The Definitive Guide to Exploring File Formats

Mr. Mouse and WATTO (2004)

November 28, 2024

Introduction

- big file archives reduce file count and allow for streaming of data
- every developer creates their own archive format and even change formats between games within the same company
- *GRAs* - Game Resource Archives
  - An archive is a file that stores many smaller individual data files within it
  - A GRA is an archive specific to a game that contains resources the game uses
- *GRAFs* - Game Resource Archive Formats
  - GRAF describes the way the archive is constructed and how and where the files are found within the archive
  - Formats are usually defined by the requirements of the game engine but will be structured to allow for quick access
    - For example, resources for one specific level can be grouped together
  - Archives must store any type of file in the same "universal" manner
  - Archives must tell the engine what files are contained within the archive and must allow for a consistent accessing method

Tools

- Hex Workshop is highly recommended (but only available on Windows)
  - Allows for color mapping, bookmarks, and GoTo functionality
- *GRAIS* - GRA Id String or "magic"
- Some GRAFs' offsets may not count the 4 bytes for the archive magic (relative offsets). Others might (absolute offsets).

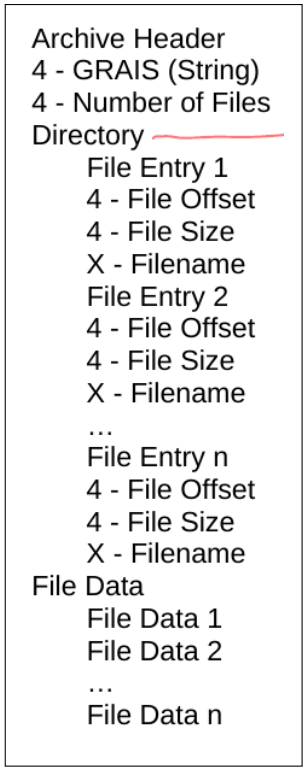
Terms, Definitions, and Data Structures

- Files are a series of bytes stored one after another that can be interpreted to mean something and are stored on disk
- Bits - 0 / 1
- Bytes - 0 - 255 — 0x00 - 0xFF
- 16 bit numbers - words / short - 0 - 65535
- 32 bit numbers - dword / long - 0 - 4.2 billion
- 64 bit numbers - qwords / long / doubles - commonly used for floating point values
- Strings - typically a group of 8 bit characters that are readable in English
  - Be careful as some strings may be in foreign languages or obscure unicode formats and may seem unintelligible but are still strings
- Hex numbering - 4 bits - $2^4 = 16$ values - 0-F
- GRAs rarely used signed numbers. They use offsets greater than zero and file sizes greater than zero. So it makes no sense to have negative values represented.
  - If numbers are negative, the field typically uses the highest bit to signify it is negative
    - Note: When numbers are represented as such there will be a -0 and a +0
- Big-Endian - the big bit is at the end (reading left ot right)
  - Little-Endian - the little bit is at the end (normal)
  - Assume Little-Endian values by default
- Offsets start at 0

Archive Patterns

Directory Archives

- Most common archive in use
- A Directory is present that contains a property list of all files in the archive including file name, file size, and file offset information
  - The presence of this directory / rolodex / table describing the files is the key characteristic of this archive type
- A separate File Data section contains the file data
- If the directory is not stored at the head, there will be an offset (usually 4 bytes) to where it is stored
- If it's not at the head, it will be at the tail.
- The layout is as follows:



Tree Archives

- Tree directories have 3 sections, Directory Entries, File Entries, and File Data sections
- The Directory Entries section will have a series of entries pointing to other directory entries. These offsets can either be another directory or the start of the file list for that directory.
- It's not quite clear how this would work with a directory that contains other directories as well as files.
- This method allows for complex directory trees to be stored in a small space
  - This leads to slightly smaller archives but increases read time for the archive
- This type of archive is rarely found in the wild.
- An example layout is:

\data\sounds\snd1.wav
\data\sounds\snd2.wav
\data\images\temp\pic1.bmp

The following graphic shows the structure of the archive that contains these 3 files.

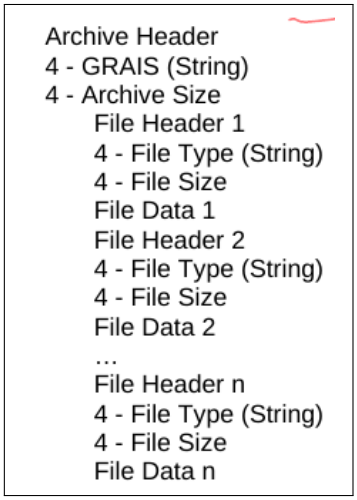
Archive Header
4 - GRAIS (String) ***HEAD***
4 - Number of Directories at Root ***1*** } data about root
4 - Number of Files ***3***
Directory Entries
    Directory Entry 1
    X - Filename ***data***
    4 - Subdirectory Offset ***offset to Directory Entry 2***
    4 - Number of Files in Directory ***0***
    4 - Number of Subdirectories in Directory ***2***
    Directory Entry 2
    X - Filename ***sounds***
    4 - Subdirectory Offset ***offset to File Entry 1***
    4 - Number of Files in Directory ***2***
    4 - Number of Subdirectories in Directory ***0***
    Directory Entry 3
    X - Filename ***images***
    4 - Subdirectory Offset ***offset to Directory Entry 4***
    4 - Number of Files in Directory ***0***
    4 - Number of Subdirectories in Directory ***1***
    Directory Entry 4
    X - Filename ***temp***
    4 - Subdirectory Offset ***offset to File Entry 3***
    4 - Number of Files in Directory ***1***
    4 - Number of Subdirectories in Directory ***0***
File Entries
    File Entry 1
    4 - File Offset ***offset to File Data 1***
    4 - File Size ***size of File Data 1***
    X - Filename ***snd1.wav***
    File Entry 2
    4 - File Offset ***offset to File Data 2***
    4 - File Size ***size of File Data 2***
    X - Filename ***snd2.wav***
    File Entry 3
    4 - File Offset ***offset to File Data 3***
    4 - File Size ***size of File Data 3***
    X - Filename ***pic1.bmp***
File Data

    File Data 1
    File Data 2
    File Data 3

} file entry 2 is read because it parent dir lists two files and these are stored sequentially

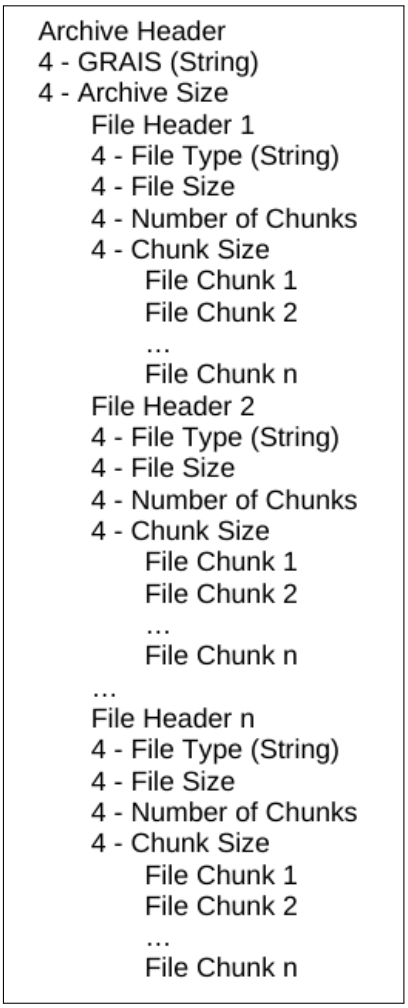
Chunked Archives

- Simplest type of archive
- Files are stored sequentially with all data in a file header
- Typically read by reading all the metadata first and then going back to read the file data
- Based on EA IFF85 standard
  - Used for many files including WAV and AVI



Split Chunk Archives

- Very similar to chunked archives except files are segmented into chunks that are the same size to allow for efficient buffer while reading the file



External Directory Archives

- Very similar to directory archive except file data section is stored in a different file
- Files will have the same name but a different extensions (.dir / .arc)

Checking Your Results

Common Types of Fields

- These fields are very common:
  - File size, File Offset, Number of Files, Magic (GRAIS)
- These fields occur in some archives but have a chance of not appearing:
  - First File Offset, Archive Name, Filename Offset, Filename Directory Offset, Total File Data Size, Total Directory Size, Archive Size, Number of Directories, Directory Offset, File Extension / Type, File ID, Archive Version, Filename Length, Decompressed File Size, Checksum, Timestamp

Validating Your Fields

- Always check to see if the field is what you think it is
- Make sure to try with multiple instances of that field (or through multiple archives)
- Ex: One way to validate file data offset is to go to that offset and see if you recognize the magic at that location. Some common headers:
  - RIFF - wav
  - BM - bitmap
  - GIF - gif
  - JFIF - jpg
- If you think your files are compressed and you've found the file size field, try looking for a decompressed file size field for each entry
  - Look for a field that's always slighltly larger than the file size field.
- Another tip is if your archive contains a directory, try and find a repeatable pattern and get the number of file entries in that directory. This number will usually be somewhere at the start of the archive. Also you can try to see if it works for multiple archives following the same GRAF.

Padding

- It is common to encounter padding in an archive. This padding can occur at the end of file data or file entries within a directory.
- Typically padding is added to get to multiples of 4 bytes. This is done for buffering reasons.
- Some archives pad their file data to multiples of 2048 bytes.

File name Patterns

- File names tell the reader how to open and edit the files
- Many archives don't store file names as they take up a lot of space
- Some filenames are stored in a separate directory
  - Typically, the file entries will have an offset to the file name
- File names are usually stored in the same order the files appear in the archive
- Some archives will have a file name length field just before the file name
  - These sizes will usually include the null terminating byte as part of the length
- Some archives will use the space character (0x20) instead of the null byte

Encryption and Compression

The Basics

- Encryption to protect resources
- Compression to reduce archive size
- Use a disassembler to find decryptiong and decompression algorithms used by the actual engine
- Encryption relies on bitwise operations:
  - AND, OR, XOR, NOT, SHL (shift-left), SHR
  - These can be used for fast division / multiplication, color inverting, and switching of bits on and off
- An example of the usefulness of bitwise operations is in graphics cards. Assume you have a resolution of 640x480 pixels but only have screen memory of 4x 320x200 bytes. We cannot encode all the pixels in the screen memory (we would miss the last 80 pixel columns). What we can do is reduce the color range of each pixel (to 0-15 swatch based coloring) and store two pixels per byte. Odd pixels (1-indexed) would be stored in the lower nibble and even pixels in the higher nibble. To do this we would need to SHL the values of the even pixels and OR them with the values of the odd pixels.
- XOR is very common in encryption as it's reversable: a XOR b XOR b = a

Encryption

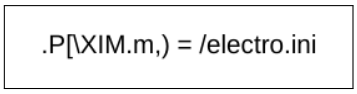
- Encryption techniques, though numerous, are purely logical in nature
- First you must be certain that the file you are looking at is encrypted
- Get more references. Find more files / archives that are encrypted and compare them to see if you can find patterns
- Use a disassembler to find resource name strings. These can be the decrypted and expected file name and can give you a target to reach while working out the decryption algorithm.

- Example: Painkiller Encryption
  - scripts.pak contained file name strings such as:

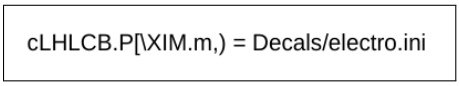
```
cLHLCB.P[\XIM.m,)
lOOO\A.[WVSJ/4j/&#_
iJRRYD.IR\*&1/&'(a8=<
...

Or, in hexadecimals:
63 4C 48 4C 43 42 1C 50 5B 5C 58 49 4D 2E 6D 2C 29 20
6C 4F 4F 4F 5C 41 1B 5B 57 56 53 4A 2F 34 6A 2F 26 23
69 4A 52 52 59 44 16 49 52 5C 2A 26 31 2F 26 27 28 61 38 3D 3C
```

  - We know these are strings that are file names. This is a good starting piont
  - This archive also had a 4 byte entry before each file name listing the length of the string. We can therefore assume the characters are encrypted in place and their positions match those in the original string.
  - We also know that file names usually have the structure `directory\directory\filename.extension`
  - It's also safe to assume the extension is 3 bytes in length
  - Looking at the string we see that the first character is in the 0x60 range and the rest are in the 0x40 range followed by a character in the 0x10 range. Since we know that alphabetic characters are stored together we can assume the 0x10 is a directory delimiter (0x5C '\' or 0x2F '/'). Also, it's likely the directory starts with a capital letter.
  - Additionally, we see that the 4th last character is in the 0x60 range and is likely a dot (0x2E)

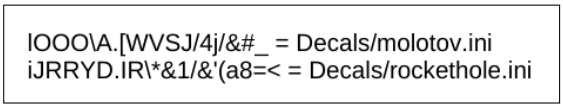
- Let's try using XOR. We know what some original characters might be so let's try finding the seed for each specific XOR operation. Using the first entry
  63 4C 48 4C 43 42 1C 50 5B 5C 58 49 4D 2E 6D 2C 29 20
  - backslash (0x5C): 0x5C XOR 0x1C = 0x40
  - dot (0x2E): 0x2E XOR 0x6D = 0x43
  - forward slash (0x2F): 0x2F XOR 0x1C = 0x33
- Now let's look at the differences for the seeds. We know that the dot is 8 characters away from the directory delimiter. Looking at the differences:
  - dot - backslash = 0x43 - 0x40 = 0x03
  - dot - forwardslash = 0x43 - 0x33 = 0x10
- 0x10 is 16 which is 2x the number of characters between the delimiter and the dot. Let's try adding 2 to the seed for every subsequent XOR we do. Starting from the forwards slash we use 0x33 as our XOR seed and then add 2 to it each time:



- Now we can go backwards from the slash and do the rest of the string (subracting 2 for each seed):



- The seed used for the first letter of the string is 0x27 (39)
- Repeating the above process for the other two strings we get:



- The second string is seeded with 0x28 (40) and the third with 0x2D (45)

- Let's also compare the size of the strings. In order the sizes are: 0x12 (18), 0x12 (18), 0x15 (21)
- If we assume the encryption method uses the string size to encrypt, we can observe that while the first and second strings are the same length, their seed has a difference of 1.
- Perhaps the seed calculation takes into account the index of the file. This would account for the difference between the first and second strings. But it doesn't work for the third.
- We can also observe that doing a shift left on the string length gets us close to the seed value:
  - 0x12 (18) $<$ 1 = 0x24 (36)
  - 0x12 (18) $<$ 1 = 0x24 (36)
  - 0x15 (21) $<$ 1 = 0x2A (42)
- At this point we need more information. If we apply the above to more strings in the archive we see that:
  shift left(size) - seed value = {-3, -2, -1, 0, 1}
- This range is not very symmetric. If we consider a different shift left (inclusive shift left), we could make the range symmetric. Inclusive shift left sets the rightmost bit after the shift making the result be (size * 2 + 1).
- But how does it know what values to use when decrypting? We need more data. Make a table! Doing so we will find that the length gives an index into the repeating set {-2, -1, 0, 1, 2}. A size of 0 would return -2. A size of 3 would return 0. A size of 5 would return -2 again. And so on.
- The final encryption for the strings would be:
  - (shift left (0x12) + 1) + 1 (from file index) + 1 (code in set using size as an index into your table) = 0x27
  - (shift left (0x12) + 1) + 2 (from file index) + 1 (code in set using size as an index into your table) = 0x28
  - (shift left (0x15) + 1) + 3 (from file index) + -1 (code in set using size as an index into your table) = 0x2D

- When trying to find the decryption algorithm there is a lot of guessing and second-guessing. Keep trying things and work until you solve it!

Compression

- Many different compression algorithms
- ZLib is common as it's open source and there are no licensing deals needed
  - Identified by the lowercase 'x' as the first byte
- Some decompression algorithms require the decompressed file size to work
- PKZip is another common compression algorithm
  - Some archives reimplement the algorithm and put them into their own file format
- You either need a good strategy or an in depth investigation
- You can always disassemble the game and reverse engineer the decompression algorithm
  - Along the same vein, you can also investigate method names or libraries that are included. If you see something called LZHUncompress(), or if you find an error string saying "RAR: Error in CRC", you can pinpoint the algorithm that way.
- These days, the best way is probably to brute force a bunch of compression algorithms using the QuickBMS brute force script and then finding files that make sense after decompression.

ASCII

1. Standard

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

2. Extended

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	∑
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	ø
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	