The Format of the Guild Wars 2 Archive File

Jon Dahm

March 25, 2014

Contents

Notes

Libraries

To my knowledge, there are two major C++ libraries for working with the Archive file. Github user Ahom has created a library for working with File Records and extracting images that you can find here. Github user Rhoot has created a library that will extract information from a large number of files within the Archive. You can find his work here. Most of the information in this document has come from these projects.

Endianness and Numbers

All numbers I list in this document are decimal (base 10) unless specified otherwise. Hexadecimal numbers are followed by a subscript x ($\texttt{1A}_x$). Sometimes a single byte will be listed as a character rather than a number. In these cases the value of that byte is the ASCII code of the character listed.

When I list values, sometimes I will list them as full numbers (like $\texttt{40CB}_x$) and sometimes I will list them as individual bytes (like [$\texttt{CB}_x$,$\texttt{40}_x$]). When I list the individual bytes, they are listed in the order they appear in the Archive. When I list them as full numbers, that is their actual value.

The Archive is arranged in little-endian format. This means that if you see a 16-bit value [$\texttt{CB}_x$,$\texttt{40}_x$], its actual value is $\texttt{40CB}_x$.

Disclaimer

I do not condone use of this document to modify the archive for any reason. Modifying the archive is a direct violation of the Terms of Service you agreed to follow when you bought the game.

1 File Records

This chapter will introduce you to the main portions of the Archive, from which you can find every file represented within. After reading this chapter, you should be able to produce a list of all files within the archive. Additionally, if a file within the archive is referenced by its ID, you should be able to retrieve it.

1.1 The Archive Header

The Archive begins with a 40-byte header which describes some of the properties of the Archive and points to the Main File Table. The format of this header can be found in Table 1.

Table 1: the Archive header

Byte	Size	Value	Description
0	1	Version	Version of the Archive. Seems to always be 97_x
1	3	Identifier	Identifies this file as the Archive file, as opposed to a MS Word file. Always $[45_x, 4E_x, 1A_x]$.
4	4	Header Size	Size of this header. Always 40.
8	4	(unknown)	Always $CABA0001_x$.
12	4	Chunk Size	Size of each chunk in the file. Always 512.
16	4	(unknown)[1]	Always $8ED0A720_x$.
20	4	(unknown)	Always 00040002_x.
24	8	MFT Offset	The offset from the beginning of the Archive to the Main File Table.
32	4	MFT Size	Size of the Main File Table in bytes.
36	4	(unknown)	Always 0.

1.2 The Main File Table

The Main File Table (MFT) is a list of all of the files in the Archive. Its structure begins with a 24-byte-long header, whose format is given in Table 2. The header is followed by a number of 24-byte entries that make up the

table. Each entry refers to a single file and some associated metadata. The entries are not listed in any particular order. See Table 3 for details.

The first fifteen entries in the MFT are reserved for special files in the Archive. They are documented below:

1	Archive Header
2	File ID Table (See Section 1.3)
3	MFT (self reference)
4–15	Blank Entries

Table 2: the MFT header

Byte	Size	Value	Description
0	4	Identifier	Identifies the start of the MFT. Always `['M','f','t',1A`$_x$`]`.
4	8	(unknown)	
12	4	Length	Number of entries in the table plus one.
16	8	(unknown)	Always 0.

Table 3: an MFT entry

Byte	Size	Value	Description
0	8	Offset	Offset from the beginning of the Archive to the start of the file.
8	4	Archived Size	Size in bytes of the file within the archive.
12	2	Compression	Type of compression the file is under. See below.
14	2	Flags	Other flags. See below.
16	4	(unknown)	Always 0.
20	4	(unknown)	Always `4867 4BC7`$_x$.

Valid values for Compression:

0	Uncompressed
8	Huffman Compression

Valid values for Flags:

1	In Use
2	(unknown)

1.3 The File ID Table

The File ID Table gives each file in the MFT an ID. Each entry in the table has the format listed in table 4. The entries are not listed in any particular order.

For the most part, each entry has only one ID. However, many have more than one ID each. As of the time of this writing, approximately a third of the files in the Archive have two IDs, and none have more. *More research must be done into why some entries have multiple IDs.*

Additionally, some entries may contain nil values for either field. I haven't found a significant number of these, but they exist. I have only found entries where both fields are nil, and none where only one was nil. My recommendation is to discard any entries with nil fields.

Table 4: a File ID Table entry

Byte	**Size**	**Value**	**Description**
0	4	File ID	
4	4	MFT Entry Index	Indices start at 1

2 Files and Compression

This chapter will introduce you to how to identify files and decompress files that have been compressed. Additionally, I'll discuss the compression used on many of the texture files in the Archive. After reading this chapter, you should be able to, given the address of the start of a file, provide its raw data, whether the file was compressed or not.

2.1 File Types

Every file starts with an 8-byte header identifying the type of file and how large it is. The first 4 bytes of the header are the file's type identifier, typically represented by four character codes (4CC). The second 4 bytes tell you how long the uncompressed file is, if the file is compressed.

In the latest version of the Archive at the time of this writing, 99% of the files were compressed. All of these files are represented in the general file header by one 4CC. To find the actual 4CC defining the file type, you have to decompress the file, which we will go over in the next section.

The following table describes all 4CCs that appear in the general file header, listed in decreasing order of frequency:

[08_x,00_x,01_x,80_x]	Compressed File
['A','T','E','X']	General Use Texture
['A','T','E','U']	UI Texture
['K','B','2','f']	(unknown)
['K','B','2','g']	(unknown)
[$7C_x$,$1A_x$,'I','z']	(unknown)
[97_x,'A','N',$1A_x$]	(unknown)

2.2 Compression

Note to self: Add illustrations.

Compression is a difficult subject to describe tersely. The compression used in the Archive is very similar to that produced by the DEFLATE algorithm. If you are familiar with the DEFLATE algorithm, you may notice them. To keep things (relatively) short, however, I won't describe every difference between the two.

Data is compressed using Huffman codes and back-copying. The former is a method of taking a set of data and compressing it as small as possible, and the latter is a method of further compressing the data by replacing reoccuring data with a refrence to the last time it occured. I won't go into

the details of how all this works, so if you aren't familiar with either of these, read this fantastic article on zlib which does a wonderful job explaining the concepts. Be sure to understand these concepts well before continuing in this section, or you will be lost. If this is well beyond you, and you don't care particularly about implementing a decompression algorithm yourself, just use Ahom's decompression algorithm and skip the rest of this chapter.

To begin, it is incredibly important to note the order in which bits are read. Strangely enough, bytes aren't read from beginning to end — instead, they are split into little-endian 32-bit values, and read from highest bit to lowest. For illustration, see Figure BLAH. When I refer to ordering of elements in this section, I assume that bits are being read in this order.

Next, every 64KiB, 4 bytes are skipped. As of the writing of this document, I am unaware the purpose of this. I would guess that those 4 bytes are a check on the previous data in order to help detect corruption.

The compressed data starts with a single byte that represents an adjustment to any back-copy sizes encountered in the data. This should be saved for later use. The rest of the data is split into blocks.

Each block begins with two Huffman Trees describing the Huffman codes for the literal/copy-length alphabet and the copy-offset alphabet. These are followed by 4 bits which represent the number of codes from the first alphabet to expect in this section. The rest of the block is the Huffman codes representing the information compressed in this section.

In the next subsection, I'll describe how you generate Huffman codes from the Huffman Trees presented in each block.

2.3 Huffman Trees

Each tree can represent a variable number of values. The first 16 bits are an unsigned value representing how many values this tree is giving Huffman codes to. This is followed by a number of entries describing sometimes several codes at once. These entries are compressed using predefined codes found in Appendix A.

Each entry represents at least one value and its code. The first entry refers to the highest values the tree represents, with each successive entry referring to a lower value. The highest three bits state how many more values this entry applies to. The lowest 5 bits state how long the Huffman codes are for these values. If the length is 0, those values aren't actually represented in the tree, and you can skip over them.

As it turns out, in order to generate a valid Huffman code for a value, all you need to know is how long the Huffman code for it is. The following

algorithm derives the Huffman codes for all values whose lengths you know are non-zero.

Sort all of your value+code-length pairs first in ascending order of length, then in ascending order of value. Assign the first value a code of all 1's. For each successive value that uses the same length code, decrement the code by one. When you reach a value that uses more code bits, multiply the last code by 2 and then subtract one. Continue this process until you have assigned each value a Huffman code.

The Tree representing the literal/copy-length alphabet cannot have more than 285 values in it. The Tree representing the copy-offset alphabet cannot have more than 34 values.

2.4 Translating Huffman Codes to Data

In each block, after the Huffman Trees, there are 4 bits describing how many codes from the literal/copy-length alphabet there in the block. The number is determined by adding one to the value of the 4 bits and then multiplying by 1000_x. If the end of the file has been reached, then this number may be greater than the actual number of codes, so you'll have to watch to make sure you don't overshoot the end of the stream.

There are two modes to translating the codes to data — literal, where each code matches one byte, and copy, where extra data follows the code describing how many bytes to copy from where in the output stream generated so far. If the value of the code translated is less than 100_x, then the output is a byte with that value. If the value is greater than 100_x, then you have to copy previous output back into the stream.

Following a copy code are additional bits that add to the length represented by the code itself. Table 5 provides the base lengths for each value and how many additional bits you must read and add to the base length.

After that is a code from the copy-offset alphabet. This also has additional bits following it to add to it. Table 6 details the base offsets and the number of additional bits for each value.

To calculate the total length of the copy, add the base length, the value of the additional length bits, and the copy size adjustment value from the beginning of the file. To calculate the total offset of the copy, add the base offset and the additional offset bits. It may be helpful to note that the sliding window on this algorithm appears to be 128KiB.

Table 5: Copy Length Table

Code	Base	Additional Bits		Code	Base	Additional Bits
100_x	1	0		110_x	33	3
101_x	2	0		111_x	41	3
102_x	3	0		112_x	49	3
103_x	4	0		113_x	57	3
104_x	5	0		114_x	65	4
105_x	6	0		115_x	81	4
106_x	7	0		116_x	97	4
107_x	8	0		117_x	113	4
108_x	9	1		118_x	129	5
109_x	11	1		119_x	161	5
$10A_x$	13	1		$11A_x$	193	5
$10B_x$	15	1		$11B_x$	225	5
$10C_x$	17	2		$11C_x$	256	0
$10D_x$	21	2				
$10E_x$	25	2				
$10F_x$	29	2				

Table 6: Copy Offset Table

Code	Base	Additional Bits		Code	Base	Additional Bits
0_x	1_x	0		12_x	201_x	8
1_x	2_x	0		13_x	301_x	8
2_x	3_x	0		14_x	401_x	9
3_x	4_x	0		15_x	601_x	9
4_x	5_x	1		16_x	801_x	10
5_x	7_x	1		17_x	$C01_x$	10
6_x	9_x	2		18_x	1001_x	11
7_x	D_x	2		19_x	1801_x	11
8_x	11_x	3		$1A_x$	2001_x	12
9_x	19_x	3		$1B_x$	3001_x	12
A_x	21_x	4		$1C_x$	4001_x	13
B_x	31_x	4		$1D_x$	6001_x	13
C_x	41_x	5		$1E_x$	8001_x	14
D_x	61_x	5		$1F_x$	$C001_x$	14
E_x	81_x	6		20_x	10001_x	15
F_x	$C1_x$	6		21_x	18001_x	15
10_x	101_x	7				
11_x	181_x	7				