

The Format of the Guild Wars 2 Archive File

Jon Dahm

May 3, 2014

Contents

Notes	iii
1 File Records	1
1.1 The Archive Header	1
1.2 The Main File Table	1
1.3 The File ID Table	3
2 Files and Compression	5
2.1 File Types	5
2.2 File Compression	6
2.2.1 Huffman Trees	7
2.2.2 Translating Huffman Codes to Data	7
3 Uncompressed Files	10
4 Compressed Files	11
5 Pack Files	12
5.1 Headers	12
5.2 Common Structures	13
5.3 Animation Sequence Pack File	15
5.4 Audio Pack File	15
5.5 Bank Pack File	15
5.6 Bank Index Pack File	16
5.7 Collide Model Manifest Pack File	16
5.8 Composite Pack File	16
5.9 Content Manifest Pack File	16
5.10 Emote Animation Pack File	16
5.11 EULA Pack File	16
5.12 Havok Pack File	17

5.13 Map Pack File	17
5.13.1 Unknown Chunk	17
5.13.2 Unknown Chunk	17
5.13.3 Unknown Chunk	17
5.13.4 Unknown Chunk	17
5.13.5 Unknown Chunk	17
5.13.6 Unknown Chunk	17
5.13.7 Unknown Chunk	17
5.13.8 Unknown Chunk	17
5.13.9 Unknown Chunk	17
5.13.10 Unknown Chunk	18
5.13.11 Unknown Chunk	18
5.13.12 Unknown Chunk	18
5.13.13 Unknown Chunk	18
5.13.14 Unknown Chunk	18
5.13.15 Unknown Chunk	18
5.13.16 Unknown Chunk	18
5.13.17 Unknown Chunk	18
5.14 Map Metadata Pack File	18
5.15 Map Shadow Pack File	18
5.16 Material Pack File	19
5.16.1 Unknown Chunk (GRMT)	19
5.16.2 Unknown Chunk (DX9S)	19
5.17 Model Pack File	19
5.17.1 Unknown Chunk	19
5.17.2 Unknown Chunk	20
5.17.3 Unknown Chunk	20
5.17.4 Unknown Chunk	20
5.17.5 Unknown Chunk	20
5.17.6 Unknown Chunk	20
5.18 Paged Image Pack File	20
5.19 Portal Manifest Pack File	20
5.20 Scene Pack File	20
5.21 Script Pack File	20
5.22 Text Manifest Pack File	20
5.23 Text Variant Pack File	21
5.24 Text Voice Pack File	21

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 ($1A_x$). 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 $40CB_x$) and sometimes I will list them as individual bytes (like $[CB_x, 40_x]$). 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 $[CB_x, 40_x]$, its actual value is $40CB_x$.

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.

Chapter 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 1.1](#).

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 1.2](#). 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 1.3](#) 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	Reserved Entries (blank)

Table 1.1: the Archive header

Type	Name	Description
FourCC	Identifier	Identifies this file as the Archive. Always $97_x, 'A', 'N', 1A_x$.
uint32	Header Size	Always 40.
uint32	(unknown)	Always $CABA\ 0001_x$.
uint32	Chunk Size	Size of each chunk in the file. Always 512.
uint32	(unknown)	Always $8ED0\ A720_x$.
uint32	(unknown)	
uint64	MFT Offset	The offset from the beginning of the Archive to the Main File Table.
uint32	MFT Size	Size of the Main File Table in bytes.
uint32	(unknown)	Always $0000\ 0000_x$.

Table 1.2: the MFT header

Type	Name	Description
FourCC	Identifier	Always $['M', 'f', 't', 1A_x]$.
uint64	(unknown)	
uint32	Length	Number of entries in the table plus one.
uint64	(unknown)	Always $0000\ 0000_x$.

Table 1.3: an MFT entry

Type	Name	Description
uint64	Location	Offset from the beginning of the Archive to the start of the file.
uint32	Archived Size	Size in bytes of the file within the archive.
uint16	Compression	Type of compression the file is under. See below.
uint16	Flags	Other details about the file. See below.
uint32	(unknown)	Always 0000 0000 _x .
uint32	(unknown)	Always 4867 4BC7 _x .

Valid values for Compression:

0 Uncompressed

8 Huffman Compressed

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 1.4](#). 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 1.4: a File ID Table entry

Type	Name	Description
uint32	File ID	
uint32	MFT Entry Index	Indices start at 1

Chapter 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 Textur
['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 File Compression

TODO: 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 reoccurring data with a reference to the last time it occurred. 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.

2.2.1 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.

2.2.2 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 2.1](#) provides the base lengths for each value and how many additional bits you must read and add to the base length.

Table 2.1: 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			

After that is a code from the copy-offset alphabet. This also has additional bits following it to add to it. [Table 2.2](#) 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 2.2: 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			

Chapter 3

Uncompressed Files

TODO: Fill out this chapter.

Chapter 4

Compressed Files

TODO: Fill out this chapter.

Chapter 5

Pack Files

This chapter will introduce you to one file type – The Pack File. This file type is used to store a large portion of the game data, and has many subtypes for data such as animations, models, maps, textures, and audio. You’ll learn how to navigate through the pack file to grab individual portions of the data it holds, as well as what data each subtype contains.

5.1 Headers

Each Pack File begins with a 12-byte header identifying the type of data contained. This header includes the 4-byte Character Code identifier labeling the file as a Pack File. The format of this header can be found in [Table 5.1](#).

Table 5.1: the Pack File header

Type	Name	Description
char[2]	Identifier	Always [‘P’, ‘F’]
uint32	Unknown	Always 01 _x
uint16	Header Size	Always 12
FourCC	Type	4 Character Codes defining the data held in this Pack File.

Each Pack File is split into blocks of data, called Chunks. The first chunk follows immediately after the Pack File header. Each one contains a pointer to the next one.

Chunks are identified by 4 character codes, which determines the format

of the chunk and the data stored in it. While different Pack Files may share chunks with the same identifier, these chunks may not be the same. For example, the Content Manifest and Map Metadata Pack Files both have a chunk labeled 'Main,' but their formats are different.

Chunks begin with a 16-byte header describing where the next chunk is in the file and the type of data stored in this chunk. The format can be found in [Table 5.2](#).

Table 5.2: the Chunk header

Type	Name	Description
FourCC	Identifier	Type of Chunk
uint32	Next Chunk	Number of bytes after the end of this field that the next chunk appears
uint16	Unknown	
uint16	Header Size	Always 12
Ptr<?>	Unknown	Pointer to an unknown data structure

In [Table 5.3](#) you will find all the known Pack File types, as well as the chunks each one contains. The following sections will go into each chunk and its format.

5.2 Common Structures

Before we go into the different Pack File types and their formats, we should go over some common structures found in the pack files. There are three basic structures and three others that build off of them.

We'll start with the Pointer. In the documentation, we'll shorten it to Ptr for brevity. This structure consists of one 32-bit offset that references data elsewhere in the file. The address of the data referenced is the address of the Ptr plus the offset.

The next basic structure is the Array – shortened to Arr. It is simply one data type T repeated N times. It will be written as Arr<T,N>, T being the data type and N being the number of data types. It takes up $size(T) * N$ bytes.

The last basic structure is the File Reference – shortened to FileRef. It occupies 6 bytes, but only 4 of those bytes are useful information. According to Johan Skld, this structure represents a File ID, and it is only valid if the first two 16-bit values are both above 100_x. In his project, he calculates the

Table 5.3: Pack File Types sorted by FourCC

FourCC	Value	Name	Included Chunks
['A', 'B', 'I', 'X']	58494241 _x	Bank Index	['B', 'I', 'D', 'X']
['A', 'B', 'N', 'K']	4b4e4241 _x	Bank	['B', 'K', 'C', 'K']
['A', 'M', 'A', 'T']	54414d41 _x	Material	['G', 'R', 'M', 'T'], ['D', 'X', '9', 'S']
['A', 'M', 'S', 'P']	50534d41 _x	Script	['A', 'M', 'S', 'P']
['a', 'n', 'i', 'c']	63696E61 _x	Animation Sequence	['s', 'e', 'q', 'n']
['A', 'S', 'N', 'D']	444e5341 _x	Audio	['A', 'S', 'N', 'D']
['C', 'I', 'N', 'P']	504e4943 _x	Scene	['C', 'S', 'C', 'N']
['c', 'm', 'a', 'C']	43616d63 _x	Collide Model Manifest	['m', 'a', 'i', 'n']
['c', 'm', 'p', 'c']	63706d63 _x	Composite	['c', 'o', 'm', 'p']
['c', 'n', 't', 'c']	63746e63 _x	Content Manifest	['M', 'a', 'i', 'n']
['e', 'm', 'o', 'c']	636f6d65 _x	Emote Animation	['a', 'n', 'i', 'm']
['e', 'u', 'l', 'a']	616c7565 _x	EULA	['e', 'u', 'l', 'a']
['h', 'v', 'k', 'C']	436b7668 _x	Havok	['h', 'a', 'v', 'k']
['m', 'a', 'p', 'c']	6370616d _x	Map	['a', 'u', 'd', 'i'], ['m', 's', 'n', '00 _x '], ['p', 'a', 'r', 'm'], ['s', 'h', 'o', 'r'], ['s', 'u', 'r', 'f'], ['t', 'r', 'n', 'i'], ['a', 'r', 'e', 'a'], ['h', 'a', 'v', 'k'], ['c', 'u', 'b', 'e'], ['d', 'c', 'a', 'l'], ['e', 'n', 'v', '00 _x '], ['l', 'g', 'h', 't'], ['p', 'r', 'p', '2'], ['r', 'i', 'v', 'e'], ['s', 'h', 'e', 'x'], ['t', 'r', 'n', '00 _x '], ['z', 'o', 'n', '2']
['m', 'M', 'e', 't']	74654d6d _x	Map Metadata	['M', 'a', 'i', 'n']
['M', 'O', 'D', 'L']	4c444f4d _x	Model	['A', 'N', 'I', 'M'], ['M', 'O', 'D', 'L'], ['G', 'E', 'O', 'M'], ['P', 'R', 'P', 'S'], ['R', 'O', 'O', 'T'], ['S', 'K', 'E', 'L']
['m', 'p', 's', 'd']	6473706d _x	Map Shadow	['s', 'h', 'a', 'd']
['P', 'I', 'M', 'G']	474d4950 _x	Paged Image	['P', 'G', 'T', 'B']
['p', 'r', 'l', 't']	746c7270 _x	Portal Manifest	['m', 'f', 's', 't']
['t', 'x', 't', 'm']	6d747874 _x	Text Manifest	['t', 'x', 't', 'm']
['t', 'x', 't', 'V']	56747874 _x	Text Variant	['v', 'a', 'r', 'i']
['t', 'x', 't', 'v']	76747874 _x	Text Voice	['t', 'x', 't', 'v']

File ID as $first - FF_x + (second - 100_x) * FF00_x$. *More research must be done into whether this equation is correct.*

Moving on to composite structures, the first we'll discuss is the String and Wide String – shortened to Str and WStr, respectively. These are simply Ptrs to C Strings using either traditional or wide characters, but are common enough that they warrant their own listing here.

The next composite structure is the Vector – shortened to Vtr. This is similar to an array, but the size is determined at run time and not at compile time, and it always takes up 8 bytes regardless of the number of elements it represents. The first 32-bit value in this structure, N, is the number of elements in the Vector. The second is a Pointer of type $Ptr<Arr<T,N>>$, where T is the data type represented by the Vector.

The last composite structure we'll discuss is the Reference List – shortened to RefList. This is a Vtr of pointers to data types. It is equivalent to $Vtr<Ptr<T>>$, where T is the data type we're pointing to.

5.3 Animation Sequence Pack File

TODO: Finish this section

5.4 Audio Pack File

TODO: Finish this section

5.5 Bank Pack File

This type has only one chunk, whose FourCC is ['B', 'K', 'C', 'K']. It has 3 fields, as documented in [Table 5.4](#).

Table 5.4: the BKCK chunk format

Type	Name	Description
byte[16]	(unknown)	
Vtr<Sound>	Sounds (see Table 5.5)	
Ptr<byte[16]>	(unknown)	

TODO: Research the purpose of these fields and this file.

Table 5.5: the Sound structure format (ABNK)

Type	Name	Description
uint32	Voice ID	
uint32	Flags	
byte[16]	(unknown)	
float32	Length	
float32	Offset	
byte[16]	(unknown)	
Vtr<byte>	Audio Data	

5.6 Bank Index Pack File

This Pack File has a single chunk whose FourCC is ['B','I','D','X']. There is only one field in this chunk, and that is a Vtr<Translation>. Translation is a structure with a single field of type RefList<FileRef>.

TODO: Research the purpose of this file and what it represents.

5.7 Collide Model Manifest Pack File

TODO: Finish this section

5.8 Composite Pack File

TODO: Finish this section

5.9 Content Manifest Pack File

TODO: Finish this section

5.10 Emote Animation Pack File

TODO: Finish this section

5.11 EULA Pack File

TODO: Finish this section

5.12 Havok Pack File

TODO: Finish this section

5.13 Map Pack File

TODO: Finish this section

5.13.1 Unknown Chunk

TODO: Finish this section

5.13.2 Unknown Chunk

TODO: Finish this section

5.13.3 Unknown Chunk

TODO: Finish this section

5.13.4 Unknown Chunk

TODO: Finish this section

5.13.5 Unknown Chunk

TODO: Finish this section

5.13.6 Unknown Chunk

TODO: Finish this section

5.13.7 Unknown Chunk

TODO: Finish this section

5.13.8 Unknown Chunk

TODO: Finish this section

5.13.9 Unknown Chunk

TODO: Finish this section

5.13.10 Unknown Chunk

TODO: Finish this section

5.13.11 Unknown Chunk

TODO: Finish this section

5.13.12 Unknown Chunk

TODO: Finish this section

5.13.13 Unknown Chunk

TODO: Finish this section

5.13.14 Unknown Chunk

TODO: Finish this section

5.13.15 Unknown Chunk

TODO: Finish this section

5.13.16 Unknown Chunk

TODO: Finish this section

5.13.17 Unknown Chunk

TODO: Finish this section

5.14 Map Metadata Pack File

TODO: Finish this section

5.15 Map Shadow Pack File

TODO: Finish this section

5.16 Material Pack File

The Material Pack File has two chunks – [‘G’, ‘R’, ‘M’, ‘T’] and [‘D’, ‘X’, ‘9’, ‘S’].

5.16.1 Unknown Chunk (GRMT)

This chunk has 11 fields:

Table 5.6: GRMT Chunk Format

Type	Name	Description
byte	Texture Array Range	
byte	Texture Count	
byte	Texture Transformation Range	
byte	Sort Order	
byte	Sort Triangles	
byte	Process Animations	
uint32	Debug Flags	
uint32	Flags	
uint32	Texture Type	
Arr<uint32,4>	Texture Masks	
Vtr<uint64>	Texture Tokens	

5.16.2 Unknown Chunk (DX9S)

TODO: Finish this section

5.17 Model Pack File

TODO: Finish this section

5.17.1 Unknown Chunk

TODO: Finish this section

5.17.2 Unknown Chunk

TODO: Finish this section

5.17.3 Unknown Chunk

TODO: Finish this section

5.17.4 Unknown Chunk

TODO: Finish this section

5.17.5 Unknown Chunk

TODO: Finish this section

5.17.6 Unknown Chunk

TODO: Finish this section

5.18 Paged Image Pack File

TODO: Finish this section

5.19 Portal Manifest Pack File

TODO: Finish this section

5.20 Scene Pack File

TODO: Finish this section

5.21 Script Pack File

TODO: Finish this section

5.22 Text Manifest Pack File

TODO: Finish this section

5.23 Text Variant Pack File

TODO: Finish this section

5.24 Text Voice Pack File

TODO: Finish this section

Appendix A

Static Huffman Trees

The static tree used when defining trees for decompressing files:

Value	Huffman Code	Number of Bits
08 _x	111 _b	3
09 _x	110 _b	3
0A _x	101 _b	3
00 _x	1001 _b	4
07 _x	1000 _b	4
0B _x	0111 _b	4
0C _x	0110 _b	4
06 _x	01011 _b	5
29 _x	01010 _b	5
2A _x	01001 _b	5
E0 _x	01000 _b	5
04 _x	001111 _b	6
05 _x	001110 _b	6
20 _x	001101 _b	6
28 _x	001100 _b	6
2B _x	001011 _b	6
2C _x	001010 _b	6
40 _x	001001 _b	6
4A _x	001000 _b	6
03 _x	0001111 _b	7
0D _x	0001110 _b	7
25 _x	0001101 _b	7
26 _x	0001100 _b	7
27 _x	0001011 _b	7

48_x	0001010_b	7
49_x	0001001_b	7
24_x	00010001_b	8
47_x	00010000_b	8
$4B_x$	00001111_b	8
$4C_x$	00001110_b	8
69_x	00001101_b	8
$6A_x$	00001100_b	8
23_x	000010111_b	9
46_x	000010110_b	9
60_x	000010101_b	9
63_x	000010100_b	9
67_x	000010011_b	9
68_x	000010010_b	9
88_x	000010001_b	9
89_x	000010000_b	9
$A0_x$	000001111_b	9
$E8_x$	000001110_b	9
01_x	0000011011_b	10
02_x	0000011010_b	10
$2D_x$	0000011001_b	10
43_x	0000011000_b	10
44_x	0000010111_b	10
45_x	0000010110_b	10
65_x	0000010101_b	10
66_x	0000010100_b	10
80_x	0000010011_b	10
87_x	0000010010_b	10
$8A_x$	0000010001_b	10
$A8_x$	0000010000_b	10
$A9_x$	0000001111_b	10
$C0_x$	0000001110_b	10
$C9_x$	0000001101_b	10
$E9_x$	0000001100_b	10
$0E_x$	00000010111_b	11
$4D_x$	00000010110_b	11
64_x	00000010101_b	11
$6B_x$	00000010100_b	11
$6C_x$	00000010011_b	11

84_x	00000010010 _b	11
85_x	00000010001 _b	11
$8B_x$	00000010000 _b	11
$A4_x$	00000001111 _b	11
$A5_x$	00000001110 _b	11
AA_x	00000001101 _b	11
$C8_x$	00000001100 _b	11
$E5_x$	00000001011 _b	11
83_x	000000010101 _b	12
86_x	000000010100 _b	12
$A6_x$	000000010011 _b	12
$A7_x$	000000010010 _b	12
$C7_x$	000000010001 _b	12
CA_x	000000010000 _b	12
$E7_x$	00000001111 _b	12
22_x	0000000011101 _b	13
$2E_x$	0000000011100 _b	13
$8C_x$	0000000011011 _b	13
$C4_x$	0000000011010 _b	13
$E4_x$	0000000011001 _b	13
$E6_x$	0000000011000 _b	13
$4E_x$	00000000101111 _b	14
$6D_x$	00000000101110 _b	14
$C6_x$	00000000101101 _b	14
EC_x	00000000101100 _b	14
$0F_x$	000000001010111 _b	15
10_x	000000001010110 _b	15
11_x	000000001010101 _b	15
$8D_x$	000000001010100 _b	15
AB_x	000000001010011 _b	15
AC_x	000000001010010 _b	15
CC_x	000000001010001 _b	15
EA_x	000000001010000 _b	15
12_x	0000000010011111 _b	16
13_x	0000000010011110 _b	16
14_x	0000000010011101 _b	16
15_x	0000000010011100 _b	16
16_x	0000000010011011 _b	16
17_x	0000000010011010 _b	16

18 _x	0000000010011001 _b	16
19 _x	0000000010011000 _b	16
1A _x	0000000010010111 _b	16
1B _x	0000000010010110 _b	16
1C _x	0000000010010101 _b	16
1D _x	0000000010010100 _b	16
1E _x	0000000010010011 _b	16
1F _x	0000000010010010 _b	16
21 _x	0000000010010001 _b	16
2F _x	0000000010010000 _b	16
30 _x	0000000010001111 _b	16
31 _x	0000000010001110 _b	16
32 _x	0000000010001101 _b	16
33 _x	0000000010001100 _b	16
34 _x	0000000010001011 _b	16
35 _x	0000000010001010 _b	16
36 _x	0000000010001001 _b	16
37 _x	0000000010001000 _b	16
38 _x	0000000010000111 _b	16
39 _x	0000000010000110 _b	16
3A _x	0000000010000101 _b	16
3B _x	0000000010000100 _b	16
3C _x	0000000010000011 _b	16
3D _x	0000000010000010 _b	16
3E _x	0000000010000001 _b	16
3F _x	0000000010000000 _b	16
41 _x	0000000001111111 _b	16
42 _x	0000000001111110 _b	16
4F _x	0000000001111101 _b	16
50 _x	0000000001111100 _b	16
51 _x	0000000001111011 _b	16
52 _x	0000000001111010 _b	16
53 _x	0000000001111001 _b	16
54 _x	0000000001111000 _b	16
55 _x	0000000001110111 _b	16
56 _x	0000000001110110 _b	16
57 _x	0000000001110101 _b	16
58 _x	0000000001110100 _b	16
59 _x	0000000001110011 _b	16
5A _x	0000000001110010 _b	16

5B _x	0000000001110001 _b	16
5C _x	0000000001110000 _b	16
5D _x	0000000001101111 _b	16
5E _x	0000000001101110 _b	16
5F _x	0000000001101101 _b	16
61 _x	0000000001101100 _b	16
62 _x	0000000001101011 _b	16
6E _x	0000000001101010 _b	16
6F _x	0000000001101001 _b	16
70 _x	0000000001101000 _b	16
71 _x	0000000001100111 _b	16
72 _x	0000000001100110 _b	16
73 _x	0000000001100101 _b	16
74 _x	0000000001100100 _b	16
75 _x	0000000001100011 _b	16
76 _x	0000000001100010 _b	16
77 _x	0000000001100001 _b	16
78 _x	0000000001100000 _b	16
79 _x	0000000001011111 _b	16
7A _x	0000000001011110 _b	16
7B _x	0000000001011101 _b	16
7C _x	0000000001011100 _b	16
7D _x	0000000001011011 _b	16
7E _x	0000000001011010 _b	16
7F _x	0000000001011001 _b	16
81 _x	0000000001011000 _b	16
82 _x	0000000001010111 _b	16
8E _x	0000000001010110 _b	16
8F _x	0000000001010101 _b	16
90 _x	0000000001010100 _b	16
91 _x	0000000001010011 _b	16
92 _x	0000000001010010 _b	16
93 _x	0000000001010001 _b	16
94 _x	0000000001010000 _b	16
95 _x	0000000001001111 _b	16
96 _x	0000000001001110 _b	16
97 _x	0000000001001101 _b	16
98 _x	0000000001001100 _b	16
99 _x	0000000001001011 _b	16
9A _x	0000000001001010 _b	16

9B _x	0000000001001001 _b	16
9C _x	0000000001001000 _b	16
9D _x	0000000001000111 _b	16
9E _x	0000000001000110 _b	16
9F _x	0000000001000101 _b	16
A1 _x	0000000001000100 _b	16
A2 _x	0000000001000011 _b	16
A3 _x	0000000001000010 _b	16
AD _x	0000000001000001 _b	16
AE _x	0000000001000000 _b	16
AF _x	0000000000111111 _b	16
B0 _x	0000000000111110 _b	16
B1 _x	0000000000111101 _b	16
B2 _x	0000000000111100 _b	16
B3 _x	0000000000111011 _b	16
B4 _x	0000000000111010 _b	16
B5 _x	0000000000111001 _b	16
B6 _x	0000000000111000 _b	16
B7 _x	0000000000110111 _b	16
B8 _x	0000000000110110 _b	16
B9 _x	0000000000110101 _b	16
BA _x	0000000000110100 _b	16
BB _x	0000000000110011 _b	16
BC _x	0000000000110010 _b	16
BD _x	0000000000110001 _b	16
BE _x	0000000000110000 _b	16
BF _x	0000000000101111 _b	16
C1 _x	0000000000101110 _b	16
C2 _x	0000000000101101 _b	16
C3 _x	0000000000101100 _b	16
C5 _x	0000000000101011 _b	16
CB _x	0000000000101010 _b	16
CD _x	0000000000101001 _b	16
CE _x	0000000000101000 _b	16
CF _x	0000000000100111 _b	16
D0 _x	0000000000100110 _b	16
D1 _x	0000000000100101 _b	16
D2 _x	0000000000100100 _b	16
D3 _x	0000000000100011 _b	16
D4 _x	0000000000100010 _b	16

D5 _x	0000000000100001 _b	16
D6 _x	0000000000100000 _b	16
D7 _x	0000000000011111 _b	16
D8 _x	0000000000011110 _b	16
D9 _x	0000000000011101 _b	16
DA _x	0000000000011100 _b	16
DB _x	0000000000011011 _b	16
DC _x	0000000000011010 _b	16
DD _x	0000000000011001 _b	16
DE _x	0000000000011000 _b	16
DF _x	0000000000010111 _b	16
E1 _x	0000000000010110 _b	16
E2 _x	0000000000010101 _b	16
E3 _x	0000000000010100 _b	16
EB _x	0000000000010011 _b	16
ED _x	0000000000010010 _b	16
EE _x	0000000000010001 _b	16
EF _x	0000000000010000 _b	16
F0 _x	0000000000001111 _b	16
F1 _x	0000000000001110 _b	16
F2 _x	0000000000001101 _b	16
F3 _x	0000000000001100 _b	16
F4 _x	0000000000001011 _b	16
F5 _x	0000000000001010 _b	16
F6 _x	0000000000001001 _b	16
F7 _x	0000000000001000 _b	16
F8 _x	0000000000000111 _b	16
F9 _x	0000000000000110 _b	16
FA _x	0000000000000101 _b	16
FB _x	0000000000000100 _b	16
FC _x	0000000000000011 _b	16
FD _x	0000000000000010 _b	16
FE _x	0000000000000001 _b	16
FF _x	0000000000000000 _b	16