

# HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE

## HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE (DATA STRUCTURES)

---

5 May, 2007 (02:42) | [Fundamentals](#), [Forensic tools](#), [Computing theory](#), [Digital forensics](#)

In a previous post I covered “**The basics of how digital forensics tools work.**” In that post, I mentioned that one of the steps an analysis tool has to do is to translate a stream of bytes into usable structures. This is the first in a series of three posts that examines this step (translating from a stream of bytes to usable structures) in more detail. In this post I’ll introduce the different phases that a tool (or human if they’re that unlucky) goes through when recovering digital evidence. The second post will go into more detail about each phase. Finally, the third post will show an example of translating a series of bytes into a usable data structure for a FAT file system directory entry.

### **Data Structures, Data Organization, and Digital Evidence**

Data structures are central to computer science, and consequently bear importance to digital forensics. In **The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition)**, Donald Knuth provides the following definition for a data structure:

#### **Data Structure: A table of data including structural relationships**

In this sense, a “table of data” refers to how a data structure is composed. This definition does not imply that arrays are the only data structure (which would exclude other structures such as linked lists.) The units of information that compose a data structure are often referred to as fields. That is to say, a data structure is composed of one or more fields, where each field contains information, and the fields are adjacent (next) to each other in memory (RAM, hard disk, usb drive, etc.)

The information the fields contain falls into one of two categories, the data a user wishes to represent (e.g. the contents of a file), as well as the structural relationships (e.g. a pointer to the next item in a linked list.) It’s useful to think of the former (data) as data, and the latter (structural relationships) as metadata. Although the line between the two is not always clear, and depends on the context of interpretation. What may be considered data from one perspective, may be considered metadata from another perspective. An example of this would be a Microsoft Word document, which from a file system perspective is data. However, from the perspective of Microsoft Word, the file contains both data (the text) as well as metadata (the formatting, revision history, etc.)

The design of a data structure not only includes the order of the fields, but also the higher level design goals for the programs which access and manipulate the data structures. For instance, efficiency has long been a desirable aspect of many computer programs. With society’s increased

# HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE

dependence on computers, other higher level design goals such as security, multiple access, etc. have also become desirable. As a result, many data structures contain fields to accommodate these goals.

Another important aspect in computing is how to access and manipulate the data structures and their related fields. Knuth defines this under the term “data organization”:

**Data Organization: A way to represent information in a data structure, together with algorithms that access and/or modify the structure.**

An example of this would be a field that contains the bytes 0x68, 0x65, 0x6C, 0x6C, and 0x6F. One way to interpret these bytes is as the ASCII string “hello”. In another interpretation, these bytes can be the integer number 448378203247 (decimal). Which one is it? Well there are scenarios where either could be correct. To answer the question of correct interpretation requires information beyond just the data structure and field layout, hence the term data organization. Even with self-describing data structures, information about how to access and manipulate the “self-describing” parts (e.g. type “1” means this is a string) is still needed.

So where does all this information for data organization (and data structures) come from? There are a few common sources. Perhaps the first would be a document from the organization that designed the data structures and the software that accesses and manipulates them. This could be either a formal specification, or one or more informal documents (e.g. entries in a knowledge base.) Another source would be reverse engineering the code that accesses and manipulates the data structures.

If you’ve read through all of this, you’re might be asking “So how does this relate to digital forensics?” The idea is that data structures are a type of digital evidence. Realize that the term “digital evidence” is somewhat overloaded. In one context, a disk image is digital evidence (i.e. what was collected during evidence acquisition), and in another context, an email extracted from a disk image is digital evidence. This series focuses on the latter, digital evidence extracted from a stream of bytes. Typically this would occur during the analysis phase, although (especially with activities such as verification) this can occur prior to the evidence acquisition phase.

## The 5 Phases

Now that we’ve talked about what data structures are and how they relate to digital forensics, lets see how to put this to use with our forensic tools. What we’re about to do is describe five abstract phases, meaning all tools may not implement them directly, and some tools don’t focus on all five

# HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE

phases. These phases can also serve as a methodology for recovering data structures, should you happen to be in the business of writing digital forensic tools.

1. Location
2. Extraction
3. Decoding
4. Interpretation
5. Reconstruction

The results of each phase are used as input for the next phase, in a linear fashion.

An example will help clarify each phase. Consider the recovery of a FAT directory entry from a disk image. The first task would be to locate the desired directory entry, which could be accomplished through different mechanisms such as calculation or iteration. The next task is to extract out the various fields of the data structure, such as the name, the date and time stamps, the attributes, etc. After the fields have been extracted, fields where individual bits represent sub fields can be decoded. In the example of the directory entry, this would be the attributes field, which denotes if a file is considered hidden, to be archived, a directory, etc. Once all of the fields have been extracted and decoded, they can be interpreted. For instance, the seconds field of a FAT time stamp is really the seconds divided by two, so the value must be multiplied by two. Finally, the data structure can be reconstructed using the facilities of the language of your choice, such as the time class in Python.

There are a few interesting points to note with recovery of data structures using the above methodology. First, not all tools go through all phases, at least not directly. For instance, file carving doesn't directly care about data structures. Depending on how you look at it, file carving really does go through all five phases, it just uses an **identify function**. In addition, file carving does care about (parts of) data structures, it cares about the fields of the data structures that contain "user information", not about the rest of the fields. In fact, much file carving is done with a built-in assumption about the data structure: that the fields that contain "user information" are stored in contiguous locations.

Another interesting point is the distinction between extraction, decoding, and interpretation. Briefly, extraction and decoding focus on extracting information (from stream of bytes and already extracted bytes respectively), whereas interpretation focuses on computation using extracted and decoded information. The next post will go into these distinctions in more depth.

# HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE

A third and subtler point comes from the transition of data structures between different types of memory, notably from RAM to a secondary storage device such as a hard disk or USB thumb drive. Not all structural information may make the transition from RAM, and as a result is lost. For instance, a linked list data structure, which typically contains a pointer field to the next element in the list, may not record the pointer field when being written to disk. More often than not, such information isn't necessary to read consistent data structures from disk, otherwise the data organization mechanism wouldn't really be consistent and reliable. However, if an analysis scenario does require such information (it's theoretically possible), the data structures would have to come directly from RAM, as opposed to after they've been written to disk. This problem doesn't stem from the five phases, but instead stems from a loss of information during the transition from RAM to disk.

In the next post, we'll cover each phase in more depth, and examine some of the different activities that can occur at each phase.

## THE FIVE PHASES OF RECOVERING DIGITAL EVIDENCE

---

8 May, 2007 (16:02) | [Fundamentals](#), [Forensic tools](#), [Computing theory](#), [Digital forensics](#)

This is the second post in a series about the five phases of recovering data structures from a stream of bytes (a form of digital evidence recovery). In the last post we discussed what data structures were, how they related to digital forensics, and a high level overview of the five phases of recovery. In this post we'll examine each of the five phases in finer grained detail.

In the previous post, we defined five phases a tool (or human if they're that unlucky) goes through to recover data structures. They are:

1. Location
2. Extraction
3. Decoding
4. Interpretation
5. Reconstruction

We'll now examine each phase in more detail...

### Location

# HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE

The first step in recovering a data structure from a stream of bytes is to locate the data structure (or at least the fields of the data structure we're interested in.) Currently, there are 3 different commonly used methods for location:

1. Fixed offset
2. Calculation
3. Iteration

The first method is useful when the data structure is at a fixed location relative to a defined starting point. This is the case for a FAT boot sector, which is located in the first 512 bytes of a partition. The second method (calculation) uses values from one or more other fields (possibly in other data structures) to calculate the location of a data structure (or field). The last method (iteration) examines "chunks" of data, and attempts to identify if the chunks are "valid", meaning the (eventual) interpretation of the chunk fits predetermined rules for a given data structure.

These three methods aren't mutually exclusive, meaning they can be combined and intermixed. It might be the case that locating a data structure requires all three methods. For example the **ils** tool from Sleuthkit, when run against a FAT file system ils first recovers the boot sector, then calculates the start of the data region, and finally iterates over chunks of data in the data region, attempting to validate the chunks as directory entries.

While all three methods require some form of **a priori** knowledge, the third method (iteration) isn't necessarily dependent on knowing the fixed offset of a data structure. From a purist perspective, iteration itself really is location. Iteration yields a set of possible locations, as opposed to the first two methods which yield a single location. The validation aspect of iteration is really a combination of the rest of the phases (extraction, decoding, interpretation and reconstruction) combined with post recovery analysis.

Another method for location, that is less common than the previous three is location by outside knowledge from some source. This could be a person who has already performed location, or it could be the source that created the data structure (e.g. the operating system). Due to the flexible and dynamic nature of computing devices, this isn't commonly used, but it is a possible method.

## Extraction

Once a data structure (or the relevant fields) have been located, the next step is to extract the fields of the data structure out of the stream of bytes. Realize that the "extracting" is really the application of type information. The information from the stream is the same, but we're using more information

# HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE

about how to access (and possibly manipulate) the value of the field(s). For example the string of bytes 0x64 and 0x53 can be extracted as an ASCII string composed of the characters “dS”, or it could be the (big endian) value 0x6453 (25683 decimal). The information remains the same, but how we access and manipulate the values (e.g. concatenation vs. addition) differs. Knowledge of the type of field provides the context for how to access and manipulate the value, which is used during later phases of decoding, interpretation, and reconstruction.

The extraction of a field that is composed of multiple bytes also requires knowledge of the order of the bytes, commonly referred to as the “endianess”, “byte ordering”, or “big vs. little endian”. Take for instance the 16-bit hexadecimal number 0x6453. Since this is a 16-bit number, we would need two bytes (assuming 8-bit bytes) to store this number. So the value 0x6453 is composed of the (two) bytes 0x64 and 0x53

It’s logical to think that these two bytes would be adjacent to each other in the stream of bytes, and they typically are. The question is now what is the order of the two bytes in the stream?

0x64, 0x53 (big endian)

or

0x53, 0x64 (little endian)

Obviously the order matters.

## Decoding

After the relevant fields of a data structure have been located and extracted, it’s still possible further extraction is necessary, specifically for bit fields (e.g. flags, attributes, etc.) The difference between this phase and the extraction phase is that extraction extracts information from a stream of bytes and decoding extracts information from the extraction phase. Alternatively, the output from the extraction phase is used as the input to the decoding phase. Both phases however focus on extracting information. Computation using extracted information is reserved for later phases (interpretation and reconstruction).

Another reason to distinguish this phase from extraction is that most (if not all) computing devices can only read (at least) whole bytes, not individual bits. While a human with a hex dump could potentially extract a single bit, software would need to read (at least) a whole byte and extract the various bits within the byte(s).

# HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE

There isn't much that happens at this phase, as much of the activity focuses around accessing various bits.

## Interpretation

The interpretation phase takes the output of the decoding phase (or the extraction phase if the decoding phase uses only **identity functions**) and performs various computations using the information. While extraction and decoding focus on extracting and decoding values, interpretation focuses on computation using the extracted (and decoded) values.

Two examples of interpretation are unit conversion, and the calculation of values used during the reconstruction phase. An example of unit conversion would be converting the seconds field of a FAT time stamp from its native format (seconds/2) to a more logical format (seconds). A useful computation for reconstruction might be to calculate the size of the FAT region (in bytes) for a FAT file system (bytes per sector \* size of one FAT structure (in sectors) \* number of FAT structures.)

Since this phase is used heavily by the reconstruction phase, it's not uncommon to see this phase embodied in the code for reconstruction. However this phase is still a logically separate component.

## Reconstruction

This is the last phase of recovering digital evidence. Information from previous phases is used to reconstruct a usable representation of the data structure (or at least the relevant fields.) Possible usable representations include:

- A language specific construct or class (e.g. Python date object or a C integer)
- Printable text (e.g. output from fsstat)
- A file (e.g. file carving)

The idea is that the output from this phase can be used for some further analysis (e.g. timeline generation, analyzing email headers, etc.) Some tools might also perform some error checking during reconstruction, failing if the entire data structure is unable to be properly recovered. While this might be useful in some automated scenarios, it has the downside of potentially missing useful information when only part of the structure is available or is valid.

At this point, we've gone into more detail of each phase and hopefully explained in enough depth the purpose and types of activity that happen in each. The next (and last) post in this series is an

# HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE

example of applying the five phases to recovering a short name directory entry from a FAT file system.

## RECOVERING A FAT FILESYSTEM DIRECTORY ENTRY IN FIVE PHASES

Posted: 24 May 2007 05:00 PM GMT-06:00

This is the last in a series of posts about five phases that digital forensics tools go through to recover data structures (digital evidence) from a stream of bytes. The **first post** covered fundamental concepts of data structures, as well as a high level overview of the phases. The **second post** examined each phase in more depth. This post applies the five phases to recovering a directory entry from a FAT file system.

The directory entry we'll be recovering is from the HoneyNet Scan of the Month #24. You can download the file by **visiting the SOTM24 page**. The entry we'll recover is the 3rd directory entry in the root directory (the short name entry for `_IMMYJ~1.DOC`, istat number 5.)

### Location

The first step is to locate the entry. It's at byte offset `0x2640` (9792 decimal). How do we know this? Well assuming we know we want the third entry in the root directory, we can calculate the offset using values from the boot sector, as well as the fact that each directory entry is `0x20` (32 decimal) bytes long (this piece of information came from the FAT file system specification.) There is an implicit step that we skipped, recovering the boot sector (so we could use the values). To keep this post to a (semi) reasonable length, we'll skip this step. It is fairly straightforward though. The calculation to locate the third entry in the root directory of the image file is:

3rd entry in root directory = (bytes per sector) \* [(length of reserved area) + [(number of FATs) \* (size of one FAT)]] + (offset of 3rd directory entry)

bytes per sector = `0x200` (512 decimal)

length of reserved area = 1 sector

number of FATs = 2

size of one FAT = 9 sectors

size of one directory entry = `0x20` (32 decimal) bytes

# HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE

offset of 3rd directory entry = size of one directory entry \* 2 (start at 0 since it's an offset)

3rd entry in root directory =  $0 \times 200 * (1 + (2 * 9)) + (0 \times 20 * 2) = 0 \times 2640$  (9792 decimal)

Using xxd, we can see the hex dump for the 3rd directory entry:

```
$ xxd -g 1 -u -l 0x20 -s 0x2640 image
0002640: E5 49 4D 4D 59 4A 7E 31 44 4F 43 20 00 68 38 46 .IMMYJ~1DOC .h8F
0002650: 2B 2D 2B 2D 00 00 4F 75 8F 2C 02 00 00 50 00 00 +-+-..Ou.,...P..
```

## Extraction

Continuing to the extraction phase, we need to extract each field. For a short name directory entry, there are roughly 12 fields (depending on whether you consider the first character of the file name as it's own field.) The multibyte fields are stored in little endian, so we'll need to reverse the bytes that we see in the output from xxd.

To start, the first field we'll consider is the name of the file. This starts at offset 0 (relative to the start of the data structure) and is 11 bytes long. It's the ASCII representation of the name.

```
0002640: E5 49 4D 4D 59 4A 7E 31 44 4F 43 20 00 68 38 46 .IMMYJ~1DOC .h8F
0002650: 2B 2D 2B 2D 00 00 4F 75 8F 2C 02 00 00 50 00 00 +-+-..Ou.,...P..
```

File name = \_IMMYJ~1.DOC (\_ represents the byte 0xE5)

The next field is the attributes field, which is at offset 12 and 1 byte long. It's an integer and a bit field, so we'll examine it further in the decoding phase.

```
0002640: E5 49 4D 4D 59 4A 7E 31 44 4F 43 20 00 68 38 46 .IMMYJ~1DOC .h8F
0002650: 2B 2D 2B 2D 00 00 4F 75 8F 2C 02 00 00 50 00 00 +-+-..Ou.,...P..
```

Attributes =  $0 \times 20$

Continuing in this manner, we can extract the rest of the fields:

```
0002640: E5 49 4D 4D 59 4A 7E 31 44 4F 43 20 00 68 38 46 .IMMYJ~1DOC .h8F
0002650: 2B 2D 2B 2D 00 00 4F 75 8F 2C 02 00 00 50 00 00 +-+-..Ou.,...P..
```

Reserved =  $0 \times 00$

```
0002640: E5 49 4D 4D 59 4A 7E 31 44 4F 43 20 00 68 38 46 .IMMYJ~1DOC .h8F
0002650: 2B 2D 2B 2D 00 00 4F 75 8F 2C 02 00 00 50 00 00 +-+-..Ou.,...P..
```

Creation time (hundredths of a second) =  $0 \times 68$

# HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE

```
0002640: E5 49 4D 4D 59 4A 7E 31 44 4F 43 20 00 68 38 46 .IMMYJ~1DOC .h8F
0002650: 2B 2D 2B 2D 00 00 4F 75 8F 2C 02 00 00 50 00 00 +++-..Ou.,...P..
```

Creation time = 0x4638

```
0002640: E5 49 4D 4D 59 4A 7E 31 44 4F 43 20 00 68 38 46 .IMMYJ~1DOC .h8F
0002650: 2B 2D 2B 2D 00 00 4F 75 8F 2C 02 00 00 50 00 00 +++-..Ou.,...P..
```

Creation date = 0x2D2B

```
0002640: E5 49 4D 4D 59 4A 7E 31 44 4F 43 20 00 68 38 46 .IMMYJ~1DOC .h8F
0002650: 2B 2D 2B 2D 00 00 4F 75 8F 2C 02 00 00 50 00 00 +++-..Ou.,...P..
```

Access date = 0x2D2B

```
0002640: E5 49 4D 4D 59 4A 7E 31 44 4F 43 20 00 68 38 46 .IMMYJ~1DOC .h8F
0002650: 2B 2D 2B 2D 00 00 4F 75 8F 2C 02 00 00 50 00 00 +++-..Ou.,...P..
```

High word of first cluster = 0x0000

```
0002640: E5 49 4D 4D 59 4A 7E 31 44 4F 43 20 00 68 38 46 .IMMYJ~1DOC .h8F
0002650: 2B 2D 2B 2D 00 00 4F 75 8F 2C 02 00 00 50 00 00 +++-..Ou.,...P..
```

Modification time = 0x754F

```
0002640: E5 49 4D 4D 59 4A 7E 31 44 4F 43 20 00 68 38 46 .IMMYJ~1DOC .h8F
0002650: 2B 2D 2B 2D 00 00 4F 75 8F 2C 02 00 00 50 00 00 +++-..Ou.,...P..
```

Modification date = 0x2C8F

```
0002640: E5 49 4D 4D 59 4A 7E 31 44 4F 43 20 00 68 38 46 .IMMYJ~1DOC .h8F
0002650: 2B 2D 2B 2D 00 00 4F 75 8F 2C 02 00 00 50 00 00 +++-..Ou.,...P..
```

Low word of first cluster = 0x0002

```
0002640: E5 49 4D 4D 59 4A 7E 31 44 4F 43 20 00 68 38 46 .IMMYJ~1DOC .h8F
0002650: 2B 2D 2B 2D 00 00 4F 75 8F 2C 02 00 00 50 00 00 +++-..Ou.,...P..
```

Size of file = 0x00005000 (bytes)

## Decoding

With the various fields extracted, we can decode the various bit-fields. Specifically the attributes, dates, and times fields. The attributes field is a single byte, with the following bits used to represent the various attributes:

- Bit 0: Read only
- Bit 1: Hidden
- Bit 2: System

# HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE

- Bit 3: Volume label
- Bit 4: Directory
- Bit 5: Archive
- Bits 6 and 7: Unused
- Bits 0, 1, 2, 3: Long name

When decoding the fields in a FAT file system, the right most bit is considered bit 0. To specify a long name entry, bits 0, 1, 2, and 3 would be set. The value we extracted from the example was 0x20 or 0010 0000 in binary. The bit at offset 5 (starting from the right) is set, and represents the “Archive” attribute.

Date fields for a FAT directory entry are encoded in two byte values, and groups of bits are used to represent the various sub-fields. The layout for all date fields (modification, access, and creation) is:

- Bits 0-4: Day
- Bits 5-8: Month
- Bits 9-15: Year

Using this knowledge, we can decode the creation date. The value we extracted was 0x2D2B which is 0010 1101 0010 1011 in binary. The day, month, and year fields are thus decoded as:

0010 1101 0010 1011

Creation day: 01011 binary = 0xB = 11 decimal

0010 1101 0010 1011

Creation month: 1001 binary = 0x9 = 9 decimal

0010 1101 0010 1011

Creation year: 0010110 binary = 0x16 = 22 decimal

A similar process can be applied to the access and modification dates. The value we extracted for the access date was also 0x2D2B, and consequently the access day, month, and year values are identical to the respective fields for the creation date. The value we extracted for the modification date was 0x2C8F (0010 1100 1000 1111 in binary). The decoded day, month, and year fields are:

0010 1100 1000 1111

Modification day: 01111 binary = 0xF = 15 decimal

# HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE

0010 1100 1000 1111

Modification month: 0100 binary =  $0 \times 4 = 4$  decimal

0010 1100 1000 1111

Modification year: 0010110 binary =  $0 \times 16 = 22$  decimal

You might have noticed the year values seem somewhat small (i.e. 22). This is because the value for the year field is an offset starting from the year 1980. This means that in order to properly interpret the year field, the value 1980 ( $0 \times 7BC$ ) needs to be added to the value of the year field. This is done during the next phase (interpretation).

The time fields in a directory entry, similar to the date fields, are encoded in two byte values, with groups of bits used to represent the various sub-fields. The layout to decode a time field is:

- Bits 0-4: Seconds
- Bits 5-10: Minutes
- Bits 11-15: Hours

Recall that we extracted the value  $0 \times 4638$  (0100 0110 0011 1000 in binary) for the creation time. Thus the decoded seconds, minutes, and hours fields are:

0100 0110 0011 1000

Creation seconds = 11000 binary =  $0 \times 18 = 24$  decimal

0100 0110 0011 1000

Creation minutes = 110001 binary =  $0 \times 31 = 49$  decimal

0100 0110 0011 1000

Creation hours = 01000 binary =  $0 \times 8 = 8$  decimal

The last value we need to decode is the modification time. The bit-field layout is the same for the creation time. The value we extracted for the modification time was  $0 \times 754F$  (0111 0101 0100 1111 in binary). The decoded seconds, minutes, and hours fields for the modification time are:

0111 0101 0100 1111

Modification seconds = 01111 binary =  $0 \times F = 15$  decimal

0111 0101 0100 1111

Modification minutes = 101010 binary =  $0 \times 2A = 42$  decimal

# HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE

0111 0101 0100 1111

Modification hours = 01110 binary = 0xE = 14 decimal

## Interpretation

Now that we've finished extracting and decoding the various fields, we can move into the interpretation phase. The values for the years and seconds fields need to be interpreted. The value of the years field is the offset from 1980 (0x7BC) and the seconds field is the number of seconds divided by two. Consequently, we'll need to add 0x7BC to each year field and multiply each second field by two. The newly calculated years and seconds fields are:

- Creation year = 22 + 1980 = 2002
- Access year = 22 + 1980 = 2002
- Modification year = 22 + 1980 = 2002
- Creation seconds = 24 \* 2 = 48
- Modification seconds = 15 \* 2 = 30

We also need to calculate the first cluster of the file, which simply requires concatenating the high and the low words. Since the high word is 0x0000, the value for the first cluster of the file is the value of the low word (0x0002).

In the next phase (reconstruction) we'll use Python, so there are a few additional values that are useful to calculate. The first order of business is to account for the hundredths of a second associated with the seconds field for creation time. The value we extracted for the hundredths of a second for creation time was 0x68 (104 decimal). Since this value is greater than 100 we can add 1 to the seconds field of creation time. Our new creation seconds field is:

- Creation seconds = 48 + 1 = 49

This still leaves four hundredths of a second left over. Since we'll be reconstructing this in Python, we'll use the Python `time` class which accepts values for hours, minutes, seconds, and microseconds. To convert the remaining four hundredths of a second to microseconds multiply by 10000. The value for creation microseconds is:

- Creation microseconds = 4 \* 10000 = 40000

The other calculation is to convert the attributes field into a string. This is purely arbitrary, and is being done for display purposes. So our new attributes value is:

# HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE

- Attributes = "Archive"

## Reconstruction

This is the final phase of recovering our directory entry. To keep things simple, we'll reconstruct the data structure as a Python **dictionary**. Most applications would likely use a Python object, and doing so is a fairly straight forward translation. Here is a snippet of Python code to create a dictionary with the extracted, decoded, and interpreted values (don't type the >>> or ...):

```
$ python
>>> from datetime import date, time
>>> dirEntry = dict()
>>> dirEntry["File Name"] = "\xE5IMMYJ~1DOC"
>>> dirEntry["Attributes"] = "Archive"
>>> dirEntry["Reserved Byte"] = 0x00
>>> dirEntry["Creation Time"] = time(8, 49, 49, 40000)
>>> dirEntry["Creation Date"] = date(2002, 9, 11)
>>> dirEntry["Access Date"] = date(2002, 9, 11)
>>> dirEntry["First Cluster"] = 2
>>> dirEntry["Modification Time"] = time(14, 42, 30)
>>> dirEntry["Modification Date"] = date(2002, 4, 15)
>>> dirEntry["size"] = 0x5000
>>>
```

If you wanted to print out the values in a (semi) formatted fashion you could use the following Python code:

```
>>> for key in dirEntry.keys():
... print "%s == %s" % (key, str(dirEntry[key]))
...
```

And you would get the following output

```
Modification Date == 2002-04-15
Creation Date == 2002-09-11
First Cluster == 2
File Name == ?IMMYJ~1DOC
Creation Time == 08:49:49.040000
Access Date == 2002-09-11
Reserved Byte == 0
Modification Time == 14:42:30
```

# HOW FORENSIC TOOLS RECOVER DIGITAL EVIDENCE

```
Attributes == Archive  
size == 20480  
>>>
```

At this point, there are a few additional fields that could have been calculated. For instance, the file name could have been broken into the respective 8.3 (base and extension) components. It might also be useful to calculate the allocation status of the associated file (in this case it would be unallocated). These are left as exercises for the reader ;).

This concludes the 3-post series on recovering data structures from a stream of bytes. Hopefully the example helped clarify the roles and activities of each of the five phases. Realize that the five phases aren't specific to recovering file system data structures, they apply to network traffic, code, file formats, etc.