can get the description of the error message by making a call to the C-API function called ‘*OSLoadString*’ and passing the value in Status for the string to be loaded.

SPECIAL VALUES IN THE COMPOUND FILE

Special values are used in the compound file to indicate things such as types of objects / sectors, where the iteration through sectors, streams and fields end and more. These values are retrieved by reading the appropriate bytes in a sector and performing the conversion of the byte values to the corresponding decimal values.

Constant Name	Description	Binary / Decimal Value
REGSECT	Regular sector number	0x00000000 - 0xFFFFFFF9 0 - 4294967289
MAXREGSECT	Maximum regular sector number	0xFFFFFFFA 4294967290
DIFSECT	Specifies a DIFAT sector	0xFFFFFFFC 4294967292
FATSECT	Specifies a FAT sector	0xFFFFFFFD 4294967293
ENDOFCHAIN	End of a linked chain of sectors.	0xFFFFFFFE 4294967294
FREESECT	Specifies an unallocated sector in the FAT, Mini FAT, or DIFAT.	0xFFFFFFFF 4294967295
REGSID	Regular stream ID to identify the directory entry.	0x00000000 through 0xFFFFFFF9 0 – 4294967289
MAXREGSID	Maximum regular stream ID.	0xFFFFFFFA 4294967290
NOSTREAM	Terminator or empty pointer	0xFFFFFFFF 4294967295
DIRECTORY_TYPE_UNALLOCATED	Unknown or unallocated Directory	0x00 0
DIRECTORY_TYPE_STORAGE	Storage Object	0x01 1
DIRECTORY_TYPE_STREAM	Stream Object	0x02 2
DIRECTORY_TYPE_ROOT	Root Storage Object	0x05 5

The above values are referred to over the following pages in the parsing of the file.

PARSING THE OLE/2 FILE

Now that the OLE/2 object has been extracted to the filesystem we can go ahead and parse it. The aim of this is to locate the content in the file (for example an embedded Word document), extract it and save it to a separate file enabling it to be handled and edited as a normal file.

The process of parsing the file is described over the following pages in the order in which they should be performed.

OPEN THE FILE

We first open the file in binary format using the NotesStream class:

```
Dim Session As New NotesSession
Dim Stream As NotesStream

Set Stream = Session.CreateStream

If Not Stream.Open("c:\Temp\EXT12345.ole", "binary") Then
	MessageBox "The OLE Compound file could not be opened.", 0+16, "Open Compound File"
	Exit Sub
End If

If Stream.Bytes = 0 Then
	MessageBox "The OLE Compound file has no content.", 0+16, "Open Compound File"
	Exit Sub
End If
```

READ THE HEADER

The header is the first thing in the file to read (first 512 bytes of the file). This gives us important information about the structure of the file enabling it be correctly parsed:

- Signature - The first 8 bytes must match specific values to confirm the file is indeed a compound binary file.
- Version – Let's us know if the file is version 3 or version 4.
- Byte Order – Indicates the byte order for integer fields (ie. Little-Endian)
- Size of each sector – version 3 sectors are 512 bytes, and version 4 is 4,096 bytes.
- Size of each Mini sector – Normally 64 bytes
- Number of Directory Sectors – The number of directory sectors. Not applicable to version 3 files.
- Number of FAT Sectors – The number of FAT Sectors in the file.
- First Directory Sector Location – The starting sector number for the directory stream.
- Mini Stream Cutoff Size - Specifies the cutoff size for content to be stored in the mini stream. This is normally 4,096 bytes.
- First Mini FAT Sector Location – The starting sector number for the mini FAT.
- Number of Mini FAT Sectors – The number of mini FAT sectors in the file.
- First DIFAT Sector Location – The starting sector number for the double-indirect file allocation table.
- Number of DIFAT Sectors – The count of the number of DIFAT sectors in the file.
- DIFAT - An array of 32-bit integer fields contains the first 109 FAT sector locations of the compound file.

If the file is version 4 we skip over the remaining 3,584 bytes (as all sectors in version 4 are 4.096 bytes in size and the header only occupies 512 bytes).

The content retrieved for the header in a typical compound file is shown below:

- Signature: must always be - 0xD0, 0xCF, 0x11, 0xE0, 0xA1, 0xB1, 0x1A, 0xE1
- CLSID: must always be - {00000000-0000-0000-0000-000000000000}
- Major version: 3
- Minor version: 62
- Byte order: Little endian
- Sector size: 512
- Mini stream sector size: 64
- Number of FAT sectors: 178
- Number of directory sectors: 0
- First directory sector location: 2
- Transaction signature number: 0
- Mini stream cutoff size: 4,096
- Number of mini FAT sectors: 1
- First mini FAT sector location: 5
- Number of double-indirect file allocation table (DIFAT) sectors: 1
- First DIFAT location: 117

READ ALL SECTORS

After reading the header sector the remaining sectors in the file should now be read. Knowing the size of each sector (determined by information obtained from the header) we continue reading all bytes in chunks equivalent to the size of each sector.

For version 3 the sector size is 512 bytes so we continuously read chunks of 512 bytes until we reach the end of the file. We found it best to use a list element to save each chunk of bytes to. We defined the list element as a ‘Type’ that can hold the bytes and the key for each list element is the sector number starting at zero.

```
Type Sector
        Bytes As Variant
End Type

Dim Sectors List As Sector
Dim NumSectorsFound As Double

Do
        Sectors(NumSectorsFound).Bytes = Stream.Read(SectorSize)
        NumSectorsFound = NumSectorsFound + 1
Loop Until Stream.IsEOS

Call Stream.Close
```

The key for each list element is the sector number beginning at 0 (zero) where the first sector is the sector immediately following the header sector.

PARSE DIFAT SECTORS

If the *First DIFAT Sector Location* in the header is not set to ENDOFCHAIN then we need to iterate over all DIFAT sectors to retrieve the FAT sector locations which are additional to the first 109 locations that were obtained from the header.

We start by retrieving the sector specified in the *First DIFAT Sector Location* in the header. Using the sector number as the lookup key the sector is obtained from the list element of all sectors (built when all sectors were read in the previous step). The list of fields in the sector is iterated to get each FAT sector number which are then added to the original DIFAT array of the first 109 FAT sectors in the header. If the FAT sector number is set to FREESECT it is ignored and not added to the array.

The final field in a DIFAT sector gives us the sector number of the next DIFAT sector to iterate through. If the value of the final field is ENDOFCHAIN then there are no more DIFAT sectors remaining, otherwise we start the process again by looking up that sector in the list element of sectors and iterating through it.

In the example from the Header on the previous page the First DIFAT Sector Location is 117. We retrieve sector 117 and iterate over the fields the sector which gives us the following values:

```
Field 0: 116
Field 1: 118
Field 2: 119
Field 3: 120
Field 4: 121
Field 5: FREESECT
.
.
.
Field 127: ENDOFCHAIN
```

The above information tells us that in addition to the 109 sectors retrieved from the header data is also contained in sectors 116, 118, 119, 120, and 121. Any field that has a value of '*FREESECT*' is ignored. The final field in the sector indicates the next sector to iterate for more DIFAT sectors or '*ENDOFCHAIN*' if there are no further sectors that need to be checked.

We now have the full chain of FAT sectors in the compound file to iterate through to retrieve data in the file.

PARSE EACH SECTOR IN THE DIFAT

The DIFAT array contains the list of FAT sectors to iterate through for retrieving the usable data in the compound file. Any other sectors outside this list can be ignored as they're not used.

Each element in the DIFAT array contains a sector number so iterating from the beginning of the array to the end we lookup each sector in the list element built when the whole file was parsed.

We now build a new list element for the entire sector chain and add each of these sectors to it where the lookup key is an incrementing number starting at zero (0).

RETRIEVE THE DIRECTORY ENTRIES

Each storage object or stream object within a compound file is represented by a directory entry.

The different types of directory entries are:

- Root Storage Object
- Storage Object
- Stream Object
- Unknown or unallocated

The *Stream* directory entries is what we need to get a pointer to the content to be ultimately extracted from the compound file.

Each directory entry is 128 bytes in size. The number of possible directory entries is calculated by dividing the sector size by 128 (ie. number of directory entries will be 4 or 32).

Most Lotus Notes OLE objects have the following directories:

- Root Storage
- Application Class (eg. Word.Document, PowerPoint.Show, xmlfile, etc)
- CompObj
- Package
- Ole10Native

Note: The file will contain either a directory called ‘Package’ or ‘Ole10Native’ but not both.

The first directory entry must always be the Root Storage directory and is the parent for all other directory entries.

We start parsing the directory entries by retrieving the sector specified in the *First Directory Sector Location* value in the file header. Every 128 bytes in this sector represents a directory entry and this is broken down into the following information:

- **Directory Entry Name** – the name of the directory entry. (64 bytes)
- **Directory Entry Name Length** – length of the directory entry name. (2 bytes)
- **Object Type** – indicates the type of directory entry. (1 byte)
- **Color Flag** – red / black flag. Used for binary searches. (1 bytes)
- **Left Sibling** – directory id of the left child in the red / black binary tree. (4 bytes)
- **Right Sibling** – directory id of the right child in the red / black binary tree. (4 bytes)
- **Child Id** – directory id of the direct child of the directory entry. (4 bytes)
- **CLSID** - contains an object / application class GUID, if this entry is for a storage object or root storage object. For a stream object, this field MUST be set to all zeroes. (16 bytes)
- **State Bits** - contains the user-defined flags if this entry is for a storage object or root storage object. (4 bytes)
- **Creation Time** – contains the creation time for a storage object, or all zeroes to indicate that the creation time of the storage object was not recorded. (8 bytes)
- **Modified Time** – contains the modification time for a storage object, or all zeroes to indicate that the modification time of the storage object was not recorded. (8 bytes)
- **Starting Sector Location** – contains the first sector location if this is a stream object. For a root storage object, this field MUST contain the first sector of the mini stream, if the mini stream exists. For a storage object, this field MUST be set to all zeroes. (4 bytes)

- **Stream Size** – contains the size of the user-defined data if this is a stream object. For a root storage object, this field contains the size of the mini stream. For a storage object, this field MUST be set to all zeroes. (8 bytes)

The content for the directory entries for a typical compound file is shown below.

Directory 0 name: Root Entry

- Length of name: 22
- Object type: Root Storage
- Color flag: Red
- Left sibling id: None - Unallocated
- Right sibling id: None - Unallocated
- Child id: 1
- CLSID: {00000000-0000-0000-0000-000000000000}
- State bits: 00 00 00 00
- Creation time:
- Modified time: 25/1/2019 12:27:15
- Starting sector location: 6
- Stream size: 960

Directory 1 name: PowerPoint.Show.12

- Length of name: 38
- Object type: Storage
- Color flag: Black
- Left sibling id: None - Unallocated
- Right sibling id: None - Unallocated
- Child id: 3
- CLSID: {CF4F55F4-8F87-4D47-80BB-5808164BB3F8}
- State bits: 00 00 00 00
- Creation time: 25/1/2019 12:27:13
- Modified time: 25/1/2019 12:27:15
- Starting sector location: 0
- Stream size: 0

Directory 2 name: CompObj

- Length of name: 18
- Object type: Stream
- Color flag: Black
- Left sibling id: None - Unallocated
- Right sibling id: 5
- Child id: None - Unallocated
- CLSID: {00000000-0000-0000-0000-000000000000}
- State bits: 00 00 00 00
- Creation time:
- Modified time:
- Starting sector location: 13
- Stream size: 124

Directory 3 name: Package

- Length of name: 16
- Object type: Stream
- Color flag: Black
- Left sibling id: 4
- Right sibling id: 2
- Child id: None - Unallocated
- CLSID: {00000000-0000-0000-0000-000000000000}
- State bits: 00 00 00 00
- Creation time:
- Modified time:
- Starting sector location: 186
- Stream size: 11534803

PARSE THE MINI-FAT SECTORS

If the header contains a location for the first mini-FAT sector we need to retrieve and parse those sectors. Mini-FAT sectors are essentially the same as normal FAT sectors but since they take up less space they are more efficient to use for smaller files.

The mini-FAT is used to allocate space in the mini stream. The mini stream is divided into smaller, equal-length sectors, and the sector size that is used for the mini stream is specified in the header (64 bytes).

The number of sectors in each mini-FAT sector is determined by dividing the standard sector size by the mini-FAT size. For version 3 this is 512 / 64 resulting in 8, and for version 4 this is 4096 / 64 resulting in 64.

For a version 3 compound file there must be 128 fields in the mini-FAT to fill the 512 byte sector it is contained in. Therefore the number of fields for each mini-FAT sector is 128 / 8 which is 16.

For a version 4 compound file there must be 1024 fields in the mini-FAT to fill the 4096 byte sector it is contained in. Therefore the number of fields for each mini-FAT sector is 1024 / 64 which is 16.

We now have the required information to successfully parse the mini-FAT sectors. We iterate each sector in the mini-FAT and within each we iterate through each field in the sector. This gives us the full chain of sectors in the FAT containing the data that will be extracted.

EXTRACT THE CONTENT

Having obtained the data for the directories (and the sectors in the FAT / Mini-FAT) we can now go ahead and extract the content we are after.

There are some rules though about how the data is retrieved from the compound file:

- Only directories which have an object type of 'Stream' are referenced for the content. If the name of the directory though is 'CompObj' it is ignored.
- The starting sector number and the size of the content is obtained from the stream directory. If the size is less than the mini-stream cutoff size (ie. 4,096 bytes) the sector number applies to the Mini-FAT sector table otherwise it applies to the standard FAT sector table.
- If the directory name is called 'Ole10Native' this indicates the OLE file has been stored in the OLE/1 format. The OLE/1 content requires additional processing after the content has been retrieved because it basically is a filesystem within a filesystem.

The first thing to check is the size of the stream. If it's less than the stream cutoff size we need to iterate through the mini-FAT sector chain for the data otherwise we iterate through the standard FAT sector chain.

Starting at the sector specified as the starting sector location value in the stream directory we retrieve the data for that sector. Then we lookup the sector chain (or mini-FAT sector chain) for the next sector to retrieve data from. We keep doing this until the next sector to retrieve data from is set to *ENDOFCHAIN.*

We also need to keep in mind the stream size specified in the directory and ensure we only retrieve the specified number of bytes (no more and no less) otherwise we won't have the complete data package.

We now have all the data for the content to be extracted. If the stream directory name is not called *Ole10Native* we can simply write the data directly to the filesystem using the NotesStream class and writing it as binary data. You'll be able to then open the file using the application it was originally created with.

If the stream directory name is called *Ole10Native* we need to perform additional parsing of the data as it's stored in OLE/1 format. The structure of the data is as follows:

- **Total size** – total size of the data stream (4 bytes)
- **Flags 1** – unknown flags (2 bytes)
- **Label** – usually the name of the original file without the directory (read until null character found)
- **Filename** – full path / filename of the original file (read until null character found)
- **Flags 2** – unknown flags (2 bytes)
- **Unknown1Length** – length of the unknown data in the following field (1 byte)
- **Unknown1** – unknown data (#bytes of specified in previous field)
- **Unknown2** – unknown bytes (3 bytes)
- **Command** – usually the path / filename where the file was last edited (read until null character found)
- **Native data size** – the size of the actual data to be extracted (4 bytes)
- **Native data** – the actual data (length of the data is specified in the previous field).

After reading the header and obtaining the native data size we can now retrieve the data content. As before we can write the data directly to the filesystem using the NotesStream class and writing it as binary data. The file can be opened using the application it was originally created with.

SUMMARY

That completes all the steps required to parse and extract the content from OLE/1 and OLE/2 objects.

The following pages demonstrate how to use the CompoundFileParser class to perform the above processing and extract content from OLE objects.

COMPOUNDFILEPARSER CLASS

The CompoundFileParser class is a LotusScript library written by Alex Elliott of AGECOM to make the process of extracting content from OLE objects easy and quick.

The script library and sample application that accompanies it can be downloaded from the AGECOM website at: https://www.agecom.com.au/CompoundFileParser

The methods to call in the class for extracting and processing OLE objects is explained below. Both OLE/1 and OLE/2 objects are handled in the code.

****

Import the CompoundFileParser script library into the code

****

The first step in using the script library is to add it to your code:

```
Use “CompoundFileParser”
```

****

Declare an instance of the CompoundFileParser class

****

A variable needs to be declared and assigned the class name as its type:

```
Dim CFP As CompoundFileParser
```

****

Get a handle to the Notes document and embedded OLE object

****

Get a handle to the Notes document containing the OLE object then get a handle to the OLE object itself:

```
Dim Session As New NotesSession
Dim Doc As NotesDocument
Dim BodyItem As NotesRichTextItem
Dim AttachmentName As String
Dim OLEFileClass As String
Dim TargetFileName As String
Dim TempFolder As String

‘ Get the folder to save temporary files to
TempFolder = Environ(“Temp”)
If Instr(1, TempFolder, “/”) > 0 Then
    If Right(TempFolder, 1) <> “/” Then
        TempFolder = TempFolder & “/”
    End If
Else
    If Right(TempFolder, 1) <> “\” Then
        TempFolder = TempFolder & “\”
    End If
End If
```

```
Set Doc = Session.DocumentContext
Set BodyItem = Doc.GetFirstItem(“Body”)
If Not IsEmpty(BodyItem.EmbeddedObjects) Then
    Forall EmbeddedObject In BodyItem.EmbeddedObjects
      If EmbeddedObject.Type = EMBD_OBJECT Then
         ‘ We now have a handle to the embedded object
         ‘ (additional code here – see following section)
      End If
   End Forall
End If
```

****

Process the embedded OLE object

****

This code would normally be included in the above code excerpt where it says ‘*additional code here*’:

```
‘ Get the name of the embedded object
If Trim(EmbeddedObject.Source) <> “” Then
   AttachmentName = EmbeddedObject.Source
Else
   AttachmentName = EmbeddedObject.Name
End If

‘ We ignore objects starting with ‘stg’ as these are part of a series of storage objects which are retrieved through the
‘ parent object
If Lcase(Left(AttachmentName, 3)) <> “stg” Then
   ‘ Get the application class for the object
   OLEFileClass = EmbeddedObject.Class

   ‘ Generate the target filename to save the extracted content to
   TargetFileName = “C:\NotesExport\OLEObjects\” & Cstr(Doc.NoteID) & “\” & Remove_Disallowed_FileChars(GetClassFileName(OLEFileClass, AttachmentName, True))

   ‘ Make sure the filename is unique
   TargetFileName = FileSystemObject.GetUniqueFileName(TargetFileName)

   If Lcase(Right(EmbeddedObject.Name, 4)) = “.ole” Or Lcase(Right(EmbeddedObject.Source, 4)) = “.ole” Then
      ‘ This is an OLE/1 object which can be extracted using DXL
      TargetFileName = ExtractOLEObjectByDXL(Doc, AttachmentName, TargetFileName)
   Else
      ‘ This is an OLE/2 object
      FilePath = TempFolder & Remove_Disallowed_FileChars(AttachmentName)
      If LCase(Right(FilePath, 4)) <> ".ole" Then
         FilePath = FilePath & ".ole"
      End If
```

```
            ' Compound File Parser - 1. Initialize new instance of class
            Set CFP = New CompoundFileParser()

            ' Compound File Parser - 2. Extract the raw embedded OLE object to the filesystem
            FilePath = CFP.ExtractOLEObject(Doc, AttachmentName, FilePath, True)
            If FilePath <> "" Then
                ' OLE Object was successfully extracted

                ' Compound File Parser - 3. Open the extracted ole / compound file
                If CFP.OpenFile(FilePath) Then
                    ' Compound File Parser - 4. Parse the contents of the file
                    Call CFP.ParseFile()

                    If OLEFileClass = "" Then
                        ' OLE File class wasn't obtained from the embedded object.
                        ' Try to get it from the compound file
                        OLEFileClass = CFP.GetOLEFileClass()
                        If OLEFileClass <> "" Then
                            ' Update the target filename so that it has the correct extension
                            TargetFileName = TargetFolder & Remove_Disallowed_FileChars(GetClassFileName(OLEFileClass,
AttachmentName, True))
                            TargetFileName = FileSystemObject.GetUniqueFileName(TargetFileName)
                        End If
                    End If

                    ' Compound File Parser - 5. Check if the original filename was retrieved from the ole object
                    OLESourceFilename = CFP.GetOLEFileName()
                    If OLESourceFilename <> "" Then
                        ‘ Original filename was found. Change the target filename to match the original file
                        If InStr(1, OLESourceFilename, "\") > 0 Then
                            OLESourceFileName = Trim(StrRightBack(OLESourceFileName, "\"))
                        End If
                        If InStr(1, OLESourceFilename, "/") > 0 Then
                            OLESourceFileName = Trim(StrRightBack(OLESourceFileName, "/"))
                        End If

                        TargetFileName = TargetFolder & Remove_Disallowed_FileChars(GetClassFileName(OLEFileClass,
OLESourceFileName, True))

                        ' Make sure the filename is unique
                        TargetFileName = FileSystemObject.GetUniqueFileName(TargetFileName)
                    End If

                    ' Compound File Parser - 5. Extract the usable content from the OLE file and save it
                    TargetFileName = CFP.ExtractOLEStream(TargetFileName, False)
                End If

                ' Compound File Parser - 6. Remove the temporary ole file
                Call FileSystemObject.RemoveFile(FilePath)
            End If
        End If
End If
```

The end result of running the above code is the content within the embedded ole object has now been extracted and save to the filesystem. You will be able to open the target file just like any other file and access the extracted content.

Tip:

If you'd like to see the parsed contents of the OLE object in a format you can easily read you can make a call to the 'WriteFileInformation()' method of the CompoundFileParser class which will create a text file.

You must first perform the above steps to process and parse the OLE object then call this method passing it the filename to write the information to.

Example: Call CFP.WriteFileInformation("c:\Temp\ParsedOLEFile.txt")

THE SAMPLE APPLICATION

The sample application containing the Compound File Parser class demonstrates how to process OLE objects. You can download it from the AGECOM website at:
https://www.agecom.com.au/CompoundFileParser

The sample application contains several documents with a mixture of OLE/1 and OLE/2 objects of different classes (ie. Microsoft Word Document, PowerPoint Presentation, xml file).

The application contains the following script libraries you can copy to your own applications:

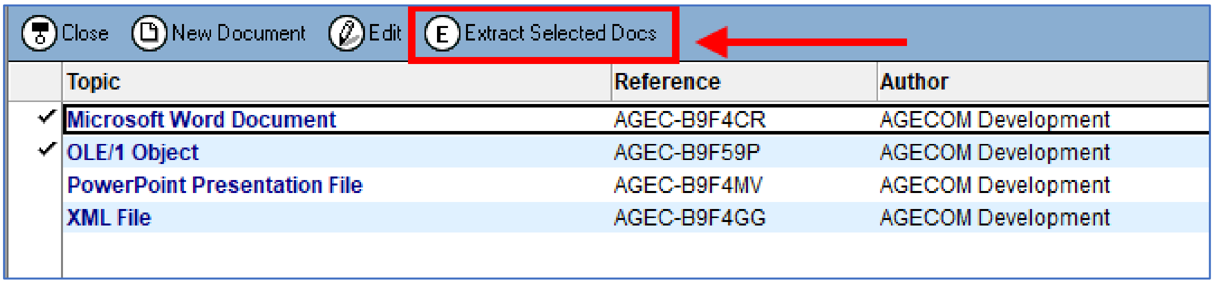
- CompoundFileParser – main script library containing all the code for processing OLE/2 objects.
- NotesDXLProcessing – contains the code for processing OLE/1 objects which can be easily processed using Lotus Notes DXL methods.
- Common – contains common functions and variables used by the above two script libraries.

The application has an agent called *Extract Selected Documents* which demonstrates the processing of each selected Notes document and the embedded OLE objects in them. The agent detects the type of OLE object (ie. OLE/1 or OLE/2) and calls the appropriate methods to process them.

USING THE APPLICATION

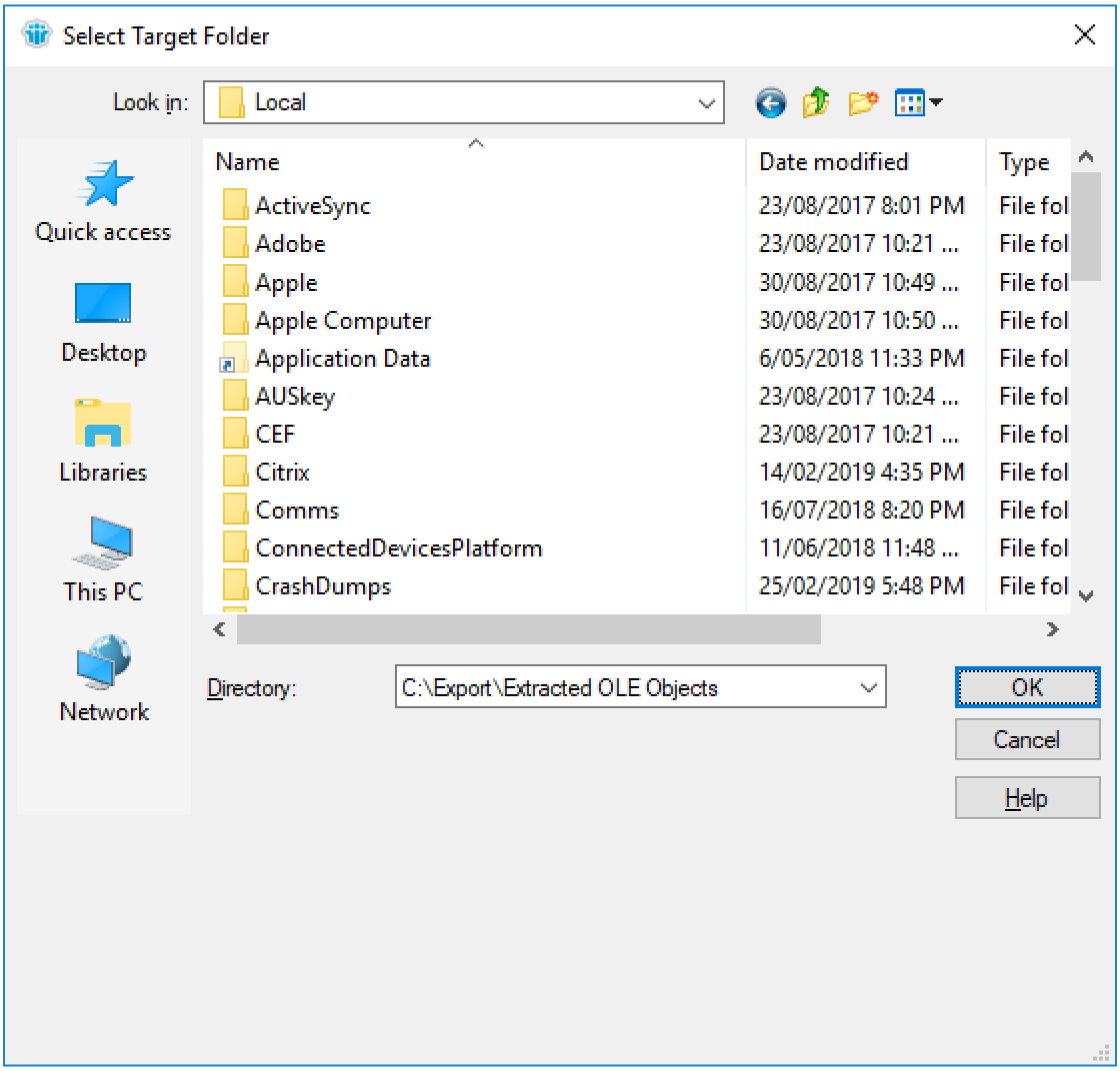
After downloading the sample application from the AGECOM website here are the steps to try it out.

Open the application in the Lotus Notes, select one or more documents in the view then click the *Extract Selected Docs* button at the top of the view:



	Topic	Reference	Author
✓	Microsoft Word Document	AGEC-B9F4CR	AGECOM Development
✓	OLE/1 Object	AGEC-B9F59P	AGECOM Development
	PowerPoint Presentation File	AGEC-B9F4MV	AGECOM Development
	XML File	AGEC-B9F4GG	AGECOM Development

Select the target folder to save the extracted OLE objects to and click the Ok button:



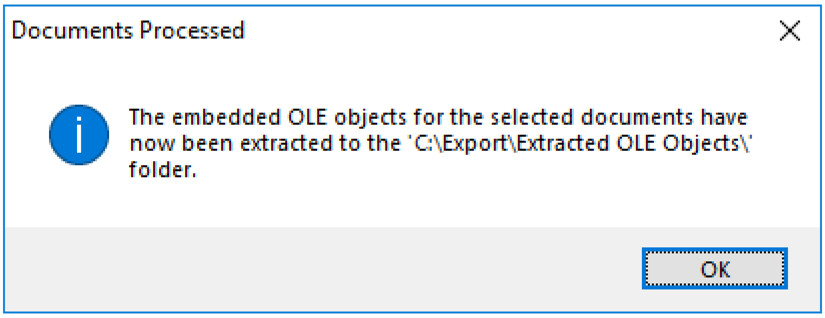
Select Target Folder

Look in: Local

Name	Date modified	Type
ActiveSync	23/08/2017 8:01 PM	File fol
Adobe	23/08/2017 10:21 ...	File fol
Apple	30/08/2017 10:49 ...	File fol
Apple Computer	30/08/2017 10:50 ...	File fol
Application Data	6/05/2018 11:33 PM	File fol
AUSkey	23/08/2017 10:24 ...	File fol
CEF	23/08/2017 10:21 ...	File fol
Citrix	14/02/2019 4:35 PM	File fol
Comms	16/07/2018 8:20 PM	File fol
ConnectedDevicesPlatform	11/06/2018 11:48 ...	File fol
CrashDumps	25/02/2019 5:48 PM	File fol

Directory: C:\Export\Extracted OLE Objects

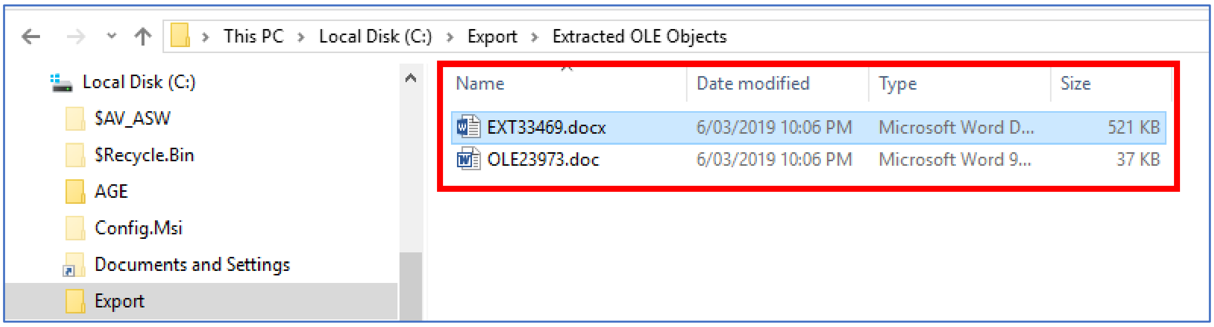
After the OLE objects have been extracted for the selected documents a confirmation box will be displayed:



Documents Processed

The embedded OLE objects for the selected documents have now been extracted to the 'C:\Export\Extracted OLE Objects\' folder.

You can now open the selected target folder to view and open the extracted ole objects:



Name	Date modified	Type	Size
EXT33469.docx	6/03/2019 10:06 PM	Microsoft Word D...	521 KB
OLE23973.doc	6/03/2019 10:06 PM	Microsoft Word 9...	37 KB