Processing Standard Zip Files with C# compression/decompression classes

Uzi Granot
11 Feb 2014 CPOL

This project will give you the tools needed to compress and decompress files using the Deflate method of compression, and to read and write standard Zip files.

Download demo - 43.2 KB

Download source - 391.4 KB

1. Introduction

ZIP files are among the most commonly used files. They are integral part of the computing scene since 1989. Did you ever wonder how they really work? I did. I created this project to satisfy my curiosity. I wanted to fully understand the fine details of file compression and decompression programming. I purchased years ago the book "Text Compression" by Timothy C. Bell, John G. Cleary and Ian H. Witten. This is an excellent introduction to the subject of text compression. Two algorithms caught my attention the Adaptive Dictionary by Ziv and Lempel and Huffman coding. It just happen that Mr. Lempel was one of my teachers at the Technion while I studied for B.Sc in Electrical Engineering.

The most used compression method today is the "deflate" method. It is built into Microsoft Windows Explorer. The source code is readily available for download from many websites in a number of languages. My goal for this project was to document the code as much as possible. My starting point was the C# code written by Mike Krueger. I have reorganized the code, rewritten significant part of it, renamed many of the variables to longer more meaningful names and added many comments. In addition, I have added support to read and write ZIP files compatible with the Windows Explorer Send To Compressed (zipped) Folder option. To test the code, I developed an application to read existing ZIP files and extract some or all of the files, to create a new ZIP file or to edit existing ZIP file by adding and deleting files and folders.

The project is made of the following logical blocks:

- `DeflateMethod` and `DeflateTree` classes. The compression classes.
- `InflateMethod` and `InflateTree` classes. The decompression classes.
- `DeflateNoHeader`, `DeflateZLib` and `DeflateZipFile` classes. Derived classes from the `DeflateMethod`. They provide file compression support. The `DeflateZIPFile` has the tools for creating ZIP files.
- `InflateNoHeader`, `InflateZLib` and `InflateZipFile` classes. Derived classes from `InflateMethod`. They provide file decompression support. The `InflateZIPFile` has the tools for reading ZIP files.
- `CRC32`, `Adler32` and `BitReverse` are support classes for both deflate and inflate classes.
- `UZipDotNet` class. The application that ties all the compression and decompression classes.
- Miscellaneous support classes.

Other examples of using the compression and decompression classes are given in "PDF File Analyzer With C# Parsing Classes" and "PDF File Writer C# Class Library" articles by Uzi Granot published at

CodeProject.com. In these two examples PDF streams are compressed and decompressed with /FlateDecode filter. The two source modules InflateMethod.cs and DeflateMethod.cs provide methods to compress and decompress from one byte array to another byte array.

2. References

- "Text Compression" by Timothy C. Bell, John G. Cleary, Ian H. Witten. Prentice Hall Advanced Reference Series. Computer Science 1990.
- DEFLATE Compressed Data Format Specification version 1.3. RFC 1951. By Peter Deutsch. Aladdin Enterprises. 1996.
- GZIP file format specification version 4.3. RFC 1952. By Peter Deutsch. Aladdin Enterprises 1996.
- ZLIB Compressed Data Format Specification version 3.3. RFC 1950. By Peter Deutsch Aladdin Enterprises and Head-Loup Gailly Info-ZIP 1996.
- APPNOTE.TXT - .ZIP File Format Specification. Version: 6.3.2. Revised: September 28, 2007. Copyright (c) 1989 - 2007 PKWARE Inc.
- Algorithm.txt file part of ZLIB 1.2.5 download. Compression algorithm (deflate) and Decompression algorithm (inflate) By Jean-Loup Gailly and Mark Adler.
- Source code download of ICSharpCode.SharpZipLib.Zip Copyright © 2001 Mike Krueger.

3. Compression

The general idea behind the Deflate compression method is the replacement of a repeated string with the distance to the previous occurrence of the string and the length of the string. For example, in this article the word compression appears many times. After the first appearance all subsequence appearances will be a reference to the previous word. If the program cannot find a matching string of at least 3 bytes in the previous text, the program sends the current literal to the output stream. The result is a mix of literals and distance length pairs. The Literals and distance length pairs are encoded as a combination of codes and extra bits. The codes go through Huffman coding process that produces variable length bit strings. Short strings for high frequency codes and long strings for low frequency codes.

Let us look at this process in more details. The program reads the input file into a read buffer. The buffer is scanned one byte at a time. The program looks back in the buffer for the longest possible match to the current string at the read pointer. To speed up the process, the program is using a 16 bit (65K) hash table. The current 3 bytes at the read pointer are converted into a hash index into the hash table. Each entry in the hash table contains a pointer to the previous 3 byte sequence with the same hash index value. The resulted pointer is used to compare the current string to previous possible match. The program will continue moving backward in the buffer based on the hash linked chain of possible matches. If one or more matches are found, the longest one is taken. The distance to the matching string and the length of the match are put into the block buffer. If no match is found the current literal or byte is put into a block buffer. An entry in the block buffer with zero distance is a literal and non zero distance is a matched string.

The program has two methods to find string matches. DeflateFast and DeflateSlow. The main difference between them is that DeflateSlow will not commit a match to the block buffer before trying the next byte. If the next byte yields longer match, the byte at the current pointer is sent to the buffer in the same way as no match. Match length is minimum 3 bytes and maximum 258 bytes. The maximum distance to a previous match is 32768 bytes. This is the scan window. The read buffer will always have at least 32768 characters before the current pointer and 262 bytes look ahead after the current pointer except for the beginning and end of the input file.

An interesting case of a matched string is the case of a repeated character. For example, a source file containing a line of 80 dashes in a row. When the read pointer is on the second dash of this line, the matching process will produce a distance of 1 and length of 79. In other words, the current string and the previous matching string overlap.

The block buffer is 16K long. When the block buffer is full the encoding process starts. Literals remain unchanged. The encoded values are the same as the literal values. Literals being bytes have a range of 0 to 255. Length range is 3 to 258. It is encoded as a combination of code and extra bits. See table below. For example if length is 7 the code is 261 and there are no extra bits. If the length is 100, the code is 279 there are 4 extra bits '0001'. The program combines literals and length codes into one entity called symbol. If the symbol value is 0 to 255 it is a literal. If the symbol value is 256 it is an end of block marker. If the symbol is 257 to 285 it is length. If the symbol is between 265 and 284 it is a length and it is followed by extra bits.

C#

```
// Length Code (See RFC 1951 3.2.5)
//            Extra               Extra               Extra
//       Code Bits Length(s) Code Bits Lengths   Code Bits Length(s)
//       ---- ---- ------     ---- ---- -------   ---- ---- -------
//       257   0     3       267   1   15,16     277   4   67-82
//       258   0     4       268   1   17,18     278   4   83-98
//       259   0     5       269   2   19-22     279   4   99-114
//       260   0     6       270   2   23-26     280   4  115-130
//       261   0     7       271   2   27-30     281   5  131-162
//       262   0     8       272   2   31-34     282   5  163-194
//       263   0     9       273   3   35-42     283   5  195-226
//       264   0    10       274   3   43-50     284   5  227-257
//       265   1  11,12      275   3   51-58     285   0    258
//       266   1  13,14      276   3   59-66
```

Distance has a range of 1 to 32768. It is encoded as a combination of code and extra bits. See table below. For example distance 3 is code 2 with no extra bits. Distance 1000 is code 19 with 8 extra bits 11100111 (231)

C#

```
// Distance Codes (See RFC 1951 3.2.5)
//            Extra           Extra               Extra
//       Code Bits Dist  Code Bits   Dist     Code Bits Distance
//       ---- ---- ----  ---- ----  ------    ---- ---- --------
//         0   0    1     10   4     33-48    20    9   1025-1536
//         1   0    2     11   4     49-64    21    9   1537-2048
//         2   0    3     12   5     65-96    22   10   2049-3072
//         3   0    4     13   5     97-128   23   10   3073-4096
//         4   1   5,6    14   6    129-192   24   11   4097-6144
//         5   1   7,8    15   6    193-256   25   11   6145-8192
//         6   2   9-12   16   7    257-384   26   12  8193-12288
//         7   2  13-16   17   7    385-512   27   12 12289-16384
//         8   3  17-24   18   8    513-768   28   13 16385-24576
//         9   3  25-32   19   8   769-1024   29   13 24577-32768
```

Taking the repeated dash example above, the encoded result of 80 consecutive dashes will be:

Literal	**Length of 79**		**Distance of 1**	
Dash	Code	Extra Bits	Code	Extra Bits
45	277	1100 (12)	0	None

At this point we have two types of codes literals/Length codes in the range of 0 to 285 and distance codes in the range of 0 to 29. For simplicity we will refer to the literal/length codes as literal codes. For each of code type, the program calculates the number of times each code appears in the block. This is the code frequency. The program builds two frequency arrays. Literal frequency array of 286 elements and distance frequency array of 30 elements. Based on the usage frequency of each code we build Huffman tree and assign a bit string of varying length to each code in such a way that frequently used codes get shorter bit strings and infrequently used codes get longer bit strings.

The conversion of codes to Huffman codes is done in `DeflateTree` class. There are three instances of the `DeflateTree` class. Literal tree, Distance tree and Bit Length tree. So far we talked about the first two. The Bit Length tree is used to transmit the coding information of the other two trees. This process will be discussed later. The first step in converting codes to Huffman codes is to create a tree with just leaf nodes. One node for each used code. Each node has three values associated with it. The code, the frequency and pointer to a pair of children nodes. Unused codes are ignored. The tree is sorted by frequency order. Low frequency to high frequency. The program starts at the two lowest frequency nodes. It creates a new parent node with a combined frequency of the two leafs. The pointer to the two children nodes is saved at the parent node. The program places the new node in the tree just before a node with the next higher frequency. This process continues for each pair of nodes until all pairs of nodes are scanned. Next, the program transverses the tree using recursive method and assign to each node the distance to the root. This distance is the number of bits required to represent the code. It is the length of the code. The length of the code is saved in an array. Below you will see an example of six letters

alphabet. Step 1 shows the usage frequency of each letter. Step 2 is the same table sorted by frequency. Step 3 shows the parent nodes added and the pointers to leaf nodes. The bit length column shows the number of steps starting from the root to reach each leaf node.

The maximum number of bits required to represent a symbol is limited to 15 bits for both literal and distance trees. Bit length trees (discussed later) are limited to 7 bits. If the calculation above yields more than the maximum allowed, the program readjust the frequency of the least used codes in such a way as to create a more symmetric tree. This will effectively reduce the number of bits for the least used codes at the expense of the next higher frequency codes. At the end of this process we will have a literal code bit length array of 286 elements and a distance code bit length array of 30 elements. At this time we know the code length in bits for each code, but we do not know the actual value of the codes. Code values will be assigned later.

The literal and the distance code length arrays will have to be included in the compressed file in order for the decompression program to read and decode the compressed symbols. As part of the Deflate process these two arrays will be compressed too for inclusion in the compressed file. The literals in this case are the lengths of the bit strings. As indicated before, the range of code length is 1 to 15. In addition we need zero to indicate not used code and three other repeat symbols to compress repeated codes. In total the bit length array is 19 elements long. The program builds a frequency array for the combined codes of the literal and distance tree. The bit length tree is built in the same way as the literal and distance trees were built before. One difference is maximum code length. In this case it is 7 bits. The second difference is the bit length order array. The bit length array is 19 elements long. The last three are the most likely elements to be used. The program reorganized the bit length array by using translation order array. The output of this process is such that likely elements come first and unused elements are hopefully at the end. The system transmits up to the last used code.

The deflate compression algorithm has a potential problem. The problem is that incompressible files can become longer than the original file after compression. Obviously it is better to transmit the original file rather than a longer version of it. The ZIP file header and the ZLIB file header have a compression method flag. In our case it can be either Deflate (8) or Stored (0). If the entire file in not compressible, it will be Stored. The entire file will be copied as is to the compressed file archive. If the input file is less than 8 bytes long,

no compression will be attempted and the file will be stored as is. If the file is compressible, we will use the `Deflate` method. Within a compressible file some blocks can be incompressible. In this case we want to copy these blocks from the input stream to the output stream unchanged. Compressible blocks have two options: dynamic tree option and static tree option. The dynamic tree option is a block that the Huffman tree information is calculated by the program and is saved at the beginning of the block. Static tree option is using a fixed Huffman tree. It was defined by the inventor of the Deflate process. This tree is not stored in the compressed file. The decoding program has all the information to build it. The following table describes these static trees:

C#

```
// Static literal codes and length codes  (See RFC 1951 3.2.6)
//       Lit Value    Bits        Codes
//       ---------    ----        -----
//         0 - 143     8          00110000 through 10111111
//       144 - 255     9          110010000 through 111111111
//       256 - 279     7          0000000 through 0010111
//       280 - 287     8          11000000 through 11000111
//       Note that literal/length codes 286-287 will never actually
//       occur in the compressed data.
//
// Static distance codes (See RFC 1951 3.2.6)
//       Distance codes 0-31 are represented by (fixed-length) 5-bit
//       codes, with possible additional bits as shown in the table
//       shown in Paragraph 3.2.5, above.  Note that distance codes 30-
//       31 will never actually occur in the compressed data.
//
```

The advantage of the static tree is no overhead. The tree is known to the de-compressor and it is not included in the file. The advantage of the dynamic tree is better compression in most cases. To summarize, the entire file can be either stored or compressed. If the file is compressed it is divided into blocks. Each block can be stored or compressed using static tree or compressed using dynamic tree. The logic that determines which of these options is described below.

If the input file is very small, less than 8 bytes, it is stored. All other files go through the process that was described so far. At the end of each block we know the frequency of each symbol and we know the length in bits of each symbol and we know how many extra bits will be included in the file. Based on this information, the program calculates what would be the length of the block if it was encoded using dynamic trees including the inherited dynamic trees overhead. Next the program calculates the length of the block if static trees will be used. The program will compare the dynamic length, static length and the uncompressed block length to decide the shortest option. If the block is the last block of the file, the program will perform one more test. It will compare the expected overall length of the compressed file to the original size of the uncompressed file. If it is longer, the overall compression method will be changed from `Deflate` to `Stored`. The program will rewind the input and output files and copy the input file to the output file.

In the case that dynamic tree option was selected, a translation table between codes and bit strings is needed. We know the length of each bit string for each code. So the next step is building an array of bit strings. The program scans the code length array for each of the 15 possible lengths. From 1 to 15. Let say that the smallest bit length used is 3 bits. The program will assign a 3 bit code to each code whose bit length is 3. The starting code is zero. Let say we have 3 codes of 3 bits in total. The codes will be 000, 001 and 010. The next bit length in use is 5 bits. The first code will be 01100. It will be the continuation of the previous series plus additional bits. If we have 5 codes of 5 bits, they will be 01100, 01101, 01110, 01111 and 10000. If you look at the `BuildCodes` method of `DeflateTree` class, the codes are first left justified within a 16 bit unsigned integer and then the bits are reversed within the 16 bit integer. The final value is reversed codes. This process is simplifying the work of the program decompressing the file.

The compressing program has to include the translation table between codes and bit strings in the compressed file in order for the decompressing program to decode the data. This will be accomplished by including the bit string length of each code in the file. The decoding program will go through the process described in the previous paragraph to create the translation table. The bit stream length array is being compressed by detecting consecutively repeating symbols. To start with each element is zero to 15. Zero for unused code and 1 to 15 for all possible bit lengths. The program adds three additional codes for compressing repeated bit string length: RepeatSymbol_3_6 (16), RepeatSymbol_3_10 (17) and RepeatSymbol_11_138 (18). In total there are 19 symbols. The program will use RepeatSymbol_3_6 to compress repeated used codes. The program will use RepeatSymbol_3_10 and RepeatSymbol_11_138 to compress repeated unused codes. The 19 symbols go through another transformation. The three most likely used codes are the repeat symbols. The program changes the order of all the 19 symbols in such a way as to give most likely symbols small values and least likely symbols high values. If the last few symbols are not used, they will not be transmitted.

Finally the program is ready to output one compressed block. At the beginning of each block there is a 3 bits header. Two bits to specify block type: Stored 00, Static Tree 01 and Dynamic tree 10. The third bit is set for the last block of the file. If it is a stored block, the next two bytes (16 bits) are the length of the block (maximum 65535). The next two bytes are the length of the block inverted. The program makes sure that the length of the block is byte aligned. After these 4 bytes the block is copied from the input file. For static tree the compressed data starts right after the 3 bit header. For dynamic tree the program sends the three trees information to the output file. First the lengths of the three trees and next the trees themselves. For both static and dynamic trees this is the time to actually write the compressed data to the output file. The program loop through the block buffer one entry at a time converting the literal/length symbols to bit strings plus optional extra bits. If the literal/length symbol was length, the program will convert the matching distance to bit string and optional extra bits. At the end of the block, the program writes the end of block marker.

After a block is written to the output file the process continues to the end of the file one block at a time.

The full details are given in the C# code itself. It is a very CPU intensive application. The search for matching strings is probably the piece of code that consumes most of the CPU time.

4. Decompression

Decoding Deflate compressed files is more straight forward than compressing them. The decompressing program reads the bit strings symbols, converts them to codes, literal codes go directly to the output file, length and distance codes are converted into bytes strings and these strings are sent to the output file.

Let us look at this process in more details. If the compressed file is ZIP or ZLIB file, the compression method code is either Stored (0) or Deflate (8). The attached program does not support any other compression method. If the compression method is Stored, the decompression program just copies the input file to the output file. If the compression method is Deflate the decompression program decodes the file and recreates the original file.

The compressed file is divided into blocks. The first 3 bits of each block is the block header. The first 2 bits determine the type of block: stored 00, static tree 01 and dynamic tree 10. The third bit is the last block flag.

If a block is a stored block, the program aligns the read pointer to the next byte boundary. It reads the next 16 bits as an unsigned integer. This is the length in bytes of the block. The following 16 bits are the same length but inverted. The program reads that number, inverts it and compares it to the first number. If the test fails, the file is invalid. Next the program reads the specified number of bytes from the input file to the output file.

If a block is a dynamic tree block, the program reads the trees information. There are three trees: bit length tree, literal/length tree and distance tree. Each tree is an array of code lengths. The array's index is the code and each item is the length in bits of each code. This information is sufficient to recreate the bit string associated with the code. This process is described above in the compression section. Reading the trees information is done in the following sequence. First the program reads the length of each of the three trees. Next the program reads the bit length array. Using the bit length array the program recreates the bit strings of all the 19 items of this array. Next the program reads the literals/lengths and distance arrays using the information from the bit length array. The program recreates the translation tables between bit strings and codes for the literals/lengths tree and the distance tree. After reading and processing the trees information, the read pointer is now at the start of the actual compressed data.

If a block is a static tree block, the program has all the information it needs to create the literals/length tree and the distance tree. The start of the compressed data is right after the 3 bits header.

The compressed file is made of bit strings of varying length. Each code can be 1 to 15 bits plus optional extra bits. In the compression program the task of converting a code to a bit string is simple. We have one array of bit string codes and one matching array of bit string lengths. We index into these two arrays and move the right number of bits from the bit string codes array to the output stream. The reverse process is not obvious. We have a stream of bits. Codes have variable length. One code follows another without any separators. To be able to read the codes in an unambiguous way, the codes are constructed in such a way that no code is a prefix to another code. For example, if 0101 is a valid 4 bit code then 0 or 01 or 010 are not valid codes. To decode this stream of bits one needs to create a tree of nodes. Each node has two values: child-zero and child-one. These values are either pointers to other nodes of the tree or the translated symbol. For example, we have a six letters alphabet A to F. The variable bit string for each one is: A-00, B-1110, C-01, D-110, E-10 and F-1111. We create a tree as per the diagram below. The input bit is 01001110. If we follow the bits down the tree until reaching a letter and then starting again at the root, the result will be CAB.

This method works fine but it is slow. The compressed file will be processed one bit a time. To speed up the process, we want to process a block of bits at one time. The program was set to 9 bits. We create an array of 512 values. The program will read a block of 9 bits from the input file and index into this array. If the next code in the input buffer is equal or less than 9 bits, then the value in the array is our code. If the next code in the input buffer is longer than 9 bits then value in the array is a pointer to a sub-tree starting from the 10^{th} bit. To illustrate this process based on the example given above, we will take a block of three bits and array of 8 elements. The code for letter A is 00. Since our decoder will pick 3 bits the extra bit after the A can be either one or zero. We will place A in both element zero of the table and in element one. After decoding the A we will move the bit pointer by 2 bits. If a code has fewer bits than the table was designed for it will populate multiple entries in the table. The code for letter D is 110. It is three bits, exactly the number the decoder will pick. In this case we have one entry. The letters B and F have codes longer than 3 bits. In this case we will place a pointer in location 7 to an area of two elements above the table. The first is for B and the second for F.

Index	Pointer	Code	Code Length
0		A(000)	2
1		A(001)	2
2		C(010)	2
3		C(011)	2
4		E(100)	2
5		E(101)	2
6		D(110)	3
7	8		
8		B(1110)	4
9		F(1111)	4

The decompression process is identical for both static and dynamic trees. This process gets as input two trees, literal/length and distance. The process does not know or care if the trees are static or dynamic. The decompression loop for one block is straight forward. It reads the next symbol from the file as described above. The symbol value range is 0 to 285. If it is 0 to 255 it is a literal. The literal is saved in the output file. If it is 256, it is the end of the block. If the symbol is 257 to 285 it is length. Based on the length table above, the program gets the base length and how many extra bits to read. The length is calculated as base value plus the value of the extra bits. After decoding length, the program reads the next symbol as a distance. Again, using the distance table above, the program calculates the distance from base value plus the extra bits. Using the distance and length the program copies the string from the current location less the distance to the current position of the write buffer. A special case is when the source and destination strings overlap. The array copy routine will not work in this case. The program will move one byte at a time for the overlapping area. If the block header had the last block flag set, the decompression is done when the end of block marker is received.

5. ZIP file

ZIP file is an archive that contains individual files and directory paths. The file is made of three parts: file area, central directory and end of directory record. The table below shows an example of a ZIP archive with three files. The file area contains either compressed files or directory paths. Each file entry is made of header and the compressed file data. File entry for directory has no data just the header. The central directory contains file headers for all the files. The file headers in the central directory are similar but not the same as the headers in the file area. The end of directory record has a pointer to the start of the central directory. To read a ZIP file one should read the end of directory record at the end of the file, use the pointer to the start of the central directory, load the central directory into memory, and access individual files based on the directory information. At the start of each file header and at the start of the end of directory record there is 32 bit signature. When processing the file the program always verifies the signature marker. To read the file, the program reads the last 512 bytes of the file. It scans the block backward looking for the 32 bits central directory file header signature. Once found the program reads the number of central directory entries and the start position of the central directory.

Files Area	Central Directory	End Of Directory

<table>
<tr><th colspan="6">Files Area</th><th colspan="3">Central Directory</th><th>End Of Directory</th></tr>
<tr><td colspan="2">File 1</td><td colspan="2">File 2</td><td colspan="2">File 3</td><td rowspan="2">File 1 Header</td><td rowspan="2">File 2 Header</td><td rowspan="2">File 3 Header</td><td rowspan="2">Pointer to start of central directory</td></tr>
<tr><td>Header</td><td>Data</td><td>Header</td><td>Data</td><td>Header</td><td>Data</td></tr>
</table>

Reading ZIP archive is done in the `InflateZipFile` class. Writing ZIP archives is done in the `DeflateZipFile` class. The central directory information is maintained in an array in memory. Each element is defined in the ZipDirectory structure.

The ZIP support has limitations:

- No support for multiple disks.
- Compression method is limited to Deflate or Stored.
- Version is assumed to be 20.
- General purpose bit flag is assumed to be 0.

Below you will find the definition of the file area file header, central directory file header and the end of central directory record. This information was obtained from PKWARE. APPNOTE.TXT - .ZIP File Format Specification. Version: 6.3.2. Revised: September 28, 2007. Copyright (c) 1989 - 2007 PKWARE Inc. The source code of the three classes mentioned above has all the details.

C#

```
//       File area file header
//
//       Pos               Len
//       0                 4               Local file header signature = 0x04034b50
//       4                 2               Version needed to extract (minimum)
//       6                 2               General purpose bit flag
//       8                 2               Compression method
//       10                2               File last modification time
//       12                2               File last modification date
//       14                4               CRC-32
//       18                4               Compressed size
//       22                4               Uncompressed size
//       26                2               File name length (n)
//       28                2               Extra field length (m)
//       30                n               File name
//       30+n              m               Extra field (NTFS file date and time)
//
//       End of central directory record:
//
//       Pos               Len
//       0                 4               End of central directory signature = 0x06054b50
//       4                 2               Number of this disk
//       6                 2               Disk where central directory starts
//       8                 2               Number of central directory records on this disk
//       10                2               Total number of central directory records
//       12                4               Size of central directory (bytes)
//       16                4               Offset of start of central directory, relative to start
of archive
//       20                2               ZIP file comment length (n)
//       22                n               ZIP file comment
//
//       Central directory file header
//
//       Pos               Len
//       0                 4               Central directory file header signature = 0x02014b50
//       4                 2               Version made by
//       6                 2               Version needed to extract (minimum)
//       8                 2               General purpose bit flag
//       10                2               Compression method
//       12                2               File last modification time
//       14                2               File last modification date
//       16                4               CRC-32
```

```
//       20              4               Compressed size
//       24              4               Uncompressed size
//       28              2               File name length (n)
//       30              2               Extra field length (m)
//       32              2               File comment length (k)
//       34              2               Disk number where file starts
//       36              2               Internal file attributes
//       38              4               External file attributes
//       42              4               Offset of local file header
//       46              n               File name
//       46+n            m               Extra field
//       46+n+m          k               File comment
//
```

5.1. NTFS File Date and Time

The Windows OS uses the file header extra field in the file area to save the file date and time in local time. To get the time use either `FileInfo.LastWriteTime.ToFileTime()` or `File.GetLastWriteTime(FileName).ToFileTime()`. To set the time, first convert the bytes into DateTime format: `FileModifyTime = DateTime.FromFileTime(BitConverter.ToInt64(TimeField, 12))`. Next use either `FileInfo.LastWriteTime = FileModifyTime` or `File.SetLastWriteTime(FileName, ModifyTime)`.

C#

```
//       NTFS File date and time
//
//       Pos             Len
//       0               2               NTFS tag signature = 0x000a
//       2               2               Length 32 bytes = 0x0020
//       4               4               Reserved area = 0x00000000
//       8               2               File date and time signature = 0x0001
//       10              2               Length 24 bytes = 0x0018
//       12              8               File last write time (DateTime format)
//       20              8               File last access time (DateTime format)
//       28              8               File last creation time (DateTime format)//
```

6. UZipDotNet Application

The UZipDotNet application was developed to test the compression and decompression classes. If you want to test the executable program outside the development environment, create a directory UZipDotNet and copy the UZipDotNet.exe program into this directory and run the program. If you run the project from the Visual C# development environment, make sure you define a working directory in the Debug tab of the project properties. This program was developed using Microsoft Visual C# 2005. If you want to run it with Visual C# 2010, the development environment will convert it with no errors.

Start the program, you will see a screen similar to the one below.

The available options are:

6.1. Open

The open button allows you to open an existing ZIP file. The content of the ZIP archive central directory will be displayed on the screen. The Pos Hex check box allows you to display file position in hexadecimal. The ZIP archive is open for reading. Internally the ZIP archive is open by `InflateZipFile` class. If you want to update the archive (add or delete files), the system will display the Archive Update dialog box (see below). If you press OK the file will be reopened for update. Internally the ZIP archive will be controlled by `DeflateZipFile` class.

6.2. Extract

The extract button allows you to extract files from the ZIP archive to a directory. To extract a subset of files, select the files before clicking on the Extract button. The extract files dialog box defines the root folder for the extracted files and extraction options.

6.3. New

The new button allows you to define a name of a new empty ZIP file. After pressing the button, you will get a standard "Save As" .NET dialog box. Select a directory and a name for the new file. The New button will be renamed to Save. After you add files you will need to press the save button to append the central directory and the end of directory record to the ZIP archive.

6.4. Add

After the ZIP archive name is defined you need to add files or folders to be included in the archive. If the compression level is set to zero, the files will be stored in the archive. If the compression level is 1 to 3, the files are compressed with the deflate fast routine. If the compression level is 4 to 9, the files are compressed with the deflate slow routine. The default is 6. When you add files, the file area of the zip archive is being updated. The central directory will be saved at the end of the process when the Save button is pressed.

6.5. Delete

The Delete button allows you to delete files from the archive. First select the files to be deleted and press the Delete button. The delete process deletes the file entry in the memory array. The file itself will be removed from the archive when the Save button in pressed.

6.6. Test

The Test button allows you to test the compression and decompression classes. This button has no effect on the open ZIP archive. It allows a developer to take any file compress it and then decompress it and compare the input and decompressed files. The screen has three options: compression level, compressed file type and byte by byte compare of the input file and the decompressed file. If the compression level is set to zero, the files will be stored in the archive. If the compression level is 1 to 3, the files are compressed with the deflate fast routine. If the compression level is 4 to 9, the files are compressed with the deflate slow routine. The default is 6. The ZIP file type option will produce a standard ZIP archive with one compress file in it. The ZLIB file type will add a 16 bit header before the compressed file content and a 32 bit Adler32 checksum at the end of the file. The No Header file type is the compressed file data with no header or trailer. If No Header is selected, the program cannot decompress a file that was Stored. In other words, compression level zero or incompressible file. If you want the program to compare byte by byte the input file and the decompressed file, set the compare files check box. To start the test press the Open button. A standard .net open file dialog box will be displayed. Select a file and press OK. If for example you select *TestFile.txt* as your input file, the compressed file will be saved in the working directory of the program. The compressed file name will be *TestFile.zip* (or.zlib or .def). The program will decompress the compressed file and save the result in *TestFileDecomp.txt*. The input file and the decompressed file should be identical. The results of this process will be displayed on the screen.

7. Using the Compression/Decompression Classes Source Code

The `TestForm` class was developed to test the compression/decompression classes. As such it is an example of how to call these classes. The previous paragraph (Test Button) describes how to use the `TestForm` class. The program supports three types of files: ZIP, ZLIB and No Header. Blow you will find a description of how include the compression and decompression classes in your code for each of these file types.

7.1. No Header File Type

Compression using No Header file type allows you to compress a single file. The result is a compressed file with no header or trailer. To compress a file, create an instant of the `DeflateNoHeader` class. Call the constructor with compression level value (0 to 9). Calling the constructor with no argument will set the level to 6 the default level. Next call the `Compress` method with input file name and output file name. Please note: the output file can be a compressed file or a stored file (exact copy of the input file). After compression you should check the `CompFunction` property.

C#

```
// create compressiom object
// CompLevel is compression level number 0 to 9. Default is 6
// calling DeflateNoHeader with no argument result in CompLevel=6 the default level
DeflateNoHeader Def = new DeflateNoHeader(CompLevel);

// compress file
// return value: false=no error, true=error
if(Def.Compress(InputFileName, OutputFileName))
         {
         // ExceptionStack is Array of Strings
         // ExceptionStatck[0] is the error message
         // All other strings are stack entries from UZipDotNet namespace
         MessageBox.Show("Compress file Error\n" + Def.ExceptionStack[0] + "\n" +
Def.ExceptionStack[1]);
         return;
         }

// InflateNoHeader cannot decompress stored file
// you must check CompFunction to make sure the file will not be decompressed
if(Def.CompFunction == DeflateMethod.CompFunc.Stored)
         {
         // set a flag to indicate that the output is the same as the input file.
         // in other words you cannot decompress the result
         }
```

To decompress a file create an instant of the `InflateNoHeader` class. Next, call the `Decompress` method with input file name and output file name. You can only decompress file that was compressed. Calling the `Decompress` method with a stored file will result in an error.

C#

```
// create decompression object
InflateNoHeader Inf = new InflateNoHeader();

// decompress the file
// return value: false=no error, true=error
if(Inf.Decompress(InputFileName, OutputFileName))
         {
         // ExceptionStack is Array of Strings
         // ExceptionStatck[0] is the error message
         // All other strings are stack entries from UZipDotNet namespace
         MessageBox.Show("Decompress file Error\n" + Inf.ExceptionStack[0] + "\n" +
Inf.ExceptionStack[1]);
         return;
         }
```

7.2. ZLIB File Type

Compression using ZLIB file type allows you to compress a single file. The result is a compressed file with a header and a checksum trailer. The header contains a compression method flag. Either `Deflate` or `Stored`. The checksum trailer is Adler32 checksum. This file type has two significant advantages over No Header. The header has the information for the decompressing program as to the method of compression. The second advantage is the checksum. The decompressing program can verify the integrity of the decompressed file. To compress a file create an instant of the DeflateZLib class. Call the constructor with compression level value (0 to 9). Calling the constructor with no argument will set the level to 6 the default level. Next call the Compress method with input file name and output file name.

C#

```
// create decompression object
InflateNoHeader Inf = new InflateNoHeader();

// decompress the file
// return value: false=no error, true=error
if(Inf.Decompress(InputFileName, OutputFileName))
         {
         // ExceptionStack is Array of Strings
         // ExceptionStatck[0] is the error message
         // All other strings are stack entries from UZipDotNet namespace
         MessageBox.Show("Decompress file Error\n" + Inf.ExceptionStack[0] + "\n" +
Inf.ExceptionStack[1]);
         return;
         }
```

To decompress a file create an instant of the `InflateZLib` class. Next, call the `Decompress` method with input file name and output file name.

C#

```
// create decompression object
InflateZLib Inf = new InflateZLib();

// decompress the file
// return value: false=no error, true=error
if(Inf.Decompress(InputFileName, OutputFileName))
         {
         // ExceptionStack is Array of Strings
         // ExceptionStatck[0] is the error message
         // All other strings are stack entries from UZipDotNet namespace
         MessageBox.Show("Decompress file Error\n" + Inf.ExceptionStack[0] + "\n" +
Inf.ExceptionStack[1]);
         return;
         }
```

7.3. ZIP File Type

Compression using ZIP file type allows you to compress multiple files into one archive. The result is a file made of three parts: files area, central directory and end of directory record. The ZIP file format is described above in 5. ZIP File section. To compress files, create an instant of the `DeflateZipFile` class. Call the constructor with compression level value (0 to 9). Calling the constructor with no argument will set the level to 6 the default level. Next call the `CreateArchive` method with archive (output) file name. The file name should have .ZIP file extension. The next step is to add files to the archive. Call the Compress method one or more times for the files to be added. There are two arguments to the Compress method: FullFileName and ArchiveFileName. The FullFileName is the full name of the file on the local system. The FullFileName can be absolute or relative. The `ArchiveFileName` is the name of the file in the archive directory. `ArchiveFileName` can be just a file name or a file name with path. However, `ArchiveFileName` cannot start with a drive letter, a server name or a backslash. Each time you call the Compress method, the program compresses the file and appends the result to the files area of the ZIP archive. The central directory information for this file is saved in memory. After all files are

compressed, call the SaveArchive method. This method will append the central directory information and the end of directory record to the file. You now have a standard ZIP archive that can be decompressed by Windows or any other unzip program.

The `DeflateZipFile` class has additional public methods:

- `OpenArchive` allows you to open existing ZIP archive for updating.
- `SaveDirectoryPath` allows you to add directory paths to the central directory. No compression is involved. These entries are useful to recreate directory structures while the archive is being decompressed. Directory entries are not needed if the directories are not empty. If directory structure is to be created with empty directories, these entries must be present.
- `Delete` method allows you to remove entry from the central directory. The file itself will be deleted from the archive at the time the `SaveArchive` method is called.
- `ClearArchive` allows you to close the output file and delete it. It is useful when a user creates a new archive and before saving the archive decides to cancel the operation.
- IsOpen will return true if the `DeflateZipFile` exists and has an open output file.

The `DeflateZipFile` class exposes the following public properties:

- `ArchiveName` the name of the open ZIP file archive.
- `IsEmpty` returns true if the archive is not open or the central directory is empty.
- `ExceptionStack` is a string array. If an exception is thrown during processing, the class will save the exception message in element zero and the exception stack entries that are from `UZipDotNet` name space.
- ZipDir the list of central directory entries.

C#

```
// create compressiom object
// CompLevel is compression level number 0 to 9. Default is 6
// calling DeflateNoHeader with no argument result in CompLevel=6 the default level
DeflateZipFile Def = new DeflateZipFile(CompLevel);

// create empty zip file
if(Def.CreateArchive(CompFileName))
         {
         // ExceptionStack is Array of Strings
         // ExceptionStatck[0] is the error message
         // All other strings are stack entries from UZipDotNet namespace
         MessageBox.Show("Compress file Error\n" + Def.ExceptionStack[0] + "\n" +
Def.ExceptionStack[1]);
         return;
         }

// compress one file
if(Def.Compress(FullFileName, ArchiveFileName))
         {
         // ExceptionStack is Array of Strings
         // ExceptionStatck[0] is the error message
         // All other strings are stack entries from UZipDotNet namespace
         MessageBox.Show("Compress file Error\n" + Def.ExceptionStack[0] + "\n" +
Def.ExceptionStack[1]);
         return;
         }

// save archive
if(Def.SaveArchive())
         {
         // ExceptionStack is Array of Strings
         // ExceptionStatck[0] is the error message
         // All other strings are stack entries from UZipDotNet namespace
         MessageBox.Show("Compress file Error\n" + Def.ExceptionStack[0] + "\n" +
Def.ExceptionStack[1]);
         return;
         }
```

To decompress a file create an instant of the `InflateZipFile` class. Call the `OpenZipFile` method with input file name. The class will open the ZIP file and read the central directory. Call one or more times the `DecompressZipFile` method for each file you want to extract from the archive. The first argument is one of the elements of the `ZipDir` array. This array holds the directory of the archive.

The `InflateZipFile` class has additional public methods:

- `ExtractAll` allows you to open existing ZIP archive and extract all files to a directory of your choice.
- `IsOpen` will return true if the InflateZipFile exists and has an open input file.

The `DeflateZipFile` class exposes the following public properties:

- `ArchiveName` the name of the open ZIP file archive.
- `IsEmpty` returns true if the archive is not open or the central directory is empty.
- `ExceptionStack` is a string array. If an exception is thrown during processing, the class will save the exception message in element zero and the exception stack entries that are from `UZipDotNet` name space.
- `ZipDir` the list of central directory entries.
- `ZipDirPosition` the position of the ZIP central directory within the archive.
- `ReadTotal` the size of the current compressed file. The size does not include the file header
- `WriteTotal` the size of the current decompressed file.

C#

```
// create decompressiom objectt
InflateZipFile Inf = new InflateZipFile();

// open a zip archive
if(Inf.OpenZipFile(ReadFileName))
         {
         // ExceptionStack is Array of Strings
         // ExceptionStatck[0] is the error message
         // All other strings are stack entries from UZipDotNet namespace
         MessageBox.Show("Decompress file Error\n" + Inf.ExceptionStack[0] + "\n" +
Inf.ExceptionStack[1]);
         return;
         }

// decompress one file
// Inf.ZipDir[n] is one of the file headers loaded by the OpenZipFile menthod
// RootPathName is the path name of the destination folder
// NewFileName is either null or a new name for the extracted file. If null, the original name is
used
// CreatePath is a boolean if true the system will create a path if it does not exist
// OverWrite is a boolean if true the system will overwrite existing file withe the same name.
// If false and a file withe the same name exist, the operation will be aborted with application
exception
if(Inf.DecompressZipFile(Inf.ZipDir[n], RootPathName, NewFileName, CreatePath, OverWrite))
         {
         // ExceptionStack is Array of Strings
         // ExceptionStatck[0] is the error message
         // All other strings are stack entries from UZipDotNet namespace
         MessageBox.Show("Compress file Error\n" + Inf.ExceptionStack[0] + "\n" +
Inf.ExceptionStack[1]);
         return;
         }

// close zip file
Inf.CloseZipFile();
```

8. Source Code Samples Of Visual C# controls

If you are looking for code samples for some of the Visual C# controls, here is a list of controls used within this application:

- `OpenFileDialog`, `SaveFileDialog`, `SendToRecycleBin`
- `DataGridView`, `DataGridViewSortCompareEventHandler`, `TreeView`, `ListView`, `ListBox`, `RichTextBox`
- `SplitterPanel`, `NumericUpDown`
- `XmlSerializer`, `XmlTextWriter`, `XmlTextReader`
- `FileSystemWatcher`, `FileSystemEventHandler`, `RenamedEventHandler`
- `DriveInfo`, `FileInfo`, `DirectoryInfo`
- `Timer`, `EventHandler`
- `Exception`, `ApplicationException`

9. Points of Interest

For most of my career I used computer languages that compile directly into machine code. Before using C# I did my development with MASM, C and C++. On the transition to C# I was not sure about the performance of the .net architecture. Since the compression process is very CPU intensive I decided to convert the C# code to C++ and see the speed difference. I was pleasantly surprised. The difference was very small.

10. Other Open Source Software by This Author

- Android Color Selector for Programmers is a free App available on Google Play.
- Google Play In-App Billing Demo App This article is an example of Google Play In-App Billing Version 3.
- PDF File Writer C# Class Library
  Prize winner in Competition "Best overall article of April 2013"
  Prize winner in Competition "Best C# article of April 2013"
  PDF File Writer is a C# class library allowing .NET applications to create PDF files.
- PDF File Analyzer With C# Parsing Classes
  PDF File Analyzer is designed to read, parse, and display the internal structure of PDF files.

11. History

- 2012/03/30: Version 1.0 Original revision.
- 2012/05/03: Version 1.1
  - InflateTree.cs
    - Method: InflateTree.BuildTree(...)
    - Remove program test exception. See note at the source code
  - ProcessFilesForm.cs
    - Method ProcessFilesForm.ExtractFile()
    - Fix error reporting after call to DecompressZipFile(...)
- 2012/10/23 Version 1.2
  - AddFilesAndFoldersForm.cs
    - Method: OnAddButton(...)
    - Fix problem related to adding a folder at the root directory of a drive

License

About the Author

Uzi Granot
Canada
No Biography provided

Comments and Discussions

56 messages have been posted for this article Visit **https://www.codeproject.com/Articles/359758/Processing-Standard-Zip-Files-with-Csharp-compress** to post and view comments on this article, or click **here** to get a print view with messages.