The Format of the Guild Wars 2 Archive File

Jon Dahm

May 6, 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.

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 tothe 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 thestart 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 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.

2.2.1 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
100_x	1	0
101_x	2	0
102_x	3	0
103_x	4	0
104_x	5	0
105_x	6	0
106_x	7	0
107_x	8	0
108_x	9	1
109_x	11	1
$10A_x$	13	1
$10B_x$	15	1
$10C_x$	17	2
$10D_x$	21	2
$10E_x$	25	2
$10F_x$	29	2

Code	Base	Additional Bits
110_x	33	3
111_x	41	3
112_x	49	3
113_x	57	3
114_x	65	4
115_x	81	4
116_x	97	4
117_x	113	4
118_x	129	5
119_x	161	5
$11A_x$	193	5
$11B_x$	225	5
$11C_x$	256	0

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
0_x	1_x	0
1_x	2_x	0
2_x	3_x	0
3_x	4_x	0
4_x	5_x	1
5_x	7_x	1
6_x	9_x	2
7_x	D_x	2
8_x	11_x	3
9_x	19_x	3
A_x	21_x	4
B_x	31_x	4
C_x	41_x	5
D_x	61_x	5
E_x	81_x	6
F_x	$C1_x$	6
10_x	101_x	7
11_x	181_x	7

Code	Base	Additional Bits
12_x	201_x	8
13_x	301_x	8
14_x	401_x	9
15_x	601_x	9
16_x	801_x	10
17_x	$C01_x$	10
18_x	1001_x	11
19_x	1801_x	11
$1A_x$	2001_x	12
$1B_x$	3001_x	12
$1C_x$	4001_x	13
$1D_x$	6001_x	13
$1E_x$	8001_x	14
$1F_x$	$C001_x$	14
20_x	10001_x	15
21_x	18001_x	15

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 thatthe 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 $first - \texttt{FF}_x + (second - \texttt{100}_x) * \texttt{FF00}_x$. *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)	

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	

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

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	

TODO: Research what all these fields do.

5.16.2 Unknown Chunk (DX9S)

This chunk has 3 fields and 5 different structures.

Table 5.7: DX9S Chunk Format

Type	Name	Description
Vtr<Sampler>	Samplers	A list of all samplers for this Material.See Table 5.8.
Vtr<Shader>	Shaders	A list of all shaders for this Material.See Table 5.9.
Vtr<Technique>	Techniques	See Table 5.10.

Table 5.8: Sampler Format

Type	Name	Description
uint32	Texture Index	
Vtr<uint32>	State	

Table 5.9: Shader Format

Type	Name	Description
Vtr<uint32>	Shader	
Vtr<uint32>	Constant Registers	
Vtr<uint32>	Constant Tokens	
uint16	Number of Instructions	

Table 5.10: Technique Format

Type	Name	Description
String	Name	
Vtr<RefList<Effect>>	Passes	See Table 5.11.
uint16	Max PS Version	
uint16	Max VS Version	

Table 5.11: Effect Format

Type	Name	Description
uint64	Token	
Vtr<uint32>	Render States	
Vtr<uint32>	Sampler Index	
uint32	Pixel Shader	
uint32	Vertex Shader	
Vtr<uint32>	Texture Gen	
Vtr<uint32>	Texture Transform	
uint32	VS Gen Flags	
uint32	Pass Flags	

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

This Pack File has only one chunk, whose FourCC is `['A','M','S','P']`. However, there is a lot of data in that single chunk, with 22 different structures included in it. See Table 5.12 for the chunk's format.

Table 5.12: the AMSP chunk format.

Type	Name	Description
uint64	Music Cue	
uint64	Reverb Override	
uint64	Snapshot	
Ptr<AudioSettings>	Audio Settings	(see Table 5.13)
Vtr<Handler>	Handlers	(see Table 5.30)
Vtr<MetaSoundData>	Meta Sound Data	(see Table 5.31)
Vtr<ScriptRef>	Script References	(see Table 5.32)
Vtr<TriggerKey>	Trigger Keys	(see Table 5.33)
uint32	Flags	
uint32	Sound Pool Count	
float32	Fade-in Time	
float32	Sound Pool Delay	
float32	Volume	
byte	Music Cue Priority	
byte	Music Mute Priority	

Table 5.13: the AudioSettings structure

Type	Name	Description
uint64	Default Snapshot	
uint64	Effects Bus	
float32	Distance Scale	
float32	Doppler Scale	
float32	Focus Transition	
Vtr<BusData>	Busses	(see Table 5.14)
Vtr<Category>	Categories	(see Table 5.17)
Vtr<MusicCondition>	Music Conditions	(see Table 5.25)
Vtr<Playlist>	Playlists	(see Table 5.26)
Vtr<Reverb>	Reverbs	(see Table 5.28)
Vtr<Snapshot>	Snapshots	(see Table 5.29)
Ptr<FileRef>	Bank Index File	
Ptr<FileRef>	Bank Script File	
Ptr<FileRef>	Music Script File	

Table 5.14: the BusData structure

Type	Name	Description
uint64	ID	
uint32	Flags	
uint64	Output	
Ptr<BusDynamicData>	Dynamic Data	(see Table 5.15)

Table 5.15: the BusDynamicData structure

Type	Name	Description
uint64	ID	
uint32	Flags	
float32	Volume	
Vtr<DSPData>	DSP Data	(see Table 5.16)

Table 5.16: the DSPData structure

Type	Name	Description
uint32	Type	
uint32	Flags	
Vtr<float32>	Properties	

Table 5.17: the Category structure

Type	Name	Description
uint64	ID	
uint64	Volume Group ID	
uint64	Output Bus ID	
Ptr<Attenuation>	Attenuation	(see Table 5.18)
Ptr<CategoryDynamicData>	Dynamic Data	(see Table 5.24)
float32	Mute Fade Time	
uint32	Flags	
uint32	Max Audible	
byte	Max Audible Behavior	

Table 5.18: the Attenuation structure

Type	Name	Description
float32	Doppler	
DynamicParamData	Low Pass	(see Table 5.19)
DynamicParamData	High Pass	
DynamicParamData	Pan 3D	
DynamicParamData	Reverb	
DynamicParamData	Spread 3D	
DynamicParamData	Volume A	
DynamicParamData	Volume B	

Table 5.19: the DynamicParamData structure

Type	Name	Description
Ptr<Envelope>	Envelope	(see Table 5.20)
Ptr<RandomParam>	Random Param	(see Table 5.22)
float32	Value	
byte	Type	

Table 5.20: the Envelope structure

Type	Name	Description
uint64	Offset Parameter	
Vtr<EnvelopePoint>	Envelope Points	(see Table 5.21)
byte	Offset Type	

Table 5.21: the EnvelopePoint structure

Type	Name	Description
float32	Offset	
float32	Value	

Table 5.22: the RandomParam structure

Type	Name	Description
RangeData	Time	(see Table 5.23)
RangeData	Value	

Table 5.23: the RangeData structure

Type	Name	Description
float32	Max	
float32	Min	
byte	Min (byte)	

Table 5.24: the CategoryDynamicData structure

Type	Name	Description
uint64	ID	
float32	Volume	
float32	Non-Focus Gain	
float32	Low Pass	
float32	High Pass	
float32	Reverb Direction	
float32	Reverb Room	
uint32	Flags	

Table 5.25: the MusicCondition structure

Type	Name	Description
uint64	ID	
uint32	Flag	
Vtr<byte>	Byte Code	

Table 5.26: the Playlist structure

Type	Name	Description
uint64	Category	
uint64	ID	
uint64	Primary Playlist ID	
uint64	Secondary Playlist ID	
Vtr<FileNameData>	Files	(see Table 5.27)
float32	Fade-In Time	
float32	Fade-Out Time	
uint32	Flags	
RangeData	Initial Silence	
RangeData	Interval Silence	
RangeData	Max Play Length	
DynamicParamData	Volume	
byte	File Iterate Mode	

Table 5.27: the FileNameData structure

Type	Name	Description
uint64	Condition	
uint64	Language	
float32	Volume	
float32	Weight	
Ptr<FileRef>	File	
byte	Audio Type	
byte	Note Base	
byte	Note Min	
byte	Note Max	

Table 5.28: the Reverb structure

Type	Name	Description
uint64	ID	
uint32	Flags	
float32	Room	
float32	Room HF	
float32	Room LF	
float32	Decay Time	
float32	Decay HF Ratio	
float32	Reflections	
float32	Reflections Delay	
float32	Reverb	
float32	Reverb Delay	
float32	Reference HF	
float32	Reference LF	
float32	Diffusion	
float32	Density	
float32	Echo Delay	
float32	Echo Decay Ration	
float32	Echo Wet Mix	
float32	Echo Dry Mix	

Table 5.29: the Snapshot structure

Type	Name	Description
uint64	ID	
float32	Blend-In Time	
float32	Blend-Out Time	
uint32	Flags	
Vtr<BusDynamicData>	Busses	(see Table 5.15)
Vtr<CategoryDynamicData>	Categories	(see Table 5.24)
byte	Priority	

Table 5.30: the Handler structure

Type	Name	Description
uint64	ID	
uint32	Flags	
Vtr<byte>	Byte Code	

Table 5.31: the MetaSoundData structure

Type	Name	Description
uint64	Category	
uint64	End Cue	
uint64	ID	
uint64	Offset Bone	
uint64	Playlist ID	
Vtr<DSPData>	DSP	(see Table 5.16)
Ptr<Attenuation>	Attenuation	(see Table 5.18)
Vtr<FileNameData>	Files	(see Table 5.27)
float32	Channel Fade-In	
float32	Channel Fade-Out	
float32	End Cue Offset	
float32	Fade-In Time	
float32	Fade-Out Time	
float32[3]	Position Offset	
uint32	Channel Max	
uint32	Flags	
uint32	Loop Count	
DynamicParamData	Depth	(see Table 5.19)
DynamicParamData	Pan	
DynamicParamData	Pitch	
DynamicParamData	Pitch MS	
DynamicParamData	Volume	
DynamicParamData	Volume MS	
RangeData	Initial Delay	(see Table 5.23)
RangeData	Play Length	
RangeData	Position Offset Angle	
RangeData	Position Range	
RangeData	Repeat Count	
RangeData	Repeat Time	
RangeData	Start Time Offset	
byte	Channel Mode	
byte	Channel Priority	
byte	File Iterate Mode	
byte	Loop Mode	
byte	Music Priority	
byte	Playback Mode	
byte	Position Mode	

Table 5.32: the ScriptRef structure

Type	Name	Description
uint64	ID	
Ptr<FileRef>	File	

Table 5.33: the TriggerKey structure

Type	Name	Description
uint64	ID	
Vtr<TriggerMarker>	Markers	(see Table 5.34)

Table 5.34: the TriggerMarker structure

Type	Name	Description
uint64	Cue	
uint64	End	
float32	Time	
byte	Type	

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

Value	Huffman Code	Number of Bits
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

Value	Huffman Code	Number of Bits
$8A_x$	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
$E7_x$	000000001111_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

Value	Huffman Code	Number of Bits
$0F_x$	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

Value	Huffman Code	Number of Bits
$3E_x$	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

Value	Huffman Code	Number of Bits
$7E_x$	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

Value	Huffman Code	Number of Bits
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

Value	Huffman Code	Number of Bits
$\texttt{F3}_x$	0000000000001100_b	16
$\texttt{F4}_x$	0000000000001011_b	16
$\texttt{F5}_x$	0000000000001010_b	16
$\texttt{F6}_x$	0000000000001001_b	16
$\texttt{F7}_x$	0000000000001000_b	16
$\texttt{F8}_x$	0000000000000111_b	16
$\texttt{F9}_x$	0000000000000110_b	16
$\texttt{FA}_x$	0000000000000101_b	16
$\texttt{FB}_x$	0000000000000100_b	16
$\texttt{FC}_x$	0000000000000011_b	16
$\texttt{FD}_x$	0000000000000010_b	16
$\texttt{FE}_x$	0000000000000001_b	16
$\texttt{FF}_x$	0000000000000000_b	16